ТЕМА. *Історія створення та платформа JAVA.*

План

1. *Історія виникнення і розвиток JAVA.*
2. *Відмінності Java і Java Script.*
3. *Платформа JAVA.*

Рекомендована література: 1,2 3.

Теоретичні відомості

Java широко відома як новітня об'єктно-орієнтована мова, що легка у вивченні і дозволяє створювати програми, які можуть виконуватися на будь-якій платформі без яких-небудь доопрацювань Ще з Java чомусь завжди пов'язана тема кави (зображення логотипів, назви продуктів і так далі). Програмісти можуть додати до цього опису, що мова схожа на спрощений C або C++ з додаванням garbage collector'a- автоматичного збирача "сміття" (механізм звільнення пам'яті, яка більше не використовується програмою). Також відомо, що Java орієнтована на Internet, і найпоширеніше її застосування - невеликі програми, аплети, які запускаються в браузері і є частиною HTML-сторінок.З часу офіційного оголошення Java пройшло достатньо багато часу для "просто модної технології". Конференція розробників Java One, яка вперше була організована в 1996 році, вже через рік зібрала більше 10000 учасників і стала найбільшою конференцією із створення програмного забезпечення в світі (кожен наступний рік число учасників росло приблизно на 5000). Спеціальна програма Sun, об'єднуюча розробників Java по всьому світу, Javadeveloper Connection, також була запущена в 1996 році, через рік вона налічувала більше 100.000 розробників, а в 2000 році - більше 1,5 мільйонів. На сьогодні число програмістів на Java налічує 3 мільйони.Було випущено п'ять основних версій мови, починаючи з 1.0 в 1995 році і закінчуючи 1.4 в лютому 2002 року. Наступна версія 1.5 випущена в 2004 році. Всі версії і документацію до них завжди можна було безкоштовно отримати на офіційному web-сайті Java <http://java.sun.com/>. Один з перших продуктів для Java - JDK 1.1 (засіб розробки на Java) - протягом перших трьох тижнів після оголошення був завантажений більше 220.000 разів. Версія 1.4 була завантажена більше 2 мільйонів разів за перших 5 місяців. Практично всі провідні виробники програмного забезпечення ліцензіювали технологію Java і регулярно оголошують про вихід побудованих на ній продуктів. Це і "блакитний гігант" IBM, і творець платформи Macintosh фірма Apple, і лідер в області реляційних БД Oracle, і навіть головний конкурент фірми Sun - корпорація Microsoft - ліцензіювала Java ще в березні 1996 року.

Історія створення Java. Якщо пошукати в Interneti історію створення Java, з'ясовується, що спочатку мова називалася OaK ("дуб"), а робота з його створення почалася ще в 1990 році з досить скандальної історії усередині корпорації Sun. Спочатку Гослінг спробував модифікувати C++, щоб створити мову для написання програм, мінімально орієнтованих на конкретні платформи. Проте дуже скоро стало зрозуміло, що це практично неможливо. Основна перевага C++ - швидкість програм, але зовсім не їх надійність. А надійність роботи для звичайних користувачів повинна бути так само абсолютно гарантована, як сумісність звичайних електричної вилки і розетки. Тому в

червні 1991 року Гослінг, який написав свою першу мову програмування в 14 років, починає розробку заміни C++. Створюючи новий каталог і роздумуючи, як його назвати, він виглянув у вікно, і погляд його зупинився на зростаючому під ним дереві. Так мова отримала свою першу назву - OAK (дуб). Декілька років потому, після проведення маркетингових досліджень, ім'я змінили на Java.Відродження OAKДля переможного виходу OAK не хватало останнього штриха - браузера, який підтримував би цю технологію. Саме він повинен був стати тим самим "шокуючим" додатком Нотона, який завершував майже п'ятирічну підготовчу роботу перед офіційним оголошенням нової платформи.Браузер назвали Webrunner. Нотону було потрібно всього один вихідний, щоб написати основну частину програми. Це було в липні, а у вересні 1994 року Webrunner вже демонструвався керівництву Sun. Невеликі програми, написані на OAK для розповсюдження через Internet, назвали аплетами (applets).Наступна демонстрація відбувалася на конференції, де зустрічалися розробники Internet-додатків і представники індустрії розваг. Коли Гослінг почав презентацію Webrunner, слухачі не виявили великої цікавості, вирішивши, що це просто клон Mosaic. Тоді Гослінг провів мишкою над складною тривимірною моделлю хімічної молекули.Слідуючи за курсором, модель поверталася по всіх напрямах! Зараз дана функція, можливо, не справляє такого враження, проте у той час це було подібно до переходу від картинки до кінематографа. Наступний приклад демонстрував анімоване сортування. Спочатку зображався набір відрізків різної довжини. Потім синя і червона лінії починали бігати по цьому набору, сортуючи відрізки за розміром. Приклад теж нехитрий, такий, що проте наочно демонструє, що на стороні клієнта з'явилася повноцінна програмна платформа. Обидва ці аплети зараз є стандартними прикладами і входять до складу Java Development Kitлюбой версії. Згодом, OAK був перейменований в Java, а Webrunner почав називатися Hotjava. Багато хто тоді дивувався, що ж послужило приводом для такого рішення. Легенда свідчить, що Java - це сорт кави (така кава дійсно є), яку дуже любили програмісти. Мабуть, схожим чином народилася і назва Hotjava ("гаряча Java"). Тема кави назавжди залишиться в назвах і логотипах (технологія створення компонентів названа Java Beans - зерна кави, спеціальний формат для архівації файлів з Java-программами JAR - банка з кавою і так далі), а саму мову критики почали називати "для кавоварок".

Синтаксис *Java* 23 березня 1995 року в газеті Sun Jose Mercury News в статті з описом нової технології була приведена адреса офіційного сайту <Http://java.sun.com/>, який і по цей день є основним джерелом інформації по Java.

Java виходить в світ 23 травня 1995 року технології Java і Hotjava були офіційно оголошені Sun і тоді ж представники компанії повідомили, що нова версія найпопулярнішого браузера Netscape Navigator 2.0 підтримуватиме нову технологію. По суті, це означало, що відтепер Java стає такою ж невід'ємною частиною WWW, як і HTML. Вже другий раз презентація закінчилася овеціями - переможний хід Java почався.

Історія розвитку Java. Браузери. За допомогою Java web-сторінку можна наповнити не тільки звичайним текстом, але і динамічними елементами - простими відеовставками типу земної кулі, що обертається, або інтерактивними елементами типу моделі хімічної молекули, що обертається; рядками, що біжать, містять, наприклад, біржові індекси або прогноз погоди.

Але насправді Java - це більше, ніж прикраса HTML. Оскільки це повноцінна мова програмування, з її допомогою можна створити складний призначений для користувача інтерфейс. У найпершій версії Java Development Kit(засіб розробки на Java) був приклад аплета, що представляє прості електронні таблиці. Незабаром з'явився текстовий редактор, що дозволяє міняти стиль і колір тексту. Звичайно, були ігрові аплети, повчальні,

моделюючі, фізичні і інші системи. Наприклад, клієнт, що зробив замовлення в магазині або відправив посилку поштою, діставав можливість стежити за доставкою через Internet.

Проте сьогодні, як відомо, найпоширеніший браузер - Microsoft Internet Exploreri його

використовують більше 95% користувачів.

Варто зупинитися на мові Java Script, яка також вельми поширена і яку до цих пір багато хто пов'язує з Java, мабуть, унаслідок схожості імен. Втім, деякі загальні риси у них дійсно є.

4 грудня 1995 року компанії Netscape і Sun спільно оголошують нову "мову сценаріїв" (scripting language) Java Script. Код Java Script описується прямо в HTML-тексті (хоча можна і підвантажувати його з окремих файлів з розширенням js. Ця мова призначена для створення додатків, які зв'язують об'єкти і ресурси на клієнтській машині або на сервері. Таким чином, Java Script, з одного боку, розширює і доповнює HTML, а з іншого боку - доповнює Java. За допомогою Java пишуться об'єкти-аплети, якими можна управляти через мову сценаріїв.

Загальні властивості Java Script і Java:

- легкість в освоєнні. По цьому параметру Java Script порівнюють з Visual Basic - щоб використовувати ці мови, досвід програмування не потрібний;
- кроссплатформовість. Код Java Script виконується браузером. Мається на увазі, що браузери на різних платформах повинні забезпечувати однакову функціональність для сторінок, що використовують мову сценаріїв. Проте це виконується приблизно в тому ж ступені, що і підтримка самого HTML, - відмінностей все ж таки дуже багато;
- відвертість; специфікація мови відкрита для використання і обговорення співтовариством розробників;
- всі перераховані властивості дозволяють стверджувати, що Java Script добре пристосований для Internet-програмування;
- синтаксиси мов Java Script і Java дуже схожі.
- мова Java Script не об'єктно-орієнтована (хоча деякі аспекти об'єктно-орієнтованого підходу підтримуються), але дозволяє використовувати різні об'єкти, що надаються браузером;
- схожа історія появи і розвитку. Обидві мови були оголошені компаніями Sun і Netscape з інтервалом в декілька місяців.
- Не дивлячись на велику кількість схожих характеристик, Java і Java Script - абсолютно різні мови, і насамперед - за призначеннями. Якщо спочатку Java позиціонувався як мова для створення Internet-додатків (аплетів), то зараз вже очевидно, що Java - це повноцінна мова програмування. Що стосується java Script, то він повністю виправдовує свою назву мови сценаріїв, залишаючись розширенням HTML.

Мережеві комп'ютери. Коли стало зрозуміло, що нова технологія має небувалий попит, розробникам захотілося укріпити і розвинути успіх і поширеність Java. за допомогою Java-апплетів управляють роботом, що вивчає поверхню Марса ("Java допомагає робити історію!").

Зупинимося докладніше на тому, чому по відношенню до Java використовується термін "**платформа**", чим Java відрізняється від звичайної мови програмування. Як правило, платформою називають поєднання апаратної архітектури ("залізо"), яка визначається типом використовуваного процесора (Intel x86, Sun SPARC, POWERPC і ін.), з операційною системою (MS Windows, Sun Solaris, Linux, Mac OS і ін.). При написанні програм розробник завжди користується засобами цільової платформи для доступу до мережі, підтримка потоків виконання, роботи з графічним користувацьким інтерфейсом (GUI) і іншими можливостями. Звичайно, різні платформи, через технічних, історичних і інших причин, підтримують різні інтерфейси (API, Application Programming Interface), а значить, і програма може виконуватися тільки під тією платформою, під яку вона була написана.А значить, і мови програмування повинні бути спочатку орієнтовані на якусь конкретну платформу.

Синтаксис і основні концепції легко розповсюдити на будь-яку систему (хоча це і не завжди ефективно), але бібліотеки, компілятор і, природно, бінарний виконуваний код специфічні для кожної платформи. Так було з самого початку епохи комп'ютерних обчислень, а тому лише небагато, дійсно вдалих програм підтримувалися відразу на декількох системах, що приводило до деякої ізоляції світів програмного забезпечення для різних операційних систем.

Було б дивно, якби з розвитком комп'ютерної індустрії розробники не спробували створити універсальну платформу, під якою могли працювати всі програми. Особливо такому кроку сприяв бурхливий розвиток Глобальної мережі Internet, яка об'єднала користувачів незалежно від типу використовуваних процесорів і операційних систем. Саме тому творці Java задумали розробити не просто ще одну мову програмування, а універсальну платформу для виконання додатків, тим більше що спочатку Oak створювався для різних побутових приладів, від яких чекати сумісності не доводиться.Яким же чином можна "згладити" відмінності і різноманіття операційних систем? Спосіб не новий, але ефективний - за допомогою віртуальної машини. Додатки на мові Java виконуються в спеціальному, універсальному середовищі, яке називається Java Virtual Machine. JVM - це програма, яка пишеться спеціально для кожної реальної платформи, щоб, з одного боку, приховати всі її особливості, а з іншої - надати єдине середовище виконання для Java - додатків. Фірма Sun і її партнери створили JVM практично для всіх сучасних операційних систем. Коли мова йде про браузер з підтримкою Java, мається на увазі, що в нім є вбудована віртуальна машина.Докладніше JVM розглядається нижчим, але необхідно сказати, що розробники Sun доклали зусилля, щоб зробити цю машину цілком реальною, а не тільки віртуальною. 29 травня 1996 року оголошується операційна система Java OS (фінальна версія випущена в березні наступного року). Згідно прес-релізу, це була "можливо, сама невелика і швидка операційна система, підтримуюча Java". Дійсно, розробники прагнули до того, щоб забезпечити можливість виконувати Java-додатки на найширшому спектрі пристроїв - мережеві комп'ютери, кишенькові комп'ютери (PDA), принтери, ігрові приставки, мобільні телефони і так далі Очікувалося, що Java OS буде реалізована на всіх апаратних платформах. Це було необхідно для початкової мети творців Java - легкість додавання новій функціональності і сумісності в будь-які електричні прилади, якими користується сучасний споживач. Це був перший крок, що просуває платформу Java на один рівень вниз, - на рівень операційних систем. Передбачалося зробити і наступний крок - створити апаратну архітектуру, центральний процесор, який би безпосередньо виконував інструкції Java без всякої віртуальної машини. Пристрій з такою реалізацією став би повноцінним Java-пристроєм.

Властивості Java.

1.Кроссплатформенність. Вже говорилося, що вона досягається за рахунок використання віртуальної машини JVM, яка є звичайною програмою, виконуваною операційною системою і що надає Java-додаткам всі необхідні можливості. Оскільки всі параметри JVM специфіковані, то залишається єдине завдання - реалізувати віртуальні машини на всіх існуючих і використовуваних платформах.Початковий код будь-якої програми на мові Java представляється звичайними текстовими файлами, які можуть бути створені в будь-якому текстовому редакторі або спеціалізованому засобі розробки і мають розширення .java. Ці файли подаються на вхід Java-компілятору, який транслює їх в спеціальний байт-код Java. Саме цей компактний і ефективний набір інструкцій підтримується JVM і є невід'ємною частиною платформи Java.Результат роботи компілятора зберігається в бінарних файлах з розширенням .class. Java-додаток, що складається з таких файлів, подається на вхід віртуальній машині, яка починає їх виконувати, або інтерпретувати, оскільки сама є програмою.

2. Об'єктна орієнтованість Java, що в Java практично все реалізовано у вигляді об'єктів - потоки виконання (threads) і потоки даних (streams), робота з мережею, робота із зображеннями, з призначеним для користувача інтерфейсом, обробка помилок і так далі. Будь-яке застосування на Java - це набір класів, що описують нові типи об'єктів.

Основні особливості об'єктної моделі Java. У мові використовується альтернативний підхід - спеціальний тип "інтерфейс .В Java застосовується строга типізація. Це означає, що будь-яка змінна і будь-який вираз має тип, відомий вже на момент компіляції. Такий підхід застосований для спрощення виявлення проблем, адже компілятор відразу повідомляє про помилки і вказує їх розташування в коді. Пошук же виняткових ситуацій (exceptions - так в Java називаються некоректні ситуації) під час виконання програми (runtime) зажадає складного тестування, при цьому причина дефекту може виявитися зовсім в іншому класі.

У Java існує всього 8 типів даних, які не є об'єктами. *Вони були визначені з найпершої версії і ніколи не мінялися. Це п'ять цілочисельних типів: byte, short, int, long, а також до них відносять символьні char. Потім два дробові типи float і double і, нарешті, булевий тип boolean. Такі типи називаються «прості», або «примітивні» (від англійського primitive), і вони детально розглядаються в лекції, присвяченій типам даних. Решта всіх типів - об'єктні або посилальні (англ. reference).*

Синтаксис Java дійсно створений на основі синтаксису мов C/c++, так що якщо подивитися на початковий код програм, написаних на цих мовах і на Java, то не відразу вдається зрозуміти, яка з них на якій мові написана, **але вважати, що Java така ж мова, як і C++, - велика помилка.**Модель Java була побудована наново, причому відповідно до зовсім інших цілей. Що ж до об'єктної моделі, то вона швидше була побудована за зразком таких мов, як Smalltalk від IBM, або розроблена ще в 60-і роки в Норвезькому Обчислювальному Центрі мова Simula, на яку посилається сам творець Java Джеймс Гослінг.3. властивість Java - легкість в освоєнні і розробці. І перше місце тут займає робота з пам'яттю.У Java із самого початку був введений механізм автоматичної збірки сміття (від англійського garbage collector). При розробці на Java програміст взагалі не думає про звільнення пам'яті. Віртуальна машина сама підраховує кількість посилань на кожен об'єкт, і якщо воно стає рівним нулю, то такий об'єкт позначається для обробки garbage collector. Таким чином, програміст повинен стежити лише за тим, щоб не залишалося посилань на непотрібні об'єкти. Складальник сміття - це фоновий потік

виконання, який регулярно проглядає існуючі об'єкти і видаляє вже не потрібні. З програми ніяк не можна вплинути на работу garbage collector, можна тільки явно ініціювати його черговий прохід за допомогою стандартної функції. Ясно, що це істотно спрощує розробку програм, особливо для початкуючих програмістів.Окрім введення garbage collector, були зроблені і інші кроки для полегшення розробки. Деякі з них вже згадувалися - відмова від множинного спадкоємства, спрощення синтаксису і ін. Можливість створення багатопотокових застосувань була реалізована в першій же версії Java (дослідження показали, що це дуже зручно для користувачів, а існуючі стандарти спираються на телетайпні системи, які застаріли багато років тому). Проте те, що створення і підтримка систем дійсно простіше на Java, ніж на C/C++, давно є загальновизнаним фактом. Втім, все-таки ці мови створені для різних цілей, і кожен має свої незаперечні переваги.4. властивість Java - безпека . Початкова націленість на розподілені застосування, і особливо рішення виконувати аплети на клієнтській машині, зробили питання захисту одним з найпріоритетніших. При роботі будь-якої віртуальної машини Java виконує цілий комплекс мерів. Далі приводиться лише короткий опис деяких з них. **По-перше**, це правила роботи з пам'яттю. Вже мовилося, що очищення пам'яті проводиться автоматично. Резервування її також визначається JVM, а не компілятором, або явним чином з програми, розробник може лише вказати, що він хоче створити ще один новий об'єкт. Покажчики по фізичних адресах відсутні принципово. **По-друге,** наявність віртуальної машини-інтерпретатора значно полегшує відсікання небезпечних кодів на кожному етапі роботи. Спочатку байт-код завантажується в систему, як правило, у вигляді class-файлів. JVM ретельно перевіряє, чи всі вони підкоряються загальним правилам безпеки Java і чи не створені зловмисниками за допомогою якихось інших засобів (і чи не спотворені при передачі). Потім, під час виконання програми, інтерпретатор легко може перевірити кожну дію на допустимість. Можливості класів, які були завантажені з локального диска або по мережі, істотно розрізняються (користувач легко може призначати або відміняти конкретні права). Наприклад, аплети за умовчанням ніколи не дістануть доступ до локальної файлової системи. Такі вбудовані обмеження є у всіх стандартних бібліотеках Java.Нарешті, існує механізм підписання аплетів і інших застосувань, що завантажуються по мережі. Спеціальний сертифікат гарантує, що користувач отримав код саме в тому вигляді, в якому його випустив виробник. Це, звичайно, не дає додаткових засобів захисту, але дозволяє клієнтові або відмовитися від роботи з додатками ненадійних виробників, або відразу побачити, що до програми внесені неавторизовані зміни. У гіршому разі він знає, хто відповідальний за заподіяний збиток. Сукупність описаних властивостей Java дозволяє стверджувати, що мова вельми пристосована для розробки Internet- і інтранет (внутрішні мережі корпорацій)-додатків.Важлива відмітна особливість Java - це його динамічність. Мова дуже вдало задумана, в його розвитку беруть участь сотні тисяч розробників і багато крупних компаній. Основні етапи цього розвитку коротко висвітлені в наступному розділі.Отже, підведемо підсумки. Java-платформа володіє наступними перевагами:

- переносимість, або кроссплатформеність;
- об'єктна орієнтованість, створена ефективна об'єктна модель;
- звичний синтаксис C/C++;
- вбудована і прозора модель безпеки;
- орієнтація на Internet-завдання, мережеві розподілені застосування;
- динамічність, легкість розвитку і додавання нових можливостей;
- простота освоєння.

Але не слід вважати, що легше освоєння означає, що вивчати мову не потрібно зовсім. Щоб писати дійсно хороші програми, створювати великі складні системи, необхідне чітке розуміння всіх базових концепцій Java і використовуваних бібліотек. Саме цьому і присвячений даний курс.

Основні версії і продукти Java Відразу обмовимося, що під продуктами тут розуміються програмні рішення від компанії Sun, реалізації, що є "зразками" (reference implementation). Отже, вперше Java була оголошена 23 травня 1995 року. Основними продуктами, доступними на той момент у вигляді бета-версій, були:

- Java language specification, JLS, специфікація мови Java (що описує лексику, типи даних, основні конструкції і так далі);
- специфікація JVM;
- Java Development Kit, JDK - засіб розробника, що складається в основному з утиліт, стандартних бібліотек класів і демонстраційних прикладів.

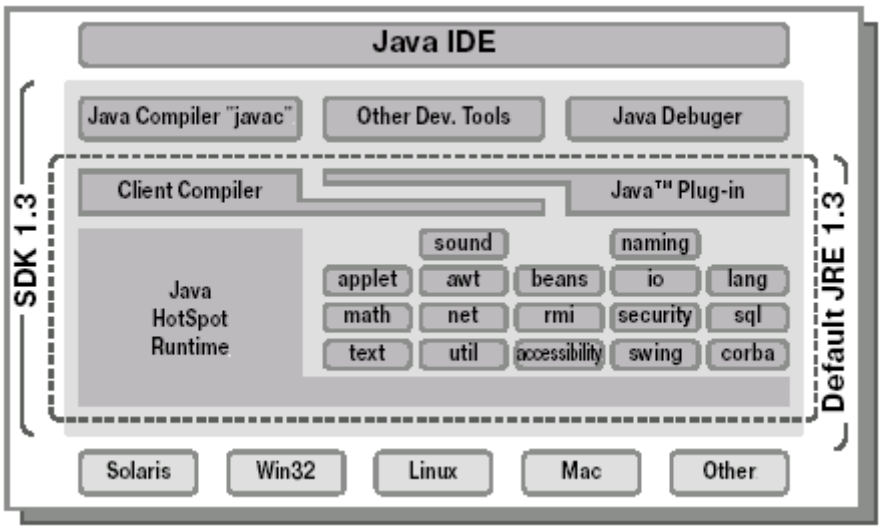
Специфікація мови була складена настільки вдало, що практично без змін використовується і до цього дня. Звичайно, було внесено велику кількість уточнень, докладніших описів, були додані і деякі нові можливості (наприклад, оголошення внутрішніх класів), проте основні концепції залишаються незмінними. Даний курс у великій мірі спирається саме на специфікацію мови.JDK довгий час було базовим засобом розробки додатків. Воно не містить ніяких текстових редакторів, а оперує тільки вже існуючими Java-файлами. Компілятор представлений утилітою javac (javacompiler). Віртуальна машина реалізована програмою java. Для тестових запусків аплетів існує спеціальна утиліта appletviewer. Нарешті, для автоматичної генерації документації на основі початкової коди додається засіб javadoc.

Перша версія містила всього 8 стандартних бібліотек:

- java.lang - базові класи, необхідні для роботи будь-якого застосування (назва - скорочення від language);
- java.util - багато корисних допоміжних класів;
- java.applet - класи для створення аплетів;
- java.awt, java.awt.peer - бібліотека для створення графічного інтерфейсу користувача (GUI), називається Abstract Window Toolkit, AWT, детально описується в лекції 11;
- java.awt.image - додаткові класи для роботи із зображеннями;
- java.io - робота з потоками даних (streams) і з файлами;
- java.net - робота з мережею.

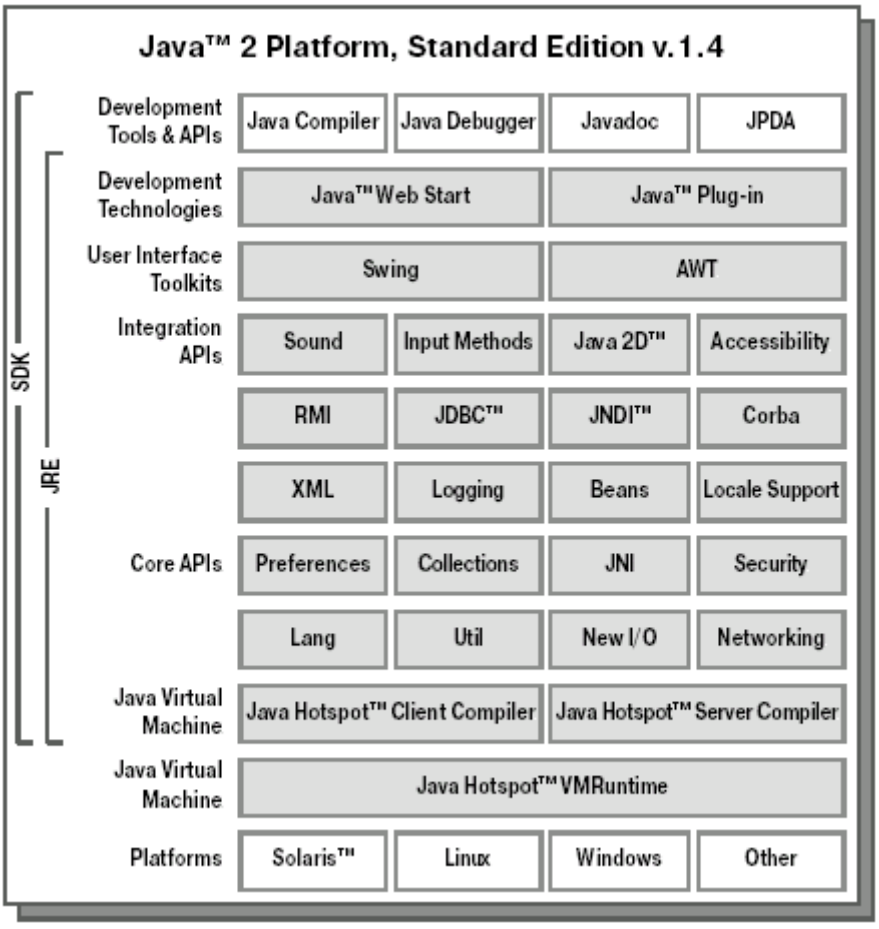
Таким чином, всі бібліотеки починаються з java, саме вони є стандартними. Всі останні (що починаються з com, org і ін.) можуть мінятися в будь-якій версії без підтримки сумісності.У березні 1999 року оголошується новий продукт - високошвидкісна платформа (engine) Java Hotspot. Була оптимізована робота з потоками виконання, істотно перероблені алгоритми автоматичного складальника сміття (garbage collector) і багато що інше. Прискорення дійсно було дуже істотним, завжди помітне неозброєним поглядом за декілька хвилин роботи з Java-додатками.Нова платформа може працювати в двох режимах - клієнтському і серверному. Режими розрізнялися настройками і іншими оптимізуючими алгоритмами. За умовчанням робота йде в клієнтському режимі.На

закінчення для демонстрації рівня розвитку Standard Edition приведемо стандартні діаграми, що описують всі технології, що становлять, з документації до версій 1.3 і



1.4.

1.1.Складові технології версії 1.3.



1.2. Складові технології версії 1.4.

Висновок

У цій лекції розглянуто, яка непроста ситуація склалася в корпорації Sun в епоху розвитку персональних комп'ютерів в кінці 1990 року. Патрік Нотон в своєму листі зумів виявити дійсні причини такого положення і позначити дійсні цілі для створення успішного продукту. Завдяки цьому за підтримки Джеймса Гослінга почався проект Green. Одним з продуктів, створених в рамках цього проекту, стала абсолютно нова платформа OAK. Для її просування Sun заснувала дочірню компанію Firstperson, але справжній успіх прийшов, коли платформу, перейменувавши в Java, зорієнтували на застосування в Internet.Глобальна мережа з'явилася в квітні 1993 року з виходом першого браузера mosaic 1.0 і завойовувала призначену для користувача аудиторію з вражаючою швидкістю.

Першим прикладом Java-додатків стали аплети, що запускаються за допомогою спеціального створеного браузера Hotjava. Нарешті, після майже чотирилітньої історії створення і розвитку, Java була офіційно представлена світу. Завдяки підписанню ліцензійної угоди з Netscape, ця подія стала справді тріумфальною. Були розглянуті різні варіанти застосування Java. Окремо була описана мова Java Script, яка, не дивлячись на схожість в назві, має не так багато спільного з Java. Детально розглянуті відмітні особливості java. Описані базові продукти від Sun: JDK і JRE. Коротко освітлена історія розвитку версій платформи Java, включаючи технології, що додаються, і продукти.

ТЕМА. *Основи об'єктно- орієнтованого програмування.*

План

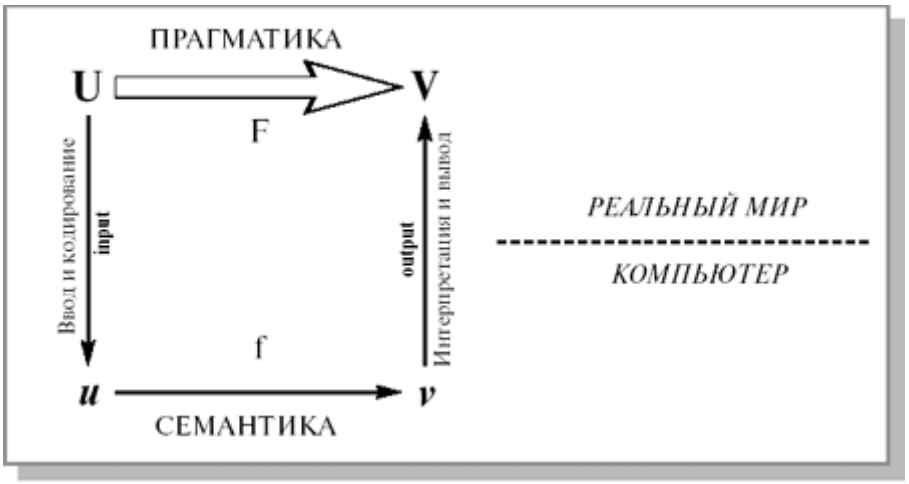
1. *Об'єктно-орієнтований підхід до розробки ПЗ.*
2. *Об'єкт. Основні властивості об'єктної моделі.*
3. *Клас. Види відносин між класами.*

Рекомендована література: 1,2 3.

Теоретичні відомості

Як показала практика, традиційні методи процедурного програмування не здатні справитися ні з наростаючою складністю програм і їх розробки, ні з необхідністю підвищення їх надійності. У другій половині 80-х років виникла настійна потреба в новій методології програмування, яка була б здатна вирішити весь цей комплекс проблем. Нею стало об'єктно-орієнтоване програмування (ООП).

Об'єктно-орієнтований підхід до проектування заснований на представленні наочної області завдання у вигляді безлічі моделей для незалежної від мови розробки програмної системи на основі її прагматики. **Прагматика** визначається метою розробки програмної системи, наприклад, обслуговування клієнтів банку, управління роботою аеропорту, обслуговування чемпіонату світу по футболу і тому подібне. У формулюванні мети беруть участь предмети і поняття реального світу, що мають відношення до створюваної системи. При об'єктно-орієнтованому підході ці предмети і поняття замінюються моделями, тобто певними формальними конструкціями.



Мал. 2.2. Семантика (сенс програми з погляду комп'ютера, що виконує її) і прагматика (сенс програми з погляду її користувачів) [3].

Модель містить не всі ознаки і властивості предмету, що представляється нею, або поняття, а тільки ті, які істотні для програмної системи, що розробляється. Таким чином, модель "бідніша" предмету, що простіше представляється нею, або поняття.

Простота моделі по відношенню до реального предмету дозволяє зробити її формальною. Завдяки такому характеру моделей при розробці можна чітко виділити всі залежності і операції над ними в створюваній програмній системі. Це спрощує як розробку і вивчення (аналіз) моделей, так і їх реалізацію на комп'ютері.

Об'єктно-орієнтований підхід володіє такими перевагами, як:

- зменшення складності програмного забезпечення;
- підвищення надійності програмного забезпечення;
- забезпечення можливості модифікації окремих компонентів програмного забезпечення без зміни останніх його компонентів;
- забезпечення можливості повторного використання окремих компонентів програмного забезпечення.

Систематичне застосування об'єктно-орієнтованого підходу дозволяє розробляти добре структуровані, надійні в експлуатації, програмні системи, що досить просто модифікуються. Цим пояснюється інтерес програмістів до об'єктно-орієнтованого підходу і об'єктно-орієнтованих мов програмування. ООП є одним з напрямів теоретичного і прикладного програмування, що найінтенсивніше розвиваються.

Об'єкти. *За визначенням називатимемо* ***об'єктом*** *поняття, абстракцію або будь-який предмет з чітко окресленими межами, що має сенс в контексті даної прикладної проблеми*. Введення об'єктів переслідує дві мети:

- розуміння прикладного завдання (проблеми);
- введення основи для реалізації на комп'ютері.

Приклади об'єктів: *кватирка, Банк "Імперіал", Петро Сидоров, справа № 7461, ощадкнижка і так далі*. Кожен об'єкт має певний час життя. В процесі виконання програми, або функціонування якої-небудь реальної системи, можуть створюватися нові об'єкти і знищуватися що вже існують.

Об'єкт - це мислима або реальна суть, що володіє характерною поведінкою, і відмітними характеристиками, що є важливим в наочній області. Кожен об'єкт має стан, володіє чітко певною поведінкою і унікальною ідентичністю.

Стан. Розглянемо приклад. Будь-яка людина може знаходитися в деякому положенні (стані): стояти, сидіти, лежати, і - в той же час здійснювати якісь дії. Наприклад, людина може стрибати, якщо вона стоїть, і не може, якщо вона лежить, для цього їй потрібно буде спочатку встати. Також в об'єктно-орієнтованому програмуванні **стан об'єкту** може визначатися наявністю або відсутністю зв'язків між модельованим об'єктом та іншими об'єктами.

Наприклад, якщо у людини є вудка (у нього є зв'язок з об'єктом "Вудка"), він може ловити рибу, а якщо вудки немає, то така дія неможлива. З цих прикладів видно, що набір дій, які може здійснювати людина, залежить від параметрів об'єкту, що його моделює.

Для розглянутих вище прикладів такими характеристиками, або атрибутами, об'єкту "Чоловік" є:

- поточне положення людини (стоїть, сидить, лежить);
- наявність вудки (є чи ні).

У конкретному завданні можуть з'явитися і інші властивості, наприклад, фізичний стан, здоров'я (хвора людина зазвичай не стрибає).

Стан (state) - сукупний результат поведінки об’єкту: одна із стабільних умов, в яких об’єкт може існувати, охарактеризованих кількісно; у будь-який момент часу стан об’єкту включає перелік (зазвичай статичний) властивостей об’єкту і поточні значення (зазвичай динамічні) цих властивостей.

Поведінка. Для кожного об’єкту існує певний набір дій, які з ним можна провести. Наприклад, можливі дії з деяким файлом операційної системи ПК:

- створити;
- відкрити;
- читати з файлу;
- писати у файл;
- закрити;
- видалити.

Результат виконання дій залежить від стану об’єкту на момент здійснення дії, тобто не можна, наприклад, видалити файл, якщо він відкритий ким-небудь (заблокований). В той же час дії можуть міняти внутрішній стан об’єкту - при відкритті або закритті файлу властивість "відкритий" набуває значень "та чи ні", відповідно.

Програма, написана з використанням ООП, зазвичай складається з безлічі об’єктів, і все ці об’єкти взаємодіють між собою. *Взаємодія між об’єктами в програмі відбувається за допомогою передачі повідомлень між ними.*

У термінології об’єктно-орієнтованого підходу поняття "дія", "повідомлення" і "метод" є синонімами. Тобто вирази "виконати дію над об’єктом", "викликати метод об’єкту" і "послати повідомлення об’єкту для виконання якої-небудь дії" еквівалентні. Остання фраза з’явилася з наступної моделі. Програму, побудовану за технологією ООП, можна уявити собі як віртуальний простір, заповнений об’єктами, які умовно "живуть" деяким життям. Їх активність виявляється в тому, що вони викликають один у одного методи, або посилають один одному повідомлення. **Зовнішній інтерфейс об’єкту**, *або набір його методів, - це опис того, які повідомлення він може приймати.*

Поведінка (behavior) - дії і реакції об’єкту, виражені в термінах передачі повідомлень і зміни стану; видима ззовні і відтворна активність об’єкту.

Унікальність. Унікальність - це те, що відрізняє об’єкт від інших об’єктів. Наприклад, у вас може бути декілька однакових монет. Навіть якщо абсолютно всі їх властивості (атрибути) однакові (рік випуску, номінал і так далі) і при цьому ви можете використовувати їх незалежно одна від одної, вони як і раніше залишаються різними монетами.

У машинному уявленні під параметром унікальності об’єкту найчастіше розуміється адреса розміщення об’єкту в пам’яті.

Identity (унікальність) об’єкту полягає в тому, що завжди можна визначити два посилання на один і той же об’єкт або на різні об’єкти. При цьому два об’єкти можуть у всьому бути схожими, їх образ в пам’яті може представлятися однаковими послідовностями байтів, але, проте, їх **унікальність** може бути різна.

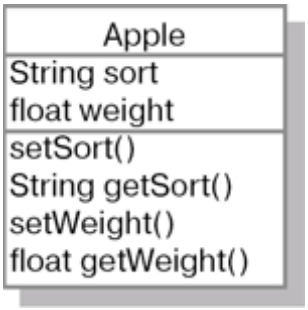
Найбільш поширеною помилкою є розуміння унікальності як імені посилання на об’єкт. Це невірно, оскільки на один об’єкт може указувати декілька посилань, і посилання можуть міняти свої значення (посилатися на інші об’єкти).

Отже, унікальність (identity) - властивість об'єкту; те, що відрізняє його від інших об'єктів (автор не згоден з перекладом російського видання

Класи Всі монети з попереднього прикладу належать одному і тому ж классу об'єктів (саме з цим пов'язана їх подібність). Номінальна вартість монети, метал, з якого вона виготовлена, форма - це атрибути класу. Сукупність атрибутів і їх значень характеризує об'єкт. Разом з терміном "атрибут" часто використовують терміни "властивість" і "поле", які воб'єктно-орієнтованому програмуванні є синонімами. Всі об'єкти одного і того ж класу описуються однаковими наборами атрибутів. *Проте об'єднання об'єктів в класи визначається не наборами атрибутів, а семантикою.* Так, наприклад, об'єкти "стайня" і "кінь" можуть мати однакові атрибути: ціна і вік. При цьому вони можуть відноситися до одного класу, якщо розглядаються в завданні просто як товар, або до різних класів, якщо в рамках поставленого завдання використовувати по-різному, тобто над ними здійснюватимуться різні дії.

Об'єднання об'єктів в класи дозволяє розглянути завдання в більш загальній постановці. Клас має ім'я (наприклад, "кінь"), яке відноситься до всіх об'єктів цього класу. Крім того, в класі вводяться імена атрибутів, які визначені для об'єктів. У цьому сенсі описання класу аналогічно опису типу структури або запису (record), що широко застосовуються в процедурному програмуванні; при цьому кожен об'єкт має той же сенс, що і екземпляр структури (змінна або константа відповідного типу).

Формально клас - *це шаблон поведінки об'єктів певного типу із заданими параметрами, що визначають стан.* Всі екземпляри одного класу (об'єкти, породжені від одного класу) мають один і той же набір властивостей і загальна поведінка, тобто однаково реагують на однакові повідомлення.



Відповідно до UML (Unified Modelling Language - уніфікована мова моделювання), клас має наступне графічне уявлення.

Клас зображається у вигляді прямокутника, що складається з трьох частин. У верхній частині поміщається назван класу, в середній - властивості об'єктів класу, в ніжній - дії, які можна виконувати з об'єктами данного класу (методи).

Кожен клас також може мати **спеціальні методи**, які автоматично викликаються при створенні і знищенні об'єктів цього класу:

- конструктор (constructor) - виконується при створенні об'єктів;
- **деструкція (destructor)** - виконується при знищенні об'єктів.

Зазвичай конструктор і деструкція мають спеціальний синтаксис, який може відрізнятися від синтаксису, використовуваного для написання звичайних методів класу.

Інкапсуляція. *Інкапсуляція (encapsulation) - це захована реалізації класу і відділення його внутрішнього уявлення від зовнішнього (інтерфейсу).* При використанні

об'єктно-орієнтованого підходу не прийнято застосовувати прямий доступ до властивостей якого-небудь класу з методів інших класів. Для доступу до властивостей класу прийнято задіювати спеціальні методи цього класу для отримання і зміни його властивостей. Усередині об'єкту дані і методи можуть володіти різним ступенем відвертості (або доступності). Ступені доступності, прийняті в мові Java, детально будуть розглянуті в лекції6. Вони дозволяють тонше управляти властивістю інкапсуляції.

Відкриті члени класу складають зовнішній інтерфейс об'єкту. Це та функціональність, яка доступна іншим класам. Закритими зазвичай оголошуються всі властивості класу, а також допоміжні методи, які є деталями реалізації і від яких не повинні залежати інші частини системи.

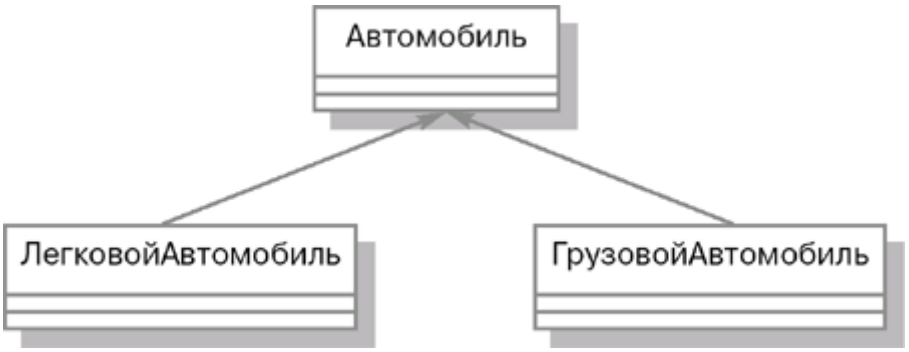
Завдяки захованню реалізації за зовнішнім інтерфейсом класу можна міняти внутрішню логіку окремого класу, не міняючи код решти компонентів системи. Ця властивість називається **модульність.**

Забезпечення доступу до властивостей класу тільки через його методи також дає ряд переваг. **По-перше**, так набагато простіше контролювати коректні значення полів, адже пряме звернення до властивостей відстежувати неможливе, а значить, їм можуть привласнити некоректні значення.

По-друге, не складе труднощів змінити спосіб зберігання даних. Якщо інформація почне зберігатися не в пам'яті, а в довготривалому сховищі, такому як файлова система або база даних, потрібно буде змінити лише ряд методів одного класу, а не вводити цю функціональність у всі частини системи. Програмний код, написаний з використанням даного принципу, легко відлагоджувати.

Спадкоємство. *Спадкоємство (inheritance) - це відношення між класами, при якому клас використовує структуру або поведінку іншого класу (одиночне спадкоємство), або інших (множинне спадкоємство) класів.* Спадкоємство вводить ієрархію "загальне/особисте", в якій підклас успадковує від одного або декількох більш загальних суперкласів. Підкласи зазвичай доповнюють або перевизначають успадковану структуру і поведінку.

Як приклад можна розглянути завдання, в якому необхідно реалізувати класи "Легковий автомобіль" і "Вантажний автомобіль". Очевидно, ці два класи мають загальну функціональність. Так, обидва вони мають 4 колеса, двигун, можуть переміщатися і так далі. Всіма цими властивостями володіє будь-який автомобіль, незалежно від того, вантажний він або легковий, 5-ти або 12-тимісцевий. Розумно винести ці загальні властивості і функціональність в окремий клас, наприклад, "Автомобіль" і успадковувати від нього класи "Легковий автомобіль" і "Вантажний автомобіль", щоб уникнути повторного написання одного і того ж коду в різних класах.



Відношення узагальнення позначається суцільною лінією з трикутною стрілкою на кінці. Стрілка указує на більш загальний клас (клас-предок або суперклас), а її відсутність - на більш спеціальний клас (клас-нащадок або підклас). Використання спадкоємства сприяє зменшенню об’єму коду, створеного для опису схожої суті, а також сприяє написанню ефективнішого і гнучкішого коду.

У розглянутому прикладі застосовано одиночне спадкоємство. Деякий клас також може успадковувати властивості і поведінку відразу декількох класів. Найбільш популярним прикладом застосування множинного успадкування являється проектування системи обліку товарів в зоомагазині. Всі тварини в зоомагазині є спадкоємцями класу "Тварина", а також спадкоємцями класу "Товар". Тобто всі вони мають вік, потребують їжі і води і в той же час мають ціну і можуть бути продані.

Множинне спадкоємство на діаграмі зображається точно так, як і одиничне, за винятком того, що лінії спадкоємства сполучають клас-нащадок відразу з декількома суперкласами. Не всі об’єктно-орієнтовані мови програмування містять мовні конструкції для опису множинного спадкоємства. У мові Java множинне спадкоємство має обмежену підтримку через інтерфейси і буде розглянуто в лекції 8.

Поліморфізм. Поліморфізм є одним з фундаментальних понять в об’єктно-орієнтованому програмуванні ряду із спадкоємством і інкапсуляцією. Слово "поліморфізм" грецького походження і означає той, що "має багато форм". Щоб зрозуміти, що воно означає стосовно об’єктно-орієнтованого програмування, розглянемо **приклад.** Припустимо, ми хочемо створити векторний графічний редактор, в якому нам потрібно описати у вигляді класів набір графічних примітивів, - Point, Line, Circle, Box і так далі. У кожного з цих класів визначимо метод draw для відображення відповідного примітиву на екрані. Очевидно, доведеться написати код, який при необхідності відобразити малюнок послідовно перебиратиме всі примітиви, на момент відсортування, що знаходяться на екрані, і викликати метод draw у кожного з них. Людина, незнайома з поліморфізмом, найімовірніше, створить декілька масивів (окремий масив для кожного типу примітивів) і напише код, який послідовно перебере елементи з кожного масиву і викличе у кожного елементу метод draw. В результаті вийде приблизно наступний код:

```
…
//створення пустого масиву, який може
// містити об’єкти Point з максимальним
// об’ємом 1000
Point[] p = new Point[1000];
Line[] l = new Line[1000];
Circle[] c = new Circle[1000];
Box[] b = new Box[1000];
…
// припустимо, в цьому місці відбувається
// заповнення всіх масивів відповідними
// об’єктами
…
for(int i = 0; i < p.length;i++) {
//цикл з перебором всіх комірок масиву.
  //виклик методу draw() у випадку
 // якщо комірка не порожня.
  if(p[i]!=null) p[i].draw();
}
```

```
for(int i = 0; i < l.length;i++) {
  if(l[i]!=null) l.draw();
}
for(int i = 0; i < c.length;i++) {
  if(c[i]!=null) c.draw();
}
for(int i = 0; i < b.length;i++) {
  if(b[i]!=null) b.draw();
}
…
```

Недоліком написаного вище коду є дублювання практично ідентичного коду для відображення кожного типу примітивів. Також незручне те, що при подальшій модернізації нашого графічного редактора і додаванні можливості малювати нові типи графічних примітивів, наприклад Text, Star і так далі, при такому підході доведеться міняти існуючий код і додавати в нього визначення нових масивів, а також обробку елементів, що містяться в них. Використовуючи поліморфізм, ми можемо значно спростити реалізацію подібної функціональності. Перш за все, створимо загальний батьківський клас для всіх наших класів. Хай таким класом буде Point. У кожного з дочірніх класів метод draw перевизначений так, щоб відображати екземпляри кожного класу відповідним чином.

Для описаної вище ієрархії класів, використовуючи поліморфізм, можна написати наступний код:

```
…
Point p[] = new Point[1000];
p[0] = new Circle();
p[1] = new Point();
p[2] = new Box();
p[3] = new Line();
…
for(int i = 0; i < p.length;i++) {
  if(p[i]!=null) p[i].draw();
}
…
```

У описаному вище прикладі масив p[] може містити будь-які об’єкти, породжені від спадкоємців **клас capoint**. При виклику якого-небудь методу у будь-якого з елементів цього масиву буде виконаний метод того об’єкту, який міститься в осередку масиву. Наприклад, якщо в осередку p[0] знаходиться об’єкт Circle, то при виклику методу draw таким чином:

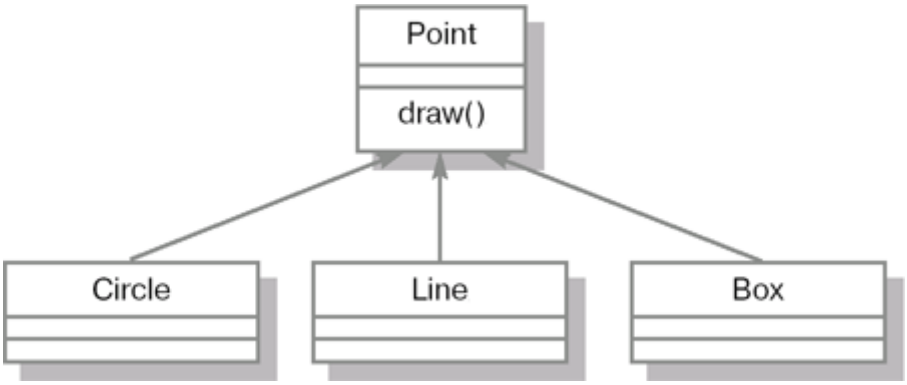
```
p[0].draw()
```

намалюється круг, а не крапка.

На закінчення приведемо формальне визначення поліморфізму.**Поліморфізм (polymorphism)** - *положення теорії типів, згідно якому імена (наприклад, змінних) можуть позначати об’єкти різних (але що мають загального батька) класів.* Отже, будь-який об’єкт, що позначається поліморфним ім’ям, може по-своєму реагувати на якийсь загальний набір операцій.

У ***процедурному програмуванні*** *теж існує поняття поліморфізму, яке відрізняється від розглянутого механізму в ООП.* Процедурний поліморфізм припускає

можливість створення декількох процедур або функцій з одним і тим же ім'ям, але різною кількістю або різними типами переданих параметрів. Такі однойменні функції називаються **перевантаженими**, а само явище - перевантаженням (overloading). Перевантаження функцій існує і в ООП і називається перевантаженням методів.



Мал. 2.3. Приклад ієрархії класів.

Прикладом використання перевантаження методів в мові Java може служити **клас Printwriter**, який застосовується, зокрема, для виведення повідомлень на консоль. Цей клас має безліч методів **println**, які розрізняються типами і/або кількістю вхідних параметрів. Ось лише декілька з них:

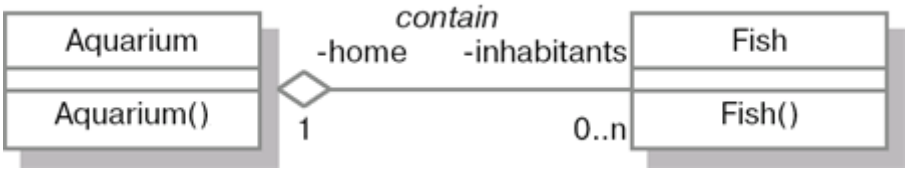
```
void println()
  // перехід на новий рядок
void println(boolean x)
  // виводить значення булевої
  // змінної (true или false)
void println(String x)
  // виводить рядок - значення
  // текстового параметру.
```

Певні складнощі виникають при виклику перевантажених методів. У Java існують спеціальні правила, які дозволяють вирішувати цю проблему. Вони будуть розглянуті у відповідній лекції.

Типи відносин між класами. Як правило, будь-яка програма, написана на об'єктно-орієнтованій мові, є деякий набір зв'язаних між собою класів. Можна провести аналогію між написанням програми і будівництвом будинку. Подібно до того, як стіна складається з цегли, комп'ютерна програма з використанням ООП будується з класів. Причому ці класи повинні мати уявлення один про одного, для того, щоб спільно виконувати поставлене завдання. Можливі наступні зв'язки між класами в рамках об'єктної моделі (приводяться лише найбільш прості і часто використовувані види зв'язків):

- агрегація (Aggregation);
- асоціація (Association);
- спадкоємство (Inheritance);
- метакласи (Metaclass).

Агрегація. *Відношення між класами типу "містить" (contain) або "складається з" називається агрегацією, або включенням.* Наприклад, якщо акваріум наповнений водою і в нім плавають рибки, то можна сказати, що акваріум агрегує в собі воду і рибок.



Таке відношення включення, або агрегації (aggregation), зображається лінією з ромбом на стороні того класу, який виступає як власник, або контейнер. Необов'язкова назва відношення записується посередині лінії. У нашому прикладі відношення contain є двонаправленим. Об'єкт класу Aquarium містить декілька об'єктів Fish. В той же час кожна рибка "знає", в якому саме акваріумі вона живе. Кожен клас має свою роль в агрегації, яка вказує, яке місце займає клас в даному відношенні. Ім'я ролі не є обов'язковим елементом позначень і може бути відсутнім на діаграмі. У прикладі можна бачити роль home класу Aquarium (акваріум є будинком для рибок), а також роль inhabitants класу Fish (рибки є мешканцями акваріума). Назва ролі зазвичай співпадає з назвою відповідного поля в класі. Зображення такого поля на діаграмі надмірно, якщо вже вказано ім'я ролі. Тобто в даному випадку клас Aquarium буде мати властивість (поле) inhabitants, а клас Fish - властивість home.

Число об'єктів, що беруть участь у відношенні, записується поряд з ім'ям ролі. Запис "0..n" означає "від нуля до безкінечності". Прийняті також позначення:

- "1..n" - від одиниці до безкінечності;
- "0" - нуль;
- "1" - один;
- "n" - фіксована кількість;
- "0..1" - нуль або один.

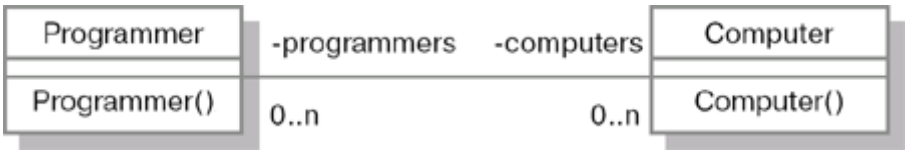
Код, що описує розглянуту модель і явище агрегації, може виглядати, наприклад, таким чином:

```
// визначення класу Fish
public class Fish {
  // визначення поля home
  // (посилання на об'єкт Aquarium)
 private Aquarium home;

  public Fish() {
  }
}
// визначення класу Aquarium
public class Aquarium {
 // визначення поля inhabitants
  // (масив посилань на об'єкти Fish)
  private Fish inhabitants[];
```

```
  public Aquarium() {
  }
}
```

Асоціація. *Якщо об'єкти одного класу посилаються на один або більш за об'єкти іншого класу, але ні в той, ні в інший бік відношення між об'єктами не носить характеру "володіння", або контейнеризації, таке відношення називають асоціацією (association).* Відношення асоціації зображається так само, як і відношення агрегації, але лінія, що зв'язує класи, - проста, без ромба.

Як приклад можна розглянути програміста і його комп'ютер. Між цими двома об'єктами немає агрегації, але існує чіткий взаємозв'язок. Так, завжди можна встановити, за якими комп'ютерами працює який-небудь програміст, а також які люди користуються окремо узятим комп'ютером. У розглянутому прикладі має місце асоціація "багато до багатьох".



В даному випадку між екземплярами класів Programmer і Computer в обидві сторони використовується відношення "0..n", оскільки програміст, в принципі, може не працювати з комп'ютером (якщо він теоретик або на пенсії). У свою чергу, комп'ютер може ніким не використовуватися (якщо він новий і ще не встановлений).

Код, відповідний розглянутому прикладу, буде, наприклад, наступним:

```
public class Programmer {
  private Computer computers[];
  public Programmer() {
  }
}
public class Computer {
 private Programmer programmers[];
  public Computer() {
  }
}
```

Метакласи. Отже, будь-який об'єкт має структуру, що складається з полів і методів. Об'єкти, що мають однакову структуру і семантику, описуються одним класом, який і є, по суті, визначенням структури об'єктів, породжених від нього.

Клас можна розглядати як об'єкт, у якого є властивості (ім'я, список полів і їх типи, список методів, список аргументів для кожного методу і так далі). Також клас може володіти поведінкою, тобто підтримувати методи. А раз для будь-якого об'єкту існує шаблон, що описує властивості і поведінку цього об'єкту, значить, його можна визначити і для класу. Такий шаблон, який задає різні класи, називається метакласом.

Щоб уявити собі, що таке метаклас, розглянемо приклад якоїсь бюрократичної організації. Вважатимемо, що всі класи в такій системі є строгими інструкціями, які

описують, що потрібно зробити, щоб породити новий об'єкт (наприклад, найняти нового службовця або відкрити новий відділ). Як і належить класам, вони описують всі властивості нових об'єктів (наприклад, зарплату і професійний рівень для співробітників, площу і майно для відділів) і їх поведінку (обов'язки службовців і функції підрозділів).

У свою чергу, написання нової інструкції можна строго регламентувати. Скажімо, необхідно використовувати спеціальний бланк, дотримуватися правил оформлення і заповнити всі обов'язкові поля (наприклад, номер інструкції і прізвища відповідальних працівників). *Така "інструкція інструкцій" і представлятиме собою метаклас в ООП.*

Отже, об'єкти породжуються від класів, а класи - від метакласу. Він, як правило, в системі тільки один. Але існують мови програмування, в яких можна створювати і використовувати власні метакласи, наприклад мова Python. Зокрема, функціональність метакласу може бути наступна: при формуванні класу він проглядатиме список всіх методів в класі і, якщо ім'я методу має вид set_xxx або get_xxx, автоматично створювати поле з ім'ям XXX, якщо такого не існує.

Оскільки метаклас сам є класом, то немає ніякого сенсу в створенні "мета-мета-класів".

У мові Java також є метаклас. Це клас, який так і називається - Class (описує класи), він розташовується в основній бібліотеці java.lang. Віртуальна машина використовує його по прямому призначенню. Коли завантажується черговий .class-файл, що містить опис нового класу, JVM породжує об'єкт класу class, який зберігатиме його структуру. Таким чином, Java використовує концепцію метакласу в найпрактичніших цілях. За допомогою Class реалізована підтримка статичних (static) полів і методів. Нарешті, цей клас містить ряд методів, корисних для розробників.

Переваги ООП. Від будь-якої методики розробки програмного забезпечення ми чекаємо, що вона допоможе нам у вирішенні наших завдань. *Але однією з найзначніших проблем проектування є складність*. Чим більше і складніше програмна система, тим важливіше розбити її на невеликі, чітко обкреслені частини. Щоб справитися з складністю, необхідно абстрагуватися від деталей. У цьому сенсі класи є вельми зручним інструментом.

- Класи дозволяють проводити конструювання з корисних компонентів, що володіють простими інструментами, що дозволяє абстрагуватися від деталей реалізації.
- Дані і операції над ними утворюють певну суть, і вони не розносяться за всією програмою, як нерідко буває у разі процедурного програмування, а описуються разом. Локалізація коду і даних покращує наочність і зручність супроводу програмного забезпечення.
- Інкапсуляція дозволяє привнести властивість модульності, що полегшує розпаралелювання виконання завдання між декількома виконавцями і оновлення версій окремих компонентів.

ООП дає можливість створювати розширювані системи. Це одна з основних переваг ООП, і саме вона відрізняє даний підхід від традиційних методів програмування. Розширюваність означає, що існуючу систему можна змусити працювати з новими компонентами, причому без внесення до неї яких-небудь змін. Компоненти можуть бути додані на етапі виконання програми.

Поліморфізм виявляється корисним переважно в наступних ситуаціях.

- Обробка різнорідних структур даних. Програми можуть працювати, не розрізняючи виду об’єктів, що істотно спрощує код. Новий вигляд може бути доданих у будь-який момент.

- Зміна поведінки під час виконання. На етапі виконання один об’єкт може бути замінений іншим, що дозволяє легко, без зміни коду, адаптувати алгоритм залежно від того, який використовується об’єкт.

- Реалізація роботи із спадкоємцями. Алгоритми можна узагальнити настільки, що вони вже зможуть працювати більш ніж з одним видом об’єктів.

Створення "каркасу" (framework). Незалежні від додатку частини наочної області можуть бути реалізовані у вигляді набору універсальних класів, або каркасу (framework), і надалі розширені за рахунок додавання частин, специфічних для конкретного застосування.

Часто багаторазового використання програмного забезпечення не вдається добитися через те, що існуючі компоненти вже не відповідають новим вимогам. ООП допомагає цього досягти без порушення роботи вже наявних клієнтів, що дозволяє витягувати максимум з багаторазового використання компонентів.

- Скорочується час на розробку, який може бути відданий іншим завданням.

- Компоненти багаторазового використання зазвичай містять значно менше помилок, ніж заново розроблені, адже вони вже не раз піддавалися перевірці.

- Коли якийсь компонент використовується відразу декількома клієнтами, поліпшення, що вносяться до його коду, одночасно роблять позитивний вплив і на безліч програм, що працюють з ним.

- Якщо програма спирається на стандартні компоненти, її структура і призначений для користувача інтерфейс стають більш уніфікованими, що полегшує її розуміння і спрощує використання.

Недоліки ООП.

Документування класів - завдання важче, ніж це було у разі процедур і модулів. Оскільки будь-який метод може бути перевизначений, в документації повинно говоритися не тільки про те, що робить даний метод, але і про те, в якому контексті він викликається.

У складних ієрархіях класів поля і методи зазвичай успадковуються з різних рівнів. І не завжди легко визначити, які поля і методи фактично відносяться до даного класу. Для отримання такої інформації потрібні спеціальні інструменти, на зразок навігаторів класів. Якщо конкретний клас розширюється, то кожен метод зазвичай скорочують перед передачею повідомлення базовому класу. Реалізація операції, таким чином, розподіляється по декількох класах, і щоб зрозуміти, як вона працює, нам доводиться уважно проглядати весь код.

Методи, як правило, коротші за процедури, оскільки вони здійснюють тільки одну операцію над даними, зате їх набагато більше. У коротких методах легко розібратися, але вони незручні тим, що код для обробки повідомлення іноді "розмазаний" по багатьох маленьких методах.

Інкапсуляцією даних не слід зловживати. Чим більше логіки і даних приховано в надрах класу, тим складніше за нього розширювати.

Багато хто вважає, що ООП є неефективним. Як же йде справа насправді? Ми повинні проводити чітку грань між неефективністю на етапі виконання, неефективністю в сенсі розподілу пам'яті і неефективністю, пов'язаною із зайвою універсалізацією.

1. Неефективність на етапі виконання. У мовах типу Smalltalk повідомлення інтерпретуються під час виконання програми шляхом здійснення їх пошуку в одній або декількох таблицях і за рахунок вибору відповідного методу. Звичайно, це повільний процес. І навіть при використанні якнайкращих методів оптимізації Smalltalk-програми вдесятеро повільніші оптимізованих С-програм.

   У гібридних мовах типу Oberon-2, Object Pascal і C++ відправка повідомлення приводить лише до виклику через покажчик процедурної змінної. На деяких машинах повідомлення виконуються лише на 10% повільніше, ніж звичайні процедурні виклики. І оскільки повідомлення зустрічаються в програмі набагато рідше за інші операції, їх дія на час виконання впливу практично не надає.

   Проте існує інший чинник, який впливає на час виконання: це інкапсуляція даних. Рекомендується не надавати прямий доступ до полів класу, а виконувати кожну операцію над даними через методи. Така схема приводить до необхідності виконання процедурного виклику кожного разу при доступі до даним. Проте якщо інкапсуляція використовується тільки там, де вона необхідна (тобто в тих випадках, коли це стає перевагою), то уповільнення цілком прийнятне.

2. Неефективність в сенсі розподілу пам'яті. Динамічне скріплення і перевірка типу на етапі виконання вимагають по ходу роботи інформації про тип об'єкту. Така інформація зберігається в дескрипторі типу і він виділяється один на клас. Кожен об'єкт має невидимий покажчик на дескриптор типу для свого класу. Таким чином, в об'єктно-орієнтованих програмах необхідна додаткова пам'ять виражається в одному покажчику для об'єкту і в одному дескрипторі типу для класу.

3. Зайва універсальність. Неефективність також може означати, що в програмі реалізовані надмірні можливості. У бібліотечному класі часто міститься більше методів, чим це реально необхідно. А оскільки зайві методи не можуть бути видалені, вони стають мертвим вантажем. Це не впливає на час виконання, але позначається на розмірі коду. Одне з можливих рішень - будувати базовий клас з мінімальним числом методів, а потім вже реалізовувати різні розширення цього класу, які дозволять наростити функціональність. Інший підхід - дати компонувальнику можливість видаляти зайві методи. Такі інтелектуальні компонувальники вже існують для різних мов і операційних систем.Але не можна стверджувати, що ООП неефективне. Якщо класи використовуються лише там, де це дійсно необхідно, то втрата ефективності через підвищення витрат пам'яті і меншої продуктивності незначна. Крім того, надійність програмного забезпечення і швидкість його написання часто буває важливішою, ніж продуктивність.

Висновок

У цій лекції розглянуто об'єктно-орієнтований підхід до розробки ПО, а також про те, що послужило передумовами до його появи і зробило його популярним. Були розглянуті ключові поняття ООП - об'єкт і клас. Далі були описані основні властивості об'єктної моделі - інкапсуляція, спадкоємство, поліморфізм. Основними видами відносин між класами є спадкоємство, асоціація, агрегація, метаклас. Також були описані правила зображення класів і зв'язків між ними на мові UML.

ТЕМА. *Лексика мови JAVA.*

План

1. *Основи лексичного аналізу програм JAVA.*
2. *Універсальне кодування Unicode.*
3. *Конструкції програми: коментарі, ідентифікатори, символьні і строкові літерали.*

Рекомендована література: 1, 3, 4

Теоретичні відомості

Кодування. Технологія Java, як платформа, спочатку спроектована для Глобальної мережі Internet, повинна бути багатомовною, а значить, звичайний набір символів ASCII (American Standard Code for Information Interchange, Американський стандартний код обміну інформацією), що включає лише латинський алфавіт, цифри і прості спеціальні знаки (дужки, розділові знаки, арифметичні операції і так далі), недостатній. Тому для запису тексту програми застосовується більш універсальне кодування unicode.Як відомо, Unicode представляє символи кодом з 2 байт, описуючи, таким чином, 65535 символів. Це дозволяє підтримувати практично всі поширені мови світу. Перші 128 символів співпадають з набором ASCII. Проте зрозуміло, що потрібне деяке спеціальне позначення, щоб мати можливість задавати в програмі будь-який символ unicode, адже ніяка клавіатура не дозволяє вводити більше 65 тисяч різних знаків. Ця конструкція представляє символ Unicode, використовуючи тільки символи ASCII. Наприклад, якщо в програму потрібно вставити знак з кодом `6917`, необхідно його представити в шістнадцятковому форматі`(1B05)` і записати:

`\u1b05` причому буква **u** повинна бути рядковою, а шістнадцяткові цифри `A, B, C, D, E, F` можна використовувати довільно, як заголовні, так і рядкові. Таким чином можна закодувати всі символи Unicode від\u0000 до `\uFFFF`. Букви російського алфавіту починаються з `\u0410` (тільки буква Е має код `\u0401`) по `\u044F` (код букви е `\u0451`). У останніх версіях JDK до складу демонстраційних застосувань і аплетів входить невелика програма Symboltest, що дозволяє проглядати весь набір символів Unicode. Її аналог нескладно написати самостійно. Для того, що кодувати великі тексти служить утиліта `native2ascii`, що також входить в JDK. Вона може працювати як в прямому режимі - переводити з різноманітних кодувань в Unicode, записаний ASCII-символами, так і в зворотному (опція `-reverse`) - з Unicode в стандартне кодування операційної системи.

У версіях мови Java до 1.1 застосовувався Unicode версії 1.1.5, в останньому випуску 1.4 використовується 3.0. Таким чином, Java стежить за розвитком стандарту і базується на сучасних версіях. Для будь-якої JDK точну версію unicode, використовувану в ній, можна дізнатися з документації до класу Character. Офіційний web-сайт стандарту, де можна отримати додаткову інформацію, - <http://www.unicode.org/>.

Отже, використовуючи просте кодування ASCII, можна ввести довільну послідовність символів Unicode. Далі буде показано, що Unicode використовується не для всіх лексем, а тільки для тих, для яких важлива підтримка багатьох мов, а саме: коментарі, ідентифікатори, символьні і строкові літерали. Для запису решти лексем цілком достатньо ASCII-символів.

Аналіз програми

Компілятор, аналізуючи програму, відразу розділяє її на:

- пропуски (white spaces);
- коментарі (comments);
- основні лексеми (tokens).

Пропуски. Пропусками в даному випадку називають всі символи, що розбивають текст програми на лексеми. Це як сам символ пропуску (space `\u0020`, десятковий код 32), так і знаки табуляції і перекладу рядка. Вони використовуються для розділення лексем, а також для оформлення коду, щоб його було легко читати. Наприклад, наступну частину програми (обчислення коріння квадратного рівняння):

```
double a = 1, b = 1, з = 6;
double D = b * b - 4 * a * з;

if (D >= 0) {
   double x1 = (-b + Math.sqrt (D)) / (2 * a);
   double x2 = (-b - Math.sqrt (D)) / (2 * a);
}
```

можна записати і у такому вигляді:

```
double a=1,b=1,c=6;double D=b*b-4*a*c;if(D>=0)
{double x1=(-b+Math.sqrt(D))/(2*a);double
x2=(-b-math.sqrt(D))/(2*a);}
```

У обох випадках компілятор згенерує абсолютно однаковий код. Єдине міркування, яким повинен керуватися розробник, - легкість читання і подальша підтримка такого коду.

Для розбиття тексту на рядки в ASCII використовується два символи - "повернення каретки" (carriage return, `CR \u000d`, десятковий код 13) і символ нового рядка (linefeed, `LF \u000a`, десятковий код 10). Щоб не залежати від особливостей використовуваної платформи, в Java застосовується найбільш гнучкий підхід. Завершенням рядка вважається:

- ASCII-символ LF, символ нового рядка;
- ASCII-символ CR, "повернення каретки";
- символ CR, за яким відразу ж слідує символ LF.

Розбиття на рядки важливе для коректного розбиття на лексеми (як вже говорилося, завершення рядка також служить роздільником між лексемами), для правильної роботи із строковими коментарями (див. наступну тему "Коментарі"), а також для виведення налагоджувальної інформації (при виведенні помилок компіляції і часу виконання указується, на якому рядку початкового коду вони виникли). Отже, пропусками в Java вважаються:

- ASCII-символ SP, space, пропуск \u0020, десятковий код 32;
- ASCII-символ HT, horizontal tab, символ горизонтальної табуляції \u0009, десятковий код 9;
- ASCII-символ FF, form feed, символ перекладу сторінки (був введений для роботи з принтером) \u000c, десятковий код 12;
- завершення рядка.

Коментарі. Коментарі не впливають на результуючий бінарний код і використовуються тільки для введення пояснень до програми.

У Java коментарі бувають двох видів:

- рядкові
- блокові

Рядкові коментарі починаються з ASCII-символов // і тривають до кінця поточного рядка. Як правило, вони використовуються для пояснення саме цього рядка, наприклад:

```
int y=1970; // рік народження
```

Блокові коментарі розташовуються між ASCII-символами /* і */, можуть займати довільну кількість рядків, наприклад:

```
/*
   Цей цикл не може починатися з нуля
   через особливості алгоритму
*/
for (int i=1; i<10; i++) {
...
}
```

Часто блокові коментарі оформляють таким чином (кожен рядок починається з *):

```
/*
   * Опис алгоритму роботи
   * наступного циклу while
   */
```

```
while (x > 0) {
   ...
}
```

Блоковий коментар не обов'язково повинен розташовуватися на декількох рядках, він може навіть знаходитися в середині оператора:

```
float s = 2*math.PI/*getRadius()*/;
// Закоментовано для відлаштування
```

В даному прикладі блоковий коментар розбиває арифметичні операції. Вираз `Math.PI` надає значення константи PI, визначене в класі `Math`. Виклик методу `getradius()` тепер закоментований і не буде проведений, змінна `s` завжди набуватиме значення `2 PI`. Завершує рядок рядковий коментар.

Коментарі не можуть знаходитися в символьних і строкових літералах, ідентифікаторах (ці поняття детально розглядаються далі в цій лекції). Наступний приклад містить **випадки неправильного застосування коментарів:**

```
// В даному прикладі текст /*.*/ стане просто
// частиною рядка s
String s = "text/*just text*/";
/*
  // Наступний рядок стане причиною помилки
  //при компіляції, оскільки коментар розбив

   ім’я методу getradius()
*/
circle.get/*comment*/Radius();
```

А такий код допустимий:

```
// Коментар може розділяти виклики функцій:
circle./*comment*/getRadius();
// Коментар може замінювати пропуски:
int/*comment*/x=1;
```

У останньому рядку між назвою типу даних `int` і назвою змінною `x` обов'язково повинен бути пропуск або, як в даному прикладі, коментар.

Коментарі не можуть бути вкладеними. Символи `/* */`, `//` не мають ніякого особливого значення усередині вже відкритих коментарів, як рядкових, так і блокових. Таким чином, в прикладі

```
/* початок коментаря /* // /** завершення: */
```

описаний тільки один блоковий коментар. А в наступному прикладі (рядки коди пронумеровані для зручності)

```
1. /*
2.    comment
3.    /*
4.    more comments
5.    */
6.    finish
7. */
```

компілятор видасть помилку. Блоковий коментар почався в рядку 1 з комбінації символів `/*`. Друга відкриваюча комбінація `/*` на рядку 3 буде проігнорована, оскільки знаходиться вже усередині коментаря. Символи `*/` у рядку 5 завершать його, а рядок 7 породить помилку - спроба закрити коментар, який не був початий.

Будь-які коментарі повністю віддаляються з програми під час компіляції, тому їх можна використовувати необмежено, не побоюючись, що це вплине на бінарний код. Основне їх призначення - зробити програму простою для розуміння, у тому числі і для інших розробників, яким доведеться в ній розбиратися з якої-небудь причини. Також коментарі часто використовуються для тимчасового виключення частин коди, наприклад:

```
int x = 2;
int y = 0;
/*
if (x > 0)
   y = y + x*2;
else
   y = -y - x*4;
*/
y = y*y;// + 2*x;
```

В даному прикладі закоментований вираз `if-else` і оператор складання `+2*x`.

Як вже говорилося вище, коментарі можна писати символами Unicode, тобто на будь-якій мові, зручній розробникові.

Окрім цього, існує особливий вид блокового коментаря - коментар розробника. Він застосовується для автоматичного створення документації коду. У стандартне постачання JDK, починаючи з версії 1.0, входить спеціальна **утиліта javadoc**. На вхід їй подається початковий код класів, а на виході виходить зручна документація в HTML-форматі, яка описує всі класи, всі їх поля і методи. При цьому активно

використовуються гіперпосилання, що істотно спрощує вивчення програми (наприклад, читаючи опис методу, можна за допомогою одного натискання мишки перейти на опис типів, використовуваних як аргументи або повертаючого значення). Проте зрозуміло, що однієї назви методу і перерахування його аргументів недостатньо для розуміння його роботи. Необхідні додаткові пояснення від розробника.

Коментар розробника записується так само, як і блоковий. Єдина відмінність в початковій комбінації символів - для документації коментар необхідно починати з `/**`. Наприклад:

```
/**
   * Обчислення модуля цілого числа.
   * Цей метод повертає
   * абсолютне значення аргументу x.
 */
int getabs(int x){
   if (x>=0)
      return x;
   else
      return -x;
}
```

Перша пропозиція повинна містити коротке резюме всього коментаря. Надалі воно буде використано як пояснення цієї функції в списку всіх методів класу (нижче будуть описані всі конструкції мови, для яких застосовується коментар розробника). Оскільки в результаті створюється HTML-документація, то і коментар необхідно писати по правилах HTML. Допускається застосування тегів, таких як `<b>` і `<p>` . Проте теги заголовків з `<h1>` по `<h6>` і `<hr>` використовувати не можна, оскільки вони активно застосовуються javadoc для створення структури документації.

Символ `*` на початку кожного рядка і попередні йому пропуски і знаки табуляції ігноруються. Їх можна не використовувати взагалі, але вони зручні, коли необхідне форматування, скажімо, в прикладах коду.

```
/**
   * Перша пропозиція - коротке
   * опис методу.
   * <p>
   * Так оформляється приклад коду:
 * <blockquote>
   * <pre>
```

```
     * if (condition==true) {
     *    x = getwidth();
     *    y = x.getHeight();
   * }
     * </pre></blockquote>
     * А так описується HTML-список:
     * <ul>
     * <li> можна використовувати похилий шрифт
     * <i>курсив</i>
     * <li>ілі жирний <b>жірний</b>.
   * </ul>
     */
public void calculate (int x, int y) {
  ...
}
```

З цього коментаря згенерує HTML-код, що виглядає приблизно так:

Перша пропозиція - короткий опис методу.

Так оформляється приклад коду:

```
if (condition==true) {
  x = getwidth();
    y = x.getHeight();
}
```

А так описується HTML-список:

- Можна використовувати похилий шрифт курсив
- або жирний жирний.

Нарешті, javadoc підтримує спеціальні теги. Вони починаються з символу @. Докладний опис цих тегів можна знайти в документації. Наприклад, можна використовувати тег @see, щоб послатися на інший клас, поле або метод, або навіть на інший Internet-сайт.

```
     /**
     * Короткий опис.
     *
     * Розгорнений коментар.
     *
     * @see java.lang.String
   * @see java.lang.Math#PI
     * @see <a href="java.sun.com">Official
```

```
 * Java site</a>
   */
```

Перше посилання указує на клас `String` (`java.lang` - назва бібліотеки, в якій знаходиться цей клас), друга, - на полі `PI` класу `Math` (символ `#` розділяє назву класу і його полів або методів), третя посилається на офіційний сайт Java.

Коментарі розробника можуть бути записані перед оголошенням класів, інтерфейсів, полів, методів і конструкторів. Якщо записати коментар `/** . */` у іншій частині коду, то помилки не буде, але він не потрапить в документацію, javadoc, що генерується. Крім того, можна описати пакет (так називаються бібліотеки, або модулі, в Java). Для цього необхідно створити спеціальний файл `package.html`, зберегти в нім коментар і помістити його в каталог пакету. HTML-текст, що міститься між тегами `<body>` і `</body>`, буде поміщений в документацію, а перша пропозиція використовуватиметься для короткої характеристики цього пакету.

Лексеми. Отже, ми розглянули пропуски (у широкому сенсі цього слова, тобто всі символи, що відповідають за форматування тексту програми) і коментарі, вживані для введення пояснень до коду. З погляду програміста вони застосовуються для того, щоб зробити програму більш читаною і зрозумілою для подальшого розвитку.

З погляду компілятора, а точніше його частини, що відповідає за лексичний розбір, основна роль пропусків і коментарів - служити роздільниками між лексемами, причому самі роздільники далі відкидаються і на код, що компілює, не впливають. Наприклад, всі наступні приклади оголошення змінній еквівалентні:

```
// Використовуємо пропуск як роздільник.
int x = 3 ;
// тут роздільником є переклад рядка
int
x
=
3
;
// тут розділяємо знаком табуляції
int x = 3 ;
/*
   * Єдиний принципово необхідний
   * роздільник між назвою типу даних
   * int і ім'ям змінної x тут описаний
   * коментарем блокового типу.
```

```
   */
int/**/x=3;
```

Звичайно, лексеми дуже різноманітні, і саме вони визначають багато властивостей мови. Розглянемо всі їх види детальніше.

Види лексем. Нижче перераховані всі види лексем в Java:

- ідентифікатори (identifiers);
- ключові слова (key words);
- літерали (literals);
- роздільники (separators);
- оператори (operators).

Розглянемо їх окремо.

Ідентифікатори. Ідентифікатори - це імена, які даються різним елементам мови для спрощення доступу до них. Імена мають пакети, класи, інтерфейси, поля, методи, аргументи і локальні змінні (всі ці поняття детально розглядаються в наступних лекціях). Ідентифікатори можна записувати символами Unicode, тобто на будь-якій зручній мові. Довжина імені не обмежена.Ідентифікатор складається з букв і цифр. Ім'я не може починатися з цифри. Java-букви, використовувані в ідентифікаторах, включають ASCII-символи `A-Z` (`\u0041-\u005a`), `a-z` (`\u0061-\u007a`), а також знаки підкреслення `_` (ASCII underscore `\u005f`) і долара `$` (`\u0024`). Знак долара використовується тільки при автоматичній генерації коду (щоб виключити випадковий збіг імен), або при використанні яких-небудь старих бібліотек, в яких допускалися імена з цим символом. Java-цифри включають звичайні ASCII-цифри `0-9` (`\u0030-\u0039`).

Для ідентифікаторів не допускаються збіги із зарезервованими словами (це ключові слова, булеві літерали `true` і `false` і null-литерал `null`). Звичайно, якщо 2 ідентифікатори включають різні букви, які однаково виглядають (наприклад, латинська і російська букви `A`), то вони вважаються різними.

У цій лекції вже застосовувалися наступні ідентифікатори:

```
Character, a, b, з, D, x1, x2, Math, sqrt, x
y, i, s, PI, getradius, circle, getabs
calculate, condition, getwidth, getheight
java, lang, String
```

Також допустимими є ідентифікатори:

```
Computer, Color_red _, averylongnameofthemethod
```

Ключові слова.

Ключові слова - це зарезервовані слова, що складаються з ASCII-символов і виконуючі різні завдання мови. Ось їх повний список (48 слів):

```
abstract  double      int         strictfp
boolean      else        interface  super
break     extends     long        switch
byte      final       native      synchronized
case      finally     new         this
catch     float       package     throw
char      for         private     throws
class     goto        protected   transient
const     if          public      try
continue  implements  return      void
default   import      short       volatile
do        instanceof  static      while
```

Ключові слова `goto` і `const` зарезервовані, але не використовуються. Це зроблено для того, щоб компілятор міг правильно відреагувати на їх використання в інших мовах. Навпаки, обидва булеві літерали `true`, `false` і null-литерал `null` часто вважають ключовими словами (можливо, тому, що багато засобів розробки підсвічують їх таким же чином), проте це саме літерали.

Літерали. Літерали дозволяють задати в програмі значення для числових, символьних і строкових виразів, а також null-литералов. Всього у Java визначено **6 видів** літералів:

- цілочисельний (integer);
- дріб (floating-point);
- булевий (boolean);
- символьний (character);
- строковий (string);
- null-литерал (null-literal).

Розглянемо їх окремо.

Цілочисельні літерали дозволяють задавати цілочисельні значення в десятковому, вісімковому і шістнадцятковому вигляді. Десятковий формат традиційний і нічим не відрізняється від правил, прийнятих в інших мовах. Значення у вісімковому вигляді починаються з нуля, і, звичайно, використання цифр `8` і `9` заборонено. Запис шістнадцяткових чисел починається з `0x` або `0x` (цифра 0 і латинська ASCII-буква `X` в довільному регістрі). Таким чином, нуль можна записати трьома різними способами:

```
0
00
0x0
```

Як завжди, для запису цифр 10-15 в шістнадцятковому форматі використовуються букви A, B, C, D, E, F, прописні або рядкові. Приклади таких літералів:

```
0xabcdef, 0xcafe, 0xdec
```

Типи даних розглядаються нижче, проте тут необхідно згадати два цілочисельні типи int і long довжиною 4 і 8 байт, відповідно (або 32 і 64 бита, відповідно). Обидва ці типи знакові, тобто тип int зберігає значення від -2^{31} до 231-1, або від -2.147.483.648 до 2.147.483.647. По замовчуванні цілочисельний літерал має тип int, а значить, в програмі допустимо використовувати літерали тільки від 0 до 2147483648, інакше виникне помилка компіляції. При цьому літерал 2147483648 можна використовувати тільки як аргумент унарного оператору - :

```
int x = -2147483648;  \ вірно
int y = 5-2147483648; \ тут виникне
                      \ помилка компіляції
```

Відповідно, допустимі літерали у вісімковому записі повинні бути від 00 до 017777777777 (=231-1), з унарним оператором - допустимо також -020000000000 (= -231). Аналогічно для шістнадцяткового формату - от0x0 до 0x7fffffff (=231-1), а також -0x80000000 (= -231).

Тип long має довжину 64 бита, а значить, дозволяє зберігати значення -263 до 263-1. Щоб ввести такий літерал, необхідно в кінці поставити латинську букву L або l, тоді все значення трактуватиметься як long. Аналогічно можна виписати максимальні допустимі значення для них:

```
9223372036854775807l
0777777777777777777777l
0x7fffffffffffffffl
// найбільші від'ємні значення:
-9223372036854775808L
-01000000000000000000000L
-0x8000000000000000L
```

Інші приклади цілочисельних літералів типу long:

```
0l, 123l, 0xc0b0l
```

Дробовими літералами є числа з плаваючою десятковою крапкою. Правила запису таких чисел такі ж, як і в більшості сучасних мов програмування.

Приклади:

```
3.14
2.
.5
7e10
3.1E-20
```

Таким чином, дробовий літерал складається з наступних складових частин:

- ціла частина;
- десяткова крапка (використовується ASCII-символ крапка);
- дробова частина;
- показник ступеня (складається з латинської ASCII-букви `E` в довільному регістрі і цілого числа з опціональним знаком `+` або `-`);
- закінчення-покажчик типу.

Ціла і дробова частини записуються десятковими цифрами, а покажчик типу (аналог покажчика `L` або `l` для цілочисельних літералів типу `long`) має два можливі значення - латинська ASCII-буква `D` (для типу `double`) або `F` (для типу `float`) в довільному регістрі. Вони будуть детально розглянуті нижче.

Необхідними частинами є:

- хоч би одна цифра в цілій або дробовій частині;
- десяткова крапка або показник ступеня, або покажчик типу.

Решта всіх частин необов'язкових. Таким чином, "мінімальні" дробові літерали можуть бути записані, наприклад, так:

```
1.
.1
1e1
1f
```

У Java є два дробові типи, згадані вище, - `float` і `double`. Їх довжина - 4 і 8 байт або 32 і 64 бита, відповідно. Дробовий літерал має тип `float`, якщо він закінчується на латинську букву `F` в довільному регістрі. Інакше він розглядається як значення типу `double` і може включати закінчення `Dілі d`, як ознака типу `double` (використовується тільки для наочності).

```
// float-литералы:
1f, 3.14f, 0f, 1e+5f
```

```
// double-литералы:
0., 3.14d, 1e-4, 31.34e45d
```

У Java дробові числа 32-бітового типу `float` і 64-бітового типу `double` зберігаються в пам'яті в бінарному вигляді у форматі, стандартизованому специфікацією IEEE 754 (повна назва - IEEE Standard for Binary Floating-point Arithmetic, Ansi/ieee Standard 754-1985 (IEEE, New York)). У цій специфікації описані не тільки кінцеві дробові величини, але і ще декілька особливих значень, а саме:

- позитивна і негативна нескінченність (positive/negative infinity);
- значення "Не число", Not-a-number, скорочено NAN;
- позитивний і від'ємний нулі.

Для цих значень немає спеціальних позначень. Щоб отримати такі величини, необхідно або провести арифметичну операцію (наприклад, результатом ділення нуль на нуль `0.0/0.0` є `NAN`), або звернутися до констант в класах `Float` і `Double`, а саме `Positive_infinity`, `Negative_infinity` і `NAN`. Детальніше робота з цими особливими значеннями розглядається в наступній лекції.

Типи даних накладають обмеження на можливі значення літералів, як і для цілочисельних типів. Максимально позитивне кінцеве значення дробового літерала:

- для `float`: 3.40282347e+38f
- для `double`: 1.79769313486231570e+308

Крім того, для дробових величин стає важливим ще одне граничне значення - мінімальне позитивне ненульове значення:

- для `float`: 1.40239846e-45f
- для `double`: 4.94065645841246544e-324

Спроба вказати літерал з дуже великим абсолютним значенням (наприклад, 1e40f) приведе до помилки компіляції. Така величина повинна представлятися нескінченністю. Аналогічно, вказівка літерала з дуже малим ненульовим значенням (наприклад, 1e-350) також приводить до помилки. Це значення повинне бути закруглене до нуля. Проте якщо округлення приводить не до нуля, то компілятор проведе його самого:

```
// помилка, вираз повинен бути закруглене до 0
0.00000000000000000000000000000000000000001f
// помилки немає, компілятор сам округляє до 1
1.00000000000000000000000000000000000000001f
```

Стандартних можливостей вводити дробові значення не в десятковій системі в Java немає, проте класи `Float` idouble надають багато допоміжних методів, у тому числі і для такого завдання.

Логічні літерали мають два можливі значення - `true` і `false`. Ці два зарезервовані слова не є ключовими, але також не можуть використовуватися як ідентифікатор.

Символьні літерали описують один символ з набору Unicode, ув'язнений в одиночні лапки, або апострофи (ASCII-символ single quote `\u0027`). Наприклад:

```
'a' // латинська буква a

' ' // пропуск

'k' // грецька буква каппа
```

Також допускається спеціальний запис для опису символу через його код (див. тему "Кодування"). Приклади:

```
'\u0041' // латинська буква A

'\u0410' // російська буква A

'\u0391' // грецька буква A
```

Символьний літерал повинен містити строго один символ, або спеціальну послідовність, що починається з `\`. Для запису спеціальних символів (що не відображаються і службових, таких як `"`, `'`, `\`) використовуються наступні позначення:

```
\b  \u0008  backspace BS - забій

\t  \u0009  horizontal tab HT - табуляція

\n  \u000a  linefeed LF - кінець рядка

\f  \u000c  form feed FF - кінець сторінки

\r  \u000d  carriage return CR -

      повернення каретки

\"  \u0022  double quote " - подвійна лапка

\'  \u0027  single quote ' - одинарна лапка

\  \u005c  backslash \ - зворотна коса межа

\ шістнадцятиричний код  від \u0000 до \u00ff символу

                  у шістнадцятковому форматі.
```

Перша колонка описує стандартні позначення спеціальних символів, використовувані в Java-программах. Друга колонка представляє їх в стандартному виді Unicode-символов. Третя колонка містить англійські і російські описи. Використання \ у комбінації з іншими символами приведе до помилки компіляції.

Підтримка введення символів через вісімковий код забезпечується для сумісності з C. Наприклад:

```
'\101' // Еквівалентно '\u0041'
```

Проте таким чином можна задати лише символи \u0000 до \u00ff (тобто з кодом від 0 до 255), тому Unicode-наслідування передбачуваний.

Оскільки обробка Unicode- наслідування (\uhhhh) проводиться раніше лексичного аналізу, то наступний приклад є помилкою:

```
'\u000a' // символ кінця рядка
```

Компілятор спочатку перетворить \u000a у символ кінця рядка і лапки опиняться на різних рядках коди, що є помилкою. Необхідно використовувати спеціальну послідовність:

```
'\n' // правильне позначення кінця рядка
```

Аналогічно і для символу \u000d (повернення каретки) необхідно використовувати позначення \r.

Спеціальні символи можна використовувати у складі як символьних, так і строкових літералів. **Строкові літерали** складаються з набору символів і записуються в подвійних лапках. Довжина може бути нульовою або скільки завгодно великою. Будь-який символ може бути представлений спеціальною послідовністю, що починається з \ (див. "Символьні літерали").

```
"" // літерал нульової довжини

"\"" //літерал, що складається з одного символу "

"Простий текст" //літерал довжини 13
```

Строковий літерал не можна розбивати на декілька рядків в коді програми. Якщо потрібне текстове значення, що складається з декількох рядків, то необхідно скористатися спеціальними символами \n і/або \r. Якщо ж текст просто дуже довгий, щоб уміщатися на одному рядку коду, можна використовувати оператора конкатенації рядків +. Приклади строкових літералів:

```
// вираз-константа, складений з двох
// літералів
"Довгий текст " +
"з перенесенням"
/*
   * Строковий літерал, що містить текст
   * з двох рядків:
   * Hello, world!
 * Hello!
   */
"Hello, world!\r\nHello!"
```

На строкові літерали розповсюджуються ті ж правила, що і на символьних відносно використання символів нового рядка `\u000a` і\u000d.

Кожен строковий літерал є екземпляром класу `String`. Це визначає деякі незвичайні властивості строкових літералів, які будуть розглянуті в наступній лекції.

Null-литерал може набувати всього одного значення: `null`. Це літерал посилального типу, причому це посилання нікуди не посилається, об'єкт відсутній. Зрозуміло, його можна застосовувати до посилань будь-якого об'єктного типу даних. Типи даних детально розглядаються в наступній лекції.

Роздільники - це спеціальні символи, які використовуються в службових цілях мови. Призначення кожне з них буде розглянуто по ходу викладу курсу. Ось їх повний список:

```
( ) [ ] { } ; .,
```

Оператори використовуються в різних операціях - арифметичних, логічних, бітових, операціях порівняння і привласнення. Наступні 37 лексем (всі складаються тільки з ASCII-символов) є операторами мови Java:

```
=   >   <   !   ~   ?   :
==  <=  >=  !=  &&  ||  ++    --
```

```
+   -   *   /   &   |   ^   %   <<   >>   >>>

+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

Більшість з них цілком очевидні і добре відомі з інших мов програмування, проте деякі нюанси в роботі з операторами в Java все ж таки присутні, тому в кінці лекції приводяться короткі коментарі до них.

Приклад програми

На закінчення для прикладу приведемо просту програму (традиційне Hello, world!), а потім класифікуємо і підрахуємо використовувані лексеми:

```
public class Demo {
/**
   * Основний метод, з якого починається
 * виконання будь-якої Java програми.
   */
     public static void main (String args[])
     {
        System.out.println("Hello, world!");
  }
}
```

Отже, в приведеній програмі є один коментар розробника, 7 ідентифікаторів, 5 ключових слів, 1 строковий літерал, 13 роздільників і жодного оператора. Цей текст можна зберегти у файлі Demo.java, скомпілювати і запустити. Результатом роботи буде, як очевидно:

```
Hello, world!
```

Доповнення. Робота з операторами. Розглянемо деякі деталі використання операторів в Java. Тут будуть описані подробиці, що відносяться до роботи самих операторів. У наступній лекції детально розглядаються особливості, що виникають при використанні різних типів даних (наприклад, значення операції`1/2` дорівнює `0`, а `1/2.` рівне `0.5`).

Оператори привласнення і порівняння

По-перше, звичайно ж, розрізняються оператор присвоєння `=` і оператор порівняння `==`.

```
x = 1; // присвоювання змінною x значення 1

x == 1 // порівнюємо значення змінної x з

       // одиницею
```

Оператор порівняння завжди повертає булеве значення true або false. Оператор привласнення повертає значення правого операнда. Тому звичайна друкарська помилка, коли ці оператори плутають:

```
// приклад викличе помилку компілятора

if (x=0) { // тут повинен застосовуватися оператор

           // порівняння ==

...

}
```

у Java легко усувається. Оскільки вираз `x=0` має числове значення 0, а не булеве (і тим більше не сприймається як завжди істинне), то компілятор повідомляє про помилку (необхідно писати x==0).

Умова "не рівна" записується як `!=`. Наприклад:

```
if (x!=0) {

   float f = 1./x;

}
```

Поєднання якого-небудь оператора з оператором привласнення `=` (див. нижній рядок в повному переліку в розділі "Оператори") використовується при зміні значення змінної. Наприклад, наступні два рядки еквівалентні:

```
x = x + 1;

x += 1;
```

Арифметичні операції. Разом з чотирма звичайними арифметичними операціями `+`, `- *`, `/`, існує оператор отримання залишку від ділення `%`, який може бути застосований як до цілочисельних аргументів, так і до дробів.

Робота з цілочисельними аргументами підкоряється простим правилам. Якщо ділиться значення `a` на значення `b`, то вираз `(a/b)*b+(a%b)` повинно в точності дорівнювати `a`. Тут, звичайно, оператор ділення цілих чисел `/` завжди повертає ціле число. Наприклад:

```
9/5 повертає 1
```

```
9/(-5) повертає -1
(-9)/5 повертає -1
(-9)/(-5) повертає 1
```

Залишок може бути позитивним, тільки якщо ділене було додатнім. Відповідно, залишок може бути від'ємним тільки у разі від'ємного діленого.

```
9%5 повертає 4
9%(-5) повертає 4
(-9)%5 повертає -4
(-9)%(-5) повертає -4
```

Спроба отримати залишок від ділення на `0` приводить до помилки.

Ділення із залишком для дробових чисел може бути проведене по двох різних алгоритмах. Один з них повторює правила для цілих чисел, і саме він представлений оператором `%`. Якщо в розглянутому прикладі ділення `9` на `5` перейти до дробових чисел, значення залишку у всіх варіантах не зміниться (воно буде також дробом, звичайно).

```
9.0%5.0 повертає 4.0
9.0%(-5.0) повертає 4.0
(-9.0)%5.0 повертає -4.0
(-9.0)%(-5.0) повертає -4.0
```

Проте стандарт IEEE 754 визначає інші правила. Такий спосіб представлений методом стандартного классamath. IEEE remainder(`double f1, double f2`). Результат цього методу - значення, яке рівне `f1-f2*n`, де `n` - ціле число, найближче до значення `f1/f2`, а якщо два цілі числа однаково близькі до цього відношення, то вибирається парне. За цим правилом значення залишку буде іншим:

```
Math.IEEEremainder(9.0, 5.0) повертає -1.0
Math.IEEEremainder(9.0, -5.0) повертає -1.0
Math.IEEEremainder(-9.0, 5.0) повертає 1.0
Math.IEEEremainder(-9.0, -5.0) повертає 1.0
```

Унарні оператори інкрементації `++` і декрементації `--`, як завжди, можна використовувати як справа, так і зліва.

```
int x=1;
int y=++x;
```

В даному прикладі оператор `++` стоїть перед змінною `x`, це означає, що спочатку відбудеться інкрементація, а потім значення `x` буде використано для ініціалізації `y`. В результаті після виконання цих рядків значення `xi y` дорівнюватимуть `2.`

```
int x=1;
```

```
int y=x++;
```

А в даному прикладі спочатку значення `x` буде використано для ініціалізації `y`, і лише потім відбудеться інкрементация. В результаті значення `x` дорівнюватиме `2`, а `y` дорівнюватиме `1`.

Логічні оператори "і" і "або" (`&` і `|`) можна використовувати в двох варіантах. Це пов'язано з тим, що, як легко переконатися, для кожного оператора можливі випадки, коли значення першого операнда відразу визначає значення всього логічного виразу. Якщо другим операндом є значення деякої функції, то з'являється вибір - викликати її чи ні, причому це рішення може позначитися як на швидкості, так і на функціональності програми.

Перший варіант операторів (`&`, `|`) завжди обчислює обидва операнди, другий же - (`&&`, `||`) не продовжуватиме обчислення, якщо значення виразу вже очевидно. Наприклад:

```
int x=1;
(x>0) | calculate(x)  // в такому виразі
                      // відбудеться виклик
                      // calculate
(x>0) || calculate(x) // а в цьому - ні
```

Логічний оператор заперечення "не" записується як `!` і, звичайно, має тільки один варіант використання. Цього оператора міняє булеве значення на протилежне.

```
int x=1;
x>0    // вираз істинний
!(x>0) // вираз помилковий
```

Оператор з умовою `? :` складається з трьох частин - умови і двох виразів. Спочатку обчислюється умова (булевий вираз), а на підставі результату значення всього оператора визначається першим виразом у разі отримання істини і другим - якщо умова помилкова. Наприклад, так можна обчислити модуль числа `x`:

```
x>0 ? x : -x
```

Бітові операції. Перш ніж переходити до бітових операцій, необхідно уточнити, яким саме образом цілі числа представляються в двійковому вигляді. Звичайно, для ненегативних величин це практично очевидно:

```
0   0
1   1
2   10
3   11
4   100
```

```
5   101
```

і так далі. Проте як представляються від'ємні числа? По-перше, вводять поняття знакового біта. Перший біт починає відповідати за знак, а саме `0` означає позитивне число, `1` - від'ємне. Але не слід думати, що решта біт залишається незмінними. Наприклад, якщо розглянути 8-бітове уявлення:

```
-1   10000001   // це НЕВІРНО!
-2   10000010   // це НЕВІРНО!
-3   10000011   // це НЕВІРНО!
```

Такий підхід невірний! Зокрема, ми отримуємо відразу два представлення нуля - `00000000` і `100000000`, що нераціонально. Правильний алгоритм можна уявити собі так. Щоб набути значення `-1`, треба від `0 відняти` 1:

```
  00000000
- 00000001
------------
- 11111111
```

Отже `-1` у двійковому вигляді представляється як `11111111`. Продовжуємо застосовувати той же алгоритм (віднімаємо 1):

```
0 00000000
-1 11111111
-2 11111110
-3 11111101
```

і так далі до значення `10000000`, яке є найбільшим по модулю негативним числом. Для 8-бітового уявлення найбільше позитивне число `01111111 (=127)`, а найменше отріцательноє10000000 (=-128). Оскільки всього 8 біт визначає `28=256` значень, причому одне з них відводиться для нуля, то стає ясно, чому найбільші по модулю позитивні і негативні значення розрізняються на одиницю, а не співпадають.

Як відомо, бітові операції "і", "або", що "виключає або" приймають два аргументи і виконують логічну дію попарно над відповідними бітами аргументів. При цьому використовуються ті ж позначення, що і для логічних операторів, але, звичайно, тільки в першому (одиничному) варіанті. Наприклад, обчислимо вираз `5&6`:

```
  00000101
& 00000110
-------------
  00000100
```

```
// число 5 в двійковому вигляді
```

```
// число 6 в двійковому вигляді


//виконали операцію "і" попарно над бітами
// у кожній позиції
```

Тобто вираз 5&6 рівне 4.

Виключення складає лише оператора "не" або "NOT", який для побітових операцій записується як `~` (для логічних було `!`). Цього оператора міняє кожен біт в числі на протилежний. Наприклад `~(-1)=0`. Можна легко встановити загальне правило для отримання бітового представлення негативних чисел:

Якщо `n` - ціле позитивне число, то `-n` в бітовому уявленні дорівнює `~(n-1)`.

Нарешті, залишилося розглянути лише оператори побітового зрушення. У Java є один оператор зрушення вліво і два варіанти зрушення управо. Така відмінність пов'язана з наявністю знакового біта.

При зрушенні вліво оператором `<<` всі біти числа зміщуються на вказану кількість позицій вліво, причому позиції, що звільнилися справа, заповнюються нулями. Ця операція аналогічна множенню на `2n` і діє цілком передбачено, як при додатних, так і при від'ємних аргументах.

Розглянемо приклади застосування операторів зрушення для значень типу `int`, тобто 32-бітових чисел. Хай додатним аргументом буде число `20`, а від'ємним `-21`.

```
// Зрушення вліво для позитивного числа 20
20 << 00 = 00000000000000000000000000010100 = 20
20 << 01 = 00000000000000000000000000101000 = 40
20 << 02 = 00000000000000000000000001010000 = 80
20 << 03 = 00000000000000000000000010100000 = 160
20 << 04 = 00000000000000000000000101000000 = 320
...
20 << 25 = 00101000000000000000000000000000 = 671088640
20 << 26 = 01010000000000000000000000000000 = 1342177280
20 << 27 = 10100000000000000000000000000000 = -1610612736
20 << 28 = 01000000000000000000000000000000 = 1073741824
20 << 29 = 10000000000000000000000000000000 = -2147483648
20 << 30 = 00000000000000000000000000000000 = 0
20 << 31 = 00000000000000000000000000000000 = 0
// Зрушення вліво для негативного числа -21
-21 << 00 = 11111111111111111111111111101011 = -21
-21 << 01 = 11111111111111111111111111010110 = -42
-21 << 02 = 11111111111111111111111110101100 = -84
-21 << 03 = 11111111111111111111111101011000 = -168
-21 << 04 = 11111111111111111111111010110000 = -336
-21 << 05 = 11111111111111111111110101100000 = -672
...
-21 << 25 = 11010110000000000000000000000000 = -704643072
```

```
-21 << 26 = 10101100000000000000000000000000 = -1409286144
-21 << 27 = 01011000000000000000000000000000 = 1476395008
-21 << 28 = 10110000000000000000000000000000 = -1342177280
-21 << 29 = 01100000000000000000000000000000 = 1610612736
-21 << 30 = 11000000000000000000000000000000 = -1073741824
-21 << 31 = 10000000000000000000000000000000 = -2147483648
```

Як видно з прикладі, несподіванки виникають тоді, коли значущі біти починають займати першу позицію і впливати на знак результату.

При зрушенні управо всі біти аргументу зміщуються на вказану кількість позицій, відповідно, управо. Проте встає питання - яким значенням заповнювати позиції, що звільняються, зліва, у тому числі і що відповідає за знак. Є два варіанти. Оператор >> використовує для заповнення цих позицій значення знакового біта, тобто результат завжди має той же знак, що і початкове значення. Інший оператор >>> заповнює їх нулями, тобто результат завжди додатний.

```
// Зрушення управо для позитивного числа 20
// Оператор >>
20 >> 00 = 00000000000000000000000000010100 = 20
20 >> 01 = 00000000000000000000000000001010 = 10
20 >> 02 = 00000000000000000000000000000101 = 5
20 >> 03 = 00000000000000000000000000000010 = 2
20 >> 04 = 00000000000000000000000000000001 = 1
20 >> 05 = 00000000000000000000000000000000 = 0
// Оператор >>>
20 >>> 00 = 00000000000000000000000000010100 = 20
20 >>> 01 = 00000000000000000000000000001010 = 10
20 >>> 02 = 00000000000000000000000000000101 = 5
20 >>> 03 = 00000000000000000000000000000010 = 2
20 >>> 04 = 00000000000000000000000000000001 = 1
20 >>> 05 = 00000000000000000000000000000000 = 0
```

Очевидно, що для додатнього аргументу оператори >> і >>> працюють абсолютно однаково. Подальше зрушення на більшу кількість позицій також даватиме нульовий результат.

```
// Зрушення управо для негативного числа -21
// Оператор >>
-21 >> 00 = 11111111111111111111111111101011 = -21
-21 >> 01 = 11111111111111111111111111110101 = -11
-21 >> 02 = 11111111111111111111111111111010 = -6
-21 >> 03 = 11111111111111111111111111111101 = -3
-21 >> 04 = 11111111111111111111111111111110 = -2
-21 >> 05 = 11111111111111111111111111111111 = -1
// Оператор >>>
-21 >>> 00 = 11111111111111111111111111101011 = -21
-21 >>> 01 = 01111111111111111111111111110101 = 2147483637
-21 >>> 02 = 00111111111111111111111111111010 = 1073741818
```

```
-21 >>> 03 = 00011111111111111111111111111101 = 536870909
-21 >>> 04 = 00001111111111111111111111111110 = 268435454
-21 >>> 05 = 00000111111111111111111111111111 = 134217727
...
-21 >>> 24 = 00000000000000000000000011111111 = 255
-21 >>> 25 = 00000000000000000000000001111111 = 127
-21 >>> 26 = 00000000000000000000000000111111 = 63
-21 >>> 27 = 00000000000000000000000000011111 = 31
-21 >>> 28 = 00000000000000000000000000001111 = 15
-21 >>> 29 = 00000000000000000000000000000111 = 7
-21 >>> 30 = 00000000000000000000000000000011 = 3
-21 >>> 31 = 00000000000000000000000000000001 = 1
```

Як видно з прикладів, ці операції аналогічні діленню на `2n`. Причому, якщо для додатних аргументів із зростанням `n` результат закономірно прагне до `0`, то для від'ємних граничним значенням є `-1`.

Висновок

У цій лекції розглянуто основи лексичного аналізу програм Java. Для їх запису застосовується універсальне кодування Unicode, що дозволяє використовувати будь-яку мову окрім традиційної англійської. Ще раз нагадаємо, що використання Unicode можливо і необхідно в наступних конструкціях:

- коментарі;
- ідентифікатори;
- символьні і строкові літерали.

Останні ж (пропуски, ключові слова, числові, булеві і null-літерали, роздільники і оператори) легко записуються із застосуванням лише ASCII-символов. В той же час будь-який Unicode-символ також можна задати у вигляді спеціальної послідовності ASCII-символів.

Під час аналізу компілятор виділяє з тексту програми <пропуски> (були розглянуті всі символи, які розглядаються як пропуски) і коментарі, які повністю віддаляються з коду (були розглянуті всі види коментарів, зокрема коментар розробника). Пропуски і всі види коментарів служать для розбиття тексту програми на лексеми. Були розглянуті всі види лексем, зокрема всі види літералів.

У доповненні були розглянуті особливості застосування різних операторів.

ТЕМА. *Основні типи даних та операції над ними. Метаклас Class.*

План

1. *Основні типи даних: прості та об'єктні.*
2. *Операції над значеннями різних типів.*
3. *Особливості Метакласу Class.*

Рекомендована література: 3, 4.

Теоретичні відомості

Типи даних визначають основні можливості будь-якої мови. Крім того, Java є мовою, що строго типізується, а тому чітке розуміння моделі типів даних дуже допомагає в написанні якісних програм. Лекція починається з введення поняття змінною, на прикладі якої ілюструються особливості застосування типів в Java. Описується розділення всіх типів на простих і посилальних, операції над значеннями різних типів, а також особливий клас Class, який грає роль метакласу в Java.

Java є мовою, що строго типізується. Це означає, що будь-яка змінна і будь-який вираз мають відомий тип ще на момент компіляції. Таке строге правило дозволяє виявляти багато помилок вже під час компіляції. Компілятор, знайшовши помилку, указує точне місце (рядок) і причину її виникнення, а динамічні "баги" (від англійського bugs) необхідно спочатку виявити за допомогою тестування (що може зажадати значних зусиль), а потім знайти місце в коді, яке їх породило. Тому чітке розуміння моделі типів даних в Java дуже допомагає в написанні якісних програм.

Всі типи даних розділяються на дві групи. Першу складають 8 *простих*, або *примітивних* (від англійського primitive), типів даних. Вони підрозділяються на три підгрупи:

1. *Цілочисельні*
   - Byte
   - Short
   - Int
   - Long
   - char (також є цілочисельним типом)
2. *дробові*
   - float
   - double
8. *булеві*
   - boolean

Другу групу складають *об'єктні*, або *посилальні* (від англійського reference), типи даних. Це все класи, інтерфейси і масиви. У стандартних бібліотеках перших версій Java знаходилося декілька сотів класів і інтерфейсів, зараз їх вже тисячі. Окрім стандартних, написано багато-багато класів і інтерфейси, що складають будь-яку Java-программу.

Ілюструвати логіку роботи з типами даних найпростіше на прикладі змінних.

Змінні

Змінні використовуються в програмі для зберігання даних. Будь-яка змінна має три базові характеристики:

- ім'я;
- тип;
- значення.

Ім'я унікально ідентифікує змінну і дозволяє звертатися до неї в програмі. Тип описує, які величини може зберігати змінна. Значення - поточна величина, що зберігається в змінній на даний момент.

Робота із змінною завжди починається з її *оголошення* (declaration). Звичайно, воно повинне включати ім'я оголошуваної змінної. Як було сказано, в Java будь-яка змінна має строгий тип, який також задається при оголошенні і ніколи не міняється. Значення може бути вказане відразу (це називається ініціалізацією), а в більшості випадків завдання початкової величини можна і відкласти.

Деякі приклади оголошення змінних примітивного типу int з ініціалізаторами і без таких:

```
int a;
int b = 0, з = 3+2;
int d = b+c;
int e = a = 5;
```

З прикладів видно, що ініціалізатором може бути не тільки константа, але і арифметичний вираз. Іноді цей вираз може бути обчислений під час компіляції (таке як 3+2), тоді компілятор відразу записує результат. Іноді ця дія відкладається на момент виконання програми (наприклад, b+c). У останньому випадку декільком змінним привласнюється одне і те ж значення, проте оголошується лише перша з них (у даному прикладі e), останні вже повинні існувати.

Резюмуємо: ***оголошення*** змінних і можлива ініціалізація при оголошенні описуються таким чином. Спочатку указується тип змінної, потім її ім'я і, якщо необхідно, ініціалізатор, який може бути константою або виразом, що обчислюється під час компіляції або виконання програми. Зокрема, можна користуватися вже оголошеними змінними. Далі можна поставити кому і оголосити нову змінну такого самого типу.

Після оголошення змінна може застосовуватися в різних виразах, в яких братиметься її поточне значення. Також у будь-який момент можна змінити значення, використовуючи оператора привласнення, приблизно так само, як це робилося в ініціалізаторах.

Крім того, при оголошенні змінній може бути використане ключове слово final. Його указують перед типом змінної, і тоді її необхідно відразу ініціалізувати і вже більше ніколи не міняти її значення. Таким чином, final-змінні стають чимось на зразок констант, але насправді деякі ініціалізатори можуть обчислюватися тільки під час виконання програми, генеруючи різні значення.

Простий приклад оголошення final- змінної:

```
final double pi=3.1415;
```

Примітивні і посилальні типи даних

Тепер на прикладі змінних можна проілюструвати відмінність між примітивними і *посилальними* типами даних. Розглянемо приклад, коли оголошуються дві змінні одного типу, прирівнюються один одному, а потім значення однієї з них змінюється. Що відбудеться з другою змінною?

Візьмемо простий тип int:

```
int a=5;   // оголосили першу змінну і
           // ініціалізували її
int b=a;   // оголосили другу змінну і
```

```
                // прирівняли її до першої
a=3;        // змінили значення першої
print(b);  // перевіряємо значення другої
```

Тут і далі ми вважаємо, що функція print(...) дозволяє нам деяким (неважливо, яким саме) способом дізнатися значення її аргументу (як правило, для цього використовують функцію із стандартної бібліотеки system.out.println(...), яка виводить значення на системну консоль).

В результаті ми побачимо, що значення змінної b не змінилося, воно залишилося рівним 5. Це означає, що змінні *простого* типу зберігають безпосередньо свої значення і при прирівнюванні двох змінних відбувається копіювання даного значення. Щоб ще раз підкреслити цю особливість, приведемо ще один приклад:

```
byte b=3;
int a=b;
```

У даному прикладі відбувається перетворення типів (воно детально розглядається у відповідній лекції). Для нас зараз важливо констатувати, що змінна b зберігає значення 3 типу byte, а змінна a - значення 3 типу int. Це два різних значення, і в другому рядку при привласненні відбулося копіювання.

Тепер розглянемо *посилальний* тип даних. Змінні таких типів завжди зберігають посилання на деякі об'єкти. Розглянемо для прикладу клас, що описує крапку на координатній площині з цілочисельними координатами. Опис класу - це окрема тема, але в нашому простому випадку воно тривіальне:

```
class Point {
  int x, y;
}
```

Тепер складемо приклад, аналогічний приведеному вище для int-змінних, вважаючи, що вираз new Point(3,5) створює новий об'єкт-крапку з координатами (3,5).

```
Point p1 = new Point(3,5);
Point p2=p1;
p1.x=7;
print(p2.x);
```

У третьому рядку ми змінили горизонтальну координату точки, на яку посилалася змінна p1, і тепер нас цікавить, як це позначилося на крапці, на яку посилається змінна p2. Провівши такий експеримент, можна переконатися, що на цей раз ми побачимо оновлене значення. Тобто об'єктні змінні після прирівнювання залишаються "зв'язаними" один з одним, зміни однією позначаються на іншій.

Таким чином, примітивні змінні є дійсними сховищами даних. Кожна змінна має значення, не залежне від останніх. Посилальні ж змінні зберігають лише посилання на об'єкти, причому різні змінні можуть посилатися на один і той же об'єкт, як це було в нашому прикладі. В цьому випадку їх можна порівняти із спостерігачами, які з різних позицій дивляться на один і той же об'єкт і однаково бачать всі зміни, що відбуваються з ним. Якщо ж один спостерігач змінить об'єкт спостереження, то він перестає бачити і зміни, що відбуваються з колишнім об'єктом:

```
Point p1 = new Point(3,5);
Point p2=p1;
p1 = new Point(7,9);
print(p2.x);
```

В даному прикладі ми отримаємо 3, тобто після третього рядка змінні p1 і p2 посилаються на різні об'єкти і тому мають різні значення.

Тепер легко зрозуміти сенс літерала null. Таке значення може прийняти змінна будь-якого посилального типу. Це означає, що її посилання нікуди не указує, об'єкт відсутній. Відповідно, будь-яка спроба звернутися до об'єкту через таку змінну (наприклад, викликати метод або узяти значення поля) приведе до помилки.

Також значення null можна передати як будь-який об'єктний аргумент при виклику функцій (хоча на практиці багато методів вважають таке значення некоректним).

Пам'ять в Java з погляду програміста представляється не нулями і одиницями або набором байтів, а як якийсь віртуальний простір, в якому існують об'єкти. І доступ до пам'яті здійснюється не за фізичною адресою або покажчиком, а лише через посилання на об'єкти. Посилання повертається при створенні об'єкту і далі може бути збережене в змінній, передана як аргумент і так далі Як вже говорилося, допускається наявність декількох посиланнях на один об'єкт. Можлива і протилежна ситуація - коли на якийсь об'єкт не існує жодного посилання. Такий об'єкт вже недоступний програмі і є "сміттям", тобто марно займає апаратні ресурси. Для їх звільнення не потрібні ніяких зусиль. До складу будь-якої віртуальної машини обов'язково входить автоматичний складальник сміття garbage collector - фоновий процес, який якраз і займається знищенням непотрібних об'єктів.

Дуже важливо пам'ятати, що об'єктна змінна, на відміну від примітивної, може мати значення іншого типу, не співпадаючого з типом змінної. Наприклад, якщо тип змінної - якийсь клас, то змінна може посилатися на об'єкт, породжений від спадкоємця цього класу. Всі випадки подібного неспівпадання будуть розглянуті в наступних розділах курсу.

Тепер розглянемо *примітивні* і *посилальні* типи даних детальніше.

Примітивні типи

Як вже говорилося, існує 8 *простих типів даних*, які діляться на *цілочисельні* (*integer*), *дроби*(*floating-point*) і *булеві* (*boolean*).

Цілочисельні типи

Цілочисельні типи - це byte, short, int, long, також до них відносять і char. Перші чотири типи мають довжину 1, 2, 4 і 8 байт відповідно, довжина char - 2 байти, це безпосередньо виходить з того, що всі символи Java описуються стандартом Unicode. Довжини типів приведені тільки для оцінки областей значення. Як вже мовилося, пам'ять в Java представляється віртуальною і обчислити, скільки фізичних ресурсів займе та або інша змінна, так прямолінійно не вийде.

4 основних типи є знаковими. char доданий до цілочисельних типів даних, оскільки з погляду JVM символ і його код - поняття взаємооднозначні. Звичайно, код символу завжди додатний, тому char - єдиний беззнаковий тип. Ініціалізувати його можна як символьним, так і цілочисельним літералом. Зрештою char - повноцінний числовий тип даних, який може брати участь, наприклад, в арифметичних діях, операціях порівняння і тому подібне У таблиці зведені дані по всіх розібраних типах:

Таблиця 4.1. Цілочисельні типи даних.

Назва типу	Довжина(байти)	Область значень
byte	1	-128 .. 127
short	2	-32.768 .. 32.767

int	4	-2.147.483.648 .. 2.147.483.647
long	8	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 (приблизно 10^{19})
char	2	'\u0000' .. '\uffff', или 0 .. 65.535

Звернете увагу, що int вміщає приблизно 2 мільярди, а тому підходить у багатьох випадках, коли не потрібні надвеликі числа. Щоб уявити собі розміри типу long, вкажемо, що саме він використовується в Java для відліку часу. Як і в багатьох мовах, час відлічується від 1 січня 1970 року в мілісекундах. Так от, місткість long дозволяє відлічувати час впродовж мільйонів століть(!), причому як в майбутнє, так і в минуле.

Чому були виділені саме ці два типи, int і long? Річ у тому, що цілочисельні літерали мають тип int по замовчуванні, або тип long, якщо коштує буква L або l. Саме тому коректним літералом вважається тільки таке число, яке укладається в 4 або 8 байт, відповідно. Інакше компілятор визнає це помилкою. Таким чином, наступні літерали є коректними:

```
1
-2147483648
2147483648l
0l
111111111111111111l
```

Над цілочисельними аргументами можна проводити наступні операції:

- операції порівняння (повертають булеве значення)
- <, <= >, >=
- ==, !=
- числові операції (повертають числове значення)
- унарні операції + і –
- арифметичні операції +, - *, /, %
- операції інкремента і декремента (у префіксній і постфіксній формі): ++ і –
- операції бітового зрушення <<, >>, >>>
- бітові операції ~, &, |, ^
- оператор з умовою ? :
- оператор приведення типів
- оператор конкатенації з рядком +

Оператори порівняння цілком очевидні і окремо ми їх розглядати не будемо. Їх результат завжди *булевого типу (*true або false).

Робота числових операторів також зрозуміла, до того ж пояснювалася в попередній лекції. Єдине уточнення можна зробити щодо операторів + і -, які можуть бути як бінарними (мати два операнди), так і унарними (мати один операнд). Бінарні операнди є операторами додавання і віднімання, відповідно. Унарний оператор + повертає значення, рівне аргументу (+x завжди рівне x). Унарний оператор -, застосований до значення x,

повертає результат, рівний 0-x. Несподіваний ефект має місце в тому випадку, якщо аргумент дорівнює найменшому можливому значенню примітивного типу.

```
int x=-2147483648;   // найменше можливе
int y=-x;            // значення типу int
```

Тепер значення змінної y насправді рівне не 2147483648, оскільки таке число не укладається в область значень типу int, а в точності дорівнює значенню x! Іншими словами, в даному прикладі вираз -x==x є істинним!

Річ у тому, що якщо при виконанні числових операцій над цілими числами виникає переповнювання і результат не може бути збережений в даному примітивному типі, то Java не створює ніяких помилок. Замість цього всі старші біти, які перевищують місткість типу, просто відкидаються. Це може привести не тільки до втрати точної абсолютної величини результату, але навіть до спотворення його знаку, якщо на місці знакового біта опиниться протилежне значення.

```
int x= 300000;
print(x*x);
```

Результатом такого прикладу буде:

```
-194313216
```

Повертаючись до інвертування числа -2147483648, ми бачимо, що математичний результат рівний в точності+231, або, в двійковому форматі, 1000 0000 0000 0000 0000 0000 0000 0000 (одиниця і 31 нуль). Але тип int сприймає першу одиницю як знаковий біт, і результат виходить рівним -2147483648.

Таким чином, явне виписування в коді літералів, які дуже великі для використовуваних типів, приводить до помилки компіляції (див. лекцію 3). Якщо ж переповнювання виникає в результаті виконання операції, "зайві" біти просто відкидаються.

Підкреслимо, що вираз типу -5 не є цілочисельним літералом. Насправді воно складається з літерала 5 і оператора -. Нагадаємо, що деякі літерали (наприклад, 2147483648) можуть зустрічатися тільки у поєднанні з унарним оператором -.

Крім того, числові операції в Java володіють ще однією особливістю. Хоча цілочисельні типи мають довжину 8, 16, 32 і 64 бита, обчислення проводяться тільки з 32-х і 64-х бітовою точністю. А це означає, що перед обчисленнями може потрібно перетворити тип одного або декількох операндів.

Якщо хоч би один аргумент операції має тип long, то всі аргументи приводяться до цього типу і результат операції також буде типу long. Обчислення буде проведено з точністю в 64 бита, а більш старші біти, якщо такі з'являються в результаті, відкидаються.

Якщо ж аргументів типу long немає, то обчислення проводиться з точністю в 32 бита, і всі аргументи перетворяться в int (це відноситься до byte, short, char). Результат також має тип int. Всі біти старше 32-го ігноруються.

Ніякого способу дізнатися, чи відбулося переповнювання, не немає. Розширимо розглянутий приклад:

```
int i=300000;
print(i*i);   // множення з точністю 32 бита
long m=i;
```

```
print(m*m);   // множення з точністю 64 бита
print(1/(m-i)); // спробуємо отримати різницю
        // значень int і long
```

Результатом такого прикладу буде:

```
-194313216
90000000000
```

Потім ми отримаємо помилку ділення на нуль, оскільки змінні i і m хоч і різних типів, але зберігають однакове математичне значення і їх різниця дорівнює нулю. Перше множення проводилося з точністю в 32 бита, більш старші біти були відкинуті. Друге - з точністю в 64 бита, відповідь не спотворилася.

Питання приведення типів, і зокрема спеціальний оператор для такої дії, детально розглядається в наступних лекціях. Проте тут хотілося б відзначити декілька прикладів, які не такі очевидні і можуть створити проблеми при написанні програм. По-перше, підкреслимо, що результатом операції з цілочисельними аргументами завжди є ціле число. А значить, в наступному прикладі

```
double x = 1/2;
```

Змінній x буде привласнено значення 0, а не 0.5, як можна було б чекати. Детально операції з дробовими аргументами розглядаються нижчим, але щоб набути значення 0.5, досить написати 1./2 (тепер перший аргумент дріб і результат не буде закруглений).

Як уже згадувалося, час в Java вимірюється в мілісекундах. Спробуємо обчислити, скільки мілісекунд міститься в тижні і в місяці:

```
print(1000*60*60*24*7);
  // обчислення для тижня
print(1000*60*60*24*30);
  // обчислення для місяця
```

Необхідно перемножити кількість мілісекунд в одній секунді (1000), секунд - в хвилині (60), хвилин - в годині (60), годинника - в дні (24) і днів - в тижні і місяці (7 і 30, відповідно). Отримуємо:

```
604800000
-1702967296
```

Очевидно, в другому обчисленні відбулося переповнювання. Досить зробити останній аргумент величиною типу long:

```
print(1000*60*60*24*30l);
  // обчислення для місяця
```

Отримуємо правильний результат:

```
2592000000
```

Подібні обчислення розумно перекладати на 64-бітову точність не на останній операції, а заздалегідь, щоб уникнути переповнювання.

Зрозуміло, що типи більшої довжини можуть зберігати більший спектр значень, а тому Java не дозволяє привласнити змінної меншого типу значення більшого типу. Наприклад, такі рядки викличуть помилку компіляції:

```
// приклад викличе помилку компіляції
int x=1;
byte b=x;
```

Хоча для програміста і очевидно, що змінна b повинна набути значення 1, що легко укладається в типі byte, проте компілятор не може обчислювати значення змінної x при обробці другого рядка, він знає лише, що її тип - int.

А ось менш очевидний приклад:

```
// приклад викличе помилку компіляції
byte b=1;
byte c=b+1;
```

І тут компілятор не зможе успішно завершити роботу. При операції складання значення змінної b буде перетворено в тип int і таким же буде результат складання, а значить, його не можна так просто привласнити змінній типу byte.

Аналогічно:

```
// приклад викличе помилку компіляції
int x=2;
long y=3;
int z=x+y;
```

Тут результат складання буде вже типу long. Так само некоректна така ініціалізація:

```
// приклад викличе помилку компіляції
byte b=5;
byte c=-b;
```

Навіть унарний оператор "-" проводить обчислення з точністю не менше 32 біти.

Хоча у всіх випадках ініціалізація змінних приводилася тільки для прикладу, а предметом розгляду були числові операції, вкажемо коректний спосіб перетворити тип числового значення:

```
byte b=1;
byte c=(byte) -b;
```

Отже, всі числові оператори повертають результат типу int або long. Проте існує два виключення.

Перше з них - оператори інкрементації і декрементації. Їх дія полягає в збільшенні або відніманні одиниці із значення змінної, після чого результат зберігається в цій змінній і значення всієї операції дорівнює значенню змінної (до або після зміни, залежно від того, є оператор префіксним або постфіксним). А значить, і тип значення співпадає з типом змінної. (Насправді, обчислення все одно проводяться з точністю мінімум 32 бита, проте при привласненні змінній результату його тип знижується.)

```
byte x=5;
byte y1=x++;
 // на момент початку виконання x рівний 5
```

```
byte y2=x--;
  // на момент початку виконання x рівний 6
byte y3=++x;
  // на момент початку виконання x рівний 5
byte y4=--x;
  // на момент початку виконання x рівний 6
print(y1);
print(y2);
print(y3);
print(y4);
```

В результаті отримуємо:

```
5
6
6
5
```

Жодних проблем з привласненням результату операторів ++ і -- змінним типу byte. Завершуючи розгляд цих операторів, приведемо ще один приклад:

```
byte x=-128;
print(-x);
byte y=127;
print(++y);
```

Результатом буде:

```
128
-128
```

Цей приклад ілюструє питання перетворення типів при обчисленнях і випадки переповнювання.

Другим виключенням є оператор з умовою ? :. Якщо другий і третій операнди мають однаковий тип, то і результат операції буде такого ж типу.

```
byte x=2;
byte y=3;
byte z=(x>y)? x : y;
  // вірно, x і y однакового типу
byte abs=(x>0)? x : -x;
  // невірно!
```

Останній рядок невірний, оскільки третій аргумент містить числову операцію, отже, його тип int, а значить, і тип всієї операції буде int, і привласнення некоректне. Навіть якщо другий аргумент має тип byte, а третій - short, значення буде типу int.

Нарешті, розглянемо оператора конкатенації з рядком. Оператор + може приймати як аргумент строкові величини. Якщо одним з аргументів є рядок, а другим - ціле число, то число буде перетворено в текст і рядки об'єднаються.

```
int x=1;
print("x="+x);
```

Результатом буде:

```
x=1
```

Зверніть увагу на наступний приклад:

```
print(1+2+"text");
print("text"+1+2);
```

Його результатом буде:

```
3text
text12
```

Окремо розглянемо роботу з типом char. Значення цього типу можуть повноцінно брати участь в числових операціях:

```
char c1=10;
char c2='a';
  // латинська буква A (\u0041, код 65)
int i=c1+c2-'b';
```

Змінна i набуде значення 9.

Розглянемо наступний приклад:

```
char c='a';
print(c);
print(c+1);
print("c="+c);
print('c'+'='+c);
```

Його результатом буде:

```
A
66
c=a
225
```

У першому випадку в метод print було передано значення типу char, тому відобразився символ. У другому випадку був переданий результат складання, тобто число, і саме число з'явилося на екрані. Далі при складанні з рядком тип char був перетворений в текст у вигляді символу. Нарешті в останньому рядку відбулося складання трьох чисел: 'c' (код 99), '=' (код 61) і змінній з (тобто код 'a' - 65).

Для кожного примітивного типу існують спеціальні допоміжні класи-обгортки (wrapper classes). Для типів byte, short, int, long, char це Byte, Short, Integer, Long, Character. Ці класи містять багато корисних методів для роботи з цілочисельними значеннями.

Наприклад, перетворення з тексту в число. Крім того, є клас Math, який хоч і призначений в основному для роботи з дробовими числами, але також надає деякі можливості і для цілих.

На закінчення підкреслимо, що єдині операції з цілими числами, при яких Java генерує помилки, - це ділення на нуль (оператори / і %).

Дробові типи

Дробові типи - це float і double. Їх довжина - 4 і 8 байт, відповідно. Обидва типи знакові. Нижче в таблиці зведені їх характеристики:

Таблиця 4.2. Дробові типи даних.

Назва типу	Довжина (байти)	Область значень
float	4	3.40282347e+38f ; 1.40239846e-45f
double	8	1.79769313486231570e+308 ; 4.94065645841246544e-324

Для цілочисельних типів область значень задавалася верхньою і нижньою межами, вельми близькими по модулю. Для дробових типів додається ще одне обмеження - наскільки можна наблизитися до нуля, іншими словами - яке найменше позитивне ненульове значення. Таким чином, не можна задати літерал свідомо більший, ніж дозволяє відповідний тип даних, це приведе до помилки ***overflow***. І не можна задати літерал, значення якого по модулю дуже мало для даного типу, компілятор згенерує помилку ***underflow***.

```
// приклад викличе помилку компіляції
float f = 1e40f;
// значення дуже велике, overflow
double d = 1e-350;
// значення дуже мало, underflow
```

Нагадаємо, що якщо в кінці літерала стоїть буква F або f, то літерал розглядається як значення типу float. За замовчуванням дробовий літерал має тип double, за бажанням це можна підкреслити буквою D або d.

Над дробовими аргументами можна проводити наступні операції:

- операції порівняння (повертають булеве значення)
- <, <= >, >=
- ==, !=
- числові операції (повертають числове значення)
- унарні операції + і –
- арифметичні операції +, - *, /, %
- операції інкремента і декремента (у префіксній і постфіксній формі): ++ і –
- оператор з умовою ? :

- оператор приведення типів
- оператор конкатенації з рядком +

Практично всі оператори діють по тих же принципах, які передбачені для цілочисельних операторів (оператор ділення із залишком % розглядався в попередній лекції, а оператори ++ і -- також збільшують або зменшують значення змінної на одиницю). Уточнимо лише, що оператори порівняння коректно працюють і у разі порівняння цілочисельних значень з дробами. Таким чином, в основному необхідно розглянути питання переповнювання і перетворення типів при обчисленнях.

Для дробових обчислень з'являється вже два типи переповнювання - *overflow* і *underflow*. Проте, Java і тут ніяк не повідомляє про виникнення подібних ситуацій. Немає ні помилок, ні інших способів виявити їх. Більш того, навіть ділення на нуль не приводить до некоректної ситуації. А значить, дробові обчислення взагалі не породжують ніяких помилок.

Така свобода пов'язана з наявністю спеціальних значень дробового типу. Вони визначаються специфікацією IEEE 754 і вже перераховувалися в лекції 3:

- позитивна і негативна нескінченність (positive/negative infinity);
- значення "Не число", Not-a-number, скорочено NAN;
- позитивний і негативний нулі.

Всі ці значення представлені як для типу float, так і для double.

Позитивну і негативну нескінченність можна отримати таким чином:

```
1f/0f    // позитивна нескінченність
         // тип float
-1d/0d   // негативна нескінченність
         // тип double
```

Також в класах Float і Double визначені константи Positive_infinity і Negative_infinity. Як видно з прикладу, такі величини виходять при діленні кінцевих величин на нуль.

Значення NAN можна набути, наприклад, в результаті наступних дій:

```
0.0/0.0         // ділення нуль на нуль
(1.0/0.0)*0.0   // множення нескінченності на нуль
```

Ця величина також представлена константами NAN в класах Float і Double.

Величини позитивний і негативний нуль записуються очевидним чином:

```
0.0    // дробовий літерал зі значенням
       // позитивного нуля
+0.0   // унарна операція +, її значення -
       // позитивний нуль
-0.0   // унарна операція -, її значення -
       // негативний нуль
```

Всі дробові значення строго впорядковані. Негативна нескінченність менше будь-якого іншого дробового значення, позитивна - більше. Значення +0.0 і -0.0

вважаються рівними, тобто вираз 0.0==-0.0 істинно, а 0.0>-0.0 - помилково. Проте інші оператори розрізняють їх, наприклад, вираз 1.0/0.0 дає позитивну нескінченність, а 1.0/-0.0 - негативну.

Єдине виключення - значення NAN. Якщо хоч би один з аргументів операції порівняння дорівнює NAN, то результат свідомо буде false (для оператора != відповідно завжди true). Таким чином, єдине значення x, при якому вираз x!=x істинний, - саме NAN.

Повертаємося до питання переповнення в числових операціях. Якщо значення, що набуває, дуже велике по модулю (*overflow*), то результатом буде нескінченність відповідного знаку.

```
print(1e20f*1e20f);
print(-1e200*1e200);
```

В результаті отримуємо:

```
Infinity
-Infinity
```

Якщо результат, навпаки, виходить дуже малий (*underflow*), то він просто округляється до нуля. Так само поступають і у тому випадку, коли кількість десяткових знаків перевищує допустиме:

```
print(1e-40f/1e10f);   // underflow для float
print(-1e-300/1e100);  // underflow для double
float f=1e-6f;
print(f);
f+=0.002f;
print(f);
f+=3;
print(f);
f+=4000;
print(f);
```

Результатом буде:

```
0.0
-0.0

1.0E-6
0.002001
3.002001
4003.002
```

Як видно, в останньому рядку був втрачений 6-й розряд після десяткової крапки.

Інший приклад (із специфікації мови Java):

```
double d = 1e-305 * Math.PI;
print(d);
```

```
for (int i = 0; i < 4; i++)
print(d /= 100000);
```

Результатом буде:

```
3.141592653589793E-305
3.1415926535898E-310
3.141592653E-315
3.142E-320
0.0
```

Таким чином, як і для цілочисельних значень, явне виписування в коді літералів, які дуже великі (*overflow*) або дуже малі (*underflow*) для використовуваних типів, приводить до помилки компіляції (див. лекцію 3). Якщо ж переповнювання виникає в результаті виконання операції, то повертається одне із спеціальних значень.

Тепер перейдемо до перетворення типів. Якщо хоч би один аргумент має тип double, то значення всіх аргументів приводяться до цього типу і результат операції також матиме тип double. Обчислення буде проведено з точністю в 64 бита.

Якщо ж аргументів типу double немає, а хоч би один аргумент має тип float, то всі аргументи приводяться до float, обчислення проводиться з точністю в 32 бита і результат має тип float.

Ці твердження вірні і у випадку, якщо один з аргументів цілочисельний. Якщо хоч би один з аргументів має значення NAN, то і результатом операції буде NAN.

Ще раз розглянемо простий приклад:

```
print(1/2);
print(1/2.);
```

Результатом буде:

```
0
0.5
```

Достатньо одного дробового аргументу, щоб результат операції також мав дробовий тип.

Складніший приклад:

```
int x=3;
int y=5;
print (x/y);
print((double)x/y);
print(1.0*x/y);
```

Результатом буде:

```
0
0.6
0.6
```

Вперше обидва аргументи були цілими, тому в результаті вийшов нуль. Проте оскільки обидва операнди представлено змінними, в даному прикладі не можна просто поставити десяткову крапку і таким чином перевести обчислення в дробовий тип. Необхідно або перетворити один з аргументів (другий вивід на екран), або вставити ще одну фіктивну операцію з дробовим аргументом (останній рядок).

Приведення типів детально розглядається в іншій лекції, проте звернемо тут увагу на декілька моментів.

По-перше, при приведенні дробових значень до цілих типів дробова частина просто відкидається. Наприклад, число3.84 буде перетворене в ціле 3, а -3.84 перетвориться на -3. Для математичного округлення необхідно скористатися методом класу Math.round(.).

По-друге, при приведенні значень int до типу float і при приведенні значень типу long до типу float і double можливості втрати точності, не дивлячись на те, що ці дробові типи вміщають набагато більші числа, чим відповідні цілі. Розглянемо наступний приклад:

```
long l=111111111111L;
float f = l;
l = (long) f;
print(l);
```

Результатом буде:

```
111111110656
```

Тип float не зміг зберегти всі значущі розряди, хоча перетворення від long до float відбулося без спеціального оператора на відміну від зворотного переходу.

Для кожного примітивного типу існують спеціальні допоміжні класи-обгортки (wrapper classes). Для типів float і double це Float і Double. Ці класи містять багато корисних методів для роботи з дробовими значеннями. Наприклад, перетворення з тексту в число.

Крім того, клас Math надає велику кількість методів для операцій над дробовими значеннями, наприклад, одержання квадратного кореня, піднесення до будь-якої степені, тригонометричні та інші. Також в цьому класі визначені константи PI і підстава натурального логарифму E.

Булевий тип

Булевий тип представлений всього одним типом boolean, який може зберігати всього два можливі значення -true і false. Величини саме цього типу виходять в результаті операцій порівняння.

Над булевими аргументами можна проводити наступні операції:

- операції порівняння (повертають булеве значення)
- ==, !=
- логічні операції (повертають булеве значення)
- !
- &, |, ^
- &&, ||
- оператор з умовою ? :
- оператор конкатенації з рядком +

Логічні оператори && і || обговорювалися в попередній лекції. У операторові з умовою ? : першим аргументом може бути тільки значення типу boolean. Також допускається, щоб другий і третій аргументи одночасно мали булевий тип.

Операція конкатенації з рядком перетворює булеву величину на текст "true" або "false" залежно від значення.

Тільки булеві вирази допускаються для управління потоком обчислень, наприклад, як критерій умовного переходу if.

Ніяке число не може бути інтерпретоване як булевий вираз. Якщо передбачається, що ненульове значення еквівалентне істині (по правилах мови 3), то необхідно записати x!=0. Посилальні величини можна перетворювати в boolean виразом ref!=null.

Посилальні типи

Отже, вираз посилального типу має значення або null, або посилання, вказуюче на деякий об’єкт у віртуальній пам’яті JVM.

Об’єкти і правила роботи з ними

Об’єкт (object) - це екземпляр деякого класу, або екземпляр масиву. Масиви детально розглядатимуться у відповідній лекції. Клас - це опис об’єктів однакової структури, і якщо в програмі такий клас використовується, то опис присутній в єдиному екземплярі. Об’єктів цього класу може не бути зовсім, а може бути створено скільки завгодно багато.

Об’єкти завжди створюються з використанням ключового слова new, причому одне слово new породжує строго один об’єкт (або зовсім жодного, якщо відбувається помилка). Після ключового слова вказується ім’я класу, від якого ми збираємося породити об’єкт. Створення об’єкту завжди відбувається через виклик один з конструкторів класу (їх може бути декілька), тому на закінчення ставляться дужки, в яких перераховані значення аргументів, переданих вибраному конструктору. У приведених вище прикладах, коли створювалися об’єкти типу Point, вираз new Point (3,5) означав звернення до конструктора класу point, у якого є два аргументи типу int. До речі, обов’язкове оголошення такого

конструктору в спрощеному оголошенні класу було відсутнє. Оголошення класів розглядається в наступних лекціях, проте приведемо правильне визначення Point:

```
class Point {
  int x, y;
  /**
    * Конструктор приймає 2 аргументи
    * якими ініціалізував поля об'єкту.
    */
  Point (int newx, int newy){
    x=newx;
    y=newy;
  }
}
```

Якщо конструктор відпрацював успішно, то вираз new повертає посилання на створений об'єкт. Це посилання можна зберегти в змінній, передати як аргумент в який-небудь метод або використовувати іншим способом. JVM завжди займається підрахунком посилань, що зберігаються, на кожен об'єкт. Як тільки виявляється, що посилань більше немає, такий об'єкт призначається для знищення складальником сміття (garbage collector). Відновити посилання на такий "втрачений" об'єкт неможливо.

```
Point p=new Point(1,2);
  // Створили об'єкт, отримали на нього посилання
Point p1=p;
  // тепер є 2 посилання на крапку (1,2)
p=new Point(3,4);
  // залишилося одне посилання на крапку (1,2)
p1=null;
```

Посилань на об'єкт-крапку (1,2) більше немає, доступ до нього загублений і він незабаром буде знищений складальником сміття.

Будь-який об'єкт породжується тільки із застосуванням ключового слова new. Єдине виключення - екземпляри класу String. Записуючи будь-який строковий літерал, ми

автоматично породжуємо об’єкт цього класу. Оператор конкатенації +, результатом якого є рядок, також неявно породжує об’єкти без використання ключового слова new.

Розглянемо приклад:

```
"abc"+"def"
```

При виконанні цього виразу буде створено три об’єкти класу String. Два об’єкти породжуються строковими літералами, третій представлятиме результат конкатенації.

Операція створення об’єкту - одна з самих ресурсоємних в Java. Тому слід уникати непотрібних породжень. Оскільки при роботі з рядками їх може створюватися досить багато, компілятор, як правило, намагається оптимізувати такі вирази. У розглянутому прикладі, оскільки всі операнди є константами часу компіляції, компілятор сам здійснить конкатенацію і вставить в код вже результат, скоротивши таким чином кількість створюваних об’єктів до одного.

Крім того, у версії Java 1.1 була введена технологія *reflection*, яка дозволяє звертатися до класів, методів і полів, використовуючи лише їх ім’я в текстовому вигляді. З її допомогою також можна створити об’єкт без ключового слова new, проте ця технологія досить специфічна, застосовується в окремих випадках, а крім того, досить проста і тому в даному курсі не розглядається. Все ж таки приведемо приклад її застосування:

```
Point p = null;
try {
  // у наступному рядку, використовуючи лише
  // текстове ім’я класу Point, породжується
  // об’єкт без застосування ключового слова new
p=(Point) Class.forName("Point").newInstance();
} catch (Exception e) { // обробка помилок
   System.out.println(e);
}
```

Об’єкт завжди "пам’ятає", від якого класу він був породжений. З іншого боку, як вже вказувалося, можна посилатися на об’єкт, використовуючи посилання іншого типу. Приведемо приклад, який будемо ще багато раз використовувати. Спочатку опишемо два класи, Parent і його спадкоємець Child:

```
// Оголошуємо клас Parent
class Parent {
}
```

```
// Оголошуємо клас Child і успадковуємо

// його від класу Parent

class Child extends Parent {

}
```

Поки нам не потрібно визначати які-небудь поля або методи. Далі оголосимо змінну одного типу і проініціалізуєм її значенням іншого типу:

```
Parent p = new Child();
```

Тепер змінна типу Parent указує на об'єкт, породжений від класу Child.

Над посилальними значеннями можна проводити наступні операції:

- звернення до полів і методів об'єкту
- оператор instanceof (повертає булеве значення)
- операції порівняння == і != (повертають булеве значення)
- оператор наведення типів
- оператор з умовою ? :
- оператор конкатенації з рядком +

Звернення до полів і методів об'єкту можна назвати основною операцією над посилальними величинами. Здійснюється вона за допомогою символу . (крапка). Приклади її застосування наводитимуться.

Використовуючи оператор instanceof, можна дізнатися, від якого класу походить об'єкт. Даний оператор має два аргументи. Зліва указується посилання на об'єкт, а справа - ім'я типу, на сумісність з яким перевіряється об'єкт. Наприклад:

```
Parent p = new Child();
// перевіряємо змінну p типу Parent
// на сумісність з типом Child
print(p instanceof Child);
```

Результатом буде true. Таким чином, оператор instanceof спирається не на тип посилання, а на властивості об'єкту, на який вона посилається. Але цей оператор повертає дійсне значення не тільки для того типу, від якого був породжений об'єкт. Додамо до вже оголошених класів ще один:

```
// Оголошуємо новий клас і успадковуємо
// його від класу Child
class Childofchild extends Child { }
```

Тепер заведемо змінну нового типу:

```
Parent p = new Childofchild();
print(p instanceof Child);
```

У першому рядку оголошується змінна типу Parent, яка ініціалізувалася посиланням на об'єкт, породжений від Childofchild. У другому рядку оператора instanceof

аналізує сумісність посилання типу Parent з класом child, причому задіяний об'єкт не породжений ні від першого, ні від другого класу. Проте, оператор поверне true, оскільки клас, від якого цей об'єкт породжений, успадковується від Child.

Додамо ще один клас:

```
class Child2 extends Parent {
}
```

І знову оголосимо змінну типу Parent:

```
Parent p=new Child();
print(p instanceof Child);
print(p instanceof Child2);
```

Змінна p має тип Parent, а значить, може посилатися на об'єкти типу Child або Child2. Оператор instanceof допомагає розібратися в ситуації:

```
true
false
```

Для посилання, рівного null, оператор instanceof завжди повертатиме значення false.

З вивченням властивостей об'єктної моделі Java ми повертатимемося до алгоритму роботи оператора instanceof.

Оператори порівняння == і != перевіряють рівність (або нерівність) об'єктних величин саме по посиланню. Проте часто потрібне альтернативне порівняння - за значенням. Порівняння за значенням має справу з поняттям стан об'єкту. Сам сенс цього виразу розглядається в ООП, що ж до реалізації в Java, той стан об'єкту зберігається в його полях. При порівнянні по посиланню ні тип об'єкту, ні значення його полів не враховуються, true повертається тільки в тому випадку, якщо обидва посилання указують на один і той же об'єкт.

```
Point p1=new Point(2,3);
Point p2=p1;
Point p3=new Point(2,3);
print(p1==p2);
print(p1==p3);
```

Результатом буде:

```
true
false
```

Перше порівняння виявилося істинним, оскільки змінна p2 посилається на той же об'єкт, що і p1. Друге ж порівняння помилкове, не дивлячись на те, що змінна p3 посилається на об'єкт-крапку з точно такими ж координатами. Проте це інший об'єкт, який був породжений іншим виразом new.

Якщо один з аргументів оператора = рівний null, а інший - ні, те значення такого виразу буде false. Якщо ж обидва операнди null, то результат буде true.

Для коректного порівняння за значенням існує спеціальний метод equals, який буде розглянутий пізніше. Наприклад, рядки треба порівнювати таким чином:

```
String s = "abc";
s=s+1;
print(s.equals("abc1"));
```

Операція з умовою ? : працює як завжди і може приймати другий і третій аргументи, якщо вони обидва одночасно типу посилання. Результат такого оператора також матиме об’єктний тип.

Як і прості типи, посилальні величини можна складати з рядком. Якщо посилання рівне null, то до рядка додається текст "null". Якщо ж посилання указує на об’єкт, то у нього викликається спеціальний метод (він буде розглянутий нижче, його ім’я tostring()) і текст, який він поверне, буде доданий до рядка.

Клас Object

У Java множинне спадкоємство відсутнє. Кожен клас може мати тільки одного батька. Таким чином, ми можемо прослідкувати ланцюжок спадкоємства від будь-якого класу, піднімаючись все вище. Існує клас, на якому такий ланцюжок завжди закінчується, це клас Object. Саме від нього успадковуються всі класи, в оголошенні яких явно не вказаний інший батьківський клас. А значить, будь-який клас безпосередньо, або через своїх батьків, є спадкоємцем Object. Звідси витікає, що методи цього класу є у будь-якого об'єкту (поля bobject відсутні), а тому вони представляють особливий інтерес.

Розглянемо основні з них.

getclass()

Цей метод повертає об'єкт класу Class, який описує клас, від якого був породжений цей об'єкт. Клас Class буде розглянутий нижче. У нього є метод getname(), що повертає ім'я класу:

```
String s = "abc";

Class cl=s.getClass();

print(cl.getName());
```

Результатом буде рядок:

```
java.lang.String
```

На відміну від оператора instanceof, метод getclass() завжди повертає точно той клас, від якого був породжений об'єкт.

equals()

Цей метод має один аргумент типу Object і повертає boolean. Як вже говорилося, equals() служить для порівняння об'єктів за значенням, а не по посиланню. Порівнюється стан об'єкту, у якого викликається цей метод, з переданим аргументом.

```
Point p1=new Point(2,3);

Point p2=new Point(2,3);

print(p1.equals(p2));
```

Результатом буде true.

Оскільки сам Object не має полів, а значить, і стани, в цьому класі метод equals повертає результат порівняння по посиланню. Проте при написанні нового класу можна перевизначити цей метод і описати правильний алгоритм порівняння за значенням (що і зроблено в більшості стандартних класів). Відповідно, в клас Point також необхідно додати перевизначений метод порівняння:

```
public boolean equals(Object про){

  // Спочатку необхідно переконатися, що
```

```
// переданий об’єкт сумісний з типом
 // Point
 if (про instanceof Point) {
   // Типи сумісні, можна провести
    // перетворення
    Point p = (Point)o;
    // Повертаємо результат порівняння
    // координат
    return p.x==x && p.y==y;
  }
 // Якщо об’єкт не сумісний з Point
// повертаємо false
 return false;
}
```

hashcode()

Даний метод повертає значення int. Мета hashcode() - представити будь-який об’єкт цілим числом. Особливо ефективно це використовується в хеш-таблицах (у Java є стандартна реалізація такого зберігання даних, вона буде розглянута пізніше). Звичайно, не можна зажадати, щоб різні об’єкти повертали різні хеш-коди, але, принаймні, необхідно, щоб об’єкти, рівні за значенням (метод equals() повертає true), повертали однакові хеш-коди.

У класі Object цей метод реалізований на рівні JVM. Сама віртуальна машина генерує число хеш-кодів, грунтуючись на розташуванні об’єкту в пам’яті.

tostring()

Цей метод дозволяє отримати текстовий опис будь-якого об’єкту. Створюючи новий клас, даний метод можна перевизначити і повертати докладніший опис. Для класу Object і його спадкоємців, що не перевизначили tostring (), метод повертає наступний вираз:

```
getclass().getName()+"@"+hashCode()
```

Метод getname() класу Class вже приводився в приклад, а хеш-код ще додатково обробляється спеціальною функцією для уявлення в шістнадцятковому форматі.

Наприклад: print(new Object());

Результатом буде:

java.lang.Object@92d342

В результаті цей метод дозволяє по текстовому опису зрозуміти, від якого класу був породжений об'єкт і, завдяки хеш-коду, розрізняти різні об'єкти, створені від одного класу.

Саме цей метод викликається при конвертації об'єкту в текст, коли він передається як аргумент операторові конкатенації рядків.

finalize()

Даний метод викликається при знищенні об'єкту автоматичним складальником сміття (garbage collector). У класі Object він нічого не робить, проте в класі-спадкоємцеві дозволяє описати всі дії, необхідні для коректного видалення об'єкту, такі як закриття з'єднань з БД, мережевих з'єднань, зняття блокувань на файли і так далі. У звичайному режимі безпосередньо цей метод викликати не потрібно, він відпрацює автоматично. Якщо необхідно, можна звернутися до нього явним чином.

У методі finalize() потрібно описувати тільки додаткові дії, пов'язані з логікою роботи програми. Все необхідне для видалення об'єкту JVM зробить сама.

Клас String

Як вже вказувалося, клас String займає в Java особливе положення. Екземпляри тільки цього класу можна створювати без використання ключового слова new. Кожен строковий літерал породжує екземпляр String, і це єдиний літерал (окрім null), що має об'єктний тип.

Потім значення будь-якого типу може бути приведене до рядка за допомогою оператора конкатенації рядків, який був розглянутий для кожного типу, як примітивного, так і об'єктного.

Ще однією важливою властивістю даного класу є незмінність. Це означає, що, породивши об'єкт, що містить якесь значення-рядок, ми вже не можемо змінити дане значення - необхідно створити новий об'єкт.

```
String s="a";

s="b";
```

У другому рядку змінна змінила своє значення, але тільки створивши новий об'єкт класу String.

Оскільки кожен строковий літерал породжує новий об'єкт, що являється дуже ресурсоємною операція в Java, часто компілятор прагне оптимізувати цю роботу.

По-перше, якщо використовується декілька літералів з однаковим значенням, для них буде створений один і той самий об'єкт.

```
String s1 = "abc";

String s2 = "abc";

String s3 = "a"+"bc";

print(s1==s2);

print(s1==s3);
```

Результатом буде:

```
true

true
```

Тобто у разі, коли рядок конструюється з констант, відомих вже на момент компіляції, оптимізатор також підставляє один і той самий об'єкт.

Якщо ж рядок створюється виразом, який може бути обчислений тільки під час виконання програми, то породжуватиметься новий об'єкт:

```
String s1="abc";

String s2="ab";

print(s1==(s2+"c"));
```

Результатом буде false, оскільки компілятор не може передбачити результат складання значення змінної з константою.

У класі String визначений метод intern(), який повертає один і той самий об'єкт-рядок для всіх екземплярів, рівних за значенням. Тобто якщо для посилань s1 і s2 вірний вираз s1.equals(s2), то вірно is1.intern()==s2.intern().

Зрозуміло, в класі перевизначені методи equals() і hashcode(). Метод tostring() також перевизначений і повертає він сам об'єкт-рядок, тобто для будь-якого посилання s типу String, не рівному null, вірний вираз s==s.toString().

Клас Class

Клас Class є метакласом для всіх класів Java. Коли JVM завантажує файл .class, який описує деякий тип, в пам'яті створюється об'єкт класу Class, який зберігатиме цей опис.

Наприклад, якщо в програмі є рядок

```
Point p=new Point(1,2);
```

то це означає, що в системі створені наступні об'єкти:

1. об'єкт типу Point, що описує крапку (1,2);

2. об’єкт класу Class, що описує клас Point;

3. об’єкт класу Class, що описує клас Object. Оскільки клас Point успадковується від Object, його опис також необхідний;

4. об’єкт класу Class, що описує клас Class. Це звичайний Java-класс, який повинен бути завантажений по загальних правилах.

Одне із застосувань класу Class вже було розглянуто - використання методу getclass() класу Object. Якщо продовжити останній приклад з крапкою:

```
Class cl=p.getClass();      // це об’єкт №2 із списку

Class cl2=cl.getClass();   // це об’єкт №4 із списку

Class cl3=cl2.getClass();   // знову об’єкт №4
```

Вираз cl2==cl3 вірно.

Інше застосування класу Class також приводилося в прикладі застосування технології reflection.

Окрім прямого використання метакласу для зберігання в пам’яті опису класів, Java використовує ці об’єкти і для інших цілей, які будуть розглянуті нижче (статичні змінні, синхронізація статичних методів і так далі).

Висновок

Типи даних - одна з ключових тим курсу. Неможливо написати жодної програми, не використовуючи їх. Ось список деяких операцій, де застосовуються типи:

- оголошення типів;
- створення об’єктів;
- при оголошенні полів - тип поля;
- при оголошенні методів - вхідні параметри, повернене значення;
- при оголошенні конструкторів - вхідні параметри;
- оператор приведення типів;
- оператор instanceof;
- оголошення локальних змінних;
- багато інших - обробка помилок, import-вираження і так далі.

Принципові відмінності між примітивними і посилальними типами даних розглядатимуться і далі по ходу курсу. Вивчення об’єктної моделі Java дасть основу для докладнішого викладу об’єктних типів - звичайних і абстрактних класів, інтерфейсів і масивів. Після приведення типів будуть описані зв’язки між типом змінної і типом її значення.

У обговоренні майбутньої версії Java 1.5 згадуються темплейти (templates), які істотно розширять поняття типу даних, якщо дійсно увійдуть до стандарту мови.

У лекції розглянуто те, що Java є мовою, що строго типізується, тобто тип всіх змінних і виразів визначається вже компілятором. Це дозволяє істотно підвищити надійність і якість коду, а також робить необхідним розуміння програмістами об'єктної моделі.

Всі типи в Java діляться на дві групи - фіксовані прості, або примітивні, типи (8 типів) і численна група об'єктних типів (класів). Примітивні типи дійсно є сховищами даних свого типу. Посилальні змінні зберігають посилання на деякий об'єкт сумісного типу. Вони також можуть набувати значення null, не вказуючи ні на який об'єкт. JVM підраховує кількість посилань на кожен об'єкт і активізує механізм автоматичної збірки сміття для видалення невживаних об'єктів.

Були розглянуті змінні. Вони характеризуються трьома основними параметрами - ім'я, тип і значення. Будь-яка змінна повинна бути оголошена і при цьому може ініціалізуватись. Можливе використання модифікатора final.

Примітивні типи складаються з п'яти цілочисельних, включаючи символьний тип, два дробових і одного булевого. Цілочисельні літерали мають обмеження, пов'язані з типами даних. Були розглянуті всі оператори над примітивними типами, тип повертаючого значення і тонкості їх використання.

Потім вивчалися об'єкти, способи їх створення і оператори, що виконують над ними різні дії, зокрема принцип роботи оператора instanceof. Далі були розглянуті найголовніші класи в Java - Object, Class, String.

ТЕМА. *Імена. Пакети. Затіняюче та закриваюче оголошення.*

План

1. *Імена. Зона видимості імені.*
2. *Пакети. Створення фізичної і логічної структури класів.*
3. *Затіняючи(Shadowing) та закриваюче (Obscuring) оголошення.*

Рекомендована література: 3, 4.

Теоретичні відомості

Імена (names) використовуються в програмі для доступу до оголошених (declared) раніше "об'єктів", "*елементів*", "конструкцій" мови (всі ці слова-синоніми були використані тут в їх загальному сенсі, а не як терміни ООП, наприклад). Конкретніше, в Java є імена:

- *пакети*;
- класи;
- інтерфейси;
- *елементи* (member) посилальних типів:
  - поля;
  - методи;
  - внутрішні класи і інтерфейси;
- аргументи:
  - методів;
  - конструкторів;
  - обробників помилок;
- локальні змінні.

Відповідно, всі вони повинні бути оголошені спеціальним чином, що поступово розглядатиметься по ходу курсу. Так само оголошуються конструктори, проте їх ім'я співпадає з ім'ям класу, тому вони не потрапили в цей список.

Нагадаємо, що *пакети* (packages) в Java - це спосіб логічно групувати класи, що необхідний, оскільки часто кількість класів в системі складає декілька тисяч, або навіть десятків тисяч. Окрім класів і інтерфейсів в *пакетах*, можуть знаходитися вкладені пакети. Синонімами цього слова в інших мовах є бібліотека або модуль.

Імена бувають ***простими*** (simple), такими, що складаються з одного ідентифікатора (вони визначаються під час оголошення) і ***складеними*** (qualified), такими, що складаються з послідовності ідентифікаторів, розділених крапкою. Для пояснення цих термінів необхідно розглянути ще одне поняття.

У *пакетів* і посилальних *типів* (класів, інтерфейсів, масивів) є *елементи* (members). Доступ до *елементів здійснюється за* допомогою виразу, що складається з імен, наприклад, *пакету* і класу, розділених крапкою.

Далі класи і інтерфейси називатимуться об'єднуючим терміном ***тип*** (type).

Елементами *пакету* є класи, що містяться в нім, і інтерфейси, а також вкладені *пакети*. Щоб отримати *складене ім'я пакету*, необхідно до повного імені *пакету*, в якому він розташовується, додати крапку, а потім його власне *просте ім'я*. Наприклад, *складене ім'я* основного *пакету* мови Java - java.lang (тобто просте *ім'я* цього пакету lang, і він знаходиться в охоплюючому пакеті java). Усередині нього є вкладений пакет, призначений для *типів* технології reflection, яка згадувалася в попередніх розділах. Проста назва пакету reflect, а значить, складена - java.lang.reflect.

Просте ім'я класів і інтерфейсів дається при оголошенні, наприклад, Object, String, Point. Щоб получити складне *ім'я* таких *типів*, треба до *складеного імені пакету*, в якому знаходиться *тип*, через крапку добавити просте *ім'я типу*. Наприклад, java.lang.Object, java.lang.reflect.Method або com.myfirm.MainClass. Сенс останнього виразу такий: спочатку йде звернення до *пакету* com, потім до його *елементу* - вкладеного *пакету myfirm*, а потім до *елементу пакету* myfirm - класу Mainclass. Тут com.myfirm - *складене ім'я пакету*, де лежить клас Mainclass, а Mainclass - *просте ім'я*. Складаємо їх і розділяємо крапкою - виходить повне ім'я класу com.myfirm.MainClass.

Для посилальних *типів елементами* є поля і методи, а також внутрішні *типи* (класи і інтерфейси). Елементи можуть бути як безпосередньо оголошені в класі, так і отримані по спадку від батьківських класів і інтерфейсів, якщо такі є. *Просте ім'я елементів* також дається при ініціалізації. Наприклад, tostring(), PI, Innerclass. *Складене ім'я* виходить шляхом об'єднання *простого* або *складеного імені типу*, або змінної об'єктного *типу* з ім'ям *елементу*. **Наприклад**, ref.toString(), java.lang.Math.PI,OuterClass. InnerClass. Інші звернення до *елементів* посилальних *типів* вже неодноразово застосовувалися в попередніх розділах.

Імена і ідентифікатори

Тепер, коли ми розглянули *прості* і *складені* імена, уточнимо різницю між ідентифікатором (нагадаємо, що це вид лексеми) і ім'ям. Зрозуміло, що *просте ім'я* складається з одного ідентифікатора, а *складене* - з декількох. Проте не всякий ідентифікатор входить до складу імені.

По-перше, у виразі оголошення (declaration) ідентифікатор ще не є ім'ям. Іншими словами, він стає ім'ям після першої появи в коді в місці оголошення.

По-друге, існує можливість звертатися до полів і методів об'єктного *типу* не через ім'я *типу* або об'єктної змінної, а через посилання на об'єкт, отримане в результаті виконання виразу. Приклад такого виклику:

```
country.getCity().getStreet();
```

У даному прикладі getstreet є не ім'ям, а ідентифікатором, оскільки відповідний метод викликається у об'єкту, отриманого в результаті виклику методу getcity(). Причому country.getCity якраз являється складеним *ім'ям* методу.

Нарешті, ідентифікатори також використовуються для назв влучний (label). Ця конструкція розглядається пізнішим, проте приведемо приклад, що показує, що простори імен і назв міток повністю розділені.

```
num:
  for (int num = 2; num <= 100; num++) {
    int n = (int)Math.sqrt(num)+1;
    while (--n != 1) {
      if (num%n==0) {
        continue num;
      }
    }
    System.out.print(num+" ");
```

```
  }
```
Результатом будуть прості числа менше 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Ми бачимо, що тут застосовуються однойменні змінна і мітка num, причому остання використовується для виходу з внутрішнього циклу while на зовнішній for.

Очевидно, що зручніше використовувати *просте ім'я*, а не *складене*, оскільки воно коротше і його легко запам'ятати. Проте зрозуміло, що якщо в системі є дуже багато класів з безліччю змінних, можна зіткнутися з ситуацією, коли в різних класах є однойменні змінні або методи. Для вирішення цій і інших подібних проблем вводиться нове поняття - зона *видимості*.

Зона видимості (введення)

Щоб не примушувати програмістів, що спільно працюють над різними класами однієї системи, координувати імена, які вони дають різним конструкціям мови, у кожного імені є *зона видимості*(scope). Якщо звернення, наприклад, до поля, йде з частини коду, *видимості* його імені, *що* потрапляє в *область*, то можна користуватися *простим ім'ям*, якщо немає - необхідно застосовувати *складене*.

Наприклад:

```
class Point {
  int x,y;
  int getx() {
    return x; // просте ім'я
  }
}
class Test {
  void main() {
    Point p = new Point();
   p.x=3; // складене ім'я
  }
}
```

Видно, що до поля x зсередини класу можна звертатися по *простому імені*. До нього ж з іншого класу можна звернутися тільки по *складеному імені*. Воно складається з імені змінної, що посилається на об'єкт, і імені поля.

Тепер необхідно розглянути *зони видимості* для всіх елементів мови. Проте раніше з'ясуємо, що таке пакети, як і для чого вони використовуються?

Пакети

Програма на Java є набором *пакетів* (packages). Кожний пакет може включати вкладені *пакети*, тобто вони утворюють ієрархічну систему.

Крім того, *пакети* можуть містити класи і інтерфейси і таким чином групують *типи*. Це необхідно відразу для декількох цілей. По-перше, чисто фізично неможливо працювати з великою кількістю класів, якщо вони "звалені в купу". По-друге, модульна декомпозиція полегшує проектування системи. До того ж, як буде показано нижче, існує

спеціальний рівень доступу, що дозволяє *типам* з одного *пакету* тісніше взаємодіяти один з одним, ніж з класами із інших *пакетів*. Таким чином, за допомогою *пакетів* проводиться логічне групування типів. З ООП відомо, що велика зв'язність системи, тобто середня кількість класів, з якими взаємодіє кожен клас, помітно ускладнює розвиток і підтримку такої системи. Використовуючи *пакети*, набагато простіше організувати ефективну взаємодію підсистем один з одним.

Нарешті, кожен *пакет* має свій простір імен, що дозволяє створювати однойменні класи в різних *пакетах*. Таким чином, розробникам не доводиться витрачати час на вирішення конфлікту імен.

Елементи пакету

Ще раз повторимо, що *елементами пакету* є вкладені *пакети* і типи (класи і інтерфейси). Однойменні *елементи* заборонені, тобто не може бути однойменного класу і інтерфейсу, або вкладеного *пакету* і типу. Інакше виникне помилка компіляції.

Наприклад, в JDK 1.0 *пакет* java містив *пакети* applet, awt, io, lang, net,util і не містив жодного *типу*. У пакет java.awt входив вкладений пакет image і 46 класів і інтерфейсів.

Складене ім'я будь-якого *елементу пакету* - це *складене ім'я* цього пакету плюс *просте ім'я елементу*. Наприклад, для класу Object в пакеті java.lang *складеним ім'ям* буде java.lang.Object, а для пакету image в пакеті java.awt - java.awt.image.

Ієрархічна структура пакетів була введена для зручності організації зв'язаних пакетів, проте вкладені пакети, або сусідні, тобто вкладені в один і той же пакет, не мають ніяких додаткових зв'язків між собою, окрім обмеження на неспівпадіння імен. Наприклад, *пакети* space.sun, space.sun.ray, space.moon ifactory.store абсолютно "рівні" між собою і *типи* одного з цих пакетів не мають ніякого особливого доступу до *типів* інших пакетів.

Платформна підтримка пакетів

Простим способом організації пакетів і *типів* є звичайна файлова структура. Розглянемо вироджений приклад, коли всі пакети, початковий і бінарний код розташовуються в одному каталозі і його підкаталогах.

У цьому кореневому каталозі повинна бути тека java, відповідна основному пакету мови, а в ній, у свою чергу, вкладені теки applet, awt, io, lang, net, util.

Припустимо, розробник працює над моделлю сонячної системи, для чого створив класи Sun, Moon і Test і розташував їх в пакеті space.sunsystem. У такому разі в кореневому каталозі повинна бути тека space, відповідна однойменному пакету, а в ній - тека sunsystem, в якій зберігаються класи цього розробника.

Як відомо, початковий код розташовується у файлах з розширенням .java, а бінарний - з розширенням .class. Таким чином, вміст теку sunsystem може виглядати таким чином:

Moon.java

Moon.class

Sun.java

Sun.class

Test.java

Test.class

Іншими словами, початковий код класів

space.sunsystem.Moon

space.sunsystem.Sun

space.sunsystem.Test

зберігається у файлах

space\sunsystem\moon.java

space\sunsystem\sun.java

space\sunsystem\test.java

а бінарний код - у відповідних .class-файлах. Звернете увагу, що перетворення імен пакетів у файлові шляхи зажадало заміни роздільника „.” (крапки) на символ-роздільник файлів (для Windows це зворотний слеш \). Таке перетворення може виконати як компілятор для пошуку початкових текстів і бінарної коди, так і віртуальна машина для завантаження класів і інтерфейсів.

Зверніть увагу, що було б помилкою запускати Java прямо з теки space\sunsystem і намагатися звертатися до класу Test, не дивлячись на те, що файл-опис лежить саме в ній. Необхідно піднятися на два рівні каталогів вище, щоб Java, побудувавши шлях з імені пакету, змогла виявити потрібний файл.

Крім того, важливо, що Java завжди розрізняє регістр ідентифікаторів, а значить, назви файлів і каталогів повинні точно відповідати запрограмованим іменам. Хоча в деяких випадках операційна система може забезпечити доступ, незважаючи на регістр, при зміні обставин розбіжності можуть привести до збоїв.

Існує спеціальний вираз, що оголошує пакет (детально розглядається нижче). Воно передує оголошенню *типу* і позначає, якому пакету належатиме цей *тип*. Таким чином, набір доступних пакетів визначається набором доступних файлів, що містять оголошення *типів* і пакетів. Наприклад, якщо створити порожній каталог, або заповнити його сторонніми файлами, це зовсім не приведе до появи пакету в Java.

Які файли доступні для утиліт Java SDK (компілятора, інтерпретатора і так далі), встановлюється на рівні операційної системи, адже утиліти - це звичайні програми, які виконуються під управлінням ОС і, звичайно, слідують її правилам. Наприклад, якщо пакет містить один *тип*, але файл, що описує його, недоступний поточному користувачеві ОС для читання, для Java цей *тип* і цей *пакет* не існуватимуть.

Зрозуміло, що далеко не завжди зручно зберігати всі файли в одному каталозі. Часто класи знаходяться в різних місцях, а деякі можуть навіть розповсюджуватися у вигляді архівів, для прискорення завантаження через мережу. Копіювати всі такі файли в одну теку було б вкрай скрутно.

Тому Java використовує спеціальну змінну оточення, яка називається classpath. Аналогічно тому, як змінна path допомагає системі знаходити і завантажувати динамічні бібліотеки, ця змінна допомагає працювати з Java-класами. Її значення повинне складатися з шляхів до каталогів або архівів, розділених крапкою з комою. З версії 1.1 підтримуються архіви типів ZIP і JAR (Java Archive) - спеціальний формат, розроблений на основі ZIP для Java.

Наприклад, змінна classpath може мати таке значення:

.;c:\java\classes;d:\lib\3Dengine.zip;

d:\lib\fire.jar

В результаті всі вказані каталоги і вміст всіх архівів "додається" до початкового кореневого каталога. Java у пошуках класу шукатиме його за описаним вище правилом у всіх вказаних теках і архівах по порядку. Зверніть увагу, що першим в змінній вказаний поточний каталог (представлений крапкою). Це робиться для того, щоб пошук завжди починався з початкового кореневого каталога. Звичайно, такий запис не є обов'язковим і робиться на розсуд розробника.

Не дивлячись на явні зручності такої конструкції, вона таїть в собі і небезпеку. Якщо класи, що розробляються, зберігаються в деякому каталозі і він вказаний в classpath пізніше, ніж якийсь інший каталог, в якому виявляються однойменні *типи*, розібратися в такій ситуації буде непросто. До класів вноситимуться зміни, які ніяк не виявляються при запуску через те, що Java насправді завантажує одні і ті ж файли із сторонньої теки.

Тому до даного змінного середовища оточення необхідно відноситися з особливою увагою. Корисно пам'ятати, що необов'язково встановлювати її значення відразу для всієї операційної системи. Його можна явно указувати при кожному запуску компілятора або віртуальної машини як опцію, що, по-перше, ніколи не вплине на інші Java-программы, а по-друге, помітно спрощує пошук помилок, пов'язаних з некоректним значенням classpath.

Нарешті, можна застосовувати і альтернативні підходи до зберігання *пакетів* і файлів з початковим і бінарним кодом. Наприклад, як таке сховище може використовуватися база даних. Більш того, існує обмеження на розміщення оголошень класів в .java-файлах, яке розглядається нижчим, а при використанні БД будь-які обмеження можна зняти. Проте, при такому підході рекомендується надавати утиліти імпорту/експорту з урахуванням обмеження для перетворень із/у файли.

Модуль компіляції

Модуль компіляції (compilation unit) зберігається в текстовому .java-файлі і є одиничною порцією вхідних даних для компілятора. Він складається з трьох частин:

- оголошення *пакету*;
- import-вираження;
- оголошення верхнього рівня.

Оголошення *пакету* одночасно указує, якому пакету належатимуть всі оголошувані нижче *типи*. Якщо даний вираз відсутній, значить, ці класи розташовуються в безіменному пакеті (інша назва - пакет по замовчанні).

Import-вираження дозволяють звертатися до *типів* з інших пакетів по їх *простих іменах*, "імпортувати" їх. Ці вирази також необов'язкові.

Нарешті, оголошення верхнього рівня містять оголошення одного або декількох *типів*. Назва "верхнього рівня" протиставляє ці класи і інтерфейси, розташовані в пакетах, внутрішнім *типам*, які є *елементами* і розташовуються усередині інших типів. Як не дивно, ця частина також є необов'язковою, в тому сенсі, що у разі її відсутності компілятор не видасть помилки. Проте ніяких .class-файлів згенеровано теж не буде.

Доступність модулів компіляції визначається підтримкою платформи, оскільки утиліти Java є звичайними програмами, які виконуються операційною системою по загальних правилах.

Розглянемо всі три частини детальніше.

Оголошення пакету

Перший вираз в модулі компіляції - оголошення *пакету*. Воно записується за допомогою ключового слова package, після якого вказується повне ім'я *пакету*.

Наприклад, першим рядком (після коментарів) у файлі java/lang/object.java йде:

```
package java.lang;
```

Це одночасно служить оголошенням *пакету* lang, вкладеного в пакет java, і вказівкою, що оголошуваний нижче клас Object знаходиться в даному пакеті. Так складається повне ім'я класу java.lang.Object.

Якщо цей вираз відсутній, то такий *модуль компіляції* належить безіменному пакету. Цей пакет за умовчанням обов'язково повинен підтримуватися реалізацією Java-платформы. Зверніть увагу, що він не може мати вкладених пакетів, оскільки *складене ім'я* пакету повинне обов'язково починатися з імені пакету верхнього рівня.

Таким чином, найпростіша програма може виглядати таким чином:

```
class Simple {
  public static void main(String s[]){
   System.out.println("Hello!");
  }
}
```

Цей *модуль компіляції* належатиме безіменному пакету.

Пакет за умовчанням був введений в Java для полегшення написання дуже невеликих або тимчасових застосувань, для експериментів. Якщо ж програма розповсюджуватиметься для користувачів, то рекомендується розташувати її в пакеті, який, у свою чергу, повинен бути правильно названий. Угоди по іменуванню розглядаються нижчим.

Доступність пакету визначається по доступності модулів компіляції, в яких він оголошується. Точніше, пакет доступний тоді і тільки тоді, коли виконується будь-яка з наступних двох умов:

- доступний *модуль компіляції* з оголошенням цього пакету;
- доступний один з вкладених пакетів цього пакету.

Таким чином, для наступного коду:

```
package space.star;


class Sun {
}
```

якщо файл, який зберігає цей *модуль компіляції*, доступний Java-платформі, то пакети space і вкладений в нього star (повна назва space.star) також стають доступні для Java.

Якщо пакет доступний, то *зона видимості* його оголошення - всі доступні модулі компіляції. Простіше кажучи, всі існуючі пакети доступні для всіх класів, ніяких обмежень на доступ до пакетів в Java не немає.

Потрібно, щоб пакети java.lang і java.io, а значить, і java, завжди були доступні для Java-платформы, оскільки вони містять класи, необхідні для роботи будь-якого застосування.

Імпорт-вираження

Як буде розглянуто нижче, *зона видимості* оголошення *типу* - пакет, в якому він розташовується. Це означає, що всередині даного пакету допускається звернення до *типу* по його *простому імені*. Зі всіх інших пакетів необхідно звертатися по *складеному імені*, тобто повне ім'я пакету плюс *просте ім'я* типу, розділені крапкою. Оскільки пакети можуть мати досить довгі імена (наприклад, додатковий пакет у складі Jdk1.2 називається com.sun.image.codec.jpeg), а *тип* може багато разів використовуватися в модулі компіляції, таке обмеження може привести до ускладнення початкового коду і складнощів в розробці.

Для вирішення цієї проблеми вводяться import-вираження, що дозволяють імпортувати *типи* в *модуль компіляції* і далі звертатися до них по *простих іменах*. Існує два види таких виразів:

- імпорт одного *типу*;
- імпорт пакету.

Важливо підкреслити, що імпортуючі вирази є, по суті, підказкою компілятора. Він користується ними, щоб для кожного *простого імені типу* з іншого пакету отримати його повне ім'я, яке і потрапляє в код, що компілюється. Це означає, що імпортуючих виразів може бути дуже багато, включаючи і ті, що імпортують невживані пакети і *типи*, але це ніяк не відіб'ється ні на розмірі, ні на якості бінарних кодів. Також байдуже, звертатися до *типу* по його повному імені, або включити його в імпортуючий вираз і звертатися по *простому імені* - результат буде один і той же.

Імпортуючі вирази мають ефект тільки всередині модуля компіляції, в якому вони оголошені. Всі оголошення *типів* вищого рівня, що знаходяться в цьому ж модулі, можуть однаково користуватися імпортованими *типами*. До імпортованих *типів* можливий і звичайний доступ по повному імені.

Вираз, що імпортує один *тип*, записується за допомогою ключового слова import і повного імені *типу*. Наприклад:

```
import java.net.URL;
```

Такий вираз означає, що надалі в цьому модулі компіляції *просте ім'я* URL позначатиме однойменний клас з пакету java.net. Спроба імпортувати *тип*, недоступний на момент компіляції, викличе помилку. Якщо один і той же тип імпортується кілька разів, то це не створює помилки, а дубльовані вирази ігноруються. Якщо ж імпортуються типи з однаковими *простими іменами* з різних пакетів, то така ситуація породить помилку компіляції.

Вираз, що імпортує пакет, включає повне ім'я пакету таким чином.

```
import java.awt.*;
```

Цей вираз робить доступними всі типи, що знаходяться в пакеті java.awt, по їх *простому імені*. Спроба імпортувати пакет, недоступний на момент компіляції, викличе помилку. Імпортування одного пакету багато разів не створює помилки, дубльовані вирази ігноруються. Зверніть увагу, що імпортувати вкладений пакет не можна.

Наприклад:

```
// приклад викличе помилку компіляції
import java.awt.image;
```

Створюється враження, що тепер ми можемо звертатися до типів пакету java.awt.image по спрощеному імені, наприклад, image.ImageFilter. Насправді приклад викличе помилку компіляції, оскільки даний вираз розцінюється як імпорт типу, а в пакеті java.awt відсутній тип image.

Аналогічно, вираз

```
import java.awt.*;
```

не робить доступнішими класи пакету java.awt.image, їх необхідно імпортувати окремо.

Оскільки пакет java.lang містить *типи*, без яких неможливо створити жодну програму, він неявним чином імпортується в кожен *модуль компіляції*. Таким чином, всі типи з цього пакету доступні по їх простих *іменах* без яких-небудь додаткових зусиль. Спроба імпортувати даний пакет ще раз буде проігнорована.

Допускається одночасно імпортувати пакет і який-небудь тип з нього:

```
import java.awt.*;
import java.awt.Point;
```

Може виникнути питання, як же краще поступати - імпортувати типи окремо або весь пакет відразу? Чи є яка-небудь різниця в цих підходах?

Різниця полягає в алгоритмі роботи компілятора, який приводить кожне *просте ім'я* до повного. Він складається з трьох кроків:

- спочатку є видимими вирази, що імпортують типи;
- потім інші типи, оголошені в поточному пакеті, зокрема в поточному модулі компіляції;
- нарешті, є видимими вирази, що імпортують пакети.

Таким чином, якщо тип явно імпортований, то неможливе ні оголошення нового типу з таким же ім'ям, ні доступ по *простому імені* до однойменного типу в поточному пакеті.

Наприклад:

```
// приклад викличе помилку компіляції
package my_geom;

import java.awt.Point;

class Point {
}
```

Цей модуль викличе помилку компіляції, оскільки ім'я Point в оголошенні вищого типу розглядатиметься як звернення до імпортованого класу java.awt.Point, а його перевизначити, звичайно, не можна.

Якщо в пакеті оголошений тип:

```
package my_geom;

class Point {
}
```

то в іншому модулі компіляції:

```
package my_geom;

import java.awt.Point;
```

```
class Line {
  void main() {
    System.out.println(new Point());
  }
}
```

складається невизначена ситуація - який з класів, my_geom.Point або java.awt.Point, використовуватиметься при створенні об'єкту? Результатом буде:

```
java.awt.Point[x=0,y=0]
```

Відповідно до правил, ім'я Point було потрактоване на основі імпорту типу. До класу поточного пакету все ще можна звертатися по повному імені: my_geom.Point. Якби розглядався безіменний пакет, то звернутися до такого "перекритого" типу було б вже неможливо, що є додатковим аргументом до рекомендації розташовувати важливі програми в іменованих пакетах.

Тепер розглянемо імпорт пакету. Його ще називають "імпорт на вимогу", маючи на увазі, що ніякого "завантаження" всіх типів імпортованого пакету відразу при вказівці імпортуючого виразу не відбувається, їх повні імена підставляються у міру використання *простих імен* в коді. Можна імпортувати пакет і задіювати тільки один тип (або навіть жодного) з нього.

Змінимо розглянутий вище приклад:

```
package my_geom;
import java.awt.*;
class Line {
  void main() {
    System.out.println(new Point());
    System.out.println(new Rectangle());
  }
}
```

Тепер результатом буде:

```
my_geom.Point@92d342
java.awt.Rectangle[x=0,y=0,width=0,height=0]
```

Тип Point знайшовся в поточному пакеті, тому компілятору не довелося виконувати пошук по пакету java.awt. Другий об'єкт породжується від класу Rectangle, якого не існує в поточному пакеті, зате він виявляється в java.awt.

Також коректний тепер приклад:

```
package my_geom;
import java.awt.*;
class Point {
}
```

Таким чином, імпорт пакету не перешкоджає оголошенню нових типів або зверненню до існуючих типів поточного пакету по *простих іменах*. Якщо все ж таки потрібно працювати саме із зовнішніми типами, то можна скористатися імпортом типу, або звертатися до них по повних іменах. Крім того, вважається, що імпорт конкретних типів допомагає при прочитанні коду відразу зрозуміти, які зовнішні класи і інтерфейси використовуються в цьому модулі компіляції. Проте повністю покладатися на таке міркування не варто, оскільки можливі випадки, коли імпортовані типи не використовуються і, навпаки, в коді немає звернення до інших типів по повному імені.

Оголошення верхнього рівня

Далі *модуль компіляції* може містити одне або декілька оголошень класів і інтерфейсів. Детально формат такого оголошення розглядається в наступних лекціях, проте приведемо коротку інформацію і тут.

Оголошення класу починається з ключового слова class, інтерфейсу - interface. Далі указується ім'я типу, а потім у фігурних дужках описується тіло типу. Наприклад:

```
package first;

class Firstclass {

}

interface Myinterface {

}
```

Зона видимості типу - пакет, в якому він описаний. З інших пакетів до типу можна звертатися або складеному *імені*, або за допомогою імпортуючих виразів.

Проте, окрім *зони видимості*, в Java також є засоби розмежування доступу. За умовчанням тип оголошується доступним тільки для інших типів свого пакету. Щоб інші пакети також могли використовувати його, можна вказати ключове слово public:

```
package second;

public class Openclass {

}

public interface Publicinterface {

}
```

Такі типи доступні для всіх пакетів.

Оголошення верхнього рівня описують класи і інтерфейси, що зберігаються в пакетах. У версії Java 1.1 були введені внутрішні (inner) типи, які оголошуються усередині інших типів і є їх *елементами* разом з полями і методами. Дана можливість є допоміжною і досить заплутаною, тому в курсі докладно не розглядається, хоча деякі приклади і пояснення допоможуть в цілому її освоїти.

Якщо пакети, початковий і бінарний код зберігають у файловій системі, то Java може накладати обмеження на оголошення класів в модулях компіляції. Це обмеження створює помилку компіляції у випадку, якщо опис типу не виявиться у файлі з назвою, складеною з імені типу і розширення (наприклад, java), і при цьому:

- тип оголошений як public і, значить, може використовуватися з інших пакетів;
- тип використовується з інших модулів компіляції в своєму пакеті.

Ці умови означають, що в модулів компіляції може бути максимум один тип відповідає цим умовам.

Іншими словами, в модулі компіляції може бути максимум один public тип, і його ім'я і ім'я файлу повинні співпадати. Якщо ж в нім є не-public типу, імена яких не співпадають з ім'ям файлу, то вони повинні використовуватися тільки усередині цього модуля компіляції.

Якщо ж для зберігання пакетів застосовується БД, то таке обмеження не повинне накладатися.

На практиці ж програмісти часто поміщають в один *модуль компіляції* тільки один тип, незалежно від того, public він чи ні. Це істотно спрощує роботу з ними. Наприклад, опис класу space.sun.Sizехранится у файлі space\sun\size.java, а бінарний код - у файлі Size.class у тому ж каталозі. Саме так влаштовані всі стандартні бібліотеки Java.

Звернете увагу, що при оголошенні класів цілком допускаються перехресні звернення. Зокрема, наступний приклад абсолютно коректний:

```
package test;
/*
  * Клас Human, що описує людину
  */
class Human {
  String name;
  Car car;   // що належить людині машина
}
/*
  * Клас Car, що описує автомобіль
 */
class Car {
  String model;
  Human driver;   // водій, керівник
                  // машиною
}
```

Крім того, клас Car був використаний раніше, ніж був оголошений. Таке перехресне застосування типів також допускається у випадку, якщо вони знаходяться в різних пакетах. Компілятор повинен підтримувати можливість транслювати їх одночасно.

Унікальність імен пакетів

Оскільки Java створювався як мова, призначена для розповсюдження додатків через Internet, а додатки складаються із структури пакетів, необхідно зробити деякі зусилля, щоб не відбувся конфлікт імен. Імена двох використовуваних пакетів можуть співпасти після значного часу після їх створення. Виправити таке положення звичайному програмістові буде вкрай складно.

Тому творці Java пропонують наступний спосіб унікального іменування пакетів. Якщо програма створюється розробником, у якого є Internet-сайт, або ж він працює на організацію, у якої є сайт, і доменне ім'я такого сайту, наприклад, company.com, то імена пакетів повинні починатися з цих же слів, виписаних в зворотному порядку: com.company. Подальші вкладені пакети можуть носити назви підрозділів компанії, пакетів, прізвищ розробників, імена комп'ютерів і так далі

Таким чином, пакет верхнього рівня завжди записується ASCII-буквами в нижньому регістрі і може мати одне з наступних імен:

- трьохбуквені com, edu, gov, mil, net, org, int (цей список розширюється);
- двохбуквені, такі, що позначають імена країн, такі як ru, su, de, uk та інші.

Якщо ім'я сайту противоречит вимогам до ідентифікаторів Java, то можна зробити наступні кроки:

- якщо в імені знаходиться заборонений символ, наприклад, тире, то його можна замінити знаком підкреслення;
- якщо ім'я співпадає із зарезервованим словом, можна в кінці додати знак підкреслення;
- якщо ім'я починається з цифри, можна на початку додати знак підкреслення.

Приклади імен пакетів, складених по таких правилах:

com.sun.image.codec.jpeg

org.omg.CORBA.ORBPackage

oracle.jdbc.driver.OracleDriver

Проте, звичайно, ніхто не вимагає, щоб Java-пакети були обов'язково доступні на Internet-сайті, який дав їм ім'я. Швидше була зроблена спроба скористатися існуючою системою імен замість того, щоб створювати нову для іменування бібліотек.

Зона видимості імен

Зоною видимості оголошення деякого елементу мови називається частина програми, звідки допускається звернення до цього *елементу* по простому *імені*.

При розгляді кожного елементу мови указуватиметься його *зона видимості*, проте має сенс зібрати цю інформацію в одному місці.

Зона видимості доступного пакету - вся програма, тобто будь-який клас може використовувати доступний пакет. Проте необхідно пам'ятати, що звертатися до пакету можна тільки по його повному *складеному імені*. До пакету java.lang ні з якого місця не можна звернутися як до просто lang.

Зоною видимості імпортованого типу є всі оголошення верхнього рівня в цьому модулі компіляції.

Зоною видимості типу (класу або інтерфейсу) верхнього рівня є пакет, в якому він оголошений. З інших пакетів доступ можливий або по *складеному імені*, або за допомогою імпортуючого виразу, який допомагає компілятору відтворити *складене ім'я*.

Зона видимості *елементів* класів або інтерфейсів - це все тіло типу, в якому вони оголошені. Якщо звернення до цих *елементів* походить з іншого типу, необхідно скористатися *складеним ім'ям*. Ім'я може бути складене з *простого* або *складеного* імені типу, імені об'єктної змінної або ключових слів super або this, після чого через крапку указується *просте ім'я елементу*.

Аргументи методу, конструктора або обробника помилок видні тільки усередині цих конструкцій і не можуть бути доступні ззовні.

Зона видимості локальних змінних починається з моменту їх ініціалізації і до кінця блоку, в якому вони оголошені. На відміну від полів типів, локальні змінні не мають значень за умовчанням і повинні ініціалізовуватися явно.

```
 int x;
for (int i=0; i<10; i++) {
 int t=5+i;
}
// тут змінна t вже недоступна
// оскільки блок, в якому вона була
// оголошена, вже завершений, а змінна
// x ще недоступна, оскільки поки не була
// ініціалізувала
```

Певні проблеми виникають, коли відбувається перекриття *зон видимості* і виникає конфлікт імен різних конструкцій мови.

"**Затінююче" оголошення (Shadowing)**

Найпоширенішими випадками виникнення конфлікту імен є вираз, що імпортує пакет, і оголошення локальних змінних, або параметрів методів, конструкторів, обробників помилок. Імпорт пакету детально розглядався в цьому розділі. Якщо імпортований і поточний пакети містять однойменні типи, то їх *області* перетинаються. Як вже мовилося, перевага віддається типу з поточного пакету. Також розповідалося про те, як цю проблему вирішувати.

Перейдемо до проблеми перекриття імен полів класу і локальних змінних. Приклад:

```
class Human {
  int age;
  // вік
  int getage() {
    return age;
  }
  void setage(int age){
```

```
    age=age; // ???
  }
}
```

У класі Human (людина) оголошено поле age (вік). Зручно визначити також метод setage(), який повинен встановлювати нове значення віку для людини. Цілком логічно зробити у методі setage() один вхідний аргумент, який також називатиме age (адже як цей аргумент передаватиметься нове значення віку). Виходить, що в реалізації методу setage() потрібно написати age=age, в першому випадку маючи на увазі поле класу, в другому - параметр методу. Зрозуміло, що хоча з погляду компілятора це коректна конструкція, спроба послатися на дві різні змінні через одне ім'я успіхом не увінчається. Треба відмітити, що такі помилки трапляються деколи навіть у досвідчених розробників.

По-перше, розглянемо, через що виникла конфліктна ситуація. Є два елементи мови - аргумент методу і поле класу, *зони видимості* які перетнулися. *Зона видимості* поля класу більша, вона охоплює все тіло класу, тоді як *зона видимості* аргументу методу включає тільки сам метод. У такому разі всередині області перетину по *простому імені* доступний саме аргумент методу, а поле класу "затінюється" (shadowing) оголошенням параметра методу.

Залишається питання, як в такій ситуації все ж таки звернутися до поля класу. Якщо доступ по простому імені *неможливий*, треба скористатися *складеним*. Тут найзручніше застосувати спеціальне ключове слово this(воно детально розглядатиметься в наступних розділах). Слово this має значення посилання на об'єкт, усередині якого воно застосовується. Якщо викликати метод setage() у об'єкті класу Human і використовувати в цьому методі слово this, то його значення буде посиланням на даний об'єкт.

Виправлений варіант прикладу:

```
class Human {
  int age;   // вік
   void setage(int age){
     this.age=age;   // вірне привласнення!
   }
}
```

Конфлікт імен, що виникає через затінююче оголошення, досить легко виправити за допомогою ключового слова this або інших конструкцій мови, залежно від обставин. Найбільшою проблемою є те, що компілятор ніяк не повідомляє про такі ситуації, і найскладніше - виявити її за допомогою тестування або контрольного перегляду коду.

"Затемнююче" оголошення (Obscuring)

Може виникнути ситуація, коли *просте ім'я* може бути одночасно розглянуте як ім'я змінної, типу або пакету.

Приведемо приклад, який частково ілюструє такий випадок:

```
import java.awt.*;
public class Obscuring {
   static Point Test = new Point(3,2);
   public static void main (String s[]) {
```

```
      print(Test.x);
    }
}
class Test {
  static int x = -5;
}
```

У методі main() *просте ім'я* Test одночасно позначає ім'я поля класу Obscuring і ім'я іншого типу, що знаходиться в тому ж пакеті, - Test. За допомогою цього імені відбувається звернення до поля x, яке визначене і в класі java.awt.Point і Test.

Результатом цього прикладу стане 3, тобто змінна має вищий пріоритет. У свою чергу, тип має вищий пріоритет, ніж пакет. Таким чином, звернення до доступного в звичайних умовах типу або пакету може виявитися неможливим, якщо є оголошення однойменної змінної або типу, що має вищий пріоритет. Таке оголошення називається таким, що "затемнює" (obscuring).

Ця проблема швидше за все не виникне, якщо слідувати угодам іменування *елементів* мови Java.

Угоди іменування

Для того, щоб код, написаний на Java, було легко читати і зрозуміти не тільки його авторові, але і іншим розробникам, а також для усунення деяких конфліктів імен, пропонуються наступні угоди іменування *елементів* мови Java. Стандартні бібліотеки і класи Java також слідують їм там, де це можливо.

Угоди регулюють іменування наступних конструкцій:

- пакети;
- типи (класи і інтерфейси);
- методи;
- поля;
- поля-константи;
- локальні змінні і параметри методів і ін.

Розглянемо їх послідовно.

Правила побудови імен пакетів вже детально розглядалися в цьому розділі. Ім'я кожного пакету починається з маленької букви і є, як правило, одне недовге слово. Якщо потрібно скласти назву з декількох слів, можна скористатися знаком підкреслення або починати наступне слово з великої букви. Ім'я пакету верхнього рівня зазвичай відповідає доменному імені першого рівня. Назва java і javax (Java extension) зарезервовані компанією Sun для стандартних пакетів Java.

При виникненні ситуації "затемнюючого" оголошення (obscuring) можна змінити ім'я локальної змінної, що не спричинить глобальних змін в коді. Випадок же конфлікту з ім'ям типу не повинен виникати, згідно правилам іменування типів.

Імена типів починаються з великої букви і можуть складатися з декількох слів, кожне наступне слово також починається з великої букви. Звичайно, треба прагнути до того, щоб імена були описовими, такими, що "говорять".

Імена класів, як правило, є іменниками:

Human

Highgreenoak

Arrayindexoutofboundsexception

(Останній приклад - помилка, що виникає при використанні індексу масиву, який виходить за межі допустимого.)

Аналогічно задаються імена інтерфейсів, хоча вони не обов'язково повинні бути іменниками. Часто використовується англійський суфікс "able":

Runnable

Serializable

Cloneable

Проблема "затемнюючого" оголошення (obscuring) для типів зустрічається рідко, оскільки імена пакетів і локальних змінних (параметрів) починаються з маленької букви, а *типів* - з великою.

Імена методів повинні бути дієсловами і позначати дії, які здійснює даний метод. Ім'я повинне починатися з маленької букви, але може складатися з декількох слів, причому кожне наступне слово починається із великої букви. Існує ряд прийнятих назв для методів:

- якщо методи призначені для читання і зміни значення змінній, то їх імена починаються, відповідно, з get і set, наприклад, для змінній size це будуть getsize() і setsize();
- метод, що повертає довжину, називається length(), наприклад, в класі String;
- ім'я методу, який перевіряє булеву умову, починається з is, наприклад, isvisible() у компоненті графічного призначеного для користувача інтерфейсу;
- метод, який перетворить величину у формат F, називається tof(), наприклад, метод tostring(), який приводить будь-який об'єкт до рядка.

Взагалі, рекомендується скрізь, де можливо, називати методи схожим чином, як в стандартних класах Java, щоб вони були зрозумілі всім розробникам.

Поля класу мають імена, записувані в тому ж стилі, що і для методів, починаються з маленької букви, можуть складатися з декількох слів, кожне наступне слово починається із заголовної букви. Імена повинні бути іменниками, наприклад, поле name в класі Human, або size в класі Planet.

Як для полів вирішується проблема "затемнюючим" оголошенням (obscuring), вже обговорювалося.

Поля можуть бути константами, якщо в їх оголошенні є ключове слово final. Їх імена складаються з послідовності слів, скорочень, абревіатур. Записуються вони тільки великими буквами, слова розділяються знаками підкреслення:

PI

Min_value

Max_value

Іноді константи утворюють групу, тоді рекомендується використовувати одне або декілька однакових слів на початку імен:

```
Color_red
Color_green
Color_blue
```

Нарешті, розглянемо імена локальних змінних і параметрів методів, конструкторів і обробників помилок. Вони, як правило, досить короткі, але, проте, повинні бути осмислені. Наприклад, можна використовувати абревіатуру (ім'я cp для посилання на екземпляр класу Colorpoint) або скорочення (buf для buffer).

Поширені однобуквенні скорочення:

```
byte b;
char з;
int i,j,k;
long l;
float f;
double d;
Object про;
String s;
Exception e;   // об'єкт, що представляє
               // помилку в Java
```

Дво - і трьохбуквені імена не повинні співпадати з прийнятими доменними іменами першого рівня Internet-сайтів.

Висновок

У цьому розділі був розглянутий механізм іменування *елементів* мови. Для того, щоб різні частини великої системи не залежали одна від одної, вводиться поняття "*Зона видимості* імені", поза якою необхідно використовувати не *просте*, а *складене ім'я*. Потім були вивчені *елементи* (members), які можуть бути у пакетів і посилальних *типів*. Також розглядався зв'язок термінів "ідентифікатор" (з теми "Лексика") і ім'я.

Потім були розглянуті пакети, які використовуються в Java для створення фізичної і логічної структури класів, а також для точнішого розмежування *зони видимості*. Пакет містить вкладені пакети і *типи*(класи і інтерфейси). Питання про платформову підтримку пакетів привело до розгляду модулів компіляції як текстових файлів, оскільки саме у вигляді файлів і каталогів, як правило, зберігаються і розповсюджуються Java-додатки. Тоді ж вперше було розглянуто питання розмежування доступу, оскільки доступ до модулів компіляції визначається саме платформовою підтримкою, а точніше - операційною системою.

Модуль компіляції складається з трьох основних частин - оголошення пакету, імпорт-вираження і оголошення верхнього рівня. Важливу роль грає безіменний пакет, або пакет за замовчуванням, хоча і він не рекомендується для застосування при створенні великих систем. Були вивчені деталі застосування двох видів імпорт-виражень - імпорт класу і імпорт пакету. Нарешті, було почато розгляд оголошень верхнього рівня (ця тема буде продовжена в розділі, що описує оголошення класів). Пакети, як і інші *елементи* мови, мають певні угоди по іменуванню, покликані полегшити розуміння коду і зменшити можливість виникнення помилок і двозначних ситуацій в програмі.

Опис *зони видимості* для різних *елементів* мови приводить до питання про можливі перекриття такихьобластей і, як наслідок, про конфлікти імен. Розглядаються "затінюючі" і "затемнюючі" оголошення. Для усунення або зменшення можливості виникнення таких ситуацій описуються угоди іменування для всіх *елементів* мови.

ТЕМА. *Оголошення класів. Структура оголошення заголовку класу і його тіла.*

План

1. *Оголошення класів.*
2. *Структура оголошення заголовку класу і його тіла.*
3. *Правила передачі параметрів різних типів в методи. Перевантажені методи.*

Рекомендована література: 1, 2,3, 4.

Теоретичні відомості

Оголошення класів є центральною темою курсу, оскільки будь-яка програма на Java – це набір класів. Оскільки типи є ключовою конструкцією мови, їх структура досить складна, має багато тонкощів. Тому дана тема розділена на дві лекції.

У багатьох мовах існують права доступу, які обмежують можливість використання, наприклад, змінної в класі. Наприклад, легко представити два крайні види прав доступу: це public, коли поле доступне з будь-якої точки програми, і private, коли поле може використовуватися тільки усередині того класу, в якому воно оголошене.

Призначення модифікаторів доступу

Дуже часто права доступу розцінюються як якийсь елемент безпеки коди: мовляв, необхідно захищати класи від "неправильного" використання. Наприклад, якщо в класі Human (людина) є поле age (вік людини), то який-небудь програміст навмисно або по незнанню може привласнити цьому полю від'ємне значення, після чого об'єкт почне працювати неправильно, можуть з'явитися помилки. Для захисту такого поля age необхідно оголосити його private. Це досить поширена точка зору, проте потрібно визнати, що вона далека від істини. Основним сенсом розмежування прав доступу є забезпечення невід'ємної властивості об'єктної моделі – **інкапсуляції,** тобто захована реалізації. Поправний приклад так, щоб він коректно відображав призначення модифікаторів доступу. Отже, хай в класі Human є поле age цілочисельного типу, і щоб всі охочі могли користуватися цим полем, воно оголошується public.

```
public class Human {
  public int age;
}
```

Може вийти так, що цілочисельного типу даних буде вже недостатньо і захочеться змінити тип поля на дріб. Проте якщо просто змінити int на double, незабаром всі розробники, які користувалися класом Human і його полем age, виявлять, що в їх коді з'явилися помилки, тому що поле раптом стало дробом, і в рядках, подібних цим:

```
Human h = getHuman();   // отримуємо посилання
int i=h.age;   // помилка!!
```

виникатиме помилка із-за спроби провести неявним чином звуження примітивного типу.

Виходить, що подібна зміна (загалом, невелике і локальне) зажадає модифікації багато і багатьох класів. Тому внесення його виявиться неприпустимим, невиправданим з погляду кількості зусиль, які необхідно витратити. Тобто, оголосивши один раз поле або *метод* як public, можна опинитися за ситуації, коли щонайменші зміни (імені, типу, характеристик, правил використання) надалі стануть неможливі. Навпаки, якби поле було оголошене як private, а для читання і зміни його значення були б введені додаткові *методи*, ситуація помінялася б в корені:

```
public class Human {
  private int age;
  // метод, що повертає значення age
  public int getAge() {
    return age;
  }
  // метод, що встановлює значення age
  public void setAge(int a){
    age=a;
  }
}
```

В цьому випадку з даним класом могла б працювати безліч програмістів і могла бути створене велика кількість класів, що використовують тип Human, але модифікатор private дає гарантію, що ніхто безпосередньо цим полем не користується і зміна його типу була б зовсім нескладною операцією, пов'язаною із зміною тільки в одному класі. Отримання величини віку виглядало б таким чином:

```
 Human h = getHuman();
int i=h.getAge();   // звернення через метод
```

Розглянемо, як виглядає процес зміни типу поля age:

```
public class Human {

  // поле отримує новий тип double
  private /*int*/ double age;

  // старі методи працюють з округленням
  // значення
```

```
  public int getAge() {
    return (int) Math.round(age);
  }
  public void setAge(int a){
    age=a;
  }
  // додаються нові методи для роботи
  // з типом double

  public double getExactAge() {
    return age;
  }
  public void setExactAge(double a){
  age=a;
  }
}
```

Видно, що старі *методи*, які, можливо, вже застосовуються в багатьох місцях, залишилися без зміни. Точніше, залишився без змін їх зовнішній формат, а внутрішня реалізація ускладнилася. Але така зміна не зажадає ніяких модифікацій решти класів системи. **Приклад використання**

```
Human h = getHuman();
int i=h.getAge();   // коректно
```

залишається вірним, змінна i набуває коректного цілого значення. Проте зміни вводилися для того, щоб можна було працювати з дробовими величинами. Для цього були додані нові *методи* і у всіх місцях, де потрібне точне значення віку, необхідно звертатися до них:

```
Human h = getHuman();
double d=h.getExactAge();
  // точне значення віку
```

Отже, в клас була додана нова можливість, що не зажадала ніяких змін коди.

За рахунок чого була досягнута така гнучкість? Необхідно виділити властивості об'єкту, які буде потрібно майбутнім користувачам цього класу, і зробити їх доступними (в даному випадку, public). Ті ж елементи класу, що містять деталі внутрішньої реалізації логіки класу, бажано приховувати, щоб не утворилися небажані залежності, які можуть стримувати розвиток системи. Цей приклад одночасно ілюструє і інше теоретичне правило написання об'єктів, а саме: в більшості випадків доступ до полів краще реалізовувати через спеціальні *методи* (accessors) для читання (getters) і запису (setters). Тобто само поле розглядається як деталь внутрішньої реалізації. Дійсно, якщо розглядати зовнішній інтерфейс об'єкту як що цілком складається з допустимих дій, то доступними елементами повинні бути тільки *методи*, що реалізовують ці дії. Один з випадків, в якому такий підхід приносить необхідну гнучкість, вже розглянутий. Є і інші міркування. Наприклад, повернемося до питання про коректне використання об'єкту і установку вірних значень полів. Як наслідок, правильне розмежування доступу дозволяє ввести механізми перевірки вхідних значень:

```
public void setAge(int a){
  if (a>=0) {
    age=a;
  }
}
```

В даному прикладі поле age ніколи не прийме некоректне негативне значення. (Недоліком приведеного прикладу є те, що у разі неправильних вхідних даних вони просто ігноруються, немає ніяких повідомлень, що дозволяють дізнатися, що зміни поля віку насправді не відбулося; для повноцінної реалізації *методу* необхідно освоїти роботу з помилками в Java.) Бувають і істотніші зміни логіки класу. Наприклад, дані можна почати зберігати не в полях класу, а в надійнішому сховищі, наприклад, файловій системі або базі даних. В цьому випадку *методи - аксесори* знову змінять свою реалізацію і почнуть звертатися до persistent storage (постійне сховище, наприклад, БД) для читання/запису значень. Якщо доступу до полів класу не було, а відкритими були тільки *методи для* роботи з їх значеннями, то можна змінити код цих *методів*, а зовнішні типи, які використовували даний клас, абсолютно не зміняться, логіка їх роботи залишиться тій же.

Підведемо підсумки. Функціональність класу необхідно розділяти на відкритий інтерфейс, що описує дії, які використовуватимуть зовнішні типи, і на внутрішню реалізацію, яка застосовується тільки усередині самого класу. Зовнішній інтерфейс надалі модифікувати неможливо, або дуже складно, для великих систем, тому його потрібно продумувати особливо ретельно. Деталі внутрішньої реалізації можуть бути змінені на будь-якому етапі, якщо вони не міняють логіку роботи всього класу. Завдяки такому підходу реалізується одна з базових характеристик об’єктної моделі — інкапсуляція, і забезпечується важлива перевага технології ООП — модульність.

Таким чином, модифікатори доступу вводяться не для захисту типу від зовнішнього користувача, а, навпроти, для захисту, або позбавлення, користувача від зайвих залежностей від деталей внутрішньої реалізації. Що ж до неправильного застосування класу, то його творцям потрібно прагнути до того, щоб клас був простий в застосуванні, тоді таких проблем не виникне, адже програміст не почне навмисно писати код, який породжує помилки в його програмі.Звичайно, таке розбиття на зовнішній інтерфейс і внутрішню реалізацію не завжди очевидно, часто умовно. Для полегшення завдання технічних дизайнерів класів в Java введено не два (public і private), а чотири рівні доступу. Розглянемо їх і весь механізм розмежування доступу в Java детальніше.

Розмежування доступу в Java

Рівень доступу елементу мови є статичною властивістю, задається на рівні коди і завжди перевіряється під час компіляції. Спроба звернутися до закритого елементу викличе помилку.У Java модифікатори доступу указуються для:

- типів (класів і інтерфейсів) оголошення верхнього рівня;
- елементів посилальних типів (полів, *методів*, внутрішніх типів);
- конструкторів класів.

Як наслідок, масив також може бути недоступний в тому випадку, якщо недоступний тип, на основі якого він оголошений. Всі чотири рівні доступу мають тільки елементи типів і конструктори. Це:

- public;
- private;
- protected;
- якщо не вказаний жоден з цих трьох типів, то рівень доступу визначається за умовчанням (default).

Перші два з них вже були розглянуті. Останній рівень (доступ за умовчанням) згадувався в минулій лекції – він допускає звернення з того ж пакету, де оголошений і сам цей клас. З цієї причини пакети в Java є не просто набором типів, а більш структурованою одиницею, оскільки типи усередині одного пакету можуть більше взаємодіяти один з одним, чим з типами з інших пакетів. Нарешті, protected дає доступ спадкоємцям класу. Зрозуміло, що спадкоємцям може потрібно доступ до деяких елементів батька, з якими не доводиться мати справи зовнішнім класам. Проте описана структура не дозволяє упорядкувати модифікатори доступу так, щоб кожен наступний строго розширював попередній. Модифікатор protected може бути вказаний для спадкоємця з іншого пакету, а доступ за умовчанням допускає звернення з класів – не наслідків, якщо вони знаходяться в тому ж пакеті. З цієї причини можливості protected були розширені таким чином, що він включає доступ усередині пакету. Отже, модифікатори доступу упорядковуються таким чином (від менш відкритих – до відкритішим):

private
(none) default
protected
public

Ця послідовність буде використана далі при вивченні деталей спадкоємства класів.Тепер розглянемо, які модифікатори доступу можливі для різних елементів мови.

- Пакети доступні завжди, тому у них немає модифікаторів доступу (можна сказати, що всі вони public, тобто будь-який пакет, що існує в системі, може використовуватися з будь-якої точки програми).
- Типи (класи і інтерфейси) верхнього рівня оголошення. При їх оголошенні існує всього дві можливості: вказати модифікатор public або не указувати його. Якщо доступ до типу є public, то це означає, що він доступний з будь-якої точки коди. Якщо ж він не public, то рівень доступу призначається за умовчанням: тип доступний тільки усередині того пакету, де він оголошений.
- Масив має той же рівень доступу, що і тип, на основі якого він оголошений (природно, всі примітивні типи є повністю доступними).
- Елементи і конструктори об’єктних типів. Володіють всіма чотирма можливими значеннями рівня доступу. Всі елементи інтерфейсів є public.

Для типів оголошення верхнього рівня немає необхідності у всіх чотирьох рівнях доступу. Private- типу утворювали б закриту міні-програму, ніхто не міг би їх використовувати. Типи, доступні тільки для спадкоємців, також не були визнані корисними. Розмежування доступу позначаються не тільки на зверненні до елементів об’єктних типів або пакетів (через складене ім’я або пряме звернення), але також при

виклику конструкторів, спадкоємстві, приведенні типів. Імпортувати недоступні типи забороняється.

Перевірка рівня доступу проводиться компілятором. Звернете увагу на наступні приклади:

```
public class Wheel {
  private double radius;
  public double getRadius() {
    return radius;
  }
}
```

Значення поля radius недоступно зовні класу, проте відкритий *метод* getRadius() коректно повертає його. Розглянемо наступні два модулі компіляції:

```
package first;

// Деякий клас Parent
public class Parent {
}
```

```
package first;

// Клас Child успадковується від класу Parent
// але має обмеження доступу за умовчанням
class Child extends Parent {
}

public class Provider {
  public Parent getValue() {
    return new Child();
  }
}
```

До *методу* getValue() класу Provider можна звернутися і з іншого пакету, не тільки з пакету first, оскільки метод оголошений як public. Даний *метод* повертає екземпляр класу Child, який недоступний з інших пакетів. Проте наступний виклик є коректним:

```
 package second;
import first.*;
public class Test {
  public static void main(String s[])
  {
   Provider pr = new Provider();
   Parent p = pr.getValue();
   System.out.println(p.getClass().getName());
   // (Child)p - приведе до помилки компіляції!
  }
```

```
}
```

Результатом буде:

```
first.Child
```

Тобто насправді в класі Test робота йде з екземпляром недоступного класу Child, що можливо, оскільки звернення до нього робиться через відкритий клас Parent. Спроба ж виконати явне приведення викличе помилку. Так, тип об’єкту "вгаданий" вірно, але доступ до закритого типу завжди заборонений.

Наступний приклад:

```
public class Point {
  private int x, y;

  public boolean equals(Object про){
    if (про instanceof Point) {
      Point p = (Point)o;
      return p.x==x && p.y==y;
    }
    return false;
  }
}
```

В даному прикладі оголошується клас Point з двома полями, що описують координати точки. Звернете увагу, що поля повністю закриті – private. Далі спробуємо перевизначити стандартний *метод* equals() так, щоб для аргументів, що є екземплярами класу Point, або його спадкоємців (логіка роботи оператора instanceof), у разі рівності координат поверталося дійсне значення. Звернете увагу на рядок, де робиться порівняння координат, - для цього доводиться звертатися до private-полям іншого об’єкту!

Проте, така дія коректна, оскільки private допускає звернення з будь-якої крапки класу, незалежно від того, до якого саме об’єкту воно проводиться.

Інші приклади розмежування доступу в Java розглядатимуться по ходу курсу.

Оголошення класів

Розглянемо базові можливості оголошення класів. Оголошення класу складається із заголовка і тіла класу.

Заголовок класу

Спочатку указуються модифікатори класу. Модифікатори доступу для класу вже обговорювалися. Допустимим є public, або його відсутність – доступ за умовчанням. Клас може бути оголошений як final. В цьому випадку не допускається створення спадкоємців такого класу. На своїй вітці спадкоємства він є останнім. Класом String і класи-обгортки, наприклад, є final- класи.Після списку модифікаторів указується ключове слово class, а

потім ім'я класу – коректний Java-ідентифікатор. Таким чином, найкоротшим оголошенням класу може бути такий модуль компіляції:

```
class A {
```

Фігурні дужки позначають тіло класу, але про нього пізніше. Вказаний ідентифікатор стає простим ім'ям класу. Повне складене ім'я класу будується з повного складеного імені пакету, в якому він оголошений (якщо це не безіменний пакет), і простого імені класу, розділених крапкою. Область видимості класу, де він може бути доступний по своєму простому імені, – його пакет.

Далі заголовок може містити ключове слово extends, після якого повинне бути вказане ім'я (просте або складене) доступного не –final класа. В цьому випадку оголошуваний клас успадковується від вказаного класу. Якщо вираз extends не застосовується, то клас успадковується безпосередньо від Object. Вираз extends Object допускається і ігнорується.

```
class Parent {
  // = class Parent extends Object {

final class LastChild extends Parent {

  // class WrongChild extends LastChild {
  // помилка!!
```

Спроба розширити final-класс приведе до помилки компіляції.

Якщо в оголошенні класу A вказаний вираз extends B, то клас A називають прямим спадкоємцем класу B.

Клас A вважається спадкоємцем класу B, якщо:

- A є прямим спадкоємцем B;
- існує клас C, який є спадкоємцем B, а A є спадкоємцем C (це правило застосовується рекурсивно).

Таким чином можна прослідкувати ланцюжки спадкоємства на декілька рівнів вгору. Якщо компілятор виявляє, що клас є своїм спадкоємцем, виникає помилка компіляції:

```
// приклад викличе помилку компіляції
class A extends B {
class B extends C {
class C extends A {
// помилка! Клас A став своїм спадкоємцем
```

Далі в заголовку може бути вказане ключове слово implements, за яким повинне слідувати перерахування через кому імен (простих або складених, повторення заборонені) доступних інтерфейсів:

```
public final class String implements
```

```
Serializable, Comparable {
```

В цьому випадку говорять, що клас реалізує перераховані інтерфейси. Як видно з прикладу, клас може реалізовувати будь-яку кількість інтерфейсів. Якщо вираз implements відсутній, то клас дійсно не реалізує ніяких інтерфейсів, тут значень за умовчанням немає.

Далі слідує пара фігурних дужок, які можуть бути порожніми або містити опис тіла класу.

Тіло класу може містити оголошення елементів (members) класу:

- полів;
- внутрішніх типів (класів і інтерфейсів);

і решти допустимих конструкцій:

- конструкторів;
- *ініціалізаторів*
- статичних *ініціалі заторів*.

Елементи класу мають імена і передаються по спадку, неелементи – ні. Для елементів прості імена указуються при оголошенні, складені формуються з імені класу, або імені змінної об'єктного типу, і простого імені елементу. Областю видимості елементів є все оголошення тіла класу. Допускається застосування будь-якого зі всіх чотирьох модифікаторів доступу. Нагадуємо, що угоди по іменуванню класів і їх елементів обговорювалися в минулій лекції. Неелементи не володіють іменами, а тому не можуть бути викликані явно. Їх викликає сама віртуальна машина. Наприклад, конструктор викликається при створенні об'єкту. З тієї ж причини неелементи не володіють модифікаторами доступу. Елементами класу є елементи, описані в оголошенні тіла класу і передані по спадку від класу-батька (окрім Object – єдиного класу, що не має батька) і всіх інтерфейсів, що реалізовуються, за умови достатнього рівня доступу. Таким чином, якщо клас містить елементи з доступом за умовчанням, то його спадкоємці з різних пакетів володітимуть різним набором елементів. Класи з того ж пакету можуть користуватися повним набором елементів, а з інших пакетів – тільки protected і public. private-елементи по спадку не передаються. Поля і *методи* можуть мати однакові імена, оскільки звернення до полів завжди записується без дужок, а до методів – завжди з дужками. Розглянемо всі ці конструкції детальніше.

Оголошення полів починається з перерахування модифікаторів. Можливе застосування будь-якого з трьох модифікаторів доступу, або ніякого зовсім, що означає рівень доступу за умовчанням. Поле може бути оголошене як final, це означає, що воно ініціалізується один раз і більше не мінятиме свого значення. Простий спосіб роботи з final-змінними - ініціалізація при оголошенні:

```
final double PI=3.1415;
```

Також допускається ініціалізація final-полів в кінці кожного конструктора класу. Не обов'язково використовувати для ініціалізації константи компіляції, можливе звернення до різних функцій, наприклад:

```
final long creationTime =
        System.currentTimeMillis();
```

Дане поле зберігатиме час створення об'єкту. Існує ще два спеціальні модифікатори - transient і volatile. Вони будуть розглянуті у відповідних лекціях.

Після списку модифікаторів указується тип поля. Потім йде перерахування одного або декількох імен полів з можливими *ініціалізаторами*:

```
int a;
int b=3, c=b+5, d;
Point p, p1=null, p2=new Point();
```

Імена полів, що повторюються, заборонені. Вказаний ідентифікатор при оголошенні стає простим ім'ям поля. Складене ім'я формується з імені класу або імені змінної об'єктного типу, і простого імені поля. Областю видимості поля є все оголошення тіла класу.

Забороняється використовувати поле в ініціалізації інших полів до його оголошення.

```
int y=x;
int x=3;
```

Проте в останньому поля можна оголошувати і нижче за їх використання:

```
class Point {
  int getX() {return x;}

  int y=getX();
  int x=3;

  public static void main (String s[]) {
    Point p=new Point();
    System.out.println(p.x+", "+p.y);
  }
}
```

Результатом буде:

```
3, 0
```

Даний приклад коректний, але для розуміння його результату необхідно пригадати, що всі поля класу мають значення за умовчанням:

- для числових полів примітивних типів – 0;
- для булевого типу – false;

- для посилальних – null.

Таким чином, при ініціалізації змінної y був використаний результат *методу* getX(), який повернув значення за умовчанням змінної x, тобто 0. Потім змінна x набула значення 3.

Оголошення методів

Оголошення *методу* складається із заголовка і *тіла методу*. Заголовок складається з:

- модифікаторів (доступу зокрема);
- типу повертаного значення або ключового слова void;
- імені *методу*;
- списку аргументів в круглих дужках (аргументів може не бути);
- спеціального throws-выражения.

Заголовок починається з перерахування модифікаторів. Для *методовдоступен* будь-який з трьох можливих модифікаторів доступу. Також допускається використання доступу за умовчанням. Крім того, існує модифікатор final, який говорить про те, що такий *метод* не можна перевизначати в спадкоємцях. Можна вважати, що всеметоды final-класса, а також всі private-методи будь-якого класу, є final. Також підтримується модифікатор native. *Метод*, оголошений з таким модифікатором, не має реалізації на Java. Він повинен бути написаний на іншій мові (C/C++, Fortran і так далі) і доданий в систему у вигляді завантажуваної динамічної бібліотеки (наприклад, DLL для Windows). Існує **спеціальна специфікація JNI** (Java Native Interface), що описує правила створення і використання native-методів. Така можливість для Java необхідна, оскільки багато компаній мають обширні програмні бібліотеки, написані на старіших мовах. Їх було б дуже трудомістке і неефективно переписувати на Java, тому необхідна можливість підключати їх у такому вигляді, в якому вони є. Безумовно, при цьому Java-приложения втрачають цілий ряд своїх переваг, таких, як переносимість, безпека та інші. Тому застосовувати JNI слід тільки у разі крайньої необхідності. Ця специфікація накладає вимоги на імена процедур в зовнішніх бібліотеках (вона складає їх з імені пакету, класу і самого native-метода), а оскільки бібліотеки міняти, як правило, дуже незручно, часто пишуть спеціальні бібліотеки-"обертки", до яких звертаються Java-класи через JNI, а вони самі звертаються до цільових модулів.

Після перерахування модифікаторів указується ім’я (просте або складене) типу повертаного значення; це може бути як примітивний, так і об’єктний тип. Якщо *метод* не повертає ніякого значення, указується ключове слово void. Потім визначається ім’я *методу*. Вказаний ідентифікатор при оголошенні стає простим ім’ям *методу*. Складене ім’я формується з імені класу або імені змінної об’єктного типу і простого імені *методу*. Областю видимості *методу* є все оголошення тіла класу.

Аргументи *методу* перераховуються через кому. Для кожного указується спочатку тип, потім ім’я параметра. На відміну від оголошення змінній тут забороняється указувати два імена для одного типу:

// void calc (double x, y); - помилка!

```
void calc (double x, double y);
```

Якщо аргументи відсутні, указуються порожні круглі дужки. Однойменні параметри заборонені. Створення локальних змінних в *методі*, з іменами, співпадаючими з іменами параметрів, заборонене. Для кожного аргументу можна ввести ключове слово final перед вказівкою його типу. В цьому випадку такий параметр не може міняти свого значення в *тілі методу* (тобто брати участь в операції привласнення як лівий операнд).

```
public void process(int x, final double y){
  x=x*x+Math.sqrt(x);

  // y=Math.sin(x); - так писати не можна
  // оскільки y - final!
}
```

Важливим поняттям є *сигнатура* (signature) *методу*. *Сигнатура* визначається ім'ям *методу* і його аргументами (кількістю, типом, порядком проходження). Якщо для полів забороняється збіг імен, то для *методів* в класі заборонено створення двох *методів* з однаковими *сигнатурами*.

Наприклад

```
class Point {
  void get() {
  void get(int x){
  void get(int x, double y){
  void get(double x, int y){
}
```

Такий клас оголошений коректно. Наступні пари *методів* в одному класі один з одним несумісні:

```
void get() {
int get() {

void get(int x){
void get(int y){

public int get() {
private int get() {
```

У першому випадку *методи* відрізняються типом зворотнього значення, яке, проте, не входить в визначення сигнатури. Отже, це два *методи* з однаковими *сигнатурами* і вони не можуть одночасно з'явитися в оголошенні тіла класу. Можна скласти приклад, який створив би нерозв'язну проблему для компілятора, якби був допустимий:

```
// приклад викличе помилку компіляції
class Test {
  int get() {
    return 5;
```

```
  }
  Point get() {
    return new Point(3,5);
  }

  void print(int x){
    System.out.println("it's int! "+x);
  }
  void print(Point p){
    System.out.println("it's Point! "+p.x+
                       ", "+p.y);
  }

  public static void main (String s[]) {
    Test t = new Test();
    t.print(t.get());   // Двозначність!
  }
}
```

У класі визначена заборонена пара *методів* get() з однаковими *сигнатурами* і різними повертаними значеннями. Звернемося до виділеного рядка в *методі* main, де виникає конфліктна ситуація, з якою компілятор не може справитися. Визначено два *методи* print() (у них різні аргументи, а значить, і *сигнатури*, тобто це допустимі *методи*), і щоб розібратися, який з них буде викликаний, потрібно знати точний тип повертаного значення *методу* get(), що неможливе. На основі цього прикладу можна зрозуміти, як складено поняття *сигнатури*. Дійсно, при виклику указується ім'я *методу* і перераховуються його аргументи, причому компілятор завжди може визначити їх тип. Якраз ці поняття і складають *сигнатуру*, і вимога її унікальності дозволяє компілятору завжди однозначно визначити, який *метод* буде викликаний. Так само в попередньому прикладі друга пара *методів* розрізняється ім'ям аргументів, які також не входять у визначення *сигнатури* і не дозволяють визначити, який з двох *методів* повинен бути викликаний. Аналогічно, третя пара розрізняється лише модифікаторами доступу, що також неприпустимо.

Нарешті, завершує *заголовок методу* throws-выражение. Воно застосовується для коректної роботи з помилками в Java і буде детально розглянуто у відповідній лекції.

Приклад оголошення *методу*:

```
public final java.awt.Point
  createPositivePoint(int x, int y)
  throws IllegalArgumentException
{
  return (x>0 && y>0) ?
         new Point(x, y): null;
}
```

Далі, після *заголовка методу* слідує *тіло методу*. Воно може бути порожнім і тоді записується одним символом "крапка з комою". Native-методи завжди мають тільки порожнє тіло, оскільки справжня реалізація написана на іншій мові. Звичайні

ж *методи* мають непорожнє тіло, яке описується у фігурних дужках, що показане в численних прикладах в цій і інших лекціях. Якщо поточна реалізація *методу* не виконує ніяких дій, тіло все одно повинне описуватися парою порожніх фігурних дужок:

```
public void empty() {
```

Якщо в заголовку методу вказаний тип зворотнього значення, а не void, то в *тілі методу* обов'язково повинен зустрічатися return-вираз. При цьому компілятор проводить аналіз структури *методу*, щоб гарантувати, що при будь-яких операторах галуження зворотнє значення згенерує. Наприклад, наступний приклад є некоректним:

```
// приклад викличе помилку компіляції
public int get() {
  if (condition) {
    return 5;
  }
}
```

Видно, що хоча *тіло методу* містить return-вираз, проте не при будь-якому розвитку подій зворотнє значення згенерує. А ось такий приклад є вірним:

```
public int get() {
  if (condition) {
    return 5;
  } else {
    return 3;
  }
}
```

Звичайно, значення, вказане після слова return, повинне бути сумісне за типом з оголошеним зворотнім значенням (це поняття детально розглядається в лекції 7).

У *методі* без зворотнього значення (вказане void) також можна використовувати вираз return без яких-небудь аргументів. Його можна вказати в будь-якому місці *методу* і в цій крапці виконання *методу* буде завершено:

```
public void calculate(int x, int y){
  if (x<=0 || y<=0) {
    return;   // некоректні вхідні
              // значення, вихід з методу
  }
  ...   // основні обчислення
}
```

Виразів return (з параметром або без для *методів* с/без зворотнього значення) в тілі одного *методу* може бути скільки завгодно. Проте слід пам'ятати, що безліч точок виходу в одному *методі* може помітно ускладнити розуміння логіки його роботи.

Оголошення конструкторів

Формат оголошення конструкторів схожий на спрощене оголошення *методів*. Також виділяють заголовок і тіло конструктора. Заголовок полягає, по-перше, з модифікаторів доступу (ніякі інші модифікатори недопустимі). По-друге, указується ім'я класу, яке можна розцінювати двояко. Можна вважати, що ім'я конструктора співпадає з ім'ям класу. А можна розглядати конструктор як безіменний, а ім'я класу – як тип зворотнього значення, адже конструктор може породити тільки об'єкт класу, в якому він оголошений. Це винятково справа смаку, оскільки на форматі оголошення ніяк не позначається:

```
public class Human {
  private int age;

  protected Human(int a){
    age=a;
  }

  public Human(String name, Human mother
          Human father) {
    age=0;
  }
}
```

Як видно з прикладів, далі слідує перерахування вхідних аргументів по тих же правилах, що і для *методів*. Завершує заголовок конструктора throws-вираз (у прикладі не використано, див. лекцію 10 "Виключень"). Воно має особливу важливість для конструкторів, оскільки згенерувати помилку – це для конструктора єдиний спосіб не створювати об'єкт. Якщо конструктор виконався без помилок, то об'єкт гарантовано створюється. Тіло конструктора порожнім бути не може і тому завжди описується у фігурних дужках (для простих реалізацій дужки можуть бути порожніми). У відсутність імені (або через те, що у всіх конструкторів однакове ім'я, співпадаюче з ім'ям класу) *сигнатура* конструктора визначається тільки набором вхідних параметрів по тих же правилах, що і для *методів*. Аналогічно, в одному класі допускається будь-яка кількість конструкторів, якщо у них різні *сигнатури*. Тіло конструктора може містити будь-яку кількість return-вираз без аргументів. Якщо процес виконання дійде до такого виразу, то на цьому місці виконання конструктора буде завершено. Проте логіка роботи конструкторів має і деякі важливі особливості. Оскільки при їх виклику здійснюється створення і ініціалізація об'єкту, стає зрозуміло, що такий процес не може відбуватися без звернення до конструкторів всіх батьківських класів. Тому вводиться обов'язкове правило – першим рядком в конструкторі повинне бути звернення до батьківського класу, яке записується за допомогою ключового слова super.

```
public class Parent {
  private int x, y;

  public Parent() {
    x=y=0;
  }
  public Parent(int newx, int newy){
```

```
    x=newx;
    y=newy;
  }
}
public class Child extends Parent {
  public Child() {
    super();
  }
  public Child(int newx, int newy){
    super(newx, newy);
  }
}
```

Як видно, звернення до батьківського конструктора записується за допомогою super, за яким йде перерахування аргументів. Цей набір визначає, який з батьківських конструкторів буде використаний. У приведеному прикладі в кожному класі є по два конструктори і кожен конструктор в спадкоємцеві звертається до аналогічного в батьку (це задоволено поширений, але, звичайно, не обов'язковий спосіб).

Прослідкуємо в думках весь алгоритм створення об'єкту. Він починається при виконання виразу з ключовим словом new, за яким слідує ім'я класу, від якого породжуватиметься об'єкт, і набір аргументів для його конструктора. По цьому набору визначається, який саме конструктор буде використаний, і відбувається його виклик. Перший рядок його тіла містить виклик батьківського конструктора. У свою чергу, перший рядок тіла конструктора батька міститиме виклик до його батька, і так далі. Сходження по дереву спадкоємства закінчується, очевидно, на класі Object, у якого є єдиний конструктор без параметрів. Його тіло порожнє (записується парою порожніх фігурних дужок), проте можна вважати, що саме у цей момент JVM породжує об'єкт і далі починається процес його ініціалізації. Виконання починає зворотний шлях вниз по дереву спадкоємства. У самого верхнього батька, прямого спадкоємця від Object, відбувається продовження виконання конструктора з другого рядка. Коли він буде повністю виконаний, необхідно перейти до наступного батька, на один рівень спадкоємства вниз, і завершити виконання його конструктора, і так далі. Нарешті, можна буде повернутися до конструктора початкового класу, який був викликаний за допомогою new, і також продовжити його виконання з другого рядка. Після його завершення об'єкт вважається повністю створеним, виконання виразу new буде закінчено, а як результат буде повернено посилання на породжений об'єкт.

Проілюструємо цей алгоритм наступним прикладом:

```
public class GraphicElement {
  private int x, y;   // положення на екрані
  public GraphicElement(int nx, int ny){
    super();   // звернення до конструктора
            // батька Object
    System.out.println("GraphicElement");
    x=nx;
    y=ny;
    }
}
```

```
public class Square extends GraphicElement {
  private int side;
  public Square(int x, int y, int nside){
    super(x, y);
    System.out.println("Square");
      side=nside;
  }
}
public class SmallColorSquare extends Square {
  private Color color;

  public SmallColorSquare(int x, int y
                    Color з) {
    super(x, y, 5);
    System.out.println("SmallColorSquare");
    color=c;
  }
}
```

Після виконання виразу створення об'єкту на екрані з'явиться наступне:

```
GraphicElement
Square
SmallColorSquare
```

Вираз super може стояти тільки на першому рядку конструктора. Часто можна побачити конструктори взагалі без такого виразу. В цьому випадку компілятор першим рядком за умовчанням додає виклик батьківського конструктора без параметрів (super()). Якщо у батьківського класу такого конструктора немає, вираз super обов'язково повинен бути записане явно (і саме на першому рядку), оскільки необхідна передача вхідних параметрів. Нагадаємо, що, по-перше, конструктори не мають імені і їх не можна викликати явно, тільки через вираз створення об'єкту. Крім того, конструктори не передаються по спадку. Тобто, якщо в батьківському класі оголошено п'ять різних корисних конструкторів і потрібний, щоб клас-спадкоємець мав аналогічний набір, необхідно все їх описати наново. Клас обов'язково повинен мати конструктор, інакше неможливо породжувати об'єкти ні від нього, ні від його спадкоємців. Тому якщо в класі не оголошений жоден конструктор, компілятор додає один за умовчанням. Це public-конструктор без параметрів і з тілом, описаним парою порожніх фігурних дужок. З цього виходить, що таке можливе тільки для класів, у батьків яких оголошений конструктор без параметрів, інакше виникне помилка компіляції. Звернете увагу, що якщо потім в такий клас додається конструктор (не важливо, з параметрами або без), то конструктор за умовчанням більше не вставляється:

```
/*
  * Цей клас має один конструктор.
  */
public class One {
  // Буде створений конструктор за умовчанням
  // Батьківський клас Object має
  // конструктор без параметрів.
```

```
}

/*
  * Цей клас має один конструктор.
  */
public class Two {
  // Єдиний конструктор класу Two.
  // Вираз new Two() помилково!
  public Two(int x){
  }
}
/*
  * Цей клас має два конструктори.
  */
public class Three extends Two {
  public Three() {
    super(1);   // вираз super потрібний
  }
  public Three(int x){
    super(x);   // вираз super потрібний
  }
}
```

Якщо клас має більш за один конструктор, допускається в першому рядку деяких з них указувати не super, athis – вираз, що викликає інший конструктор цього ж класу.

Розглянемо наступний приклад:

```
public class Vector {
  private int vx, vy;
  protected double length;

  public Vector(int x, int y){
    super();
    vx=x;
    vy=y;
    length=Math.sqrt(vx*vx+vy*vy);
  }

  public Vector(int x1, int y1
              int x2, int y2) {
    super();
    vx=x2-x1;
    vy=y2-y1;
    length=Math.sqrt(vx*vx+vy*vy);
  }
}
```

Видно, що обидва конструктори здійснюють практично ідентичні дії, тому можна застосувати компактніший вид запису:

```
public class Vector {
  private int vx, vy;
  protected double length;

  public Vector(int x, int y){
    super();
    vx=x;
    vy=y;
    length=Math.sqrt(vx*vx+vy*vy);
  }

  public Vector(int x1, int y1
            int x2, int y2) {
    this(x2-x1, y2-y1);
  }
}
```

Великою гідністю такого *методу* запису є те, що вдалося уникнути дублювання ідентичної коди. Наприклад, якщо процес ініціалізації об'єктів цього класу подовжиться на один крок (скажімо, додасться перевірка довжини на нуль), то таку зміну треба буде внести тільки до першого конструктора. Такий підхід допомагає уникнути випадкових помилок, оскільки зникає необхідність тиражувати зміни в декількох місцях. Зрозуміло, таке звернення до конструкторів свого класу не повинне приводити до зациклень, інакше буде видана помилка компіляції. Ланцюжок this повинен у результаті приводити до super, який повинен бути присутнім (явно або неявно) хоч би в одному з конструкторів. Після того, як відпрацюють конструктори всіх батьківських класів, буде продовжено виконання кожного конструктора, залученого в процес створення об'єкту.

```
public class Test {
  public Test() {
    System.out.println("Test()");
  }

  public Test(int x){
    this();
    System.out.println("Test(int x)");
  }
}
```

Після виконання виразу new Test(0) на консолі з'явиться:

```
Test()
Test(int x)
```

На закінчення розглянемо застосування модифікаторів доступу для конструкторів. Може викликати здивування можливість оголошувати конструктори як private. Адже вони потрібні для генерації об'єктів, а до таких конструкторів ні у кого не буде доступу. Проте у ряді випадків модифікатор private може бути корисний. Наприклад:

- private-конструктор може містити дії, що ініціалізували, а решта конструкторів використовуватиме його за допомогою this, причому пряме звернення до цього конструктора по якихось причинах небажано;
- заборона на створення об'єктів цього класу, наприклад, неможливо створити екземпляр класу Math;
- реалізація спеціального шаблону проектування з ООП Singleton, для роботи якого потрібно контролювати створення об'єктів, що неможливе у разі наявності не-private конструкторів.

Ініціалізатори

Нарешті, останньою допустимою конструкцією в тілі класу є оголошення *ініціалізаторів*. Записуються об'єктні *ініціалізатори* дуже просто – усередині фігурних дужок.

```
public class Test {
  private int x, y, z;

  // ініціалізатор об'єкту
  {
    x=3;
    if (x>0)
      y=4;
    z=Math.max(x, y);
  }
}
```

Ініціалізатори не мають імен, виконуються при створенні об'єктів, не можуть бути викликані явно, не передаються по спадку (хоча, звичайно, *ініціалізатори* в батьківському класі продовжують виконуватися при створенні об'єкту класу-спадкоємця). Було вказано вже три види коди, що ініціалізувала, в класах – конструктори, *ініціалізатори* змінних, а зараз додалися об'єктні *ініціалізатори*. Необхідно розібратися, в якій послідовності що виконується, зокрема при спадкоємстві. При створенні екземпляра класу викликаний конструктор виконується таким чином:

- якщо першим рядком йде звернення до конструктора батьківського класу (явне або додане компілятором за умовчанням), то цей конструктор виконується;
- у разі успішного виконання викликаються всі *ініціалізатори* полів і об'єкту в тому порядку, в якому вони оголошені в тілі класу;
- якщо першим рядком йде звернення до іншого конструктора цього ж класу, то він викликається. Повторне виконання *ініціалізаторів* не проводиться.

Другий пункт має ряд важливих следствий. По-перше, з нього виходить, що в *ініціалізаторах* не можна використовувати змінні класу, якщо їх оголошення записане пізнішим. По-друге, тепер можна сформулювати найбільш гнучкий підхід до ініціалізації final-полів. Головна вимога – щоб такі поля були проініціалізовані рівно один раз. Це можна забезпечити в наступних випадках:

- якщо ініціалізувати поле при оголошенні;

- якщо ініціалізувати поле тільки один раз в *ініціалізаторі* об'єкту (він повинен бути записаний після оголошення поля);
- якщо ініціалізувати поле тільки один раз в кожному конструкторі, в першому рядку якого коштує явне або неявне звернення до конструктора батька. Конструктор, в першому рядку якого стоїть this, не може і не повинен ініціалізувати final-поле, оскільки ланцюжок this-виклик приведе до конструктора csuper, в якому ця ініціалізація обов'язково присутня.

Для ілюстрації порядку виконання конструкцій, що ініціалізували, розглянемо наступний приклад:

```
public class Test {
  {
    System.out.println("initializer");
  }
  int x, y=getY();
  final int z;
  {
    System.out.println("initializer2");
  }
  private int getY() {
    System.out.println("getY() "+z);
    return z;
  }
  public Test() {
    System.out.println("Test()");
    z=3;
  }
  public Test(int x){
    this();
    System.out.println("Test(int)");
    // z=4; - не можна! final-поле вже
    // ініціалізувало
  }
}
```

Після виконання виразу new Test() на консолі з'явиться:

```
initializer
getY() 0
initializer2
Test()
```

Звернете увагу, що для ініціалізації поля y викликається *метод* getY(), який повертає значення final-поля z, яке ще не ініціалізувало. Тому у результаті поле y набуде значення за умовчанням 0, а потім поле z отримає постійне значення 3, яке ніколи вже не зміниться.

Після виконання виразу new Test(3) на консолі з'явиться:

```
initializer
```

```
getY() 0
initializer2
Test()
Test(int)
```

Додаткові властивості класів

Метод main

Отже, віртуальна машина реалізується додатком операційної системи і запускається по звичайних правилах. Програма, написана на Java, є набором класів. Зрозуміло, що потрібна якась вхідна крапка, з якою повинне починатися виконання додатку. Такою вхідною крапкою, по аналогії з мовами C/C++, є *метод* main(). Приклад його оголошення:

```
public static void main(String[] args) { }
```

Модифікатор static в цій лекції не розглядався і буде вивчений пізніше. Він дозволяє викликати *метод* main(), не створюючи об'єктів. *Метод* не повертає ніякого значення, хоча в C є можливість вказати код повернення з програми. У Java для цієї мети існує *метод System.exit(),* який закриває віртуальну машину і має аргумент типу int. Аргументом *методу* main() є масив рядків. Він заповнюється додатковими параметрами, які були вказані при виклику *методу*.

```
package test.first;
public class Test {
  public static void main(String[] args){
    for (int i=0; i<args.length; i++) {
      System.out.print(args[i]+" ");
    }
    System.out.println();
  }
}
```

Для виклику програми віртуальній машині передається як параметр ім'я класу, у якого об'явлений метод main(). Оскільки це ім'я класу, а не ім'я файлу, то не повинно вказуватися ніякого розширення (.class або .java) і розташування класу записується через крапку (роздільник імен пакетів), а не за допомогою файлового роздільника. Компілятору ж, навпаки, передається ім'я і шлях до файлу. Якщо приведений вище модуль компіляції збережений у файлі Test.java, який лежить в каталозі test\first, то виклик компілятора записується таким чином:

```
javac test\first\Test.java
```

А виклик віртуальної машини:

```
java test.first.Test
```

Щоб проілюструвати роботу з параметрами, змінимо рядок запуску додатку:

```
java test.first.Test Hello, World!
```

Результатом роботи програми буде:

```
Hello, World!
```

Параметри методів

Для кращого розуміння роботи з параметрами *методів* в Java необхідно розглянути декілька питань.

Як передаються аргументи в *методи* – за значенням або по посиланню? З погляду програми питання формулюється, наприклад, таким чином. Хай є змінна і вона як аргумент передається в деякий *метод*. Чи можуть відбутися які-небудь зміни з цією змінною після завершення роботи *методу*?

```
int x=3;
process(x);
print(x);
```

Припустимо, використовуваний *метод* оголошений таким чином:

```
public void process(int x){
  x=5;
}
```

Яке значення з'явиться на консолі після виконання прикладу? Щоб відповісти на це питання, необхідно пригадати, як змінні різних типів зберігають свої значення в Java.

Нагадаємо, що примітивні змінні є дійсними сховищами своїх значень і зміна значення одній змінній ніколи не позначиться на значенні інший. Параметр *методу* process(), хоч і має таке ж ім'я x, насправді є повноцінним сховищем цілочисельної величини. А тому привласнення йому значення 5 не позначиться на зовнішніх змінних. Тобто результатом прикладу буде 3 і аргументи примітивного типу передаються в *методи* за значенням. Єдиний спосіб змінити таку змінну в результаті роботи методу– повертати потрібні величини з *методу* і використовувати їх при привласненні:

```
public int doubler(int x){
  return x+x;
}

public void test() {
  int x=3;
  x=doubler(x);
}
```

Перейдемо до посилальних типів.

```
public void process(Point p){
```

```
  p.x=3;
}

public void test() {
  Point p = new Point(1,2);
  process(p);
  print(p.x);
}
```

Посилальна змінна зберігає посилання на об'єкт, що знаходиться в пам'яті віртуальної машини. Тому аргумент методу process() матиме як значення те ж саме посилання і, отже, посилатися на той же самий об'єкт. Зміни стани об'єкту, здійснені за допомогою одного посилання, завжди видно при зверненні до цього об'єкту за допомогою іншої. Тому результатом прикладу буде значення 3. Об'єктні значення передаються в Java по посиланню. Проте якщо змінювати не стан об'єкту, а саме посилання, то результат буде іншим:

```
public void process(Point p){
  p = new Point(4,5);
}
public void test() {
  Point p = new Point(1,2);
  process(p);
  print(p.x);
}
```

В даному прикладі аргумент *методу* process() після привласнення починає посилатися на інший об'єкт, ніж початкова змінна p, а значить, результатом прикладу стане значення 1. Можна сказати, що посилальні величини передаються за значенням, але значенням є саме посилання на об'єкт. Тепер можна уточнити, що означає можливість оголошувати параметри *методів* і конструкторів як final. Оскільки зміни значень параметрів (але не об'єктів, на які вони посилаються) ніяк не позначаються на змінних поза *методом*, модифікатор final говорить лише про те, що значення цього параметра не мінятиметься впродовж роботи *методу*. Зрозуміло, для аргументу final Point p вираз p.x=5 є допустимим (забороняється p=new Point(5, 5)).

Переобтяжені методи

Переобтяженими (overloaded) методами називаються *методи* одного класу з однаковими іменами. *Сигнатури у* них повинні бути різними і відмінність може бути тільки в наборі аргументів. Якщо в класі параметри *переобтяжених методів* помітно розрізняються: наприклад, у одного *методу* один параметр, у іншого – два, то для Java це абсолютно незалежні *методи* і збіг їх імен може служити тільки для підвищення наочності роботи класу. Кожен виклик, залежно від кількості параметрів, однозначно адресується тому або іншому *методу*. Проте якщо кількість параметрів однакова, а типи їх розрізняються трохи, при виклику може скластися подвійна ситуація, коли декілька *переобтяжених методів* однакові добре підходять для використання. Наприклад, якщо оголошені типи Parent і Child, де Child розширює Parent, то для наступних двох *методів*:

```
void process(Parent p, Child з){
void process(Child з, Parent p){
```

можна сказати, що вони допустимі, їх *сигнатури* розрізняються. Проте при виклику

```
process(new Child(), new Child());
```

виявляється, що обидва *методи* однаково годяться для використання. Інший приклад, *методи*:

```
process(Object про){
process(String s){
```

і приклади викликів:

```
process(new Object());
process(new Point(4,5));
process("abc");
```

Очевидно, що для перших двох викликів підходить тільки перший *метод*, і саме він буде викликаний. Для останнього ж виклику підходять обидва *переобтяжені методи*, проте клас String є більш "специфічним", або вузьким, чим клас Object. Дійсно, значення типу String можна передавати як аргументи типу Object, зворотне ж невірно. Компілятор спробує відшукати найбільш специфічний *метод*, відповідний для вказаних параметрів, і викличе саме його. Тому при третьому виклику буде використаний другий *метод*.

Проте для попереднього прикладу такий підхід не дає однозначної відповіді. Обидва *методи* однаково специфічні для вказаного виклику, тому виникне помилка компіляції. Необхідно, використовуючи явне приведення, вказати компілятору, який *метод* слід застосувати:

```
process((Parent)(new Child()), new Child());
// або
process(new Child(),(Parent)(new Child()));
```

Це вірно і у разі використання значення null:

```
process((Parent)null, null);
// або
process(null,(Parent) null);
```

Висновок

У цій лекції почався розгляд ключової конструкції мови Java – оголошення класу.

Перша тема присвячена засобам розмежування доступу. Головне питання – для чого цей механізм вводиться в практично кожній сучасній мові високого рівня. Необхідно розуміти, що він призначений не для забезпечення "безпеки" або "захисту" об’єкту від

якихось неправильних дій. Найважливіше завдання – розділити зовнішній інтерфейс класу і деталі його реалізації з тим, щоб надалі скористатися такими перевагами ООП, як інкапсуляція і модульність.

Потім були розглянуті всі чотири модифікатори доступу, а також можливість їх застосування для різних елементів мови. Перевірка рівня доступу виконується вже під час компіляції і забороняє лише явне використання типів. Наприклад, з ними все ж таки можна працювати через їх відкритіших спадкоємців.

Оголошення класу складається із заголовка і тіла класу. Формат заголовка був детально описаний. Для вивчення тіла класу необхідно пригадати поняття елементів (members) класу. Ними можуть бути поля, *методи* і внутрішні типи. Для *методів* важливим поняттям є *сигнатура*.

Крім того, в тілі класу оголошуються конструктори і *ініціалізатори*. Оскільки вони не є елементами, до них не можна звернутися явно, вони викликаються самою віртуальною машиною. Також конструктори і *ініціалізатори не* передаються по спадку.

Додатково був розглянутий *метод* main, який викликається при старті віртуальної машини. Далі описуються тонкощі, що виникають при передачі параметрів, і пов’язане з цим питання про *переобтяжені методи*.

Класи Java ми продовжимо розглядати в наступних лекціях.

ТЕМА. *Перетворення типів. Види перетворень та ситуації застосування.*

План

1. *Правила роботи з типами даних в JAVA.*
2. *Види перетворень типів даних.*
3. *Ситуації застосування перетворень.*

Рекомендована література: 1, 2, 4, 5.

Теоретичні відомості

Java є мовою, що строго типізується, а це означає, що кожен вираз і кожна змінна має строго певний тип вже на момент компіляції. Тип встановлюється на основі структури вживаних виразів і типів літералів, змінних і методів, використовуваних в цих виразах.

Наприклад:

```
long a=3;
a = 5+’A’+a;
print("a="+Math.round(a/2F));
```

Розглянемо, як в даному прикладі компілятор встановлює тип кожного виразу і які перетворення (conversion) типів необхідно здійснити при кожній дії.

- У першому рядку літерал 3 має тип за умовчанням, тобто int. При привласненні цього значення змінній типу long необхідно провести перетворення.
- У другому рядку спочатку проводиться складання значень типу int і char. Другий аргумент буде перетворений так, щоб операція проводилася з точністю в 32 бита. Другий оператор складання знову зажадає перетворення, оскільки наявність змінної a збільшує точність до 64 битий.
- У третьому рядку спочатку буде виконана операція ділення, для чого значення long треба буде привести до типу float, оскільки другий операнд - дробовий літерал. Результат буде переданий в метод Math.round, який проведе математичне округлення і поверне цілочисельний результат типу int. Це значення необхідно перетворити в текст, щоб здійснити подальшу конкатенацію рядків. Як буде показано нижче, ця операція проводиться в два етапи - спочатку простій тип приводиться до об’єктного классу-"обертке" (в даному випадку int до Integer), а потім у отриманого об’єкту викликається метод toString(), що дає перетворення до рядка.

Даний приклад показує, що навіть прості рядки можуть містити численні перетворення, часто непомітні для розробника. Часто бувають і такі випадки, коли

програмістові необхідно явно змінити тип деякого виразу або змінної, наприклад, щоб скористатися відповідним методом або конструктором.

Пригадаємо вже розглянутий приклад:

```
int b=1;
byte c=(byte) -b;
int i=c;
```

Тут в другому рядку необхідно провести явне перетворення, щоб привласнити значення типу intпеременной типу byte. У третьому ж рядку зворотне приведення проводиться автоматично, неявним для розробника чином. Розглянемо спочатку, які переходи між різними типами можна здійснити.

Види перетворень. У Java передбачено сім видів перетворень:

- тотожне (identity);
- розширення примітивного типу (widening primitive);
- звуження примітивного типу (narrowing primitive);
- розширення об'єктного типу (widening reference);
- звуження об'єктного типу (narrowing reference);
- перетворення до рядка (String);
- заборонені перетворення (forbidden).

Розглянемо їх окремо.

Тотожне перетворення. Найпростішим є тотожне перетворення. У Java перетворення виразу будь-якого типу до точно такому ж типу завжди допустимо і успішно виконується.

Навіщо потрібне тотожне приведення? Є дві причини для того, щоб виділити таке перетворення в особливий вигляд. По-перше, з теоретичної точки зору тепер можна стверджувати, що будь-який тип в Java може брати участь в перетворенні, хоч би в тотожному. Наприклад, примітивний тип boolean не можна привести ні до якого іншому типу, окрім нього самого. По-друге, іноді в Java можуть зустрічатися такі вирази, як довгий послідовний виклик методів:

```
print(getCity().getStreet().getHouse().getFlat().getRoom());
```

При виконання такого виразу спочатку викликається перший метод getCity(). Можна припустити, що повертаним значенням буде об'єкт класу City. У цього об'єкту далі буде викликаний наступний методgetStreet(). Щоб дізнатися, значення якого типу він поверне, необхідно подивитися опис класу City. У цього значення буде викликаний наступний метод (getHouse()), і так далі. Щоб дізнатися результуючий тип всього виразу, необхідно проглянути опис кожного методу і класу. Компілятор без зусиль справиться з таким завданням, проте розробникові буде нелегко прослідкувати весь ланцюжок. В цьому випадку можна скористатися тотожним перетворенням, виконавши приведення до точно такому ж типу. Це нічого не змінить в структурі програми, але значно полегшить читання коди:

```
print((MyFlatImpl)(getCity().getStreet().getHouse().getFlat()));
```

Перетворення примітивних типів (розширення і звуження)

Очевидно, що наступні чотири види приведень легко представляються у вигляді таблицы 7.1.

Таблиця 7.1. Види перетворень.	
простій тип, розширення	посилальний тип, розширення
простій тип, звуження	посилальний тип, звуження

Для простих типів розширення означає, що здійснюється перехід від менш ємкого типу до більш ємкому. Наприклад, від типу byte (довжина 1 байт) до типу int (довжина 4 байти). Такі перетворення безпечні в тому сенсі, що новий тип завжди гарантовано вміщає в себе всі дані, які зберігалися в старому типі, і таким чином не відбувається втрати даних. Саме тому компілятор здійснює його самого, непомітно для розробника:

```
byte b=3;
int a=b;
```

У останньому рядку значення змінної b типу byte буде перетворено до типу змінної a (тобто, int) автоматично, ніяких спеціальних дій для цього робити не потрібно.

Наступні 19 перетворень є такими, що розширюють:

- від byte до short, int, long, float, double
- від short до int, long, float, double
- від char до int, long, float, double
- від int до long, float, double
- від long до float, double
- від float до double

Звернете увагу, що не можна провести перетворення до типу char від типів меншої або рівнішої довжини (byte,short), або, навпаки, до short від char без втрати даних. Це пов'язано з тим, що char, на відміну від решти цілочисельних типів, є беззнаковим. Проте, слід пам'ятати, що навіть при розширенні дані все-таки можуть бути в особливих випадках спотворені. Вони вже розглядалися в попередній лекції, це приведення значень int до типу float і приведення значень типу long до типу float або double. Хоча ці дробові типи вміщають набагато більші числа, чим відповідні цілі, але у них менше значущих розрядів.

Повторюваний цей приклад:

```
long a=111111111111L;
float f = a;
a = (long) f;
print(a);
```

Результатом буде:

```
111111110656
```

Зворотне перетворення - звуження - означає, що перехід здійснюється від більш ємкого типу до менш ємкому. При такому перетворенні є ризик втратити дані. Наприклад, якщо число типу int було більше 127, то при приведенні його до byte значення бітів старше восьмого будуть втрачені. У Java таке перетворення повинне здійснюватися явним чином, тобто програміст в коді повинен явно вказати, що він має намір здійснити таке перетворення і готовий втратити дані.

Наступні 23 перетворення є такими, що звужують:

- від byte до char
- від short до byte, char
- від char до byte, short
- від int до byte, short, char
- від long до byte, short, char, int
- від float до byte, short, char, int, long
- від double до byte, short, char, int, long, float

При звуженні цілочисельного типу до вужчому цілочисельному всі старші біти, що не потрапляють в новий тип, просто відкидаються. Не проводиться ніякого округлення або інших дій для отримання коректнішого результату:

```
print((byte)383);
print((byte)384);
print((byte)-384);
```

Результатом буде:

```
127
-128
-128
```

Видно, що знаковий біт при звуженні не зробив ніякого впливу, оскільки був просто відкинутий - результат приведення зворотних чисел (384 і -384) виявився однаковим. Отже, може бути втрачене не тільки точне абсолютне значення, але і знак величини.

Це вірно і для типу char:

```
char c=40000;
print((short)c);
```

Результатом буде:

```
-25536
```

Звуження дробового типу до цілочисельного є складнішою процедурою. Вона проводиться в два етапи. На першому кроці дробове значення перетвориться в long, якщо цільовим типом є long, або в int - інакше (цільовий тип byte, short, char або int). Для цього початкове дробове число спочатку математично округляється у бік нуля, тобто дробова

частина просто відкидається. Наприклад, число 3,84 буде закруглено до 3, а -3,84 перетвориться на -3. При цьому можуть виникнути особливі випадки:

- якщо початкове дробове значення є NAN, то результатом першого кроку буде 0 вибраного типу (т.е.int або long);
- якщо початкове дробове значення є позитивною або негативною нескінченністю, то результатом першого кроку буде, відповідно, максимально або мінімальне можливе значення для вибраного типу (тобто для int або long);
- нарешті, якщо дробове значення було кінцевим величиною, але в результаті округлення вийшло дуже велике по модулю число для вибраного типу (тобто для int або long), то, як і в попередньому пункті, результатом першого кроку буде, відповідно, максимально або мінімальне можливе значення цього типу. Якщо ж результат округлення укладається в діапазон значень вибраного типу, то він і буде результатом першого кроку.

На другому кроці проводиться подальше звуження від вибраного цілочисельного типу до цільового, якщо таке потрібне, тобто може мати місце додаткове перетворення від int до byte, short або char.

Проілюструємо описаний алгоритм перетворенням від нескінченності до всіх цілочисельних типів:

```
float fmin = Float.NEGATIVE_INFINITY;
float fmax = Float.POSITIVE_INFINITY;
print("long: " + (long)fmin + ".." +
    (long)fmax);

print("int: " + (int)fmin + ".." +
    (int)fmax);

print("short: " + (short)fmin + ".." +
    (short)fmax);

print("char: " + (int)(char)fmin + ".." +
    (int)(char)fmax);

print("byte: " + (byte)fmin + ".." +
    (byte)fmax);
```

Результатом буде:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

Значення long і int цілком очевидні - дробова нескінченність перетворилася в, відповідно, мінімально і максимально можливі значення цих типів. Результат для наступних трьох типів (short, char, byte) є, по суті, подальше звуження значень, отриманих

для int, згідно другому кроку процедури перетворення. А робиться це, як було описано, просто за рахунок відкидання старших бітів. Пригадаємо, що мінімальне можливе значення в бітовому вигляді представляється як 1000..000 (всього 32 бита для int, тобто одиниця і 31 нуль). Максимально можливе - 1111..111 (32 одиниці). Відкидаючи старші біти, отримуємо для негативної нескінченності результат 0, однаковий для всіх трьох типів. Для позитивної ж нескінченності отримуємо результат, всі біти якого дорівнюють 1. Для знакових типів byte і short така комбінація розглядається як -1, а для беззнакового char - як максимально можливе значення, тобто 65535.

Може скластися враження, що для char приведення дає точне значення. Проте це був окремий випадок - відкидання бітів в більшості випадків все ж таки дає спотворення. Наприклад, звуження дробового значення 2 мільярди:

```
float f=2e9f;
print((int)(char)f);
print((int)(char)-f);
```

Результатом буде:

```
37888
27648
```

Звернете увагу на подвійне приведення для значень типу char в двох останніх прикладах. Зрозуміло, що перетворення від char до int не приводить до втрати точності, але дозволяє роздруковувати не символ, а його числовий код, що зручніше для аналізу.

На закінчення ще раз звернемо увагу на те, що примітивні значення типу boolean можуть брати участь тільки в тотожних перетвореннях.

Перетворення посилальних типів (розширення і звуження). Переходимо до посилальних типів. Перетворення об’єктних типів краще всього ілюструється за допомогою дерева спадкоємства. Розглянемо невеликий приклад спадкоємства:

```
// Оголошуємо клас Parent
class Parent {
  int x;
}

// Оголошуємо клас Child і успадковуємо
// його від класу Parent
class Child extends Parent {
  int y;
}

// Оголошуємо другого спадкоємця
// класу Parent - клас Child2
class Child2 extends Parent {
  int z;
}
```

У кожному класі оголошено поле з унікальним ім'ям. Розглядатимемо це поле як приклад набору унікальних властивостей, властивих деякому об'єктному типу. Три оголошені класи можуть породжувати три види об'єктів. Об'єкти класу Parent володіють тільки одним полем x, а значить, тільки посилання типу Parent можуть посилатися на такі об'єкти. Об'єкти класу Child володіють полем y і полемо x, отриманим по спадку від класу Parent. Отже, на такі об'єкти можуть указувати посилання типу Child або Parent. Другий випадок вже ілюструвався наступним прикладом:

```
Parent p = new Child();
```

Звернете увагу, що за допомогою такого посилання p можна звертатися лише до поля x створеного об'єкту. Поле y недоступно, оскільки компілятор, перевіряючи коректність виразу p.y, не може передбачити, що посилання p указуватиме на об'єкт типу Child під час виконання програми. Він аналізує лише тип самою змінною, а вона оголошена як Parent, але в цьому класі немає поля y, що і викличе помилку компіляції. Аналогічно, об'єкти класу Child2 володіють полем z і полем x, отриманим по спадку від класу Parent. Значить, на такі об'єкти можуть указувати посилання типу Child2 або Parent. Таким чином, посилання типу Parent можуть указувати на об'єкт будь-якого з трьох даних типів, а посилання типу Child і Child2 - тільки на об'єкти такого самого типу. Тепер можна перейти до перетворення посилальних типів на основі такого дерева спадкоємства. Розширення означає перехід від конкретнішого типу до менш конкретному, тобто перехід від дітей до батьків. У нашому прикладі перетворення від будь-якого спадкоємця (Child, Child2) до батька (Parent) є розширення, перехід до більш загального типу. Подібно злучаю з примітивними типами, цей перехід проводиться самій JVM при необхідності і непомітний для розробника, тобто не вимагає ніяких додаткових зусиль, оскільки він завжди проходить успішно: завжди можна звертатися до об'єкту, породженого від спадкоємця, за типом його батька.

```
Parent p1=new Child();
Parent p2=new Child2();
```

У обох рядках змінним типу Parent привласнюється значення іншого типу, а значить, відбувається перетворення. Оскільки це розширення, воно проводиться автоматично і завжди успішно. Зверніть увагу, що при подібному перетворенні з самим об'єктом нічого не відбувається. Не дивлячись на те, що, наприклад, поле y класу Child тепер недоступно, це не означає, що воно зникло. Така істотна зміна структури об'єкту неможлива. Він був породжений від класу Child і зберігає всі його властивості. Змінився лише тип посилання, через яке йде звернення до об'єкту. Цю ситуацію можна умовно порівняти з розглядом якогось предмету через підзорну трубу. Якщо перейти від труби з великим збільшенням до слабкішою, то видимих деталей стане менше, але сам предмет, звичайно, ніяк від цього не зміниться.

Наступні перетворення є такими, що розширюють:

- від класу A до класу B, якщо A успадковується від B (важливим окремим випадком є перетворення від будь-якого посилального типу до Object);
- від null-типа до будь-якого об'єктного типу.

Другий випадок ілюструється наступним прикладом:

```
Parent p=null;
```

Порожнє посилання null не володіє яким-небудь конкретним посилальним типом, тому іноді говорять про спеціальномnull-типе. Проте на практиці важливо, що таке значення можна прозоро перетворити до будь-якого об'єктного типу.

Зворотний перехід, тобто рух по дереву спадкоємства вниз, до спадкоємців, є звуженням. Наприклад, для даного випадку, перехід від посилання типу Parent, яка може посилатися на об'єкти трьох класів, до посилання типу Child, яка може посилатися на об'єкти лише одного з трьох класів, очевидно, є звуженням. Такий перехід може виявитися неможливим. Якщо посилання типу Parent посилається на об'єкт типаParent або Child2, то перехід до Child неможливий, адже в обох випадках об'єкт не володіє полем y, яке оголошене в класі Child. Тому при звуженні розробникові необхідно явним чином указувати на те, що необхідно спробувати провести таке перетворення. JVM під час виконання перевірить коректність переходу. Якщо він можливий, перетворення буде проведено. Якщо ж немає - виникне помилка.

```
Parent p=new Child();
Child c=(Child) p;
  // перетворення буде успішним.
Parent p2=new Child2();
Child c2=(Child) p2;
  // під час виконання виникне помилка!
```

Щоб перевірити, чи можливий бажаний перехід, можна скористатися оператором instanceof:

```
Parent p=new Child();
if (p instanceof Child) {
  Child з = (Child)p;
}
Parent p2=new Child2();
if (p2 instanceof Child) {
  Child з = (Child)p2;
}
Parent p3=new Parent();
if (p3 instanceof Child) {
  Child з = (Child)p3;
}
```

У даному прикладі помилок не виникне. Перше перетворення можливе, і воно буде здійснено. У другому і третьому випадках умови операторів if не спрацюють і спроб некоректного переходу не буде. На даний момент можна назвати лише одне звужуюче перетворення:

- від класу A до класу B, якщо B успадковується від A (важливим окремим випадком є звуження типу Object до будь-якого іншого посилального типу).

З вивченням решти посилальних типів (інтерфейсів і масивів) цей список розширюватиметься.

Перетворення до рядка

Це перетворення вже не раз згадувалося. Будь-який тип може бути приведений до рядка, тобто до екземпляра классаString. Таке перетворення є винятковим внаслідок того, що охоплює абсолютно всі типи, у тому числі і boolean, про який мовилося, що він не може брати участь ні в якому іншому приведенні, окрім тотожного.

Нагадаємо, як перетворяться різні типи.

- Числові типи записуються в текстовому вигляді без втрати точності уявлення. Формальне таке перетворення відбувається в два етапи. Спочатку на основі примітивного значення породжується екземпляр відповідного класса-"обертки", а потім у нього викликається метод toString(). Але оскільки ці дії зовні непомітні, багато JVM оптимізують їх і перетворять примітивні значення в текст безпосередньо.
- Булева величина приводиться до рядка "true" або "false" залежно від значення.
- Для об’єктних величин викликається метод toString(). Якщо метод повертає null, то результатом буде рядок "null".
- Для null-значения генерується рядок "null".

Заборонені перетворення

Не всі переходи між довільними типами допустимі. Наприклад, до заборонених перетворень відносяться: переходи від будь-якого посилального типу до примітивного, від примітивного - до посилального (окрім перетворень до рядка). Вже згадуваний приклад - тип boolean - не можна привести ні до якого іншому типу, окрім boolean(як завжди - за винятком приведення до рядка). Потім, неможливо привести один до одного типи, що знаходяться не на одній, а на сусідніх гілках дерева спадкоємства. У прикладі, який розглядався для ілюстрації перетворень посилальних типів, перехід від Child до Child2 заборонений. Насправді, посилання типу Child може указувати на об’єкти, породжені тільки від класу Child або його спадкоємців. Це виключає вірогідність того, що об’єкт буде сумісний з типом Child2.

Застосування перетворень. Тепер, коли розглянуті всі види перетворень, перейдемо до ситуацій в коді, де можуть зустрітися або потрібно приведення.

Такі ситуації можуть бути згруповані таким чином.

- Привласнення значень змінним (assignment). Не всі переходи допустимі при такому перетворенні - обмеження вибрані так, щоб не могла виникнути помилкова ситуація.
- Виклик методу. Це перетворення застосовується до аргументів методу, що викликається, або конструктора. Допускаються майже ті ж переходи, що і для привласнення значень. Таке приведення ніколи не породжує помилок. Так само приведення здійснюється при поверненні значення з методу.
- Явне приведення. В цьому випадку явно указується, до якого типу потрібно привести початкове значення. Допускаються всі види перетворень, окрім

приведень до рядка і заборонених. Може виникати помилка часу виконання програми.

- Оператора конкатенації проводить перетворення до рядка своїх аргументів.
- Числове розширення (numeric promotion). Числові операції можуть зажадати зміни типу аргументу(ів). Це перетворення має особлива назва - розширення (promotion), оскільки вибір цільового типу може залежати не тільки від початкового значення, але і від другого аргументу операції.

Розглянемо всі випадки детальніше.

Привласнення значень. Такі ситуації неодноразово застосовувалися в цій лекції для ілюстрації видів перетворення. Приведення може потрібно, якщо змінною одного типу привласнюється значення іншого типу. Можливі наступні комбінації. Якщо поєднання цих двох типів утворює заборонене приведення, виникне помилка. Наприклад, примітивні значення не можна привласнювати об’єктним змінним, включаючи наступні приклади:

```
// приклад викличе помилку компіляції

// примітивне значення не можна
// привласнити об’єктній змінній
Parent p = 3;

// приведення до классу-"обернення"
// також заборонено
Long a=5L;
// універсальне приведення до рядка
// можливо тільки для оператора +
String s=true;
```

Далі, якщо поєднання цих двох типів утворює розширення (примітивних або посилальних типів), то воно буде здійснено автоматично, неявним для розробника чином:

```
int i=10;
long a=i;
Child з = new Child();
Parent p=c;
```

Якщо ж поєднання виявляється звуженням, то виникає помилка компіляції, такий перехід не може бути проведений неявно:

```
// приклад викличе помилку компіляції
int i=10;
short s=i;   // помилка! звуження!
Parent p = new Child();
Child c=p;   // помилка! звуження!
```

Як уже згадувалося, в подібних випадках необхідно виконувати перетворення явно:

```
int i=10;
short s=(short) i;
Parent p = new Child();
Child c=(Child) p;
```

Докладніше явне звуження розглядається нижче.

Тут може викликати здивування наступна ситуація, яка не породжує помилок компіляції:

```
byte b=1;
short s=2+3;
char c=(byte) 5+’a’;
```

У першому рядку змінної типу byte привласнюється значення цілочисельного літерала типу int, що є звуженням. У другому рядку змінної типу short привласнюється результат складання двох літералів типу int, а тип цієї суми також int. Нарешті, в третьому рядку змінної типу char привласнюється результат складання числа 5, приведеного до типу byte, і символьного літерала. Проте всі ці приклади коректні. Для зручності розробника компілятор проводить додатковий аналіз при привласненні значень змінним типу byte, short і char. Якщо таким змінним привласнюється величина типаbyte, short, char або int, причому її значення може бути набуте вже на момент компіляції, і виявляється, що це значення укладається в діапазон типу змінної, то явного приведення не вимагається. Якби такої можливості не було, довелося б писати так:

```
byte b=(byte) 1;
  // перетворення необов’язкове
short s=(short)(2+3);
  // перетворення необов’язкове
char c=(char)((byte)5+’a’);
  // перетворення необов’язкове

// перетворення необхідне, оскільки
// число 200 не укладається в тип byte
byte b2=(byte) 200;
```

Виклик методу. Це приведення виникає у разі, коли викликається метод з оголошеними параметрами одних типів, а при виклику передаються аргументи інших типів. Оголошення методів розглядається в наступних лекціях курсу, проте такий простий приклад цілком зрозумілий:

```
// оголошення методу з параметром типу long
void calculate(long l){
  ...
}
void main() {
  calculate(5);
}
```

Як видно, при виклику методу передається значення типу int, а не long, як визначено в оголошенні цього методу. Тут компілятор робить ті ж кроки, що і при приведенні в процесі привласнення значень змінним. Якщо типи утворюють заборонене перетворення, виникне помилка.

```
// приклад викличе помилку компіляції

void calculate(long a){
  ...
}

void main() {
  calculate(new Long(5));
  // тут буде помилка
}
```

Якщо звуження, то компілятор не зможе здійснити приведення і буде потрібно явні вказівки.

```
void calculate(int a){
  ...
}

void main() {
  long a=5;
  // calculate(a);
  // звуження! так буде помилка.
  calculate((int)a);   // коректний виклик
}
```

Нарешті, у разі розширення, компілятор здійснить приведення сам, як і було показано в прикладі на початку цього розділу. Треба відзначити, що, на відміну від ситуації привласнення, при виклику методів компілятор не проводить перетворень примітивних значень від byte, short, char або int до byte, short або char. Це привело б до ускладнення роботи з переобтяженими методами. Наприклад:

```
// приклад викличе помилку компіляції

// оголошуємо переобтяжені методи
// з аргументами (byte, int) і (short, short)
int m(byte a, int b) { return a+b; }
int m(short a, short b) { return a-b; }

void main() {
  print(m(12, 2));   // помилка компіляції!
}
```

В даному прикладі компілятор видасть помилку, оскільки при виклику аргументи мають тип (int, int), а методу з такими параметрами немає. Якби компілятор проводив перетворення для цілих величин, подібно до ситуації з привласненням значень, то приклад

став би коректним, але довелося б докладати додаткові зусилля, щоб вказати, яким з двох можливих переобтяжених методів хотілося б викликати.

Аналогічне перетворення буде потрібно при поверненні значення з методу, якщо тип результату і заявлений тип зворотного значення не співпадають.

```
long get() {
  return 5;
}
```

Хоча у виразі return вказаний цілочисельний літерал типу int, у всіх місцях, де буде викликаний цей метод, буде набуто значення типу long. Для такого перетворення діють ті ж правила, що і для привласнення значення.

На закінчення розглянемо приклад, що включає всі розглянуті випадки перетворення:

```
short get(Parent p){
  return 5+'A';
  // приведення при поверненні значення
}
void main() {
  long a =  5L;
  // приведення при привласненні значення
  get(new Child());
  // приведення при виклику методу
}
```

Явне перетвореня. Явне приведення вже багато разів використовувалося в прикладах. При такому перетворенні зліва від виразу, тип значення якого необхідно перетворити, в круглих дужках указується цільовий тип. Якщо перетворення пройде успішно, то результат буде точно вказаного типу.

Приклади:

```
(byte)5
(Parent)new Child()
(Flat)getCity().getStreet(
    ).getHouse().getFlat()
```

Якщо комбінація типів утворює заборонене перетворення, виникає помилка компіляції. Допускаються тотожні перетворення, розширення простих і об'єктних типів, звуження простих і об'єктних типів. Перші три завжди виконуються успішно. Останні два можуть стати причиною помилки виконання, якщо значення виявилися несумісними. Як наслідок, вираз null завжди може бути успішно перетворене до будь-якого посилального типу. Але можна знайти спосіб все-таки закодувати заборонене перетворення.

```
Child c=new Child();
// Child2 c2=(Child2) з;
```

```
// заборонене перетворення
Parent p=c;   // розширення
Child2 c2=(Child2) p;   // звуження
```

Такий код буде успішно скомпільований, проте, зрозуміло, при виконання він завжди генеруватиме помилку в останньому рядку. "Одурювати" компілятор сенсу немає.

Оператор конкатенації рядків. Цей оператор вже розглядався достатньо детально. Якщо обома його аргументами є рядки, то відбувається звичайна конкатенація. Якщо ж тип String має лише один з аргументів, то другою необхідно перетворити в текст. Це єдина операція, при якій проводиться універсальне приведення будь-якого значення до типу String. Це одна з властивостей, що виділяють клас String із загального ряду. Правила перетворення вже були детально описані в цій лекції, а оператор конкатенації розглядався в лекції "Типи даних".

Невеликий приклад:

```
int i=1;
double d=i/2.;
String s="text";
print("i="+i+", d="+d+", s="+s);
```

Результатом буде:

```
i=1, d=0.5, s=text
```

Числове розширення. Останній вид перетворень застосовується при числових операціях, коли потрібно привести аргумент(и) до типу завдовжки в 32 або 64 бита для проведення обчислень. Таким чином, при числовому розширенні здійснюється тільки розширення примітивних типів. Розрізняють унарне і бінарне числове розширення.

Унарне числове розширення - це перетворення розширює примітивні типи byte, short або char до типів int по правилах розширення примітивних типів.

Унарне числове розширення може виконуватися при наступних операціях:

- унарні операції + і -;
- бітове заперечення ~;
- операції бітового зрушення <<, >>, >>>.

Операторів зрушення мають два аргументи, але вони розширюються незалежно один від одного, тому дане перетворення є унарним. Таким чином, результат виразу 5<<3L має тип int. Взагалі, результат операторів зрушення завжди має тип int або long.

Бінарне числове розширення - це перетворення розширює всі примітивні числові типи, окрім double, до типів int, long, float, doubleпо правилам розширення примітивних типів. Бінарне числове розширення відбувається при числових операторах, що мають два аргументи, по наступних правилах:

- якщо будь-який з аргументів має тип double, то і другою приводиться до double;
- інакше, якщо будь-який з аргументів має тип float, то і другою приводиться до float;
- інакше, якщо будь-який з аргументів має тип long, то і другою приводиться до long;
- інакше обидва аргументи приводяться до int.

Бінарне числове розширення може виконуватися при наступних операціях:

- арифметичні операції +, - *, /, %;
- операції порівняння <, <= >, >= ==, !=;
- бітові операції &, |, ^;
- в деяких випадках для операції з умовою ? :.

Приклади роботи всіх цих операторів з урахуванням розширення детально розглядалися в попередніх лекціях.

Тип змінної і тип її значення

Тепер, коли були детально розглянуті всі приклади перетворень, потрібно повернутися до питання змінної і її значень. Як вже говорилося, змінна визначається трьома базовими характеристиками: ім'я, тип, значення. Ім'я дається довільним чином і ніяк не позначається на властивостях змінної. А ось значення завжди має деякий тип, не обов'язково співпадаючий з типом самої змінної. Тому необхідно розглянути всі можливі типи змінних і з'ясувати, значення яких типів вони можуть мати. Почнемо із змінних примітивних типів. Оскільки ці змінні дійсно зберігають само значення, то їх тип завжди точно співпадає з типом значення.

Проілюструємо це правило на прикладі:

```
byte b=3;
char c='A'+3; long m=b+c;
double d=m-3F ;
```

Тут змінна b зберігатиме значення типу byte після звуження цілочисельного літерала типу int. Змінна з зберігатиме тип charпосле того, як компілятор здійснить звужуюче перетворення результату підсумовування, який матиме тип int. Для змінній mвыполнится розширення результату підсумовування типу від int до типу long. Нарешті, змінна d зберігатиме значення типу double, що вийшло в результаті розширення результату різниці, який має тип float.

Переходимо до посилальних типів. По-перше, значення будь-якої змінної такого типу - посилання, яке може указувати лише на об'єкти, породжені від тих або інших

класів, і далі обговорюються тільки властивості даних класів. (Також об'єкти можуть породжуватися від масивів, ця тема розглядається в окремій лекції.) Крім того, посилальна змінна будь-якого типу може мати значення null. Більшість дій над такою змінною, наприклад, звернення до полів або методів, приведе до помилки. Отже, який зв'язок між типом посилальної змінної і її значенням? Тут головне обмеження - перевірка компілятора, який стежить, щоб всі дії, що виконуються над об'єктом, були коректні. Компілятор не може передбачити, на об'єкт якого класу реально посилатиметься та або інша змінна. Все, що він має в своєму розпорядженні, - тип самої змінної. Саме його і використовує компілятор для перевірок. А значить, всі допустимі значення змінної повинні гарантовано володіти властивостями, визначеними в класі-типі цієї змінної. Таку гарантію дає тільки спадкоємство. Звідси отримуємо правило: посилальна змінна типу A може вказувати на об'єкти, породжені від самого типу A або його спадкоємців.

```
Point p = new Point();
```

В даному прикладі змінна і її значення однакового типу, тому над об'єктом можна здійснювати всі можливі для даного класу дії.

```
Parent p = new Child();
```

Таке привласнення коректне, оскільки клас Child породжений від Parent. Проте тепер допустимі дії над змінній p, а значить, над об'єктом, тільки що створеним на основі класу Child, обмежені можливостями класу Parent. Наприклад, якщо в класі Child визначений якийсь новий метод newChildMethod(), то спроба його викликати p.newChildMethod() породжуватиме помилку компіляції. Необхідно підкреслити, що ніяких змін з самим об'єктом не відбувається, обмеження породжується використовуваним способом доступу до цього об'єкту - змінної типу Parent. Щоб показати, що об'єкт не втратив ніяких властивостей, проведемо наступне звернення:

```
((Child)p).newChildMethod();
```

Тут на початку проводиться явне звуження до типу Child. Під час виконання програми JVM перевірить, чи сумісний тип об'єкту, на який посилається змінна p, з типом Child. У нашому випадку це саме так. В результаті виходить посилання типу Child, тому стає допустимим виклик методу newChildMethod(), який викликається у об'єкту, створеного в попередньому рядку. Звернемо увагу на важливий окремий випадок - змінна типу Object може посилатися на об'єкти будь-якого типу.

Надалі, з вивченням нових типів (абстрактних класів, інтерфейсів, масивів) цей список продовжуватиметься, а поки коротко узагальнимо те, що було розглянуте в даному розділі.

Таблиця 4.1. Цілочисельні типи даних.

Тип змінної	Допустимі типи її значення
Примітивний	У точності співпадає з типом змінної
Посилальний	• null • співпадаючий з типом змінної

	• класи-спадкоємці від типу змінної
Object	• null • будь-який посилальний

Висновок

У цій лекції були розглянуті правила роботи з типами даних в мові Java, що строго типізувалася. Оскільки компілятор строго відстежує тип кожної змінної і кожного виразу, у разі зміни цього типу необхідно чітко розуміти, які дії допустимі, а які немає, з погляду компілятора і віртуальної машини.

Були розглянуті всі види приведення типів в Java, тобто перехід від одного типу до іншого. Вони розбиваються на 7 груп, починаючи з тотожного і закінчуючи забороненими. Основні 4 види визначаються звужуючими або розширюючими переходами між простими або посилальними типами. Важливо пам’ятати, що при явному звуженні числових типів старші біти просто відкидаються, що деколи приводить до несподіваного результату. Що стосується перетворення посилальних значень, то тут діє правило - перетворення ніколи не породжує нових і не змінює існуючих об’єктів. Міняється лише спосіб роботи з ними.

Особливим в Java є перетворення до рядка.

Потім були розглянуті всі ситуації в програмі, де можуть відбуватися перетворення типів. Перш за все, це привласнення значень, коли перетворення часто відбувається непомітно для програміста. Виклик методу багато в чому схожий на ініціалізацію. Явне приведення дозволяє здійснити бажаний перехід у тому випадку, коли компілятор не дозволяє зробити це неявно. Перетворення при виконанні числових операцій робить істотний вплив на результат.

На закінчення був розглянутий зв’язок між типом змінної і типом її значення.

ТЕМА. *Об'єктна модель JAVA.*

План

1. *Ключові властивості об'єктної моделі JAVA.*
2. *Статичні елементи.*
3. *Абстрактні методи та класи.*
4. *Інтерфейси.*

Рекомендована література: 1, 2, 3, 5.

Теоретичні відомості

Розглянемо технічні особливості Java, а саме ключові властивості об'єктної моделі Java, такі як статичні елементи, абстрактні методи і класи, інтерфейси, що є альтернативою множинного спадкоємства. Без цих потужних конструкцій мова Java була б нездібна вирішувати серйозні завдання.

Статичні елементи. До цього моменту під полями об'єкту ми завжди розуміли значення, які мають сенс лише в контексті деякого екземпляра класу. Наприклад:

```
class Human {
  private String name;
}
```

Перш, ніж звернутися до поля name, необхідно отримати заслання на екземпляр класу Human, неможливо взнати ім'я взагалі, воно завжди належить якійсь конкретній людині.

Але бувають дані і іншого характеру. Передбачимо, необхідно зберігати кількість всіх людей (екземплярів класу Human, що існують в системі). Зрозуміло, що загальне число людей не є характеристикою якогось однієї людини, воно відноситься до всього типа в цілому. Звідси з'являється назва "Поле класу", на відміну від "поля об'єкту". Оголошуються такі поля за допомогою модифікатора static:

```
class Human {
  public static int totalCount;
}
```

Аби звернутися до такого поля, заслання на об'єкт не потрібне, сповна досить імені класу:

```
Human.totalCount++;
  // народження ще однієї людини
```

Для зручності дозволено звертатися до статичних полів і через заслання:

```
Human h = new Human();
h.totalCount=100;
```

Проте таке звернення конвертується компілятором. Він використовує типа заслання, в даному випадку змінна h об'явлена як Human, тому останній рядок буде неявно перетворений в:

```
Human.totalCount=100;
```

У цьому можна переконатися на наступному прикладі:

```
Human h = null;
h.totalCount+=10;
```

Значення заслання рівне null, але це не має значення через описану конвертацію. Даний код успішно скомпілюється і коректно виконається. Таким чином, в наступному прикладі

```
Human h1 = new Human(), h2 = new Human();
Human.totalCount=5;
h1.totalCount++;
System.out.println(h2.totalCount);
```

всі звернення до змінної totalCount наводять до одного єдиного поля, і результатом роботи такої програми буде 6. Це поле існуватиме в єдиному екземплярі незалежно від того, скільки об'єктів було породжено від даного класу, і чи був взагалі створений хоч один об'єкт.

Аналогічно оголошуються статичні методи.

```
class Human {
  private static int totalCount;

  public static int getTotalCount() {
    return totalCount;
  }
}
```

Для виклику статичного методу заслання на об'єкт не потрібний.

```
Human.getTotalCount();
```

Хоча для зручності звернення через заслання дозволені, але береться до уваги лише тип заслання:

```
Human h=null;
h.getTotalCount();      // два еквівалентних
Human.getTotalCount();  // звернення до одного
                        // і тому ж методу
```

Хоча наведений приклад технічно коректний, все ж використання заслання на об'єкт для звернення до статичних полів і методів не рекомендується, оскільки це ускладнює код.

Звернення до статичного поля є коректним незалежно від того, чи були породжені об'єкти від цього класу і в якій кількості. Наприклад, стартовий метод main() запускається до того, як програма створить хоч би один об'єкт.

Окрім полів і методів, статичними можуть бути ініціалізатори. Вони також називаються ініціалізаторами класу, на відміну від ініціалізаторів об'єкту, що розглядалися раніше. Їх код виконується один раз під час завантаження класу в пам'ять віртуальної машини. Їх запис починається з модифікатора static:

```
class Human {
```

```
  static {
    System.out.println("Class loaded");
  }
}
```

Якщо оголошення статичного поля поєднується з його ініціалізацією, то поле ініціалізувалося також однократно при завантаженні класу. На оголошення і вживання статичних полів накладаються ті ж обмеження, що і для динамічних, - не можна використовувати поле в ініціалізаторах інших полів або в ініціалізаторах класу до того, як це поле оголошене:

```
class Test {
  static int a;
  static {
    a=5;
    // b=7;   // не Можна використовувати до
            // оголошення!
  }
  static int b=a;
}
```

Це правило поширюється лише на звернення до полів по простому імені. Якщо використовувати складене ім'я, то звертатися до поля можна буде раніше (вище в тексті програми), ніж воно буде оголошено:

```
class Test {
  static int b=Test.a;
  static int a=3;
  static {
    System.out.println("a="+a+", b="+b);
  }
}
```

Якщо клас буде завантажений в систему, на консолі з'явиться текст:

```
a=3, b=0
```

Видно, що поле b при ініціалізації набуло значення за умовчанням поля a, тобто 0. Потім полю a було привласнено значення 3.

Статичні поля також можуть бути оголошені як final, це означає, що вони мають бути проинициализированы строго один раз і потім вже більше не міняти свого значення. Аналогічно, статичні методи можуть бути оголошені як final, а це означає, що їх не можна перекривати в класах-спадкоємцях.

Для ініціалізації статичних полів можна користуватися статичними методами і не можна звертатися до динамічних. Вводять спеціальні поняття – статичний і динамічний контексти. До статичного контексту відносять статичні методи, статичні ініціалізатори, ініціалізатори статичних полів. Всі останні частини коди мають динамічний контекст.

При виконанні коди в динамічному контексті завжди є об'єкт, з яким йде робота в даний момент. Наприклад, для динамічного методу це об'єкт, в якого він був викликаний, і так далі.

Навпаки, із статичним контекстом асоційованих об'єктів немає. Наприклад, як вже вказувалося, стартовий метод main() викликається в той момент, коли жоден об'єкт ще не створений. При зверненні до статичного методу, наприклад, MyClass.staticMethod(), також може не бути жодного екземпляра MyClass. Звертатися до статичних методів класу Math можна, а створювати його екземпляри не можна.

А раз немає асоційованих об'єктів, то і користуватися динамічними конструкціями не можна. Можна лише посилатися на статичні поля і викликати статичні методи. Або звертатися до об'єктів через заслання на них, отримані в результаті виклику конструктора або як аргумент методу і тому подібне

```
class Test {
  public void process() {
  }
  public static void main(String s[]){
    // process(); - помилка!
    // в якого об'єкту його викликати?

    Test test = new Test();
    test.process(); // так правильно
  }
}
```

Ключові слова this і super

Ці ключові слова вже згадувалися, розглядалися і деякі випадки їх вживання. Тут вони будуть описані детальніше.

Якщо виконання коди відбувається в динамічному контексті, то має бути об'єкт, що асоціюється з ним. В цьому випадку ключове слово thisвозвращает заслання на даний об'єкт:

```
class Test {
  public Object getThis() {
    return this;
    // Перевіримо, куди вказує це заслання
  }
  public static void main(String s[]){
    Test t = new Test();
    System.out.println(t.getThis()==t);
   // Порівняння
```

```
  }
}
```

Результатом роботи програми буде:

```
true
```

Тобто усередині методів слово this повертає заслання на об'єкт, в якого цей метод викликаний. Воно необхідне, якщо потрібно передати аргумент, рівний засланню на даний об'єкт, в який-небудь метод.

```
class Human {
  public static void register(Human h){
    System.out.println(h.name+
      " is registered.");
  }

  private String name;
  public Human (String s) {
    name = s;
    register(this); // саморегистрация
  }

  public static void main(String s[]){
    new Human("John");
  }
}
```

Результатом буде:

```
John is registered.
```

Інше вживання this розглядалося в разі "затемняючих" оголошень:

```
class Human {
  private String name;

  public void setName(String name){
    this.name=name;
  }
}
```

Слово this можна використовувати для звернення до полів, які оголошуються нижче:

```
class Test {
```

```
  // int b=a; не можна звертатися до
  // неоголошеному полю!
  int b=this.a;
  int a=5;
  {
    System.out.println("a="+a+", b="+b);
  }
  public static void main(String s[]){
    new Test();
  }
}
```

Результатом роботи програми буде:

```
a=5, b=0
```

Все відбувається так само, як і для статичних полів – b набуває значення за умовчанням для a, тобто нуль, а затема ініціалізувався значенням 5.

Нарешті, слово this застосовується в конструкторах для явного виклику в першому рядку іншого конструктора цього ж класу. Там же може застосовуватися і слово super, лише вже для звернення до конструктора батьківського класу.

Інші вживання слова super також пов'язані із зверненням до батьківського класу об'єкту. Наприклад, воно може потрібно в разі перевизначення (overriding) батьківського методу.

Перевизначенням називають оголошення методу, сигнатура якого збігається з одним з методів батьківського класу.

```
class Parent {
  public int getValue() {
    return 5;
  }
}

class Child extends Parent {
  // Перевизначення методу
  public int getValue() {
    return 3;
  }


  public static void main(String s[]){
    Child з = new Child();

    // приклад виклику перевизначеного методу
    System.out.println(з.getValue());
```

```
  }
}
```

Виклик перевизначеного методу використовує механізм поліморфізму, який детально розглядається в кінці цієї лекції. Проте ясно, що результатом виконання прикладу буде значення 3. Неможливо, використовуючи заслання типа Child, отримати з методу getValue() значення 5, батьківський метод перекритий і вже недоступний.

Інколи при перевизначенні буває корисно скористатися результатом роботи батьківського методу. Передбачимо, він робив складні обчислення, а перевизначений метод повинен повернути закруглений результат цих обчислень. Зрозуміло, що набагато зручніше звернутися до батьківського методу, ніж заново описувати весь алгоритм. Тут застосовується слово super. З класу спадкоємця з його допомогою можна звертатися до перевизначених методів батька:

```
class Parent {
  public int getValue() {
    return 5;
  }
}

class Child extends Parent {

  // перевизначення методу
  public int getValue() {
    // звернення до методу батька
    return super.getValue()+1;
  }

  public static void main(String s[]){
    Child з = new Child();
    System.out.println(з.getValue());
  }
}
```

Результатом роботи програми буде значення 6.

Звертатися за допомогою ключового слова super до перевизначеного методу батька, тобто на два рівні спадкоємства вгору, неможливо. Якщо батьківський клас перевизначив функціональність свого батька, значить, вона не буде доступна його спадкоємцям.

Оскільки ключові слова this і super вимагають наявність асоційованого об’єкту, тобто динамічного контексту, використання їх в статичному контексті заборонено.

Ключове слово abstract

Наступне важливе поняття, яке необхідно розглянути, - ключове слово abstract.

Інколи має сенс описати лише заголовок методу, без його тіла, і таким чином оголосити, що даний метод існуватиме в цьому класі. Реалізацію цього методу, тобто його тіло, можна описати пізніше.

Розглянемо приклад. Передбачимо, необхідно створити набір графічних елементів, неважливо, яких саме. Наприклад, вони можуть бути геометричними фігурами – круг, квадрат, зірка і т.д.; або елементи призначеного для користувача інтерфейсу – кнопки, поля введення і так далі Зараз це не має вирішального значення. Крім того, існує спеціальний контейнер, який займається їх отрисовкой. Зрозуміло, що зовнішній вигляд кожної компоненты унікальний, а значить, відповідний метод (назвемо його paint()) буде реалізований в різних елементах по-різному.

Але в той же час в компонент може бути багато загального. Наприклад, будь-яка з них займає деяку прямокутну область контейнера. Складні контури фігури необхідно вписати в прямокутник, аби можна було аналізувати перекриття, перевіряти, чи не вилазить компонент за кордони контейнера, і так далі Кожна фігура може мати колір, яким її треба малювати, може бути видимою, або невидимою і так далі Вочевидь, що корисно створити батьківський клас для всіх компонент і один раз оголосити в нім всі загальні властивості, аби кожна компонента лише успадковувала їх.

Але як поступити з методом отрисовки? Адже батьківський клас не є якою-небудь фігурою, у нього немає візуальної вистави. Можна оголосити метод paint() у кожній компоненте незалежно. Але тоді контейнер повинен буде володіти складною функціональністю, аби аналізувати, яка саме компонента зараз обробляється, виконувати приведення типа і лише після цього викликати потрібний метод.

Саме тут зручно оголосити абстрактний метод в батьківському класі. У нього немає зовнішнього вигляду, але відомо, що він є у кожного спадкоємця. Тому заголовок методу описується в батьківському класі, тіло методу у кожного спадкоємця своє, а контейнер може спокійно користуватися лише базовим типом, не роблячи жодних приведень.

Наведемо спрощений приклад:

```
// Базова арифметична операція
abstract class Operation {
  public abstract int calculate(int a, int b);
}
// Складання
class Addition extends Operation {
  public int calculate(int a, int b){
    return a+b;
   }
}

// Віднімання
class Subtraction extends Operation {
   public int calculate(int a, int b){
     return a-b;
   }
}
```

```
class Test {
  public static void main(String s[]){
    Operation o1 = new Addition();
    Operation o2 = new Subtraction();

    o1.calculate(2, 3);
    o2.calculate(3, 5);
  }
}
```

Видно, що виконання операцій складання і віднімання в методі main() записуються однаково.

Звернете увагу – оскільки абстрактний метод не має тіла, після опису його заголовка ставиться крапка з комою. А раз у нього немає тіла, то до нього не можна звертатися, поки його спадкоємці не опишуть реалізацію. Це означає, що не можна створювати екземпляри класу, в якого є абстрактні методи. Такий клас сам оголошується абстрактним.

Клас може бути абстрактним і в тому випадку, якщо у нього немає абстрактних методів, але має бути абстрактним, якщо такі методи є. Розробник може вказати ключове слово abstract в списку модифікаторів класу, якщо хоче заборонити створення екземплярів цього класу. Класи-спадкоємці повинні реалізувати (implements) всі абстрактні методи (якщо вони є) свого абстрактного батька, аби їх можна було оголошувати неабстрактними і породжувати від них екземпляри.

Звичайно, клас не може бути одночасний abstract і final. Це ж вірно і для методів. Крім того, абстрактний метод не може бути private, native, static.

Сам клас може без обмежень користуватися своїми абстрактними методами.

```
abstract class Test {
  public abstract int getX();
  public abstract int getY();
  public double getLength() {
    return Math.sqrt(getX()*getX()+
                getY()*getY());
  }
}
```

Це коректно, оскільки метод getLength() може бути викликаний лише в об’єкту. Об’єкт може бути породжений лише від неабстрактного класу, який є спадкоємцем від Test, і повинен був реалізувати всі абстрактні методи.

З цієї ж причини можна оголошувати змінні типа абстрактний клас. Вони можуть мати значення null або посилатися на об’єкт, породжений від неабстрактного спадкоємця цього класу.

Інтерфейси

Концепція абстрактних методів дозволяє запропонувати альтернативу множинному спадкоємству. У Java клас може мати лише одного батька, оскільки при множинному спадкоємстві можуть виникати конфлікти, які заплутують об'єктну модель. Наприклад, якщо в класу є два батько, які мають однаковий метод з различнойреализацией, то який з них успадкує новий клас? І яка буде функціональність батьківського класу, який втратило свого методу?

Всі ці проблеми не виникають в тому випадку, якщо успадковуються лише абстрактні методи від декількох батьків. Навіть якщо успадковано декілька однакових методів, все одно у них немає реалізації і можна один раз описати тіло методу, яке використовуватиметься при виклику будь-якого з цих методів.

Саме так влаштовані інтерфейси в Java. От їх не можна породжувати об'єкти, але інші класи можуть реалізовувати їх.

Оголошення інтерфейсів

Оголошення інтерфейсів дуже схоже на спрощене оголошення класів.

Воно починається із заголовка. Спочатку вказуються модифікатори.Інтерфейс може бути оголошений як public і тоді він буде доступний для загального використання, або модифікатор доступу може не вказуватися, в цьому випадку інтерфейс доступний лише для типів свого пакету. Модифікатор abstract для інтерфейсу не потрібний, оскільки всеинтерфейсы є абстрактними. Його можна вказати, але робити цього не рекомендується, аби не захаращувати код.

Далі записується ключове слово interface і ім'я інтерфейсу.

Після цього може слідувати ключове слово extends і список інтерфейсів, від яких успадковуватиметься оголошуваний інтерфейс. Батьківських типів може бути багато, головне, аби не було повторень і аби відношення спадкоємства не утворювало циклічної залежності.

Спадкоємство інтерфейсів дійсно дуже гнучке. Так, якщо є два інтерфейси, A і B, причому Внаследуется від A, то новий інтерфейс C може успадковуватися від них обох. Втім, зрозуміло, що вказівка спадкоємства від A є надлишковою, всі елементи цього інтерфейсу і так будуть отримані по спадку через інтерфейс B.

Потім у фігурних дужках записується тіло інтерфейсу.

```
public interface Drawble extends Colorable
  Resizable {
}
```

Тіло інтерфейсу складається з оголошення елементів, тобто полів-констант і абстрактних методів. Всі поляинтерфейса мають бути public final static, так що ці

модифікатори вказувати необов'язково і навіть небажано, аби не захаращувати код. Оскільки поля оголошуються фінальними, необхідно їх відразу ініціалізувати.

```
public interface Directions {
  int RIGHT=1;
  int LEFT=2;
  int UP=3;
  int DOWN=4;
}
```

Всі методи інтерфейсу є public abstract і ці модифікатори також необов'язкові.

```
public interface Moveable {
  void moveRight();
  void moveLeft();
  void moveUp();
  void moveDown();
}
```

Як ми бачимо, опис інтерфейсу набагато простіший, ніж оголошення класу.

Реалізація інтерфейсу

Кожен клас може реалізовувати будь-які доступні інтерфейси. При цьому в класі мають бути реалізовані всі абстрактні методи, що з'явилися при спадкоємстві від інтерфейсів або батьківського класу, аби новий клас міг бути оголошений неабстрактним.

Якщо з різних джерел успадковуються методи з однаковою сигнатурою, то досить один раз описатьреализацию і вона застосовуватиметься для всіх цих методів. Проте якщо у них різне повертане значення, то виникає конфлікт:

```
interface A {
 int getValue();
}

interface B {
 double getValue();
}
```

Якщо спробувати оголосити клас, що реалізовує обоє ці інтерфейсу, то виникне помилка компіляції. У класі опиняється два різні методи з однаковою сигнатурою, що є нерозв'язним конфліктом. Це єдине обмеження на набір інтерфейсів, які може реалізовувати клас.

Подібний конфлікт з полями-константами не настільки критичний:

```
interface A {
  int value=3;
}
interface B {
  double value=5.4;
}
class C implements A, B {
  public static void main(String s[]){
    C з = new C();
    // System.out.println(з.value); - помилка!
    System.out.println(((A)c).value);
    System.out.println(((B)c).value);
  }
}
```

Як видно з прикладу, звертатися до такого поля через сам клас не можна, компілятор не зможе зрозуміти, яке з двох полів потрібно використовувати. Але можна за допомогою явного приведення послатися на одне з них.

Отже, якщо ім'я інтерфейсу вказане після implements в оголошенні класу, то клас реалізує цей інтерфейс. Спадкоємці даного класу також реалізують інтерфейс, оскільки їм дістаються по спадку його елементи.

Якщо інтерфейс A успадковується від інтерфейсу B, а клас реалізує A, то вважається, що інтерфейс B також реалізується цим класом з тієї ж причини – всі елементи передаються по спадку в два етапи – сначалаинтерфейсу A, потім класу.

Нарешті, якщо клас C1 успадковується від класу C2, клас C2 реалізує інтерфейс A1, а інтерфейс A1 успадковується від інтерфейсу A2, то клас C1 також реалізує інтерфейс A2.

Все це дозволяє стверджувати, що змінні типа інтерфейс також допустимі. Вони можуть мати значення null, або посилатися на об'єкти, породжені від класів, що реалізовують цей інтерфейс. Оскільки об'єкти породжуються лише від класів, а всі вони успадковуються від Object, це означає, що значення типа интерфейсобладают всіма елементами класу Object.

Вживання інтерфейсів

До цих пір інтерфейси розглядалися з технічної точки зору – як їх оголошувати, які конфлікти можуть виникати, як їх вирішувати. Проте поважно розуміти, як застосовуються інтерфейси з концептуальної точки зору.

Поширена думка, що інтерфейс – це повністю абстрактний клас, в цілому вірно, але воно не відображає всіх переваг, які дають інтерфейси об'єктної моделі. Як вже наголошувалося, множинне спадкоємство породжує ряд конфліктів, але відмова від нього, хоч і робить мову простіше, але не усуває ситуації, в яких потрібні подібні підходи.

Візьмемо як приклад дерева спадкоємства класифікацію живих організмів. Відомо, що рослини і тварини належать до різних царств. Основною відмінністю між ними є те, що рослини поглинають неорганічні елементи, а тварини харчуються органічними речовинами. Тварини діляться на дві великі групи – птиці і ссавці. Передбачимо, що на основі цієї класифікації побудовано дерево спадкоємства, в кожному класі визначені елементи з врахуванням спадкоємства від батьківських класів.

Розглянемо таку властивість живого організму, як здатність харчуватися комахами. Вочевидь, що цю властивість не можна приписати всій групі птиць, або ссавців, а тим більше рослин. Але існують представники кожній з названих груп, які цією властивістю володіють, – для рослин це росянка, для птиць, наприклад, ластівки, а для ссавців – мурахоїди. Причому, вочевидь, "реалізована" ця властивість в кожного виду зовсім по-різному.

Можна було б оголосити відповідний метод (скажімо, consumeInsect(Insect)) у кожного представника незалежно. Але якщо завдання полягає в моделюванні, наприклад, зоопарку, то однотипну процедуру – годування комахами – довелося б описувати для кожного виду окремо, що істотно ускладнило б код, причому без якої-небудь користі.

Java пропонує інше рішення. Оголошується інтерфейс InsectConsumer:

```
public interface InsectConsumer {
  void consumeInsect(Insect i);
}
```

Його реалізують всі відповідні тварини і рослини:

```
// росянка розширює клас рослина
public class Sundew extends
  Plant implements InsectConsumer {
    public void consumeInsect(Insect i){
    ...
    }
}

// ластівка розширює клас птиця
public class Swallow extends
  Bird implements InsectConsumer {
    public void consumeInsect(Insect i){
    ...
    }
}
// мурахоїд розширює клас ссавець
public class AntEater extends
  Mammal implements InsectConsumer {
    public void consumeInsect(Insect i){
    ...
    }
}
```

В результаті в класі, що моделює зоопарку, що служить, можна оголосити відповідний метод:

```
// службовець, що відповідає за годування
// розширює клас службовець
class FeedWorker extends Worker {

  // за допомогою цього методу можна нагодувати
  // і росянку, і ластівку, і мурахоїда
  public void feedOnInsects(InsectConsumer
                    consumer) {
    ...
    consumer.consumeInsect(insect);
    ...
  }
}
```

В результаті удалося звести роботу з однією властивістю трьох різнорідних класів в одне місце, зробити код більш універсальним. Звернете увагу, що при додаванні ще одного насекомоядного така модель зоопарку не зажадає жодних змін, аби обслуговувати новий вигляд, на відміну від первинного громіздкого рішення. Завдяки введенню інтерфейсу удалося відокремити класи, що реалізовують його (живі організми) і що використовують його (службовець зоопарку). Після будь-яких змін цих класів за умови збереження інтерфейсу їх взаємодія не порушиться.

Даний приклад ілюструє, як інтерфейси надають альтернативний, строгіший і гнучкіший підхід замість множинного спадкоємства.

Поліморфізм

Раніше були розглянуті правила оголошення класів з врахуванням їх спадкоємства. У цій лекції було введено поняття перевизначеного методу. Проте поліморфізм вимагає глибшого вивчення. При оголошенні однойменних полів або методів із співпадаючими сигнатурами відбувається перекриття елементів з батьківського класу, що успадковує. Розглянемо, як функціонують класи і об’єкти в таких ситуаціях.

Поля

Почнемо з полів, які можуть бути статичними або динамічними. Розглянемо приклад:

```
class Parent {
  int a=2;
}
class Child extends Parent {
  int a=3;
}
```

Перш за все, потрібно сказати, що таке оголошення коректне. Спадкоємці можуть оголошувати поля з будь-якими іменами, навіть співпадаючими з батьківськими. Потім, необхідно зрозуміти, як два однойменні поля співіснуватимуть. Дійсно, об'єкти класу Child міститимуть відразу дві змінних, а оскільки вони можуть відрізнятися не лише значенням, але і типом (адже це два незалежні поля), саме компілятор визначатиме, яке із значень використовувати. Компілятор може спиратися лише на типа заслання, за допомогою якого відбувається звернення до поля:

```
Child з = new Child();
System.out.println(c.a);
Parent p = з;
System.out.println(p.a);
```

Обоє заслання вказують на один і той же об'єкт, породжений від класу Child, але одна з них має такого ж типа, а інша – Parent. Звідси слідують і результати:

```
3
2
```

Оголошення поля в класі-спадкоємцеві "приховало" батьківське поле. Дане оголошення так і називається –"скрывающим" (hiding). Це особливий випадок перекриття зон видимості, відмінний від "затінюючого" (shadowing) і "затуляючого" (obscuring) оголошень. Проте, батьківське поле продовжує існувати. До нього можна звернутися і явно:

```
class Child extends Parent {
  int a=3;
    // приховуюче оголошення
  int b=((Parent)this).a;
    // громіздкіше оголошення
  int c=super.a;
    // простіше
}
```

Змінні b і з набудуть значення, що зберігається в батьківському полі a. Хоча вираження з super простіше, воно не дозволить звернутися на два рівні вгору по дереву спадкоємства. Адже цілком можливо, що в батьківському класі це поле також було таким, що приховує і в батьку батька зберігається ще одне значення. До нього можна звернутися явним приведенням, як це робиться для b.

Розглянемо наступний приклад:

```
class Parent {
  int x=0;
  public void printX() {
    System.out.println(x);
  }
```

```
}
class Child extends Parent {
  int x=-1;
}
```

Який буде результат наступних рядків?

```
new Child().printX();
```

Значення якого поля буде роздруковано? Метод викликається за допомогою заслання типа Child, але це не зіграє жодної ролі. Викликається метод, визначений в класі Parent, і компілятор, звичайно, розцінив звернення до поля x в цьому методі саме як до поля класу Parent. Тому результатом буде 0.

Перейдемо до статичних полів. Насправді, для них проблем і конфліктів, пов'язаних з поліморфізмом, не існує.

Розглянемо приклад:

```
class Parent {
  static int a=2;
}
class Child extends Parent {
  static int a=3;
}
```

Який буде результат наступних рядків?

```
Child з = new Child();
System.out.println(c.a);
Parent p = з;
System.out.println(p.a);
```

Потрібно пригадати, як компілятор обробляє звернення до статичних полів через посилальні значення. Неважливо, на який об'єкт вказує заслання. Більш того, вона може бути навіть рівна null. Все визначається типом заслання.

Тому даний приклад еквівалентний:

```
System.out.println(Child.a)
System.out.println(Parent.a)
```

А його результат сумнівів вже не викликає:

```
3
```

```
2
```

Можна привести наступне пояснення. Статичне поле належить класу, а не об'єкту. В результаті появи класів-спадкоємців з приховуючими (hiding) оголошеннями ніяк не позначається на роботі з вихідним полем. Компілятор завжди може визначити, через заслання якого типа відбувається звернення до нього.

Звернете увагу на наступний приклад:

```
class Parent {
  static int a;
}

class Child extends Parent {
}
```

Який буде результат наступних рядків?

```
Child.a=10;
Parent.a=5;
System.out.println(Child.a);
```

В даному прикладі поле a не було приховано і передалося по спадку класу Child. Проте результат показує, що це все ж одне поле:

```
5
```

Не дивлячись на те, що до поля класу йдуть звернення через різні класи, змінна всього одна.

Отже, спадкоємці можуть оголошувати поля з іменами, співпадаючими з батьківськими полями. Такі объявленияназывают що приховують. При цьому об'єкти міститимуть обоє значення, а компілятор кожного разу визначатиме, з яким з них треба працювати.

Методи

Розглянемо випадок перевизначення (overriding) методів:

```
class Parent {
  public int getValue() {
    return 0;
  }
}
class Child extends Parent {
  public int getValue() {
```

```
    return 1;
  }
}
```

І рядки, що демонструють роботу з цими методами:

```
Child з = new Child();
System.out.println(з.getValue());
Parent p = з;
System.out.println(p.getValue());
```

Результатом буде:

```
1
1
```

Можна бачити, що батьківський метод повністю перекритий, значення 0 ніяк не можна отримати через заслання, вказуюче на об'єкт класу Child. У цьому ключова особливість поліморфізму – спадкоємці можуть змінювати батьківську поведінку, навіть якщо звернення до них виробляється по засланню батьківського типа. Нагадаємо, що, хоча старий метод зовні вже недоступний, усередині класу-спадкоємця до нього все ж можна звернутися за допомогою super.

Розглянемо складніший приклад:

```
class Parent {
  public int getValue() {
    return 0;
  }
  public void print() {
    System.out.println(getValue());
  }
}

class Child extends Parent {
  public int getValue() {
    return 1;
  }
}
```

Що з'явиться на консолі після виконання наступних рядків?

```
Parent p = new Child();
p.print();
```

За допомогою заслання типа Parent викликається метод print(), оголошений в класі Parent. З цього методу робиться звернення до getValue(), яке в класі Parent повертає 0. Але компілятор вже не може передбачити, до динамічного методу якого класу станеться звернення під час роботи програми. Це визначає віртуальна машина на основі об'єкту, на який вказує заслання. І раз цей об'єкт породжений отChild, то існує лише один метод getValue().

Результатом роботи прикладу буде:

```
1
```

Даний приклад демонструє, що перевизначення методів повинне вироблятися з обережністю. Якщо дуже сильно змінити логіку їх роботи, порушити прийняті угоди (наприклад, почати повертати null як значення посилального типа, якщо батьківський метод такого не допускав), це може привести до збоїв в роботі батьківського класу, а значить, об'єкту спадкоємця. Більш того, існують і деякі обов'язкові обмеження.

Пригадаємо, що заголовок методу складається з модифікаторів, повертаного значення, сигнатури і throws-выражения. Сигнатура (ім'я і набір аргументів) залишається незмінною, якщо говорити про перевизначення. Повертане значення також не може мінятися, інакше це приведе до появи двох різних методів з однаковими сигнатурами.

Розглянемо модифікатори доступу.

```
class Parent {
  protected int getValue() {
    return 0;
  }
}

class Child extends Parent {
  /* ??? */ protected int getValue() {
    return 1;
  }
}
```

Хай батьківський метод був оголошений як protected. Зрозуміло, що метод спадкоємця можна залишити з таким же рівнем доступу, але чи можна його розширити (public), або звузити (доступ за умовчанням)? Декілька рядків для перевірки:

```
Parent p = new Child();
p.getValue();
```

Звернення до методу здійснюється за допомогою заслання типа Parent. Саме компілятор виконує перевірку рівня доступу, і він орієнтуватиметься на батьківський клас. Але заслання-то вказує на об'єкт, породжений від Child, і по правилах поліморфізму виконуватися буде метод саме цього класу. А значить, доступ до перевизначеного методу

не може бути більш обмеженим, ніж до початкового. Отже, методи з доступом за умовчанням можна перевизначати з таким же доступом, або protected або public. Protected-методы перевизначаються такими ж, або public, а для public міняти модифікатор доступу і зовсім не можна.

Що стосується private-методов, то вони визначені лише усередині класу, зовні не видні, а тому спадкоємці можуть без обмежень оголошувати методи з такими ж сигнатурами і довільними повертаними значеннями, модифікаторами доступу і так далі

Аналогічні обмеження накладаються і на throws-выражение, яке буде розглянуто в наступних лекціях.

Якщо абстрактний метод перевизначається неабстрактним, то говорять, що він його реалізував (implements). Як не дивно, абстрактний метод може перевизначити інший абстрактний, або навіть неабстрактний, метод. У першому випадку така дія може мати сенс лише при зміні модифікатора доступу (розширенні), либоthrows-выражения. У другому випадку повністю втрачається стара реалізація методу, що може потрібно в особливих випадках.

Перейдемо до статичних методів. Розглянемо приклад:

```
class Parent {
  static public int getValue() {
    return 0;
  }
}

class Child extends Parent {
  static public int getValue() {
    return 1;
  }
}
```

І рядки, що демонструють роботу з цими методами:

```
Child з = new Child();
System.out.println(з.getValue());
Parent p = з;
System.out.println(p.getValue());
```

Аналогічно злучаю із статичними змінними, згадуємо алгоритм обробки компілятором таких звернень до статичних елементів і отримуємо, що код еквівалентний наступним рядкам:

```
System.out.println(Child.getValue());
System.out.println(Parent.getValue());
```

Результатом буде:

```
1
0
```

Тобто статичні методи, подібно до статичних полів, належать класу і поява спадкоємців на них не позначається.

Статичні методи не можуть перекривати звичайні, і навпаки.

Поліморфізм і об’єкти

На закінчення розглянемо декілька особливостей, витікаючих з властивостей поліморфізму.

По-перше, тепер можна точно сформулювати, що є елементами посилального типа. Посилальний тип володіє наступними елементами:

- безпосередньо оголошеними в його телі;
- оголошеними в його батьківському класі і інтерфейсах, що реалізовуються , окрім:
  - o private-элементов;
  - o "прихованих" елементів (полів і статичних методів, прихованих однойменними елементами);
  - o перевизначених (динамічних) методів.

По-друге, продовжимо розглядати взаємозв’язок типа змінною і типів її можливих значень. До випадків, описаних в попередній лекції, додаються ще два. Змінна типа абстрактний клас може посилатися на об’єкти, породжені неабстрактним спадкоємцем цього класу. Змінна типа інтерфейс може посилатися на об’єкти, породжені від класу, що реалізовує даний інтерфейс.

Зведемо ці дані в таблицю.

Таблиця 8.1. Взаємозв’язок типа змінною і типів її можливих значень.

Тип змінної	**Допустимі типи її значення**
Абстрактний клас	• null • неабстрактний спадкоємець •
Інтерфейс	• null • класи, що реалізовують інтерфейс, а саме: • що реалізовують безпосередньо (заголовок містить implements);

	• успадковані від реалізовуючих класів; • що реалізовують спадкоємців цього інтерфейсу; • змішаний випадок - спадкоємство від класу, що реалізовує спадкоємця інтерфейсу •

Таким чином, Java надає гнучку і потужну модель об'єктів, що дозволяє проектувати найскладніші системи. Необхідно добре знатися на її основних властивостях і механізмах – спадкоємство, статичні елементи, абстрактні елементи, інтерфейси, поліморфізм, розмежування доступу та інші. Всі вони дозволяють уникати дублюючої коди, полегшують розвиток системи, додавання нових можливостей і зміну старих, допомагають забезпечувати мінімальну зв'язність між частинами системи, тобто підвищують модульність. Також вдалі технічні рішення можна багато разів використовувати в різних системах, скорочуючи і спрощуючи процес їх створення.

Для досягнення таких важливих цілей потрібне не лише знання Java, але і володіння об'єктно-орієнтованим підходом, основними способами проектування систем і перевірки якості архітектурних рішень. Платформа Java є основою і вельми зручним інструментом для вживання всіх цих технологій.

Висновок

У цій лекції були розглянуті особливості об'єктної моделі Java. Це, по-перше, статичні елементи, що дозволяють використовувати интерфейскласса без створення об'єктів. Потрібно пам'ятати, що, хоча для звернення до статичних елементів можна задіювати посилальну змінну, насправді її значення не використовується, компілятор грунтується лише на її типові.

Для правильної роботи із статичними елементами вводяться поняття статичного і динамічного контексту.

Далі розглядалося використання ключових слів this і super. Вираження this надає заслання, вказуюче на об'єкт, в контексті якого воно зустрічається. Ця конструкція допомагає уникати конфліктів імен, а також застосовується в конструкторах.

Слово super дозволяє задіювати властивості батьківського класу, що необхідне для реалізації перевизначених методів, а також в конструкторах.

Потім було введено поняття абстрактного методу і класу. Абстрактний метод не має тіла, він лише вказує, що метод з такою сигнатурою має бути реалізований в класі-спадкоємцеві. Оскільки він не має власної реалізації, класи з абстрактними методами також мають бути оголошені з модифікатором abstract, який вказує, що від них не можна породжувати об'єкти. Основна мета абстрактних методів – описати в батьківському класі якомога більше загальних властивостей спадкоємців, нехай навіть і у вигляді заголовків методів без реалізації.

Наступне важливе поняття – особливий тип в Java, інтерфейс. Його ще називають повністю абстрактним класом, оскільки всі його методи обов'язково абстрактні, а поля final static. Відповідно, на основеинтерфейсов неможливо створювати об'єкти.

Інтерфейси є альтернативою множинному спадкоємству. Класи не можуть мати більш за одного батька, але вони можуть реалізовувати скільки завгодно інтерфейсів. Таким чином, інтерфейси описують загальні властивості класів, що не знаходяться на одній гілці дерева спадкоємства.

Нарешті, важливою властивістю об'єктної моделі є поліморфізм. Було детально вивчено поведінку полів і методів, як статичних, так і динамічних, при перевизначенні. Що дозволило перейти до питання відповідності типів змінній і її значення.

ТЕМА. *Масиви.*

План

1. *Масиви як тип даних в Java.*
2. *Особливості ініціалізації масивів.*
3. *Створення та оперування даними.*
4. *Механізм клонування.*

Рекомендована література: 1, 2**,** 3, 4.

Теоретичні відомості

На відміну від звичайних змінних, які зберігають лише одне значення, масиви (arrays) використовуються для зберігання цілого набору значень. Кількість значень в масиві називається його довжиною, самі значення – елементами масиву. Значень може не бути зовсім, в цьому випадку масив вважається порожнім, а його довжина рівною нулю.

Елементи не мають імен, доступ до них здійснюється по номеру індексу. Якщо масив має довжину n, відмінну від нуля, то коректними значеннями індексу є числа від 0 до n-1. Всі значення мають однакового типа і говориться, що масив заснований на цьому базовому типові. Масиви можуть бути засновані як на примітивних типах (наприклад, для зберігання числових значень 100 вимірів), так і на посилальних (наприклад, якщо потрібно зберігати опис 100 автомобілів в гаражі у вигляді екземплярів класу Car). Відразу домовимося, що в Java масив символів char[] і клас String являются різними типами. Їх значення можуть легко конвертуватися один в одного за допомогою спеціальних методів, але все таки вони не відносяться до ідентичних типів.

Як вже говорилося, масиви в Java є об’єктами (примітивних типів в Java всього вісім і їх кількість не міняється), їх тип безпосередньо успадковується від класу Object, тому всі елементи даного класу доступні в об’єктів-масивів. Базовий тип також може бути масивом. Таким чином конструюється масив масивів, або багатовимірний масив. Робота з будь-яким масивом включає звичайні операції, вже описані для інших типів, - оголошення, ініціалізація і так далі Почнемо послідовно вивчати їх в додатку до масивів.

Оголошення масивів. Як приклад розглянемо оголошення змінній типа "масив, заснований на примітивному типові int":

```
int a[];
```

Як ми бачимо, спочатку вказується базовий тип. Потім йде ім’я змінної, а пара квадратних дужок вказує на те, що використовуваний тип є саме масивом. Також допустимий запис:

```
int[] a;
```

Кількість пар квадратних дужок вказує на розмірність масиву. Для багатовимірних масивів допускається змішаний запис:

```
int[] a[];
```

Змінна a має типа "двовимірний масив, заснований на int". Аналогічно оголошуються масиви з базовим об'єктним типом:

```
Point p, p1[], p2[][];
```

Створення змінної типа масив ще не створює екземпляри цього масиву. Такі змінні мають об'єктного типа і зберігають заслання на об'єкти, проте спочатку мають значення null (якщо вони є полями класу; нагадаємо, що локальні змінні необхідно явно ініціалізувати). Аби створити екземпляр масиву, потрібно скористатися ключовим словом new, після чого вказується тип масиву і в квадратних дужках – довжина масиву.

```
int a[]=new int[5];
Point[] p = new Point[10];
```

Змінна ініціалізувалася засланням, вказуючим на тільки що створений масив. Після його створення можна звертатися до елементів, використовуючи заслання на масив, далі в квадратних дужках вказується індекс елементу. Індекс міняється від нуля, пробігаючи всю довжину масиву, до максимально допустимого значення, на одиницю меншого довжини масиву.

```
int array[]=new int[5];
  for (int i=0; i<5; i++) {
    array[i]=i*i;
}
for (int j=0; j<6; j++) {
  System.out.println(j+"*"+j+"="+array[j]);
}
```

Результатом виконання програми буде:

```
0*0=0
1*1=1
2*2=4
3*3=9
4*4=16
```

І далі з'явиться помилка часу виконання, оскільки індекс перевищить максимально можливе для такого масиву значення. Перевірка, чи не виходить індекс за допустимі межі, відбувається лише під час виконання програми, тобто компілятор не намагається виявити цю помилку навіть в таких явних випадках, як:

```
int i[]=new int[5];
i[-2]=0;   // помилка! індекс не може
          // бути негативним
```

Помилка виникне лише на етапі виконання програми.

Хоча при створенні масиву необхідно вказувати його довжину, це значення не входить у визначення типа масиву, важлива лише розмірність. Таким чином, одна змінна може посилатися на масиви різної довжини:

```
int i[]=new int[5];
...
i=new int[7];   // змінна та ж, довжина
                // масиву інша
```

Проте для об'єкту масиву довжина обов'язково повинна вказуватися при створенні і вже ніяк не може бути змінена. У останньому прикладі для привласнення змінного заслання на масив більшої довжини потрібно було створити новий екземпляр.

Оскільки для екземпляра масиву довжина є постійною характеристикою, для всіх масивів існує спеціальне поле length, що дозволяє взнати її значення. Наприклад:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
  p[i]=new Point(i, i);
}
```

Значення індексу масиву завжди має типа int. При зверненні до елементу можна також використовувати byte,short або char, оскільки ці типи автоматично розширюються до int. Спроба задіювати long приведе до помилки компіляції.

Відповідно, і поле length має типа int, а теоретична максимально можлива довжина масиву равняется231-1, тобто дещо більше 2 млрд.

Продовжуючи розглядати типа масиву, підкреслимо, що як базовий тип може використовуватися будь-який тип Java, у тому числі:

- інтерфейси. В такому разі елементи масиву можуть мати значення null або посилатися на об'єкти будь-якого класу, що реалізовує цей інтерфейс;
- абстрактні класи. В цьому випадку елементи масиву можуть мати значення null або посилатися на об'єкти будь-якого неабстрактного класу-спадкоємця.

Оскільки масив є об'єктним типом даних, його значення можуть бути приведені до типа Object або, що те ж саме, привласнені змінною типа Object. Наприклад

```
Object про = new int[4];
```

Це дає цікаву можливість для масивів, заснованих на типові Object, зберігати як елемент заслання на самого себе:

```
Object arr[] = new Object[3];
arr[0]=new Object();
arr[1]=null;
```

```
arr[2]=arr;   // Елемент посилається
          // на весь масив!
```

Ініціалізація масивів

Тепер, коли ми з'ясували, як створювати екземпляри масиву, розглянемо, які значення приймають його елементи.

Якщо створити масив на основі примітивного числового типа, то спочатку після створення всі елементи масиву мають значення за умовчанням, тобто 0. Якщо масив оголошений на основі примітивного типа boolean, то і в цьому випадку всі елементи матимуть значення за умовчанням false. Вище розглядався приклад ініціалізації елементів за допомогою циклу for.

Розглянемо створення масиву на основі посилального типа. Передбачимо, це буде клас Point. При створенні екземпляра масиву із застосуванням ключового слова new не створюється жоден об'єкт класу Point, створюється лише один об'єкт масиву. Кожен елемент масиву матиме порожнє значення null. У цьому можна переконатися на простому прикладі:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
  System.out.println(p[i]);
}
```

Результатом будуть лише слова null.

Далі потрібно ініціалізувати елементи масиву окремо, наприклад, в циклі. Взагалі, створення масиву довжиною n можна розглядати як заклад n змінних і працювати з елементами масиву (у останньому прикладі p[i]) по правилах звичайних змінних.

Крім того, існує і інший спосіб створення масивів – ініціалізатори. В цьому випадку ключове слово newне використовується, а ставляться фігурні дужки, і в них через кому перераховуються значення всіх елементів масиву. Наприклад, для числового масиву явна ініціалізація записується таким чином:

```
int i[]={1, 3, 5};
int j[]={};   // еквівалентно new int[0]
```

Довжина масиву обчислюється автоматично, виходячи з кількості введених значень. Далі створюється масив такої довжини і кожному його елементу привласнюється вказане значення.

Аналогічно можна породжувати масиви на основі об'єктних типів, наприклад:

```
Point p=new Point(1,3);
Point arr[]={p, new Point(2,2), null, p};
// або
String sarr[]={"aaa", "bbb", "cde"+"xyz"};
```

Проте ініціалізатор не можна використовувати для анонімного створення екземплярів масиву, тобто не для ініціалізації змінної, а, наприклад, для передачі параметрів методу або конструктора.

Наприклад:

```
public class Parent {
  private String[] values;

  protected Parent(String[] s){
    values=s;
  }
}

public class Child extends Parent {

  public Child(String firstName
           String lastName) {
    super(???);
    // потрібне анонімне створення масиву
  }
}
```

У конструкторі класу Child необхідно здійснити звернення до конструктора батька і передати як параметр заслання на масив. Теоретично можна передати null, але це приведе в більшості випадків до некоректної роботи класів. Можна вставити вираження new String[2], але тоді замість значень firstName иlastName будуть передані порожні рядки. Спроба записати {firstName, lastName} приведе до помилки компіляції, так можна лише ініціалізувати змінні.

Коректне вираження виглядає так:

```
new String[]{firstName, lastName}
```

Що є деякою сумішшю вираження, що створює масиви за допомогою new, і ініціалізатора. Довжина масиву визначається кількістю вказаних значень.

Багатовимірні масиви

Тепер перейдемо до розгляду багатовимірних масивів. Так, в наступному прикладі

```
int i[][]=new int[3][5];
```

змінна і посилається на двовимірний масив, який можна уявити собі у вигляді таблиці 3x5. Сумарно в такому масиві міститься 15 елементів, до яких можна звертатися через комбінацію індексів від (0, 0) до(2, 4). Приклад заповнення двовимірного масиву через цикл:

```
int pithagor_table[][]=new int[5][5];
```

```
for (int i=0; i<5; i++) {
  for (int j=0; j<5; j++) {
    pithagor_table[i][j]=i*j;
    System.out.print(pithagor_table[i][j]+
      "\t");
  }
  System.out.println();
}
```

Результатом виконання програми буде:

```
0   0   0   0   0
0   1   2   3   4
0   2   4   6   8
0   3   6   9   12
0   4   8   12  16
```

Проте такий погляд на двовимірні і багатовимірні масиви є неповним. Точніший підхід полягає в тому, що в Java немає двовимірних, і взагалі багатовимірних масивів, а є масиви, базовими типами яких є також масиви. Наприклад, тип int[] означає "масив чисел", а int[][] означає "масив масивів чисел". Пояснимо таку точку зору.

Якщо створити двовимірний масив і визначити змінну x, яка на нього посилається, то, використовуючи x і два числа в парі квадратних дужок кожне (наприклад, x[0][0]), можна звернутися до будь-якого елементу двовимірного масиву. Але в той же час, використовуючи x і одне число в парі квадратних дужок, можна звернутися до одновимірного масиву, який є елементом двовимірного масиву. Його можна проініціалізировать новим масивом з деякою іншою довжиною і таблиця перестане бути прямокутною – вона набуде довільної форми. Зокрема, можна одному з одновимірних масивів привласнити навіть значення null.

```
int x[][]=new int[3][5];
  // прямокутна таблиця
x[0]=new int[7];
x[1]=new int[0];
x[2]=null;
```

Після таких операцій масив, на який посилається змінна x, назвати прямокутним ніяк не можна. Зате добре видно, що це просто набір одновимірних масивів або значень null.

Корисно підрахувати, скільки об’єктів породжується вираженням new int[3][5]. Правильний підрахунок такий: створюється один масив масивів (один об’єкт) і три масиви чисел, кожен завдовжки 5 (три об’єкти). Разом, чотири об’єкти.

У розглянутому прикладі три з них (масиви чисел) були тут же перевизначені новими значеннями. Для таких випадків корисно використовувати спрощену форму вираження створення масивів:

```
int x[][]=new int[3][];
```

Такий запис породжує один об’єкт – масив масивів – і заповнює його значеннями null. Тепер зрозуміло, що і в цьому, і в попередньому варіанті вираження x.length повертає значення 3 – довжину масиву масивів. Далі можна за допомогою виразів x[i].length взнати довжину кожного вкладеного масиву чисел, за умови, що і неотрицательно і менше x.length, а також x[i] не рівне null. Інакше виникатимуть помилки під час виконання програми.

Взагалі, при створенні багатовимірних масивів за допомогою new необхідно вказувати всі пари квадратних дужок, відповідно кількості вимірів. Але заповненою обов’язково має бути лише крайня ліва пара, це значення задасть довжину верхнього масиву масивів. Якщо заповнити наступну пару, то цей масив заповниться не значеннями за умовчанням null, а новими створеними масивами з меншою на одиницю розмірністю. Якщо заповнена друга пара дужок, то можна заповнити третю, і так далі.

Аналогічно, для створення багатовимірних масивів можна використовувати ініціалізатори. В цьому випадку застосовується стільки вкладених фігурних дужок, скільки потрібний:

```
int i[][] = {{1,2}, null {3} {}};
```

В даному прикладі породжується чотири об’єкти. Це, по-перше, масив масивів завдовжки 4, а по-друге, три масиви чисел з довжинами 2, 1, 0, відповідно.

Всі розглянуті приклади і твердження однаково вірні для багатовимірних масивів, заснованих як на примітивних, так і на посилальних типах.

Клас масиву

Оскільки масив є об’єктним типом даних, можна спробувати уявити собі, як виглядало б оголошення класу такого типа. Насправді ці оголошення не зберігаються у файлах, або ще якому-небудь форматі. Враховуючи, що масив може бути оголошений на основі будь-якого типа і мати довільну розмірність, це фізично нездійсненно, та і не потрібний. Замість цього під час виконання додатка віртуальна машина генерує ці оголошення динамічно на основі базового типа і розмірності, а потім вони зберігаються в пам’яті у вигляді таких же екземплярів класу Class, як і для будь-яких інших типів.

Розглянемо гіпотетичне оголошення класу для масиву, заснованого на кімсь об’єктному типові Element.

Оголошення класу починається з перерахування модифікаторів, серед яких особливу роль грають модифікатори доступу. Клас масиву матиме такий же рівень доступу, як і базового типа. Тобто якщо Element оголошений як public-класс, то і масив матиме рівень доступу public. Для будь-якого примітивного типа клас масиву буде public. Можна також вказати модифікатор final, оскільки жоден клас не може успадковуватися від класу масиву.

Потім слідує ім’я класу, на якому можна детально не зупинятися, оскільки до типа масив звернення йде не по його імені, а по імені базового типа і набору квадратних дужок.

Потім потрібно вказати батьківський клас. Всі масиви успадковуються безпосередньо від класу Object. Далі перераховуються інтерфейси, які реалізує клас. Для масиву це будуть інтерфейси Cloneable иSerializable. Перший з них детально розглядається в кінці цієї лекції, а другий буде описаний в наступних лекціях.

Тіло класу містить оголошення одне public final поля length типа int. Крім того, перевизначений методclone() для підтримки інтерфейсу Cloneable.

Зведемо все вищесказане у формальний запис класу:

```
[public] class A implements Cloneable
            java.io.Serializable {
  public final int length;
    // ініціалізувався при створенні
  public Object clone() {
    try { return super.clone();}
    catch (CloneNotSupportedException e) {
      throw new InternalError(e.getMessage());
    }
  }
}
```

Таким чином, екземпляр типа масив є повноцінним об’єктом, який, зокрема, успадковує всі методи, визначені в класі Object, наприклад, toString(), hashCode() і останні.

Наприклад:

```
// результат роботи методу toString()
System.out.println(new int[3]);
System.out.println(new int[3][5]);
System.out.println(new String[2]);

// результат роботи методу hashCode()
System.out.println(new float[2].hashCode());
```

Результатом виконання програми буде:

```
[I@26b249
[[I@82f0db
[Ljava.lang.String;@92d342
7051261
```

Перетворення типів для масивів

Тепер, коли масив введений як повноцінний тип даних в Java, розглянемо, який вплив він зробить на перетворення типів.

Раніше детально розглядалися переходи між примітивними і звичайними (масивами, що немає) посилальними типами. Хоча масиви є об'єктними типами, їх також буде корисно розділити за базовим типом на дві групи – засновані на примітивному або посилальному типові.

Майте на увазі, що переходи між масивами і примітивними типами є забороненими. Перетворення між масивами і іншими об'єктними типами можливі лише для класу Object і интерфейсовCloneable і Serializable. Масив завжди можна привести до цих трьох типів, зворотний же перехід є звуженням і повинен вироблятися явним чином по розсуду розробника. Таким чином, інтерес представляють лише переходи між різними типами масивів. Вочевидь, що масив, заснований на примітивному типові, принципово не можна перетворити до типа масиву, заснованого на посилальному типові, і навпаки.

Поки не зупинятимемося на цьому детально, проте відмітимо, що перетворення між типами масивів, заснованих на різних примітивних типах, неможливі ні за яких умов.

Для посилальних же типів такого строгого правила немає. Наприклад, якщо створити екземпляр масиву, заснованого на типові Child, то заслання на нього можна привести до типа масиву, заснованого на типові Parent.

```
Child c[] = new Child[3];
Parent p[] = з;
```

Взагалі, існує універсальне правило: масив, заснований на типові A, можна привести до масиву, заснованого на типові B, якщо сам тип Априводится до типа B.

```
// якщо допустиме таке приведення:
B b = (B) new A();
// то допустимо і приведення масивів:
B b[]=(B[]) new A[3];
```

Застосовуючи це правило рекурсивно, можна перетворювати багатовимірні масиви. Наприклад, масив Child[][]можно привести до Parent[][], оскільки їх базові типи наводяться (Child[] до Parent[]) також на основі цього правила (оскільки базові типи Child і Parent наводяться через правила спадкоємства).

Як завжди, розширення можна проводити неявно (як в попередньому прикладі), а звуження – лише явним приведенням.

Повернемося до масивів, заснованих на примітивному типові. Неможливість їх участі в перетвореннях типів зв'язана, звичайно, з відмінностями між простими і посилальними типами даних. Оскільки елементами об'єктних масивів є заслання, вони легко можуть брати участь в приведенні. Навпаки, елементи простих типів дійсно зберігають числові або булеві значення. Передбачимо, таке перетворення здійсненно:

```
// приклад викличе помилку компіляції
byte b[]={1, 2, 3};
int i[]=b;
```

В такому разі, елементи b[0] і i[0] зберігали б значення різних типів. Отже, перетворення зажадало б копіювання з одночасним перетворенням типа всіх елементів вихідного масиву. В результаті був би створений новий масив, елементи якого дорівнювали б за значенням елементам вихідного масиву.

Але перетворення типа не може породжувати нові об'єкти. Такі операції повинні виконуватися лише явним чином із застосуванням ключового слова new. З цієї причини перетворення типів масивів, заснованих на примітивних типах, заборонені.

Якщо ж копіювання елементів дійсно потрібне, то потрібно спочатку створити новий масив, а потім скористатися стандартною функцією System.arraycopy(), яка ефективно виконує копіювання елементів одного масиву в інший.

Помилка ArrayStoreException

Перетворення між типами масивів, заснованих на посилальних типах, може стати причиною однієї досить неочевидної помилки.

Розглянемо приклад:

```
Child c[] = new Child[5];
Parent p[]=c;
p[0]=new Parent();
```

З точки зору компілятора код абсолютно коректний. Перетворення в другому рядку допустимо. У третьому рядку елементу масиву типа Parent привласнюється значення того ж типа.

Проте при виконанні такої програми виникне помилка. Не можна забувати, що перетворення не міняє об'єкт, змінюється лише спосіб доступу до нього. У свою чергу, об'єкт завжди "пам'ятає", від якого типа він був породжений. З врахуванням цих зауважень стає ясно, що в третьому рядку робиться спроба додати в массивChild значення типа Parent, що некоректно.

Дійсно, адже змінна з продовжує посилатися на цей масив, а значить, наступним рядком може бути таке звернення:

```
з[0].onlyChildMethod();
```

де метод onlyChildMethod() визначений лише в класі Child. Дане звернення досконале коректно, а значить, недопустима ситуація, коли елемент з[0] посилається на об'єкт, несумісний з Child.

Таким чином, не дивлячись на відсутність помилок компіляції, віртуальна машина при виконанні програми завжди здійснює додаткову перевірку перед привласненням значення елементу масиву. Необхідно упевнитися, що реальний масив, що існує на момент виконання, дійсно може зберігати привласнюване значення. Якщо ця умова порушується, то виникає помилка, яка называетсяArrayStoreException.

Може скластися враження, що розібрана ситуація є надуманою, - навіщо перетворювати масив і тут же задавати для нього невірне значення? Проте перетворення при привласненні значень є лише прикладом. Розглянемо оголошення методу:

```
public void process(Parent[] p){
  if (p!=null && p.length>0) {
    p[0]=new Parent();
  }
}
```

Метод виглядає абсолютно коректним, всі потенційно помилкові ситуації перевіряються if-выражением. Проте наступний виклик цього методу все одно наводить до помилки:

```
process(new Child[3]);
```

І це буде якраз помилка ArrayStoreException.

Змінні типа масив і їх значення

Завершимо опис взаємозв'язку типа змінною і типа значень, які вона може зберігати.

Як завжди, масиви, засновані на простих і посилальних типах, ми описуємо окремо.

Змінна типа масив примітивних величин може зберігати значення лише такого самого типа, або null.

Змінна типа "масив посилальних величин" може зберігати наступні значення:

1. null;
2. значення такого самого типа, що і тип змінної;
3. всі значення типа масив, заснований на типові, що наводиться до базового типа вихідного масиву.

Всі ці твердження безпосередньо виходять з розглянутих вище особливостей приведення типів масивів.

Ще раз нагадаємо про винятковий клас Object. Змінні такого типа можуть посилатися на будь-які об'єкти, породжені як від класів, так і від масивів.

Зведемо всі ці твердження в таблицю.

Таблиця Таблиці. 9.1.. Тип змінної і тип її значення.

Тип змінної	Допустимі типи її значення
Масив простих чисел	•null •у точності співпадаючий з типом змінної •
Масив посилальних значень	•null •співпадаючий з типом змінної •масиви посилальних значень, що задовольняють наступній умові: якщо тип змінної – масив на основі типа A, то значення типа масив на основі типа B допустимо тоді і лише тоді, коли B наводиться до A •
Object	•null •будь-який посилальний, включаючи масиви •

Клонування

Механізм клонування, як випливає з назви, дозволяє породжувати нові об'єкти на основі того, що існує, які володіли б таким самим станом, що і початковий. Тобто очікується, що для вихідного об'єкту, представленого засланням x, і результату клонування, повертаного методом x.clone(), вираження

```
x == x.clone()
```

має бути достеменним, як і вираження

```
x.clone().getClass() == x.getClass()
```

Нарешті, вираження

```
x.equals(x.clone())
```

також вірно. Реалізація такого методу clone() ускладнюється цілим рядом потенційних проблем, наприклад:

•клас, від якого породжений об'єкт, може мати всілякі конструктори, які до того ж можуть бути недоступні (наприклад, модифікатор доступу private);

- ланцюжок спадкоємства, якому належить вихідний клас, може бути задоволений довгою, і кожен батьківський клас може мати свої поля – недоступні, але важливі для відтворення стани вихідного об'єкту;
- залежно від логіки реалізації можлива ситуація, коли не всі поля повинні копіюватися для коректного клонування; одні можуть виявитися зайвими, інші зажадають додаткових обчислень або перетворень;
- можлива ситуація, коли об'єкт не можна клонувати, щоб не порушити цілісність системи.

Тому було реалізовано наступне рішення.

Клас Object містить метод clone(). Розглянемо його оголошення:

```
protected native Object clone()
   throws CloneNotSupportedException;
```

Саме він використовується для клонування. Далі можливі два варіанти.

Перший варіант: розробник може в своєму класі перевизначити цей метод і реалізувати його на свій розсуд, вирішуючи перераховані проблеми так, як того вимагає логіка системи, що розробляється. Згадані умови, які мають бути достеменними для клонованого об'єкту, не є обов'язковими і програміст може їм не слідувати, якщо це потрібно для його класу.

Другий варіант передбачає використання реалізації методу clone() у самому класі Object. Те, що він оголошений як native, говорить про те, що його реалізація надається віртуальною машиною. Природно, перераховані труднощі легко можуть бути здолані самій JVM, адже вона зберігає в пам'яті всі властивості об'єктів.

При виконанні методу clone() спочатку перевіряється, чи можна клонувати вихідний об'єкт. Якщо розробник хоче зробити об'єкти свого класу доступними для клонування через Object.clone(), то він повинен реалізувати в своєму класі інтерфейс Cloneable. У цьому інтерфейсі немає жодного елементу, він служить лише ознакою для віртуальної машини, що об'єкти можуть бути клоновані. Якщо перевірка не виконується успішно, метод породжує помилку CloneNotSupportedException.

Якщо інтерфейс Cloneable реалізований, то породжується новий об'єкт від того ж класу, від якого був створений вихідний об'єкт. При цьому копіювання виконується на рівні віртуальної машини, жодні конструктори не викликаються. Потім значення всіх полів, оголошених, успадкованих або оголошених в батьківських класах, копіюються. Отриманий об'єкт повертається як клон.

Звернете увагу, що сам клас Object не реалізує інтерфейс Cloneable, а тому спроба виклику newObject().clone() наводитиме до помилки. Метод clone() призначений швидше для використання в спадкоємцях, які можуть звертатися до нього за допомогою вираження super.clone(). При цьому можуть бути зроблені наступні зміни:

- модифікатор доступу розширений до public;
- видалено попередження про помилку CloneNotSupportedException;

- результуючий об’єкт може бути модифікований будь-яким способом, на розсуд розробника.

Нагадаємо, що всі масиви реалізують інтерфейс Cloneable і, таким чином, доступні для клонування.

Поважно пам’ятати, що всі поля клонованого об’єкту прирівнюються, їх значення ніколи не клонуються. Розглянемо приклад:

```
public class Test implements Cloneable {
  Point p;
  int height;

  public Test(int x, int в, int z){
    p=new Point(x, в);
    height=z;
  }

  public static void main(String s[]){
    Test t1=new Test(1, 2, 3), t2=null;
    try {
      t2=(Test) t1.clone();
    } catch (CloneNotSupportedException e) {}
    t1.p.x=-1;
    t1.height=-1;
    System.out.println("t2.p.x=" + t2.p.x + ",
      t2.height=" + t2.height);
  }
}
```

Результатом роботи програми буде:

```
t2.p.x=-1, t2.height=3
```

З прикладу видно, що примітивне поле було скопійоване і далі існує незалежно у вихідному і клонованому об’єктах. Зміну одну не позначається на іншому.

А ось посилальне поле було скопійоване по засланню, обоє об’єкту посилаються на один і той же екземпляр классаPoint. Тому зміни, що відбуваються з вихідним об’єктом, позначаються на клонованому.

Цього можна уникнути, якщо перевизначити метод clone() у класі Test.

```
public Object clone() {
   Test clone=null;
   try {
     clone=(Test) super.clone();
   } catch (CloneNotSupportedException e) {
     throw new InternalError(e.getMessage());
   }
```

```
    clone.p=(Point)clone.p.clone();
    return clone;
  }
```

Звернете увагу, що результат методу Object.clone() доводиться явно наводити до типа Test, хоча його реалізація гарантує, що клонований об'єкт буде породжений саме від цього класу. Проте тип повертаного значення в даному методі для універсальності оголошений як Object, тому явне звуження необхідне.

Тепер метод main можна спростити:

```
public static void main(String s[]){
  Test t1=new Test(1, 2, 3);
  Test t2=(Test) t1.clone();
  t1.p.x=-1;
  t1.height=-1;
  System.out.println("t2.p.x=" + t2.p.x +
    ", t2.height=" + t2.height);
}
```

Результатом буде:

```
t2.p.x=1, t2.height=3
```

Тобто тепер всі поля вихідного і клонованого об'єктів стали незалежними.

Реалізація такого "неглибокого" клонування в методі Object.clone() необхідна, оскільки інакше клонування другорядного об'єкту могло б привести до величезних витрат ресурсів, адже цей об'єкт може містити заслання на значиміші об'єкти, а ті при клонуванні також почали б копіювати свої поля, і так далі. Крім того, типом поля клонованого об'єкту може бути клас, не реалізовуючий Cloneable, що наводило б до додаткових проблем. Як показано в прикладі, при необхідності додаткове копіювання можна додати самостійно.

Клонування масивів

Отже, будь-який масив може бути клонований. У цьому розділі хотілося б розглянути особливості, що виникають через те, що Object.clone() копіює лише один об'єкт.

Розглянемо приклад:

```
int a[]={1, 2, 3};
int b[]=(int[])a.clone();
a[0]=0;
System.out.println(b[0]);
```

Результатом буде одиниця, що цілком очевидно, оскільки весь масив представлений одним об'єктом, який не залежатиме від своєї копії. Ускладнюємо приклад:

```
int a[][]={{1, 2}, {3}};
int b[][]=(int[][]) a.clone();

if (...) {
  // перший варіант:
  a[0]=new int[]{0};
  System.out.println(b[0][0]);
} else {
  // другий варіант:
  a[0][0]=0;
  System.out.println(b[0][0]);
}
```

Розберемо, що відбуватиметься в цих двох випадках. Почнемо з того, що в першому рядку створюється двомірний масив, що складається з двох одновимірних. Разом три об'єкти. Потім, на наступному рядку при клонуванні буде створений новий двомірний масив, що містить заслання на ті ж самі одновимірні масиви.

Тепер нескладно передбачити результат обох варіантів. У першому випадку у вихідному масиві міняється заслання, що зберігається в першому елементі, що не принесе жодних змін для клонованого об'єкту. На консолі з'явиться 1.

У другому випадку модифікується існуючий масив, що позначиться на обох двомірних масивах. На консолі з'явиться 0.

Звернете увагу, що якщо з прикладу прибрати умову if-else, так, щоб відпрацьовував перший варіант, а потім другий, то результатом буде знову 1, оскільки в частині другого варіанту модифікуватися буде вже новий масив, породжений в частині першого варіанту.

Таким чином, в Java надається потужний, ефективний і гнучкий механізм клонування, який легко застосовувати і модифікувати під конкретні потреби. Особлива увага повинна приділятися копіюванню об'єктних полів, які за умовчанням копіюються лише по засланню.

Висновок

У цій лекції було розглянуто пристрій масивів в Java. Подібно до масивів в інших мовах, вони є набором значень одного типа. Основною властивістю масиву є довжина, яка в Java може дорівнювати нулю. Інакше, масив володіє елементами в кількості, рівній довжині, до яких можна звернутися, використовуючи індекс, що змінюється від 0 до величини довжини без одиниці. Довжина задається при створенні масиву і в створеного масиву не може бути змінена. Проте вона не входить у визначення типа, а тому одна змінна може посилатися на масиви одного типа з різною довжиною.

Створити масив можна за допомогою ключового слова new, оскільки всі масиви, включаючи визначені на основі примітивних значень, мають об'єктного типа. Інший спосіб – скористатися ініціалізатором і перерахувати значення всіх елементів. У першому випадку елементи набувають значень за умовчанням (0, false, null).

Особливим чином в Java влаштовані багатовимірні масиви. Вони, по суті, є одновимірними, заснованими на масивах меншої розмірності. Такий підхід дозволяє одноманітно працювати з багатовимірними масивами. Також він дає можливість створювати не лише "прямокутні" масиви, але і масиви будь-якої конфігурації.

Хоча масив і є посилальним типом, робота з ним частенько має деякі особливості. Розглядаються правила приведення типа масиву. Як для будь-якого об’єктного типа, приведення до Object є таким, що розширює. Приведення масивів, заснованих на посилальних типах, багато в чому підкоряється звичайним правилам. А ось примітивні масиви перетворювати не можна. З перетвореннями зв’язано і виникнення помилки ArrayStoreException, причина якої – неможливість точного відстежування типів в перетвореному масиві для компілятора.

На закінчення розглядаються останні випадки взаємозв’язку типа змінною і її значення.

Нарешті, вивчається механізм клонування, що існує з найперших версій Java і дозволяє створювати точні копії об’єктів, якщо їх класи дозволяють це робити, реалізовуючи інтерфейс Cloneable. Оскільки стандартне клонування породжує лише один новий об’єкт, це наводить до особливих ефектів при роботі з об’єктними полями класів і масивами.

ТЕМА. *Оператори і структура коду.*

План

1. *Мітки, оператори умовного переходу.*
2. *Цикли, оператори break і continue.*
3. *Помилки при роботі програми. Виключення .*

Рекомендована література: 1, 2, 3, 4.

Теоретичні відомості

Управління потоком обчислень є фундаментальною основою всієї мови програмування. У даній лекції будуть розглянуті основні мовні конструкції і способи їх вживання. Порядок виконання програми визначається операторами. Оператори можуть містити інших операторів або вирази.

Нормальне і перерване виконання операторів

Послідовність виконання операторів може бути безперервною, а може і уриватися (при виникненні певних умов). Виконання оператора може бути перерване, якщо в потоці обчислень будуть виявлені оператори

break
continue
return

Тоді управління буде передано в інше місце (відповідно до правил обробки цих операторів, яких ми розглянемо пізніше).

Нормальне виконання оператора може бути перерване також при виникненні виняткових ситуацій, які теж будуть розглянуті пізніше. Явне збудження виняткової ситуації за допомогою оператора throw також перериває нормальне виконання оператора і передає управління виконанням програми (далі просто управління) в інше місце.

Переривання нормального виконання завжди викликається певною причиною. Приведемо список таких причин:

- break (без вказівки мітки);
- break (з вказівкою мітки);
- continue (без вказівки мітки);
- continue (з вказівкою мітки);
- return (з поверненням значення);
- return (без повернення значення);
- throw з вказівкою об'єкту Throwable, а також всі виключення, що викликаються віртуальною машиною Java.

Вирази можуть завершуватися нормально і передчасно (аварійно). В даному випадку термін "аварійно" сповна застосовний, оскільки причиною незвичайної послідовності виконання вираження може бути лише виникнення виняткової ситуації. Якщо в операторові міститься вираження, то в разі його аварійного завершення виконання оператора теж буде завершено передчасно (тобто нормальний хід виконання оператора буде порушений). В тому випадку, якщо в операторові є вкладений оператор і його завершення відбувається ненормально, то так само ненормально завершується оператор, вкладений, що містить (в деяких випадках це не так, що обмовлятиметься особливо).

Блоки і локальні змінні

Блок - це послідовність операторів, оголошень локальних класів або локальних змінних, ув'язнених в дужки. Зона видимості локальних змінних і класів обмежена блоком, в якому вони визначені.

Оператори в блоці виконуються зліва направо, зверху вниз. Якщо всі оператори (вирази) в блоці виконуються нормально, то і весь блок виконується нормально. Якщо який-небудь оператор (вираження) завершується ненормально, то і весь блок завершується ненормально. Не можна оголошувати декілька локальних змінних з однаковими іменами в межах видимості блоку. Приведений нижче код викличе помилку часу компіляції.

```
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    int x;
    lbl: {
      int x = 0;
      System.out.println("x = " + x);
    }
  }
}
```

В той же час не слід забувати, що локальні змінні перекривають видимість змінних-членів. Так, наступний приклад відпрацює нормально.

```
public class Test {
  static int x = 5;
  public Test() { }
  public static void main(String[] args){
    Test t = new Test();
    int x = 1;
    System.out.println("x = " + x);
  }
 }
```

На консоль буде виведено x = 1.

Те ж саме правило застосовно до параметрів методів.

```
public class Test {
  static int x;
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    t.test(5);
    System.out.println("Member value x = "
      + x);
  }
  private void test(int x){
    this.x = x + 5;
    System.out.println("Local value x = "
      + x);
  }
}
```

В результаті роботи цього прикладу на консоль буде виведено:

```
Local value x = 5
Member value x = 10
```

На наступному прикладі продемонструємо, що зона видимості локальної змінної обмежена зоною видимості блоку, або оператора, в межах якого дана змінна оголошена.

```
public class Test {
  static int x = 5;
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    {
      int x = 1;
      System.out.println("First block x = "
        + x);
    }
    {
      int x = 2;
      System.out.println("Second block x ="
        + x);
    }
    System.out.print("For cycle x = ");
    for(int x =0;x<5;x++){
      System.out.print(" " + x);
    }
  }
}
```

Даний приклад відкомпілюється без помилок і на консоль буде виведений наступний результат:

```
First block x = 1
Second block x =2
For cycle x = 0 1 2 3 4
```

Слід пам'ятати, що визначення локальної змінної є виконуваний оператор. Якщо задана ініціалізація змінної, то вираження виконується зліва направо і його результат привласнюється локальній змінній. Використання неініціалізованих локальних змінних заборонене і викликає помилку компіляції.

Наступний приклад коди

```
public class Test {
  static int x = 5;
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    int x;
    int в = 5;
    if( в > 3) x = 1;
    System.out.println(x);
  }
}
```

викличе помилку часу компіляції, оскільки можливі умови, при яких змінна x може не ініціалізувати до її використання (не дивлячись на те, що в даному випадку оператор if(в > 3) і наступне за ним вираження x = 1; виконуватимуться завжди).

Порожній оператор

Крапка з комою (;) є порожнім оператором. Дана конструкція сповна застосовна там, де не передбачається виконання жодних дій. Передчасне завершення порожнього оператора неможливе.

Мітки

Будь-який оператор, або блок, може мати мітку. Мітку можна вказувати як параметр для операторів break і continue. Область видимостиметки обмежується оператором, або блоком, до якого вона відноситься. Так, в наступному прикладі ми отримаємо помилку компіляції:

```
public class Test {
  static int x = 5;
  static {
  }
  public Test() {
```

```
  }
  public static void main(String[] args){
    Test t = new Test();
    int x = 1;
    Lbl1: {
      if(x == 0) break Lbl1;
    }

    Lbl2:{
      if(x > 0) break Lbl1;
    }
  }
}
```

В разі, якщо є декілька вкладених блоків і операторів, допускається звернення з внутрішніх блоків кметкам, що відноситься до зовнішніх.

Цей приклад є сповна коректним:

```
public class Test {
  static int x = 5;
  static {

  }
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    int L2 = 0;
    Test: for(int i = 0; i< 10;i++){
      test: for(int j = 0; j< 10;j++){
        if( i*j > 50) break Test;
      }
    }
  }
  private void test() {
    ;
  }
}
```

У цьому ж прикладі можна побачити, що мітки використовують простір імен, відмінний від простору імен змінних, методів і класів.

Традиційне використання міток не рекомендується, особливо в об'єктно-орієнтованих мовах, оскільки серйозно ускладнює розуміння порядку виконання коди, а значить, і його тестування і відладку. Для Java цю заборону можна вважати не настільки строгою, оскільки найнебезпечніший оператор goto відсутній. У деяких ситуаціях (як в розглянутому прикладі з вкладеними циклами) використання міток сповна виправдано, але, звичайно, їх вживання слід обмежувати лише найнеобхіднішими випадками.

Оператор if

Мабуть, найбільш поширеною конструкцією в Java, як і в будь-якому другом структурній мові програмування, є оператор умовного переходу.

У загальному випадку конструкція виглядає так:

```
 if (логічне вираження) вираження або блок 1
else вираження або блок 2
```

Логічне вираження може бути будь-якою мовною конструкцією, яка повертає булевий результат. Відзначимо відмінність від мови C, в якій як логічне вираження можуть використовуватися різні типи даних, де відмінне від нуля вираження трактується як дійсне значення, а нуль - як помилкове. У Java можливе використання лише логічних виразів. Якщо логічне вираження набуває значення "істину", то виконується вираження або блок 1, інакше - вираження або блок 2. Друга частина оператора (else) не є обов'язковою і може бути опущена. Тобто конструкція if(x == 5) System.out.println("Five") сповна допустима.

Оператори if-else можуть каскадувати.

```
 String test = "smb";
if( test.equals("value1") {
   ...
} else if (test.equals("value2") {
   ...
} else if (test.equals("value3") {
   ...
} else {
   ...
}
```

Слід пам'ятати, що оператор else відноситься до найближчого до нього оператора if. В даному випадку остання умова else виконуватиметься, лише якщо не виконане попереднє. Завершальна конструкція elseотносится до найостаннішої умови if і буде виконана лише в тому випадку, якщо жодне з вищеперелічених умов не буде достеменним. Якщо хоч би одна з умов виконана, то все подальші виконуватися не будуть.

Наприклад:

```
 ...
int x = 5;
if( x < 4){
   System.out.println("Менше 4");
} else if (x > 4) {
   System.out.println("Більше 4");
} else if (x == 5) {
   System.out.println("Рівне 5");
} else{
```

```
    System.out.println("Інше значення");
}
```

Пропозиція "Дорівнює 5" в даному випадку надруковано не буде.

Оператор switch

Оператор switch() зручно використовувати у разі потреби множинного вибору. Вибір здійснюється на основі цілочисельного значення.

Структура оператора:

```
switch(int value){
  case const1:
    вираження або блок
  case const2:
    вираження або блок
  case constn:
    вираження або блок
  default:
    вираження або блок
}
```

Причому, фраза default не є обов'язковою.

Як параметр switch може використовуватися змінна типу byte, short, int, char або вираження. Вираження повинне зрештою повертати параметр один з вказаних раніше типів. У операторові case не можуть застосовуватися значення примітивного типа long і посилальних типів Long, String, Integer, Byte і так далі

При виконанні оператора switch виробляється послідовне порівняння значення x з константами, вказаними після case, і в разі збігу виконується вираження наступного за цією умовою. Якщо вираження виконане нормально і немає передчасного його завершення, то виробляється порівняння для подальших case. Якщо ж вираження, наступне за case, завершилося ненормально, то буде припинено виконання всього оператора switch. Якщо не виконаний жоден оператор case, то виконається оператор default, якщо він є в даному switch. Якщо оператора default немає і жодне з умов case не виконане, то жодне з виразів switch також виконано не буде.

Слід звернути увагу, що, на відміну від багатоланкового if-else, якщо яка-небудь умова case виконана, то виконання switch не припиниться, а перевірятимуться наступні за ним умови. Якщо цього необхідно уникнути, то після коди наступного за оператором case використовується оператор break, що перериває подальше виконання оператора switch. Після оператора case повинен слідувати літерал, який може бути інтерпретований як 32-бітове ціле значення. Тут не можуть застосовуватися вирази і змінні, якщо вони не є final static.

Розглянемо приклад:

```
int x = 2;
```

```
switch(x){
  case 1:
  case 2:
    System.out.println("Рівне 1 або 2");
    break;
  case 3:
  case 4:
    System.out.println("Рівне 3 або 4");
    break;
  default:
    System.out.println(
     "Значення не визначене");
}
```

В даному випадку на консоль буде виведений результат "Рівне 1 або 2". Якщо ж прибрати операторів break, то будуть виведений всі три рядки.

Ось така конструкція викличе помилку часу компіляції.

```
 int x = 5;
switch(x){
  case в:   // лише константи!
    ...
    break;
}
```

У операторові switch не може бути два case з однаковими значеннями.

Тобто конструкція

```
 switch(x){
  case 1:
  System.out.println("One");
  break;
  case 1:
  System.out.println("Two");
  break;
  case 3:
  System.out.println("Tree or other value");
}
```

недопустима.

Також в конструкції switch може бути застосований лише один оператор default.

Управління циклами

У мові Java є три основні конструкції управління циклами:

- цикл while;

- цикл do;
- цикл for.

Цикл while

Основна форма циклу while може бути представлена так:

```
while(логічне вираження)
  вираження, що повторюється, або блок;
```

У даній мовній конструкції вираження, що повторюється, або блок виконуватиметься до тих пір, поки логічне вираження матиме дійсне значення. Цей багато разів виконуваний блок називають тілом циклу. Оператори **continue і break** можуть змінювати нормального виконання тіла циклу. Так, якщо в телі циклу зустрівся оператор continue, то оператори, наступні за ним, будуть пропущені і виконання циклу почнеться спочатку. Якщо continue використовується з міткою і мітка належить до даного while, то виконання його буде аналогічним. Якщо мітка не відноситься до данному while, його виконання буде припинено і управління буде передано на оператора, або блок, до якого відноситься мітка.

Якщо зустрівся оператор **break**, то виконання циклу буде припинено.

Якщо виконання блоку було припинене по якійсь іншій причині (виникла виняткова ситуація), то виконання всього циклу буде припинено з тієї ж причини.

Розглянемо декілька прикладів:

```
public class Test {
  static int x = 5;
  public Test() {
  }
  public static void main(String[] args){
     Test t = new Test();
     int x = 0;
     while(x < 5){
     x++;
     if(x % 2 == 0) continue;
     System.out.print(" " + x);
     }
  }
}
```

На консоль буде виведено

```
1   3   5
```

тобто вивід на друк всіх парних чисел буде пропущений.

```
public class Test {
 static int x = 5;
```

```
 public Test() { }
 public static void main(String[] args){
  Test t = new Test();
  int x = 0;
  int в = 0;
  lbl: while(в  < 3){
  y++;
  while(x < 5){
   x++;
   if(x % 2 == 0) continue lbl;
   System.out.println("x=" + x + " y="+y);
   }
  }
 }
}
```

На консоль буде виведено

```
 x=1 y=1
x=3 y=2
x=5 y=3
```

тобто при виконанні умови if(x % 2 == 0) continue lbl; цикл по змінній x буде перерваний, а цикл по змінній в почне нову ітерацію.

Типовий варіант використання вираження while():

```
  int i = 0;
while( i++ < 5){
  System.out.println("Counter is " + i);
}
```

Слід пам'ятати, що цикл while() буде виконаний лише в тому випадку, якщо на момент початку його виконання логічне вираження буде достеменним. Таким чином, при виконанні програми може мати місце ситуація, коли цикл while() не буде виконаний жодного разу.

```
 boolean b = false;
while(b){
  System.out.println("Executed");
}
```

В даному випадку рядок System.out.println("Executed"); виконана не буде.

Цикл do

Основна форма циклу do має наступний вигляд:

```
 do
  вираження, що повторюється, або блок;
```

```
while(логічне вираження)
```

Цикл do виконуватиметься до тих пір, поки логічне вираження буде достеменним. На відміну від циклу while, цей цикл буде виконаний, як мінімум, один раз.

Типова конструкція циклу do:

```
 int counter = 0;
do {
  counter ++;
  System.out.println("Counter is "
    + counter);
} while(counter < 5);
```

У останньому виконання циклу do аналогічно виконанню циклу while, включаючи використання операторів breaki continue.

Цикл for

Досить часто буває необхідно змінювати значення якої-небудь змінної в заданому діапазоні і виконувати послідовність операторів, що повторюється, з використанням цієї змінної. Для виконання такої послідовності дій як не можна краще личить конструкція циклу for.

Основна форма циклу for виглядає таким чином:

```
 for(вираження ініціалізації; умова;
   вираження оновлення)
вираження, що повторюється, або блок;
```

Ключовими елементами даної мовної конструкції є пропозиції, ув’язнені в круглі дужки і розділені крапкою з комою. Вираження ініціалізації виконується до початку виконання тіла циклу. Найчастіше використовується як деяка стартова умова (ініціалізація, або оголошення змінній). Умова має бути логічним вираженням і трактується точно так, як і логічне вираження в циклewhile(). Тіло циклу виконується до тих пір, поки логічне вираження достеменне. Як і у випадку з цикломwhile(), тіло циклу може не виконатися жодного разу. Це відбувається, якщо логічне вираження набуває значення "брехню" до початку виконання циклу. Вираження оновлення виконується відразу після виконання тіла циклу і до того, як перевірена умова продовження виконання циклу. Зазвичай тут використовується вираження инкрементации, але може бути застосовано і будь-яке інше вираження.

Приклад використання циклу for():

```
 ...
for(counter=0;counter<10;counter++){
  System.out.println("Counter is "
    + counter);
}
```

У даному прикладі передбачається, що змінна counter була оголошена раніше. Цикл буде виконаний 10 разів і будуть надруковані значення лічильника від 0 до 9.

Дозволяється визначати змінну прямо в пропозиції:

```
 for(int cnt = 0;cnt < 10; cnt++){
  System.out.println("Counter is " + cnt);
}
```

Результат виконання цієї конструкції буде аналогічний попередньому. Проте потрібно звернути увагу, що зона видимості змінної cnt буде обмежена тілом циклу. Будь-яка частина конструкції for() може бути опущена. У виродженому випадку ми отримаємо оператора for з порожніми значеннями

```
 for(;;){
  ...
}
```

В даному випадку цикл виконуватиметься нескінченно. Ця конструкція аналогічна конструкції while(true){}. Умови, в яких вона може бути застосована, ми розглянемо пізніше.

Можливе також розширене використання синтаксису оператора for(). Пропозиція і вираження можуть складатися з декількох частин, розділених комами.

```
 for(i = 0, j = 0; i<5;i++, j+=2){
  ...
}
```

Використання такої конструкції сповна правомірно.

Оператори break і continue

В деяких випадках потрібно змінити хід виконання програми. У традиційних мовах програмування для цих цілей застосовується оператор goto, проте в Java він не підтримується. Для цих цілей застосовуються оператори break і continue.

Оператор continue

Оператор continue може використовуватися лише в циклах while, do, for. Якщо в потоці обчислень зустрічається оператор continue, то виконання поточної послідовності операторів (виразів) має бути припинене і управління буде передано на початок блоку, що містить цього оператора.

```
...
int x = (int)(Math.random()*10);
int arr[10] = {....}
for(int cnt=0;cnt<10;cnt++){
  if(arr[cnt]== x) continue;
```

```
  ...
}
```

В даному випадку, якщо в масиві arr зустрінеться значення, рівне x, то виконається оператор continue і всі оператори до кінця блоку будуть пропущені, а управління буде передано на початок циклу. Якщо оператор continue буде застосований поза контекстом оператора циклу, то буде видана помилка часу компіляції. В разі використання вкладених циклів операторові continue, як адреса переходу, може бути вказана мітка, що відноситься до одного з цих операторів.

Розглянемо приклад:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    for(int i=0; i < 10; i++){
      if(i % 2 == 0) continue;
      System.out.print(" i=" + i);
    }
  }
}
```

В результаті роботи на консоль буде виведено:

```
 i=1 i=3 i=5 i=7 i=9
```

При виконанні умови в рядку 7 нормальна послідовність виконання операторів буде перервана і управління буде передано на початок циклу. Таким чином, на консоль виводитимуться лише непарні значення.

Оператор break

Цей оператор, як і оператор continue, змінює послідовність виконання, але не повертає виконання до початку циклу, а перериває його.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    int [] x = {1,2,4,0,8};
    int в  = 8;
    for(int cnt=0;cnt < x.length;cnt++){
      if(0 == x[cnt]) break;
      System.out.println("y/x = " + y/x[cnt]);
    }
  }
}
```

На консоль буде виведено:

```
y/x = 8
y/x = 4
y/x = 2
```

При цьому помилки, пов'язаної з діленням на нуль, не станеться, оскільки якщо значення елементу масиву буде равно0, то буде виконано умову в рядку 9 і виконання циклу for буде перервано. Як аргумент break може бути вказана мітка. Як і у випадку з continue, не можна вказувати як аргумент мітки блоків, в яких оператор break не міститься.

Іменовані блоки

У реальній практиці досить частий використовуються вкладені цикли. Відповідно, може виникнути ситуація, коли з вкладеного циклу потрібно перервати зовнішній. Просте використання break або continue не вирішує цього завдання, проте в Java можна іменувати блок коди і явно вказати операторам, до якого з них відноситься виконувана дія. Робиться це шляхом привласнення мітки операторам do, while, for.

Мітка - це будь-яка допустима у даному контексті лексема, що закінчується двокрапкою.

Розглянемо наступний приклад:

```
 ...
int array[][] = {...};
for(int i=0;i<5;i++){
  for(j=0;j<4; j++){
    ...
    if(array[i][j]== caseValue) break;
    ...
  }
}
...
```

В даному випадку при виконанні умови буде перервано виконання циклу по j, цикл по і продовжиться з наступного значення. Для того, щоб перервати виконання обох циклів, використовується мітка:

```
...
int array[][] = {:.};
outerLoop: for(int i=0;i<5;i++){
  for(j=0;j<4; j++){
    ...
    if(array[i][j]== caseValue)
      break outerLoop;
    ...
  }
}
```

```
...
```

Оператор break також може використовуватися з іменованими блоками.

Між операторами break і continue є ще одна істотна відмінність. Оператор break може використовуватися з будь-яким іменованим блоком, в цьому випадку його дія в чомусь схоже на дію goto. Оператор continue(як і наголошувалося раніше) може бути використаний лише в телі циклу. Тобто така конструкція буде сповна прийнятною:

```
lbl:{
  ...
  if( val > maxVal) break lbl;
  ...
}
```

Тоді як оператора continue тут застосовувати не можна. В даному випадку при виконанні умови ifвыполнение блоку з міткою lbl буде перервано, тобто управління буде передано на оператора (вираження), наступного безпосередньо за закриваючою фігурною дужкою.

Мітки використовують простір імен, відмінний від простору імен класів і методів.

Так, наступний приклад коди буде сповна працездатним:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    t.test();
  }
  void test() {
    Test: {
      test: for(int i =0;true;i++){
        if(i % 2 == 0) continue test;
        if(i > 10) break Test;
        System.out.print(i + " ");
      }
    }
  }
}
```

Для складання міток застосовуються ті ж синтаксичні правила, що і для змінних, за тим виключенням, що мітки завжди закінчуються двокрапкою. Мітки завжди мають бути прив'язані до якого-небудь блоку коди. Допускається використання міток з однаковими іменами, але не можна застосовувати однакові імена в межах видимості блоку. Тобто така конструкція допустима:

```
lbl: {
```

```
  ...
  System.out.println("Block 1");
  ...
}
...
lbl: {
  ...
  System.out.println("Block 2");
  ...
}
```

А така немає:

```
lbl: {
  ...
  lbl: {
    ...
  }
  ...
}
```

Оператор return

Цей оператор призначений для повернення управління з методу, що викликається, в той, що викликає. Якщо в послідовності операторів виконується return, то управління негайно (якщо це не обумовлено особливо) передається в зухвалий метод. Оператор return може мати, а може і не мати аргументів. Якщо метод не повертає значень (оголошений як void), то в цьому і лише цьому випадку вираження return застосовується без аргументів. Якщо повертане значення є, то return обов'язково повинен застосовуватися з аргументом, чиє значення і буде повернено.

Як аргумент return може використовуватися вираження

```
return (x*y +10) /11;
```

В цьому випадку спочатку буде виконано вираження, а потім результат його виконання буде переданий в зухвалий метод. Якщо вираження буде завершено ненормально, то і оператор return буде завершений ненормально. Наприклад, якщо під час виконання вираження в операторові return виникне виключення, то жодного значення метод не поверне, оброблятиметься помилка.

У методі може бути більш за одного оператора return.

Оператор synchronized

Цей оператор застосовується для виключення взаємного впливу декількох потоків при виконанні коди, він буде детально розглянутий в лекції 12, присвяченою потокам виконання.

Помилки при роботі програми. Виключення (Exceptions)

При виконанні програми можуть виникати помилки. У одних випадках це викликано помилками програміста, в інших - зовнішніми причинами. Наприклад, може виникнути помилка ввода/вывода при роботі з файлом або мережевим з'єднанням. У класичних мовах програмування, наприклад, в З, потрібно було перевіряти деяку умову, яка вказувала на наявність помилки, і залежно від цього робити ті або інші дії.

Наприклад:

```
 ...
int statusCode = someAction();
if (statusCode){
  ... обробка помилки
} else {
  statusCode = anotherAction();
  if(statusCode){
    ... обробка помилки ...
  }
}
...
```

У Java з'явилося простіше і елегантніше рішення - обробка виняткових ситуацій.

```
 try{
  someAction();
  anotherAction();
} catch(Exception e){
  // обробка виняткової ситуації
}
```

Легко відмітити, що такий підхід є не лише витонченим, але і надійнішим і простішим для розуміння.

Причини виникнення помилок

Існує три причини виникнення виняткових ситуацій.

- Спроба виконати некоректне вираження. Наприклад, ділення на нуль, або звернення до об'єкту по засланню, рівною null, спроба використовувати клас, опис якого (class-файл) відсутній, і так далі У таких випадках завжди можна точно вказати, в якому місці сталася помилка, - саме в некоректному вираженні.
- Виконання оператора throw Цей оператор застосовується для явного породження помилки. Вочевидь, що і тут можна вказати місце виникнення виняткової ситуації.
- Асинхронні помилки під час виконання програми.

Причиною таких помилок можуть бути збої усередині самої віртуальної машини (адже вона також є програмою), або виклик методу stop() в потоку виконання Thread.

В цьому випадку неможливо вказати точне місце програми, де відбувається виняткова ситуація. Якщо ми спробуємо зупинити потік виконання (викликавши метод stop()), нам не удасться передбачити, при виконанні якого саме вираження цей потік зупиниться.

Таким чином, всі помилки в Java діляться на синхронних і асинхронних. З першими порівняно простіше працювати, оскільки принципово можливо знайти точне місце в коді, яке є причиною виникнення виняткової ситуації. Звичайно, Java є строгою мовою в тому сенсі, що всі вирази до точки збою обов'язково будуть виконані, і в той же час жодне подальше вираження ніколи виконано не буде. Поважно пам'ятати, що помилки можуть виникати як унаслідок недостатньої уважності програміста (відсутній потрібний клас, або індекс масиву вийшов за допустимі кордони), так і по незалежних від нього причинах (стався розрив мережевого з'єднання, збій апаратного забезпечення, наприклад, жорсткого диска і ін.).

Асинхронні помилки набагато складніше у виявленні і виправленні. Звичайному розробникові дуже важко виявити причини збоїв у віртуальній машині. Це можуть бути помилки творців JVM, несумісність з операційною системою, апаратний збій і багато що інше. Все ж сучасні віртуальні машини реалізовані досить добре і подібні збої відбуваються украй рідко (за умови використання якісних комплектуючих).

Аналогічна ситуація спостерігається і у випадку з примусовою зупинкою потоків виконання. Оскільки ця дія виконується операційною системою, ніколи не можна передбачити, в якому саме місці зупиниться потік. Це означає, що програма може багато разів відпрацювати коректно, а потім несподівано дати збій просто через те, що потік зупинився в якомусь іншому місці. З цієї причини примусова зупинка не рекомендується. У лекції 12 розглядаються приклади коректного управління життєвим циклом потоку.

При виникненні виняткової ситуації управління передається від коди, що викликала виняткову ситуацію, на найближчий блок catch (або вгору по стеку) і створюється об'єкт, успадкований від класса Throwable, або його нащадків (див. діаграму ієрархії класів-виключень), який містить інформацію про виняткову ситуацію і використовується при її обробці. Власне, в блоці catch вказується саме клас оброблюваної ситуації. Детально обробка помилок розглядається нижчим.

Ієрархія, по якій передається інформація про виняткову ситуацію, залежить від того, де ця виняткова ситуація виникла. Якщо це

- метод, те управління передаватиметься в те місце, де даний метод був викликаний;
- конструктор, те управління передаватиметься туди, де спробували створити об'єкт (як правило, застосовуючи оператора new);
- статичний ініціалізатор, те управління буде передано туди, де сталося перше звернення до класу, що зажадало його ініціалізації.

Допускається створення власних класів виняткових ситуацій. Здійснюється це за допомогою механізму спадкоємства, тобто клас призначеної для користувача виняткової ситуації має бути успадкований від класса Throwable, або його нащадків.

Обробка виняткових ситуацій

Конструкція try-catch

У загальному випадку конструкція виглядає так:

```
try {
  ...
} catch(SomeExceptionClass e){
  ...
} catch(AnotherExceptionClass e){
  ...
}
```

Працює вона таким чином. Спочатку виконується код, ув'язнений у фігурні дужки оператора try. Якщо під час його виконання не відбувається жодних нештатних ситуацій, то далі управління передається за закриваючу фігурну дужку останнього оператора catch, що асоціюється з даним оператором try.

Якщо в межах try виникає виняткова ситуація, то далі виконання коди виробляється по одному з перерахованих нижче сценаріїв.

Виникла виняткова ситуація, клас якої вказаний як параметр один з блоків catch. В цьому випадку виробляється виконання блоку коди, що асоціюється з даним catch (ув'язненого у фігурні дужки). Далі, якщо код в цьому блоці завершується нормально, то і весь оператор try завершується нормально і управління передається на оператора (вираження), наступного за закриваючою фігурною дужкою последнегосatch. Якщо код в catch завершується не штатно, то і весь try завершується нештатний з тієї ж причини.

Якщо виникла виняткова ситуація, клас якої не вказаний як аргумент ні в одному catch, то виконання всього try завершується нештатний.

Конструкція try-catch-finally

Оператор finally призначений для того, щоб забезпечити гарантоване виконання якого-небудь фрагмента коди. Незалежно від того, чи виникла виняткова ситуація в блоці try, чи заданий відповідний блоксatch, чи не виникла помилка в самому блоці catch, - все одно блок finally буде врешті-решт виконаний.

Послідовність виконання такої конструкції наступна: якщо оператор try виконаний нормально, то буде виконаний блок finally. У свою чергу, якщо оператор finally виконується нормально, то і весь оператор tryвыполняется нормально.

Якщо під час виконання блоку try виникає виключення і існує оператор catch, який перехоплює даного типа виключення, відбувається виконання зв'язаного з catch блоку. Якщо блок catch виконується нормально, або ненормально, все одно потім виконується блок finally. Якщо блок finally завершується нормально, то оператор try завершується так само, як завершився блок catch.

Якщо в списку операторів catch не знаходиться такого, який обробив би виникле виключення, то все одно виконується блок finally. В цьому випадку, якщо finally завершиться нормально, весь try завершиться ненормально з тієї ж причини, по якій було порушено виконання try.

У всіх випадках, якщо блок finally завершується ненормально, то весь try завершиться ненормально з тієї ж причини.

Розглянемо приклад вживання конструкції try-catch-finally.

```
try {
  byte [] buffer = new byte[128];
  FileInputStream fis =
    new FileInputStream("file.txt");
  while(fis.read(buffer) > 0){
    ... обробка даних ...
  }
} catch(IOException es){
    ... обробка виключення ...
} finally {
  fis.flush();
  fis.close();
}
```

Якщо в даному прикладі помістити операторів очищення буфера і закриття файлу відразу після закінчення обробки даних, то при виникненні помилки ввода/вывода коректного закриття файлу не станеться. Ще раз відзначимо, що блок finally буде виконаний в будь-якому разі, незалежно від того, сталася обробка виключення чи ні, виникло це виключення чи ні.

У конструкції try-catch-finally обов'язковим є використання однієї з частин оператора catch илиfinally. Тобто конструкція

```
try {
  ...
} finally {
  ...
}
```

є сповна допустимою. В цьому випадку блок finally при виникненні виняткової ситуації має бути виконаний, хоча сама виняткова ситуація оброблена не буде і буде передана для обробки на вищий рівень ієрархії.

Якщо обробка виняткової ситуації в коді не передбачена, то при її виникненні виконання методу буде припинено і виняткова ситуація буде передана для обробки коду більш високого рівня. Таким чином, якщо виняткова ситуація станеться в методі, що викликається, то управління буде передано зухвалому методу і обробку виняткової ситуації повинен виробити він. Якщо виняткова ситуація виникла в коді самого високого рівня, то управління буде передано виконуючій системі Java і виконання програми буде припинено (точніше - буде зупинений потік виконання, в якому сталася така помилка).

Використання оператора throw

Окрім того, що зумовлена виняткова ситуація може бути збуджена виконуючою системою Java, програміст сам може явно породити помилку. Робиться це за допомогою оператора throw.

Наприклад:

```
 ...
public int  calculate(int theValue){
  if( theValue < 0){
  throw new Exception(
    "Параметр для обчислення не повинен
     бути негативним");
  }
}
...
```

В даному випадку передбачається, що як параметр методу може бути передано лише позитивне значення; якщо ця умова не виконана, то за допомогою оператора throw породжується виняткова ситуація. (Для успішної компіляції також потрібно в заголовку методу вказати throws Exception - цей вираз розглядається нижчим.)

Метод повинен делегувати обробку виняткової ситуації коду, що викликав його. Для цього в сигнатурі методу застосовується ключове слово throws, після якого мають бути перераховані через кому всі виняткові ситуації, які може викликати даний метод. Тобто наведений вище приклад має бути приведений до наступного вигляду:

```
 ...
public int  calculate(int theValue)
  throws Exception {
   if( theValue < 0){
     throw new Exception(
      "Some descriptive info");
   }
}
...
```

Таким чином, створення виняткової ситуації в програмі виконується за допомогою оператора throw з аргументом, значення якого може бути приведене до типа Throwable.

В деяких випадках після обробки виняткової ситуації може виникнути необхідність передати інформацію про неї в зухвалий код.

В цьому випадку помилка з’являється повторно.

Наприклад:

```
 ...
```

```
try {
  ...
} catch(IOException ex){
  ...
  // Обробка виняткової ситуації
  ...
  // Повторне збудження винятковою
  // ситуації
  throw ex;
}
```

Розглянемо ще один випадок.

Передбачимо, що оператор throw застосовується усередині конструкції try-catch.

```
try {
  ...
  throw new IOException();
  ...
} catch(Exception e){
  ...
}
```

В цьому випадку виключення, збуджене в блоці try, не буде передано для обробки на вищий рівень ієрархії, а обробиться в межах блоку try-catch, оскільки тут міститься оператор, який може це виключення перехопити. Тобто станеться неявна передача управління на відповідний блок catch.

Виключення, що перевіряються і не перевіряють

Всі виняткові ситуації можна розділити на дві категорії: що перевіряються (checked) і не перевіряють(unchecked).

Всі виключення, що породжуються від Throwable, можна розбити на три групи. Вони визначаються трьома базовими типами: спадкоємцями Throwable - класами Error і Exception, а також спадкоємцем Exception - RuntimeException.

Помилки, породжені від Exception (і що не є спадкоємцями RuntimeException), є такими, що перевіряються. Тобто під час компіляції перевіряється, чи передбачена обробка можливих виняткових ситуацій. Як правило, це помилки, пов'язані з оточенням програми (мережевим, файловим введенням-виводом і ін.), які можуть виникнути незалежно від того, коректно написаний код чи ні. Наприклад, відкриття мережевого з'єднання або файлу може привести до виникнення помилки і компілятор вимагає від програміста передбачити деякі дії для обробки можливих проблем. Таким чином підвищується надійність програми, її стійкість при можливих збоях.

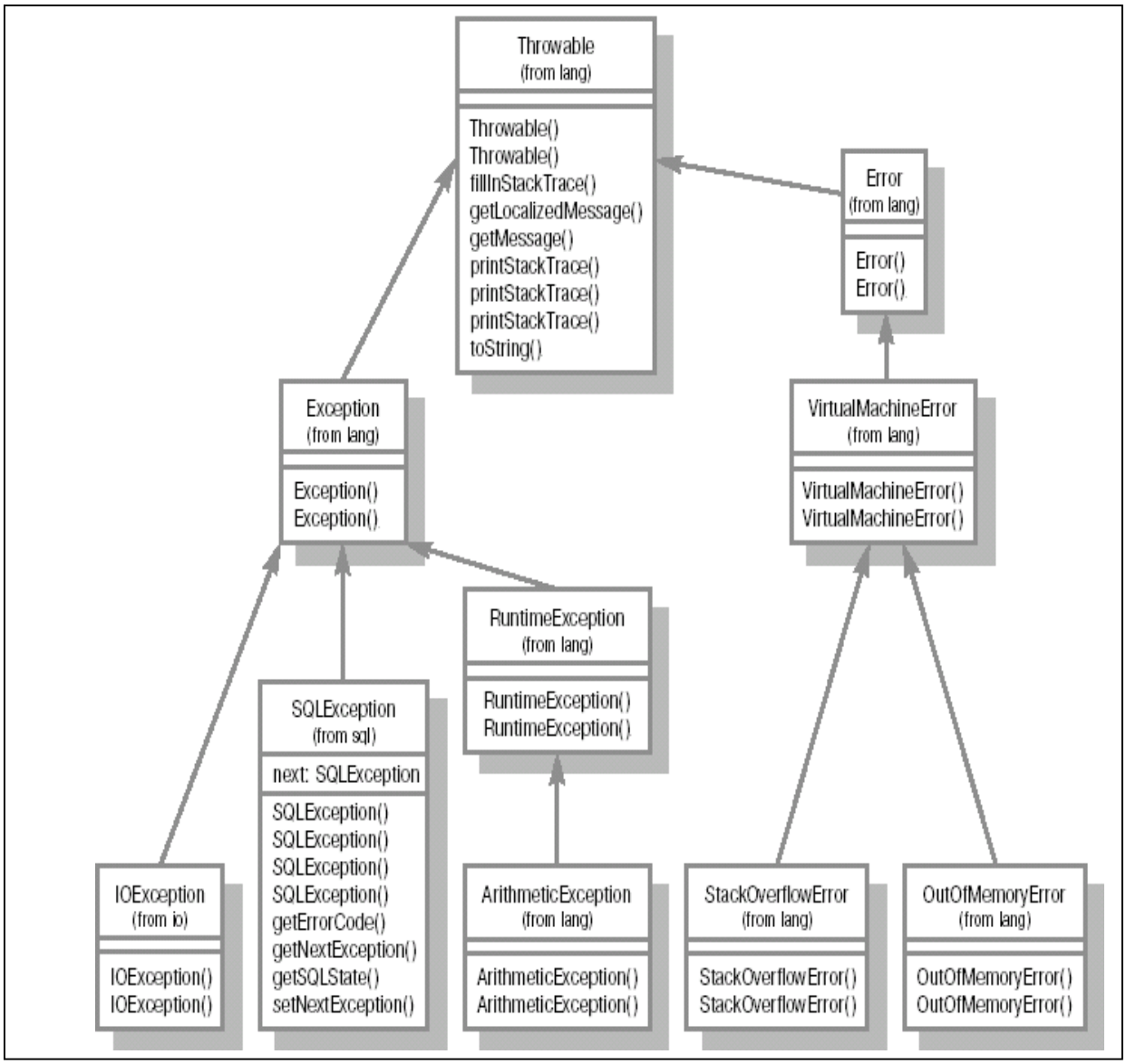
Виключення, породжені від RuntimeException, є такими, що не перевіряють і компілятор не вимагає обов'язкової їх обробки.

Як правило, це помилки програми, які при правильному кодуванні виникати не повинні (например,IndexOutOfBoundsException - вихід за кордони масиву, java.lang.ArithmeticException - ділення на нуль). Тому, аби не захаращувати програму, компілятор залишає на розсуд програміста обробку таких виключень за допомогою блоків try-catch.

Виключення, породжені від Error, також не є такими, що перевіряються. Вони призначені для того, щоб повідомити додаток про виникнення фатальної ситуації, яку програмним способом усунути практично неможливо (хоча формально обробник допускається). Вони можуть свідчити про помилки програми, але, як правило, це неусувні проблеми на рівні JVM. Як приклад можна привестиStackOverflowError (переповнювання стека), OutOfMemoryError (брак пам'яті).

Якщо в конструкції обробки виключень використовується декілька операторів catch, класи виключень потрібно перераховувати в них послідовно, від менш загальних до загальнішим. Розглянемо два приклади:

```
 try {
   ...
}
catch(Exception e){
   ...
}
catch(IOException ioe){
   ...
}
Catch(UserException ue){
   ...
}
```



Мал. 10.1. Ієрархія класів стандартних виключень.

У даному прикладі при виникненні виняткової ситуації (клас, породжений від Exception) виконуватиметься завжди лише перший блок catch. Останні не будуть виконані ні за яких умов. Ця ситуація відстежується компілятором, який повідомляє про UnreachableCodeException (помилка - недосяжний код). Правильно дана конструкція виглядатиме так:

```
 try {
  ...
}
catch(UserException ue){
  ...
}
catch(IOException ioe){
  ...
```

```
}
catch(Exception e){
  ...
}
```

В цьому випадку виконуватиметься послідовна обробка виключень. І в разі, якщо не передбачена обробка того типа виключення, яке виникло (наприклад, AnotherUserException), буде виконаний блокcatch(Exception e){:}

Якщо спрацьовує один з блоків catch, то останні блоки в даній конструкції try-catch виконуватися не будуть.

Створення призначених для користувача класів виключень

Як вже наголошувалося, допускається створення власних класів виключень. Для цього досить створити свій клас, успадкувавши його від будь-якого спадкоємця java.lang.Throwable (або від самого Throwable).

Приклад:

```
public class UserException extends Exception {
  public UserException() {
    super();
  }
  public UserException(String descry){
    super(descry);
  }
}
```

Відповідно, дане виключення створюватиметься таким чином:

```
 throw new UserException(
  "Додатковий опис");
```

Перевизначення методів і виключення

При перевизначенні методів слід пам’ятати, що якщо перевизначуваний метод оголошує список можливих виключень, то переопределяющий метод не може розширювати цей список, але може його звужувати. Розглянемо приклад:

```
public class BaseClass{
  public void method () throws IOException {
  ...
  }
}

public class LegalOne extends BaseClass {
  public void method () throws IOException {
    ...
  }
```

```
}

public class LegalTwo extends BaseClass {
  public void method () {
    ...
  }
}

public class LegalTree extends BaseClass {
  public void method ()
    throws
     EOFException,MalformedURLException {
    ...
  }
}

public class IllegalOne extends BaseClass {
  public void method ()
    throws
     IOException,IllegalAccessException {
    ...
  }
}

public class IllegalTwo extends BaseClass {
  public void method () {
    ...
    throw new Exception();
  }
}
```

В даному випадку:

- визначення класу LegalOne буде коректним, оскільки перевизначення методу method() вірне (список помилок не змінився);
- визначення класу LegalTwo буде коректним, оскільки перевизначення методу method() вірне (новий метод не може викидати помилок, а значить, не розширює список можливих помилок старого методу);
- визначення класу LegalTree буде коректним, оскільки перевизначення методу method() буде вірним (новий метод може створювати виключення, які є підкласами виключення, що збуджується в старому методі, тобто список звузився);
- визначення класу IlegalOne буде некоректним, оскільки перевизначення методу method() невірно (IllegalAccessException не є підкласом IOException, список розширився);
- визначення класу IlegalTwo буде некоректним: хоча заголовок method() оголошений вірно (список не розширився), в телі методу кидається виключення, не вказане в throws.

Особливі випадки

Під час виконання коди можуть виникати ситуації, які майже не описані в літературі.

Розглянемо таку ситуацію:

```
import java.io.*;
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    try {
      test.doFileInput("bogus.file");
    }
    catch (IOException ex) {
      System.out.println("Second exception handle stack trace");
      ex.printStackTrace();
    }
  }

  private String doFileInput(String fileName)
      throws FileNotFoundException,IOException {
    String retStr = "";
    java.io.FileInputStream fis = null;
    try {
      fis = new java.io.FileInputStream(fileName);
    }
    catch (FileNotFoundException ex) {
      System.out.println("First exception handle stack trace");
      ex.printStackTrace();
      throw ex;
    }
    return retStr;
  }
}
```

Результат роботи виглядатиме таким чином:

```
 java.io.FileNotFoundException: bogus.file (The system cannot find
    the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:64)
  at experiment.Test.doFileInput(Test.java:33)
  at experiment.Test.main(Test.java:21)
First exception handle stack trace
java.io.FileNotFoundException: bogus.file (The system cannot find
```

```
    the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:64)
  at experiment.Test.doFileInput(Test.java:33)
  at experiment.Test.main(Test.java:21)
Second exception handle stack trace
```

Оскільки при вторинному збудженні використовується один і той же об'єкт Exception, стік в обох випадках міститиме одну і ту ж послідовність викликів. Тобто при повторному збудженні виключення, якщо ми використовуємо той же об'єкт, зміни його параметрів не відбувається.

Розглянемо інший приклад:

```
import java.io.*;

public class Test {

  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    try {
      test.doFileInput();
    }
    catch (IOException ex) {
      System.out.println("Exception hash code " + ex.hashCode());
      ex.printStackTrace();
    }
  }

  private String doFileInput()
      throws FileNotFoundException,IOException{
    String retStr = "";
    java.io.FileInputStream fis = null;
    try {
      fis = new java.io.FileInputStream("bogus.file");
    }
    catch (FileNotFoundException ex) {
      System.out.println("Exception hash code " + ex.hashCode());
      ex.printStackTrace();
      fis = new java.io.FileInputStream("anotherBogus.file");
      throw ex;
    }
    return retStr;
  }
}

java.io.FileNotFoundException: bogus.file (The system cannot find
    the file specified)
```

```
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:64)
    at experiment.Test.doFileInput(Test.java:33)
    at experiment.Test.main(Test.java:21)
Exception hash code 3214658

java.io.FileNotFoundException:  (The system cannot find the path
      specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:64)
    at experiment.Test.doFileInput(Test.java:38)
    at experiment.Test.main(Test.java:21)
Exception hash code 6129586
```

Нескладно відмітити, що, хоча послідовність викликів одна і та ж, в методах, що викликаються і викликають, обробляються різні об'єкти виключень.

Висновок

У даній лекції розглянуті основні мовні конструкції.

Для організації циклів в Java призначені три основних конструкции:while, do, for. Для зміни порядку виконання операторів застосовуються continue і break (з міткою або без). Також існують два оператори галуження: if і switch.

Важливою темою є обробка помилок, оскільки без неї не обходиться жодна програма, адже причиною збоїв може служити не лише помилка програміста, але і зовнішні події, наприклад, розрив мережевого з'єднання. Основною конструкцією обробки виняткових ситуацій є try-catch-finally. Для явної ініціалізації виняткової ситуації служить ключове слово throw.

Помилки діляться на тих, що перевіряються і не перевіряють. Аби підвищити надійність програми, компілятор вимагає обробки виключень, класи яких успадковуються від Exception, окрім классов-наследниковRuntimeException. Передбачається, що такі помилки можуть виникати не стільки помилково розробника, скільки по зовнішніх неконтрольованих причинах.

Класи, успадковані від RuntimeException, описують програмні збої. Очікується, що програміст зведе вірогідність таких помилок до мінімуму, а тому, аби не захаращувати код, вони являютсянепроверяемыми, компілятор залишає обробку на розсуд розробника. Помилки-спадкоємці Error свідчать про фатальні збої, тому їх також необов'язково обробляти.

Методи, код яких може породжувати виключення, що перевіряються, повинні або самі їх обробляти, або в заголовку методу має бути вказане ключове слово throws з перерахуванням необроблюваних виключень, що перевіряються. На помилки, що не перевіряють , це правило не поширюється.

Перевизначений (overridden) метод не може розширювати список можливих виключень вихідного методу.

ТЕМА. *Пакет java..awt.*

План

1. *Базові бібліотеки Java.*
2. *Віконний інтерфейс користувача (GUI).*

Рекомендована література: 1, 2, 3, 4.

Теоретичні відомості

Оскільки Java-додатки призначені для роботи на різноманітних платформах, реалізація графічного призначеного для користувача інтерфейсу (GUI) має бути або однаковою для будь-якої платформи, або, навпаки, програма повинна мати вигляд, типовий для даної операційної системи. Через низку обставин, для основної бібліотеки по створенню GUI був вибраний другий підхід. По-перше, це зайвий раз показувало гнучкість Java – дійсно, користувачі різних платформ могли працювати з одним і тим же Java- додатком, не міняючи своїх звичок. По-друге, така реалізація забезпечувала велику продуктивність, оскільки була заснована на можливостях операційної системи. Зокрема, це означало і компактніший, простій, а значить, і надійніший код.

Бібліотеку назвали AWT – Abstract Window Toolkit. Слово abstract в назві указує, що всі стандартні компоненти не є самостійними, а працюють в зв'язці з відповідними елементами операційної системи.

Дерево компонентів

Component

Абстрактний клас Component є базовим для всіх компонент AWT і описує їх основні властивості. Візуальний компонент в AWT має прямокутну форму, може бути відображений на екрані і може взаємодіяти з користувачем.

Розглянемо основні властивості цього класу.

Положення

Положення компоненту описується двома цілими числами (тип int) x і y. У Java (як і в багатьох мовах програмування) вісь x проходить традиційно – горизонтально, направлена управо, а вісь y – вертикально, але направлена вниз, а не вгору, як прийнято в математиці. Для опису положення компоненту призначений спеціальний клас – Point (крапка). У цьому класі визначено два public int поля x і y, а також безліч конструкторів і допоміжних методів для роботи з ними. Клас point застосовується в багатьох типах AWT, де треба задати крапку на плщині. Для компоненту ця крапка задає положення лівого верхнього кута. Встановити положення компоненту можна за допомогою методу setLocation(), який може приймати як аргументи пару цілих чисел, або Point. Дізнатися поточне положення можна за допомогою методу getLocation(),

повертаючого Point, або за допомогою методів getX() і getY(), які з'явилися з версії Java 1.2.

Розмір

Як було сказано, компонент AWT має прямокутну форму, а тому його розмір описується також двома цілочисельними параметрами – width (ширина) і height (висота). Для опису розміру існує спеціальний клас Dimension (розмір), в якому визначено два public int поля width і height, а також допоміжні методи. Встановити розмір компоненту можна за допомогою методу setSize, який може приймати як аргументи пару цілих чисел, або Dimension. Дізнатися поточні розміри можна за допомогою методу getSize(), повертаючого Dimension, або за допомогою методів getWidth() і getHeight(), які з'явилися з версії Java 1.2.

Сумісне положення і розмір компоненту задають його межі. Область, займану компонентом, можна описати або чотирма числами (x, y, width, height), або екземплярами класів Point і Dimension, або спеціальним класом Rectangle (прямокутник). Як легко здогадатися, в цьому класі визначено чотири public int поля, з якими можна працювати і у вигляді пари об'єктів Point і Dimension. Задати межу об'єкту можна за допомогою методу setBounds, який може приймати чотири числа, або Rectangle. Дізнатися поточне значення можна за допомогою методу getBounds(), повертаючого Rectangle.

Видимість

Існуючий компонент може бути як видний користувачеві, так і бути прихованим. Ця властивість описується булевим параметром visible. Методи для управління – setVisible, що приймає булевий параметр, иisVisible, що повертає поточне значення. Зрозуміло, невидимий компонент не може взаємодіяти з користувачем.

Доступність

Навіть якщо компонент відображається на екрані і видний користувачеві, він може не взаємодіяти з ним. В результаті події від клавіатури, або миші не виходитимуть і не оброблятимуться компонентом. Такий компонент називається disabled. Якщо ж компонент активний, його називають enabled. Як правило, компонент деяким чином міняє свій зовнішній вигляд, коли стає недоступним (наприклад, стає сірим, менш помітним), але, взагалі кажучи, це необов'язково (хоча дуже зручно для користувача).

Для зміни цієї властивості застосовується метод setEnabled, що приймає булевий параметр (true відповідає enabled, false – disabled), а для набуття поточного значення – isEnabled.

Кольори

Зрозуміло, для побудови сучасного графічного інтерфейсу користувача необхідна робота з квітами. Компонент володіє двома властивостями, що описують кольори, – foreground і background кольори. Перша властивість задає, яким кольором виводити написи, малювати лінії і так далі Друге – задає колір фону, яким закрашується вся область, займана компонентом, перед тим, як промальовувався зовнішній вигляд. Для завдання

кольору в AWT використовується спеціальний клас Color. Цей клас володіє досить обширною функціональністю, тому розглянемо основні характеристики.

Колір задається 3 цілочисельними характеристиками, відповідними моделі RGB, – червоний, синій, зелений. Кожна з них може мати значення від 0 255 (проте, їх тип визначений як int). В результаті (0, 0, 0) відповідає чорному, а (255, 255, 255) – білому.

Клас Color є незмінним, тобто, створивши екземпляр, відповідний якому-небудь кольору, змінити параметри RGB вже неможливо. Це дозволяє оголосити в класі Color ряд констант, що описують базові кольори: білий, чорний, червоний, жовтий і так далі. Наприклад, замість того, щоб задавати синій колір числовими параметрами (0, 0, 255), можна скористатися константами Color.blue або Color.BLUE (другий варіант з'явився в пізніших версіях).

Для роботи з властивістю компоненту foreground застосовують методи setForeground і getForeground, а дляbackground – setBackground і getBackground.

Шрифт

Раз зображення компоненту може включати написи, необхідна властивість, що описує шрифт для їх промальовування. Для завдання шрифту в AWT існує спеціальний клас Font, який включає три параметри – ім'я шрифту, розмір і стиль.

Ім'я шрифту задає зовнішній стиль відображення символів. Імена зазнали ряд змін з розвитком Java. У версії 1.0 потрібне, щоб JVM підтримувала наступні шрифти: TimesRoman, Helvetica, Courier. Можуть підтримуватися і інші сімейства, це залежить від деталей реалізації конкретної віртуальної машини. Щоб дізнатися повний список під час виконання програми, можна скористатися методом утилитного классаToolkit. Екземпляри цього класу не можна створити уручну, тому повністю такий запит виглядатиме таким чином:

```
Toolkit.getDefaultToolkit().getFontList()
```

В результаті буде повернений масив рядків-імен сімейств підтримуваних шрифтів.

У Java 1.1 три обов'язкові імена були оголошені deprecated. Замість них був введений новий список, який містив більш універсальні назви, не залежні від конкретної операційної системи: Serif, SansSerif,Monospaced. У Java 2 бібліотека AWT була істотно переглянута і доповнена. Щоб усунути неоднозначності з різною підтримкою шрифтів на різних платформах, було проведено розділення на логічні і фізичні шрифти. Друга група визначається можливостями операційної системи, це ті ж шрифти, які можуть використовувати інші програми, запущені на цій платформі. Перша група складається з 5 обов'язкових сімейств (додалися Dialog і DialogInput). JVM встановлює відповідність між ними і найбільш відповідними з доступних фізичних шрифтів. Метод getFontList класу Toolkit був оголошений deprecated. Тепер отримати список всіх доступних фізичних шрифтів можна таким чином:

```
GraphicsEnvironment.
  getLocalGraphicsEnvironment().
  getAvailableFontFamilyNames()
```

Клас Font є незмінним. Після створення можна дізнатися задане логічне ім'я (метод getName) і відповідне йому фізичне ім'я сімейства (метод getFamily).

Повернемося до останніх параметрів, необхідних для створення екземпляра Font. Розмір шрифту визначає, очевидно, величину символів. Проте конкретні значення вимірюються не в пикселах, а в умовних одиницях (як і в багатьох текстових редакторах). Для різних сімейств шрифтів символи однакового розміру можуть мати різну ширину і висоту, зміряну в пикселах. Як і у разі імені шрифту, програміст може вказати будь-яке значення розміру, а JVM поставить йому у відповідність максимально близький з доступних. Нарешті, останній параметр – стиль. Цей параметр визначає, чи буде шрифт жирним, похилим і так далі Якщо ніякі з цих властивостей не потрібні, указується Font.PLAIN (параметр має тип int і в класі Font визначений набір констант для зручності роботи з ним). Значення Font.BOLD задає жирний шрифт, а Font.ITALIC – похилий. Для поєднання цих властивостей (жирний похилий шрифт) необхідно провести логічний склад: Font.BOLD|Font.ITALIC.

Для роботи з цією властивістю класу Component призначені методи setFont і getFont. Отже, ми розглянули основні властивості класу Component. Як легко бачити, всі вони призначені для опису графічного представлення компоненту, тобто відображення на екрані.

Існує ще одна важлива властивість іншого характеру. Очевидно, що практично завжди призначений для користувача інтерфейс полягає із понад одного компоненту. У великих застосуваннях їх зазвичай значно більше. Для зручності організації роботи з ними компоненти об'єднуються в контейнери. У AWT існує клас, який так і називається, – Container. Його розгляд – наша наступна тема. Важливо відзначити, що компонент може знаходитися лише в одному контейнері – при спробі додати його в інший він віддаляється з першого. Дана властивість якраз і відповідає за зв'язок компоненту з контейнером. Властивість називається parent. Завдяки ньому компонент завжди "знає", в якому контейнері він знаходиться.

Container

Контейнер описується класом Container, який є наслідником Component, а значить, володіє всіма властивостями графічного компоненту. Проте основне його завдання – групувати інші компоненти. Для цього в нім оголошений цілий ряд методів. Для додавання служить метод add, для видалення – remove і removeAll (останній видаляє всі компоненти). Компоненти, що додаються, зберігаються у впорядкованому списку, тому для видалення можна вказати або посилання на компонент, який і буде видалений, або його порядковий номер в контейнері. Також визначені методи для отримання компонент, присутніх в контейнері, – всі вони досить очевидні, тому перерахуємо їх з короткими поясненнями:

- getComponent(int n) – повертає компонент з вказаним порядковим номером;
- getComponents() – повертає всі компоненти у вигляді масиву;
- getComponentCount() – повертає кількість компонент;
- getComponentAt(int x, int y) або (Point p) – повертає компонент, який включає вказану крапку;

- findComponentAt(int x, int y) або (Point p) – повертає видимий компонент, що включає вказану крапку.

Ми вже знаємо, що положення компоненту (location) задається координатами лівого верхнього кута. Важливо, що ці значення відлічуються від лівого верхнього кута контейнера, який таким чином є центром системи координат для кожного компоненту, що знаходиться в нім. Якщо важливе розташування компоненту на екрані безвідносно його контейнера, можна скористатися методом getLocationOnScreen.

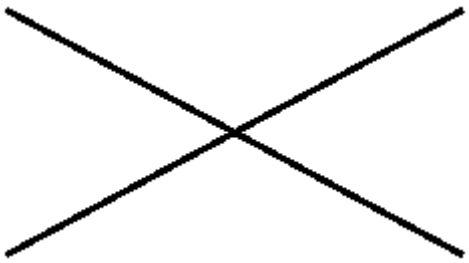
Завдяки спадкоємству контейнер також має властивість size. Цей розмір задається незалежно від розміру і положення вкладених компонент. Таким чином, компоненти можуть розташовуватися частково або повністю за межами свого контейнера (що це означає, буде розглянуто нижче, але принципово це допустимо). Раз контейнер успадковується від Component, він сам є компонентом, а значить, може бути доданий в інший, вищестоящий контейнер. В той же час компонент може знаходитися лише в одному контейнері. Це означає, що всі елементи складного призначеного для користувача інтерфейсу об'єднуються в ієрархічне дерево. Така організація не тільки полегшує операції над ними, але і задає основні властивості всієї роботи AWT. Одним з них є принцип отрисовки компонентів.

Алгоритм перемальовки

Почнемо з перемальовки окремого компоненту – що визначає його зовнішній вигляд? Для цього завдання призначений метод paint. Цей метод викликається кожного разу, коли необхідно відобразити компонент на екрані. У нього є один аргумент, тип якого – абстрактний клас Graphics. У цьому класі визначена безліч методів для перемальовки простих графічних елементів – ліній, прямокутників і багатокутників, кіл і овалів, тексту, картинок і так далі Спадкоємці класу Component перевизначать метод paint і, користуючись методами Graphics, задають алгоритм промальовування свого зовнішнього вигляду:

```
public void paint(Graphics g){
  g.drawLine(0, 0, getWidth(), getHeight());
  g.drawLine(0, getHeight(), getWidth(), 0);
}
```

В даному прикладі компонент відображатиметься двома лініями, що проходять по його діагоналях:



Методи класу Graphics для відмальовки

Розглянемо оглядові методи класу Graphics, призначені для перемальовки.

drawLine(x1, y1, x2, y2)

Цей метод відображає лінію завтовшки в 1 пиксел, що проходить з крапки (x1, y1) в (x2, y2). Саме він використовувався в попередньому прикладі.

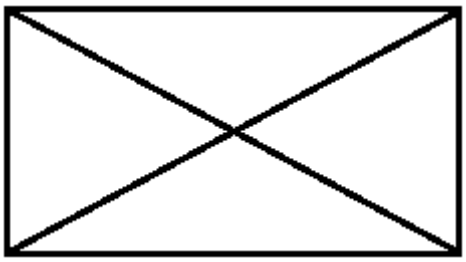
drawRect(int x, int y, int width, int height)

Цей метод відображає прямокутник, чий лівий верхній кут знаходиться в крапці (x, y), а ширина і висота дорівнюють width і height відповідно. Права сторона пройде по лінії x+width, а нижняя – y+height.

Припустимо, ми хочемо доповнити попередній приклад малюванням рамки навколо компоненту (периметр). Зрозуміло, що лівий верхній кут знаходиться в крапці (0, 0). Якщо ширина компоненту рівна, наприклад, 100 пікселам, то координата x пробігає значення від 0 до 99. Це означає, що ширина і висота мальованого прямокутника мають бути зменшені на одиницю. Насправді з тієї ж причини в попередньому прикладі таке зменшення на одиницю має бути присутнім і в останніх методах:

```
public void paint(Graphics g){
  g.drawLine(0,0,getWidth()-1, getHeight()-1);
  g.drawLine(0,getHeight()-1, getWidth()-1,0);
  g.drawRect(0,0,getWidth()-1, getHeight()-1);
}
```

В результаті компонент прийме наступний вигляд:



fillRect(int x, int y, int width, int height)

Цей метод закрашує прямокутник. Ліва і права сторони прямокутника проходять по лініях x і x+width-1соответственно, а верхня і нижня – y і y+height-1 відповідно. Так, щоб замалювати всі пікселі компоненту, необхідно передати наступні аргументи:

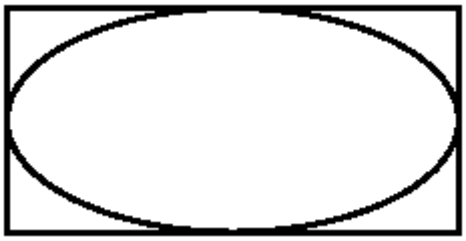
```
g.fillRect(0, 0, getWidth(), getHeight());
```

drawOval(int x, int y, int width, int height)

Цей метод малює овал, вписаний в прямокутник, що задається вказаними параметрами. Очевидно, що якщо прямокутник має рівні сторони (тобто є квадратом), овал стає колом. Знову для того, щоб вписати овал в межі компоненту, необхідно відняти по одиниці з ширини і висоти:

```
g.drawRect(0, 0, getWidth()-1, getHeight()-1);
g.drawOval(0, 0, getWidth()-1, getHeight()-1);
```

Результат:



fillOval(int x, int y, int width, int height)

Цей метод закрашує вказаний овал.

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Цей метод малює дугу – частину овалу, що задається першими чотирма параметрами. Дуга починається з кута startAngle і має кутовий розмір arcAngle. Початковий кут відповідає напряму годинникової стрілки, вказуючої на 3 години. Кутовий розмір відлічується проти годинникової стрілки. Таким чином, розмір в 90 градусів відповідає дузі в чверть овалу (верхню праву). Кути "розтягнуті" відповідно до розміру прямокутника. В результаті, наприклад, кутовий розмір в 45 градусів завжди задає межу дуги по лінії, що проходить з центру прямокутника в його правий верхній кут.

fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

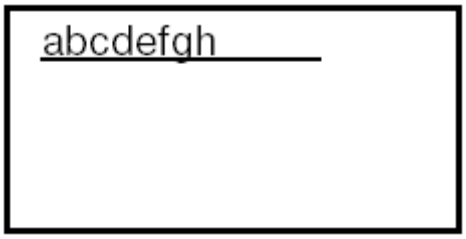
Цей метод закрашує сектор, обмежений дугою, що задається параметрами.

drawString(String text, int x, int y)

Цей метод виводить на екран текст, що задається першим параметром. Крапка (x, y) задає позицію найлівішого символу. Для наочності приведемо приклад:

```
g.drawString("abcdefgh", 15, 15);
g.drawLine(15, 15, 115, 15);
```

Результатом буде:



Стан Graphics

Екземпляр класу Graphics зберігає параметри, необхідні для відмальовки. Розглянемо їх по порядку.

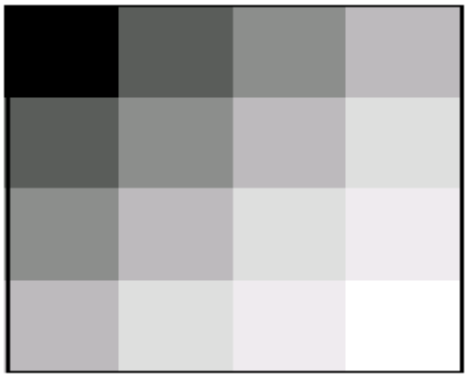
Колір

Очевидно, що для перемальовки ліній, овалів, тексту і так далі необхідно використовувати той або інший колір. За умовчанням він задається властивістю foreground компоненту. У будь-який момент його можна змінити за допомогою методу setColor.

Розглянемо приклад:

```
public void paint(Graphics g){
  for (int i=0; i<4; i++) {
    for (int j=0; j<4; j++) {
      int з = (int)((i+j)*255/6);
      g.setColor(new Color(з, з, з));
      g.fillRect(i*getWidth()/4, j*getHeight()/4, getWidth()/4, getHeight()/4);
    }
  }
}
```

В результаті компонент матиме наступний вигляд:



Дізнатися поточне значення кольору для перемальовки можна за допомогою методу getColor.

Шрифт

Метод drawString не має аргументу, задаючого шрифт для виведення тексту на екран. Цей параметр також є частиною стану Graphics. Його значення за умовчанням задається відповідною властивістю компоненту, проте може бути змінено за допомогою методу setFont. Для набуття поточного значення служить метод getFont.

Clip (обмежувач)

Хоча методи класу Graphics можуть набувати будь-яких значень аргументів, задаючих значення координат (в межах типу int), існує додатковий обмежувач – clip. Будь-які зміни поза цим обмежувачем на екрані з'являтися не будуть. Наприклад, якщо викликати метод drawLine(-100, -100, 1000, 1000), то на компоненті відобразиться лише частина лінії, яка поміщається в його межі.

Розміри обмежувача можна змінювати. Метод clipRect(int x, int y, int width, int height) обчислює перетин вказаного прямокутника і поточної області clip. Результат стане новим обмежувачем. Таким чином, цей метод може тільки звужувати область clip. Метод setClip(int x, int y, int width, int height) встановлює обмежувач довільно у формі прямокутника. Метод getClipBounds повертає поточне значення у вигляді об'єкту Rectangle. При появі додатку на екрані кожен видимий компонент має відмалювати повністю. Тому при першому виклику методу paint, як правило, область clip збігається з межами компоненту. Проте при подальшій роботі це не завжди так.

Розглянемо наступний приклад:

```
public void paint(Graphics g){
  Color з = new Color(
    (int)(Math.random()*255),
   (int)(Math.random()*255),
   (int)(Math.random()*255));
  g.setColor(c);
  //g.setClip(null);
  g.fillRect(0, 0, getWidth(), getHeight());
}
```

Як видно з коду, при кожному виклику методу paint генерується нове значення кольору, після чого цим кольором закрашується весь компонент. Проте оскільки в Graphics є обмежувач, закрашена буде тільки область clip, що дозволить її побачити. Після запуску програми компонент буде повністю забарвлений одним кольором. Якщо тепер за допомогою миші "узяти" вікно якого-небудь іншого застосування і поволі "провести" їм поверх компоненту, то він забарвиться приблизно таким чином (ліва картинка):

Якщо ж провести швидко, то вийде картинка, подібна правою в прикладі вище. Добре видно, що компонент перемальовувався не повністю, а частинами. Обмежувач виставляється відповідно до тієї області, яка опинилася "пошкоджена" і потребує перемальовування. Для складних компонентів можна ввести логіку, яка, використовуючи clip, буде перемальовувати не всі елементи, а тільки деякі з них, для збільшення продуктивності. У прикладі закоментований один рядок, в якому передається значення null в метод setClip. Такий виклик знімає всі обмеження, тому компоненту кожного разу перефарбовуватиметься повністю, міняючи при цьому колір. Проте ніяким чином не можна змінити стан пікселів поза компонентом – обмежувач не може бути ширше, ніж межі компоненту.

Методи repaint і update

Окрім paint в класі Component оголошено ще два методи, що відповідають за перемальовування компоненту. Як було розглянуто, виклик paint ініціюється операційною системою, якщо виникає необхідність перемальовувати вікно додатку, або частину його. Проте може потрібно відновити зовнішній вигляд, керуючись програмною логікою. Наприклад, відобразити результат операції обчислення, або роботи з мережею. Можна змінити стан компоненту (значення його полів), але операційна система не відстежить таку зміну і не ініціює перемальовування.

Для програмної ініціалізації перемальовування компоненту служить метод repaint. Звичайно, у нього немає аргументу типу Graphics, оскільки програміст не повинен створювати екземпляри цього класу (точніше, його спадкоємців, адже Graphics – абстрактний клас). Метод repaint можна викликати без аргументів. В цьому випадку компонент перемальовує максимально швидко. Можна вказати аргумент типу long – кількість мілісекунд. Система ініціалізувала перемальовування через вказаний час. Можна вказати чотири числа типу int (x, y, width, height), задаючи прямокутну область компоненту, яка потребує перемальовування. Нарешті, можна вказати все 5 параметрів – і затримку за часом, і область перемальовування.

Якщо перемальовування ініціюється додатком, то система викликає не метод paint, а метод update. У нього вже є аргумент типу Graphics і за умовчанням він лише закрашує всю область компоненту фоновим кольором (властивість background), а потім викликає метод paint. Навіщо ж було вводити цей додатковий метод, якщо можна було відразу викликати paint? Річ у тому, що оскільки перемальовування ініціюється додатком, для складних компонентів стає можливою деяка оптимізація оновлення зовнішнього вигляду. Наприклад, якщо зміна полягає в появі нового графічного елементу, то можна уникнути повторного перемальовування останніх елементів – перевизначити метод update і реалізувати в нім відображення одного тільки нового елементу. Якщо ж компонент має просту структуру, можна залишити метод update без змін.

Промальовування контейнера

Тепер, коли відомо, як працює промальовування компоненту, перейдемо до розгляду контейнера. Для його коректного відображення необхідно виконати дві дії. По-перше, намалювати сам контейнер, адже він є спадкоємцем компоненти, а значить, має метод paint, який може бути перевизначений для завдання особливого зовнішнього вигляду такого контейнера. По-друге, ініціювати перемальовки всіх компонентів, вкладених в нього.

Перший крок нічим не відрізняється від промальовування звичайного компоненту. Як правило, контейнер не містить ніяких особливих елементів відображення, адже основну його площу займають вкладені компоненти. Тому перейдемо до другого кроку.

Якщо контейнер не порожній, значить, в нім є одна або декілька компонент. Вони відмалюють послідовно в тому порядку, в якому були додані. Проте недостатньо просто в циклі викликати метод paint для кожного компоненту.

По-перше, якщо компоненту невидима (властивість visible виставлена в false), то, очевидно, метод paint у неї викликатися не повинен.

По-друге, центр координат компоненту знаходиться в лівому верхньому кутку його контейнера, а у контейнера – в лівому верхньому кутку його контейнера. Таким чином, при переході від отрисовки контейнера до отрисовке лежачого в нім компоненту необхідно змінити (перенести) центр системи координат. Потім необхідно встановити clip відповідно до розміру чергового компоненту. Необхідно виставити значення за умовчанням для кольору і шрифту, тим більше що попередній компонент міг змінити їх непередбачуваним чином.

У результаті виходить зручнішим створити новий екземпляр Graphics для кожного компоненту. Для цього існує метод create, який породжує копію Graphics, причому йому можна передати аргументи (int x, int y, int width, int height). В результаті у нового Graphics буде зміщений центр координат в крапку (x, y), aclip-область буде отримана перетином існуючого обмежувача з прямокутником (0, 0, width, height) (у нових координатах). Метод create створює копію без зміни цих параметрів.

Такі копії буває зручно породжувати і в рамках одного виклику методу paint, якщо в нім описаний дуже складний алгоритм. Після використання такого об'єкту Graphics його необхідно особливим чином звільняти – викликом методу dispose(). Якщо необхідно тільки змістити точку відліку координат, можна використовувати методtranslate (int x, int y). Таким чином, контейнер своїм методом paint отрисовывает себе і всі вкладені в нього компоненти. Якщо якісь з них, у свою чергу, є контейнерами, то процес ієрархічно продовжується углиб. У результаті весь AWT інтерфейс, яким би складним він не був, складається з дерева контейнерів і компонент, отрисовка яких починається з самого верхнього контейнера і по гілках розвивається углиб до кожного видимого компоненту.

Окремий інтерес представляє цей самий верхній контейнер. Як правило, це вікно операційної системи, що одночасно є контейнером для Java-компонент. Саме операційна система ініціалізувала процес отрисовки, відповідає за згортання і розгортання вікна,

зміна його розміру і так далі. З боку Java для роботи з вікном використовується клас Window, який є спадкоємцем Container і розглядається нижчим.

Спадкоємці класу Component

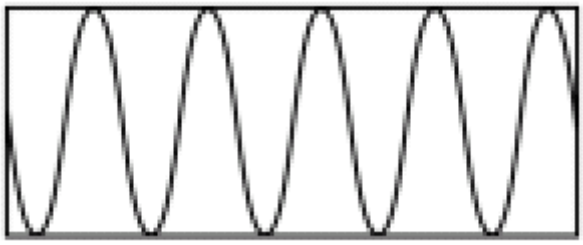
Тепер, коли розглянуті основні принципи роботи класів Component иContainer, розглянемо їх спадкоємців, за допомогою яких і будується функціональний призначений для користувача інтерфейс. Почнемо із спадкоємців класу Component.

Клас Canvas

Клас Canvas є простим спадкоємцем Component. Він не додає ніякої нової функціональності, але саме його потрібно використовувати як суперклас для створення призначеного для користувача компоненту з деяким нестандартним зовнішнім виглядом. Нижче приведений приклад визначення компоненту, який відображає графік функції sin(x):

```
public class SinCanvas extends Canvas {

  public void paint(Graphics g){
    int height = getHeight(),
        width = getWidth();
   // Обчислюємо масштаб таким чином
  // щоб на компоненті завжди уміщалося
  // 5 періодів
   double k=2*Math.PI*5/width;
   int sy = calcY(0, width, height, до);
   for (int i=1; i<width; i++) {
     int nsy = calcY(i, width, height, до);
     g.drawLine(i-1, sy, i, nsy);
     sy=nsy;
   }
  }

  // метод, що обчислює значення функції
  // для відображення на екрані
  private int calcY(int x, int width
                    int height, double до) {
    double dx = (x-width/2.)*k;
    return (int)(height/2.*(1-Math.sin(dx)));
  }
}
```

Як видно з прикладу, достатньо лише перевизначити метод paint. От як виглядає такий компонент:



Клас Label

Як зрозуміло з назви, цей компонент відображає напис. Відповідно, і його основний конструктор приймає один аргумент типу String – текст напису. За допомогою стандартних властивостей класу Component – шрифт, колір, фоновий колір – можна міняти вид напису. Текст можна змінити і після створення Label за допомогою методу setText.

Звернете увагу, що при цьому компонент сам оновлює свій вигляд на екрані. Такою особливістю володіють всі стандартні компоненти AWT.

Клас Button

Цей компонент дозволяє додати в інтерфейс стандартні кнопки. Основний конструктор приймає як аргумент String – напис на кнопці. Як обробляти натиснення на кнопку і інші призначені для користувача події, розглядається нижчим.

Класи Checkbox і CheckboxGroup

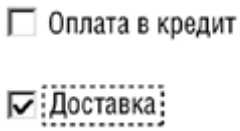
Компонент Checkbox має два способи застосування.

Коли він використовується сам по собі, він представляє checkbox – елемент, який може бути виділений чи ні (наприклад, потрібна доставка для покупки, що оформляється, чи ні). В цьому випадку в конструктор передається лише текст – підпис до checkbox.

Розглянемо приклад, в якому в тілі контейнера додається два checkbox:

```
Checkbox payment = new
        Checkbox("Оплата в кредит");
payment.setBounds(10, 10, 120, 20);
add(payment);
Checkbox delivery = new Checkbox("Доставка");
delivery.setBounds(10, 30, 120, 20);
add(delivery);
```

Нижче приведений зовнішній вигляд такого контейнера:

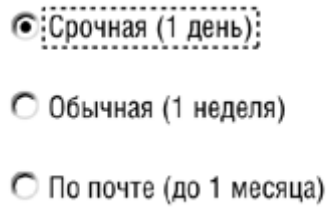


Звернете увагу, що розмір Checkbox має бути достатнім для розміщення не тільки поля для "галочки", але і для підпису.

Другий спосіб застосування компонент Checkbox призначений для організації "перемикачів" (radio buttons). В цьому випадку декілька екземплярів об'єднуються в групу, причому лише один з перемикачів може бути вибраний. В ролі такої групи виступає клас CheckboxGroup. Він не є візуальним, тобто ніяк не відображається на екрані. Його завдання – логічно об'єднати декілька Checkbox. Групу, до якої належить перемикач, можна указувати в конструкторі:

```
CheckboxGroup delivery = new CheckboxGroup();
Checkbox fast = new Checkbox(
      "Термінова (1 день)", delivery, true);
fast.setBounds(10, 10, 150, 20);
add(fast);
Checkbox normal = new Checkbox(
      "Звичайна (1 тиждень)", delivery, false);
normal.setBounds(10, 30, 150, 20);
add(normal);
Checkbox postal = new Checkbox(
      "Поштою (до 1 місяця)",
       delivery, false);
postal.setBounds(10, 50, 150, 20);
add(postal);
```

Нижче приведений зовнішній вигляд такого контейнера:



У прикладі при виклику конструктора класу Checkbox окрім тексту підпису і групи, указується стан перемикача (булевий параметр). Звернете увагу на зміну зовнішнього вигляду компоненту (форма поля змінилася з квадратною на круглу, як і прийнято в традиційних GUI).

Класи Choice і List

Компонент Choice служить для вибору користувачем одного з декількох можливих варіантів (випадний список). Розглянемо приклад:

```
Choice color = new Choice();
color.add("Білий");
color.add("Зелений");
color.add("Синій");
color.add("Чорний");
add(color);
```

У звичайному стані компонент відображає тільки вибраний варіант. В процесі вибору відображається весь набір варіантів. На малюнку представлений випадний список в обох станах:

Звернете увагу, що для компоненту Choice завжди є вибраний елемент.

Компонент List, подібно Choice, надає користувачеві можливість вибирати варіанти із списку запропонованих. Відмінність полягає в тому, що List відображає відразу декілька варіантів. Кількість задається в конструкторі:

```
List accessories = new List(3);
accessories.add("Чохол");
accessories.add("Навушники");
accessories.add("Акумулятор");
accessories.add("Блок живлення");
add(accessories);
```

От як виглядає такий компонент (верхня частина малюнка):

У списку знаходиться 4 варіанти. Проте в конструктор був переданий параметр 3, тому тільки 3 з них видні на екрані. За допомогою смуги прокрутки можна вибрати

останні варіанти. Малюнок ілюструє ще одна властивість List – можливість вибрати відразу декілька із запропонованих варіантів. Для цього треба або в конструкторі другим параметром передати булеве значення true (falsecоответствует вибору тільки одного елементу), або скористатися методом setMultipleMode.

Класи TextComponent, TextField, TextArea

Клас TextComponent є спадкоємцем Component і базовим класом для компонент, що працюють з текстом, –TextField і TextArea.

TextField дозволяє вводити і редагувати один рядок тексту. Різні методи дозволяють управляти вмістом цього поля введення:

```
TextField tf = new TextField();
tf.setText("Enter your name");
tf.selectAll();
add(tf);
```

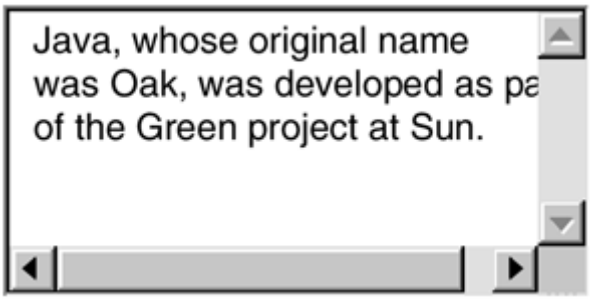
От як виглядатиме цей компонент:

У коді другий рядок встановлює значення тексту в полі введення (метод getText дозволяє набути поточного значення). Потім весь текст виділяється (є методи, що дозволяють виділити частину тексту).

Для будь-якої текстової компоненти можна задати особливий режим. У базовому класі Component визначене параметрів oenabled, яке, якщо виставлено в false, блокує всі призначені для користувача події. Для текстової компоненти вводиться нова властивість – editable (можна редагувати), методи для роботи з ним – isEditable isetEditable. Якщо текст не можна редагувати, але компонент доступний, то користувач може виділити частину, або весь текст, і, наприклад, скопіювати його в буфер.

TextField володіє ще однією властивістю. Все добре знайомі з полем введення для пароля – символи, що вводяться, не відображаються, замість них з'являється один і той же символ. Для TextField його можна встановити за допомогою методу setEchoChar (наприклад, setEchoChar('*')).

TextArea дозволяє вводити і проглядати багаторядковий текст. У конструктор передається кількість рядків і стовпців, які визначають розмір компоненту (обчислюється на основі середньої ширини символу). Ці параметри не обмежують довжину тексту, що вводиться, – при необхідності з'являються смуги прокрутки:



Клас Scrollbar

Клас Scrollbar дозволяє працювати із смугами прокрутки, які використовуються для переміщення внутрішньої області від початкової до кінцевої позиції. Смуга може бути розташована горизонтально або вертикально. Стрілки на кожному з її кінців служать для переміщення "на один крок" у відповідному напрямі. "Узявшись" курсором миші за бігунок, можна перемістити його в будь-яку позицію. За допомогою натисків миші по смузі прокрутки, але поза положенням бігунка, можна робити переміщення "на сторінку" вгору або вниз. Всі ці дії добре знайомі по багатьом призначеним для користувача інтерфейсам, наприклад, Windows. Вони повністю підтримуються компонентом Scrollbar.

Конструктор дозволяє задавати орієнтацію смуги прокрутки — для цього передбачені константи VERTICAL і HORIZONTAL. Крім того, за допомогою конструктора можна задати початкове положення бігунка, розмір "сторінки", а також мінімальне і максимальне значення, в межах яких лінійка прокрутки може змінювати параметр. Для отримання і установки поточного стану смуги прокрутки використовуються методи getValue і setValue. Нижче приведений приклад, в якому створюється і вертикальний, і горизонтальний Scrollbar.

```
int height = getHeight(), width = getWidth();
int thickness = 16;
Scrollbar hs = new Scrollbar(
        Scrollbar.HORIZONTAL, 50
        width/10, 0, 100);
Scrollbar vs = new Scrollbar(
        Scrollbar.VERTICAL, 50
        height/2, 0, 100);
add(hs); add(vs);
hs.setBounds(0, height - thickness
      width - thickness, thickness);
vs.setBounds(width - thickness, 0, thickness
      height - thickness);
```

В даному прикладі скроллируется, звичайно, порожня область:

Спадкоємці Container

Тепер перейдемо до розгляду стандартних контейнерів AWT.

Клас Panel

Подібно до того, як Canvas служить базовим класом для створення своїх компонент з особливим зовнішнім виглядом, клас Panel є суперкласом для нових контейнерів з особливою роботою з вкладеними компонентами. Втім, оскільки Panel, на відміну від Container, клас не абстрактний, його можна використовувати для ієрархічної організації складного призначеного для користувача інтерфейсу, групуючи компоненти в такі прості контейнери.

Клас ScrollPane

Вище був розглянутий компонент Scrollbar, призначений для смуги прокрутки. Якщо коштує завдання, наприклад, показати користувачеві графік деякої функції з можливістю перегляду для вивчення різних областей, необхідно створити дві смуги прокрутки, правильно їх встановити і надалі обробляти всі дії користувача, обчислювати нове положення видимої області, перемальовувати графік і так далі. В більшості випадків всі ці завдання може узяти на себе контейнер ScrollPane. Цей контейнер володіє рядом особливостей. По-перше, в нього можна помістити лише одну компоненту – при додаванні новою стара віддаляється. По-друге, відрізняється робота з вкладеним компонентом, чиї межі виходять за межі самого контейнера. Як ми розглядали раніше, "виступаючі" області ніколи не будуть відображені на екрані. У контейнері ScrollPane в цьому випадку з'являються смуги прокрутки (горизонтальна або вертикальна), за допомогою яких можна марнотратити видиму область і таким чином побачити весь компонент повністю. При цьому не потрібно робити ніяких додаткових дій – треба лише додати компонент в ScrollPane.

Може викликати здивування, чому вирішується додавання лише одного компоненту. А якщо потрібно марнотратити складнішу конструкцію? Тут і виявляється користь класу Panel. Всі елементи збираються в цей простий контейнер, який, у свою чергу, додається в ScrollPane. Конструктор цього класу може приймати параметр, задаючи логіку появи смуг прокрутки, – вони можуть бути видимі завжди, з'являтися в міру необхідності, або не з'являтися ніколи.

Клас Window

З досвіду роботи з віконними графічними інтерфейсами сучасних операційних систем ми звикли до того, що кожне застосування володіє одним або декількома вікнами.

Клас Window служить базовим класом для всіх вікон, що породжуються з Java. Зрозуміло, він також є інтерфейсом до відповідного вікна операційної системи, яка обслуговує вікна всіх застосувань. Як правило, використовується один з двох спадкоємців Window – класи Frame і Dialog, які будуть розглянуті наступними. Проте екземпляри Window не володіють ні рамкою, ні кнопками закриття або мінімізації вікна, а тому часто використовуються як заставки (так звані splash screen).

Конструктор Window вимагає як аргумент посилання на Window або Frame. Іншими словами, базові вікна не є самостійними, вони прив'язуються до інших вікон.

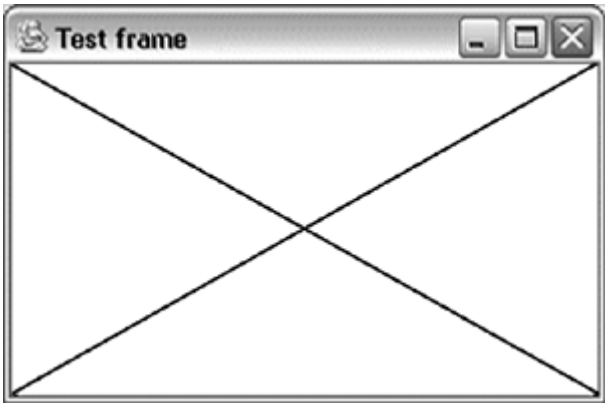
Класи Frame і Dialog

Клас Frame призначений для створення повнофункціональних вікон додатків – із смугою заголовка, рамкою, кнопками закриття, мінімізації і максимізації вікна. Оскільки Frame, як правило, є головним вікном додатку, він створюється невидимим, щоб можна було набудувати всі його параметри, додати всі вкладені контейнери і компоненти і лише потім відобразити його в підготовленому вигляді. Конструктор приймає текстовий параметр – заголовок фрейма.

Розглянемо приклад організації роботи з фреймом, який відображає компонент з першого прикладу лекції ("Алгоритм відмальовки").

```
public class TestCanvas extends Canvas {
  public void paint(Graphics g){
    g.drawLine(0, 0, getWidth(), getHeight());
    g.drawLine(0, getHeight(),getWidth(), 0);
  }

  public static void main(String arg[]){
    Frame f = new Frame("Test frame");
    f.setSize(400, 300);
    f.add(new TestCanvas());
    f.setVisible(true);
  }
}
```

Вікно запущеної програми виглядатиме таким чином:



Звернете увагу, що це вікно не закриватиметься по натисненню правої верхньої кнопки в заголовку. Причина буде роз'яснена нижче.

Якщо клас Frame призначений для створення основного вікна додатку, то екземпляри класу Dialog дозволяють відкривати додаткові вікна для взаємодії з користувачем. Це може потрібно, наприклад, для виведення критичного повідомлення, для введення параметрів і так далі. Вікно діалогу володіє стандартним оформленням – смуга заголовка, рамка. У правій частині смуги заголовка присутня лише одна кнопка – закриття вікна. Оскільки Dialog є несамостійним вікном, в конструктор необхідно передати посилання на батьківський фрейм або вікно іншого діалогу. Також можна задати заголовок вікна. Як і Frame, діалогове вікно створюється спочатку невидимим.

Важливою властивістю діалогового вікна є **модальність**. Якщо діалог модальний, то при його появі на екрані блокуються всі призначені для користувача події, що приходять в батьківське вікно такого діалогу.

Клас FileDialog

Клас FileDialog є модальним діалогом (спадкоємцем Dialog) і дозволяє легко організувати роботу з файлами. Цей клас призначений і для відкриття файлу (open file), і для збереження (save file). Вікно діалогу має зовнішній вигляд, прийнятий для поточної операційної системи.

Конструктор приймає як параметри посилання на батьківський фрейм, заголовок вікна і режим роботи. Для завдання режиму в класі визначено дві константи – LOAD і SAVE.

Після створення діалогу FileDialog його необхідно зробити видимим. Потім користувач робить свій вибір. Після закриття діалогу результат можна дізнатися за допомогою методів getDirectory (для отримання повного імені каталогу) і getFile (для отримання імені файлу). Якщо користувач натиснув кнопку "Відміна" ("Cancel"), то будуть повернені значення null.

Обробка призначених для користувача подій

Весь попередній розділ "Дерево компонентів" був присвячений завданню зовнішнього вигляду призначеного для користувача інтерфейсу. Проте до цих пір він був статичним. Перейдемо тепер до розгляду правил обробки різних подій, які можуть виникати як результат дій користувача, і не тільки. Модель обробки подій побудована на основі стандартного шаблону проектування ООП Observer/Observable. Як спостережуваний об'єкт виступає той або інший компонент AWT. Для нього можна задати один або декілька классов-наблюдателей. У AWT вони називаються слухачами (listener) і описуються спеціальними інтерфейсами, назва яких закінчується на слово Listener. Коли із спостережуваним об'єктом щось відбувається, створюється об'єкт "подія" (event), який "посилається" всім слухачам. Так слухач дізнається, наприклад, про дію користувача і може на нього відреагувати.

Кожна подія є підкласом класу java.util.EventObject. Події пакету AWT, які і розглядаються в даній лекції, є підкласами java.awt.AWTEvent. Для зручності класи різних подій і інтерфейси слухачів поміщені в окремий пакетjava.awt.event.

Перш, ніж заглиблюватися особливо подій, розглянемо, як вони застосовуються на практиці, на прикладі простої події – ActionEvent.

Подія ActionEvent

Розглянемо появу події ActionEvent на прикладі натиснення на кнопку. Припустимо, в нашому застосуванні створюється кнопка збереження файлу:

```
Button save = new Button("Save");
add(save);
```

Тепер, коли вікно додатку з цією кнопкою з'явиться на екрані, користувач зможе натиснути її. В результаті AWT згенерує ActionEvent. Щоб отримати і обробити його, необхідно зареєструвати слухача. Назва потрібного інтерфейсу прямо виходить з назви події – ActionListener. У нім всього один метод (у деяких слухачах їх декілька), який має один аргумент, – ActionEvent.

Оголосимо клас, який реалізує цей інтерфейс:

```
class SaveButtonListener
   implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent
            "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+"/"+
                  fd.getFile());
  }
}
```

Конструктор класу вимагає як параметр посилання на батьківський фрейм, без якого не вдасться создатьFileDialog. У методі actionPerformed класу ActionListener описуються дії, які необхідно зробити по натисненню користувачем на кнопку. А саме, відкривається файловий діалог, за допомогою якого визначається шлях збереження файлу. Для нашого прикладу досить вивести цей шлях на консоль.

Наступний крок – реєстрація слухача. Назва відповідного методу знову прямо виходить з назви інтерфейсу – addActionListener.

```
save.addActionListener(
    new SaveButtonListener(frame));
```

Все необхідне для обробки натиснення користувачем на кнопку зроблено. Нижче приведений повний лістинг програми:

```
import java.awt.*;
import java.awt.event.*;

public class Test {
  public static void main(String args[]){
    Frame frame = new Frame("Test Action");
    frame.setSize(400, 300);
    Panel p = new Panel();
    frame.add(p);
    Button save = new Button("Save");
    save.addActionListener(
        new SaveButtonListener(frame));
    p.add(save);

    frame.setVisible(true);
   }
}

class SaveButtonListener
    implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent
      "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+
                  fd.getFile());
  }
}
```

Після запуску програми з'явиться фрейм з однією кнопкою "Save". Якщо натиснути на неї, відкриється файловий діалог. Після вибору файлу на консолі відображається повний шлях до нього.

Події AWT

Отже, для кожної події AWT визначений клас XXEvent, інтерфейс XXListener, а в компоненті-джерелі подій – метод для реєстрації слухача addXXListener. Зовсім не обов'язково, щоб одна подія могла породжуватися лише одним компонентом як результат якогось однієї дії користувача. Наприклад, розглянутий ActionEvent генерується після натиснення на кнопку (Button), після натиснення клавіші Enter в полі введення тексту

(TextField), при подвійному клацанні миші по елементу списку (List) і так далі Дізнатися, які події генерує той або інший компонент, можна по наявності методів addXXListener.

Багато слухачів, на відміну від ActionListener, мають більш за один метод для різних видів подій. Наприклад, MouseMotionListener спостерігає за рухом миші і має два методи – mouseMoved (звичайний рух) і mouseDragged (переміщення з натиснутою кнопкою миші). Іноді буває необхідно працювати лише з одним методом, останні доводиться оголошувати і залишати порожніми. Щоб уникнути цієї даремної роботи, в пакеті java.awt.event оголошені допоміжні класи-адаптери, наприклад, MouseMotionAdapter (назва прямо виходить з назви слухача). Ці класи успадкуються від Object і реалізують відповідний інтерфейс. Адаптер – абстрактний клас, але абстрактних методів в нім немає, вони всі оголошені порожніми. Від такого класу можна успадковуватися і перевизначити тільки ті методи, які потрібні для додатку.

Класи повідомлень (event) містять допоміжну інформацію для обробки події. Метод getSource() повертає джерело об'єкту події. Конкретні спадкоємці AWTEvent можуть мати додаткові методи. Наприклад, MouseEvent повідомляє про натиснення кнопки миші, а його методи getX і getY повертають координати точки, де ця подія відбулася. Разом з методом addXXListener важливу роль грає removeXXListener. Оскільки в Java непотрібні об'єкти віддаляються з пам'яті автоматичним складальником сміття, яке підраховує посилання на об'єкти, важливо стежити за тим, щоб не залишалося посилань на непотрібні об'єкти. Якщо слухач вже виконав свою роль і не потрібніший, то явно в програмі може не залишитися посилань на нього, проте компонент зберігатиме його в своєму списку слухачів. Щоб дати спрацювати garbage collector, необхідно скористатися методом removeXXListener.

Розглянемо оглядовий всі події AWT і відповідних ним слухачів, визначених в Java починаючи з версії 1.1.

MouseMotionListener і MouseEvent

Ця подія розглядалася вищим в прикладі. Воно відповідає за переміщення курсора миші. Відповідний слухач має два методи – mouseMoved для звичайного переміщення і mouseDragged для переміщення з натиснутою кнопкою миші. Звернете увагу, що цей слухач працює не з подією MouseMotionEvent (такого класу немає), а з MouseEvent, як і MouseListener.

MouseListener і MouseEvent

Цей слухач має методи mouseEntered і mouseExited. Перший викликається, коли курсор миші з'являється над компонентом, а другий – коли виходить з його меж. Для обробки натиснень кнопки миші служать три методи: mousePressed, mouseReleased і mouseClicked. Якщо користувач натиснув, а потім відпустив кнопку, то слухач отримає всі три події у вказаному порядку. Якщо клацань було декілька, то метод getClickCount класу MouseEvent поверне кількість. Як вже указувалося, методи getX і getY повертають координати точки, де відбулася подія. Щоб визначити, яка кнопка миші була натиснута, потрібно скористатися методом getModifiers і порівняти результат з константами:

```
(event.getModifiers() &
```

```
MouseEvent.BUTTON1_MASK)!=0
```

Як правило, перша кнопка відповідає лівій кнопці миші.

KeyListener і KeyEvent

Цей слухач відстежує натиснення клавіш клавіатури і має три методи: keyTyped, keyPressed, keyReleased. Перший відповідає за введення чергового Unicode-символа з клавіатури. Метод keyPressed сигналізує про натиснення, akeyReleased – про відпуск деякої клавіші. Взаємозв'язок між цими подіями може бути нетривіальним. Наприклад, якщо користувач натисне і утримуватиме клавішу Shift і в цей час натисне клавішу "A", відбудеться одна подія типу keyTyped і декілька keyPressed/Released. Якщо користувач натисне і утримуватиме, наприклад, пропуск, то після першого keyPressed буде багато разів викликаний метод keyTyped, а після відпуску – keyReleased. У класі KeyEvent визначена безліч констант, які дозволяють точно ідентифікувати, яка клавіша була натиснута і в якому стані знаходилися службові клавіші (Ctrl, Alt, Shift і так далі).

FocusListener і FocusEvent

У кожному застосуванні один з компонентів володіє фокусом і може отримувати події від клавіатури. Фокус можна перемістити, наприклад, клацнувши мишкою по іншому компоненту, або натиснувши клавішу Tab. Інтерфейс FocusListener містить два методи – focusGained і focusLost (получен/потерян).

TextListener і TextEvent

Компоненти-наследникі TextComponent відповідають за введення тексту і породжують TextEvent. Слухач має один метод textValueChanged. З його допомогою можна відстежувати кожну зміну тексту, щоб, наприклад, видавати користувачеві підказку, грунтуючись на перших введених символах.

ItemListener і ItemEvent

Цю подію можуть генерувати такі класи, як Checkbox, Choice, List. Слухач має один методitemStateChanged, який сигналізує про зміну стану елементів.

AdjustmentListener і AdjustmentEvent

Ця подія генерується компонентом ScrollBar. Слухач має один метод adjustmentValueChanged, що сигналізує про зміну стану смуги прокрутки.

WindowListener і WindowEvent

Ця подія сигналізує про зміну стану вікна (клас Window і його спадкоємці).

Розглянемо особливо один з методів слухача – windowClosing. Цей метод викликається, коли користувач робить спробу закрити вікно, наприклад, натискаючи на відповідну кнопку в заголовку вікна. Ми бачили з прикладів раніше, що в Java вікна при

цьому не закриваються. Річ у тому, що AWT лише посилаетWindowEvent у відповідь на таку дію, а ініціювати закриття вікна повинен програміст:

```
public class WindowClosingAdapter
     extends WindowAdapter {
  public void windowClosing(WindowEvent e)
  {
    ((Window)e.getSource()).dispose();
  }
}
```

Оголошений адаптер в методі windowClosing отримує посилання на вікно, від якого прийшла подія. Зазвичай ми користувалися методом setVisible(false), щоб зробити компонент невидимим. Але оскільки Windowавтоматически породжує вікно операційної системи, існує спеціальний метод dispose, який звільняє всі системні ресурси, пов'язані з цим вікном.

Коли вікно буде закрито, у слухача викликається ще один метод – windowClosed.

ComponentListener і ComponentEvent

Цю подію відображає зміна основних параметрів компоненту – положення, розмір, властивість visible.

ContainerListener і ContainerEvent

Ця подія дозволяє відстежувати зміну списку що містяться в цьому контейнері компонент.

З розвитком Java в AWT з'являються і інші події, наприклад, що дозволяють підтримувати коліщатко миші. Проте всі вони працюють по точно такій же схемі, а тому їх можна легко освоїти самостійно.

Обробка подій за допомогою внутрішніх класів

Ще в лекції, присвяченій оголошенню класів, було вказано, що в тілі класу можна оголошувати внутрішні класи. До цих пір така можливість не була затребувана в наших прикладах, проте обробка подій AWT – якраз слушна нагода розглянути такі класи на прикладі анонімних класів.

Припустимо, в додаток додається кнопка, якої слід додати слухача. Часто буває зручно описати логіку дій в окремому методі того ж класу. Якщо вводити слухача, як робилося ранішим – в окремому класі, то це відразу породжує ряд незручностей: з'являється новий, малозмістовний клас, якому до того ж необхідно передати посилання на початковий клас і так далі.

Набагато зручніше поступити таким чином:

```
Button b = new Button();
```

```
b.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    processButton();
  }
});
```

Розглянемо детально, що відбувається в даному прикладі. Спочатку створюється кнопка, у якої потім викликається метод addActionListener. Звернемо увагу на аргумент цього методу. Може складеться враження, що проводиться спроба створити екземпляр інтерфейсу (new ActionListener()), проте це неможливо. Справу міняє фігурна дужка, яка указує, що породжується екземпляр нового класу, оголошення якого послідує за цією дужкою. Клас успадкується від Object і реалізує інтерфейс ActionListener. Йому необхідно реалізувати метод actionPerformed, що і робиться. Звернете увагу на ще одну важливу деталь – в цьому методі викликається processButton. Це метод, який ми планували розмістити в зовнішньому класі. Таким чином, внутрішній клас може безпосередньо звертатися до методів зовнішнього класу.

Такий клас називається анонімним, він не має свого імені. Проте правило, згідно якому компілятор завжди створює .class-файл для кожного класу Java, діє і тут. Якщо зовнішній клас називається Test, то після компіляції з'явиться файл Test$1.class.

Приклад додатку, що використовує модель подій

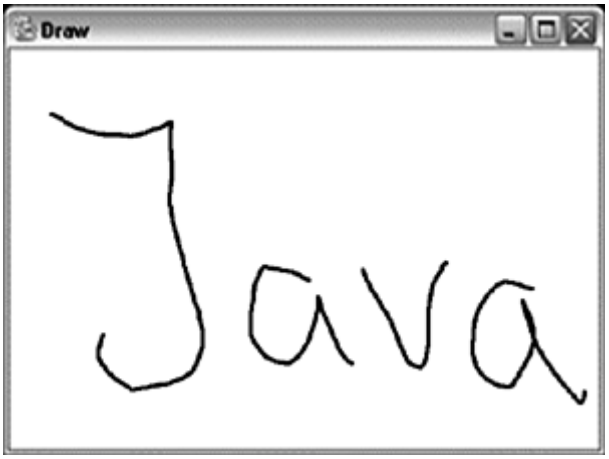
На закінчення теми, присвяченої подіям, розглянемо приклад додатку, який активно їх використовує. Спробуємо написати примітивний графічний редактор, який дозволяє малювати за допомогою курсора, – якщо переміщати його з натиснутою кнопкою миші, то з'являтиметься лінія. Натиснення пропуску очищає поле.

```
import java.awt.*;
import java.awt.event.*;
public class DrawCanvas extends Canvas {
  private int lastX, lastY;
  private int ex, ey;
  private boolean clear=false;

  public DrawCanvas () {
    super();
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e){
        lastX = e.getX();
        lastY = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e){
        ex=e.getX();
```

```
            ey=e.getY();
            repaint();
          }
        });

        addKeyListener(new KeyAdapter() {
          public void keyTyped(KeyEvent e){
            if (e.getKeyChar()==' ') {
              clear = true;
              repaint();
            }
          }
        });
      }

      public void update(Graphics g){
        if (clear) {
          g.clearRect(0, 0, getWidth(), getHeight());
          clear = false;
        } else {
          g.drawLine(lastX, lastY, ex, ey);
          lastX=ex;
          lastY=ey;
        }
      }
      public static void main(String s[]){
        final Frame f = new Frame("Draw");
        f.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e){
            f.dispose();
          }
        });
        f.setSize(400, 300);

        final Canvas з = new DrawCanvas();
        f.add(c);

        f.setVisible(true);
      }
    }
```

Клас DrawCanvas і є тим полем, на якому можна малювати. У його конструкторі ініціалізувалися всі необхідні слухачі. У разі приходу події ініціалізувалося перемальовування (метод repaint), логіка якого описана в update. Метод main , що запускається, ініціалізував frame, не забуваючи про windowClosing.

В результаті можна що-небудь намалювати:



Аплети

Перейдемо до розгляду аплетів (applets) – Java-приложений, які виконуються в браузер як частина HTML-страницы. Це означає, що такі застосування завжди візуальні. Дійсно, клас Applet є спадкоємцем AWT-компонента Panel. Сам клас знаходиться в пакеті java.applet.

Життєвий цикл аплета

Важливим питанням для розуміння роботи аплетів є їх життєвий цикл. Він описується чотирма методами.

init

Цей метод викликається браузером при конструюванні аплета. Часто всі дії, що ініціалізували, описуються тут, а не в конструкторі. Це може бути, наприклад, створення AWT-компонент, запуск потоків виконання, встановлення мережевих з'єднань і так далі

start

Цей метод викликається після ініціалізації аплета. Він потрібний з наступної причини. Аплет може містити якісь динамічні частини, наприклад, анімацію або рядок, що біжить. Якщо користувач скористається яким-небудь посиланням і піде із сторінки з аплетом, браузер не почне його знищувати – адже користувач може повернутися (натиснувши в браузер кнопку Back), і він чекатиме, що аплет збереже свій стан. Значить, аплет може опинитися в неактивному стані, коли краще припинити динамічні процеси для економії системних ресурсів.

Метод start сигналізує про перехід в активний стан.

stop

Цей метод завжди викликається після методу start і сигналізує про перехід в пасивний стан.

destroy

Після закінчення роботи аплет необхідно коректно видалити, щоб він мав можливість звільнити займані ресурси. Для цього браузер викликає метод destroy.

У останньому аплет є повноцінним AWT-компонентом і в методі init може додати інші компоненти для створення призначеного для користувача інтерфейсу, або навіть відкрити новий фрейм. Єдине, але істотне обмеження – це умова безпеки. Адже код аплета викачується по мережі, а значить, може містити в собі небезпечні дії. Тому браузер запускає віртуальну машину з обмеженнями – аплетам заборонено звертатися до файлової структури, заборонено встановлювати мережеві з'єднання з ким-небудь, окрім сервера, звідки вони були завантажені, всі вікна, що знов відкриваються, позначаються попередженням. Більш того, користувач може так набудувати свій браузер, що зовсім заборонить виконання Java. Можна, навпаки, дозволити аплетам те ж, що і локальним застосуванням.

Є і ще одне обмеження – версія Java, підтримувана браузером. Як мовилося в першій лекції, найпопулярніший на даний момент браузер – MS Internet Explorer – зупинився на підтримці лише Java 1.1, і те не в повному об'ємі. В деяких випадках можна скористатися додатковим продуктом Sun – Java Plug-in, який дозволяє встановити на браузер JVM будь-якої версії.

Продовжимо розгляд аплетів.

HTML-тег

Раз аплет є частиною HTML-сторінки, значить, необхідно якимсь чином вказати, де саме він розташовується. Для цього служить спеціальний тег <applet>. Синтаксис тега <APPLET> в даний час такий:

```
<APPLET
  CODE = appletFile
  WIDTH = pixels
  HEIGHT = pixels
  [ARCHIVE = jarFiles]
  [CODEBASE = codebaseURL]
  [ALT = alternateText]
  [NAME = appletInstanceName]
  [ALIGN = alignment]
  [VSPACE = pixels]
  [HSPACE = pixels]
>
[HTML-текст, що відображається за відсутності
підтримка Java]
</APPLET>
```

- CODE = appletClassFile; CODE – обов'язковий атрибут, задаючий ім'я файлу, в якому міститься опис класу аплета. Ім'я файлу задається відносно codebase, тобто або від поточного каталогу, або від каталогу, вказаного в атрибуті CODEBASE.
- WIDTH = pixels

- HEIGHT = pixels; WIDTH і HEIGHT - обов'язкові атрибути, задаючі розмір області аплета на HTML-сторінці.
- ARCHIVE = jarFiles; Цей необов'язковий атрибут задає список jar-файлов (розділяється комами), які заздалегідь завантажуються в Web-браузер. У них можуть міститися класи, зображення, звук і будь-які інші ресурси, необхідні аплету. Архівація найбільш необхідна саме аплетам, оскільки їх код і ресурси передаються через мережу.
- CODEBASE = codebaseURL; CODEBASE – необов'язковий атрибут, задаючий базовий URL коду аплета; є каталогом, в якому виконуватиметься пошук виконуваного файлу аплета (CODE, що задається в ознаці). Якщо цей атрибут не заданий, за умовчанням використовується каталог даного HTML-документа. За допомогою цього атрибуту можна на сторінці одного сайту розмістити аплет, що знаходиться на іншому сайті.
- ALT = alternateAppletText; Ознака ALT – необов'язковий атрибут, задаючий коротке текстове повідомлення, яке має бути виведене (як правило, у вигляді спливаючої підказки при знаходженні курсора миші над областю аплета) в тому випадку, якщо використовуваний браузер розпізнає синтаксис тега<applet>, але виконувати аплети не уміє. Це не те ж саме, що HTML-текст, який можна вставляти між <applet> і </applet> для браузеров, що взагалі не підтримують аплетів.
- NAME = appletInstanceName; NAME – необов'язковий атрибут, використовуваний для привласнення імені даному екземпляру аплета. Імена аплетам потрібні для того, щоб інші аплети на цій же сторінці могли знаходити їх і спілкуватися з ними, а також для звернень з Java Script.
- ALIGN = alignment
- VSPACE = pixels
- HSPACE = pixels; Ці три необов'язкові атрибути призначено для того ж, що і в теге IMG. ALIGN задає стиль вирівнювання аплета, можливі значення: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

Наступні два задають ширину вільного простору в пикселах зверху і знизу аплета (VSPACE), а також зліва і праворуч від нього (HSPACE).

Приведемо приклад простого аплета:

```
import java.applet.*;
import java.awt.*;
public class HelloApplet extends Applet {
  public void init() {
    add(new Label("Hello"));
  }
}
```

HTML-тег для нього:

```
<applet code=HelloApplet.class
      width=200 height =50>
</applet>
```

Передача параметрів

Існує дуже корисна можливість передавати з HTML параметри в аплет. Таким чином, можна набудувати програму без необхідності міняти її початковий код.

У HTML параметри указуються таким чином:

```
<applet code=HelloApplet.class
      width=200 height =50>
<param name="text" value="Hello!!!">
</applet>
```

У аплеті значення параметрів прочитується таким чином:

```
import java.applet.*;
import java.awt.*;

public class HelloApplet extends Applet {
  public void init() {
    String text = getParameter("text");
    add(new Label(text));
  }
}
```

Текст, що тепер виводиться, можна настроювати з HTML.

Інтерфейс AppletContext

Доступ до цього інтерфейсу з аплета надається методом getAppletContext. З його допомогою аплет може взаємодіяти із сторінкою, звідки він був завантажений, і з браузером. Так, саме в цьому інтерфейсі визначений метод getApplet, за допомогою якого можна звернутися по імені до іншого аплета, що знаходиться на тій же сторінці.

Метод showStatus міняє текст поля статусу у вікні браузера.

Метод showDocument дозволяє завантажити нову сторінку в браузер.

Менеджери компоновки

При розміщенні компонент в контейнері спочатку завжди здається зручним задавати їх розмір і положення явно за допомогою методу setBounds. Проте далі стають очевидні недоліки такого підходу. Наприклад, зручно надати користувачеві можливість змінювати розмір фрейма, а це означає необхідність перебудовувати компоненти. Розвиток додатку також може привести до додавання або видалення компоненту, після чого доведеться перераховувати координати елементів, що залишилися.

Є проблеми і іншого характеру. Ми указували, що JVM вибирає шрифти з наявних в системі. Тому під різними платформами вони можуть опинитися різного розміру або нахилу. В результаті додаток, що красиво виглядає на машині розробника, може

"поплисти" у клієнта. Навіть під однією платформою користувач може змінити системні налаштування, унаслідок чого зовнішній вигляд додатку може змінитися не в кращу сторону. Всі ці міркування наводять на думку, що було б корисно якимсь чином автоматизувати розташування компонентів. Саме для цієї мети служать менеджери компоновки. Їх завдання – обчислити і встановити розмір і місцеположення компонентів в контейнері. Взагалі кажучи, вони можуть використовувати наступний набір параметрів для своїх обчислень:

- розмір контейнера;
- початкове положення і розмір компоненту;
- його порядковий номер в наборі компонентів;
- спеціальний параметр-обмежувач (constraint), який може бути встановлений при додаванні компоненту.

У AWT кожен контейнер володіє менеджером компоновки. Якщо він равенnull, то використовуються явні параметри компонентів. Справжні ж класи менеджерів повинні реалізовувати інтерфейс LayoutManager. Цей інтерфейс приймає як constraints рядок (String). З часом це було визнано недостатньо гнучким (фірми почали розробляти і пропонувати свої менеджери, що володіють самою різною функціональністю). Тому був доданий новий інтерфейс –LayoutManager2, що приймає як обмежувач constraints.

Розглянемо роботу декількох найбільш поширених менеджерів компоновки. Але перед цим відзначимо загальний для них всіх факт. Річ у тому, що не завжди вся область контейнера підходить для розміщення в ній компонент. Наприклад, фрейм має рамку і смугу заголовка. В результаті його корисна площа менша. Тому всі менеджери компоновки починають із звернення до методу getInsets класу Container. Цей метод повертає значення типу Insets. Це клас, який має чотири відкриті поля, – top, right, bottom, left, значення яких описують відступи зі всіх чотирьох сторін, які необхідно зробити, щоб отримати область, доступну для розташування компонент.

Клас FlowLayout

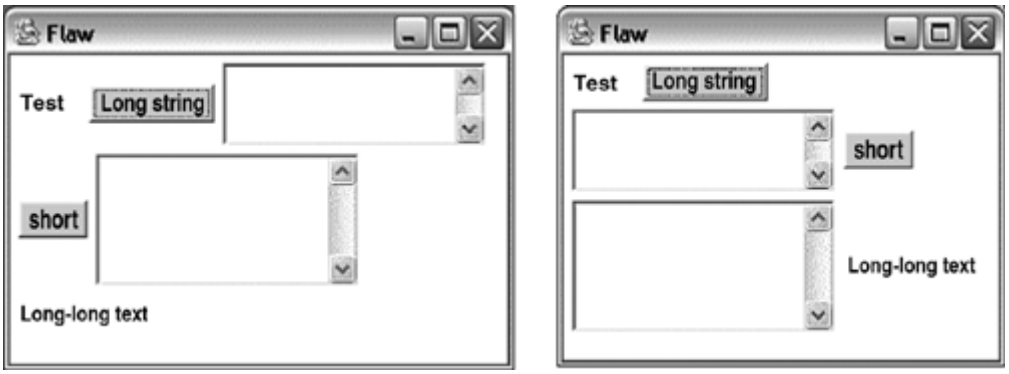
Цей менеджер є стандартним для Panel. Він не міняє розмір компонент, а тільки розташовує їх один за іншим в лінію, як букви в рядку. Коли закінчується перший "рядок", він переходить на наступну, і так далі, поки або не закінчиться область контейнера, або не будуть розташовані всі компоненти. Як параметри конструктору можна передати значення вирівнювання по горизонталі (визначені константи LEFT, RIGHT, CENTER – значення за умовчанням), а також величину необхідних відступів між компонентами по вертикалі (vgap) і горизонталі (hgap). Їх значення за умовчанням – 5 пікселів.

Розглянемо приклад:

```
final Frame f = new Frame("Flaw");
f.setSize(400, 300);
f.setLayout(new FlowLayout(FlowLayout.LEFT));
f.add(new Label("Test"));
f.add(new Button("Long string"));
f.add(new TextArea(2, 20));
```

```
f.add(new Button("short"));
f.add(new TextArea(4, 20));
f.add(new Label("Long-long text"));
f.setVisible(true);
```

Якщо тепер міняти розмір цього фрейма, то можна бачити, як перерозподіляються компоненти:



Клас BorderLayout

Цей менеджер є стандартним для контейнера Window і його спадкоємців Frame і Dialog.

BorderLayout використовує обмежувач. При додаванні компоненту необхідно вказати одну з 5 констант, визначених в цьому класі: NORTH, SOUTH, EAST, WEST, CENTER (використовується за умовчанням). Першими маються в своєму розпорядженні північний і південний компонент. Їх висота не змінюється, а ширина стає рівній ширині контейнера. Північний компонент поміщається на самий верх контейнера, південний, – вниз. Потім розташовуються східний і західний компоненти. Їх ширина не міняється, а висота стає рівній висоті контейнера за вирахуванням місця, яке зайняли перші дві компоненти. Нарешті, місце, що все залишилося, займає центральна компоненту.

Розглянемо приклад:

```
final Frame f = new Frame("Border");
f.setSize(200, 150);
f.add(new Button("North")
    BorderLayout.NORTH);
f.add(new Button("South")
    BorderLayout.SOUTH);
f.add(new Button("West")
    BorderLayout.WEST);
f.add(new Button("East")
    BorderLayout.EAST);
f.add(new Button("Center")
    BorderLayout.CENTER);
f.setVisible(true);
```

От як виглядає такий фрейм:

І в цьому менеджерові є параметри hgap і vgap (за умовчанням їх значення дорівнює нулю).

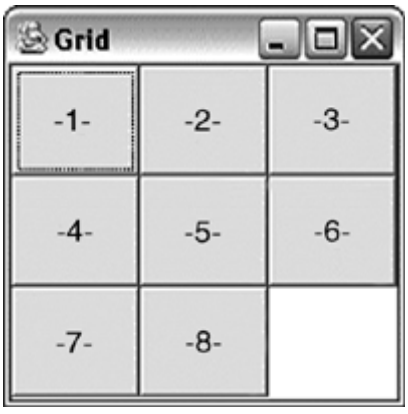
Клас GridLayout

Цей менеджер поступає таким чином – він розділяє весь контейнер на однакові прямокутні сектори (звідси і його назва – грати). Далі послідовно кожен компонент повністю займає свій сектор (таким чином, вони всі стають однакового розміру).

У конструкторі указується кількість рядків і стовпців для розбиття:

```
final Frame f = new Frame("Grid");
f.setSize(200, 200);
f.setLayout(new GridLayout(3, 3));
for (int i=0; i<8; i++) {
  f.add(new Button("-"+(i+1)+"-"));
}
f.setVisible(true);
```

От як виглядає такий фрейм:



І в цьому менеджерові є параметри hgap і vgap (за умовчанням їх значення дорівнює нулю).

Клас CardLayout

Цей менеджер поводиться подібно до колоди карт. Водночас видно лише один компонент, і він займає всю область контейнера. Програміст може управляти тим, який саме компонент показується користувачеві.

Висновок

Бібліотека AWT має безліч класів і внутрішніх механізмів. Нові версії Java додають нові можливості і переглядають старі. Проте, основні концепції були детально розглянуті в цій лекції і на їх основі можна побудувати повнофункціональний графічний інтерфейс користувача (GUI).

Стандартні компоненти AWT ієрархічно впорядковані в дерево спадкоємства з класом Component у вершині. Важливим його спадкоємцем є клас Container, який може зберігати набір компонентів. Прямі спадкоємці Component складають набір елементів ("контролів", від англ. controls), що управляють, а наслідники Container – набір контейнерів для угрупування і розташування компонентів. Для спрощення розміщення окремих елементів призначеного для користувача інтерфейсу застосовуються менеджери компоновки (Layout managers).

Особливе місце в AWT займає процедура перемальовки компонентів, яка може ініціюватися як операційною системою, так і програмою. Спеціальні класи служать для завдання таких атрибутів, як колір, шрифт і так далі

Один із спадкоємців Container – клас Window, який є самостійним вікном в багатовіконній операційній системі. Два його спадкоємця – Dialog і Frame. Для роботи з файлами визначений спадкоємець Dialog – FileDialog.

Нарешті, висловлюються принципи моделі подій від користувача, що дозволяє обробляти всі дії, які проводить клієнт, працюючи з програмою. 11 подій і відповідних ним інтерфейсів надають все необхідне для написання повноцінної GUI-програми.

Аплети – невеликі програми, призначені для роботи в браузерах як невеликі частини HTML-сторінок. Клас java.applet.Applet є спадкоємцем Panel, а тому володіє всіма властивостями AWT-компонент. Були представлені етапи життєвого циклу аплета, відмінного від циклу звичайного застосування, яке запускається методом main. Для розміщення аплета на HTML-сторінок необхідно використовувати спеціальний тег<applet>. Окрім цього, можна указувати спеціальні параметри, щоб аплет настроювався без перекомпіляції коду.

ТЕМА. *Потоки виконання. Синхронізація.*

План

1. *Особливості створення багатопотокових застосувань.*
2. *Потоки виконання.*
3. *Синхронізація.*

Рекомендована література: 1, 2,4.

Теоретичні відомості

До цих пір у всіх даних прикладах малося на увазі, що в один момент часу виконується лише один вираз або дія. Починаючи з найперших версій, віртуальні машини Java підтримують багатопоточність, тобто підтримку декількох потоків виконання (threads) одночасно.

У даній лекції спочатку розглядаються переваги такого підходу, способи реалізації і можливі недоліки.

Потім описуються базові класи Java, які дозволяють запускати потоки виконання і управляти ними. При одночасному зверненні декількох потоків до одних і тих же даних може виникнути ситуація, коли результат програми залежатиме від випадкових чинників, таких як тимчасове чергування виконання операцій декількома потоками. У такій ситуації стають необхідними механізми синхронізації, що забезпечують послідовний, або монопольний, доступ. У Java цієї мети служить ключове слово synchronized. Заздалегідь буде розглянутий підхід до організації зберігання даних у віртуальній машині. На закінчення розглядаються методи wait(), notify(), notifyAll()класса Object.

Багатопотокова архітектура

Не претендуючи на повноту викладу, розглянемо загальний пристрій багатопотокової архітектури, її достоїнства і недоліки. Реалізацію багатопотокової архітектури найпростіше уявити собі для системи, в якій є декілька центральних обчислювальних процесорів. В цьому випадку для кожного з них можна виділити завдання, яке він виконуватиме. В результаті декілька завдань обслуговуватимуться одночасно.

Проте виникає питання – яким же тоді образом забезпечується багатопотоковість в системах з одним центральним процесором, який, в принципі, виконує лише одне обчислення водин момент часу? У таких системах застосовується процедура квантування часу (time-slicing). Час розділяється на невеликі інтервали. Перед початком кожного інтервалу ухвалюється рішення, який саме потік виконання відпрацьовуватиметься впродовж цього кванта часу. За рахунок частого перемикання між завданнями емулюється багатопотокова архітектура.

Насправді, як правило, і для багатопроцесорних систем застосовується процедура квантування часу. Річ у тому, що навіть в могутніх серверах додатків процесорів не так багато (рідко буває більше десяти), а потоків виконання запускається, як правило, значно більше. Наприклад, операційна система Windows без єдиного запущеного застосування ініціалізувала десятки, а то і сотні потоків. Квантування часу дозволяє спростити управління виконанням завдань на всіх процесорах.

Тепер перейдемо до питання про переваги – навіщо взагалі може потрібно більш за один потік виконання?

Серед програмістів існує думка, що багатопотокові програми працюють швидше. Розглянувши спосіб реалізації багатопотоковості, можна стверджувати, що такі програми працюють насправді повільніше. Дійсно, для перемикання між завданнями на кожному інтервалі потрібний додатковий час, адже вони (перемикання) відбуваються досить часто. Якби процесор, не відволікаючись, виконував завдання послідовно, одну за іншою, він завершив би їх помітно швидше. Отже, переваги полягають не в цьому.

Перший тип додатків, який виграє від підтримки багатопотоковості, призначений для завдань, де дійсно потрібно виконувати декілька дій одночасно. Наприклад, буде цілком обгрунтовано чекати, що сервер загального користування почне обслуговувати декілька клієнтів одночасно. Можна легко уявити собі приклад з сфери обслуговування, коли є декілька потоків клієнтів і бажано обслуговувати їх всіх одночасно.

Інший приклад – активні ігри, або подібні застосування. Необхідно одночасно опитувати клавіатуру і інші пристрої введення, щоб реагувати на дії користувача. В той же час необхідний розраховувати і перемальовувати стан ігрового поля, що змінюється.

Зрозуміло, що у разі відсутності підтримки багатопотоковості для реалізації подібних застосувань потрібно було б реалізовувати квантування часу уручну. Умовно кажучи, одну секунду перевіряти стан клавіатури, а наступну – перераховувати і перемальовувати ігрове поле. Якщо порівняти дві реалізації time-slicing, одну – на низькому рівні, виконану засобами, як правило, операційної системи, іншу – виконувану уручну, на мові високого рівня, мало відповідного для таких завдань, то стає зрозумілим перше і, можливо, головна перевага багатопотоковості. Вона забезпечує найбільш ефективну реалізацію процедури квантування часу, істотно полегшуючи і укорочувавши процес розробки додатку. Код перемикання між завданнями на Java виглядав би куди громіздкіше, ніж незалежний опис дій для кожного потоку.

Наступна перевага виникає з того, що комп'ютер складається не тільки з одного або декількох процесорів. Обчислювальний пристрій – лише один з ресурсів, необхідних для виконання завдань. Завжди є оперативна пам'ять, дискова підсистема, мережеві підключення, периферія і так далі Припустимо, користувачеві потрібно роздрукувати великий документ і викачати великий файл з мережі. Очевидно, що обидва завдання вимагають зовсім незначної участі процесора, а основні необхідні ресурси, які будуть задіяні на межі можливостей, у них різні – мережеві підключення і принтер. Значить, якщо виконувати завдання одночасно, то уповільнення від організації квантування часу буде незначним, процесор легко справиться з обслуговуванням обох завдань. В той же час, якщо кожне завдання окремо займало, скажімо, дві години, то цілком імовірно, що і при одночасного виконання буде потрібно не більш за ті ж два годинники, а зроблено при цьому буде значно більше.

Якщо ж завдання в основному завантажують процесор (наприклад, математичні розрахунки), то їх одночасного виконання займе в кращому разі стільки ж часу, що і послідовне, а то і більше.

Третя перевага з'являється із-за можливості гнучкіше управляти виконанням завдань. Припустимо, користувач системи, не підтримуючою багатопотоковість, вирішив викачати великий файл з мережі, або провести складне обчислення, що займає, скажімо, дві години. Запустивши завдання на виконання, він може раптово виявити, що йому потрібний не цей, а який-небудь інший файл (або обчислення з іншими початковими параметрами). Проте якщо додаток займається тільки роботою з мережею (обчисленнями) і не реагує на дії користувача (не обробляються дані з пристроїв введення, таких як клавіатура або миша), то він не зможе швидко виправити помилку. Виходить, що процесор виконує більша кількість обчислень, але при цьому приносить значно менше користі.

Процедура квантування часу підтримує пріоритети (priority) завдань. У Java пріоритет представляється цілим числом. Чим більше число, тим вище пріоритет. Строгих правил роботи з пріоритетами немає, кожна реалізація може поводитися по-різному на різних платформах. Проте є загальне правило – потік з вищим пріоритетом отримуватиме більшу кількість квантів часу на виконання і таким чином зможе швидше виконувати свої дії і реагувати на дані, що поступають.

У описаному прикладі представляється розумним запустити додатковий потік, що відповідає за взаємодію з користувачем. Йому можна поставити високий пріоритет, оскільки у разі бездіяльності користувача цей потік практично не займатиме ресурси машини. У разі ж активності користувача необхідно щонайшвидше провести необхідні дії, щоб забезпечити максимальну ефективність роботи користувача.

Розглянемо тут же ще одну властивість потоків. Раніше, коли розглядалися однопоточні застосування, завершення обчислень однозначно приводило до завершення виконання програми. Тепер же додаток повинен працювати до тих пір, поки є хоч один потік виконання, що діє. В той же час частий бувають потрібні обслуговуючі потоки, які не мають ніякого сенсу, якщо вони залишаються в системі одні. Наприклад, автоматичний складальник сміття в Java запускається у вигляді фонового (низькопріоритетного) процесу. Його завдання – відстежувати об'єкти, які вже не використовуються іншими потоками, і потім знищувати їх, звільняючи оперативну пам'ять. Зрозуміло, що робота одного потоку garbage collector'а не має ніякого сенсу.

Такі обслуговуючі потоки називають демонами (daemon), цю властивість можна встановити будь-якому потоку. У результаті додаток виконується до тих пір, поки є хоч би один потік не-демон.

Розглянемо, як потоки реалізовані в Java.

Базові класи для роботи з потоками

Клас Thread

Потік виконання в Java представляється екземпляром класу Thread. Для того, щоб написати свій потік виконання, необхідно успадковуватися від цього класу і перевизначити метод run(). Наприклад

```
public class MyThread extends Thread {
  public void run() {
    // деяка довга дія, обчислення
    long sum=0;
    for (int i=0; i<1000; i++) {
      sum+=i;
    }
    System.out.println(sum);
  }
}
```

Метод run() містить дії, які повинні виконуватися в новому потоці виконання. Щоб запустити його, необхідно створити екземпляр класу-наступника і викликати успадкований метод start(), який повідомляє віртуальну машину, що потрібно запустити новий потік виконання і почати виконувати в нім метод run().

```
MyThread t = new MyThread();
t.start();
```

Внаслідок чого на консолі з’явиться результат:

```
499500
```

Коли метод run() завершений (зокрема, зустрівся вираз return), потік виконання зупиняється. Проте ніщо не перешкоджає запису нескінченного циклу в цьому методі. В результаті потік не перерве свого виконання і буде зупинений тільки при завершенні роботи всього застосування.

Інтерфейс Runnable

Описаний підхід має один недолік. Оскільки в Java множинне спадкоємство відсутнє, вимога успадковуватися від Thread може привести до конфлікту. Якщо ще раз подивитися на приведений вище приклад, стане зрозуміло, що спадкоємство проводилося тільки з метою перевизначення методу run(). Тому пропонується простіший спосіб створити свій потік виконання. Досить реалізувати інтерфейс Runnable, в якому оголошений тільки один метод, – вже знайомий void run(). Запишемо приклад, приведений вище, за допомогою цього інтерфейсу:

```
public class MyRunnable implements Runnable {
  public void run() {
    // деяка довга дія, обчислення
    long sum=0;
    for (int i=0; i<1000; i++) {
      sum+=i;
    }
    System.out.println(sum);
  }
```

```
}
```

Також трохи міняється процедура запуску потоку:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Якщо раніше об’єкт, що представляє сам потік виконання, і об’єкт з методом run(), що реалізовує необхідну функціональність, були об’єднані в одному екземплярі класу MyThread, то тепер вони розділені. Який з двох підходів зручніше, вирішується у кожному конкретному випадку.

Підкреслимо, що Runnable не є повною заміною класу Thread, оскільки створення і запуск самого потоку виконання можливо тільки через метод Thread.start().

Робота з пріоритетами

Розглянемо, як в Java можна призначати потокам пріоритети. Для цього в класі Thread існують методи getPriority() і setPriority(), а також оголошено три константи:

```
MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY
```

З назви зрозуміло, що їх значення описують мінімальне, максимальне і нормальне (за умовчанням) значення пріоритету.

Розглянемо наступний приклад:

```
public class ThreadTest implements Runnable {
  public void run() {
    double calc;
    for (int i=0; i<50000; i++) {
      calc=Math.sin(i*i);
      if (i%10000==0) {
        System.out.println(getName()+
          " counts " + i/10000);
      }
    }
  }

  public String getName() {
    return Thread.currentThread().getName();
  }

  public static void main(String s[]){
```

```
    // Підготовка потоків
    Thread t[] = new Thread[3];
    for (int i=0; i<t.length; i++) {
      t[i]=new Thread(new ThreadTest(),
                  "Thread "+i);
    }
    // Запуск потоків
    for (int i=0; i<t.length; i++) {
      t[i].start();
      System.out.println(t[i].getName()+
                      " started");
    }
  }
}
```

У прикладі використовується декілька нових методів класу Thread:

- getName()

Звернете увагу, що конструктору класу Thread передається два параметри. До реалізації Runnable добавляється рядок. Це ім'я потоку, яке використовується тільки для спрощення його ідентифікації. Імена декількох потоків можуть збігатися. Якщо його не задати, то Java генерує простий рядок виду "Thread-"и номер потоку (обчислюється простим лічильником). Саме це ім'я повертається методом getName(). Його можна змінити за допомогою методу setName().

- currentThread()

Цей статичний метод дозволяє в будь-якому місці коду отримати посилання на об'єкт класу Thread, що представляє поточний потік виконання.

Результат роботи такої програми матиме наступний вигляд:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 0 counts 0
Thread 1 counts 0
Thread 2 counts 0
Thread 0 counts 1
Thread 1 counts 1
Thread 2 counts 1
Thread 0 counts 2
Thread 2 counts 2
Thread 1 counts 2
Thread 2 counts 3
Thread 0 counts 3
Thread 1 counts 3
Thread 2 counts 4
```

```
Thread 0 counts 4
Thread 1 counts 4
```

Ми бачимо, що всі три потоки були запущені один за іншим і почали проводити обчислення. Видно також, що потоки виконуються без певного порядку, випадковим чином. Проте, в середньому вони рухаються з однією швидкістю, ніхто не відстає і не наздоганяє.

Введемо в програму роботу з пріоритетами, розставимо різні значення для різних потоків і подивимося, як це позначиться на виконанні. Змінюється тільки метод main().

```
public static void main(String s[]){
// Підготовка потоків
  Thread t[] = new Thread[3];
  for (int i=0; i<t.length; i++) {
    t[i]=new Thread(new ThreadTest(),
                "Thread "+i);
    t[i].setPriority(Thread.MIN_PRIORITY +
    (Thread.MAX_PRIORITY -
    Thread.MIN_PRIORITY)/t.length*i);
  }

// Запуск потоків
  for (int i=0; i<t.length; i++) {
    t[i].start();
    System.out.println(t[i].getName()+
                  " started");
  }
}
```

Формула обчислення пріоритетів дозволяє рівномірно розподілити всі допустимі значення для всіх потоків, що запускаються. Насправді, константа мінімального пріоритету має значення 1, максимального 10, нормального 5. Отже в простих програмах можна явно користуватися цими величинами і указувати як, наприклад, зниженого пріоритету значення 3.

Результатом роботи буде:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 2 counts 0
Thread 2 counts 1
Thread 2 counts 2
Thread 2 counts 3
Thread 2 counts 4
Thread 0 counts 0
Thread 1 counts 0
```

```
Thread 1 counts 1
Thread 1 counts 2
Thread 1 counts 3
Thread 1 counts 4
Thread 0 counts 1
Thread 0 counts 2
Thread 0 counts 3
Thread 0 counts 4
```

Потоки, як і раніше, стартують послідовно. Але потім ми бачимо, що чим вище пріоритет, тим швидше відпрацьовує потік. Проте, вельми показово, що потік з мінімальним пріоритетом (Thread 0) все ж таки дістав можливість виконати одну дію раніше, ніж відпрацював потік з вищим пріоритетом (Thread 1). Це говорить про те, що пріоритети не роблять систему однопоточної, виконуючою одноразово лише один потік з найвищим пріоритетом. Навпаки, пріоритети дозволяють одночасно працювати над декількома завданнями з урахуванням їх важливості.

Якщо збільшити параметри методу (виконувати 500000 обчислень, а не 50000, і виводити повідомлення кожне 1000-е обчислення, а не 10000-і), то можна буде наочно побачити, що всі три потоки мають можливість виконувати свої дії одночасно, просто вищий пріоритет дозволяє виконувати їх гущавині.

Демон-потоки

Демон-потоки дозволяють описувати фонові процеси, які потрібні тільки для обслуговування основних потоків виконання і не можуть існувати без них. Для роботи з цією властивістю існують методи setDaemon() і isDaemon().

Розглянемо наступний приклад:

```
public class ThreadTest implements Runnable {

  // Окрема група, в якій будуть
  // знаходитися всі потоки ThreadTest
  public final static ThreadGroup GROUP = new ThreadGroup("Daemon demo");

  // Стартове значення, указується при створенні об’єкту
  private int start;

  public ThreadTest(int s){
    start = (s%2==0)? s: s+1;
    new Thread(GROUP, this, "Thread "+ start).start();
  }

public void run() {
    // Починаємо зворотний відлік
    for (int i=start; i>0; i--) {
      try {
        Thread.sleep(300);
```

```
        } catch (InterruptedException e) {}
        // Після досягнення середини породжуємо
        // новий потік з половинним початковим
        // значенням
        if (start>2 && i==start/2)
       {
         new ThreadTest(i);
        }
      }
   }

public static void main(String s[]){
   new ThreadTest(16);
   new DaemonDemo();
   }
}
public class DaemonDemo extends Thread {
   public DaemonDemo() {
     super("Daemon demo thread");
     setDaemon(true);
     start();
   }

   public void run() {
     Thread threads[]=new Thread[10];
     while (true) {
       // Отримуємо набір всіх потоків з
       // тестової групи
       int count=ThreadTest.GROUP.activeCount();
       if (threads.length<count) threads = new Thread[count+10];
       count=ThreadTest.GROUP.enumerate(threads);

       // Роздруковуємо ім'я кожного потоку
       for (int i=0; i<count; i++) {
         System.out.print(threads[i].getName()+", ");
       }
       System.out.println();
       try {
         Thread.sleep(300);
       } catch (InterruptedException e) {}
     }
   }
}
```

Приклад 12.1. (html, txt)

В даному прикладі відбувається наступне. Потоки ThreadTest мають деяке стартове значення, що передається їм при створенні. У методі run() це значення послідовно зменшується. Досягши половини від початкової величини породжується новий потік з удвічі меншим початковим значенням. Після вичерпання лічильника потік зупиняється.

Метод main() породжує перший потік із стартовим значенням 16. В ході програми будуть додатково породжені потоки із значеннями 8, 4, 2.

За цим процесом спостерігає демон-поток DaemonDemo. Цей потік регулярно отримує список всіх існуючих потоків ThreadTest і роздруковує їх імена для зручності спостереження.

Результатом програми буде:

```
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16
Thread 16, Thread 8
Thread 16, Thread 8
Thread 16, Thread 8
Thread 16, Thread 8
Thread 16, Thread 8
Thread 16, Thread 8, Thread 4
Thread 16, Thread 8, Thread 4
Thread 8, Thread 4
Thread 4, Thread 2
Thread 2
```

Приклад 12.2. (html, txt)

Не дивлячись на те, що демон-поток ніколи не виходить з методу run(), віртуальна машина припиняє роботу, як тільки всі не-демон-потоки завершуються.

У прикладі використовувалося декілька додаткових класів і методів, які ще не були розглянуті:

- клас ThreadGroup

Всі потоки знаходяться в групах, що представляються екземплярами класу ThreadGroup. Група указується при створенні потоку. Якщо група не була вказана, то потік поміщається в ту ж групу, де знаходиться потік, породжувач його.

Методи activeCount() і enumerate() повертають кількість і повний список, відповідно, всіх потоків в групі.

- sleep()

Цей статичний метод класу Thread припиняє виконання поточного потоку на вказану кількість мілісекунд. Звернете увагу, що метод вимагає обробки

виключення InterruptedException. Він пов'язаний з можливістю активізувати метод, який припинив свою роботу. Наприклад, якщо потік зайнятий виконанням методу sleep(), тобто не діє впродовж вказаного періоду часу, його можна вивести з цього стану, викликавши метод interrupt() з іншого потоку виконання. В результаті метод sleep() урветься виключенням InterruptedException.

Окрім методу sleep(), існує ще один статичний метод yield() без параметрів. Коли потік викликає його, він тимчасово припиняє свою роботу і дозволяє відпрацювати іншим потокам. Один з методів обов'язково повинен застосовуватися усередині нескінченних циклів очікування, інакше є ризик, що такий нічого потік, що не робить, загальмує роботу останніх потоків.

Синхронізація

При багатопотоковій архітектурі додатку можливі ситуації, коли декілька потоків одночасно працюватимуть з одними і тими ж даними, використовуючи їх значення і привласнюючи нові. У такому разі результат роботи програми стає неможливо передбачити, дивлячись тільки на початковий код. Фінальні значення змінних залежатимуть від випадкових чинників, виходячи з того, який потік яку дію встиг зробити першою або останньою.

Розглянемо приклад:

```
public class ThreadTest {

 private int a=1, b=2;
 public void one() {
  a=b;
 }
 public void two() {
  b=a;
 }

 public static void main(String s[]){
  int a11=0, a22=0, a12=0;
  for (int i=0; i<1000; i++) {
   final ThreadTest про = new ThreadTest();

   // Запускаємо перший потік, який
   // викликає один метод
   new Thread() {
      public void run() {
        o.one();
      }
   }.start();

   // Запускаємо другий потік, який
   // викликає другий метод
   new Thread() {
     public void run() {
```

```
            o.two();
          }
        }.start();

        // даємо потокам час відпрацювати
        try {
          Thread.sleep(100);
        } catch (InterruptedException e) {}

        // аналізуємо фінальні значення
        if (o.a==1 && o.b==1) a11++;
        if (o.a==2 && o.b==2) a22++;
        if (o.a!=o.b) a12++;
      }
      System.out.println(a11+" "+a22+" "+a12);
    }
}
```

Приклад 12.3. (html, txt)

В даному прикладі два потоки виконання одночасно звертаються до одного і тому ж об'єкту, викликаючи у нього двох разных методу, one() і two(). Ці методи намагаються прирівняти два поля класу a і b один одному, але в різному порядку. Враховуючи, що початкові значення полів дорівнюють 1 і 2, відповідно, можна було чекати, що після того, як потоки завершать свою роботу, поля матимуть однакове значення. Проте зрозуміти, яке з двох можливих значень вони приймуть, вже неможливо. Подивимося на результат програми:

```
135 864 1
```

Перше число показує, скільки разів з тисячі обидві змінні набули значення 1. Друге число відповідає значенню 2. Таке сильне переважання одне із значень обумовлено послідовністю запусків потоків. Якщо її змінити, то і кількості випадків з 1 і 2 також міняються місцями. Третє ж число повідомляє, що на тисячу випадків відбувся один, коли поля взагалі обмінялися значеннями!

При кількості ітерацій, рівній 10000, були отримані наступні дані, які підтверджують зроблені виводи:

```
494 9498 8
```

А якщо прибрати затримку перед аналізом результатів, то отримувані дані радикально міняються:

```
0 3 997
```

Мабуть, потоки просто не встигають відпрацювати.

Отже, наочно показано, наскільки сильно і непередбачувано може мінятися результат роботи однієї і тієї ж програми, що застосовує багатопотокову архітектуру. Необхідно враховувати, що в приведеному простому прикладі затримки створювалися уручну методом Thread.sleep(). У реальних складних системах затримки можуть виникати в місцях проведення складних операцій, їх довжина непередбачувана і оцінити їх наслідки неможливо.

Для глибшого розуміння принципів багатопотокової роботи в Java розглянемо організацію пам'яті у віртуальній машині для декількох потоків.

Зберігання змінних в пам'яті

Віртуальна машина підтримує основне сховище даних (main storage), в якому зберігаються значення всіх змінних і яке використовується всіма потоками. Під змінними тут розуміються поля об'єктів і класів, а також елементи масивів. Що стосується локальних змінних і параметрів методів, то їх значення не можуть бути доступні іншим потокам, тому вони не представляють інтересу.

Для кожного потоку створюється його власна робоча пам'ять (working memory), в яку перед використанням копіюються значення всіх змінних.

Розглянемо основні операції, доступні для потоків при роботі з пам'яттю:

- use – читання значення змінної з робочої пам'яті потоку;
- assign – запис значення змінної в робочу пам'ять потоку;
- read – набуття значення змінної з основного сховища;
- load – збереження значення змінної, прочитаного з основного сховища, в робочій пам'яті;
- store – передача значення змінній з робочої пам'яті в основне сховище для подальшого зберігання;
- write – зберігає в основному сховищі значення змінної, переданою командою store.

Підкреслимо, що перераховані команди не є методами яких-небудь класів, вони недоступні програмістові. Сама віртуальна машина використовує їх для забезпечення коректної роботи потоків виконання.

Потік, працюючи із змінною, регулярно застосовує команди use і assign для використання її поточного значення і привласнення нового. Крім того, повинні здійснюватися дії з передачі значень в основне сховище і з нього. Вони виконуються в два етапи. При отриманні даних спочатку основне сховище прочитує значення командою read, а потім потік зберігає результат в своїй робочій пам'яті командою load. Ця пара команд завжди виконується разом саме в такому порядку, тобто не можна виконати одну, не виконавши іншу. При відправленні даних спочатку потік прочитує значення з робочої пам'яті командою store, а потім основне сховище зберігає його командою write. Ця пара команд також завжди виконується разом саме в такому порядку, тобто не можна виконати одну, не виконавши іншу.

Набор цих правил складався з тим, щоб операції з пам'яттю були достатньо строгі для точного аналізу їх результатів, а з іншого боку, правила повинні залишати достатній простір для різних технологій оптимізацій (регістри, черги, кеш і так далі).

Послідовність команд підкоряється наступним правилам:

- всі дії, що виконуються одним потоком, строго впорядковані, тобто виконуються одне за іншим;
- всі дії, що виконуються з однією змінною в основному сховищі пам'яті, строго впорядковані, тобто слідують одне за іншим.

За винятком деяких додаткових очевидних правил, більше ніяких обмежень немає. Наприклад, якщо потік змінив значення спочатку одній, а потім іншій змінній, то ці зміни можуть бути передані в основне сховище в зворотному порядку.

Потік створюється з чистою робочою пам'яттю і повинен перед використанням завантажити всі необхідні змінні з основного сховища. Будь-яка змінна спочатку створюється в основному сховищі і лише потім копіюється в робочу пам'ять потоків, які її застосовуватимуть.

Таким чином, потоки ніколи не взаємодіють один з одним безпосередньо, тільки через головне сховище.

Модифікатор volatile

При оголошенні полів об'єктів і класів може бути вказаний модифікатор volatile. Він встановлює строгіші правила роботи із значеннями змінних.

Якщо потік збирається виконати команду use для volatile змінної, то потрібний, щоб попередньою дією з цією змінною було обов'язкове load, і навпаки – операція load може виконуватися тільки передuse. Таким чином, змінна і головне сховище завжди мають найостанніше значення цієї змінної.

Аналогічно, якщо потік збирається виконати команду store для volatile змінної, то потрібний, щоб попередньою дією над цією змінною було обов'язкове assign, і навпаки – операція assign може виконуватися, тільки якщо наступною буде store. Таким чином, змінна і головне сховище завжди мають найостанніше значення цієї змінної.

Нарешті, якщо проводяться операції над декількома volatile змінними, то передача відповідних змін в основне сховище повинна проводитися строго в тому ж порядку.

При роботі із звичайними змінними компілятор має більше простору для маневру. Наприклад, при сприятливих обставинах може виявитися можливим передбачити значення змінної, заздалегідь обчислити і зберегти його, а потім в потрібний момент використовувати вже готовим.

Слід звернути увагу на два 64-розрядні типи, double і long. Оскільки багато платформ підтримують лише 32-бітову пам'ять, величини цих типів розглядаються як дві змінні і всі описані дії виконуються незалежно для двох половинок таких значень. Звичайно, якщо

виробник віртуальної машини вважає за можливе, він може забезпечити атомарність операцій і над цими типами. Для volatile змінних це є обов'язковою вимогою.

Блокування

У основному сховищі для кожного об'єкту підтримується блокування (lock), над яким можна провести дві дії, – встановити (lock) і зняти (unlock). Тільки один потік водночас часу може встановити блокування на деякий об'єкт. Якщо до того, як цей потік виконає операцію unlock, інший потік спробує встановити блокування, його виконання буде припинено до тих пір, поки перший потік не відпустить її.

Операції lock і unlock накладають жорстке обмеження на роботу із змінними в робочій пам'яті потоку. Після успішно виконаного lock робоча пам'ять очищається і всі змінні необхідно наново прочитувати з основного сховища. Аналогічно, перед операцією unlock необхідно всі змінні зберегти в основному сховищі.

Важливо підкреслити, що блокування є чимось на зразок прапора. Якщо блокування на об'єкт встановлене, це не означає, що даним об'єктом не можна користуватися, що його поля і методи стають недоступними, - це не так. Єдина дія, яка стає неможливою, - установка цього ж блокування іншим потоком, до тих пір, поки перший потік не виконає unlock.

У Java-программе для того, щоб скористатися механізмом блокувань, існує ключове словоsynchronized. Воно може бути застосоване в двох варіантах – для оголошення synchronized-блока і як модифікатор методу. У обох випадках дія його приблизно однакова.

Synchronized-блок записується таким чином:

```
synchronized (ref) {
  ...
}
```

Перш, ніж почати виконувати дії, описані в цьому блоці, потік зобов'язаний встановити блокування на об'єкт, на який посилається змінна ref (тому вона не може бути null). Якщо інший потік вже встановив блокування на цей об'єкт, то виконання першого потоку припиняється до тих пір, поки не вдасться виконати операцію lock.

Після цього блок виконується. При завершенні виконання (як успішному, так і у разі помилок) проводиться операція unlock, щоб звільнити об'єкт для інших потоків.

Розглянемо приклад:

```
public class ThreadTest implements Runnable {
  private static ThreadTest
              shared = new ThreadTest();
  public void process() {
    for (int i=0; i<3; i++) {
      System.out.println(
```

```
                Thread.currentThread().
            getName()+" "+i);
          Thread.yield();
        }
      }

      public void run() {
        shared.process();
      }
      public static void main(String s[]){
        for (int i=0; i<3; i++) {
          new Thread(new ThreadTest(),
            "Thread-"+i).start();
        }
      }
    }
```

У цьому простому прикладі три потоки викликають метод у одного об'єкту, щоб той роздрукував три значення. Результатом буде:

```
Thread-0 0
Thread-1 0
Thread-2 0
Thread-0 1
Thread-2 1
Thread-0 2
Thread-1 1
Thread-2 2
Thread-1 2
```

Тобто всі потоки одночасно працюють з одним методом одного об'єкту. Укладемо звернення до методу synchronized-блок:

```
public void run() {
  synchronized (shared) {
    shared.process();
  }
}
```

Тепер результат буде строго впорядкований:

```
Thread-0 0
Thread-0 1
Thread-0 2
Thread-1 0
Thread-1 1
Thread-1 2
```

```
Thread-2 0
Thread-2 1
Thread-2 2
```

Synchronized-методы працюють аналогічним чином. Перш, ніж почати виконувати їх, потік намагається заблокувати об’єкт, у якого викликається метод. Після виконання блокування знімається. У попередньому прикладі аналогічної впорядкованості можна було добитися, якщо використовувати не synchronized-блок, а оголосити метод process() синхронізованим.

Також допустимі методи static synchronized. При їх виклику блокування встановлюється на об’єкт классаClass, що відповідає за тип, у якого викликається цей метод.

При роботі з блокуваннями завжди треба пам’ятати про можливість появи deadlock – взаємних блокувань, які приводять до зависання програми. Якщо один потік заблокував один ресурс і намагається заблокувати другою, а інший потік заблокував другий і намагається заблокувати перший, то такі потоки вже ніколи не вийдуть із стану очікування.

Розглянемо простий приклад:

```
public class DeadlockDemo {

 // Два объекта-ресурса
 public final static Object one=new Object(), two=new Object();

 public static void main(String s[]){

  // Створюємо два потоки, які будуть
  // конкурувати за доступ до об’єктів
 // one і two
  Thread t1 = new Thread() {
   public void run() {
    // Блокування першого об’єкту
    synchronized(one){
    Thread.yield();
     // Блокування другого об’єкту
     synchronized (two) {
      System.out.println("Success!");
     }
    }
   }
  };
  Thread t2 = new Thread() {
   public void run() {
    // Блокування другого об’єкту
    synchronized(two){
     Thread.yield();
```

```
            // Блокування першого об’єкту
            synchronized (one) {
              System.out.println("Success!");
            }
          }
        }
      };

      // Запускаємо потоки
      t1.start();
      t2.start();
    }
}
```

Приклад 12.4. (html, txt)

Якщо запустити таку програму, то вона ніколи не закінчить свою роботу. Звернете увагу на виклики методу yield() у кожному потоці. Вони гарантують, що коли один потік виконав перше блокування і переходить до наступної, другий потік знаходиться в такому ж стані. Очевидно, що в результаті обидва потоки "замруть", не зможуть продовжити своє виконання. Перший потік чекатиме звільнення другого об’єкту, і навпаки. Саме така ситуація називається "Мертвим блокуванням", або deadlock. Якщо один з потоків встиг би заблокувати обидва об’єкти, то програма успішно б виконалася до кінця. Проте багатопотокова архітектура не дає ніяких гарантій, як саме потоки виконуватимуться один щодо одного. Затримки (які в прикладі моделюються викликами yield()) можуть виникати з логіки програми (необхідність провести обчислення), дій користувача (не відразу натиснув кнопку "OK"), зайнятості ОС (із-за браку фізичної оперативної пам’яті довелося скористатися віртуальною), значень пріоритетів потоків і так далі.

У Java немає ніяких засобів розпізнавання або запобігання ситуаціям deadlock. Також немає способу перед викликом синхронізованого методу дізнатися, чи заблокований вже об’єкт іншим потоком. Програміст сам повинен будувати роботу програми так, щоб нерозв’язні блокування не виникали. Наприклад, в розглянутому прикладі досить було організувати блокування об’єктів в одному порядку (завжди спочатку перший, потім другий) – і програма завжди виконувалася б успішно.

Небезпека виникнення взаємних блокувань примушує з особливою увагою відноситися до роботи з потоками. Наприклад, важливо пам’ятати, що якщо у об’єкту потоку був викликаний метод sleep(..), то такий потік не діятиме певний час, але при цьому всі заблоковані ним об’єкти залишатимуться недоступними для блокувань з боку інших потоків, а це потенційний deadlock. Такі ситуації украй складно виявити шляхом тестування і відладки, тому питанням синхронізації треба приділяти багато часу на етапі проектування.

Методи wait(), notify(), notifyAll() класу Object

Нарешті, перейдемо до розгляду трьох методів класу Object, завершуючи опис механізмів підтримки багатопочності в Java.

Кожен об’єкт в Java має не тільки блокування для synchronized блоків і методів, але і так званий wait-set, набір потоків виконання. Будь-який потік може викликати метод wait() будь-якого об’єкту і таким чином потрапити в його wait-set. При цьому виконання такого потоку припиняється до тих пір, поки інший потік не викличе у цього ж об’єкту метод notifyAll(), який будить всі потоки з wait-set. Метод notify() пробуджує один випадково вибраний потік з даного набору.

Проте застосування цих методів пов’язане з одним важливим обмеженням. Будь-який з них може бути викликаний потоком у об’єкту тільки після встановлення блокування на цей об’єкт. Тобто або в середині synchronized-блока з посиланням на цей об’єкт як аргумент, або звернення до методів мають бути в синхронізованих методах класу самого об’єкту. Розглянемо приклад:

```
public class WaitThread implements Runnable {
     private Object shared;

     public WaitThread(Object про){
       shared=o;
}

public void run() {
  synchronized (shared) {
     try {
       shared.wait();
     } catch (InterruptedException e) {}
     System.out.println("after wait");
  }
}

public static void main(String s[]){
  Object про = new Object();
  WaitThread w = new WaitThread(o);
  new Thread(w).start();
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) {}
  System.out.println("before notify");
  synchronized (o) {
    o.notifyAll();
  }
  }
}
```

Результатом програми буде:

```
before notify
after wait
```

Звернете увагу, що метод wait(), як і sleep(), вимагає обробки InterruptedException, тобто його виконання також можна перервати методом interrupt().

На закінчення розглянемо складніший приклад для трьох потоків:

```
public class ThreadTest implements Runnable {
     final static private Object shared=new Object();
     private int type;
     public ThreadTest(int i){
       type=i;
     }

public void run() {
  if (type==1 || type==2) {
     synchronized (shared) {
       try {
         shared.wait();
       } catch (InterruptedException e) {}
       System.out.println("Thread "+type+" after wait()");
  }
} else {
  synchronized (shared) {
     shared.notifyAll();
     System.out.println("Thread "+type+" after notifyAll()");
     }
  }
}

public static void main(String s[]){
  ThreadTest w1 = new ThreadTest(1);
  new Thread(w1).start();
  try {
     Thread.sleep(100);
  } catch (InterruptedException e) {}
  ThreadTest w2 = new ThreadTest(2);
  new Thread(w2).start();
  try {
     Thread.sleep(100);
  } catch (InterruptedException e) {}
  ThreadTest w3 = new ThreadTest(3);
  new Thread(w3).start();
  }
}
```

Приклад 12.5. (html, txt)

Результатом роботи програми буде:

```
Thread 3 after notifyAll()
Thread 1 after wait()
```

Thread 2 after wait()

Приклад 12.6. (html, txt)

Розглянемо, що відбулося. По-перше, був запущений потік 1, який тут же викликав метод wait() і припинив своє виконання. Потім те ж саме відбулося з потоком 2. Далі починає виконуватися потік 3.

Відразу звертає на себе увагу наступний факт. Ще потік 1 увійшов до synchronized-блок, а отже, встановив блокування на об'єкт shared. Але, судячи по результатах, це не перешкодило і потоку 2 зайти в synchronized-блок, а потім і потоку 3. Причому, для останнього це просто необхідно, інакше як можна "розбудити" потоки 1 і 2?

Можна зробити вивід, що потоки, перш ніж припинити виконання після виклику методу wait(), відпускають всі зайняті блокування. Отже, викликається метод notifyAll(). Як вже було сказано, всі потоки з wait-set відновлюють свою роботу. Проте щоб коректно продовжити виконання, необхідно повернути блокування на об'єкт, адже наступна команда також знаходиться усередині synchronized-блока!

Виходить, що навіть після виклику notifyAll() всі потоки не можуть відразу відновити роботу. Лише один з них зможе повернути собі блокування і продовжити роботу. Коли він покине свій synchronized-блок і відпустить об'єкт, другий потік відновить свою роботу, і так далі. Якщо з якоїсь причини об'єкт так і не буде звільнений, потік так ніколи і не вийде з методу wait(), навіть якщо буде викликаний метод notifyAll(). У розглянутому прикладі потоки один за іншим змогли відновити свою роботу.

Крім того, визначений метод wait() з параметром, який задає період тайм-ауту, після закінчення якого потік сам спробує відновити свою роботу. Але почати йому доведеться все одно з повторного отримання блокування.

Висновок

У цій лекції були розглянуті принципи побудови багатопотокового застосування. На початку розбиралися переваги і недоліки такої архітектури – як правило ОС не виділяє окремий процесор під кожен процес, а значить застосовується процедура time slicing. Було виділено три ознаки, які вказують на доцільність запуску декількох потоків в рамках програми.

Основу роботи з потоками в Java складають інтерфейс Runnable і клас Thread. З їх допомогою можна запускати і зупиняти потоки, міняти їх властивості, серед яких основні: пріоритет і властивість daemon. Головна проблема, що виникає в таких програмах, - одночасний доступ декількох потоків до одних і тих же даних, насамперед -– до полів об'єктів. Для розуміння, як в Java вирішується це завдання, був зроблений короткий огляд по організації пам'яті в JVM, роботи із змінними і блокуваннями. Блокування, не дивлячись на назву, самі по собі не обмежують доступ до змінної. Програміст використовує їх через ключове слово synchronized, яке може бути вказане в сигнатурі методу або на початку блоку. В результаті виконання не буде продовжено, поки блокування не звільниться.

Новий механізм породжує нову проблему - взаємні блокування (deadlock), до якої програміст завжди має бути готовий, тим паче, що Java не має вбудованих засобів для визначення такої ситуації. На лекції знався приклад, як організувати роботу програми без "зависання" чекаючих потоків.

На завершення розглядалися спеціалізовані методи базового класу Object, які також дозволяють управляти послідовністю роботи потоків.

ТЕМА. *Пакет java..lang*

План

1. *Основна бібліотека Java - java..lang.*
2. *Призначення і можливості класів пакету java..lang.*

Рекомендована література: 2, 3, 4.

Теоретичні відомості

До складу пакету java.lang входять класи, які складають основу для всіх інших, і тому він є найбільш важливим зі всіх, що входять в Java API. Оскільки без нього не може обійтися жоден клас, кожен модуль компіляції містить неявне імпортування цього пакету (import java.lang.*;).

Перерахуємо класи, які складають основу пакету.

Object – є кореневим в ієрархії класів.

Class – екземпляри цього класу є описами об'єктних типів в пам'яті JVM.

String – є символьним рядком, містить засоби роботи з нею.

StringBuffer – використовується для роботи (створення) рядків.

Number – абстрактний клас, що є суперкласом для классов-объектных обгорток числових примітивних типів Java.

Character – об'єктна обгортка для типу char.

Boolean – об'єктна обгортка для типу boolean.

Math – реалізує набір базових математичних функцій.

Throwable – базовий клас для об'єктів, що представляють виключення. Будь-яке виключення, яке може бути кинуте і, відповідно, перехоплене блоком catch, має бути успадковане від Throwable.

Thread – дозволяє запускати і працювати з потоками виконання в Java.Runnable – може використовуватися у поєднанні з класом Thread для опису потоків виконання.

ThreadGroup – дозволяє об'єднувати потоки в групу і проводити дії відразу над всіма потоками в ній. Існують обмеження по безпеці на маніпуляції з потоками з інших груп.

System – містить корисні поля і методи для роботи системного рівня.

Runtime – дозволяє додатку взаємодіяти з оточенням, в якому воно запущене.

Process – представляє інтерфейс до зовнішньої програми, запущеної за допомогою Runtime.

ClassLoader – відповідає за завантаження опису класів в пам'ять JVM.

SecurityManager – для забезпечення безпеки накладає обмеження на дане середовище виконання програм.

Compiler – використовується для підтримки Just-in-Time компіляторів.

Інтерфейси:

Cloneable – має бути реалізований об'єктами, які планується клонувати за допомогою засобів JVM;

Comparable – дозволяє упорядковувати (сортувати, порівнювати) об'єкти кожного класу, що реалізовує цей інтерфейс.

Object

Клас Object є базовим для всіх останніх класів. Він визначає методи, які підтримуються будь-яким класом в Java.

Метод public final native Class getClass() повертає об'єкт типу Class, відповідний класу об'єкту. Цей метод вже розглядався в лекції 4.

Метод public boolean equals(Object obj) визначає, чи є об'єкти однаковими. Якщо оператор ==провіряють рівність по посиланню (указують на один і той же об'єкт), то метод equals() – рівність за значенням (стани об'єктів однакові). Оскільки клас Object не містить полів, реалізація в нім цього методу така, що значення true буде повернено тільки у разі рівності по посиланню, тобто:

```
public boolean equals(Object obj){
  return (this == obj);
}
```

У класах-спадкоємцях цей метод при необхідності може бути перевизначений, щоб підтримати розширений стан об'єкту (наприклад, якщо додалося поле, що характеризує стан). Розглянемо порівняння обєктів-обгорток цілих чисел (клас Integer). Воно повинне по всій логіці повертати значення true, якщо рівні значення int чисел, які обернуті, навіть якщо це два різні об'єкти.

Метод equals() може бути перевизначений будь-яким способом – компілятор, звичайно ж, не проводитиме аналіз реалізації і не даватиме рекомендації. Проте існують

угоди, які необхідно дотримувати, щоб програма мала передбачену поведінку, у тому числі і з погляду інших програмістів:

- рефлексія: для будь-якого об'єктного посилання x, відмінною від null, виклик x.equals(x) повертає true;
- симетричність: для будь-яких об'єктних посилань x і y, виклик x.equals(y) повертає true тільки в тому випадку, якщо виклик y.equals(x) повертає true;
- транзитивність: для будь-яких об'єктних посилань x, y і z, якщо x.equals(y) повертає true і y.equals(z)повертає true, то виклик x.equals(z) повинен повернути true;
- несуперечність: для будь-яких об'єктних посилань x і y багатократні послідовні виклики x.equals(y)повертають одне і те ж значення (або завжди true, або завжди false);
- для будь-якої не рівною null об'єктного посилання x виклик x.equals(null) повинен повернути значення false.

Приклад:

```
package demo.lang;
public class Rectangle {
  public int sideA;
  public int sideB;
  public Rectangle(int x, int y){
    super();
    sideA = x;
    sideB = y;
  }
  public boolean equals(Object obj){
    if(!(obj instanceof Rectangle))
      return false;
    Rectangle ref = (Rectangle)obj;
    return (((this.sideA==ref.sideA)&&(this.sideB==ref.sideB))||
          (this.sideA==ref.sideB)&&(this.sideB==ref.sideA));
  }
  public static void main(String[] args){
    Rectangle r1 = new Rectangle(10,20);
    Rectangle r2 = new Rectangle(10,10);
    Rectangle r3 = new Rectangle(20,10);
    System.out.println("r1.equals(r1) == " + r1.equals(r1));
    System.out.println("r1.equals(r2) == " + r1.equals(r2));
    System.out.println("r1.equals(r3) == " + r1.equals(r3));
    System.out.println("r2.equals(r3) == " + r2.equals(r3));
    System.out.println("r1.equals(null) == " + r1.equals(null));
  }
}
```

Приклад 13.1. (html, txt)

Запуск цієї програми, очевидно, приведе до виводу на екран наступний:

```
r1.equals(r1) == true
r1.equals(r2) == false
r1.equals(r3) == true
r2.equals(r3) == false
r1.equals(null) == false
```

Приклад 13.2. (html, txt)

В даному прикладі метод equals() у класу Rectangle був перевизначений так, щоб прямокутники були рівні, якщо їх можна накласти один на одного (геометрична рівність).

Більшість стандартних класів перевизначає цей метод, строго слідуючи всім угодам.

Метод public int hashCode() повертає хеш-код (hash code) для об'єкту. Хеш-код – це ціле число, яке зіставляється з даним об'єктом. Воно дозволяє організувати зберігання набору об'єктів з можливістю швидкої вибірки (стандартна реалізація такого механізму присутня в Java і буде описана в наступній лекції).

Для цього методу також прийнятий ряд угод, яким варто слідувати при перевизначенні:

- якщо два об'єкти ідентичні, тобто виклик методу equals(Object) повертає true, то виклик методаhashCode() у кожного з цих двох об'єктів повинен повертати одне і те ж значення;
- під час одного запуску програми для одного об'єкту при виклику методу hashCode() повинне повертатися одне і те ж значення, якщо між цими викликами не торкнулися дані, використовувані для перевірки об'єктів на ідентичність в методі equals(). Це число не обов'язково має бути одним і тим же при повторному запуску тієї ж програми, навіть якщо всі дані будуть ідентичні.

У класі Object цей метод реалізований на рівні JVM. Сама віртуальна машина генерує хеш-код, грунтуючись на розташуванні об'єкту в пам'яті. Це дозволяє для різних об'єктів (нерівність по посиланню) отримувати різні хеш-коды.

Через першу угоду при перевизначенні методу equals() необхідно перевизначити також метод hashCode(). При цьому потрібно прагнути, по-перше, до того, щоб метод повертав значення щонайшвидше, інакше основна мета – швидка вибірка – не буде досягнута. По-друге, бажано для різних об'єктів, тобто коли метод equals(Object) повертає false, генерувати різні хеш-коды. В цьому випадку хеш-таблиці працюватимуть особливо ефективно. Проте, зрозуміло, що це не завжди можливо. Діапазон значений int – 232, а кількість різних рядків, або двовимірних крапок, з координатами типу int – свідомо більше.

Більшість стандартних класів перевизначає цей метод, строго слідуючи всім угодам.

Метод public String toString() повертає рядкове представлення об'єкту. У класі Object цей метод реалізований таким чином:

```
public String toString() {
  return getClass().getName() + "@" +
       Integer.toHexString(hashCode());
```

```
}
```

Тобто повертає рядок, що містить назву класу об'єкту і його хеш-код в шестнадцатеричном форматі.

У классах-наследниках цей метод може бути перевизначений для отримання наочнішого опису об'єкту. Звичайно це значення деяких полів, що характеризують екземпляр. Наприклад, для книги це може бути назва, автор і кількість сторінок:

```
package demo.lang;
public class Book {
  private String title;
  private String author;
  private int pagesNumber;
  public Book(String title, String author
            int pagesNumber) {
    super();
    this.title = title;
    this.author = author;
    this.pagesNumber = pagesNumber;
  }
  public static void main(String[] args){
    Book book = new Book("Java2","Sun",1000);
    System.out.println("object is: " + book);
  }
  public String toString(){
    return "Book: " + title + " ( " + author +
          ", " + pagesNumber + " pages )";
   }
}
```

При запуску цієї програми на екран буде виведено наступне:

```
object is: Book: Java2 ( Sun, 1000 pages )
```

Більшість стандартних класів перевизначає цей метод. Екземпляри класу String повертають посилання на самих себе (this).

Метод wait(), notify(), notifyAll() використовуються для підтримки многопоточности і були детально розглянуті в лекції 12. Вони визначені з атрибутом final і не можуть бути перевизначені в класах-спадкоємцях.

Метод protected void finalize() throws Throwable викликається Java-машиной перед тим, як garbage collector (складальник сміття) звільнить пам'ять, займану об'єктом. Цей метод вже детально розглядався в лекції 4.

Метод protected native Object clone() throws CloneNotSupportedException створює копію об'єкту. Механізм клонування детально розглядався в лекції 9.

Class

У запущеній програмі Java кожному класу відповідає об’єкт типу Class. Цей об’єкт містить інформацію, необхідну для опису класу, – поля, методи і так далі

Клас Class не має відкритого конструктора – об’єкти цього класу створюються автоматично Java-машиной у міру завантаження опису класів з class-файлов. Отримати екземпляр Class для конкретного класу можна за допомогою методу forName():

public static Class forName(String name, boolean initialize, ClassLoader loader) – повертає об’єкт Class, відповідний класу, або інтерфейсу, з назвою, вказаною в name (необхідно указувати повну назву класу або інтерфейсу), використовуючи переданий завантажувач класів. Якщо як завантажувач класів loader передане значениенull, буде узятий ClassLoader, який застосовувався для завантаження зухвалого класу. При цьому клас ініціалізує, тільки якщо значення initialize рівне true і клас не ініціалізував раніше.

Часто простіше і зручніше скористатися методом forName(), передавши тільки назву класу: public static Class forName(String className), – при цьому використовуватиметься завантажувач зухвалого класу і клас ініціалізує (якщо до цього не був).

public Object newInstance() – створює і повертає об’єкт класу, який представляється даним екземпляром Class. Створення відбуватиметься з використанням конструктора без параметрів. Якщо такого в класі немає, буде кинуто виключення InstantiationException. Це ж виключення буде кинуто, якщо об’єкт Class відповідає абстрактному класу, інтерфейсу, або якась інша причина перешкодила створенню нового об’єкту.

Кожному методу, полю, конструктору класу також відповідають об’єкти, список яких можна отримати викликом відповідних методів об’єкту Class: getMethods(), getFields(), getConstructors(),getDeclaredMethods() і так далі В результаті будуть отримані об’єкти, які відповідають за поля, методи, конструктори об’єкту. Їх можна використовувати для формування динамічних викликів Java – цей механізм називається reflection. Необхідні класи містяться в пакеті java.lang.reflection.

Розглянемо приклад використання цієї технології:

```
package demo.lang;
interface Vehicle {
  void go();
}
class Automobile implements Vehicle {
  public void go() {
    System.out.println("Automobile go!");
  }
}
class Truck implements Vehicle {
  public Truck(int i){
    super();
  }
  public void go() {
    System.out.println("Truck go!");
  }
```

```
}
public class VehicleStarter {
  public static void main(String[] args){
    Vehicle vehicle;
    String[] vehicleNames = {"demo.lang.Automobile",
      "demo.lang.Truck", "demo.lang.Tank"};
    for(int i=0; i<vehicleNames.length; i++){
      try {
        String name = vehicleNames[i];
        System.out.println("look for class for: " + name);
        Class aClass = Class.forName(name);
        System.out.println("creating vehicle...");
        vehicle = (Vehicle)aClass.newInstance();
        System.out.println("create vehicle: " + vehicle.getClass());
        vehicle.go();
      } catch(ClassNotFoundException e){
        System.out.println("Exception: " + e);
      } catch(InstantiationException e){
        System.out.println("Exception: " + e);
      }
    }
  }
}
```

Приклад 13.3. (html, txt)

Якщо запустити цю програму, на екран буде виведено наступне:

```
look for class for: demo.lang.Automobile
creating vehicle...
create vehicle: class demo.lang.Automobile
Automobile go!
look for class for: demo.lang.Truck
creating vehicle...
Instantiation exception: java.lang.InstantiationException
look for class for: demo.lang.Tank
Class not found: java.lang.ClassNotFoundException: demo.lang.Tank
```

Приклад 13.4. (html, txt)

В даному прикладі робиться спроба створити за допомогою reflection три об'єкти. Імена класів, від яких вони мають бути породжені, записані в масив vehicleNames. Об'єкт класу Automobile був успішно створений, причому, подальша робота з ним велася через інтерфейс Vehicle. Клас Truck був знайдений, але при спробі створення об'єкту було кинуто, а потім оброблено виключення java.lang.InstantiationException, оскільки конструктор без параметрів відсутній. Клас java.lang.Tank визначений не був і тому при спробі отримати відповідний йому об'єкт Class було викинуто виключення java.lang.ClassNotFoundException.

Класи-обертки

У багатьох випадках краще працювати саме з об'єктами, а не з примітивними типами. Так, наприклад, при використанні колекцій просто необхідно значення примітивних типів представляти у вигляді об'єктів.

Для цих цілей і призначені так звані класи-обертки. Для кожного примітивного типу Java існує свій клас-обертка. Такий клас є незмінним (якщо необхідний об'єкт, що зберігає інше значення, його потрібно створити наново), до того ж має атрибут final – від нього не можна успадковувати клас. Всі класи-обертки (окрім Void) реалізують інтерфейс Serializable, тому об'єкти будь-якого (окрім Void) класу-обгортки можуть бути серіалізовані. Всі класи-обертки містять статичне поле TYPE, що посилається на объектClass, відповідний примітивному типу, що обертається.

Також класи-обертки містять статичні методи для забезпечення зручного маніпулювання відповідними примітивними типами, наприклад, перетворення до рядкового вигляду.

У таблиці 13.1 приведені примітивні типи і відповідні їм класи-обертки.

Таблиця 13.1. Примітивні типи і відповідні їм класи-обертки.

Клас -обертка	**Примітивний тип**
Byte	byte
Short	short
Character	char
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean

При цьому класи-обертки числових типів Byte, Short, Integer, Long, Float, Double успадковуються від одного класу – Number. У нім оголошені методи, що повертають числове значення у всіх числових форматах Java (byte,short, int, long, float і double).

Всі класи-обертки реалізують інтерфейс Comparable. Всі класи-обертки числових типів мають методequals(Object), що порівнює примітивні значення об'єктів.

Розглянемо детальніше деякі з класів-обертків.

Integer

Найбільш часто використовувані статичні методи:

- public static int parseInt(String s) – перетворить рядок, що представляє десятковий запис цілого числа, в int;
- public static int parseInt(String s, int radix) – перетворить рядок, що представляє запис цілого числа в системі числення radix, в int.

Обидва методи можуть порушувати виключення NumberFormatException, якщо рядок, переданий на вхід, містить нецифрові символи.

Не слід плутати ці методи з іншою парою схожих методів:

```
public static Integer valueOf(String s)
public static Integer valueOf(String s
                              int radix)
```

Дані методи виконують аналогічну роботу, тільки результат представляють у вигляді обєкту-обертки.

Існує також два конструктори для створення екземплярів класу Integer:

- Integer(String s) – конструктор, що приймає як параметр рядок, що представляє числове значення.
- Integer(int i) – конструктор, що набуває числового значення.

public static String toString(int i) – використовується для перетворення значення типу int в рядок.

Далі перераховані методи, перетворюючі int в рядкове вісімкове, двійкове і шістнадцядкове уявлення:

- public static String toOctalString(int i) – вісімкове;
- public static String toBinaryString(int i) – двійкове;
- public static String toHexString(int i) – шістнадцядкове.

Є також дві статичні константи:

- Integer.MIN_VALUE – мінімальне int значення;
- Integer.MAX_VALUE – максимальне int значення.

Аналогічні константи, що описують межі відповідних типів, визначені і для всіх останніх класів-обертків числових примітивних типів.

public int intValue() повертає значення примітивного типу для даного об’єкту Integer. Класи-обгорткостальних примітивних цілочисельних типів –

Byte, Short, Long – містять аналогічні методи і константи (визначені для відповідних типів: byte, short, long).

Розглянемо приклад:

```
public static void main(String[] args){
  int i = 1;
  byte b = 1;
  String value = "1000";
  Integer iObj = new Integer(i);
  Byte bObj = new Byte(b);
  System.out.println("while i==b is " +
                (i==b));
  System.out.println("iObj.equals(bObj) is "
                + iObj.equals(bObj));
  Long lObj = new Long(value);
  System.out.println("lObj = " +
                lObj.toString());
  Long sum = new Long(lObj.longValue() +
                iObj.byteValue() +
                bObj.shortValue());
  System.out.println("The sum = " +
                sum.doubleValue());
}
```

У даному прикладі довільним чином використовуються різні варіанти класів-обертків і їх методів. В результаті виконання на екран буде виведено наступне:

```
while i==b is true
iObj.equals(bObj) is false
lObj = 1000
The sum = 1002.0
```

Класи-обертки числових типів Float і Double, що залишилися , окрім описаного для цілочисельних примітивних типів, додатково містять визначення наступних констант (вони детально знялися на лекції 4):

- NEGATIVE_INFINITY – негативна нескінченність;
- POSITIVE_INFINITY – позитивна нескінченність;
- NAN – нечислове значення.

Крім того, інший сенс має значення MIN_VALUE – замість найменшого значення воно представляє мінімальне позитивне (строго > 0) значення, яке може бути представлене цим примітивним типом.

Окрім класів-обертків для примітивних числових типів, такі визначені і для останніх примітивних типів Java.

Character

Реалізує інтерфейси Comparable і Serializable.

З конструкторів має тільки один, приймаючий char як параметр.

Окрім стандартних методів equals(), hashCode(), toString(), містить тільки два нестатичні методи:

- public char charValue() – повертає обернуте значення char;
- public int compareTo(Character anotherCharacter) – порівнює обернуті значення char як числа, тобто повертає значення return this.value – anotherCharacter.value.

Також для сумісності з інтерфейсом Comparable метод compareTo() визначений з параметром Object:

- public int compareTo(Object про) – якщо переданий об'єкт має тип Character, результат буде аналогічний виклику compareTo((Character)o), інакше буде кинуто виключення ClassCastException, так какCharacter можна порівнювати тільки з Character.

Статичні методів в класі Character досить багато, але всі вони прості і логіка їх роботи зрозуміла з назви. Більшість з них - це методи, приймаючі char і перевіряючі всілякі властивості. Наприклад:

```
public static boolean isDigit(char з)
// перевіряє, чи є char цифрою.
```

Ці методи повертають значення істина або брехня, відповідно до того, чи виконаний критерій перевірки.

Boolean

Представляє клас-обертку для примітивного типу boolean.

Реалізує інтерфейс java.io.Serializable і у всьому нагадує аналогічні класи-обертки.

Для набуття примітивного значення використовується метод booleanValue().

Void

Цей клас-обгортка, на відміну від останніх, не реалізує інтерфейс java.io.Serializable. Він не має відкритого конструктора. Більш того, екземпляр цього класу взагалі не може бути отриманий. Він потрібний тільки для отримання посилання на об'єкт Class, відповідний void. Це посилання представлене статичною константою TYPE.

Роблячи короткий висновок по класах-обертках, можна сказати, що:

- кожен примітивний тип має відповідний клас-обертку;
- всі класи-обертки можуть бути сконструйовані як з використанням примітивних типів, так і з використанням String, за винятком Character, який може бути сконструйований тільки по char;
- *класи-обертки можуть порівнюватися з використанням методу equals();*
- примітивні типи можуть витягувати з класів-обертків за допомогою відповідного метода xxxxValue() (наприклад intValue());
- *класи-обертки також є класами-утилітами, тобто надають набір статичних методів для роботи з примітивними типами;*
- *класи-обертки є незмінними.*

Math

Клас Math складається з набору статичних методів, що проводять найбільш популярні математичні обчислення, і двох констант, що мають особливе значення в математиці, – це число Пі і підстава натурального логарифма. Часто цей клас ще називають класом-утилітою (Utility class). Оскільки всі методи класу статичні, немає необхідності створювати екземпляр даного класу, тому він і не має відкритого конструктора. Не можна також і успадковуватися від цього класу, оскільки він оголошений з модифікатором final.

Отже, константи визначені таким чином:

- public static final double Math.PI – задає число ? ("пі");
- public static final double Math.E – підстава натурального логарифма.

У таблиці 13.2 приведені всі методи класу і даний їх короткий опис.

Таблиця 13.2. Методи класу Math і їх короткий опис.

Значення, що повертається	**Ім’я методу і параметри**	**Опис**
.	abs(. a)	абсолютне значення (модуль) для типів double, float, int, long
double	acos(double a)	арккосинус
double	asin(double a)	арксинус
double	atan(double a)	арктангенс
double	ceil(double a)	найменше ціле число, більше а
double	floor(double a)	ціле число, менше а
double	IEEEremainder	залишок за стандартом IEEE 754

	(double a, double b)	(детально розглядався в лекції 3)
double	sin(double a)	синус (тут і далі: аргумент має бути в радіанах)
double	cos(double a)	косинус
double	tan(double a)	тангенс
double	exp(double a)	e в степені a
double	log(double a)	натуральний логарифм a
.	max(. a, . b)	більше з двох чисел (для типів double, float, long, int)
.	min(. a, . b)	менше з двох чисел (для типів double, float, long, int)
double	pow(double a, double b)	a в степені b
double	random()	випадкове число від 0.0 до 1.0
double	rint(double a)	значення int, найближче до a
.	round(. a)	значення long для double (int для float), найближче до a
double	sqrt(double a)	квадратний корінь числа a
double	toDegrees(double a)	перетворення з радіанів в градуси
double	toRadians(double a)	перетворення з градусів в радіани

Рядки

String

Цей клас використовується в Java для представлення рядків. Він володіє властивістю незмінності. Після того, як створений екземпляр цього класу, його вміст вже не може бути модифіковане. Існує багато способів створити об’єкт String. Найбільш простій, якщо вміст рядка відомий на етапі компіляції, – написати текст в лапках:

```
String abc = "abc";
```

Можна використовувати і різні варіанти конструктора. Найбільш простій з них – конструктор, одержуючий на вході рядковий літерал.

```
String s = new String("immutable");
```

На перший погляд, ці варіанти створення рядків відрізняються тільки синтаксисом. На самій же справі відмінність є, хоча в більшості випадків воно неістотне. Розглянемо приклад:

```
public class Test {
  public Test() {
  }

  public static void main(String[] args){
    Test t = new Test();
    String s1 = "Hello world !!!";
    String s2 = "Hello world !!!";
    System.out.println("String‘s equally = " +
                    (s1.equals(s2)));
    System.out.println(
       "Strings are the same = " + (s1==s2));
  }
}
```

В результаті на консоль буде виведено:

```
String‘s equally = true
Strings are the same = true
```

Тепер декілька модифікуємо код:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    String s1 = "Hello world !!!";
    String s2 = new String("Hello world !!!");
    System.out.println("String‘s equally = " +
                    (s1.equals(s2)));
    System.out.println(
       "Strings are the same = " + (s1==s2));
  }
}
```

В результаті на консоль буде виведено:

```
String‘s equally = true
Strings are the same = false
```

Чому результат змінився? Річ у тому, що створення нового об'єкту – це одна з самих трудомістких процедур в Java. Тому компілятор прагне зменшити їх кількість, якщо це не приводить до непередбачуваної поведінки програми.

У прикладі оголошуються дві змінних, які ініціалізувалися однаковим значенням. Оскільки классString незмінний, їх значення завжди будуть однаковими. Це дозволяє компілятору завести приховану допоміжну текстову змінну, яка зберігатиме таке значення, а всі останні змінні посилатимуться на нього ж, а не породжуватимуть нові об'єкти. В результаті в першому варіанті програми створюється лише один об'єкт String. Для більшості операцій це неістотна різниця. Виняток становлять дії, які прив'язані до конкретного об'єкту, а не до його значення. Це метод equals, методи wait/notify.

У другому варіанті вказано динамічне звернення до конструктора. В цьому випадку компілятор вже не має можливості займатися оптимізацією і JVM під час виконання програми дійсно створить другий об'єкт з точно таким же значенням. Що ми і бачимо по результату виконання прикладу.

У Java для рядків визначений оператор +. При використанні цього оператора проводиться конкатенація рядків. У класі String також визначений метод:

```
public String concat(String s);
```

Він повертає новий объект-строку, доповнений справа рядком s.

Розглянемо інший приклад.

```
public class Test {
  public static void main(String[] args){
    Test t = new Test();
    String s = " prefix !";
    System.out.println(s);
    s = s.trim();
    System.out.println(s);
    s = s.concat(" suffix");
    System.out.println(s);
  }
}
```

```
prefix !
prefix !
prefix ! suffix
```

В даному випадку може скластися враження, що рядок (об'єкт String, на який посилається змінна s), можна змінювати. Насправді це не так. В результаті виконання методів trim (відсікання пропусків на початку і кінці рядка) і concat створюються нові обєкти-строки і посилання s починає указувати на новий обєкт-рядок. Таким чином, міняється значення посилання, об'єкти ж незмінні.

Як вже наголошувалося, рядок складається з двобайтових Unicode-символов. Проте у багатьох випадках потрібно працювати з рядком як з набором байт (ввод/вывод, робота з

базою даних і так далі). Перетворення рядка в послідовність байтів проводиться наступними методами:

- byte[] getBytes() – повертає послідовність байтів в кодуванні, прийнятому за умовчанням (як правило, залежить від налаштувань операційної системи);
- byte[] getBytes(String encoding) – повертає послідовність байтів у вказаній кодировкеencoding.

Для виконання зворотної операції (перетворення байтів в рядок) необхідно сконструювати новий объект-строку за допомогою наступних методів:

- String(byte[] bytes) – створює рядок з послідовності байтів в кодуванні, прийнятому за умовчанням;
- String(byte[] bytes, String enc) – створює рядок з послідовності байтів у вказаному кодуванні.

StringBuffer

Цей клас використовується для створення і модифікації рядкових виразів, які після можна перетворити вString. Він реалізований на основі масиву char[], що дозволяє, на відміну від String, модифікувати його значення після створення об’єкту.

Розглянемо найбільш часто використовувані конструктори класу StringBuffer:

- StringBuffer() – створює порожній StringBuffer;
- StringBuffer(String s) – буфер заповнюється вказаним значенням s;
- StringBuffer(int capacity) – створює екземпляр класу StringBuffer з вказаним розміром (длинachar[]). Завдання розміру не означає, що не можна буде оперувати рядками з більшою довгою, чим вказано в конструкторі. Насправді цим гарантується, що при роботі з рядками меншої довжини додаткове виділення пам’яті не буде потрібно.

Різниця між String і StringBuffer може бути продемонстрована на наступному прикладі:

```
public class Test {
  public static void main(String[] args){
    Test t = new Test();
    String s = new String("ssssss");
    StringBuffer sb =
      new StringBuffer("bbbbbb");
    s.concat("-aaa");
    sb.append("-aaa");
    System.out.println(s);
    System.out.println(sb);
  }
}
```

В результаті на екран буде виведено наступне:

```
ssssss
bbbbbb-aaa
```

У даному прикладі можна відмітити, що об'єкт String залишився незмінним, а об'єкт StringBuffer змінився.

Основні методи, використовувані для модифікації StringBuffer, це:

- public StringBuffer append(String str) – додає переданий рядок str в буфер;
- public StringBuffer insert(int offset, String str) – вставка рядка, починаючи з позиції offset(пропустивши offset символів).

Варто звернути увагу, що обидва методи мають варіанти, що приймають як параметри різні примітивні типи Java замість String. При використанні цих методів аргумент заздалегідь приводиться до рядка (за допомогою String.valueOf()).

Ще один важливий момент, пов'язаний з цими методами, – вони повертають сам об'єкт, у якого викликаються. Завдяки цьому, можливе їх використання в ланцюжку. Наприклад:

```
public static void main(String[] args){
StringBuffer sb = new StringBuffer("abc");
String str = sb.append("e").insert(4
          "f").insert(3,"d").toString();
System.out.println(str);
}
```

В результаті на екран буде виведено:

```
abcdef
```

При передачі екземпляра класу StringBuffer як параметр методу слід пам'ятати, що цей клас змінний:

```
public class Test {
  public static void main(String[] args){
    Test t = new Test();
    StringBuffer sb = new StringBuffer("aaa");
    System.out.println("Before = " + sb);
    t.doTest(sb);
    System.out.println("After = " + sb);
  }
  void doTest(StringBuffer theSb){
    theSb.append("-bbb");
  }
}
```

В результаті на екран буде виведено наступне:

```
Before = aaa
```

```
After = aaa-bbb
```

Оскільки всі об’єкти передаються по посиланню, в методі doTest, при виконанні операцій з theSB, буде модифікований об’єкт, на який посилається sb.

Системні класи

Наступні класи, які будуть розглянуті, забезпечують взаємодію з внутрішніми механізмами JVM і середовищем виконання додатку:

- ClassLoader – завантажувач класів; відповідає за завантаження опису класів в пам’ять JVM;
- SecurityManager – менеджер безпеки; містить різні методи перевірки допустимості запрошуваної операції;
- System – містить набір корисних статичних полів і методів;
- Runtime – дозволяє додатку взаємодіяти з середовищем виконання;
- Process – представляє інтерфейс для взаємодії із зовнішньою програмою, запущеною за допомогою Runtime.

ClassLoader

Це абстрактний клас, відповідальний за завантаження типів. По імені класу або інтерфейсу він знаходить і завантажує в пам’ять дані, які складають визначення типу. Зазвичай для цього використовується просте правило: назва типу перетвориться в назву class-файла, з якого і прочитується вся необхідна інформація.

Кожен об’єкт Class містить посилання на об’єкт ClassLoader, за допомогою якого він був завантажений. Для додавання альтернативного способу завантаження класів можна реалізувати свій завантажувач, успадкувавши його від ClassLoader. Наприклад, опис класу може завантажуватися через мережеве з’єднання. Методdefineclass() перетворить масив байт в екземпляр класу Class. За допомогою методу newInstance() можуть бути отримані екземпляри такого класу. В результаті завантажений клас стає повноцінною частиною виконуваного Java-приложения.

Для ілюстрації приведемо приклад, як може виглядати проста реалізація завантажувача класів, що використовує мережеве з’єднання:

```
class NetworkClassLoader extends ClassLoader {
  String host;
  int port;
  public NetworkClassLoader(String host
                            int port) {
    this.host = host;
    this.port = port;
  }
  public Class findClass(String className){
    byte[] bytes = loadClassData(className);
    return defineClass(className, bytes, 0
                       bytes.length);
  }
```

```
  private byte[] loadClassData(
                    String className) {
    byte[] result = null;
    // open connection, load the class data
    return result;
  }
}
```

В даному прикладі тільки показано, що спадкоємець завантажувача класів повинен визначити і реалізувати методи findClass() і loadClassData() для завантаження опису класу. Коли опис отриманий, масив байт передається в метод defineClass() для створення екземпляра Class. Для простоти в прикладі приведений тільки шаблонний код, без реалізації отримання байт з мережевого з'єднання.

Для отримання екземплярів класів, завантажених за допомогою цього завантажувача, можна скористатися методом loadClass():

```
try {
  ClassLoader loader =
     new NetworkClassLoader(host, port);
  Object main = loader.loadClass(
              "Main").newInstance();
  } catch(ClassNotFoundException e){
    e.printStackTrace();
  } catch(InstantiationException e){
    e.printStackTrace();
  } catch(IllegalAccessException e){
    e.printStackTrace();
  }
```

Якщо такий клас не буде знайдений, буде кинуто виключення ClassNotFoundException, якщо клас буде знайдений, але відбудеться яка-небудь помилка при створенні об'єкту цього класу – буде кинуто виключення InstantiationException, і, нарешті, якщо у зухвалого потоку немає відповідних має рацію для створення екземплярів цього класу (що перевіряється менеджером безпеки), буде кинуто виключення IllegalAccessException.

SecurityManager – менеджер безпеки

За допомогою методів цього класу додатку перед виконанням потенційно небезпечних операцій перевіряють, чи є операція допустимою в даному контексті.

Клас SecurityManager містить багато методів з іменами, що починаються з приставки check ("перевірити"). Ці методи викликаються із стандартних класів бібліотек Java перед тим, як в них будуть виконані потенційно небезпечні операції. Типовий виклик виглядає приблизно таким чином:

```
SecurityManager security =
      System.getSecurityManager();
if(security != null){
```

```
    security.checkX(.);
}
```

де X – назва потенційна небезпечній
операції: Access, Read, Write, Connect, Delete, Exec, Listen і так далі

Запобігання виклику проводиться шляхом кидання виключення – SecurityException, якщо виклик операції не дозволений (окрім методу checkTopLevelWindow, який повертає boolean значення). Для установки менеджера безпеки як поточний викликається метод setSecurityManager() у класceSystem. Відповідно, для його отримання потрібно викликати метод getSecurityManager(). В більшості випадків, якщо додаток запускається локально, будуть дозволені всі дії, оскільки в системі SecurityManager відсутній. Передбачається, що що запускається локально додатку можна повністю довіряти. Якщо ж додаток може бути небезпечне (наприклад, його код був завантажений з мережі, як це відбувається у разі аплетів), то менеджер безпеки виставляється і його вже не можна прибрати або замінити (спроби викличуть SecurityException). Він контролює роботу з локальною файловою системою, мережевими з'єднаннями, потоками виконання і так далі

System

Клас System містить набір корисних статичних методів і полів. Екземпляр цього класу не може бути створений або отриманий. Мабуть, найбільш широко використовуваною можливістю, System, що надається , є стандартний вивід, доступний через змінну System.out. Її тип – PrintStream (потоки даних детально розглядатимуться в лекції 15). Стандартний вивід можна перенаправити в інший потік (файл, масив байт і так далі, головне, щоб це був об'єкт PrintStream):

```
public static void main(String[] args){
  System.out.println("Study Java");
  try {
    PrintStream print = new PrintStream(new
      FileOutputStream("d:\\file2.txt"));
    System.setOut(print);
    System.out.println("Study well");
  } catch(FileNotFoundException e){
    e.printStackTrace();
  }
}
```

При запуску цього коду на екран буде виведено тільки

```
Study Java
```

І у файл "d:\file2.txt" буде записано

```
Study well
```

Аналогічно можуть бути перенаправлені стандартне введення System.in – викликом System.setIn(InputStream) і потік виведення повідомлень про

помилки System.err – викликом System.setErr(PrintStream) (за умовчанням всі потоки – in, out, err – працюють з консоллю додатку).

Наступні методи класу System дозволяють працювати з деякими параметрами системи:

- public static void runFinalizersOnExit(boolean value) – визначає, чи проводитиметься виклик методу finalize() у всіх об'єктів (у кого ще не викликався), коли виконання програми буде закінчено (за умовчанням виставлено значення false);
- public static native long currentTimeMillis() – повертає поточний час; цей час представляється як кількість мілісекунд, що пройшли з 1 січня 1970 року;
- public static String getProperty(String key) – повертає значення властивості з ім'ям key.

Щоб набути всіх властивостей, визначених в системі, можна скористатися наступним методом:

- public static java.util.Properties getProperties() – повертає об'єкт java.util.Properties, у якому містяться значення всіх певних системних властивостей.

Метод arrayCopy(Object source, int srcPos, Object target, int trgPos, int length) надає можливість швидкого копіювання вмісту одного масиву в іншій. Перший параметр задає початковий масив, другий, – номер позиції, починаючи з якого брати елементи для копіювання. Третій параметр – массив-"получатель", четвертий, – номер позиції в нім, починаючи з якого записуватимуться скопійовані елементи. Нарешті, останній параметр задає кількість елементів, які треба скопіювати. Обидва масиви мають бути створені, мати сумісні типи і достатню довжину, інакше згенерують відповідні виключення.

Runtime

Під час виконання додатку Java зіставляється екземпляр класу Runtime. Цей об'єкт дозволяє взаємодіяти з оточенням, в якому запущена Java-программа. Отримати його можна за допомогою статичного методу Runtime.getRuntime().

Об'єкт цього класу:

- public void exit(int status) – здійснює завершення програми з кодом завершення status (при використанні цього методу особливу увагу потрібно приділити обробці виключень – вихід буде здійснений вмить і в конструкціях try-catch-finally управління в finally передане не буде);
- public native void gc() – сигналізує складальникові сміття про необхідність запуску;
- public void runFinalization() – проводить запуск виконання методів finalize() у всіх об'єктів, цього чекаючих;
- public native long freeMemory() – повертає кількість вільної пам'яті, доступної додатку JVM. В деяких випадках ця кількість може бути збільшене, якщо викликати у об'єкту Runtime метод gc();

- public native long totalMemory() – повертає сумарну кількість пам'яті, виділену Java-машині. Ця кількість може змінюватися навіть протягом одного запуску, що залежить від реалізації платформи, на якій запущена Java-машина. Також не варто закладатися на об'єм пам'яті, займаної одним певним об'єктом, – ця величина теж залежить від реалізації Java-машины;
- public void loadLibrary(String libname) – завантажує бібліотеку з вказаним ім'ям.

Зазвичай завантаження бібліотек проводиться таким чином: у класі, що використовує native реалізації методів, додається статична ініціалізація, наприклад:

```
static { System.loadLibrary("LibFile"); }
```

Таким чином, коли клас буде завантажений і ініціалізував, необхідний код для реалізації native методів також буде завантажений. Якщо буде проведено декілька викликів завантаження бібліотеки з одним і тим же ім'ям, проведений буде тільки перший, а всі останні будуть проігноровані.

- public void load(String filename) – підвантажує файл з вказаною назвою як бібліотека. В принципі, цей метод працює так само, як і метод loadLibrary(), тільки приймає як параметр саме назву файлу, а не бібліотеки, тим самим дозволяючи завантажити будь-який файл з native кодом;
- public Process exec(String command) – в окремому процесі запускає команду, представлену переданим рядком. Об'єкт Process , що повертається, може бути використаний для взаємодії з цим процесом.

Process

Об'єкти цього класу виходять викликом методу exec() у об'єкту Runtime, що запускає окремий процес. Об'єкт класу Process може використовуватися для управління процесом і отримання інформації про нього.

Process – абстрактний клас, що визначає, які методи мають бути присутніми в реалізаціях для конкретних платформ. Методи класу Process:

- public InputStream getInputStream() – дає можливість отримувати потік введення процесу;
- getErrorStream(), getOutputStream() – методи, аналогічні getInputStream(), але одержуючі, відповідно, стандартні потоки повідомлень про помилки і виводу;
- public void destroy() – знищує процес; всі підпроцеси, запущені з нього, також будуть знищені;
- public int exitValue() – повертає код завершення процесу; за угодою, код завершення, рівний 0, означає нормальне завершення;
- public int waitFor() – вимушує поточний потік виконання припинитися до тих пір, поки не буде завершений процес, представлений цим екземпляром Process; повертає значення коду завершення процесу.

Навіть якщо в додатку Java не буде жодного посилання на об'єкт Process, процес не буде знищений і продовжуватиме асинхронно виконуватися до свого завершення. Специфікацією не обмовляється механізм, за допомогою якого виділятиметься

процесорний час на виконання процесів Process і потоків Java. Тому при проектуванні програм не варто вважатися ні на якій з них, оскільки різні Java-машини можуть демонструвати різну поведінку.

Потоки виконання

Багатопотокова архітектура в Java була детально розглянута в лекції 12. Зупинимося детальніше на методах вживаних класів.

Runnable

Runnable – це інтерфейс, що містить один-єдиний метод без параметрів: run().

Thread

Об'єкти цього класу представляють можливість запускати і управляти потоками виконання.

Отже, для управління потоками в класі Thread передбачені наступні методи:

- public void start() – проводить запуск нового потоку;
- public final void join() – якщо потік А викликає цей метод у об'єкта Thread, що представляє потік В (threadB.join()), то виконання потоку Априостанавливается до тих пір, поки не закінчить виконання потік В;
- public static void yield() – потік, з якого викликаний цей метод, тимчасово припиняється, щоб дати можливість виконуватися іншим потокам;
- public static void sleep(long millis) – потік, з якого викликаний цей метод, перейде в стан "сну" на вказану кількість мілісекунд, після чого зможе продовжити виконання. При цьому потрібно врахувати, що через час millis мілісекунд цьому потоку може бути виділене процесорний час, а може, йому доведеться і почекати трохи довше. Можна сказати, що потік продовжить виконання не раніше, ніж через час millis мілісекунд.

Існує ще декілька методів, які оголошені deprecated і рекомендується їх уникати. Це: suspend() – тимчасово припинити виконання, resume() – продовжити виконання (припинене викликом suspend()), stop() – зупинити виконання потоку.

При виклику методу stop() у потоці, який представляє цей об'єкт Thread, буде кинута опомилка ThreadDeath. Цей клас успадкований від Error. Якщо помилка не буде оброблена в програмі і, відповідно, відбудеться припинення роботи потоку, повідомлення про ненормальне завершення виведене не буде, оскільки таке завершення розглядається як нормальне. Якщо ж в програмі ця помилка обробляється (наприклад, для проведення якихось додаткових дій перед закриттям потоку), то дуже важливо поклопотатися про те, щоб ця ж помилка була кинута далі, щоб потік дійсно закінчив своє виконання. Клас ThreadDeath спеціально успадкований від Error, а не від Exception, оскільки дуже часто використовується перехоплення всіх виключень класу Exception, що не дозволить коректно зупинити потік.

Також Thread дозволяє виставляти такі властивості потоку, як:

- Name – значення типу String, яке можна використовувати для наочнішого поводження з потоками в групі;
- Daemon – виконання програми не буде припинено до тих пір, поки виконується не хоч би один daemon потік;
- Priority – визначає пріоритет потоку. У класі Thread визначені константи, задаючі мінімальне і максимальне значення для пріоритетів потоку, - MIN_PRIORITY і MAX_PRIORITY, а також значення пріоритету за умовчанням – NORM_PRIORITY.

Ці властивості можуть бути змінені тільки до того моменту, коли потік буде запущений, тобто викликаний метод start() об'єкту Thread.

Набути цих значень можна, звичайно ж, у будь-який момент життя потоку – і після його запуску, і після припинення виконання. Також можна дізнатися, в якому стані зараз знаходиться потік: викликом методові sAlive() – чи виконується ще, isInterrupted() – чи перерваний.

ThreadGroup

Для того, щоб окремий потік не міг почати зупиняти і переривати всі потоки підряд, введено поняття групи. Потік може робити вплив тільки на потоки, які знаходяться в одній з ним групі. Групу потоків представляє клас ThreadGroup. Така організація дозволяє захистити потоки від небажаної зовнішньої дії. Група потоків може містити інші групи, що дозволяє організувати всі потоки і групи в ієрархічне дерево, в якому кожен об'єкт ThreadGroup, за винятком кореневого, має батька.

Клас ThreadGroup володіє методами для зміни властивостей всіх вхідних в нього потоків, таких, як приоритет, daemon і так далі Метод list() дозволяє отримати список потоків.

Виключення

Детально механізм використання виключень описаний в лекції 10. Тут зупинимося тільки на тому, що базовим класом для всіх виключень є клас Throwable. Будь-який клас, який планується використовувати як виняток, повинен явним або неявним чином успадковуватися від нього. Клас Throwable, а також найбільш значущі його спадкоємці – класи Error, Exception, RuntimeException, – містяться саме в пакеті java.lang.

Висновок

У цій лекції ми розповіли про призначення і можливості класів, представлених в пакеті java.lang. Як Ви тепер знаєте, пакет java.lang автоматично імпортується у все Java програми і містить фундаментальні класи і інтерфейси, які складають основу для інших пакетів Java.

Були розглянуті всі найбільш важливі класи пакету java.lang:

- Object, Class – основні класи, представляючий об'єкт і клас об'єктів;
- *класи-обертки (Wrapper класи) – служать для представлення примітивних значень у вигляді об'єктів, оскільки багато класів працюють саме з об'єктами;*

- Math – клас, що надає набір статичних методів, що реалізовують базові математичні функції;
- String і StringBuffer – класи для роботи з рядками;
- System, Runtime, Process, ClassLoader, SecurityManager – системні класи, що допомагають взаємодіяти з програмним оточенням (System, Runtime, Process), завантажувати класи в JVM (ClassLoader) і управляти безпекою (SecurityManager);
- Thread, ThreadGroup, Runnable – типи, що забезпечують роботу з потоками виконання в Java;
- Throwable, Error, Exception, RuntimeException – базові класи для всіх виключень.

ТЕМА. *Пакет java.util*

План

1. *Допоміжні класи пакету java.util.*
2. *Завдання класів та їх особливості.*

Рекомендована література: 2, 3, 5.

Теоретичні відомості

В пакеті java.util міститься безліч допоміжних класів і інтерфейсів. Вони настільки зручні, що практично будь-яка програма використовує цю бібліотеку.

Робота з датами і часом

Клас Date

Клас Date спочатку надавав набір функцій для роботи з датою - для отримання поточного року, місяця і так далі Проте зараз всі перераховані методи не рекомендовані до використання і практично всю функціональність для цього надає клас Calendar.

Існує декілька конструкторів класу Date, проте рекомендовано до використання два:

Date() і Date(long date)

Другий конструктор приймає як параметр значення типу long, вказуюче на кількість мілісекунд, що пройшли з 1 січня 1970 р., 00:00:00 по Грінвічу. Перший конструктор створює екземпляр, відповідний теперішньому моменту. Фактично це еквівалентно другому варіанту new Date(System.currentTimeMillis()). Можна вже після створення екземпляра класу Date використовувати метод setTime(long time) для того, щоб задати потрібний час. Для порівняння дат служать методи after(Date date) і before(Date date), які повертають булеве значення, залежно від того, виконана умова чи ні. Метод compareTo(Date anotherDate) повертає значення типу int, яке рівне -1, якщо дата менше порівнюваною, 1 - якщо більше і 0 - якщо дати рівні. Метод toString() повертає рядковий опис дати. Проте для зрозумілішого і зручнішого перетворення дати в текст рекомендується користуватися класом SimpleDateFormat, визначеним в пакеті java.text.

Класи Calendar і GregorianCalendar

Розвиненіші засоби для роботи з датами представляє клас Calendar. Calendar є абстрактним класом. Для різних платформ

реалізуються конкретні підкласи календаря. На даний момент існує реалізація Григоріанського календаря - GregorianCalendar. Екземпляр цього класу виходить шляхом виклику статичного методу getInstance(), який повертає екземпляр класу GregorianCalendar. Підкласи класу Calendar повинні інтерпретувати об'єкт Date по-різному. В майбутньому передбачається реалізувати також місячний календар, використовуваний в деяких країнах.

Calendar забезпечує набір методів, що дозволяють маніпулювати різними "частинами" дати, тобто отримувати і встановлювати дні, місяці, тижні і так далі

Якщо при завданні параметрів календаря деякі параметри упущені, то для них будуть використані значення за умовчанням для початку відліку, тобто

YEAR = 1970, MONTH = JANUARY, DATE = 1 і так далі

Для прочитування і установки різних "частин" дати використовуються методи get(int field), set(int field, int value), add(int field, int amount), roll(int field, int amount), змінна типу int з ім'ям field вказує на номер поля, з яким потрібно провести операцію. Для зручності всі ці поля визначені вCalendar як статичні константи типу int.

Розглянемо докладніше порядок виконання перерахованих методів.

Метод set(int field,int value).

Як вже мовилося, даний метод проводить установку якого-небудь поля дати. Насправді після виклику цього методу негайного перерахунку дати не проводиться. Перерахунок дати буде здійснений тільки після виклику методів get(), getTime() або getTimeInMillis(). Таким чином, послідовна установка декілька полів не викличе непотрібних обчислень. Окрім цього, з'являється ще один цікавий ефект. Розглянемо наступний приклад. Припустимо, що дата встановлена на останній день серпня. Необхідно перекласти її на останній день вересня. Якби внутрішнє представлення дати змінювалося після виклику методу set, то при послідовній установці полів ми отримали б ось такий ефект:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR,2002);
    cal.set(Calendar.MONTH,Calendar.AUGUST);
    cal.set(Calendar.DAY_OF_MONTH,31);
    System.out.println(" Initialy set date: " + sdf.format(cal.getTime()));
    cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
```

```
            System.out.println(" Date with month changed : " +
sdf.format(cal.getTime()));
            cal.set(Calendar.DAY_OF_MONTH,30);
            System.out.println(" Date with day changed : " +
sdf.format(cal.getTime()));

          }
        }
```

Приклад 14.1. (html, txt)

Результатом буде:

```
Initialy set date: 2002 August 31 22:57:47
Date with month changed : 2002 October 01 22:57:47
Date with day changed : 2002 October 30 22:57:47
```

Приклад 14.2. (html, txt)

Як ми бачимо, в даному прикладі при зміні місяця день місяця залишився незмінним і було успадковано його попереднє значення. Але оскільки у вересні 30 днів, дата автоматично була перекладена на 1 жовтня, і коли було б встановлено 30 число, воно відносилося б вже до жовтня. У наступному прикладі прочитування дати не проводиться, відповідно, її обчислення не виконується до тих пір, поки всі поля не встановлені:

```
        public class Test {
          public Test() {
          }
          public static void main(String[] args){
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd
HH:mm:ss");

            Calendar cal = Calendar.getInstance();
            cal.set(Calendar.YEAR,2002);
            cal.set(Calendar.MONTH,Calendar.AUGUST);
            cal.set(Calendar.DAY_OF_MONTH,31);
            System.out.println(" Initialy set date: " + sdf.format(cal.getTime()));
           cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
           cal.set(Calendar.DAY_OF_MONTH,30);
            System.out.println(" Date with day and month changed : " +
sdf.format(cal.getTime()));
          }
        }
```

Приклад 14.3. (html, txt)

Результатом буде:

```
Initialy set date: 2002 August 31 23:03:51
Date with day and month changed: 2002 September 30 23:03:51
```

Приклад 14.4. (html, txt)

Метод add(int field,int delta).

Додає деякий зсув до існуючої величини поля. В принципі, те ж саме можна зробити за допомогою set(f, get(f)+ delta).

У разі використання методу add слід пам'ятати про два правила:

1. Якщо величина поля зміни виходить за діапазон можливих значень даного поля, то проводиться ділення по модулю даної величини, приватне підсумовується з наступним за старшинством полемо.
2. Якщо змінюється одне з полів, причому, після зміни молодше по відношенню до змінного поля набуває некоректного значення, то воно змінюється на те, яке максимальне близько до "старого".

```
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR,2002);
    cal.set(Calendar.MONTH,Calendar.AUGUST);
    cal.set(Calendar.DAY_OF_MONTH,31);
    cal.set(Calendar.HOUR_OF_DAY,19);
    cal.set(Calendar.MINUTE,30);
    cal.set(Calendar.SECOND,00);
    System.out.println("Current date: " + sdf.format(cal.getTime()));
    cal.add(Calendar.SECOND,75);
    System.out.println("Current date: " + sdf.format(cal.getTime()));
    cal.add(Calendar.MONTH,1);
    System.out.println("Current date: " + sdf.format(cal.getTime()));
  }
}
```

Приклад 14.5. (html, txt)

Результатом буде:

```
Current date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:31:15
Rule 2: 2002 September 30 19:31:15
```

Приклад 14.6. (html, txt)

Метод roll(int field,int delta).

Додає деякий зсув до існуючої величини поля і не проводить зміни старших полів. Розглянемо приведений раніше приклад, але з використанням методу roll.

```
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd
HH:mm:ss");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR,2002);
    cal.set(Calendar.MONTH,Calendar.AUGUST);
    cal.set(Calendar.DAY_OF_MONTH,31);
    cal.set(Calendar.HOUR_OF_DAY,19);
    cal.set(Calendar.MINUTE,30);
    cal.set(Calendar.SECOND,00);
    System.out.println("Current date: " + sdf.format(cal.getTime()));
    cal.roll(Calendar.SECOND,75);
    System.out.println("Rule 1: " + sdf.format(cal.getTime()));
    cal.roll(Calendar.MONTH,1);
    System.out.println("Rule 2: " + sdf.format(cal.getTime()));
  }
}
```

Приклад 14.7. (html, txt)

Результатом буде:

```
Current date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:30:15
Rule 2: 2002 September 30 19:30:15
```

Приклад 14.8. (html, txt)

Як видно з результатів роботи приведеного вище коду, дія правила 1 змінилося в порівнянні з методом add, а правило 2 діє так само.

Клас TimeZone

Клас TimeZone призначений для сумісного використання з класами Calendar і DateFormat. Клас абстрактний, тому від нього породжувати об'єкти не можна. Замість цього визначений статичний метод getDefault(), який повертає екземпляр спадкоємця TimeZone з налаштуваннями, узятими з операційної системи, під управлінням якої працює JVM. Для того, щоб вказати довільні параметри, можна скористатися статичним методом getTimeZone(String ID), як параметр якому передається найменування конкретного тимчасового поясу, для якого необхідно отримати об'єкт TimeZone. Набор полів, що визначають можливий набір параметрів для getTimeZone, ніде явно не описується. Замість цього визначений статичний метод String[] getAvailableIds(), який повертає можливі значення для параметра getTimeZone. Так можна визначити набір можливих параметрів для конкретного тимчасового поясу (розраховується щодо Грінвіча) String[] getAvailableIds(int offset).

Розглянемо приклад, в якому на консоль послідовно виводяться:

- часова зона за умовчанням;
- список всіх можливих часових зон;
- список часових зон, які збігаються з поточною часовою зоною.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    TimeZone tz = TimeZone.getDefault();
    int rawOffset = tz.getRawOffset();
    System.out.println("Current TimeZone" + tz.getDisplayName() +
tz.getID() + "\n\n");
    // Dispaly all available TimeZones
    System.out.println("All Available TimeZones \n");
    String[] idArr = tz.getAvailableIDs();
    for(int cnt=0;cnt < idArr.length;cnt++){
    tz = TimeZone.getTimeZone(idArr[cnt]);
    System.out.println(test.padr(tz.getDisplayName() +
      tz.getID(),64) + " raw offset=" + tz.getRawOffset() +
      ";hour offset=(" + tz.getRawOffset()/ (1000 * 60 * 60 ) + ")");
    }
    // Dispaly all available TimeZones same as for Moscow
    System.out.println("\n\n TimeZones same as for Moscow \n");
    idArr = tz.getAvailableIDs(rawOffset);
    for(int cnt=0;cnt < idArr.length;cnt++){
      tz = TimeZone.getTimeZone(idArr[cnt]);
      System.out.println(test.padr(tz.getDisplayName()+
        tz.getID(),64) + " raw offset=" + tz.getRawOffset() +
        ";hour offset=(" + tz.getRawOffset()/ (1000 * 60 * 60 ) + ")");
    }
  }
  String padr(String str,int len){
    if(len - str.length() > 0){
      char[] buf = new char[len - str.length()];
      Arrays.fill(buf' ');
      return str + new String(buf);
    }else{
      return str.substring(0,len);
    }
  }
}
```

Приклад 14.9. (html, txt)

Результатом буде:

```
Current TimeZone Moscow Standard TimeEurope/Moscow
TimeZones same as for Moscow
```

Eastern African TimeAfrica/Addis_Aba raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Asmera raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Dar_es_Sa raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Djibouti raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Kampala raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Khartoum raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Mogadishu raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Nairobi raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Aden raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Baghdad raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Bahrain raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Kuwait raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Qatar raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Riyadh raw offset=10800000;hour offset=(3)
Eastern African TIMEEAT raw offset=10800000;hour offset=(3)
Moscow Standard TimeEurope/Moscow raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Antananar raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Comoro raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Mayotte raw offset=10800000;hour offset=(3)

Приклад 14.10. (html, txt)

Клас SimpleTimeZone

Клас SimpleTimeZone, як нащадок TimeZone, реалізує його абстрактні методи і призначений для застосування в налаштуваннях, що використовують Григоріанський календар. В більшості випадків немає необхідності створювати екземпляр даного класу за допомогою конструктора. Замість цього краще використовувати статичні методи, які повертають тип TimeZone, розглянуті в попередньому параграфі. Єдина, мабуть, причина для використання конструктора - необхідність завдання нестандартних правил переходу на зимовий і літній час.

У класі SimpleTimeZone визначено три конструктори. Розглянемо якнайповніший з погляду функціональності варіант, який, окрім часової зони, задає літній і зимовий час.

```
public SimpleTimeZone(int rawOffset
    String ID
    int startMonth
    int startDay
    int startDayOfWeek
    int startTime
    int endMonth
    int endDay
    int endDayOfWeek
```

int endTime)

rawOffset - часове зміщення відносне гринвича;

ID - ідентифікатор часової зони (див. пред.параграф);

startMonth - місяць переходу на літній час;

startDay - день місяця переходу на літнє время*;

startDayOfWeek - день тижня переходу на літнє время*;

startTime - час переходу на літній час (указується в мілісекундах);

endMonth - місяць закінчення дії літнього часу;

endDay - день закінчення дії літнього времени*;

endDayOfWeek- день тижня закінчення дії літнього времени*;

endTime - час закінчення дії літнього часу (указується в мілісекундах).

Переведення годинників на зимовий і літній варіант числення часу визначається спеціальним урядовим указом. Зазвичай перехід на літній час відбувається о 2 годині в останнє воскресіння березня, а перехід на зимовий час - о 3 годині в останнє воскресіння жовтня.

Алгоритм розрахунку такий:

- якщо startDay=1 і встановлений день тижня, то обчислюватиметься перший день тижня startDayOfWeekмесяца startMonth (наприклад, перше воскресіння);
- якщо startDay=-1 і встановлений день тижня, то обчислюватиметься останній день тижня startDayOfWeekмесяца startMonth (наприклад, останнє воскресіння);
- якщо день тижня startDayOfWeek встановлений в 0, то обчислюватиметься число startDay конкретного месяцаstartMonth;
- для того, щоб встановити день тижня після конкретного числа, специфікується негативне значення дня тижня. Наприклад, щоб вказати перший понеділок після 23 лютого, використовується ось такий набор:startDayOfWeek=-MONDAY, startMonth=FEBRUARY, startDay=23
- для того, щоб вказати останній день тижня перед яким-небудь числом, указується негативне значення цього числа і негативне значення дня тижня. Наприклад, для того, щоб вказати останню суботу 23 лютого, необхідно задати такий набір параметрів: startDayOfWeek=-SATURDAY, startMonth=FEBRUARY, startDay=-23;
- все вищеперелічене відноситься також і до закінчення дії літнього часу.

Розглянемо приклад отримання поточної часової зони із завданням переходу на зимовий і літній час для Росії за умовчанням.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    SimpleTimeZone stz = new SimpleTimeZone(
      TimeZone.getDefault().getRawOffset()
      ,TimeZone.getDefault().getID()
      ,Calendar.MARCH
      ,-1
      ,Calendar.SUNDAY
      ,test.getTime(2,0,0,0)
      ,Calendar.OCTOBER
      ,-1
      ,Calendar.SUNDAY
      ,test.getTime(3,0,0,0)
      );
    System.out.println(stz.toString());
  }
  int getTime(int hour,int min,int sec,int ms){
    return hour * 3600000 + min * 60000 + sec * 1000 + ms;
  }
}
```

Приклад 14.11. (html, txt)

Результатом буде:

```
java.util.SimpleTimeZone[id=Europe/Moscow,offset=10800000,dstSavings
=3600000,useDaylight=true,
startYear=0,startMode=2,startMonth=2,startDay=-
1,startDayOfWeek=1,startTime=7200000,startTimeMode=0,
endMode=2,endMonth=9,endDay=-
1,endDayOfWeek=1,endTime=10800000,endTimeMode=0]
```

Приклад 14.12. (html, txt)

Інтерфейс Observer і клас Observable

Інтерфейс Observer визначає всього один метод, update (Observable про, Object arg), який викликається, коли об’єкт, що оглядає, змінюється.

Клас Observable призначений для підтримки об’єкту, що оглядає, в парадигмі MVC (model-view-controller), яка, як і інші проектні рішення і шаблони, описана в спеціальній літературі. Цей клас має бути успадкований, якщо виникає необхідність в тому, щоб відстежувати стан якого-небудь об’єкту. Об’єкт, що оглядає, може мати декілька оглядачів. Відповідно, вони повинні реалізувати інтерфейс Observer.

Після того, як в стані об'єкту, що оглядає, щось міняється, необхідно викликати метод notifyObservers, який, у свою чергу, викликає методи update у кожного оглядача.

Порядок, в якому викликаються методи update оглядачів, заздалегідь не визначений. Реалізація за умовчанням має на увазі їх виклик в порядку реєстрації. Реєстрація здійснюється за допомогою методу addObserver(Observer про). Видалення оглядача із списку виконується за допомогою deleteObserver(Observer про). Перед викликом notifyObservers необхідно викликати метод setChanged, який встановлює ознаку того, що об'єкт, що оглядає, був змінений.

Розглянемо приклад організації взаємодії класів:

```
public class TestObservable extends java.util.Observable {
  private String name = "";
  public TestObservable(String name){
    this.name = name;
  }

  public void modify() {
    setChanged();
  }

  public String getName() {
    return name;
  }
}

public class TestObserver implements java.util.Observer {
  private String name = "";

  public TestObserver(String name){
    this.name = name;
  }
  public void update(java.util.Observable o,Object arg){
    String str = "Called update of " + name;
    str += " from " + ((TestObservable)o).getName();
    str += " with argument " + (String)arg;
    System.out.println(str);
  }
}
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    TestObservable to = new TestObservable("Observable");
    TestObserver o1 = new TestObserver("Observer 1");
    TestObserver o2 = new TestObserver("Observer 2");
```

```
        to.addObserver(o1);
        to.addObserver(o2);
        to.modify();
        to.notifyObservers("Notify argument");
      }
    }
```

Приклад 14.13. (html, txt)

В результаті роботи на консоль буде виведено:

Called update of Observer 2 from Observable with argument Notify argument
Called update of Observer 1 from Observable with argument Notify argument

Приклад 14.14. (html, txt)

На практиці використовувати Observer не завжди зручно, оскільки в Java відсутнє множинне спадкоємство і Observer необхідно успадковувати на самому початку побудови ієрархії класів. Як варіант, можна запропонувати визначити інтерфейс, задаючий функціональність, схожу з Observer, і реалізувати його у відповідному класі.

Колекції

Часто в програмі робота йде не з одним об'єктом, а з цілою групою більш менш однотипних екземплярів (наприклад, автопарк організації). Найпростіше зробити це за допомогою масивів. Проте, не дивлячись на те, що це достатньо ефективне рішення для багатьох випадків, воно має деякі обмеження. Так, звертатися до елементу масиву можна тільки по його номеру (індексу). Також необхідно заздалегідь задати довжину масиву і більше її не міняти.

Масиви існували в Java спочатку. Крім того, було визначено два класи для організації ефективнішої роботи з наборами об'єктів: Hashtable і Vector. У JDK 1.2 набір класів, що підтримують роботу з колекціями, був істотно розширений.

Існує декілька різних типів класів-колекцій. Всі вони розроблялися, по можливості, відповідно до єдиної логіки і певних інтерфейсів і там, де це можливо, робота з ними уніфікована. Проте всі колекції відрізняються внутрішніми механізмами зберігання, швидкістю доступу до елементів, споживаною пам'яттю і іншими деталями. Наприклад, в деяких колекціях об'єкти (також звані елементами колекцій), можуть бути впорядковані, в деяких - ні. У деяких типах колекцій допускається дублювання посилань на об'єкт, в деяких - ні. Далі ми розглянемо кожен з класів-колекцій.

Класи, що забезпечують маніпулювання колекціями об'єктів, оголошені в пакеті java.util.

Інтерфейси

Інтерфейс Collection

Даний інтерфейс є коренем всієї ієрархії класів-колекцій. Він визначає базову функціональність будь-якої колекції - набір методів, які дозволяють додавати, видаляти, вибирати елементи колекції. Класи, які реалізують інтерфейс Collection, можуть містити дублікати і порожні (null) значення.

AbstractCollection, як абстрактний клас, служить основою для створення конкретних класів колекцій і містить реалізацію деяких методів, визначених в інтерфейсі Collection.

Інтерфейс Set

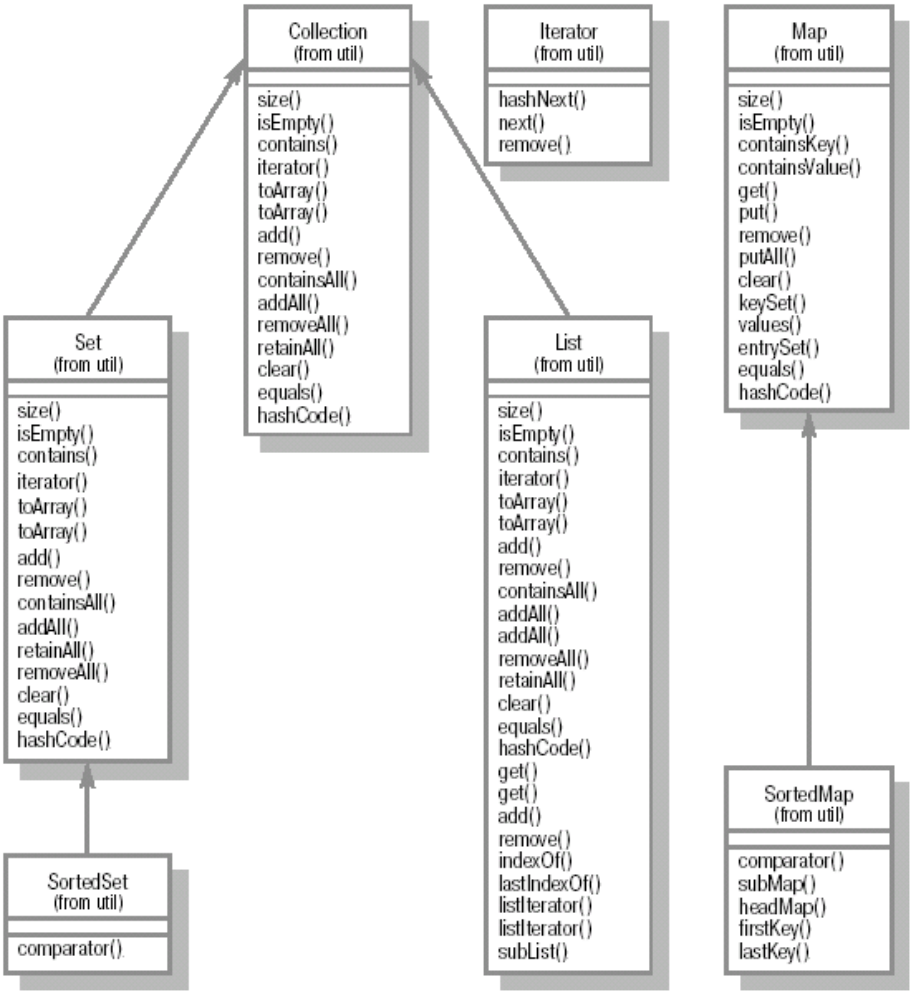
Класи, які реалізують цей інтерфейс, не допускають наявності дублікатів. У колекції цього типу дозволена наявність тільки одній посилання типу null. Інтерфейс Set розширює інтерфейс Collection, таким чином, будь-який клас, імплементуючий Set, реалізує всі методи, визначені в Collection. Будь-який об'єкт, в Set, що додається, повинен реалізувати метод equals, щоб його можна було порівняти з іншими.

AbstractSet, будучи абстрактним класом, є основою для реалізації різних варіантів інтерфейсу Set.

Інтерфейс List

Класи, що реалізовують цей інтерфейс, містять впорядковану послідовність об'єктів (об'єкти зберігаються в тому порядку, в якому вони були додані). У JDK 1.2 був перероблений клас Vector, так, що він тепер реалізує інтерфейс List. Інтерфейс List розширює інтерфейс Collection, і будь-який клас, імплементуючий List, реалізує всі методи, визначені в Collection, і в той же час вводяться нові методи, які дозволяють додавати і видаляти елементи із списку. List також забезпечує ListIterator, який дозволяє переміщатися як вперед, так і назад по елементах списку.

AbstractList, як абстрактний клас, є основою для реалізації різних варіантів інтерфейсу List.



Мал. 14.1. Основні типи для роботи з колекціями.

Інтерфейс Map

Класи, які реалізують цей інтерфейс, зберігають неврегульований набір об'єктів парами ключ/значення. Кожен ключ має бути унікальним. Hashtable після модифікації в JDK 1.2 реалізує інтерфейс Map. Порядок проходження пар ключ/значення не визначений.

Інтерфейс Map не розширює інтерфейс Collection. AbstractMap, будучи абстрактним класом, є основою для реалізації різних варіантів інтерфейсу Map.

Інтерфейс SortedSet

Цей інтерфейс розширює Set, вимагаючи, щоб вміст набору був впорядкований. Такі колекції можуть містити об'єкти, які реалізують інтерфейс Comparable, або можуть порівнюватися з використанням зовнішнього Comparator.

Інтерфейс SortedMap

Цей інтерфейс розширює Map, вимагаючи, щоб вміст колекції був впорядкований по значеннях ключів.

Інтерфейс Iterator

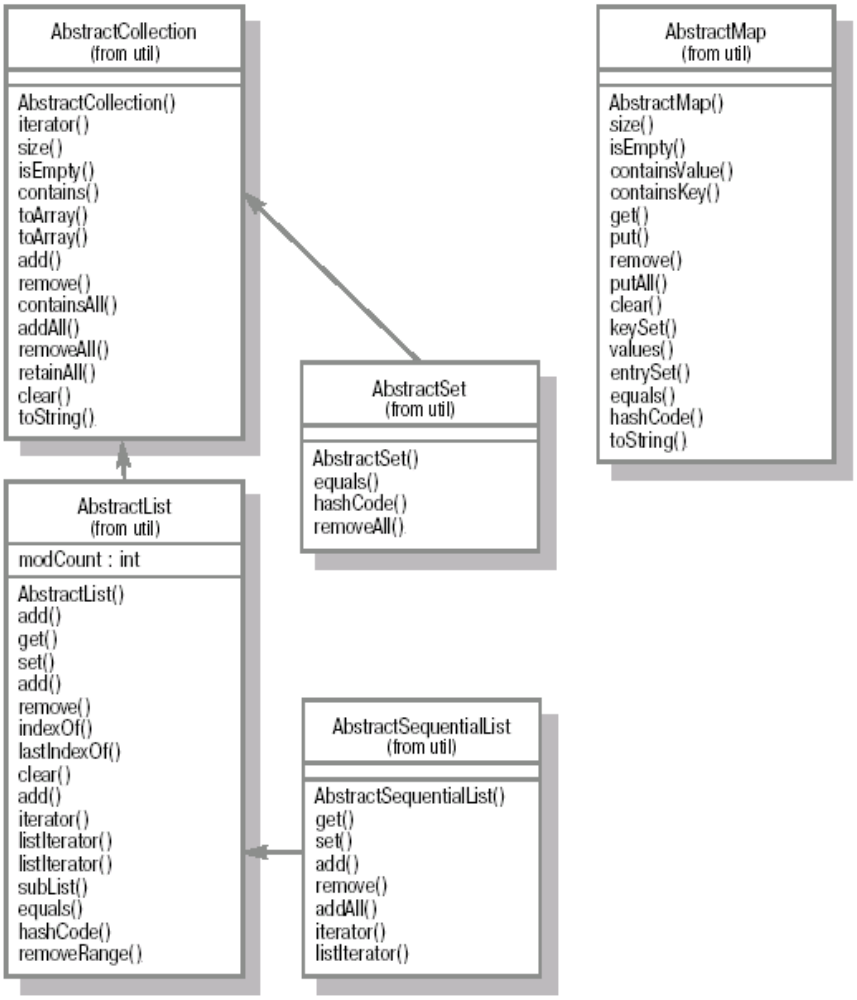
У Java 1 для перебору елементів колекції використовувався інтерфейс Enumeration. У Java 2 для цих цілей повинні застосовуватися об'єкти, які реалізують інтерфейс Iterator. Всі класи, які реалізують інтерфейс Collection, повинні реалізувати метод iterator, який повертає об'єкт, що реалізовує інтерфейс Iterator.Iterator вельми схожий на Enumeration, з тим лише відмінністю, що в нім визначений метод remove, який дозволяє видалити об'єкт з колекції, для якої Iterator був створений.

Таким чином, підводячи підсумок, перерахуємо інтерфейси, використовувані при роботі з колекціями:

java.util.Collection
java.util.Set
java.util.List
java.util.Map
java.util.SortedSet
java.util.SortedMap
java.util.Iterator

Абстрактні класи, які використовуються при роботі з колекціями

java.util.AbstractCollection - даний клас реалізує всі методи, визначені в інтерфейсі Collection, за винятком iterator і size, так що для того, щоб створити колекцію, що не модифікується, потрібно перевизначити ці методи. Для реалізації колекції, що модифікується, необхідно ще перевизначити методpublic void add(Object про) (інакше при його виклику буде збуджено виключення UnsupportedOperationException).



Мал. 14.2. Базові абстрактні класи.

Необхідно також визначити два конструктори: без аргументів і з аргументом Collection. Перший повинен створювати порожню колекцію, другий - колекцію на основі тієї, що існує. Даний клас розширюється класами AbstractList і AbstractSet.

java.util.AbstractList - цей клас розширює AbstractCollection і реалізує інтерфейс List. Для створення списку, що не модифікується, необхідно імплементувати методи public Object get(int index) і public int size(). Для реалізації списку, що модифікується, необхідно також реалізувати метод public void set(int index,Object element) (інакше при його виклику буде збуджено исключениеUnsupportedOperationException).

На відміну від AbstractCollection, в цьому випадку немає необхідності реалізовувати метод iterator, оскільки він вже реалізований поверх методів доступу до елементів списку get, set, add, remove.

java.util.AbstractSet - даний клас розширює AbstractCollection і реалізує основну функціональність, визначену в інтерфейсі Set. Слід зазначити, що цей клас не перевизначає функціональність, реалізовану в класі AbstractCollection.

java.util.AbstractMap - цей клас розширює основну функціональність, визначену в інтерфейсі Map. Для реалізації класу, що не модифікується, успадкованого від AbstractMap, досить визначити методentrySet, який повинен повертати об'єкт, що приводиться до типу AbstractSet. Цей набір (Set) не повинен забезпечувати методів для додавання і видалення елементів з набору. Для реалізації классаMap , що модифікується, необхідно також перевизначити метод put і додати в ітератор, entrySet, що повертається ().iterator(), підтримку методу remove.

java.util.AbstractSequentialList - цей клас розширює AbstractList і є основою для класу LinkedList. Основна відмінність від AbstractList полягає в тому, що цей клас забезпечує не тільки послідовний, але і довільний доступ до елементів списку, за допомогою методів get(int index), set(int index, Object element), add(int index, Object element) і remove(int index). Для того, щоб реалізувати даний клас, необхідно перевизначити методи listIterator і size. Причому, якщо реалізується список, що не модифікується, для ітератора досить реалізувати методи hasNext, next, hasPrevious, previousu index. Для списку, що модифікується, необхідно додатково реалізувати метод set, а для списків змінної довжини ще і add, і remove.

Конкретні класи колекцій

java.util.ArrayList - цей клас розширює AbstractList і вельми схожий на клас Vector. Він також динамічно розширюється, як Vector, проте його методи не є синхронізованими, унаслідок чого операції з ним виконуються швидше. Для того, щоб скористатися синхронізованою версією ArrayList, можна застосувати ось таку конструкцію:

```
List l = Collections.synchronizedList(new ArrayList(...));
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    ArrayList al = new ArrayList();
    al.add("First element");
    al.add("Second element");
    al.add("Third element");
    Iterator it = al.iterator();
    while(it.hasNext()){
      System.out.println((String)it.next());
    }
```

```
    System.out.println("\n");
    al.add(2,"Insertion");
    it = al.iterator();
    while(it.hasNext()){
      System.out.println((String)it.next());
    }
  }
}
```

Приклад 14.15. (html, txt)

Результатом буде:

```
First element
Second element
Third element

Firts element
Second element
Insertion
Third element
```

Приклад 14.16. (html, txt)

java.util.LinkedList - є реалізацією інтерфейсу List. Він реалізує всі методи интерфейсаList, окрім цього додаються ще нові методи, які дозволяють додавати, видаляти і отримувати елементи в кінці і початку списку. LinkedList є двохзв'язним списком і дозволяє переміщатися як від початку в кінець списку, так і навпаки. LinkedList зручно використовувати для організації стека.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    LinkedList ll = new LinkedList();
    ll.add("Element1");
    ll.addFirst("Element2");
    ll.addFirst("Element3");
    ll.addLast("Element4");
    test.dumpList(ll);
    ll.remove(2);
    test.dumpList(ll);
    String element = (String)ll.getLast();
    ll.remove(element);
    test.dumpList(ll);
  }
  private void dumpList(List list){
    Iterator it = list.iterator();
    System.out.println();
    while(it.hasNext()){
```

```
            System.out.println((String)it.next());
        }
      }
}
```

Приклад 14.17. (html, txt)

Результатом буде:

```
Element3
Element2
Element1
Element4

Element3
Element2
Element4

Element3
Element2
```

Приклад 14.18. (html, txt)

Класи LinkedList і ArrayList мають схожу функціональність. Проте з погляду продуктивності вони відрізняються. Так, в ArrayList помітно швидше (приблизно на порядок) здійснюються операції проходу за всім списком (ітерації) і отримання даних. LinkedList майже на порядок швидше виконує операції видалення і додавання нових елементів.

java.util.Hashtable - розширює абстрактний клас Dictionary. У JDK 1.2 клас Hashtable також реалізує інтерфейс Map. Hashtable призначений для зберігання об'єктів у вигляді пар ключ/значення. З самої назви виходить, що Hashtable використовує алгоритм хэширования для збільшення швидкості доступу до даних. Для того, щоб з'ясувати принципи роботи даного алгоритму, розглянемо декілька прикладів.

Припустимо, є масив рядків, що містить назви міст. Для того, щоб знайти елемент масиву, що містить назву міста, в загальному випадку потрібно проглянути весь масив, а якщо необхідно знайти всі елементи масиву, то для пошуку кожного, в середньому, потрібно буде проглядати половину масиву. Такий підхід може виявитися прийнятним тільки для невеликих масивів.

Як вже наголошувалося раніше, для того, щоб збільшити швидкість пошуку, використовується алгоритм хэширования. Кожен об'єкт в Java успадкований від Object. Як вже наголошувалося раніше, hash визначене як ціле число, яке унікально ідентифікує екземпляр класу Object і, відповідно, всі екземпляри класів, успадкованих від Object. Це число повертає метод hashCode(). Саме воно використовується при збереженні ключа в Hashtable таким чином: розділивши довжину масиву, призначеного для зберігання ключів, на код, отримуємо якесь ціле число, яке служить індексом для зберігання ключа в масиві array.length % hashCode().

Далі, якщо необхідно додати нову пару ключ/значение, обчислюється новий індекс, і якщо цей індекс збігається з що вже є, то створюється список ключів, на який указує елемент масиву ключів. Таким чином, при зворотному витяганні ключа необхідно обчислити індекс масиву по тому ж алгоритму і отримати його. Якщо ключ в масиві єдиний, то використовується значення елементу масиву, якщо зберігається декілька ключів, то необхідно обійти список і вибрати потрібний.

Є декілька міркувань, що відносяться до продуктивності класів, що використовують для зберігання даних алгоритм кеширования. Зокрема, розмір масиву. Якщо масив опиниться дуже малий, то зв'язані списки будуть дуже довгими і швидкість пошуку почне істотно знижуватися, оскільки проглядання елементів списку буде таким же, як в звичайному масиві. Щоб цього уникнути, задається якийсь коефіцієнт заповнення. При заповненні елементів масиву, в якому зберігаються ключі (або списки ключів) на цю величину, відбувається збільшення масиву і проводиться повторне реиндексирование. Таким чином, якщо масив опиниться дуже малий, то він швидко заповнюватиметься і проводитиметься операція повторного індексування, яка віднімає достатньо багато ресурсів. З іншого боку, якщо масив зробити великим, то при необхідності проглянути послідовно всі елементи колекції, що використовує алгоритм кеширования, доведеться обробляти велику кількість порожніх елементів масиву ключів.

Початковий розмір масиву і коефіцієнт завантаження колекції задаються при конструюванні. Наприклад:

```
Hashtable ht = new Hashtable(1000,0.60)
```

Існує також конструктор без параметрів, який використовує значення за умовчанням 101 для розміру масиву (у останній версії значення зменшене до 11) і 0.75 для коефіцієнта завантаження.

Використання алгоритму кеширования дозволяє гарантувати, що швидкість доступу до елементів колекції такого типу збільшуватиметься не лінійно, а логарифмічно. Таким чином, при частому пошуку яких-небудь значень по ключу має сенс задіювати колекції, що застосовують алгоритм кеширования.

java.util.HashMap - цей клас розширює AbstractMap і вельми схожий на клас Hashtable. HashMapпредназначен для зберігання пар об'єктів ключ/значення. Як для ключів, так і для елементів допускаються значення типу null. Порядок зберігання елементів в цій колекції не збігається з порядком їх додавання. Порядок елементів в колекції також може мінятися в часі. HashMap забезпечує постійний час доступу для операцій get і put.

Ітерація по всіх елементах колекції пропорційна її ємкості. Тому має сенс не робити розмір колекцій надмірно великим, якщо достатньо часто доведеться здійснювати ітерацію по елементах.

Методи HashMap не є синхронізованими. Для того, щоб забезпечити нормальну роботу в багатопотоковому варіанті, слід використовувати або зовнішню синхронізацію потоків, або синхронізований варіант колекції.

```
public class Test {
  private class TestObject{
    String text = "";
    public TestObject(String text){
      this.text = text;
    };
    public String getText(){
      return this.text;
    }
    public void setText(String text){
      this.text = text;
    }
  }
  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    TestObject to = null;
    HashMap hm = new HashMap();
    hm.put("Key1",t.new TestObject("Value 1"));
    hm.put("Key2",t.new TestObject("Value 2"));
    hm.put("Key3",t.new TestObject("Value 3"));
    to = (TestObject)hm.get("Key1");
    System.out.println("Object value for Key1 = " + to.getText() + "\n");
    System.out.println("Iteration over entrySet");
    Map.Entry entry = null;
    Iterator it = hm.entrySet().iterator();
    // Ітератор для перебору всіх точок входу в Map
    while(it.hasNext()){
      entry = (Map.Entry)it.next();
      System.out.println("For key = " + entry.getKey() +
        " value = " + ((TestObject)entry.getValue()).getText());
    }
    System.out.println();
    System.out.println("Iteration over keySet");
    String key = "";
    // Ітератор для перебору всіх ключів в Map
    it = hm.keySet().iterator();
    while(it.hasNext()){
      key = (String)it.next();
      System.out.println( "For key = " + key + " value = " +
        ((TestObject)hm.get(key)).getText());
    }
  }
}
```

Приклад 14.19. (html, txt)

Результатом буде:

```
Object value for Key1 = Value 1

Iteration over entrySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1

Iteration over keySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1
```

Приклад 14.20. (html, txt)

java.util.TreeMap - розширює клас AbstractMap і реалізує інтерфейс SortedMap. TreeMap містить ключі в порядку зростання. Використовується або натуральне порівняння ключів, або має бути реалізований інтерфейс Comparable. Реалізація алгоритму пошуку забезпечує логарифмічну залежність часу виконання основних операцій (containsKey, get, put і remove). Заборонено застосування null значень для ключів. При використанні дублікатів ключів посилання на об'єкт, збережений з таким же ключем, буде загублено. Наприклад:

```
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    Test t = new Test();
    TreeMap tm = new TreeMap();
    tm.put("key","String1");
    System.out.println(tm.get("key"));
    tm.put("key","String2");
    System.out.println(tm.get("key"));
  }
}
```

Результатом буде:

```
String1
String2
```

Клас Collections

Клас Collections є класом-утилітою і містить декілька допоміжних методів для роботи з класами, що забезпечують різні інтерфейси колекцій. Наприклад, для сортування елементів списків, для пошуку елементів у впорядкованих колекціях і так далі Але, мабуть, найбільш важливою властивістю цього класу є можливість отримання синхронізованих варіантів

класів-колекцій. Наприклад, для отримання синхронізованого варіанту Map можна використовувати наступний підхід:

```
HashMap hm = new HashMap();
:
Map syncMap = Collections.synchronizedMap(hm);
:
```

Як вже наголошувалося раніше, починаючи з JDK 1.2, клас Vector реалізує інтерфейс List. Розглянемо приклад сортування елементів, що містяться в класі Vector.

```
public class Test {
  private class TestObject {
    private String name = "";
    public TestObject(String name){
      this.name = name;
    }
  }
  private class MyComparator implements Comparator {
    public int compare(Object l,Object r){
      String left = (String)l;
      String right = (String)r;
      return -1 * left.compareTo(right);
    }
  }
  public Test() {
  }

  public static void main(String[] args){
    Test test = new Test();
    Vector v = new Vector();
    v.add("bbbbb");
    v.add("aaaaa");
    v.add("ccccc");
    System.out.println("Default elements order");
    test.dumpList(v);
    Collections.sort(v);
    System.out.println("Default sorting order");
    test.dumpList(v);
    System.out.println("Reverse sorting order with providing imlicit
comparator");
    Collections.sort(v,test.new MyComparator());
    test.dumpList(v);
  }
  private void dumpList(List l){
    Iterator it = l.iterator();
    while(it.hasNext()){
      System.out.println(it.next());
    }
```

```
    }
}
```

Приклад 14.21. (html, txt)

Клас Properties

Клас Properties призначений для зберігання набору властивостей (параметрів). Методи

```
String getProperty(String key)
String getProperty(String key
                   String defaultValue)
```

дозволяють набути властивості з набору.

За допомогою методу setProperty(String key, String value) цю властивість можна встановити.

Метод load(InputStream inStream) дозволяє завантажити набір властивостей з вхідного потоку (потоки даних детально розглядаються в лекції 15). Як правило, це текстовий файл, в якому зберігаються параметри. Параметри - це рядки, які є парами ключ/значення. Передбачається, що за умовчанням використовується кодування ISO 8859-1. Кожен рядок повинен закінчуватися символами \r,\n або \r\n. Рядки з файлу прочитуватимуться до тих пір, поки не буде досягнутий його кінець. Рядки, що складаються з одних пропусків, або що починаються із знаків ! або #, ігноруються, тобто їх можна трактувати як коментарі. Якщо рядок закінчується символом /, то наступний рядок вважається за її продовження. Перший символ з початку рядка, відмінний від пропуску, вважається за початок ключа. Перший пропуск, що зустрівся, двокрапка або знак рівності вважається за закінчення ключа. Всі символи закінчення ключа при необхідності можуть бути включені в назву ключа, але при цьому перед ними повинен стояти символ \. Після того, як зустрівся символ закінчення ключа, всі аналогічні символи будуть проігноровані до початку значення. Частина рядка, що залишилася, інтерпретується як значення. У рядку, що складається тільки з символів \t, \n, \r, \\, \", \', \ и\uxxxx, вони всі розпізнаються і інтерпретуються як одиночні символи. Якщо зустрінеться поєднання \ і символу кінця рядка, то наступний рядок вважатиметься за продовження поточною, також будуть проігноровані всі пропуски до початку строки-продовження.

Метод save(OutputStream inStream,String header) зберігає набір властивостей у вихідний потік у вигляді, придатному для вторинного завантаження за допомогою методу load. Символи, що вважаються службовими, кодуються так, щоб їх можна було рахувати при вторинному завантаженні. Символи в національному кодуванні будуть приведені до вигляду \uxxxx. При збереженні використовується кодування ISO 8859-1. Якщо вказаний header, то він буде поміщений в початок потоку у вигляді коментаря (тобто з символом # на початку), далі слідуватиме коментар, в якому буде вказаний час і дата збереження властивостей в потоці.

У класі Properties визначений ще метод list(PrintWriter out), який практично ідентичний save. Відрізняється лише заголовок, який змінити не можна. Крім того, рядки усікаються по ширині. Тому даний метод для збереження Properties не годиться.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    Properties props = new Properties();
    StringWriter sw = new StringWriter();
    sw.write("Key1 = Value1 \n");
    sw.write(" Key2 : Value2 \r\n");
    sw.write(" Key3 Value3 \n ");
    InputStream is = new ByteArrayInputStream(sw.toString().getBytes());

    try {
      props.load(is);
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    props.list(System.out);
    props.setProperty("Key1","Modified Value1");
    props.setProperty("Key4","Added Value4");
    props.list(System.out);
  }
}
```

Приклад 14.22. (html, txt)

Результатом буде:

```
-- listing properties --
Key3=Value3
Key2=Value2
Key1=Value1

-- listing properties --
Key4=Added Value4
Key3=Value3
Key2=Value2
Key1=Modified Value1
```

Приклад 14.23. (html, txt)

Інтерфейс Comparator

У колекціях багато методів сортування або порівняння вимагають передачі як одного з параметрів об'єкту, який реалізує інтерфейс Comparator. Цей інтерфейс визначає єдиний метод compare(Object obj1,Object obj2), який

на підставі визначеного користувачем алгоритму порівнює об'єкти, передані як параметри. Метод compare повинен повернути:

-1 якщо obj1 < obj2
0 якщо obj1 = obj2
1 якщо obj1 > obj2

Клас Arrays

Статичний клас Arrays забезпечує набір методів для виконання операцій над масивами, таких, як пошук, сортування, порівняння. У Arrays також визначений статичний метод public List aList(a[] arr), який повертає список фіксованого розміру, заснований на масиві. Зміни в List можна внести, змінивши дані в масиві.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    String[] arr = {"String 1","String 4"
                    "String 2","String 3"};
    test.dumpArray(arr);
    Arrays.sort(arr);
    test.dumpArray(arr);
    int ind = Arrays.binarySearch(arr
              "String 4");
    System.out.println(
      "\nIndex of \"String 4\" = " + ind);
  }
  void dumpArray(String arr[]){
    System.out.println();
    for(int cnt=0;cnt < arr.length;cnt++){
      System.out.println(arr[cnt]);
    }
  }
}
```

Клас StringTokenizer

Цей клас призначений для розбору рядка по лексемах (tokens). Рядок, який необхідно розібрати, передається як параметр конструктору StringTokenizer(String str). Визначено ще два переобтяжені конструктори, якими додатково можна передати строку-роздільник лексем StringTokenizer(String str, String delim) і ознака повернення роздільника лексем StringTokenizer(String str, String delim, Boolean returnDelims).

Роздільником лексем за умовчанням служить пропуск.

```
public class Test {
```

```
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    String toParse =
      "word1;word2;word3;word4";
    StringTokenizer st =
      new StringTokenizer(toParse";");
    while(st.hasMoreTokens()){
      System.out.println(st.nextToken());
    }
  }
}
```

Результатом буде:

```
word1
word2
word3
word4
```

Клас BitSet

Клас BitSet призначений для роботи з послідовностями бітів. Кожен компонент цієї колекції може набувати булевий значення, яке позначає, встановлений біт чи ні. Вміст BitSet може бути модифіковане вмістом іншого BitSet з використанням операцій AND, OR або XOR (що виключає або).

BitSet має поточний розмір (кількість встановлених бітів), може динамічно змінюватися. За умовчанням всі біти в наборі встановлюються в 0 (false). Установка і очищення бітів в BitSet здійснюється методами set(int index) і clear(int index).

Метод int length() повертає "логічний" розмір набору бітів, int size() повертає кількість пам'яті, займаною бітовою послідовністю BitSet.

```
public class Test {

  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    BitSet bs1 = new BitSet();
    BitSet bs2 = new BitSet();
    bs1.set(0);
    bs1.set(2);
    bs1.set(4);
    System.out.println("Length = " +
```

```
      bs1.length()+" size = "+bs1.size());
    System.out.println(bs1);
    bs2.set(1);
    bs2.set(2);
    bs1.and(bs2);
    System.out.println(bs1);
  }
}
```

Результатом буде:

```
Length = 5 size = 64
{0, 2, 4}
{2}
```

Проаналізувавши перший рядок виводу на консоль, можна зробити вивід, що для внутрішнього представлення BitSet використовує значення типу long.

Клас Random

Клас Random використовується для отримання послідовності псевдовипадкових чисел. Як "зерно" застосовується 48-бітове число. Якщо для ініціалізації Random задіювати одне і те ж число, буде отримана та ж сама послідовність псевдовипадкових чисел.

У класі Random визначене також декілька методів, які повертають псевдовипадкові величини для примітивних типів Java.

Додатково слід зазначити наявність двох методів: double nextGaussian() - повертає випадкове число в діапазоні від 0.0 до 1.0 розподілене по нормальному закону, і void nextBytes(byte[] arr) - заповнює масив arr випадковими величинами типу byte.

Приклад використання Random:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    Random r = new Random(100);
    // Generating the same sequence numbers
    for(int cnt=0;cnt<9;cnt++){
      System.out.print(r.nextInt() + " ");
    }
    System.out.println();
    r = new Random(100);
    for(int cnt=0;cnt<9;cnt++){
      System.out.print(r.nextInt() + " ");
```

```
    }
    System.out.println();
    // Generating sequence of bytes
    byte[] randArray = new byte[8];
    r.nextBytes(randArray);
    test.dumpArray(randArray);
  }
  void dumpArray(byte[] arr){
    for(int cnt=0;cnt< arr.length;cnt++){
      System.out.print(arr[cnt]);
    }
    System.out.println();
  }
}
```

Приклад 14.24. (html, txt)

Результатом буде:

```
-1193959466 -1139614796 837415749 -1220615319 -1429538713
118249332 -951589224 -1193959466
-1139614796 837415749 -1220615319 -1429538713 118249332 -
951589224 81;-6;-107;77;118;17;93;
-98;
```

Приклад 14.25. (html, txt)

Локалізація

Клас Locale

Клас Locale призначений для відображення певного регіону. Під регіоном прийнято розуміти не тільки географічне положення, але також мовне і культурне середовище. Наприклад, окрім того, що указується країна Швейцарія, можна вказати також і мова - французький або німецький.

Визначено два варіанти конструкторів в класі Locale:

```
Locale(String language, String country)
Locale(String language, String country
    String variant)
```

Перші два параметри в обох конструкторах визначають мову і країну, для якої визначається локаль, згідно кодуванню ISO. Список підтримуваних країн і мов можна отримати і за допомогою виклику статичних методів Locale.getISOLanguages() Locale.getISOCountries(), відповідно. У другому варіанті конструктора вказаний також рядковий параметр variant, в якому кодується інформація про платформу. Якщо тут необхідно вказати додаткові параметри, то їх потрібно розділити символом підкреслення, причому, важливіший параметр повинен слідувати першим.

Приклад використання:

```
Locale l = new Locale("ru","RU");
Locale l = new Locale("en","US","WINDOWS");
```

Статичний метод getDefault() повертає поточну локаль, сконструйовану на основі налаштувань операційної системи, під управлінням якої функціонує JVM.

Для найчастіше локалей, що використовуються, задані константи. Наприклад, Locale.US або Locale.GERMAN.

Після того, як екземпляр класу Locale створений, за допомогою різних методів можна отримати додаткову інформацію про локали.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    Locale l = Locale.getDefault();
    System.out.println(l.getCountry() + " " +
      l.getDisplayCountry() + " " + l.getISO3Country());
    System.out.println(l.getLanguage() + " " +
      l.getDisplayLanguage() + " " + l.getISO3Language());
    System.out.println(l.getVariant() + " " + l.getDisplayVariant());
  l = new Locale("ru","RU","WINDOWS");
    System.out.println(l.getCountry() + " " +
      l.getDisplayCountry() + " " + l.getISO3Country());
    System.out.println(l.getLanguage() + " " +
      l.getDisplayLanguage() + " " + l.getISO3Language());
    System.out.println(l.getVariant() + " " + l.getDisplayVariant());
  }
}
```

Приклад 14.26. (html, txt)

Результатом буде:

```
US United States USA
en English eng

RU Russia RUS
ru Russian rus
WINDOWS WINDOWS
```

Приклад 14.27. (html, txt)

Клас ResourceBundle

Абстрактний клас ResourceBundle призначений для зберігання об’єктів, специфічних для локали. Наприклад, коли необхідно отримати набір рядків, залежних від локали, використовують ResourceBundle.

Застосування ResourceBundle настійно рекомендується, якщо передбачається використовувати програму в багатоязиковому середовищі. За допомогою цього класу легко маніпулювати наборами ресурсів, залежних від локалей, їх можна міняти, додавати нові і так далі

Набор ресурсів - це фактично набір класів, що мають одне базове ім'я. Далі найменування класу доповнюється найменуванням локали, з якою зв'язується цей клас. Наприклад, якщо ім'я базового класу буде MyResources, то для англійської локали ім'я класу буде MyResources_en, для російської - MyResources_ru. Окрім цього, може додаватися ідентифікатор мови, якщо для даного регіону визначено декілька мов. Наприклад, MyResources_de_CH - так виглядатиме швейцарський варіант німецької мови. Крім того, можна вказати додаткову ознаку variant (див. опис Locale). Так, описаний раннє приклад для платформи UNIX виглядатиме таким чином: MyResources_de_CH_UNIX .

Завантаження об'єкту для потрібної локали проводиться за допомогою статичного методу getBundle.:

```
ResourceBundle myResources =
  ResourceBundle.getBundle("MyResources"
                  someLocale);
```

На основі вказаного базового імені (перший параметр), вказаною локали (другий параметр) і локали за умовчанням (задається налаштуваннями ОС або JVM) генерується список можливих імен ресурсу. Причому, вказана локаль має вищий пріоритет, ніж локаль за умовчанням. Якщо позначити складові вказаної локали (мова, країна, варіант) як 1, а локали за умовчанням - 2, то список прийме наступний вигляд:

```
baseclass + "_" + language1 + "_" + country1 + "_" + variant1
baseclass + "_" + language1 + "_" + country1 + "_" + variant1 +
  ".properties"
baseclass + "_" + language1 + "_" + country1
baseclass + "_" + language1 + "_" + country1 + ".properties"
baseclass + "_" + language1
baseclass + "_" + language1 + ".properties"
baseclass + "_" + language2 + "_" + country2 + "_" + variant2
baseclass + "_" + language2 + "_" + country2 + "_" + variant2 +
  ".properties"
baseclass + "_" + language2 + "_" + country2
baseclass + "_" + language2 + "_" + country2 + ".properties"
baseclass + "_" + language2
baseclass + "_" + language2 + ".properties"
baseclass
baseclass + ".properties"
```

Приклад 14.28. (html, txt)

Наприклад, якщо необхідно знайти ResourceBundle для локали fr_CH (Швейцарський французький), а локаль за умовчанням en_US,

при цьому назва базового класу ResourceBundle MyResources, то порядок пошуку відповідного ResourceBundle буде такий.

MyResources_fr_CH
MyResources_fr
MyResources_en_US
MyResources_en
MyResources

Результатом роботи getBundle буде завантаження необхідного класу ресурсів в пам'ять, проте дані цього класу можуть бути збережені на диску. Таким чином, якщо потрібний клас не буде знайдений, то до необхідного імені класу буде додано розширення ".properties" і буде зроблена спроба знайти файл з даними на диску.

Слід пам'ятати, що необхідно указувати повністю кваліфіковане ім'я класу ресурсів, тобто ім'я пакету, ім'я класу. Крім того, клас ресурсів має бути доступний в контексті його виклику (там, де вызываетсягetResourceBundle), тобто не бути private і так далі

Завжди повинен створюватися базовий клас без суфіксів, тобто якщо ви створюєте ресурси з ім'ям MyResource, має бути в наявності клас MyResource.class.

ResourceBundle зберігає об'єкти у вигляді пар ключ/значение. Як вже наголошувалося раніше, клас ResourceBundleабстрактный, тому при його спадкоємстві необхідно перевизначити методи:

Enumeration getKeys()
protected Object handleGetObject(String key)

Перший метод повинен повертати список всіх ключів, які визначені в ResourceBundle, другий повинен повертати об'єкт, пов'язаний з конкретним ключем.

Розглянемо приклад використання ResourceBundle:

```
public class MyResource extends ResourceBundle {

  private Hashtable res = null;
  public MyResource() {
    res = new Hashtable();
    res.put("TestKey","English Variant");
  }
  public Enumeration getKeys() {
    return res.keys();
  }
  protected Object handleGetObject(String key) throws
  java.util.MissingResourceException {
    return res.get(key);
```

```
  }
}
public class MyResource_ru_RU extends ResourceBundle {
  private Hashtable res = null;
  public MyResource_ru_RU() {
    res = new Hashtable();
    res.put("TestKey","Русский варинат");
  }
  public Enumeration getKeys() {
    return res.keys();
  }
  protected Object handleGetObject(String key)
  throws java.util.MissingResourceException {
    return res.get(key);
  }
}
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
    System.out.println(rb.getString("TestKey"));
    rb = ResourceBundle.getBundle("experiment.MyResource", new
Locale("ru","RU"));
      System.out.println(rb.getString("TestKey"));
  }
}
```

Приклад 14.29. (html, txt)

Результатом буде:

```
English Variant
Російський Варіант
```

Крім того, слід звернути увагу, що ResourceBundle може зберігати не тільки рядкові значення. У нім можна зберігати також двійкові дані, або просто методи, що реалізовують потрібну функціональність, залежно від локали.

```
public interface Behavior {
  public String getBehavior();
  public String getCapital();
  }
public class EnglishBehavior implements Behavior{
  public EnglishBehavior() {
  }
  public String getBehavior(){
    return "English behavior";
```

```
    }
    public String getCapital(){
      return "London";
    }
  }
  public class RussianBehavior implements Behavior {
    public RussianBehavior() {
    }
    public String getBehavior(){
      return "Російський варіант поведінки";
    }
    public String getCapital(){
      return "Москва";
    }
  }
  public class MyResourceBundle_ru_RU extends ResourceBundle {
    Hashtable bundle = null;
    public MyResourceBundle_ru_RU() {
      bundle = new Hashtable();
      bundle.put("Bundle description","Набор ресурсів для російської
локали");
      bundle.put("Behavior",new RussianBehavior());
    }
    public Enumeration getKeys() {
      return bundle.keys();
    }
    protected Object handleGetObject(String key) throws
        java.util.MissingResourceException {
      return bundle.get(key);
    }
  }

  public class MyResourceBundle_en_EN extends ResourceBundle {
    Hashtable bundle = null;
    public MyResourceBundle_en_EN() {
      bundle = new Hashtable();
      bundle.put("Bundle description","English resource set");
      bundle.put("Behavior",new EnglishBehavior());
    }
    public Enumeration getKeys() {
      return bundle.keys();
    }
    protected Object handleGetObject(String key) throws
        java.util.MissingResourceException {
      return bundle.get(key);
    }
  }
  public class MyResourceBundle extends ResourceBundle {
    Hashtable bundle = null;
    public MyResourceBundle() {
```

```
            bundle = new Hashtable();
            bundle.put("Bundle description","Default resource bundle");
            bundle.put("Behavior",new EnglishBehavior());
          }
          public Enumeration getKeys() {
            return bundle.keys();
          }
          protected Object handleGetObject(String key) throws
              java.util.MissingResourceException {
            return bundle.get(key);
          }
        }
        public class Using {
          public Using() {
          }
          public static void main(String[] args){
            Using u = new Using();
            ResourceBundle rb =
            ResourceBundle.getBundle("lecture.MyResourceBundle",
Locale.getDefault());
            System.out.println((String)rb.getObject("Bundle description"));
            Behavior be = (Behavior)rb.getObject("Behavior");
            System.out.println(be.getBehavior());
            System.out.println(be.getCapital());
            rb = ResourceBundle.getBundle("lecture.MyResourceBundle", new
Locale("en","EN"));
            System.out.println((String)rb.getObject("Bundle description"));
            Behavior be = (Behavior)rb.getObject("Behavior");
            System.out.println(be.getBehavior());
            System.out.println(be.getCapital());
          }
```

Приклад 14.30. (html, txt)

Результатом буде:

```
Російський набір ресурсів
Російський варіант поведінки
Москва
English resource bundle
English behavior
London
```

Приклад 14.31. (html, txt)

Класи ListResourceBundle і PropertiesResourceBundle

У класу ResourceBundle визначене два прямих нащадка ListResourceBundle і PropertiesResourceBundle.PropertiesResourceBundle зберігає набір ресурсів у файлі, який є набором рядків.

Алгоритм конструювання об'єкту, що містить набір ресурсів, був описаний в попередньому параграфі. У всіх випадках, коли як останній елемент використовується .properties, наприклад, baseclass + "_" + language1 + "_" + country1 + ".properties", мова йде про створенні ResourceBundle з файлу з наименованиембaseclass + "_" + language1 + "_" + country1 і розширенням properties. Зазвичай клас ResourceBundleпомещают в пакет resources, а файл властивостей - в каталог resources. Тоді для того, щоб инстанциировать потрібний клас, необхідно вказати повний шлях до цього класу (файлу):

```
getBundle("resources.MyResource",
        Locale.getDefault());
```

ListResourceBundle зберігає набір ресурсів у вигляді колекції і є абстрактним класом. Класи, які успадковують ListResourceBundle, повинні забезпечити:

- перевизначення методу Object[][] getContents(), який повертає масив ресурсів;
- власне двовимірний масив, що містить ресурси.

Розглянемо приклад:

```
public class MyResource extends ListResourceBundle {
  Vector v = new Vector();
  Object[][] resources = {
  {"StringKey","String"},
  {"DoubleKey",new Double(0.0)},
  {"VectorKey",v},
  };
  public MyResource() {
    super();
    v.add("Element 1");
    v.add("Element 2");
    v.add("Element 3");
  }
  protected Object[][] getContents() {
    return resources;
  }
}
public class Test {
  public Test() {
  }
  public static void main(String[] args){
    Test test = new Test();
    ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
    Vector v = (Vector)rb.getObject("VectorKey");
    Iterator it = v.iterator();
    while(it.hasNext()){
```

```
            System.out.println(it.next());
        }
      }
    }
```

Приклад 14.32. (html, txt)

Результатом буде:

```
Element 1
Element 2
Element 3
```

Створення ресурсів для локалей, відмінних від локали за умовчанням, здійснюється так само, як було показано дляResourceBundle.

Висновок

У цій лекції розглянуті допоміжні класи пакету java.util. Як можна було відмітити, вони відносяться до самих різних завдань, а тому рідкісна програма обходиться без використання хоч би одного класу цього пакету.

Нагадаємо коротко всі основні класи і їх особливості:

- Для роботи з датою і часом повинні використовуватися класи Date,Calendar. Клас Calendar абстрактний, існує конкретна реалізація цього класу GregorianCalendar.
- Інтерфейс Observer і клас Observable реалізують парадигму MVC і призначені для повідомлення одного об'єкту про зміну стану іншого.
- Колекції (Collections) не накладають обмежень на порядок проходження і дублювання елементів.
- Списки (List) підтримують порядок елементів (управляються або самими даними, або зовнішніми алгоритмами).
- Набори (Set) не допускають дубльованих елементів.
- Карти (Maps) використовують унікальні ключі для пошуку вмісту.
- Застосування масивів робить додавання, видалення і збільшення кількості елементів скрутним.
- Використання зв'язаних списків (LinkedList) забезпечує хорошу продуктивність при вставці, видаленні елементів, але знижує швидкість індексованого доступу до них.
- Використання дерев (Tree) полегшує вставку, видалення і збільшення розміру сховища, знижує швидкість індексованого доступу, але збільшує швидкість пошуку.
- Застосування хэширования полегшує вставку, видалення і збільшення розміру сховища, знижує швидкість індексованого доступу, але збільшує швидкість пошуку. Проте хэширование вимагає наявність унікальних ключів для запам'ятовування елементів даних.

- Клас Properties зручний для зберігання наборів параметрів у вигляді пар ключ/значение. Параметри можуть зберігатися в потоки (файли) і завантажуватися з них.
- Реалізація класом інтерфейсу Comparator дозволяє порівнювати екземпляри класу один з одним і, відповідно, сортувати їх, наприклад, в колекціях.
- Arrays є класом-утилітой і забезпечує набір методів, що реалізовують різні прийоми роботи з масивами. Не має конструктора.
- StringTokenizer - допоміжний клас, призначений для розбору рядків на лексеми.
- При необхідності працювати з суттю, представленою у вигляді бітових послідовностей, зручно використовувати клас BitSet.
- Маніпулювати ресурсами, які розрізняються залежно від локалізації, зручно за допомогою классовResourceBundle, ListResourceBundle, PropertiesResourceBundle. Власне локаль задається за допомогою класу Locale.

Удалено: с

Удалено: и

ТЕМА. *Пакет java.io*

План

1. *Стандартні класи пакету java. io.*
2. *Механізм серіалізації об'єктів.*
3. *Робота з файлами.*

Рекомендована література: 1, 3, 5.

Теоретичні відомості

Система ввода/вывода. Потоки даних (stream)

Переважна більшість програм обмінюються даними із зовнішнім світом. Це, безумовно, роблять будь-які мережеві застосування – вони передають і отримують інформацію від інших комп'ютерів і спеціальних пристроїв, підключених до мережі. Виявляється, можна таким самим чином представляти обмін даними між пристроями усередині однієї машини. Так, наприклад, програма може прочитувати дані з клавіатури і записувати їх у файл, або ж навпаки - прочитувати дані з файлу і виводити їх на екран. Таким чином, пристрої, звідки може проводитися прочитування інформації, можуть бути найрізноманітнішими – файл, клавіатура, вхідне мережеве з'єднання і так далі То ж стосується і пристроїв виводу – це може бути файл, екран монітора, принтер, витікаюче мережеве з'єднання і тому подібне Кінець кінцем, всі дані в комп'ютерній системі в процесі обробки передаються від пристроїв введення до пристроїв виводу.

Зазвичай частина обчислювальної платформи, яка відповідає за обмін даними, так і називається – система вводу/виводу. У Java вона представлена пакетом java.io (input/output). Реалізація системи вводу/виводу ускладнюється не тільки широким спектром джерел і одержувачів даних, але ще і різними форматами передачі інформації. Нею можна обмінюватися в двійковому уявленні, символьному або текстовому, із застосуванням деякого кодування (тільки для російської мови їх налічується більше 4 штук), або передавати числа в різних уявленнях. Доступ до даних може потрібно як послідовний (наприклад, прочитування HTML-сторінки), так і довільний (складна робота з декількома частинами одного файлу). Часто для підвищення продуктивності застосовується буферизація.

У Java для опису роботи по вводу/виводу використовується спеціальне поняття потік даних (stream). Потік даних пов'язаний з деяким джерелом, або приймачем, даних, здатним отримувати або надавати інформацію. Відповідно, потоки діляться на вхідні – такі, що читають дані і дані, що виходять – передають (що записують). Введення концепції stream дозволяє відокремити основну логіку програми, інформацією, що обмінюється, з будь-якими пристроями однаковим чином, від низькорівневих операцій з такими пристроями вводу/виводу.

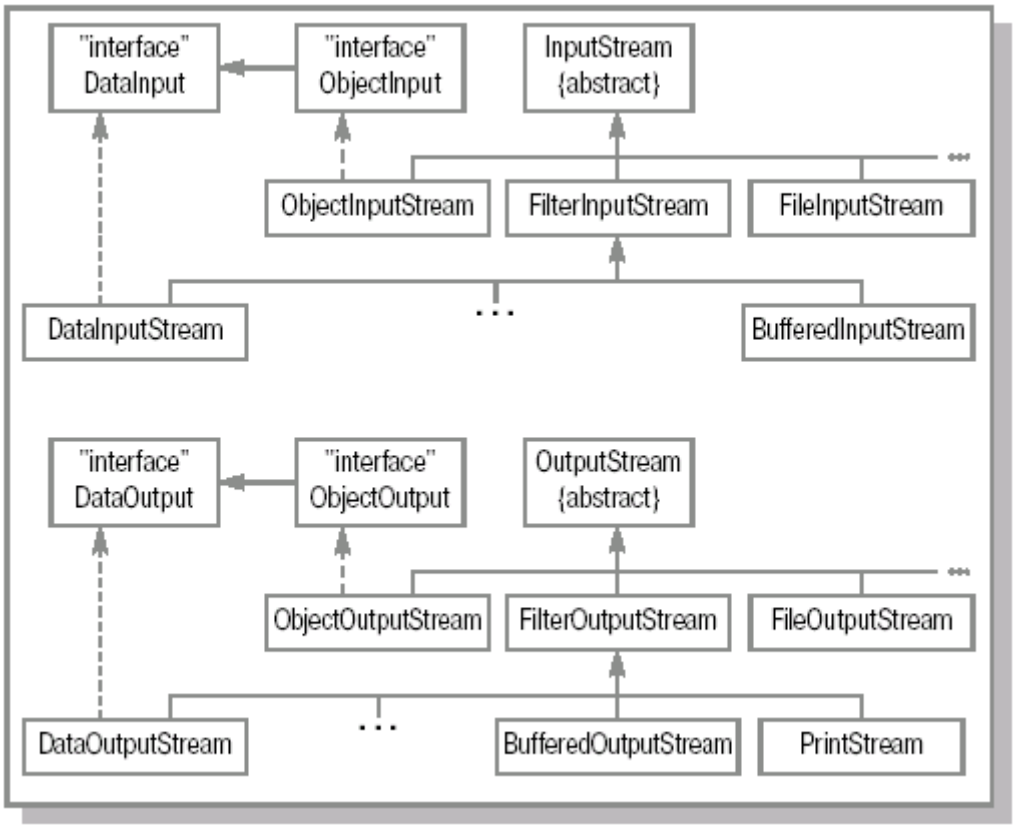
У Java потоки природним чином представляються об’єктами. Класи, що описують їх, якраз і складають основну частину пакету java.io. Вони досить різноманітні і відповідають за різну функціональність. Всі класи розділені на дві частини – одні здійснюють введення даних, інші – вивід.

Існуючі стандартні класи допомагають вирішити більшість типових завдань. Мінімальною "порцією" інформації є, як відомо, битий, що набуває значення 0 або 1 (це поняття також зручно застосовувати на найнижчому рівні, де дані передаються електричним сигналом; умовно кажучи, 1 представляється проходженням імпульсу, 0 – його відсутністю). Традиційно використовується крупніша одиниця вимірювання – байт, об’єднуюча 8 битий. Таким чином, значення, представлене одним байтом, знаходиться в діапазоні від 0 до 28-1=255, або, якщо використовувати знак, – від -128 до +127. Примітивний тип byte в Java в точності відповідає останньому – знаковому діапазону.

Базові, найбільш універсальні, класи дозволяють прочитувати і записувати інформацію саме у вигляді набору байт. Щоб їх було зручно застосовувати в різних завданнях, java.io містить також класи, що перетворюють будь-які дані в набір байт.

Наприклад, якщо потрібно зберегти результати обчислень – набір значень типу double – у файл, то їх можна спочатку перетворити на набір байт, а потім ці байти записати у файл. Аналогічні дії здійснюються і за ситуації, коли потрібно зберегти об’єкт (тобто його стан) – перетворення в набір байт і подальша їх запис у файл. Зрозуміло, що при відновленні даних в обох розглянутих випадках проробляються зворотні дії – спочатку прочитується послідовність байт, а потім вона перетвориться в потрібний формат.

На малюнку 15.1 представлені ієрархії класів вводу/виводу. Як і мовилося, всі типи поділені на дві групи. Представляючі вхідні потоки класи успадковуються від InputStream, а вихідні – від OutputStream.



Мал. 15.1. Ієрархія класів вводу/виводу.

Класи InputStream і OutputStream

InputStream – це базовий клас для потоків введення, тобто читання. Відповідно, він описує базові методи для роботи з байтовими потоками даних. Ці методи необхідні всім класам, які успадковуються отInputStream.

Проста операція представлена методом read() (без аргументів). Він є абстрактним і, відповідно, має бути визначений в класах-спадкоємцях. Цей метод призначений для прочитування рівно одного байта із потоку, проте повертає при цьому значення типу int. В тому випадку, якщо прочитування відбулося успішно, значення, що повертається, лежить в діапазоні від 0 до 255 і є отриманим байтом (значення int має 4 байти і виходить простим доповненням нулями в двійковому уявленні). Звернете увагу, що отриманий таким чином байт не володіє знаком і не знаходиться в діапазоні від -128 до +127, як примітивний тип byte в Java.

Якщо досягнутий кінець потоку, тобто в нім більше немає інформації для читання, то значення, що повертається, рівне -1.

Якщо ж рахувати з потоку дані не вдається із-за якихось помилок, або збоїв, буде кинуто виключення java.io.IOException. Цей клас успадковується від Exception, тобто його завжди необхідно обробляти явно. Річ у тому, що канали передачі інформації, будь то Internet або, наприклад, жорсткий диск, можуть давати збої незалежно від того, наскільки добре написана програма. А це означає, що потрібно бути готовим до них, щоб користувач не втратив потрібні дані.

Метод read() – це абстрактний метод, але саме з дотриманням всіх вказаних умов він має бути реалізований в класах-наслідниках.

На практиці зазвичай доводиться прочитувати не один, а відразу декілька байт – тобто масив байт. Для цього використовується метод read(), де як параметри передається масив byte[]. При виконанні цього методу в циклі проводиться виклик абстрактного методу read() (визначеного без параметрів) і результатами заповнюється переданий масив. Кількість байт, яке прочитане таким чином, дорівнює довжині переданого масиву. Але при цьому може так вийти, що дані в потоці закінчаться ще до того, як буде заповнений весь масив. Тобто можлива ситуація, коли в потоці даних (байт) міститься менше, ніж довжина масиву. Тому метод повертає значення int, яке вказує, скільки байт було реально пораховано. Зрозуміло, що це значення може бути від 0 до величини довжини переданого масиву.

Якщо ж ми спочатку хочемо заповнити не весь масив, а тільки його частина, то для цих цілей використовується метод read(), якому, окрім масиву byte[], передаються ще два int значення. Перше – це позиція в масиві, з якою слід почати заповнення, друге, – кількість байт, яке потрібно рахувати. Такий підхід, коли для отримання даних передається масив і два int числа – offset (зсув) і length (довжина), є досить поширеним і часто зустрічається не тільки в пакеті java.io.

При виклику методів read() можливе виникнення такої ситуації, коли запрошувані дані ще не готові до прочитування. Наприклад, якщо ми прочитуємо дані, що поступають з мережі, і вони ще просто не прийшли. У такому разі не можна сказати, що даних більше немає, але і вважати теж нічого - виконання зупиняється на виклику методу read() і виходить "зависання".

Щоб дізнатися, скільки байт в потоці готово до прочитування, застосовується метод available(). Цей метод повертає значення типу int, яке показує, скільки байт в потоці готово до прочитування. При цьому не варто плутати кількість байт, готових до прочитування, з тією кількістю байт, які взагалі можна буде рахувати з цього потоку. Метод available() повертає число – кількість байт, саме на даний момент готових до прочитування.

Коли робота з вхідним потоком даних закінчена, його слід закрити. Для цього викликається метод close(). Цим викликом будуть звільнені всі системні ресурси, пов'язані з потоком.

Точно так, як і InputStream – це базовий клас для потоків введення, клас OutputStream – це базовий клас для потоків виводу.

У класі OutputStream аналогічним чином визначаються три методи write() – що один приймає як параметр int, другий, – byte[] і третій – byte[], плюс два int-числа. Всі ці методи нічого не повертають (void).

Метод write(int) є абстрактним і має бути реалізований в класах-наслідниках. Цей метод приймає як параметр int, але реально записує в потік тільки byte – молодші 8 битий в двійковому уявленні. Останні 24 біта будуть проігноровані. У разі виникнення помилки цей метод кидає java.io.IOException, як, втім, і більшість методів, пов'язаних з вводом-виводом.

Для запису в потік відразу деякої кількості байт методу write() передається масив байт. Або, якщо ми хочемо записати тільки частину масиву, то передаємо масив byte[] і два int-числа – відступ і кількість байт для запису. Зрозуміло, що якщо вказати невірні параметри – наприклад, негативний відступ, негативна кількість байт для запису, або якщо сума відступ плюс довжина буде більше довжини масиву, – у всіх цих випадках кидається виключення IndexOutOfBoundsException.

Реалізація потоку може бути такій, що дані записуються не відразу, а зберігаються якийсь час в пам'яті. Наприклад, ми хочемо записати у файл якісь дані, які отримуємо порціями по 10 байт, і так 200 разів підряд. У такому разі замість 200 звернень до файлу зручніше буде скопити всі ці дані в пам'яті, а потім одним заходом записати все 2000 байт. Тобто клас вихідного потоку може використовувати деякий внутрішній механізм для буферизації (тимчасового зберігання перед відправкою) даних. Щоб переконатися, що дані записані в потік, а не зберігаються в буфері, викликається метод flush(), визначений в OutputStream. У цьому класі його реалізація порожня, але якщо який-небудь із спадкоємців використовує буферизацію даних, то цей метод має бути в нім перевизначений.

Коли робота з потоком закінчена, його слід закрити. Для цього викликається метод close(). Цей метод спочатку звільняє буфер (викликом методу flush), після чого потік закривається і звільняються всі пов'язані з ним системні ресурси. Закритий потік не може виконувати операції виводу і не може бути відкритий наново. У классеOutputStream реалізація методу close() не проводить ніяких дій.

Отже, класи InputStream і OutputStream визначають необхідні методи для роботи з байтовими потоками даних. Ці класи є абстрактними. Їх завдання – визначити загальний інтерфейс для класів, які отримують дані з різних джерел. Такими джерелами можуть бути, наприклад, масив байт, файл, рядок і так далі Всі вони, або, принаймні, найбільш поширені, будуть розглянуті далі.

Класи-реалізації потоків даних

Класи ByteArrayInputStream і ByteArrayOutputStream

Найприродніший і простіший джерело, звідки можна прочитувати байти, – це, звичайно, масив байт. Клас bytearrayinputstream представляє потік, що прочитує дані з масиву байт. Цей клас має конструктор, якому як параметр передається масив byte[]. Відповідно, при виклику методів read()що повертаються дані братимуться саме з цього масиву. Наприклад:

```
byte[] bytes = {1-1,0};
ByteArrayInputStream in =
  new ByteArrayInputStream(bytes);
int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: "
                + readedInt);
readedInt = in.read();
// readedInt=255. Проте
// (byte)readedInt дасть значення -1
```

```
System.out.println("second element read is: "
        + readedInt);
readedInt = in.read(); // readedInt=0
System.out.println("third element read is: "
        + readedInt);
```

Якщо запустити таку програму, на екрані відобразиться наступне:

```
first element read is: 1
second element read is: 255
third element read is: 0
```

При виклику методу read() дані прочитувалися з масиву bytes, переданого в конструктор ByteArrayInputStream. Звернете увагу, в даному прикладі друге лічене значення дорівнює 255, а не -1, як можна було б чекати. Щоб зрозуміти, чому це відбулося, потрібно пригадати, що метод read прочитує byte, але повертає значення int, отримане додаванням необхідного числа нулів (у двійковому уявленні). Байт, рівний -1, у двійковому уявленні має вигляд 11111111 і, відповідно, число типу int, що отримується приставкою 24-х нулів, рівне 255 (у десятковій системі). Проте якщо явно привести його до byte, набудемо початкового значення.

Аналогічно, для запису байт в масив застосовується клас ByteArrayOutputStream. Цей клас використовує усередині себе об'єкт byte[], куди записує дані, що передаються при виклику методів write(). Щоб отримати записані в масив дані, викликається метод toByteArray(). Приклад:

```
ByteArrayOutputStream out =
  new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] bytes = out.toByteArray();
```

В даному прикладі в результаті масив bytes складатиметься з двох елементів: 10 і 11.

Використовувати класи ByteArrayInputStream і ByteArrayOutputStream може бути дуже зручно, коли потрібно перевірити, що саме записується у вихідний потік. Наприклад, при відладці і тестуванні складних процесів запису і читання з потоків. Ці класи хороші тим, що дозволяють відразу проглянути результат і не потрібно створювати ні файл, ні мережеве з'єднання, ні що-небудь ще.

Класи FileInputStream і FileOutputStream

Клас FileInputStream використовується для читання даних з файлу. Конструктор такого класу як параметр приймає назву файлу, з якого проводитиметься прочитування. При вказівці рядка імені файлу потрібно враховувати, що вона буде безпосередньо передана операційній системі, тому формат імені файлу і шляху до нього може розрізнятися на різних платформах. Якщо при виклику цього конструктора передати рядок, який вказує на неіснуючий файл або каталог, то буде кинуто java.io.FileNotFoundException. Якщо ж об'єкт успішно створений, то при виклику його методів read() значення, що повертаються, прочитуватимуться з вказаного файлу.

Для запису байт у файл використовується клас FileOutputStream. При створенні об'єктів цього класу, тобто при викликах його конструкторів, окрім імені файлу, також можна вказати, чи будуть дані дописуватися в кінець файлу, або файл буде перезаписаний. Якщо вказаний файл не існує, то відразу після создання FileOutputStream він буде створений. При викликах методів write() значення, які передані записуватимуться в цей файл. Після закінчення роботи необхідно викликати метод close(), щоб повідомити систему, що робота по запису файлу закінчена. Приклад:

```
byte[] bytesToWrite = {1, 2, 3};
byte[] bytesReaded = new byte[10];
String fileName = "d:\\test.txt";
try {
  // Створити вихідний потік
  FileOutputStream outFile = new FileOutputStream(fileName);
  System.out.println("Файл відкритий для запису");

  // Записати масив
  outFile.write(bytesToWrite);
  System.out.println("Записано: " + bytesToWrite.length + " байт");

  // Після закінчення використання має бути закритий
  outFile.close();
  System.out.println("Вихідний потік закритий");

  // Створити вхідний потік
  FileInputStream inFile = new FileInputStream(fileName);
  System.out.println("Файл відкритий для читання");

  // Дізнатися, скільки байт готово до прочитування
  int bytesAvailable = inFile.available();
  System.out.println("Готово до прочитування: " + bytesAvailable + " байт");

  // Вважати в масив
  int count = inFile.read(bytesReaded,0,bytesAvailable);
  System.out.println("Лічено: " + count + " байт");
  for (i=0;i<count;i++)
    System.out.print(bytesReaded[i]+",");
  System.out.println();
  inFile.close();
  System.out.println("Вхідний потік закритий");
} catch (FileNotFoundException e) {
  System.out.println("Неможливо провести запис у файл: " + fileName);
} catch (IOException e) {
  System.out.println("Помилка ввода/вывода: " + e.toString());
}
```

Приклад 15.1. (html, txt)

Результатом роботи програми буде:

```
Файл відкритий для запису
Записано: 3 байт
Вихідний потік закритий
Файл відкритий для читання
Готово до прочитування: 3 байт
Лічено: 3 байт
1,2,3,
Вхідний потік закритий
```

Приклад 15.2. (html, txt)

При роботі з FileInputStream метод available() практично напевно поверне довжину файлу, тобто число байт, скільки взагалі з нього можна вважати. Але не варто закладатися на це при написанні програм, які повинні стійко працювати на різних платформах, - метод available() повертає число байт, яке може бути на даний момент лічене без блокування. Той факт, що, швидше за все, це число і буде довжиною файлу, є всього лише окремим випадком роботи на деяких платформах.

У приведеному прикладі для наочності закриття потоків проводилося відразу ж після закінчення їх використання в основному блоці. Проте краще закривати потоки в finally блоці.

```
...
} finally {
try{inFile.close();}catch(IOException e){};
}
```

Такий підхід гарантує, що потік буде закритий і будуть звільнені всі пов'язані з ним системні ресурси.

PipedInputStream і PipedOutputStream

Класи PipedInputStream і PipedOutputStream характеризуються тим, що їх об'єкти завжди використовуються в парі – до одного об'єкту PipedInputStream прив'язується (підключається) один об'єкт PipedOutputStream. Вони можуть бути корисні, якщо в програмі необхідно організувати обмін даними між модулями (наприклад, междупотоками виконання).

Ці класи застосовуються таким чином: створюється по об'єкту PipedInputStream і PipedOutputStream, після чого вони можуть бути сполучені між собою. Один об'єкт PipedOutputStream може бути сполучений з рівно одним об'єктом PipedInputStream, і навпаки. Потім в об'єкт PipedOutputStream записуються дані, після чого вони можуть бути лічені саме в підключеному об'єкті PipedInputStream. Таке з'єднання можна забезпечити або викликом методу connect() з передачею відповідного об'єкту PipedI/OStream (будемо так кратно позначати пару класів, в даному випадку PipedInputStream і PipedOutputStream), або передати цей об'єкт ще при виклику конструктора.

Використання зв'язки PipedInputStream і PipedOutputStream показане в наступному прикладі:

```
try {
  int countRead = 0;
  byte[] toRead = new byte[100];
  PipedInputStream pipeIn = new PipedInputStream();
  PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);

  // Прочитувати в масив, поки він повністю не буде заповнений
  while(countRead<toRead.length){

    // Записати в потік деяку кількість байт
    for(int i=0; i<(Math.random()*10); i++){
      pipeOut.write((byte)(Math.random()*127));
    }

    // Рахувати з потоку доступні дані
    // додати їх вже ліченим.
    int willRead = pipeIn.available();
    if(willRead+countRead>toRead.length)

    //Потрібно вважати тільки до межі масиву
    willRead = toRead.length-countRead;
    countRead += pipeIn.read(toRead, countRead, willRead);
  }
} catch (IOException e) {
  System.out.println ("Impossible IOException occur: ");
  e.printStackTrace();
}
```

Приклад 15.3. (html, txt)

Даний приклад носить чисто демонстративний характер (в результаті його роботи масив toRead буде заповнений випадковими числами). Явніше вигода від використання PipedI/OStream в основному виявляється при розробці багатопотокового застосування. Якщо в програмі запускається декілька потоків виконання, організувати передачу даних між ними зручно за допомогою цих класів. Для цього потрібно створити зв'язані об'єкти PipedI/OStream, після чого передати посилання на них у відповідні потоки. Потік виконання, в якому проводиться читання даних, може містити подібний код:

```
// inStream - об'єкт класу PipedInputStream
try {
  while(true){
  byte[] readedBytes = null;
  synchronized(inStream){
    int bytesAvailable = inStream.available();
    readedBytes = new byte[bytesAvailable];
    inStream.read(readedBytes);
  }
  // обробка отриманих даних з readedBytes
  // .
} catch(IOException e){
```

```
/* IOException буде кинуте, коли потік inStream, або
пов'язаний з ним PipedOutputStream, вже закритий, і при цьому
проводиться спроба прочитування з inStream */

System.out.println("робота з потоком inStream завершена");
}
```

Приклад 15.4. (html, txt)

Якщо з об'єктом inStream одночасно можуть працювати декілька потоків виконання, то необхідно використовувати блок synchronized (як і зроблено в прикладі), який гарантує, що в період між вызовамиinStream.available() і inStream.read(.) ні у якому іншому потоці виконання не проводитиметься прочитування з inStream. Тому виклик inStream.read(readedBytes) не приведе до блокування і всі дані, готові до прочитування, будуть лічені.

StringBufferInputStream

Іноді буває зручно працювати з текстовим рядком String як з потоком байт. Для цього можна скористатися класом StringBufferInputStream. При створенні об'єкту цього класу необхідно передати конструктору об'єкт String. Дані, що повертаються методом read(), прочитуватимуться саме з цього рядка. При цьому символи перетворюватимуться в байти з втратою точності – старший байт відкидається (нагадаємо, що символ char складається з двох байт).

SequenceInputStream

Клас SequenceInputStream об'єднує потік даних з інших два і більш вхідних потоків. Дані вичитуватимуться послідовно – спочатку всі дані з першого потоку в списку, потім з другого, і так далі. Кінець потоку SequenceInputStream буде досягнутий тільки тоді, коли буде досягнутий кінець потоку, останнього в списку.

У цьому класі є два конструктори – що приймає два потоки і приймаючий Enumeration (у якому, звичайно, мають бути тільки екземпляри InputStream і його спадкоємців). Коли викликається метод read(),SequenceInputStream намагається рахувати байт з поточного вхідного потоку. Якщо в нім більше за дані немає (лічене з нього значення рівне -1), у нього викликається метод close() і наступний вхідний потік стає поточним. Так продовжується до тих пір, поки не будуть отримані всі дані з останнього потоку. Якщо при прочитуванні виявляється, що більше вхідних потоків немає, SequenceInputStream повертає -1. Виклик методаclose() у SequenceInputStream закриває вхідні потоки, що все містяться в нім.

Приклад:

```
FileInputStream inFile1 = null;
FileInputStream inFile2 = null;
SequenceInputStream sequenceStream = null;
FileOutputStream outFile = null;
try {
  inFile1 = new FileInputStream("file1.txt");
  inFile2 = new FileInputStream("file2.txt");
```

```
    sequenceStream = new SequenceInputStream(inFile1, inFile2);
    outFile = new FileOutputStream("file3.txt");
    int readedByte = sequenceStream.read();
    while(readedByte!=-1){
      outFile.write(readedByte);
      readedByte = sequenceStream.read();
    }
  } catch (IOException e) {
    System.out.println("IOException: " + e.toString());
  } finally {
    try{sequenceStream.close();}catch(IOException e){};
    try{outFile.close();}catch(IOException e){};
  }
```

Приклад 15.5. (html, txt)

В результаті виконання цього прикладу у файл file3.txt буде записано вміст файлів file1.txt иfile2.txt – спочатку повністю file1.txt, потім file2.txt. Закриття потоків проводиться в блоці finally. Оскільки при виклику методу close() може виникнути IOException, необхідний try-catch блок. Причому, кожен виклик методу close() узятий в окремий try-catch блок - для того, щоб виникле виключення при закритті одного потоку не перешкодило закриттю іншого. При цьому немає необхідності закривати потоки inFile1 і inFile2 – вони будуть автоматично закриті при використанні в sequnceStream - або коли в них закінчаться дані, або при виклику у sequenceStream методу close().

Об'єкт SequenceInputStream можна було створити і іншим способом: спочатку отримати об'єкт Enumeration, що містить всі потоки, і передати його в конструктор SequenceInputStream:

```
Vector vector = new Vector();
vector.add(new StringBufferInputStream("Begin file1\n"));
vector.add(new FileInputStream("file1.txt"));
vector.add(new StringBufferInputStream("\nEnd of file1, begin file2\n"));
vector.add(new FileInputStream("file2.txt"));
vector.add(new StringBufferInputStream("\nEnd of file2"));
Enumeration en = vector.elements();
sequenceStream = new SequenceInputStream(en);
```

Приклад 15.6. (html, txt)

Якщо замінити в попередньому прикладі ініціалізацію sequenceStream на приведену тут, то у файл file3.txt, окрім вмісту файлів file1.txt і file2.txt, будуть записаний ще три рядки – одна на початку файлу, одна між вмістом файлів file1.txt і file2.txt і ще одна в кінці file3.txt.

Класи FilterInputStream і FilterOutputStream і їх спадкоємці

Завдання, що виникають при вводу/виводу вельми різноманітні, - це може бути прочитування байт з файлів, об'єктів з файлів, об'єктів з масивів, прочитування рядків, що буферизує, з масивів і так далі У такій ситуації рішення з використанням простого

спадкоємства приводить до виникнення дуже великого числа підкласів. Ефективніше застосування надбудов (у ООП цей шаблон називається адаптер) Надбудови – накладення додаткових об'єктів для набуття нових властивостей і функцій. Таким чином, необхідно створити декілька додаткових об'єктів – адаптерів до класів вводу/виводу. У java.io їх ще називають фільтрами. При цьому надстройка-фильтр включає інтерфейс об'єкту, на який надбудовується, тому може бути, у свою чергу, додатково надбудована.

У java.io інтерфейс для таких надбудов вводу/виводу надають класи FilterInputStream (для вхідних потоків) і FilterOutputStream(для вихідних потоків). Ці класи успадковані від основних базових класів вводу/виводу – InputStream і OutputStream, відповідно. Конструктор FilterInputStream приймає як параметр об'єкт InputStream і має модифікатор доступу protected.

Класи FilterI/OStream є базовими для надбудов і визначають загальний інтерфейс для об'єктів, що надбудовуються. Потоки-надстройки не є джерелами даних. Вони лише модифікують (розширюють) роботу потоку, що надбудовується .

BufferedInputStream і BufferedOutputStream

На практиці при прочитуванні із зовнішніх пристроїв введення даних майже завжди необхідне буферизувати. Для буферизації даних служать классыBufferedInputStream і BufferedOutputStream.

BufferedInputStream містить масив байт, який служить буфером для прочитуваних даних. Тобто коли байти з потоку прочитуються або пропускаються (метод skip()), спочатку заповнюється буферний масив, причому, з потоку , що надбудовується , завантажується відразу багато байт, щоб не потрібно було звертатися до нього при кожній операції read або skip. Також клас BufferedInputStream додає підтримку методів mark() і reset(). Ці методи визначені ще в класі InputStream, але там їх реалізація за замовчуванням кидає виключення IOException. Метод mark() запам'ятовує крапку у вхідному потоці, а виклик методу reset() приводить до того, що всі байти, отримані після останнього виклику mark(), прочитуватимуться повторно, перш, ніж нові байти почнуть поступати з надбудованого вхідного потоку.

BufferedOutputStream надає можливість проводити багатократний запис невеликих блоків даних без звернення до пристрою виводу при записі кожного з них. Спочатку дані записуються у внутрішній буфер. Безпосереднє звернення до пристрою виводу і, відповідно, запис в нього, відбудеться, коли буфер заповниться. Ініціювати передачу вмісту буфера на пристрій виводу можна і явним чином, викликавши метод flush(). Так само буфер звільняється перед закриттям потоку. При цьому буде закритий і потік, що надбудовується (так само поступає BufferedInputStream).

Наступний приклад наочно демонструє підвищення швидкості прочитування даних з файлу з використанням буфера:

```
try {
  String fileName = "d:\\file1";
  InputStream inStream = null;
  OutputStream outStream = null;
```

```
  //Записати у файл деяку кількість байт
  long timeStart = System.currentTimeMillis();
  outStream = new FileOutputStream(fileName);
  outStream = new BufferedOutputStream(outStream);
  for(int i=1000000; --i>=0;){
    outStream.write(i);
  }
  long time = System.currentTimeMillis() - timeStart;
  System.out.println("Writing time: " + time + " millisec");
  outStream.close();

  // Визначити час прочитування без буферизації
  timeStart = System.currentTimeMillis();
  inStream = new FileInputStream(fileName);
  while(inStream.read()!=-1){
  }
  time = System.currentTimeMillis() - timeStart;
  inStream.close();
  System.out.println("Direct read time: " + (time) + " millisec");

  // Тепер застосуємо буферизацію
  timeStart = System.currentTimeMillis();
  inStream = new FileInputStream(fileName);
  inStream = new BufferedInputStream(inStream);
  while(inStream.read()!=-1){
  }
  time = System.currentTimeMillis() - timeStart;
  inStream.close();
  System.out.println("Buffered read time: " + (time) + " millisec");
} catch (IOException e) {
  System.out.println("IOException: " + e.toString());
  e.printStackTrace();
}
```

Приклад 15.7. (html, txt)

Результатом можуть бути, наприклад, такі значення:

```
Writing time: 359 millisec
Direct read time: 6546 millisec
Buffered read time: 250 millisec
```

Приклад 15.8. (html, txt)

В даному випадку не проводилися ніяких додаткових обчислень, що займають процесорний час, тільки запис і прочитування з файлу. При цьому прочитування з використанням буфера зайняло в 10 (!) разів менше часу, чим аналогічне без буферизації. Для швидшого виконання програми запис у файл проводився з буферизацією, проте її вплив на швидкість запису неважко перевірити, прибравши з програми рядок, створючи BufferedOutputStream.

Класи BufferedI/OStream додають тільки внутрішню логіку обробки запитів, але не додають ніяких нових методів. Наступні два фільтри надають деякі додаткові можливості для роботи спотоками.

LineNumberInputStream

Клас LineNumberInputStream під час читання даних робить підрахунок, скільки рядків було прочитано з потоку. Номер рядка, на якому в даний момент відбувається читання, можна дізнатися шляхом виклику методагetLineNumber(). Також можна і перейти до певного рядка викликом методу setLineNumber(int lineNumber).

Під рядком при цьому розуміється набір байт, що закінчується або '\n', або '\r', або їх комбінацією '\r\n', саме в цій послідовності.

Аналогічний клас для витікаючого потоку відсутній. LineNumberInputStream, починаючи з версії 1.1, об'явлений deprecated, тобто використовувати його не рекомендується. Його замінив клас LineNumberReader (розглядається нижчим), принцип роботи якого такий самий.

PushBackInputStream

Цей фільтр дозволяє повернути у вхідний потік лічені з нього дані. Така дія проводиться викликом методу unread(). Зрозуміло, що забезпечується подібна функціональність за рахунок наявності в класі спеціального буфера – масиву байт, який зберігає лічену інформацію. Якщо буде проведений відкіт (викликаний метод unread), то під час наступного прочитування ці дані видаватимуться ще раз як тільки отримані. При створенні об'єкту можна вказати розмір буфера.

PrintStream

Цей клас використовується для конвертації і запису рядків в байтовий потік. У нім визначений метод print(.), що приймає як аргумент різні примітивні типи Java, а також тип Object. При виклику передавані дані будуть спочатку перетворені в рядок викликом методу String.valueOf(), після чого записані в потік. Якщо виникає виключення, воно обробляється усередині методу print і далі не кидається (дізнатися, чи відбулася помилка, можна за допомогою методу checkError()). При записі символів у вигляді байт використовується кодування, прийняте за умовчанням в операційній системі (є можливість задати її явно при запуску JVM).

Цей клас також є deprecated, оскільки робота з кодуваннями вимагає особливого підходу (часто у двобайтових символів Java старший байт просто відкидається). Тому у версії Java 1.1 з'явився додатковий набір класів, що грунтується на типах Reader і Writer. Вони будуть розглянуті пізніше. Зокрема, замість PrintStream тепер рекомендується застосовувати PrintWriter. Проте старий клас продовжує активно використовуватися, оскільки статичні поля out і err класу System мають саме це тип.

DataInputStream і DataOutputStream

До цих пір мова йшла тільки про прочитування і запис в потік даних у вигляді byte. Для роботи з іншими примітивними типами даних Java визначені

інтерфейси DataInput і DataOutput і їх реалізації – классы-фильтры DataInputStream і DataOutputStream. Їх місце в ієрархії класів вводу/виводу можна побачити нарис.15.1.

Інтерфейси DataInput і DataOutput визначають, а класи DataInputStream і DataOutputStream, відповідно, реалізують методи прочитування і запису значень всіх примітивних типів. При цьому відбувається конвертація цих даних в набір byte і назад. Читання необхідно організувати так, щоб дані запрошувалися у вигляді тих же типів, в тій же послідовності, як і проводився запис. Якщо записати, наприклад, int і long, а потім прочитувати їх як short, читання буде виконано коректно, без виняткових ситуацій, але числа будуть отримані зовсім інші.

Це наочно показано в наступному прикладі:

```
try {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream outData = new DataOutputStream(out);
  outData.writeByte(128);
  // цей метод приймає аргумент int, але записує
  // лише молодший байт
  outData.writeInt(128);
  outData.writeLong(128);
  outData.writeDouble(128);
  outData.close();
  byte[] bytes = out.toByteArray();
  InputStream in = new ByteArrayInputStream(bytes);
  DataInputStream inData = new DataInputStream(in);
  System.out.println("Читання в правильній послідовності: ");
  System.out.println("readByte: " + inData.readByte());
  System.out.println("readInt: " + inData.readInt());
  System.out.println("readLong: " + inData.readLong());
  System.out.println("readDouble: " + inData.readDouble());
  inData.close();
  System.out.println("Читання в зміненій послідовності:");
  in = new ByteArrayInputStream(bytes);
  inData = new DataInputStream(in);
  System.out.println("readInt: " + inData.readInt());
  System.out.println("readDouble: " + inData.readDouble());
  System.out.println("readLong: " + inData.readLong());
  inData.close();
  } catch (Exception e) {
    System.out.println("Impossible IOException occurs: " +
      e.toString());
  e.printStackTrace();
}
```

Приклад 15.9. (html, txt)

Результат виконання програми:

Читання в правильній послідовності:

```
readByte: -128
readInt: 128
readLong: 128
readDouble: 128.0
```

Читання в зміненій послідовності:

```
readInt: -2147483648
readDouble: -0.0
readLong: -9205252085229027328
```

Отже, значення будь-якого примітивного типу може бути передане і пораховане з потоку даних.

Серіалізация об’єктів (serialization)

Для об’єктів процес перетворення в послідовність байт і назад організований дещо складніше – об’єкти мають різну структуру, зберігають посилання на інші об’єкти і так далі Тому така процедура отримала спеціальну назву - сериализация (serialization), зворотна дія, – тобто відтворення об’єкту з послідовності байт – десериализация.

Оскільки серіалізований об’єкт – це послідовність байт, яку можна легко зберегти у файл, передати по мережі і так далі, то і об’єкт потім можна відновити на будь-якій машині, незалежно від того, де проводилася серіалізація. Зрозуміло, Java дозволяє не замислюватися при цьому про такі чинники, як, наприклад, використовувана операційна система на машині-відправнику і одержувачі. Така гнучкість зумовила широке застосування серіалізації при створенні розподілених застосувань, у тому числі і корпоративних (enterprise) систем.

Стандартна серіалізація

Для представлення об’єктів у вигляді послідовності байт визначені успадковані від DataInput і DataOutput інтерфейси ObjectInput иObjectOutput, відповідно. У java.io є реалізації цих інтерфейсів – класи ObjectInputStream і ObjectOutputStream.

Ці класи використовують стандартний механізм сериализации, який пропонує JVM. Для того, щоб об’єкт міг бути серіалізований, клас, від якого він породжений, повинен реалізовувати інтерфейс java.io.Serializable. У цьому інтерфейсі не визначений жоден метод. Він потрібний лише для вказівки, що об’єкти класу можуть брати участь в серіалізації. При спробі серіалізувати об’єкт, що не має такого інтерфейсу, буде кинутий java.io.NotSerializableException.

Щоб почати серіалізацію об’єкту, потрібний вихідний потік OutputStream, в який і записуватиметься послідовність, що згенерувала, байт. Цей потік передається в конструктор ObjectOutputStream. Потім викликом методу writeObject() об’єкт серіалізуєся і записується у вихідний потік. Наприклад:

```
ByteArrayOutputStream os =
  new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos =
  new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Щоб побачити, на що перетворився об'єкт objSave, можна проглянути вміст масиву:

```
byte[] bArray = os.toByteArray();
```

А щоб відновити об'єкт, його потрібне десеріалізувати з цього масиву:

```
ByteArrayInputStream is =
  new ByteArrayInputStream(bArray);
ObjectInputStream ois =
  new ObjectInputStream(is);
Object objRead = ois.readObject();
```

Тепер можна переконатися, що відновлений об'єкт ідентичний початковому:

```
System.out.println("readed object is: " +
                objRead.toString());
System.out.println("Object equality is: " +
  (objSave.equals(objRead)));
System.out.println("Reference equality is: " +
               (objSave==objRead));
```

Результатом виконання приведеного вище коду буде:

```
readed object is: 1
Object equality is: true
Reference equality is: false
```

Як ми бачимо, відновлений об'єкт не збігається з початковим (що очевидно – адже відновлення могло відбуватися і на іншій машині), але рівний серіалізованому за значенням.

Як завжди, для спрощення в прикладі була опущена обробка помилок. Проте, серіалізація (десеріалізація) об'єктів досить складна процедура, тому виникаючі складнощі не завжди очевидні. Розглянемо основні виключення, які може генерувати метод readObject() класу ObjectInputStream.

Припустимо, об'єкт деякого класу TestClass був серіалізований і переданий по мережі на іншу машину для відновлення. Може трапитися так, що у прочитуючої JVM на локальному диску не опиниться описи цього класу (файл TestClass.class). Оскільки стандартний механізм серіалізації записує в потік байт лише стан об'єкту, для успішної десеріалізації необхідна наявність опис класу. В результаті буде кинуто виключення ClassNotFoundException.

Причина появи java.io.StreamCorruptedException цілком очевидна з назви – неправильний формат вхідного потоку. Припустимо, відбувається спроба рахувати серіалізований об'єкт з файлу. Якщо цей файл зіпсований (для експерименту можна відкрити його в текстовому редакторові і виправити декілька символів), то стандартна процедура десеріалізації дасть збій. Ця ж помилка виникне, якщо рахувати деяку кількість байт (за допомогою методу read) безпосередньо з потоку InputStream, що надбудовується . У такому случаеObjectInputStream знову виявить збій у форматі даних і буде кинуте исключениеjava.io.StreamCorruptedException.

Оскільки ObjectOutput успадковується від DataOutput, ObjectOutputStream може бути використаний для послідовного запису декількох значень як об'єктних, так і примітивних типів в довільній послідовності. Якщо при прочитуванні буде викликаний метод readObject, а в початковому потоці наступним на черзі записано значення примітивного типу, буде кинуто виключення java.io.OptionalDataException. Очевидно, що для коректного відновлення даних з потоку їх потрібно прочитувати саме в тому порядку, в якому були записані.

Відновлення стану

Отже, серіалізація об'єкту полягає в збереженні і відновленні стану об'єкту. У Java в більшості випадків стан описується значеннями полів об'єкту. Причому, що важливе, не тільки тих полів, які були явно оголошені в класі, від якого породжений об'єкт, але і успадкованих полів.

Припустимо, ми б спробували своїми силами реалізувати стандартний механізм серіалізації. Нам передається вихідний потік, в який потрібно записати стан нашого об'єкту. За допомогою DataOutput інтерфейсу можна легко зберегти значення всіх доступних полів (для простоти вважатимемо, що вони всі примітивного типу). Проте в більшості випадків в батьківських класах можуть бути оголошені недоступні нам поля (например,private). Проте, такі поля, як правило, грають важливу роль у визначенні стану об'єкту, оскільки вони можуть впливати на результат роботи успадкованих методів. Як же зберегти їх значення?

З іншого боку, не меншою проблемою є відновлення об'єкту. Як мовилося раніше, об'єкт може бути створений тільки викликом його конструктора. У класу, від якого породжений десеріалізучий об'єкт, може бути декілька конструкторів, причому, деякі з них, або все, можуть мати аргументи. Який з них викликати? Які значення передати як аргументи?

Після створення об'єкту необхідно встановити лічені значення його полів. Проте багато класів мають спеціальні set-методи для цієї мети. У таких методах можуть відбуватися перевірки, можуть мінятися значення допоміжних полів. Чи користуватися цими методами? Якщо їх декілька, то як вибрати правильний і які параметри йому передати? Знову виникає проблема роботи з недоступними полями, отриманими по спадку. Як же в стандартному механізмі серіалізації вирішені всі ці питання?

По-перше, розглянемо докладніше роботу з інтерфейсом Serializable. Відмітимо, що клас Object не реалізує цей інтерфейс. Таким чином, існує два варіанти – або серіалізуючий клас успадковується від Serializable-класу, або немає. Перший варіант

досить простий. Якщо батьківський клас вже реалізував інтерфейс Serializable, то спадкоємцям ця властивість передається автоматично, тобто всі об'єкти, породжені від такого класу, або будь-якого його спадкоємця, можуть бути серіалізовані.

Якщо ж наш клас вперше реалізує Serializable в своїй вітці спадкоємства, то його суперклас повинен відповідати спеціальній вимозі – у нього має бути доступний конструктор без параметрів. Саме за допомогою цього конструктора буде створений десеріалізуючий об'єкт і будуть проініціалізовані всі поля, успадковані від класів, Serializable, що не успадковують .

Розглянемо приклад:

```
// Батьківський клас, не реалізовуючий Serializable
public class Parent {
  public String firstName;
  private String lastName;
  public Parent(){
    System.out.println("Create Parent");
    firstName="old_first";
    lastName="old_last";
  }
  public void changeNames() {
    firstName="new_first";
    lastName="new_last";
  }
  public String toString() {
    return super.toString()+",first="+firstName+",last="+lastName;
  }
}
// Клас Child, вперше Serializable, що реалізував
public class Child extends Parent implements Serializable {
  private int age;
  public Child(int age){
    System.out.println("Create Child");
    this.age=age;
  }
  public String toString() {
    return super.toString()+",age="+age;
  }
}
// Спадкоємець Serializable-класса
public class Child2 extends Child {
  private int size;
  public Child2(int age, int size){
    super(age);
    System.out.println("Create Child2");
    this.size=size;
  }
  public String toString() {
    return super.toString()+",size="+size;
```

```
  }
}
// Клас, що запускається, для тесту
public class Test {
  public static void main(String[] arg){
    try {
      FileOutputStream fos=new FileOutputStream("output.bin");
      ObjectOutputStream oos=new ObjectOutputStream(fos);
      Child c=new Child(2);
      з.changeNames();
      System.out.println(c);
      oos.writeObject(c);
      oos.writeObject(new Child2(3, 4));
      oos.close();
      System.out.println("Read objects:");
      FileInputStream fis=new FileInputStream("output.bin");
      ObjectInputStream ois=new ObjectInputStream(fis);
      System.out.println(ois.readObject());
      System.out.println(ois.readObject());
      ois.close();
    } catch (Exception e) { // спрощена обробка скорочено
      e.printStackTrace();
    }
  }
}
```

Приклад 15.10. (html, txt)

В даному прикладі оголошено 3 класи. Клас Parent не реалізує Serializable і, отже, не може бути серіалізований. У нім оголошено 2 поля, які при створенні набувають значень, що містять слово "old"("старий"). Окрім цього, оголошений метод, що дозволяє модифікувати ці поля. Він виставляє їм значення, що містять слово "new" ("новий'). Також перевизначений метод toString(), щоб дати можливість дізнатися значення цих полів.

Оскільки клас Parent має доступний конструктор за умовчанням, його спадкоємець може реалізувати интерфейсSerializable. Звернете увагу, що у самого класу Child такого конструктора вже немає. Також оголошено поле і модифікований метод toString().

Нарешті, клас Child2 успадковується від Child, а тому автоматично є допустимим для серіалізації. Аналогічно, має нове поле, значення якого відображає toString().

Клас Test , що запускається, серіалізує у файл output.bin два об'єкти. Звернете увагу, що у першого з них заздалегідь викликається метод changeNames(), який модифікує значення полів, успадкованих від класу Parent.

Результат виконання прикладу:

```
Create Parent
Create Child
Child@ad3ba4,first=new_first,last=new_last,age=2
```

```
Create Parent
Create Child
Create Child2
Read objects:
Create Parent
Child@723d7c,first=old_first,last=old_last,age=2
Create Parent
Child2@22c95b,first=old_first,last=old_last,age=3,size=4
```

Приклад 15.11. (html, txt)

У всіх конструкторах вставлений рядок, що виводить повідомлення на консоль. Так можна відстежити, які конструктори викликаються під час десеріалізації. Видно, що для об'єктів, породжених від Serializable-класів, конструктори не викликаються зовсім. Йде звернення лише до конструктора без параметрів не-Serializable-суперкласу.

Порівняємо значення полів першого об'єкту і його копії, отриманою десериализацией. Поля, успадковані від не-Serializable-класу (firstName, lastName), не відновилися. Вони мають значення, отримані в конструкторі Parent без параметрів. Поля, оголошені в Serializable-класу, свої значення зберегли. Це вірно і для другого об'єкту – власні поля Child2 і успадковані від Child мають такі самі значення, що і до серіалізації. Їх значення були записані, а потім лічені і безпосередньо встановлені з потоку даних.

Іноді в класі є поля, які не повинні брати участь в серіалізації. Тому може бути декілька причин. Наприклад, це поле малоістотне (тимчасова змінна) і зберігати його немає необхідності. Якщо серіалізований об'єкт передається по мережі, то виключення такого поля з серіалізації дозволяє зменшити навантаження на мережу і прискорити роботу додатку.

Деякі поля зберігають значення, які не матимуть сенсу при пересилці об'єкту на іншу машину, або при відтворенні його через якийсь час. Наприклад, мережеве з'єднання, або підключення до бази даних, в таких випадках потрібно встановлювати наново.

Потім, в об'єкті може зберігатися конфіденційна інформація, наприклад, пароль. Якщо таке поле буде серіалізовано і передано по мережі, його значення може бути перехоплене і прочитане, або навіть підмінено.

Для виключення поля об'єкту з серіалізації його необхідно оголосити з модифікатором transient. Наприклад, наступний клас:

```
class Account implements
    java.io.Serializable {
  private String name;
  private String login;
  private transient String password;
  /* оголошення інших елементів класу
  ...
  */
}
```

У такого класу поле password в серіалізації брати участь не буде і при відновленні воно набуде значення за умовчанням (в даному випадку null).

Особливої уваги вимагають статичні поля. Оскільки вони належать класу, а не об'єкту, вони не беруть участь в серіалізації. При відновленні об'єкт працюватиме з таким значенням static-поля, яке вже встановлене для його класу в цій JVM.

Граф серіалізації

До цього ми розглядали об'єкти, які мають поля лише примітивних типів. Якщо ж серіалізуючий об'єкт посилається на інші об'єкти, їх також необхідно зберегти (записати в потік байт), а при десеріалізації – відновити. Ці об'єкти, у свою чергу, також можуть посилатися на наступні об'єкти. При цьому важливо, що якщо декілька посилань указують на один і той же об'єкт, то цей об'єкт має бути серіалізований лише одного разу, а при відновленні всі посилання повинні знов указувати на нього одного. Наприклад, серіалізований об'єкт A посилається на об'єкти B і C, кожен з яких, у свою чергу, посилається на один і той же об'єкт D. Після десеріалізації не повинно виникати ситуації, коли B посилається на D1, а C – на D2, де D1 і D2 – рівні, але все таки різні об'єкти.

Для організації такого процесу стандартний механізм серіалізації будує граф, що включає всі об'єкти, що беруть участь, і посилання між ними. Якщо чергове посилання указує на деякий об'єкт, спочатку перевіряється – чи немає такого об'єкту в графі. Якщо є – об'єкт другий раз не серіалізується. Якщо немає – новий об'єкт додається в граф.

При побудові графа може зустрітися об'єкт, породжений від класу, не реалізовуючого интерфейсSerializable. В цьому випадку серіалізація уривається, генерується виключення java.io.NotSerializableException.

Розглянемо приклад:

```
import java.io.*;
class Point implements Serializable {
  double x;
  double y;
  public Point(double x, double y){
    this.x = x;
    this.y = y;
  }
  public String toString() {
    return "("+x+","+y+") reference="+super.toString();
  }
}
class Line implements Serializable {
  Point point1;
  Point point2;
  int index;
  public Line() {
    System.out.println("Constructing empty line");
  }
```

```
  Line(Point p1, Point p2, int index){
    System.out.println("Constructing line: " + index);
    this.point1 = p1;
    this.point2 = p2;
    this.index = index;
  }
  public int getIndex() { return index; }
  public void setIndex(int newIndex) { index = newIndex; }
  public void printInfo() {
    System.out.println("Line: " + index);
    System.out.println(" Object reference: " + super.toString());
    System.out.println(" from point "+point1);
    System.out.println(" to point "+point2);
  }
}
public class Main {
  public static void main(java.lang.String[] args){
    Point p1 = new Point(1.0,1.0);
    Point p2 = new Point(2.0,2.0);
    Point p3 = new Point(3.0,3.0);
    Line line1 = new Line(p1,p2,1);
    Line line2 = new Line(p2,p3,2);
    System.out.println("line 1 = " + line1);
    System.out.println("line 2 = " + line2);
    String fileName = "d:\\file";
    try{
      // записуємо об'єкти у файл
      FileOutputStream os = new FileOutputStream(fileName);
      ObjectOutputStream oos = new ObjectOutputStream(os);
      oos.writeObject(line1);
      oos.writeObject(line2);
      // міняємо стан line1 і записуємо його ще раз
      line1.setIndex(3);
      //oos.reset();
      oos.writeObject(line1);
      // закриваємо потоки
      // досить закрити тільки поток-надстройку
      oos.close();
      // прочитуємо об'єкти
      System.out.println("Read objects:");
      FileInputStream is = new FileInputStream(fileName);
      ObjectInputStream ois = new ObjectInputStream(is);
      for (int i=0; i<3; i++) { // Прочитуємо 3 об'єкти
        Line line = (Line)ois.readObject();
        line.printInfo();
      } ois.close();
    } catch(ClassNotFoundException e){
      e.printStackTrace();
    } catch(IOException e){
      e.printStackTrace();
```

```
    }
  }
}
```

Приклад 15.12. (html, txt)

У цій програмі робота йде з класом Line (лінія), який має 2 поля типу Point (лінія описується двома крапками). Клас Main , що запускається, створює два об'єкти класу Line, причому, одна з крапок у них загальна. Окрім цього, лінія має номер (поле index). Створені лінії (номери 1 і 2) записуються в потік, після чого одна з них отримує новий номер (3) і знов серіалізується.

Виконання цієї програми приведе до виводу на екран приблизно наступного:

```
Constructing line: 1
Constructing line: 2
line 1 = Line@7d39
line 2 = Line@4ec
Read objects:
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
Line: 2
  Object reference: Line@6706
  from point (2.0,2.0) reference=Point@386e
  to point (3.0,3.0) reference=Point@68ae
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
```

Приклад 15.13. (html, txt)

З прикладу видно, що після відновлення у ліній зберігається загальна крапка, що описується одним і тим же об'єктом (хеш-код 386e).

Третій записаний об'єкт ідентичний першому, причому, збігаються навіть об'єктні посилання. Не дивлячись на те, що при записі третього об'єкту значення index було змінене на 3, в десеріалізованнім об'єкті воно залишилося рівним 1. Так відбулося тому, що об'єкт, що описує першу лінію, вже був задіяний в серіалізації і, зустрівшись удруге, повторно записаний не був.

Щоб вказати, що сеанс серіалізації завершений, і дістати можливість передавати змінені об'єкти, у ObjectOutputStream потрібно викликати метод reset(). У даному прикладі для цього досить прибрати коментар в рядку

```
//oos.reset();
```

Якщо тепер запустити програму, то можна побачити, що третій об'єкт отримає номер 3.

```
Constructing line: 1
Constructing line: 2
line 1 = Line@ea2dfe
line 2 = Line@7182c1
Read objects:
Line: 1
  Object reference: Line@a981ca
  from point (1.0,1.0) reference=Point@1503a3
  to point (2.0,2.0) reference=Point@a1c887
Line: 2
  Object reference: Line@743399
  from point (2.0,2.0) reference=Point@a1c887
  to point (3.0,3.0) reference=Point@e7b241
Line: 3
  Object reference: Line@67d940
  from point (1.0,1.0) reference=Point@e83912
  to point (2.0,2.0) reference=Point@fae3c6
```

Приклад 15.14. (html, txt)

Проте це буде вже новий об'єкт, посилання на який відрізняється від першої ліченої лінії. Більш того, обидві крапки також описуватимуться новими об'єктами. Тобто в новому сеансі всі об'єкти були записані, а потім відновлені наново.

Розширення стандартної серіалізації

Деяким складно організованим класам потрібний особливий підхід для серіалізації. Для розширення стандартного механізму можна оголосити в класі два методи точно такою сигнатурою:

```
private void writeObject(
        java.io.ObjectOutputStream out)
  throws IOException;
private void readObject(
        java.io.ObjectInputStream in)
  throws IOException, ClassNotFoundException;
```

Якщо в класі оголошені такі методи, то при серіалізації об'єкту для запису його стану буде визваний writeObject, який повинен згенерувати послідовність байт і записати її в потік out, отриманий як аргумент. При цьому можна викликати стандартний механізм запису об'єкту шляхом виклику методу

```
out.defaultWriteObject();
```

Цей метод запише все не-transient і не-static поля в потік даних.

У свою чергу, при десеріалізації метод readObject повинен рахувати дані з потоку in (також отриманого як аргумент) і відновити значення полів класу. При реалізації цього методу можна звернутися до стандартного механізму за допомогою методу:

```
in.defaultReadObject();
```

Цей метод прочитує опис об'єкту з потоку і привласнює значення відповідних полів в поточному об'єкті.

Якщо ж процедура серіалізації в корені відрізняється від стандартної, то для таких класів призначений альтернативний інтерфейс java.io.Externalizable.

При використанні цього інтерфейсу в потік автоматично записується тільки ідентифікація класу. Зберегти і відновити всю інформацію про стан екземпляра повинен сам клас. Для цього в нім мають бути оголошені методи writeExternal() і readExternal() інтерфейсу Externalizable. Ці методи повинні забезпечити збереження стану, що описується полями самого класу і його суперкласу.

При відновленні Externalizable-об'єкту екземпляр створюється шляхом виклику конструктора без аргументів, після чого викликається метод readExternal.

Метод writeExternal має сигнатуру:

```
void writeExternal(ObjectOutput out)
  throws IOException;
```

Для збереження стану викликаються методи ObjectOutput, за допомогою яких можна записати як примітивні, так і об'єктні значення. Для коректної роботи у відповідному методі

```
void readExternal(ObjectInput in)
  throws IOException,ClassNotFoundException;
```

ці значення мають бути лічені в тому ж самому порядку.

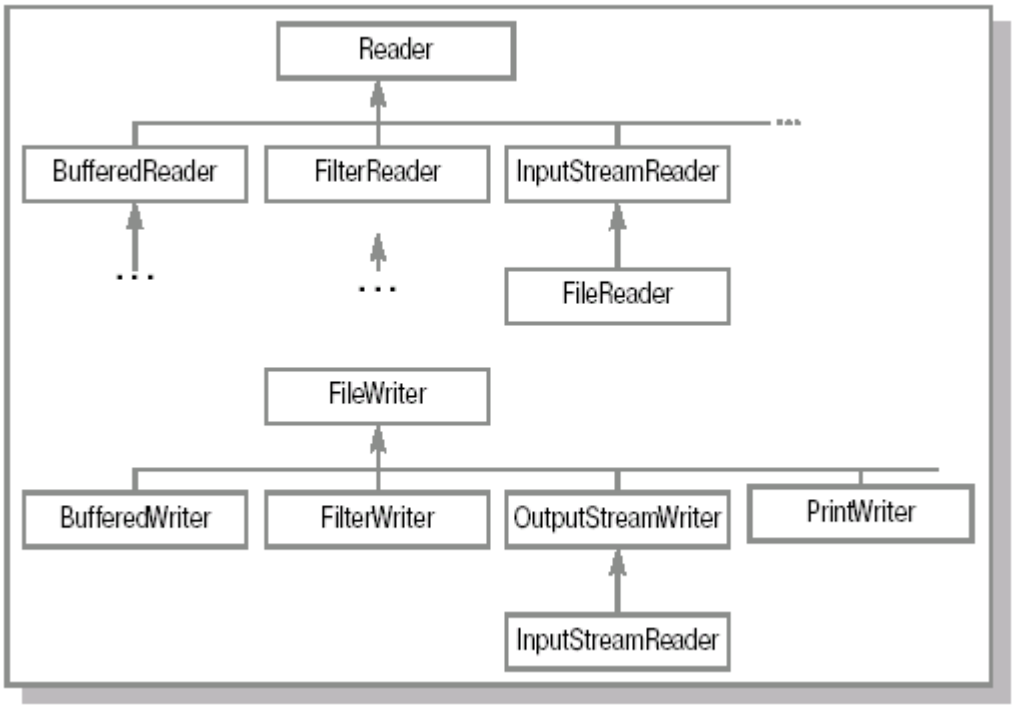
Класи Reader і Writer і їх спадкоємці

Розглянуті класи – спадкоємці InputStream і OutputStream – працюють з байтовими даними. Якщо з їх допомогою записувати або прочитувати текст, то спочатку необхідно зіставити кожному символу його числовий код. Така відповідність називається кодуванням.

Відомо, що Java використовує кодування Unicode, в якому символи представляються двобайтовим кодом. Байтові потоки часто працюють з текстом спрощено – вони просто відкидають старший байт кожного символу. У реальних же застосуваннях можуть використовувати різні кодування (навіть для російської мови їх існує декілька). Тому у версії Java 1.1 з'явився додатковий набір класів, що грунтується на типах Reader і Writer. Їх ієрархія представлена на мал. 15.2.

Ця ієрархія дуже схожа з аналогічною для байтових потоків InputStreamи OutputStream. Головна відмінність між ними – Reader і Writer працюють спотоком символів (char). Тільки читання масиву символів в Readerописывается методом read(char[]), а запис в Writer – write(char[]).

У таблиці 15.1 приведені відповідності класів для байтових і символьныхпотоков.



Мал. 15.2. Ієрархія класів Reader і Writer.

Таблиця 15.1. Відповідність класів для байтових і символьних потоків.

Байтовий потік	Символьний потік
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Немає аналога	InputStreamReader
Немає аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter

BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Немає аналога
DataOutputStream	Немає аналога
ObjectInputStream	Немає аналога
ObjectOutputStream	Немає аналога
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter
StringBufferInputStream	StringReader
Немає аналога	StringWriter
LineNumberInputStream	LineNumberReader
PushBackInputStream	PushBackReader
SequenceInputStream	Немає аналога

Як видно з таблиці, відмінності украй незначні і передбачені.

Наприклад, звичайно ж, відсутнє перетворення в символьне уявлення примітивних типів Java і об'єктів (DataInput/Output, ObjectInput/Output). Додані класи-мости, що перетворюють символьні потокив байтові: InputStreamReader і OutputStreamWriter . Саме на їх основі реалізовані FileReader иFileWriter. Метод available() класу InputStream в класі Reader відсутній, він замінений методом ready(), що повертає булеве значення, – чи готовий потік до прочитування (тобто чи буде прочитування проведено без блокування).

У останньому ж використання символьних потоків ідентичне роботі з байтовими потоками. Так, програмний код для запису символьних даних у файл виглядатиме приблизно таким чином:

```
String fileName = "d:\\file.txt";
```

```
//Рядок, який буде записаний у файл
String data = "Some data to be written and read.\n";
try{
  FileWriter fw = new FileWriter(fileName);
  BufferedWriter bw = new BufferedWriter(fw);
  System.out.println("Write some data to file: " + fileName);

  // Кілька разів записати рядок
  for(int i=(int)(Math.random()*10);--i>=0;)
    bw.write(data);
  bw.close();

  // Прочитуємо результат
  FileReader fr = new FileReader(fileName);
  BufferedReader br = new BufferedReader(fr);
  String s = null;
  int count = 0;
  System.out.println("Read data from file: " + fileName);

  // Прочитувати дані, відображаючи на екран
  while((s=br.readLine())!=null)
    System.out.println("row " + ++count + " read:" + s);
  br.close();
} catch(Exception e){
  e.printStackTrace();
}
```

Приклад 15.15. (html, txt)

Класи-мости InputStreamReader і OutputStreamWriter при перетворенні символів також використовують деяке кодування. Її можна задати, передавши в конструктор як аргумент її назву. Якщо воно не відповідатиме ніякій з відомих кодувань, буде кинуто исключениеUnsupportedEncodingException. Ось деякі з коректних значень цього аргументу (чутливого до регістра!) для поширених кодувань: "Cp1251", "UTF-8", "8859 1" і так далі

Клас StreamTokenizer

Екземпляр StreamTokenizer створюється поверх існуючого об’єкту, або InputStream, або Reader. Як ijava.util.StringTokenizer, цей клас дозволяє розбивати дані на лексеми (token), що виділяються з потоку по певних властивостях. Оскільки робота ведеться із словами, конструктор, приймаючий InputStream, оголошений як deprecated (пропонується обертати байтовий потік класом InputStreamReader і викликати другий конструктор). Загальний принцип роботи такий же, як і у StringTokenizer, – задаються параметри розбиття, після чого викликається метод nextToken(), поки не буде досягнутий кінець потоку. Способи завдання розбиття у StreamTokenizer досить різноманітні, але прості, і тому тут не розглядаються.

Робота з файловою системою

Клас File

Якщо класи потоків здійснюють реальний запис і читання даних, то клас File – це допоміжний інструмент, покликаний забезпечити роботу з файлами і каталогами.

Об'єкт класу File є абстрактним представленням файлу і шляху до нього. Він встановлює тільки відповідність з ним, при цьому для створення об'єкту неважливо, чи існує такий файл на диску. Після створення можна виконати перевірку, викликавши метод exists, який повертає значення true, якщо файл існує. Створення або видалення об'єкту класу File жодним чином не відображається на реальних файлах. Для роботи з вмістом файлу можна отримати екземпляри FileI/OStream.

Об'єкт File може указувати на каталог (дізнатися це можна шляхом виклику методу isDirectory). Метод list повертає список імен (масив String) файлів, що містяться в нім (якщо об'єкт File не указує на каталог – буде повернений null).

Наступний приклад демонструє використання об'єктів класу File:

```
import java.io.*;
public class FileDemo {
  public static void findFiles(File file, FileFilter filter
      PrintStream output) throws IOException{
    if (file.isDirectory()) {
      File[] list = file.listFiles();
      for (int i=list.length; --i>=0;) {
        findFiles(list[i], filter, output);
      }
    } else {
      if (filter.accept(file))
        output.println("\t" + file.getCanonicalPath());
    }
  }
  public static void main(String[] args){
    class NameFilter implements FileFilter {
      private String mask;
      NameFilter(String mask){
        this.mask = mask;
      }
      public boolean accept(File file){
        return (file.getName().indexOf(mask)!=-1)?true:false;
      }
    }
    File pathFile = new File(".");
    String filterString = ".java";
    try {
      FileFilter filter = new NameFilter(filterString);
      findFiles(pathFile, filter, System.out);
    } catch(Exception e){
      e.printStackTrace();
```

```
        }
        System.out.println("work finished");
    }
}
```

Приклад 15.16. (html, txt)

При виконанні цієї програми на екран будуть виведені назви (у канонічному вигляді) всіх файлів, з розширенням .java, що містяться в поточному каталозі і всіх його підкаталогах.

Для визначення того, що файл має розширення .java, використовувався інтерфейс FileFilter з реалізацією у вигляді внутрішнього класу NameFilter. Інтерфейс FileFilter визначає тільки один метод accept, що повертає значення, що визначає, чи потрапляє переданий файл в умови фільтрації. Окрім цього інтерфейсу, існує ще один різновид інтерфейсу фільтру – FilenameFilter, де метод accept визначений трохи інакше: він приймає не об'єкт файлу до перевірки, а об'єкт File, вказуючий на каталог, де знаходиться файл для перевірки, і рядок його назви. Для перевірки збігу, з урахуванням регулярних виразів, потрібно відповідним чином реалізувати метод accept. У конкретному приведеному прикладі можна було обійтися і без використання інтерфейсів FileFilter або FilenameFilter. На практиці їх можна використовувати для виклику методів list об'єктів File – в цих випадках будуть повернені файли з урахуванням фільтру.

Також клас File надає можливість отримання деякої інформації про файл.

- Методи canRead і canWrite – повертається boolean значення, чи можна буде додатку проводити читання і зміну вмісту з файлу, відповідно.
- getName – повертає рядок – ім'я файлу (або каталогу).
- getParent, getParentName – повертають каталог, де файл знаходиться у вигляді рядка назви і об'єкту File, відповідно.
- getPath – повертає шлях до файлу (при цьому в рядок перетвориться абстрактний шлях, на який указує об'єкт File).
- isAbsolutely – повертає boolean значення, чи є абсолютним шлях, яким вказаний файл. Визначення, чи є шлях абсолютним, залежить від системи, де запущена Java-машина. Так, для Windows абсолютний шлях починається з вказівки диска, або символом '\'. Для Unix абсолютний шлях починається символом '/' .
- isDirectory, isFile – повертає boolean значення, чи указує об'єкт на каталог або файл, відповідно.
- isHidden – повертає boolean значення, чи указує об'єкт на прихований файл.
- lastModified – дата останньої зміни.
- length – довжина файлу в байтах.

Також можна змінити деякі властивості файлу – методи setReadOnly, setLastModified, призначення яких очевидно з назви. Якщо потрібно створити файл на диску, це дозволяють зробити методи createNewFile, mkDir, mkDirs. Відповідно, createNewFile створює порожній файл (якщо такий ще не існує), mkDir створює каталог, якщо для нього всі батьківські вже існують, а mkDirs створить каталог разом зі всіма необхідними батьківськими.

Файл можна і видалити – для цього призначені методи delete і deleteOnExit. При виклику методу delete файл буде видалений відразу ж, а при виклику deleteOnExit після закінчення роботи Java-машини (тільки при коректному завершенні роботи) відмінити запит вже неможливо.

Таким чином, клас File дає можливість достатньо повного управління файловою системою.

Клас RandomAccessFile

Цей клас реалізує відразу два інтерфейси – DataInput і DataOutput – отже, може проводити запис і читання всіх примітивних типів Java. Ці операції, як випливає з назви, проводяться з файлом. При цьому їх можна проводити по черзі, довільним чином переміщаючись по файлу за допомогою виклику методу seek(long) (перекладає на вказану позицію у файлі). Дізнатися поточне положення покажчика у файлі можна викликом методу getFilePointer.

При створенні об’єкту цього класу конструктору як параметри потрібно передати два параметри: файл і режим роботи. Файл, з яким проводитиметься робота, указується або за допомогою String – назва файлу, або об’єктом File, йому відповідним. Режим роботи (mode) – є рядком або "r"(тільки читання), або "rw"(читання і запис). Спроба відкрити неіснуючий файл тільки на читання приведе до виключення FileNotFoundException. При відкритті на читання і запис він буде негайно створений (або ж буде кинуто виключення FileNotFoundException, якщо це неможливо здійснити).

Після створення об’єкту RandomAccessFile можна скористатися методами інтерфейсів DataInput і DataOutputдля проведення з файлом операцій прочитування і запису. Після закінчення роботи з файлом його слід закрити, викликавши метод close.

Висновок

У даній лекції ви познайомилися з таким важливим поняттям, як потоки дані (stream). Потоки є дуже ефективним способом вирішення завдань, пов’язаних з передачею і отриманням даних, незалежно від особливостей використовуваних пристроїв вводу/виводу. Як ви тепер знаєте, саме в пакеті java.io містяться стандартні класи, вирішальні завдання обміну даними в самих різних форматах.

Були описані базові класи байтових потоків InputStream і OutputStream, а також символьних потоків Readerи Writer. Всі класи потоків явним або неявним чином успадковуються від них. Короткий огляд показав, для чого призначений кожен клас, як з ним працювати, які класи не рекомендовані до використання. Вивчено, як передавати в потоки значення примітивних типів Java. Особлива увага була приділена операціям з об’єктами, для яких існує спеціальний механізм серіалізації.

Нарешті, були описані класи для роботи з файловою системою – File і RandomAccessFile.

ТЕМА. *Мережеві протоколи. Пакет java.net.*

План

1. *Побудова мережевих застосувань.*
2. *Рівні моделі OSI і їх функціональні призначення*
3. *Засоби Java для роботи з мережевими протоколами.*

Рекомендована література: 3, 4, 5.

Теоретичні відомості

Протягом останніх декількох десятиліть розміри і кількість мереж значно виросли. У 80-х роках існувала безліч типів мереж. І практично кожна з них була побудована на своєму типі устаткування і програмного забезпечення, часто не сумісних між собою. Це приводило до значних труднощів при спробі з'єднати декілька мереж (наприклад, різний тип адресації робив ці спроби практично безнадійними).

Ця проблема була розглянута Усесвітньою організацією по стандартизації (International Organization for Standardization, ISO) і було ухвалено рішення розробити модель мережі, яка могла б допомогти розробникам і виробникам мережевого устаткування і програмного забезпечення діяти спільно. В результаті в 1984 р. була створена модель OSI – модель взаємодії відкритих систем (Open Systems Interconnected). Вона складається з семи рівнів, на які розділяється завдання організації мережевої взаємодії. Схемний вони представлені в таблиці 16.1.

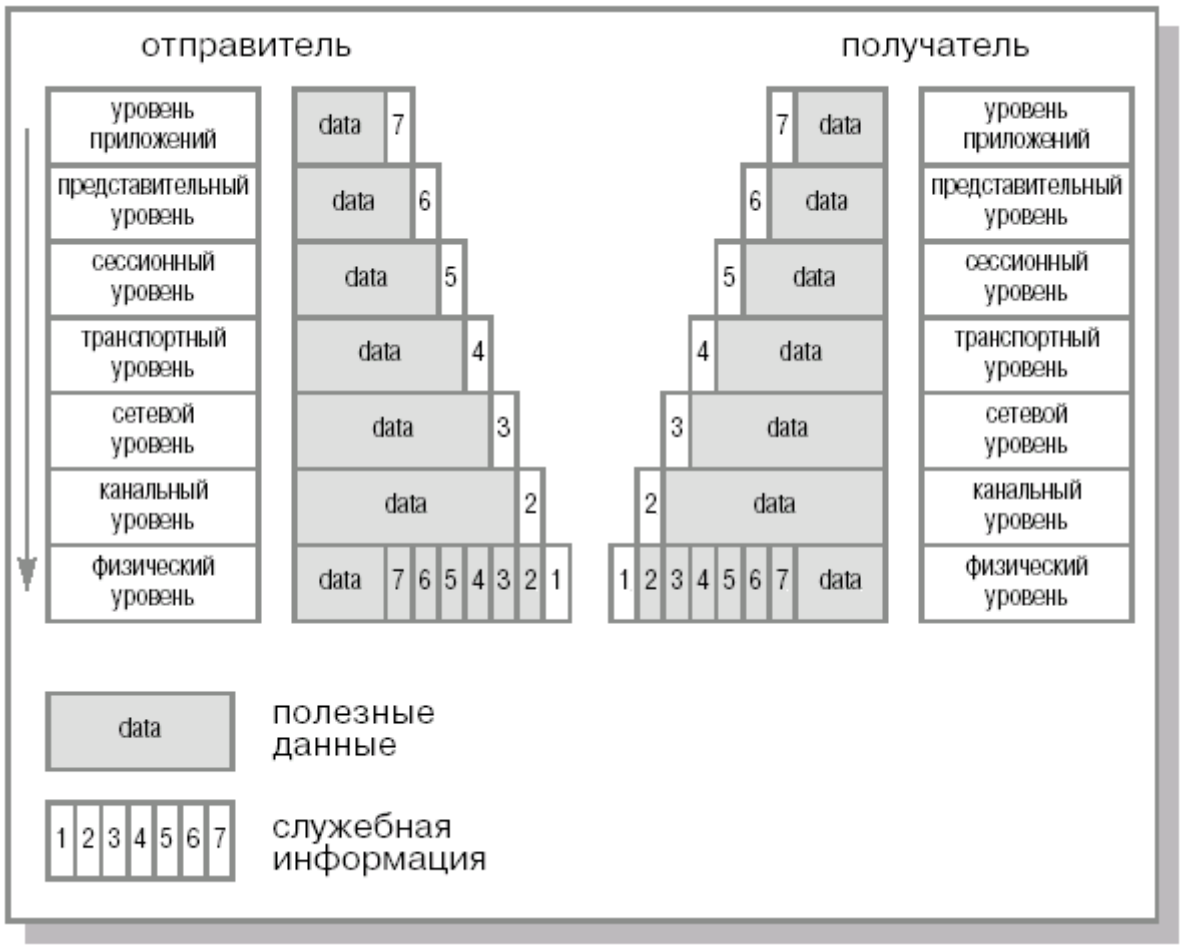
Таблиця 16.1. Рівні моделі OSI.

Номер рівня	Назва рівня	Одиниця інформації
Layer 7	Рівень додатків	Дані (data)
Layer 6	Представницький рівень	Дані (data)
Layer 5	Сесійний рівень	Дані (data)
Layer 4	Транспортний рівень	Сегмент (segment)
Layer 3	Мережевий рівень	Пакет (packet)
Layer 2	Рівень передачі даних	Фрейм (frame)

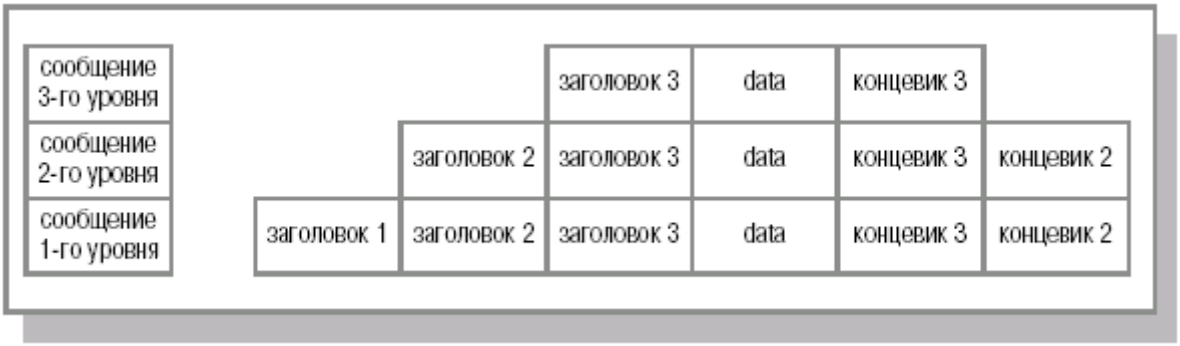
Layer 1	Фізичний рівень	Битий (bit)

Хоча сьогодні існують різноманітні моделі мереж, більшість розробників дотримуються саме цієї загальновизнаної схеми.

Розглянемо процес передачі інформації між двома комп'ютерами. Програмне забезпечення формує повідомлення на рівні 7 (додатків), що складається із заголовка і корисних даних. У заголовку міститься службова інформація, яка необхідна рівню додатків адресата для обробки інформації, що пересилається (наприклад, це може бути інформація про файл, який необхідно передати, або операцію, яку потрібно виконати). Після того, як повідомлення було сформоване, рівень додатків направляє його "вниз" на представницький рівень (layer 6). Отримане повідомлення, що складається із службової інформації рівня 7 і корисних даних, для рівня 6 представляється як одне ціле (хоча рівень 6 може прочитувати службову інформацію рівня 7). Протокол представницького рівня виконує необхідні дії на підставі даних, отриманих із заголовка рівня додатків, і додає заголовок свого рівня, в якому міститься інформація для відповідного (6-го) рівня адресата. Отримане в результаті повідомлення передається далі "вниз" сеансовому рівню, де також додається службова інформація. Доповнене повідомлення передається на наступний транспортний рівень і так далі на кожному подальшому рівні (схемний це представлено на рис.16.1). При цьому службова інформація може додаватися не тільки в початок повідомлення, але і в кінець (наприклад, на 3-му рівні, рис.16.2). У результаті виходить повідомлення, що містить службову інформацію всіх семи рівнів.



Мал. 16.1. Інкапсуляція і декапсуляция пакету.

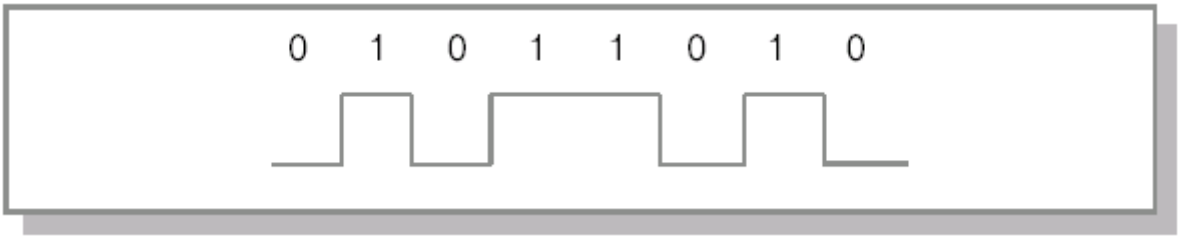


Мал. 16.2. Додавання службовій інформації в початок і кінець пакету.

Процес "обгортання" переданих даних службовою інформацією називається інкапсуляцією (encapsulation).

Далі це повідомлення передається через мережу у вигляді бітів. Битий – це мінімальна порція інформації, яка може набувати значення 0 або 1. Таким чином, все повідомлення кодується у вигляді набору нулів і одиниць, наприклад, 010110101. У простому випадку на фізичному рівні для передачі формується електричний сигнал, що складається з серії електричних імпульсів (0 - немає сигналу, 1 - є сигнал). Саме ця одиниця прийнята для вимірювання швидкості передачі інформації. Сучасні мережі зазвичай надають канали з продуктивністю в десятки і сотні Кбіт/с і Мбіт/с.

Одержувач на фізичному рівні отримує повідомлення у вигляді електричного сигналу (рис.16.3). Далі відбувається процес, зворотний інкапсуляції, - декапсуляція (decapsulation). На кожному рівні відбувається розбір службової інформації. Після декапсуляції повідомлення на першому рівні (прочитування і обробки службової інформації 1-го рівня) це повідомлення, що містить службову інформацію другого рівня і дані у вигляді корисних даних і службової інформації вищестоящих рівнів, передається на наступний рівень. На канальному (2-м) рівні знову відбувається аналіз системної інформації і повідомлення передається на наступний рівень. І так до тих пір, поки повідомлення не дійде до рівня додатків, де у вигляді кінцевих даних передається приймаючому застосуванню.



Мал. 16.3. Представлення даних у вигляді електричного імпульсу.

Як приклад можна привести звернення браузера до web-серверу. Додаток клієнта – браузер – формує запит для отримання web-страницы. Цей запит передається додатком на рівень 7 і далі послідовно на кожен рівень моделі OSI. Досягнувши фізичного рівня, наш

первинний запит "обростає" службовою інформацією кожного рівня. Після цього він передається по фізичній мережі (кабелям) у вигляді електричних імпульсів на сервер. На сервері відбувається розбір відповідної системної інформації кожного рівня, внаслідок чого посланий запит досягає додатку web-сервера. Там він обробляється, після чого клієнтові відправляється відповідь. Процес відправки відповіді аналогічний відправці запиту – за винятком того, що повідомлення посилає сервер, а отримує клієнт.

Оскільки кожен рівень моделі OSI стандартизований, споживачі можуть використовувати спільно устаткування і програмне забезпечення різних виробників. В результаті web-сервер під управлінням операційної системи Sun Solaris може передати HTML-страницу користувачеві MS Windows.

Зрозуміло, сумісність можна забезпечити лише до деякого рівня. Якщо одна машина передає дані у вигляді радіохвиль, а інша у вигляді світлових імпульсів, то їх взаємодія без використання додаткового устаткування неможлива. Тому було введено поняття мережо-незалежних і мережо-залежних рівнів.

Три нижні рівні – фізичний, канальний і мережевий – є мережо-залежними. Наприклад, зміна Ethernet на ATM спричиняє за собою повну зміну протоколу фізичного і канального рівнів.

Три верхні рівні – додатків, представницький і сесійний – орієнтовані на прикладні завдання і практично не залежать від фізичної технології побудови мережі. Так, перехід від Token Ring на Ethernet не вимагає змін в перерахованих рівнях.

Транспортний рівень є проміжним між мережо-залежними і мережо-незалежними рівнями. Він приховує всі деталі функціонування нижніх рівнів від верхніх. Це дозволяє розробникові додатків не замислюватися про технічні засоби реалізації транспортування мережевих повідомлень.

Разом з назвою повідомлення (message) в стандартах ISO для позначення одиниці даних використовують терміннпротокольний блок даних (Protocol Data Unit, PDU). У різних протоколах застосовуються і інші назви, закріплені стандартами, або просто традиційні. Наприклад, в сімействі протоколів TCP/IP протокол TCP розділяє потік даних на сегменти, протокол UDP працює з датаграмами (або дейтаграмами, від datagram), сам протокол IP використовує термін пакети. Часто так само говорять про кадри або фрейми.

Для глибшого розуміння принципів роботи мережі розглянемо кожен рівень окремо.

Physical layer (layer 1)

Як видно із загальної схеми розташування рівнів в моделі OSI, фізичний рівень (Physical layer) найперший. Цей рівень описує середовище передачі даних. Стандартизуються фізичні пристрої, що відповідають за передачу електричних сигналів (роз'єми, кабелі і так далі) і правила формування цих сигналів. Розглянемо по порядку всі складові цього рівня.

Велика частина мереж будується на кабельній структурі (хоча існують мережі, засновані на передачі інформації за допомогою, наприклад, радіохвиль). Зараз існують різні типи кабелів. Найбільш поширені з них:

- телефонний дріт;
- *коаксіальний кабель;*
- *витаючи пара;*
- *оптоволокно.*

Телефонний кабель почав використовуватися для передачі даних з часів появи перших комп'ютерів. Головною перевагою телефонних ліній була наявність вже створеної і розвиненої інфраструктури. З її допомогою можна передавати дані між комп'ютерами, що знаходяться на різних материках, так само легко, як і вести розмову людям, які знаходяться за багато тисяч кілометрів один від одного. На сьогоднішній день використання телефонних ліній також залишається популярним. Користувачі, яких влаштовує невелика швидкість передачі даних, можуть дістати доступ до Internety зі своїх домашніх комп'ютерів. Основними недоліками використання телефонного кабелю є невелика швидкість передачі, оскільки з'єднання відбувається не безпосередньо, а через телефонні станції. При цьому вимога до якості переданого сигналу при передачі даних значно вища, ніж при передачі "голосу". А оскільки більшість аналогових АТС не справляються з цим завданням (рівень "шуму", або перешкод, і якість сигналу залишає бажати кращого), то швидкість передачі даних дуже низька. Хоча при підключенні до сучасних цифрових АТС можна отримати високу і надійну швидкість зв'язку.

Коаксіальний кабель використовувався в мережах ще кілька років тому, але сьогодні це велика рідкість. Такий тип кабелю по будові практично ідентичний звичайному телевізійному коаксіальному кабелю – центральна мідна жила відокремлена шаром ізоляції від переплетення. Деякі відмінності є в електричних характеристиках (у телевізійному кабелі використовується кабель з хвильовим опором 75 Ом, в мережевому – 50 Ом).

Основними недоліками цього кабелю є низька швидкість передачі даних (до 10 Мбіт/с), схильність діям зовнішніх перешкод. Крім того, підключення комп'ютерів в таких мережах відбувається паралельно, а значить, максимальна можлива швидкість пропускання ділиться на всіх користувачів. Але, в порівнянні з телефонним кабелем, коаксиал дозволяє об'єднувати близько розташовані комп'ютери з набагато кращою якістю зв'язку і вищою швидкістю передачі даних.

Вита пара ("twisted pair") – найбільш поширений засіб для передачі даних між комп'ютерами. У даному типі кабелю використовується мідний попарно скручений дріт, що дозволяє зменшити кількість перешкод і наведень, як при передачі сигналу по самому кабелю, так і при дії зовнішніх перешкод.

Існує декілька категорій цього кабелю. Перерахуємо основні з них. Cat 3 – був стандартизований в 1991 р., електричні характеристики дозволяли підтримувати частоти передачі до 16 Мгц, використовувався для передачі даних і голосу. Вища категорія – Cat 5, була спеціально розроблена для підтримки високошвидкісних протоколів. Тому його електричні характеристики лежать в межах до 100Мгц. На такому типі кабелю працюють протоколи передачі даних 10, 100, 1000 Мбіт/с. На сьогоднішній день кабель Cat5 практично витіснив Cat 3. Основна перевага витої пари перед телефонними і коаксіальними кабелями – вища швидкість передачі даних. Також використання Cat 5 в більшості випадків дозволяє, не міняючи кабельну структуру, підвищити продуктивність мережі (переходом від 10 до 100 і від 100 до 1000 Мбіт/с).

Оптоволокно використовується для з'єднання великих сегментів мережі, які розташовуються далеко один від одного, або в мережах, де потрібна велика смуга пропускання, перешкодостійкість. Оптичний кабель складається з центрального провідника світла (серцевини) – скляного волокна, оточеного іншим шаром скла – оболонкою, що володіє меншим показником заломлення, чим серцевина. Розповсюджуючись по серцевині, промені світла не виходять за її межі, відбиваючись від покриваючого шару оболонки. Світловий промінь зазвичай формується напівпровідниковим або діодним лазером. Залежно від розподілу показника заломлення і від величини діаметру сердечника розрізняють:

- одномодовое волокно;
- багатомодове волокно.

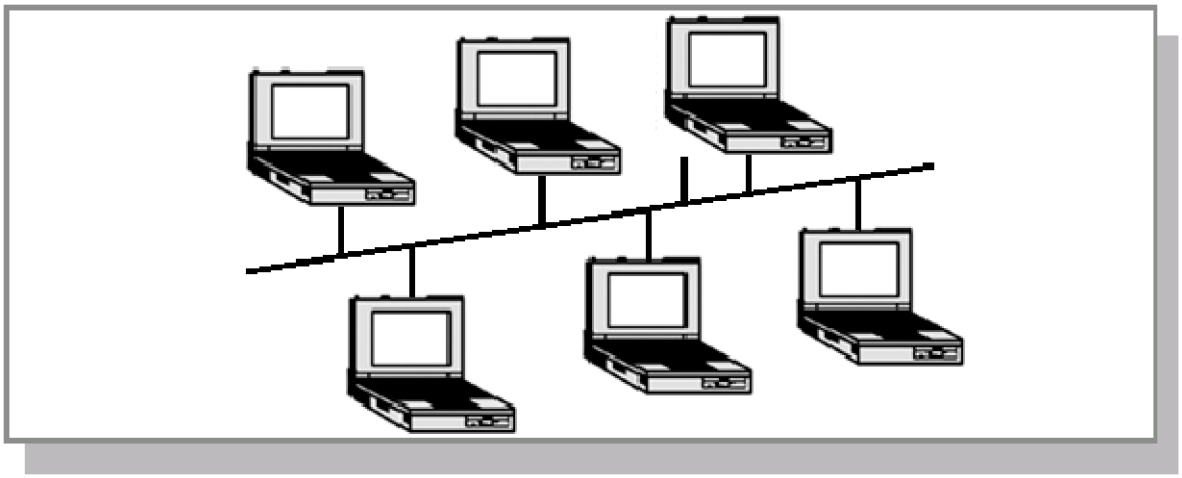
Поняття "мода" описує режим розповсюдження світлових променів в сердечнику кабелю. У одномодовом кабелі використовується провідник дуже малого діаметру, сумірного з довжиною хвилі світла. У багатомодовому кабелі застосовуються ширші сердечники, яких легко виготовити. У цих кабелях в сердечнику одночасно існує декілька світлових променів, що відбиваються від оболонки під різними кутами. Кут віддзеркалення променя називається модою світивши. Оптоволокно володіє наступними перевагами: стійкість до електромагнітних перешкод, високі швидкісні характеристики на великих відстанях. Основним недоліком є як дорожнеча самого кабелю, так і трудомісткість монтажних робіт, оскільки всі роботи виконуються на дорогому високоточному устаткуванні.

Фізичний рівень також відповідає за перетворення сигналів між різними середовищами передачі даних. Наприклад, при необхідності з'єднати сегменти мережі, побудовані на оптоволокні і витій парі, застосовують так звані конвертори (в даному випадку вони перетворять світловий імпульс в електричний).

Для включення комп'ютера в мережу використовується спеціальний пристрій – мережевий адаптер (Network adapter), що дозволяє обмінюватися наборами бітів, представленими електричними сигналами. Мережева карта (так частіше називають мережевий адаптер) зазвичай має шину ISA або PCI для підключення в комп'ютер і відповідний роз'єм для підключення до середовища передачі даних (наприклад, для витої пари, коаксіал і тому подібне).

Тепер, коли ми знаємо, як відбувається з'єднання комп'ютерів в одну мережу, розглянемо варіанти фізичної схеми такої мережі, або, іншими словами, фізичній топології (структури локальної мережі).

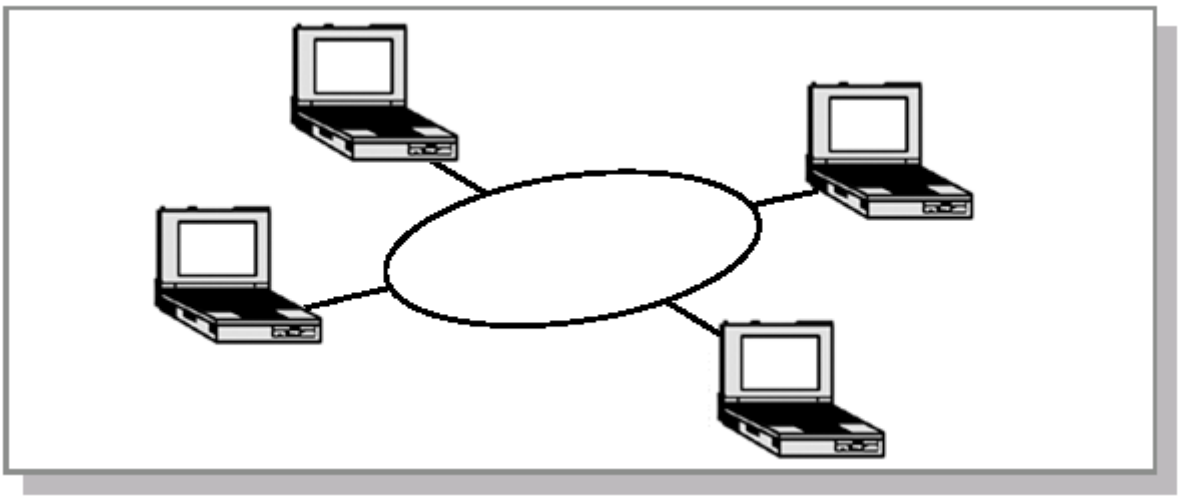
Топологія "шина" (bus) показана на мал. 16.4.



Мал. 16.4. Топологія "шина" (bus).

Всі комп'ютери і мережеві пристрої приєднані до одного дроту і фактично безпосередньо сполучені між собою.

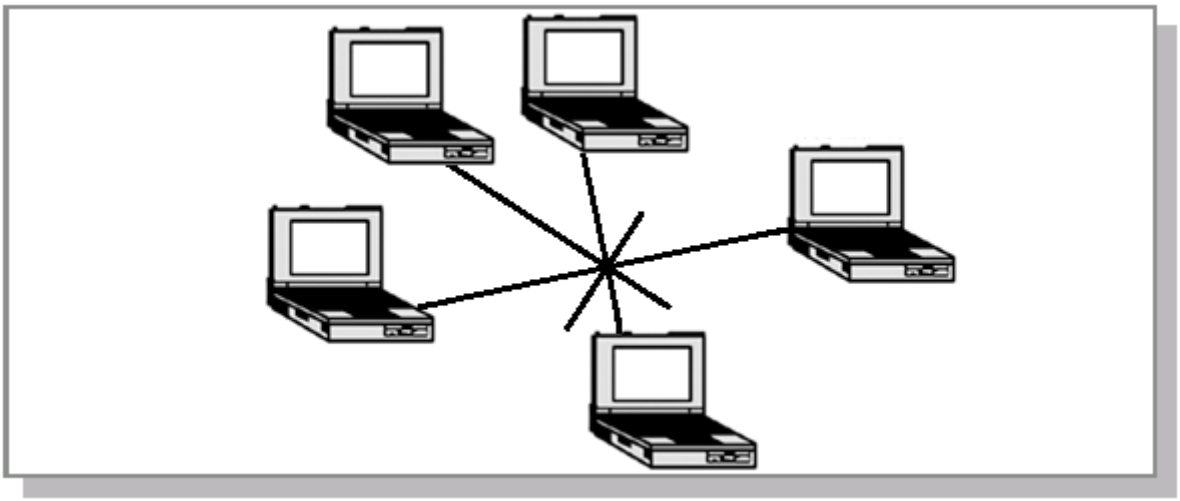
Топологія "кільце" (ring) показана на мал. 16.5.



Мал. 16.5. Топологія "кільце" (ring).

Кільце складається з мережевих пристроїв і кабелів між ними, створюючих одне замкнуте кільце.

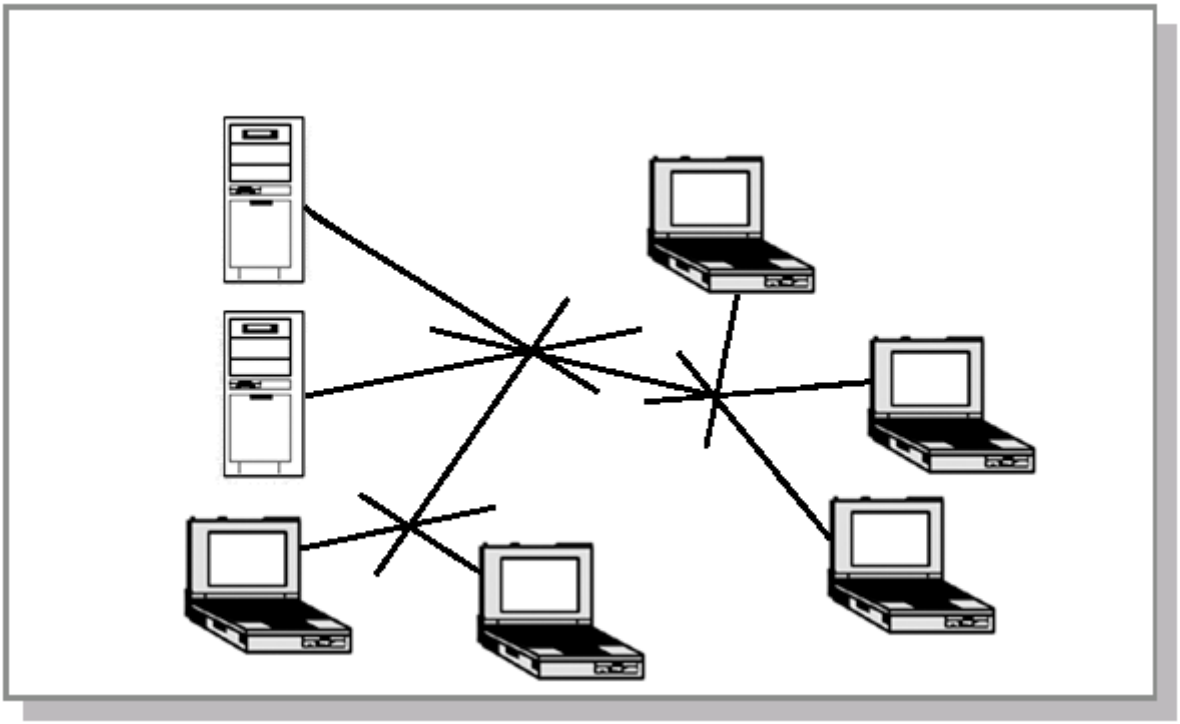
Топологія "зірка" показана на мал. 16.6.



Мал. 16.6. Топологія "зірка" (star).

Всі комп'ютери і мережеві пристрої підключені до одного центрального пристрою.

Топологія "розширена зірка" (extended star) показана на мал. 16.7.



Мал. 16.7. Топологія "розширена зірка"(extended star).

Така схема практично аналогічна топології "зірка", за одним виключенням. Кожен пристрій сполучений з локальним центральним пристроєм, а він, у свою чергу, сполучений з центром іншої "зірки".

Data layer (layer 2)

Фізичний рівень пересилає просто набір сигналів – бітів. При цьому не враховується, що декілька комп'ютерів, підключених до одного середовища передачі даних (наприклад, до одного кабелю), можуть почати одночасно передавати інформацію у вигляді електричних імпульсів, що, очевидно, приведе до змішення сигналів. Тому одному із завдань Data layer(канальний рівень) є перевірка доступності середовища передачі. Також цей рівень відповідає за доставку фреймів між джерелом і адресатом в межах мережі з однією топологією. Для забезпечення такої функціональності Data layer розділяють на два підрівні:

- логічна передача даних (Logical Link Control, LLC);
- управління доступом до середовища (Media Access Control, MAC).

LLC відповідає за перехід з другого рівня на більш вищий – третій мережевий рівень.

MAC відповідає за передачу даних на нижчий рівень – Physical layer.

Розглянемо ці підрівні детальніше.

LLC sublayer

Цей підрівень був створений для забезпечення незалежності від існуючих технологій. Він забезпечує обмін даними з мережевим (третім) рівнем незалежно від фізичного середовища передачі даних.LLC отримує дані з мережевого рівня, додає в них службову інформацію і передає пакет для подальшої інкапсуляції і обробки протоколом рівня MAC. Наприклад, це може бути Ethernet, Token Ring,Frame Relay.

MAC sublayer

Цей підрівень забезпечує доступ до фізичного рівня. Для передачі пакетів по мережі необхідно організувати ідентифікацію комп'ютерів в мережі. Для цього у кожного комп'ютера на канальному рівні визначена унікальна адреса, яка ще іноді називають фізичною адресою, або MAC-адресом.

Він записаний в незалежній пам'яті мережевої карти) і задається виробником. Довжина MAC-адреса 48 битий, або 6 байт (кожен байт складається з 8 битий), які записуються в шестнадцатеричном форматі. Перші 3 байти називаються OUI (Organizational Unique Identifier), організаційний унікальний ідентифікатор. Цей номер видається кожному виробникові мережевого устаткування міжнародною організацією IEEE (Institute of Electrical and Electronic Engineers, Інститут інженерів по електротехніці і радіоелектроніці, джерело багатьох стандартів і специфікацій). Останні 3 байти є ідентифікаційним номером самої мережевої карти. Виробник гарантує, що всі його адаптери мають різні номери. Така система адрес гарантує, що в мережі не буде двох комп'ютерів з однаковими фізичними адресами.

Записуватися фізична адреса може в різних форматах, наприклад: 00:00:B4:90:4C:8C, 00-00-B4-90-4C-8C, 0000.B490.4C8C – різні виробники використовують різні стандарти. Розглянемо, наприклад, адрес0000.1c12.3456. Тут 0000.1c – ідентифікатор виробника, а 12.3456 – ідентифікатор мережевої карти.

Один з найпоширеніших протоколів MAC-рівня – протокол Ethernet. У мережах, побудованих на його основі, застосовується спеціальний метод для організації доступу до середовища передачі даних – CSMA/CD (carrier sense multiple access/collision detect, колективний доступ з пізнанням тієї, що несе і виявленням колізій). Передбачається, що основою мережі є загальна шина (наприклад, коаксіальний кабель), до якої підключені всі комп'ютери. В результаті повідомлення, відправлене однією машиною, доставляється всім підключеним мережевим пристроям. CSMA/CD описує цілий комплекс заходів, необхідних для запобігання і коректної обработкиколлизий (collision), тобто ситуацій, коли декілька комп'ютерів одночасно почали передачу даних. Очевидно, що у такому разі ніхто не зможе отримати коректну інформацію з мережі.

Розглянемо детальніше процес передачі даних на Data layer. Хай один комп'ютер збирається послати дані іншому. Під час процесу інкапсуляції MAC-адрес цієї машини і MAC-адрес одержувача будуть записана в службові поля. Повідомлення, що згенерувало, по правилах протоколу Ethernet відсилається через загальну шину всім машинам, підключеним до цієї ділянки мережі.

Кожен комп'ютер, що отримав повідомлення, перевіряє, кому воно було адресоване. Якщо MAC-адрес, вказана у фреймі,, і MAC-адрес, записана в мережевому адаптері одержувача, збігаються, то пакет приймається і передається на вищестоящий рівень для подальшої обробки. Якщо ж адреса в пакеті не збігається з адресою мережевої карти, то такий пакет відкидається.

Іноді буває необхідно послати повідомлення, яке має бути отримане всіма вузлами локальної мережі. В цьому випадку в пакеті указується MAC-адрес одержувача у вигляді FF-FF-FF-FF-FF-FF. Ця адреса використовується для широковещания (broadcast), яке приймуть всі мережеві пристрої і передадуть на вищестоящий рівень.

Розглянемо пристрої, вживані для побудови мереж в різних топологіях.

Топологія шина ("bus") описує загальне середовище передачі даних, яка вже розглядалася для ілюстрації протоколу Ethernet. Спеціальних пристроїв для побудови такої мережі не використовується (втім, конкретні технології можуть пред'являти специфічні вимоги; наприклад, кінці коаксіального кабелю повинні підключатися до особливого пристрою – терминатору, але це не впливає на структуру мережі).

На топології кільце ("ring") грунтується протокол Token Ring. Фізично мережею є замкнуте кільце, в якому кожен комп'ютер двома відрізками кабелю з'єднується зі своїми сусідами. На відміну від мережі, що працює на основі Ethernet, тут використовується складніша схема. Передача ведеться послідовно по кільцю в одному напрямі. У мережі циркулює кадр спеціального формату – маркер (token). Якщо машина не має даних для передачі, вона при отриманні маркера передає його далі по кільцю. Інакше вона вилучає його із звернення, що дає їй доступ до мережі, і потім відправляє пакет з адресою одержувача, який починає передаватися по кільцю. Коли він доходить до адресата, той робить позначку, що пакет отриманий. Машина-відправник, отримавши підтвердження, відправляє сусідові новий маркер для забезпечення можливості іншим станціям мережі передавати дані. Хоча цей алгоритм складніший, він забезпечує властивості відмовостійкості.

При побудові мережі на основі топології "зірка" потрібно використовувати, окрім мережевих карт в комп'ютері, додаткове мережеве устаткування в центрі, куди підключаються всі "промені зірки". Наприклад, як такий пристрій може застосовуватися концентратор (hub). В цьому випадку кожен комп'ютер підключається до нього за допомогою кабелю "витаючи пара". Алгоритм роботи концентратора дуже простий – отримавши пакет на один зі своїх портів, він пересилає його на всіх останніх. В результаті знову виходить загальна шина, точніше, – логічна загальна шина, оскільки фізична структура мережі зоряно-образна. Технологія Ethernet дозволяє понизити кількість колізій за допомогою CSMA/CD. Недоліком концентратора є те, що користувачі мережі можуть "прослуховувати" чужий трафік (зокрема перехопити пароль, якщо він передається у відкритому вигляді). Загальна максимальна швидкість ділиться між всіма підключеними користувачами. Тобто, якщо швидкість передачі даних складає 10 Мбіт/с, то в середньому на кожного користувача може доводитися всього 2 Мбіт/с.

Дорожчим, але і продуктивнішим рішенням є використання комутатора (switch). Комутатор, на відміну від концентратора, має в пам'яті таблицю, що зіставляє номери його портів і MAC-адреса підключених до нього комп'ютерів. Він аналізує у кожного фрейма, що пересилається, адресу відправника, намагаючись визначити, які машини підключені до кожного з його портів. Таким чином комутатор заповнює свою таблицю. Далі при проходженні чергового фрейма він перевіряє адресу одержувача, і якщо він знає, до якого порту підключена ця машина, він посилає фрейм тільки на один цей порт. Якщо адреса одержувача комутатору невідома, то він відправляє фрейм на всі порти, крім того, з якого цей пакет прийшов. Таким чином, виходить, що якщо два комп'ютери обмінюються даними між собою, то вони не перенавантажують своїми пакетами інші порти і, відповідно, їх пакети практично неможливо перехопити.

Побудовані таким чином мережі можуть охоплювати декілька сотень машин і мати протяжність в декілька кілометрів. Як правило, така мережа охоплює одне або декілька будівель одного підприємства, а тому називається локальною мережею (Local area network, LAN).

Network layer (layer 3)

У попередній лекції ми розглянули другий рівень в моделі OSI. Одним з обмежень цього рівня є використання "плоскої" однорівневой моделі адресації. При спробі побудувати велику мережу, застосовуючи для ідентифікації комп'ютерів MAC-адреса, ми отримаємо величезну кількість broadcast-трафика. Протокол, який підтримується третім рівнем, задіює ієрархічну структуру для унікальної ідентифікації комп'ютерів.

Для прикладу уявимо собі телефонну мережу. Вона також має ієрархічну адресацію. Наприклад, в номере+7-095-101-12-34 перша цифра позначає код країни, далі йде код области/города(095), а потім указується сам телефон (101-12-34). Останній номер також є складеним. 101 – це код станції, куди підключений телефон, а 12-34 визначає місцеположення телефону. Завдяки такій ієрархічній структурі ми можемо визначити розташування необхідного абонента з найменшими витратами. Ієрархічна адресація для комп'ютерної мережі також повинна дозволяти встановлювати зв'язок між розрізненими і видаленими мережами.

На мережевому рівні (Network layer) існує декілька протоколів, які дозволяють передавати дані між мережами. Найбільш поширеним з них на сьогоднішній день є IP.

Його попередник, протокол IPX, зараз вже практично не використовується в публічних мережах, але його можна знайти в приватних, закритих мережах.

Основний пристрій, вживаний на 3-му рівні, називається роутером (router), або маршрутизатором. Він сполучає видалені локальні мережі (LAN), утворюючи глобальну мережу (Wide area network, WAN). Роутер має два або більш мережевих інтерфейсу і таким чином підключений відразу до декількох локальних мереж. Отримавши пакет з локального пристрою або комп'ютера, що належить до однієї з LAN, роутер проглядає заголовок третього рівня. На підставі отриманої інформації роутер ухвалює рішення, що робити з пакетом. Якщо одержувач пакету знаходиться в тій же локальній мережі, що і відправник, роутер ігнорує його, оскільки повідомлення, як вже розглядалося, доставляється засобами більш низькорівневих протоколів (наприклад, Ethernet).

Інакше пакет потрібно передати в одну з інших LAN, до яких підключений роутер. Основне завдання цього пристрою – вибір шляху, по якому пересилатиметься повідомлення. Оскільки може існувати безліч зв'язків між деякими двома мережами відправника і одержувача, роутер повинен вибрати найбільш оптимальний шлях. Пересилка пакету від одного вузла мережі до наступного називається hop (дослівно – стрибок, стрибок). Вибір чергового вузла, якому роутер перешле повідомлення, може залежати від багатьох чинників – завантаження мережі, найменший шлях до одержувача, вартість трафіку по різних маршрутах і так далі

Нова система адресації, що вводиться на мережевому рівні, повинна полегшувати роутеру визначення шляху для доставки пакету через глобальні мережі. Розглянемо реалізацію найбільш популярного на сьогоднішній день протоколу IP детальніше.

При проходженні даних з верхніх рівнів на нижние на мережевому рівні до пакету додається службовий заголовок цього рівня. У заголовку IP-пакета міститься необхідна для подальшої передачі інформація, така як адреси відправника і одержувача. Поняття IP-адреса дуже важливе для розуміння роботи глобальних мереж, тому зупинимося на нім детальніше.

IP-адрес

IP-адрес представляється 32-бітовим бінарним числом, яке часто записують у вигляді 4 десяткових чисел, від 0 до 255 кожне. Наприклад: 60.13.54.11, 130.154.201.1, 194.11.3.200. Логічно він складається з двох частин – адреси машини (host) і адреси мережі (network). Мережева частина IP-адреса показує, до якої мережі належить адресат, а хост-частина (host) ідентифікує мережевий пристрій в цій мережі. Комп'ютери з однаковою мережевою частиною знаходяться в одній локальній мережі, а тому можуть легко обмінюватися даними. Якщо ж у них різні network-ID, то, навіть знаходячись в одному фізичному сегменті, вони зазвичай не можуть "побачити" один одного.

Оскільки IP-адрес складається з 4-х октетів (так називають ці числа, оскільки 256=28), один, два або три перші октети можуть використовуватися для визначення мережевої адреси, останні задають host-части. Для зручності виділення адрес користувачам (адже, як правило, організації вимагається їх відразу декілька), було введено 5 класів адрес. Їх позначають латинськими буквами від A до E. У відкритих мережах використовуються перші три з них.

У таблиці 16.2 дано зразкове розбиття IP-адресов на мережеву (N) і машинну (H) частини залежно від класу мережі.

Таблиця 16.2. Зразкове розбиття IP-адресов.

	1 октет	2 октет	3 октет	4 октет
Клас A	N	H	H	H
Клас B	N	N	H	H
Клас C	N	N	N	H

Клас A

У класі A для ідентифікації мережі, до якої належить адреса, використовується перший октет, причому, перший біт завжди дорівнює 0. Останні октети задають адресу хоста. Таким чином, адреса мережі класу A може бути в діапазоні 0-126. 127-а адреса зарезервована для спеціального використання – всі адреси, со127, що починаються, вважаються за локальні для мережевого адаптера, тобто завжди відправник сам є і одержувачем. Останні вільні три октети застосовуються для завдання адреси хоста в даній мережі. Це означає, що в одній мережі може бути використане до 224 адрес (з них два крайніх, тобто 0 і 224-1, зарезервовані, вони розглядаються нижчим). Отже, в кожній з 127 мереж класу A можна адресувати 16,777,214 машин.

Діапазон адрес 10.0.0.0-10.255.255.255 в публічних мережах не використовується. Ці адреси спеціально зарезервовані для застосування в локальних мережах і глобальними маршрутизаторами не обробляються.

Клас B

У мережі класу B перші два октети (причому, перший біт завжди дорівнює 1, другий – 0) використовуються для визначення мережі, останні два октети – для визначення адреси хоста. Діапазон адрес мережі класу B лежить в межах от128.0.x.x до 191.255.x.x, що дає 16,384 таких мереж. У кожній з них може бути не більш 65,534=216-2адресов (дві крайні адреси виключаються).

У цій підмережі зарезервованими для локального використання є наступні адреси: 172.16.0.0-172.31.0.0.

Клас C

Діапазон мережі класу C визначається першими трьома октетами (перші біти завжди 110). І в десятковому вигляді ця мережа може починатися з 192 по 223. Для визначення адреси хоста використовується останній октет. Таким чином, в кожній з 2,097,152 мереж класу C може бути задіяно 28 (без двох крайніх) або 254 адреси.

Зарезервованими для локального використання є наступні адреси: 192.168.0.0-192.168.255.255.

Class D

Цей клас використовується для особливих завдань (multicast-группы). Діапазон адрес – 224.0.0.0-239.255.255.255.

Class E

Цей клас адрес зарезервований для застосування в майбутньому. Діапазон адрес – 240.0.0.0-247.255.255.255.

Дві адреси в кожній підмережі є зарезервованими. IP-адрес, в якій вся хост-часть складається з бінарних нулів, використовується для позначення адреси самої мережі. Наприклад, мережа класу A може мати адрес112.0.0.0, а комп'ютер, підключений до неї, – адреса 112.2.3.4. Адреса мережі використовується роутерами для завдання маршруту.

Друга зарезервована адреса – броадкаст-адрес (broadcast). Ця адреса застосовується, коли джерело хоче послати дані всім пристроям в локальній мережі. Для цього хост-часть заповнюється бінарними одиницями. Наприклад, для розглянутої мережі 112.0.0.0 це буде адреса 112.255.255.255, а для мережі класу B 171.10.0.0броадкаст-адрес виглядатиме як 171.10.255.255. Дані, послані за адресою 171.10.255.255, будуть отримані всіма пристроями в мережі 171.10.0.0.

Підмережі. Маска підмережі

Введення класів мереж багато в чому спростило завдання розподілу адрес по організаціях. Але не завжди має сенс використовувати, наприклад, цілу мережа класу C, якщо в ній реально буде розміщено лише 10 комп'ютерів. Для раціональнішого використання мереж організовують підмережі.

Адреса підмережі включає мережеву частину від мережі класу A, B або C і так зване поле підмережі (subnet field). Для цього значення виділяють додаткові біти, належні хост-части (тобто для адреси підмережі може бути використано до 3-х октетів з мережі класу A, до 2-х з мережі класу B, і 1 для C, відповідно). Таких бітів може бути мінімальне один (таким чином одна мережа розділяється на дві підмережі), а максимально стільки, щоб для хост-части залишалося ще два бита (інакше підмережа складатиметься лише з двох службових адрес - адреси підмережі і броадкаст-адреса). Для мереж класу A це дає від 1 до 22 бітів, для B – від 1 до 14 бітів, для C – від 1 до 6.

Розбиття на підмережі зменшує також розміри броадкаст-доменов, що необхідне, інакше для мережі класу A броадкаст-запрос може розсилатися на 16 мільйонів комп'ютерів. І якщо кожен з них пошле хоч би поодинці такому запиту, навантаження на мережу буде надмірно великим. Якщо ж комп'ютер знаходиться у виділеній підмережі, то в сусідні мережі і підмережі роутер пересилати броадкаст-запрос не буде, унаслідок чого економиться смуга пропускання фізичних каналів зв'язку.

Для визначення довжини адреси підмережі використовується спеціальне понятитие – маска підмережі. Це число визначає, яка частина IP-адреса застосовується для завдання

мережевій і подсетевой частині. Маску підмережі можна визначити наступному образом. Запишемо IP-адрес в бінарному вигляді. Всі розряди, що відносяться до network- і subnet-части, замінимо на 1, всі значення, що відносяться до host-части, - на 0. В результаті отримаємо маску підмережі.

Наприклад, маска підмережі для цілої мережі класу A виглядатиме як 255.0.0.0, для мережі класу B:255.255.0.0, для мережі класу C – 255.255.255.0. Для розділення на підмережі, як було сказано вище, потрібно деякі біти хост-частини виділити для поля підмережі. Наприклад, маска 255.255.255.192 визначає підмережу класу C, для якої кількість хостов дорівнюватиме 62.

Протоколи ARP, RARP

Коли формується пакет для відправлення, на мережевому рівні закладається IP-адрес одержувача. Проте для передачі на нижчестоячий канальний рівень також потрібно знати MAC-адрес. Для визначення відповідності IP-адресу MAC-адреса існує ARP-протокол (Address Resolution Protocol, протокол визначення адрес). Він працює таким чином.

Формується спеціальний широкомовний (broadcast) запит. Він розглядався вищим, його особливість в тому, що його отримують всі пристрої, підключені до цієї локальної мережі. У такому запиті MAC-адрес одержувача складається з одних бінарних одиниць, а в полі IP-адреса записується саме та адреса, для якої потрібно визначити MAC-адрес. Коли якийсь комп'ютер отримує такий запит, він порівнює вказану IP-адрес зі своїм. Якщо вони розрізняються, повідомлення ігнорується. Якщо вони рівні, то формується відповідь, в якій по всіх правилах вказані IP- і MAC-адреса відправника, тобто шуканої машини.

Для того, щоб не навантажувати широкомовними запитами мережа, ARP-протокол підтримує спеціальну ARP-таблицю, яка знаходиться в оперативній пам'яті і зберігає відповідність між IP- і MAC-адресами. Після вдалого визначення MAC-адреса якого-небудь вузла мережі робиться відповідний запис в таблицю, щоб при наступному відсиланні пакету не довелося знову розсилати broadcast-запити. Через деякий час запис віддаляється. Це дозволяє автоматично підстроюватися під зміни в мережі, адже у якогось вузла могли змінити MAC- або IP-адрес. Якщо відправник не знаходить IP-адрес одержувача в ARP-таблице, то знову формується і відправляється ARP-запит.

Протокол RARP (Reverse ARP – зворотний ARP) діє навпаки – він відомій MAC-адресу зіставляє IP-адрес. Це необхідно, наприклад, для роботи таких протоколів, як BOOTP (Bootstrap Protocol, протокол автоматичного налаштування) і DHCP (Dynamic Host Configuration Protocol, протокол динамічної конфігурації хостів). Їх призначення – полегшити завдання системному адміністраторові. Вони дозволяють не вводити IP-адрес в кожен комп'ютер локальної мережі, а призначають їх самих в автоматичному режимі. При завантаженні чергової машини посилається broadcast-запит – протилежний ARP-запиту. Якщо в ARP-запросе йде опит "IP одержувача відомий, MAC одержувача – ???", то в RARP-запиті "MAC одержувача відомий, IP - ???". Якщо в мережі є DHCP-сервер, він відповідає на RARP-запит, указуючи IP-адрес для цього комп'ютера (особливо це ефективно при великій кількості комп'ютерів).

Обидва ці протоколу працюють в рамках лише локальній мережі, оскільки всі пакети, що направляються в інші мережі, обробляються і маршрутизируются роутером, тому знати MAC-адрес не потрібно (відправник указує MAC-адрес самого роутера).

Transport layer (layer 4)

Розглянемо протокол 4-го транспортного рівня моделі OSI. Сімейство TCP/IP включає два таких протоколу – TCP і UDP. TCP (Transmission Control Protocol, протокол управління передачею) забезпечує віртуальні з'єднання між призначеними для користувача застосуваннями і гарантує точну доставку даних. UDP (User Datagram Protocol, протокол передачі датаграм користувача) служить для швидкого обміну спеціальними повідомленням (датаграмами) без гарантії доставки.

Основні характеристики TCP і UDP показані в таблиці. 16.3.

Таблиця 16.3. Основні характеристики TCP і UDP.

TCP	**UDP**
Для роботи встановлює з'єднання	Працює без з'єднань
Гарантована доставка даних	Гарантій доставки немає
Розбиває початкове повідомлення на сегменти	Передає повідомлення цілком у вигляді датаграм
На стороні одержувача повідомлення наново збирається з сегментів	Повідомлення, що приймаються, не об'єднуються
Пересилає наново втрачені сегменти	Підтверджень про доставку немає
Контролює потік сегментів	Ніякого контролю потоку датаграм не немає

TCP

TCP/IP є комбінацією двох рівнів, TCP і IP. IP – протокол третього рівня – забезпечує якнайкращу, але не гарантовану доставку даних через мережу. TCP – протокол четвертого рівня – дозволяє цю гарантію забезпечити. Тому спільно вони можуть надати більшу кількість сервісів.

Робота по TCP-протоколу починається зі встановлення з'єднання. Два комп'ютери (один з них ініціатор з'єднання, другий, – що приймає) обмінюються спеціальними пакетами в три етапи. Умовно їх можна назвати "запит", "підтвердження" і "підтвердження на підтвердження". Така процедура необхідна, щоб при отриманні якого-небудь старого пакету (наприклад, робиться друга спроба встановити з'єднання) не виникало ніяких неоднозначностей.

Після успішного встановлення з'єднання учасники можуть почати обмінюватися даними. Розглянемо приклад HTTP-сервера, який відправляє HTML-страницу клієнтові. Текст може бути дуже довгим, щоб уміщатися в один пакет, тому перше завдання рівня

TCP – розбити повідомлення на декілька пакетів, а на стороні відправника – зібрати їх знову в єдине ціле. Оскільки черговість пакетів поза сумнівом важлива, кожен отримує порядковий номер.

Наступне завдання протоколу – забезпечити гарантовану доставку. Робиться це за допомогою наступної процедури. Відправник посилає пакет з номером n і починає чекати. Одержувач у разі успішного приходу пакету n, відправляє підтвердження про отримання ("квитанцію"), в якому також указує номер n. Якщо відправник протягом определеннного часу (тайм-ауту) не отримує підтвердження, він рахує пакет nпотерянным і посилає його ще раз.

Зрозуміло, відправникові неефективно просто чекати, поки одержувач отримає і обробить кожен пакет поодинці. Тому процедура ускладнюється, вводиться спеціальне поняття – "вікно" (window). Вікно має деякий розмір, припустимо, 10. Це означає, що передача починається з відсилання 10 перших пакетів. Одержувач може прийняти їх не в тому порядку, в якому вони були відіслані. Проте, на кожен успішно отриманий пакет відсилається підтвердження з вказівкою номера такого пакету. Якщо відправник відіслав вже все 10 пакетів, але квитанція про отримання пакету 1 так і не прийшла, то передача припиняється, а після тайм-ауту перший пакет вважається за втрачений і пересилається ще раз. Якщо ж підтвердження приходять регулярно, то відправляються нові пакети, але не більше 10 одноразово.

Цей алгоритм носить назва "Ковзаючого вікна". Якщо уявити, що всі пакети збудовані в ряд, то вікно "ковзає" по ньому, визначаючи, які пакети готові для відсилання. Такий підхід забезпечує гарантовану доставку при максимально можливій швидкості передачі даних. Зрозуміло, протокол TCP працює не так швидко, адже частина пропускної спроможності мережі витрачається на пересилку квитанцій і повторів втрачених пакетів. Проте, велику кількість інформації потрібно доставляти саме таким чином. Зрозуміло, що частини, наприклад, тексту повинні складатися в строгому порядку і без пропусків. Були розроблені спеціальні механізми, автоматично регулюючі величини таких характеристик, як тайм-аут і розмір вікна, для досягнення оптимальної продуктивності.

UDP

На відміну від TCP, UDP не гарантує доставку даних. UDP не встановлює віртуального з’єднання, джерело просто шле спеціальні повідомлення (у UDP вони називаються датаграмами) одержувачеві. Якщо дані були доставлені некоректно, або взагалі частина пакетів загубилася, UDP не дозволяє їх відновити. Запит на отримання даних повинен буде виконуватися наново.

Здавалося б, недоліків у такого протоколу досить багато, що ставить під сумнів його ефективність. Але є сервіси, де UDP незамінний. Наприклад, при передачі потокового аудио-видео якби ми використовували TCP, то при втраті одного пакету у нас була б припинена трансляція для його повторної передачі. При використанні UDP один втрачений пакет – всього лише незначне (напевно, взагалі непомітне користувачеві) погіршення изображения/звука, при цьому передача даних не уривається. Також при використанні UDP не обов’язково встановлювати віртуальне з’єднання, не потрібно посилати квитанції – все це прискорює роботу протоколу.

Порти

Як було розглянуто, для протоколу IP достатньо знати IP-адрес, щоб обробити повідомлення. Обидва протоколи транспортного рівня, TCP і UDP, додатково використовують порти (port) для взаємодії з вищестоящими рівнями. Порт описується числом від 0 до 65535 і дозволяє операційній системі розподіляти пакети, що приходять на транспортний рівень, між різними прикладними програмами. Припустимо, користувач одночасно викачує файл з FTP-сервера і працює з видаленим сервером бази даних. От обидва цих серверів призначена для користувача машина отримуватиме по мережі пакети і необхідно правильно передавати їх відповідним застосуванням (FTP-клиенту і БД-клієнту).

Частина портів зарезервована під стандартні застосування. Наприклад, для FTP зарезервований порт 21, для telnet – 23, для HTTP – 80. Далі приведений список розподілу портів:

- порти менше 255 використовуються для публічних сервісів;
- порти з діапазону 255-1023 призначаються компаниями-разрабочиками для додатків;
- номери понад 1023 – не регульовані.

Таким чином, кажучи про встановлене TCP-соединении, мають зважаючи на 4 числа: IP-адрес і порт однієї сторони і ті ж параметри другої сторони. Наприклад, якщо користувач зі своєї машини 194.11.22.33 звернувся через браузер до web-серверу 213.180.194.129, то це означає, що встановлене з'єднання 194.11.22.33:10123-213.180.194.129:80 (номер 10123 вибраний довільно – використовується будь-який незайнятий порт).

Використовується також термін "сокет" (socket), під яким мається на увазі пара "IP-адрес:порт", – адресна "крапка" для мережевих звернень.

Session layer (layer 5)

Після транспортного рівня пакет поступає на рівень сесій. Коли додатки, запущені на різних машинах, починають взаємодіяти через мережу, то між ними відбувається безліч мини-"переговоров", обмінів, діалогів, з яких і складається мережева сесія.

Session layer координує встановлення і завершення з'єднань і сесій між додатками.

Presentation layer (layer 6)

Цей рівень відповідає за представлення даних, що пересилаються по мережі. Він забезпечує наступну функціональність: data formatting (presentation, тобто перетворення даних в зрозумілий одержувачеві формат), data encryption (шифрування), data compression (стискування даних). Presentation layer виконує одну або всі ці функції під час передачі повідомлень між 7-м і 5-м рівнями. Приведемо приклад використання рівня уявлень.

Припустимо, хост-получатель використовує EBCDIC (кодування, вживане на крупних IBM-серверах для передачі символів у вигляді чисел), а хост- відправник – ASCII

(традиційне кодування для персональних комп'ютерів). Presentation layer забезпечуватиме перетворення даних, що пересилаються, між цих машинами.

Для забезпечення безпеки при передачі приватній інформації через публічні мережі необхідне шифрування даних. Один з поширених протоколів, використовуваних для цієї мети, – SSL (Secured Sockets Layer) – може бути віднесений до рівня уявлень.

Якщо канал зв'язку володіє низькою пропускною спроможністю, доцільно застосовувати компресію даних. Представницький рівень, використовуючи математичні алгоритми, дозволяє зменшити об'єм передаваних даних. Що стосується високошвидкісних каналів, то для них використання компресії може зажадати значних обчислювальних потужностей при великих об'ємах трафіку.

Application layer (layer 7)

Останній рівень – рівень додатків, на якому визначаються взаємодіючі сторони, враховується авторизація користувача, визначається якість обслуговування (quality of service) і, власне, забезпечується виконання прикладних завдань, таких, як обмін файлами, електронними листами і так далі Рівень додатку – це не само застосування, хоча часто програми виконують деякі функції Application layer.

Вже згадувалися багато протоколів цього рівня: FTP, HTTP, telnet. Цей список легко продовжити, наприклад, протоколи POP3 і SMTP для отримання і відправки електронних листів, або протоколи DNS (Domain Name System, служба імен доменів), що забезпечують перетворення числових адрес IP- в текстові доменні імена і назад. Хоча Internet з технічної точки зору побудований на основі IP-адресації, текстові імена зрозуміліше і легше запам'ятовуються, а тому набагато поширеніші серед звичайних користувачів.

Розглянемо принцип роботи DNS детальніше. Всі звикли звертатися до, наприклад, web-серверам по доменному імені. З іншого боку для встановлення з'єднання потрібна IP-адрес. Так, при зверненні до сервера www.ru встановлюється TCP-з'єднання з хостом 194.87.0.50.

Оскільки в мережі величезна кількість серверів, DNS-імена є ієрархічними, інакше з ними було б дуже скрутно працювати. Ієрархічні частини імені записуються через крапку. Перший рівень указується останнім. Спочатку існувало 7 трьохбуквених доменів першого рівня:

- com – commercial (комерційні організації);
- org – non-profit (некомерційні організації);
- net – network service (організація роботи мережі);
- edu – educational (освіта, часто – американські університети);
- int – international (міжнародні організації);
- gov – government (уряд, організації американського уряду);
- mil – military (військові, американські військові організації).

Крім того, для кожної країни був заведений двохбуквений домен, наприклад, ru - Росія, su – СРСР, us – США, fr– Франція і так далі Останнім часом вводяться нові доменні імена верхнього рівня, такі, як biz і info.

У кожному домені першого рівня може бути безліч доменів другого рівня. Так, існує безліч сайтів в домені ru, або com. У домена другого рівня може бути безліч доменів третього рівня і так далі

Як же визначити, якій IP-адресу відповідає доменне ім'я сервера, до якого звертається користувач? Для цього існує аналогічна ієрархічна система DNS-серверів, кожен з яких відповідає за свій домен. У мережевих налаштуваннях комп'ютера указується адреса локального DNS-сервера. При запиті до нього сервер спочатку перевіряє список імен, за які відповідає він сам, і кеш. Якщо шукане ім'я йому невідоме, він робить запит вищестоящому DNS-серверу. Наприклад, при зверненні до intuit.ru буде зроблений запит до DNS-серверу, що відповідає за домен ru.

У свою чергу, сервер intuit.ru знає про всі імена в своїй зоні intuit.ru, або, у разі звернення до домена наступного рівня (наприклад, node1.host1.intuit.ru), знає адреса іншого сервера (host1.intuit.ru), який за нього відповідає, і на цей сервер перенаправляє запит.

Таким чином можна встановити IP-адрес для будь-якого зареєстрованого доменного імені.

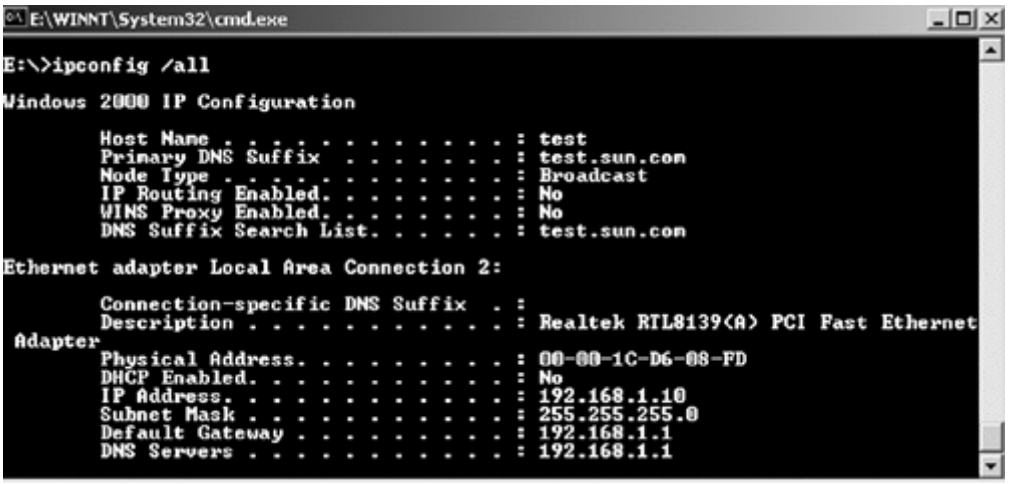
Утиліти для роботи з мережею

Розглянемо основні програми, що дозволяють читати і змінювати мережеві параметри, діагностувати і виявляти помилки при роботі мережі.

У різних ОС існують свої набори утиліт. Порівняємо їх для двох систем, наприклад, Microsoft Windows NT і Sun Solaris. Якими б різними не були ці ОС, в кожній з них реалізована модель OSI. Природно, програмна і апаратна реалізація стека цієї моделі у них розрізняється, але взаємодія всіх рівнів здійснюється за встановленим стандартом.

IPCONFIG (IFCONFIG)

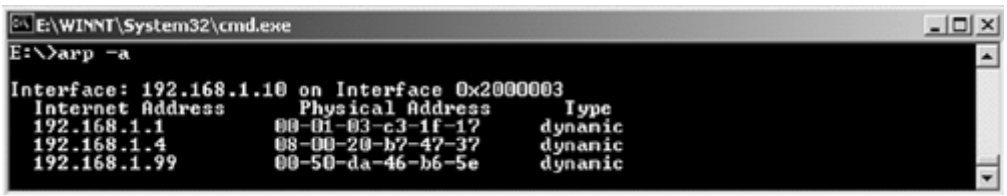
Почнемо з утиліти, яка дозволяє переглядати, перевіряти і змінювати мережеві налаштування. Зазвичай ці налаштування включають інформацію 3-го (мережевого) рівня – IP-адрес, маску підмережі і так далі Для роботи з ними в ОС Windows можна використовувати команду ipconfig. Вона видає інформацію про IP- адресі, масці підмережі (netmask), роутере за умовчанням (default gateway). Задавши додатковий параметр -all, можна отримати докладнішу інформацію – ім'я комп'ютера, ім'я домена, тип мережевої карти, MAC-адрес і так далі



У ОС Solaris для отримання IP-адреса і інших мережевих налаштувань використовується команда ifconfig. Вона також показує назву інтерфейсу, IP-адреса, маску підмережі, MAC-адрес.

ARP

Як вже було сказано раніше, в оперативній пам'яті комп'ютера знаходиться ARP-таблица. У ній містяться MAC-адрес видаленої машини і відповідна йому IP-адрес. Для проглядання цієї таблиці використовується команда arp. Наприклад, arp –a виводить всі відомі адреси MAC-.



Існує два типи записів в ARP-таблиці – статичний і динамічний. Статичний запис вноситься уручну і існує до тих пір, поки уручну ж не буде видалена, або комп'ютер (маршрутизатор) не буде перезавантажений.

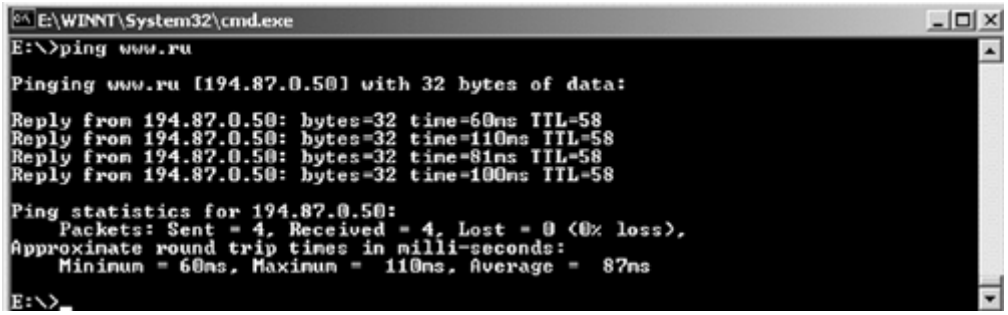
Динамічний запис з'являється при спробі відправити повідомлення на IP- адреса, для якої невідома MAC-адрес. В цьому випадку формується ARP-запрос, який дозволяє цю адресу визначити, після чого відповідний динамічний запис додається в ARP-таблицю. Зберігатися там вона буде не постійна. Після певного часу вона буде автоматично видалена, якщо до даної IP-адресу не було звернень. Затримка на отримання MAC-адреса

складає близько декількох мілісекунд, так що для користувача це буде практично непомітно, зате з'являється можливість відстежити зміни в конфігурації мережі (у відповідності IP- і MAC-адресів).

Ping

Для виявлення різних неполадок в мережі існує декілька утиліт, які дозволяють визначити, на якому рівні моделі OSI відбувся збій, або вказані невірні налаштування мережевих протоколів. Одна з таких утиліт – ping.

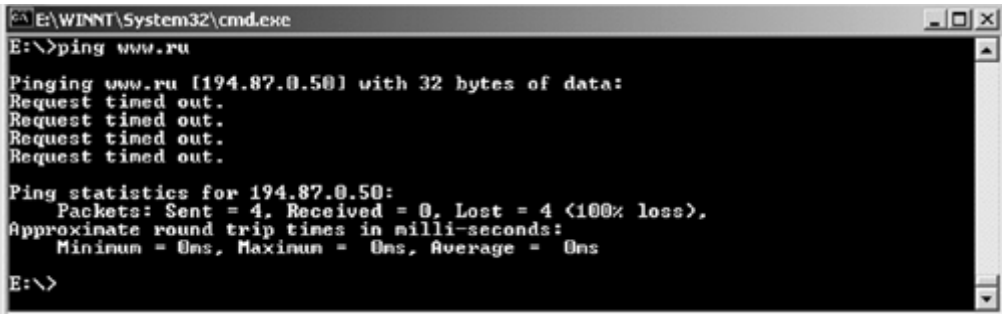
Ця утиліта дозволяє визначити помилки на мережевому рівні (layer 3), використовуючи протокол ICMP (Internet Control Message Protocol) – протокол міжмережевих повідомлень, що управляють. Формат використання цієї утиліти досить простий: ping 194.87.0.50 (де 194.87.0.50 – IP-адрес видаленого комп'ютера). Якщо мережа працює коректно, в результаті виводиться час очікування приходу відповіді від видаленого комп'ютера і час життя пакету (TTL, time to live, кількість "хопів", після якого пакет був би відкинутий; цей параметр показує, скільки залишалося допустимих переходів у пакета-відповіді).



Протокол ICMP знаходиться на стику двох рівнів – мережевого і транспортного. Основний принцип дії цього протоколу – формування ICMP эхо-запроса (echo-request) і эхо-ответа (echo-reply). Запит луни і відповідь на нього може використовуватися для перевірки досяжності хоста- одержувача і його здатності відповідати на запити. Також проходження ехо-запита і ехо- відповіді перевіряє працездатність основної частини транспортної системи, маршрутизацію на машині джерела, працездатність і коректну маршрутизацію на роутерах між джерелом і одержувачем, а також працездатність і правильність маршрутизації одержувача.

Таким чином, якщо на посланий echo-request повертається коректний echo-reply від машини, якою був посланий запит, можна сказати, що транспортна система працює коректно. І якщо браузер не може відобразити web-сторінку, то проблема, видно, не по-перше трьох рівнях моделі OSI.

З прикладу видно, що за умовчанням розмір посиланого пакету - 32 байти, далі виводиться час затримки відповіді і TTL. В даному прикладі показано успішне виконання команди ping. У випадках, коли запити echo request посилаються, але echo reply не повертаються, виводиться повідомлення про закінчення часу очікування відповіді.



Traceroute

Утиліта traceroute також використовує протокол ICMP для визначення маршруту проходження пакету. При відсиланні traceroute встановлює значення TTL послідовно від 1 до 30. Кожен маршрутизатор, через який проходить пакет на шляху до призначеного хосту, зменшує значення TTL на одиницю. За допомогою TTL відбувається запобігання зацикленню пакету в "петлях" маршрутизації, інакше пакети, що "заблукали", остаточно перенавантажували б мережу. Проте, при виході маршрутизатора або лінії зв'язку з ладу потрібно декілька додаткових переходів для розуміння, що даний маршрут втрачений і його необхідно обійти. Щоб запобігти втраті датаграми, поле TTL встановлюється на максимальну величину.

Коли маршрутизатор отримує IP-датаграмму з TTL, рівним 0 або 1, він знищує її і посилає хосту, який її відправив, ICMP-повідомлення "час закінчився" (time exceeded). Принцип роботи traceroute полягає в тому, що IP-датаграма, що містить це ICMP-повідомлення, має як адресу джерела IP-адрес маршрутизатора.

Тепер легко зрозуміти, як працює traceroute. На хост призначення відправляється IP-датаграма з TTL, рівним одиниці. Перший маршрутизатор, який повинен обробити датаграму, знищує її (оскільки TTL дорівнює 1) і відправляє ICMP-повідомлення про закінчення часу (time exceeded). Таким чином визначається перший маршрутизатор в маршруті. Потім traceroute відправляє датаграму з TTL, рівним 2, що дозволяє отримати IP-адрес другого маршрутизатора. Так продовжується до тих пір, поки датаграма не досягне хоста призначення. Утиліта traceroute може посилати як таку датаграму UDP-сообщение з номером порту, який свідомо не буде оброблений додатком (порт вище 30000), тому хост призначення відповість "порт недоступний" (port unreachable). При отриманні такої відповіді робиться вивід, що видалений хост працює коректно. Інакше максимального значення TTL (за умовчанням 30) не вистачило для того, щоб його досягти.

Розглянемо приклад виконання утиліти traceroute.

traceroute to netserv1.chg.ru (193.233.46.3), 30 hops max, 38 byte packets
1 n3-core.mipt.ru (194.85.80.1) 1.508 ms 0.617 ms 0.798 ms
2 mipt-gw-eth0.mipt.ru (193.125.142.177) 2.362 ms 2.666 ms 1.449 ms
3 msu-mipt-atm0.mipt.ru (212.16.1.1) 5.536 ms 5.993 ms 10.431 ms
4 M9-LYNX.ATM6-0.11.M9-R2.msu.net (193.232.127.229) 12.994 ms 7.830 ms 6.816 ms
5 Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37) 12.228 ms 7.041 ms 8.731 ms

6 ChgNet-gw.free.net (147.45.20.222) 77.103 ms 75.234 ms 92.334 ms
7 netserv1.chg.ru (193.233.46.3) 96.627 ms 94.714 ms 134.676 ms

Приклад 16.1. (html, txt)

Перший рядок містить ім'я і IP-адрес хоста призначення, максимальне значення TTL і розмір посиланого пакету (38 байт). Подальші рядки починаються з TTL, після чого слідує ім'я хоста, або маршрутизатора і його IP-адрес. Для кожного значення TTL відправляються три датаграми. Для кожної поверненої датаграми визначається і виводиться час повернення. Якщо протягом 3-х секунд на кожну з 3-х датаграм не була отримана відповідь, то посилається наступна датаграма, а замість значення часу виводиться зірочка. Час повернення – це час проходження датаграми від джерела (хоста, що виконує програму traceroute) до маршрутизатора. Якщо нас цікавить час, витрачений на пересилку між, наприклад, 5 і 6 вузлом, необхідно відняти із значення часу TTL 6 час TTL 5.

У кожній з операційних систем мережева частина утиліти реалізована практично однаково, але реалізація на рівні додатків розрізняється.

У ОС Solaris використовується утиліта traceroute. Як параметр задається IP-адрес, або доменне ім'я видаленого хоста, зв'язок до якого потрібно перевірити. У прикладі, приведеному вище, видно успішне виконання traceroute і коректну роботу сете- залежних рівнів (фізичний, канальний, мережевий).

У ОС Windows утиліта називається tracert. Використовується вона так само, як і в ОС Solaris (tracert netserv1.chg.ru). Принципової відмінності між утилітами tracert і traceroute немає. Особливістю traceroute є наявність більшої кількості функцій (наприклад, можна вказати, починаючи з якого TTL виводити інформацію).

У разі якої-небудь неполадки виводиться відповідне повідомлення. Наприклад, при недоступності мережі на маршрутизаторі видається повідомлення !N (net unreachable):

Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37)
947.327 ms !N 996.548 ms !N 995.257 ms

Це означає, що 147.45.20.37 – маршрутизатор, починаючи з якого, подальший маршрут недоступний. Якщо недоступний сам хост, то повідомлення виглядатиме так:

msu-mipt-atm0.mipt.ru (212.16.1.1)
5.536 ms !H 5.993 ms !H 10.431 ms !H.

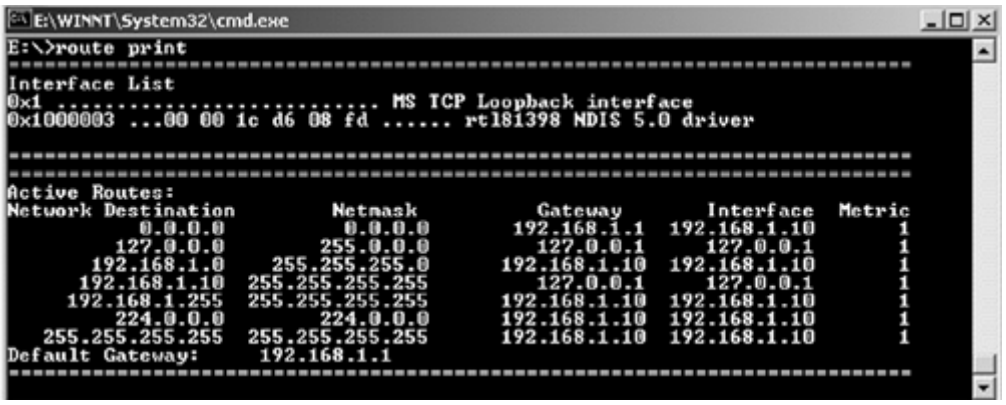
Помилка !P означає недоступність протоколу (protocol unreachable).

Route

Для перегляду і редагування таблиці маршрутів використовується утиліта route. Типовий приклад таблиці маршрутизації на персональному комп'ютері:

Для ОС Windows:

route print



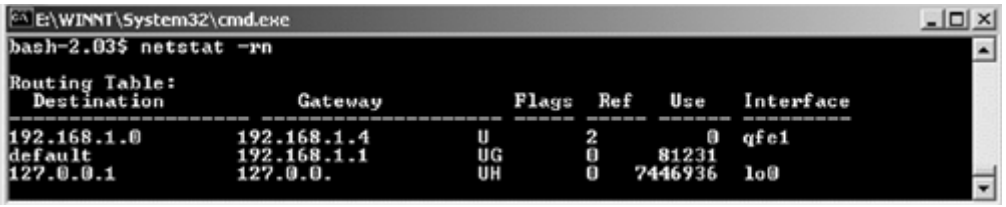
У таблиці маршрутизації указується мережа, маска мережі, маршрутизатор, через який доступна ця мережа, інтерфейс і метрика маршруту. З приведеної таблиці видно, що маршрут за умовчанням доступний через маршрутизатор 192.168.1.1. Мережа 192.168.1.0 з маскою 255.255.255.0 є локальною мережею.

При додаванні маршруту можна використовувати наступну команду.

route ADD 157.0.0.0 MASK 255.0.0.0 157.55.80.1

157.0.0.0 – видалена мережа, 255.0.0.0 – маска видаленої мережі, 157.55.80.1 – маршрутизатор, через який доступна ця мережа. Приблизно такий же синтаксис використовується при видаленні маршруту: route DELETE 157.0.0.0.

У ОС Solaris для проглядання таблиці маршрутизації використовується небагато інша команда – netstat –r.



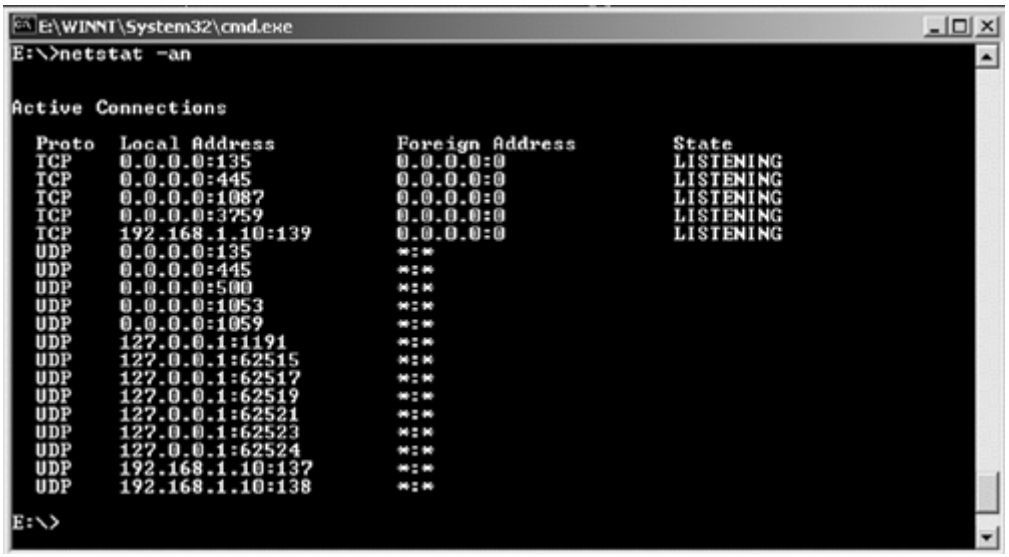
Додавання і видалення маршрутів виконується командою route: route add –net 157.6 157.6.1.20, де 157.6 – скорочена адреса підмережі, а 157.6.1.20 – маршрут, по якому ця мережа доступна. Також видалення маршрутів в таблиці маршрутизації: route del –net 157.6.

Netstat

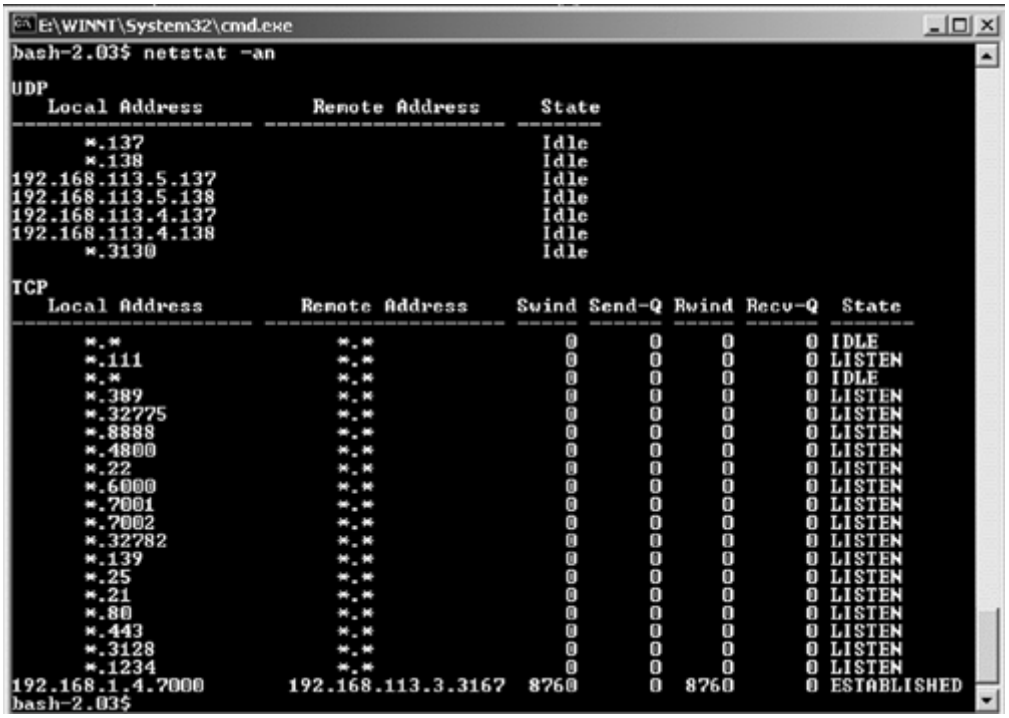
Утиліта netstat дозволяє визначити, які порти відкриті і по яких портах відбувається передача даних між вузлами мережі. Наприклад, якщо запустити web-браузер і відкрити для перегляду web-страницу, то, запустивши netstat, можна побачити наступний рядок:

TCP mycomp:3687 www.ru:http ESTABLISHED

У проведеному прикладі перше значення – TCP – тип протоколу (може бути TCP або UDP), далі слідує ім’я локальної машини і локальний порт, www.ru:http – ім’я видаленого хоста і порту, до якого проводиться звернення (оскільки використовувався порт за умовчанням для протоколу HTTP, то відображається не його числове значення 80, а ім’я протоколу), ESTABLISHED – показує, що TCP-соединение встановлене.



У ОС Windows за допомогою команди netstat –an можна отримати список всіх відкритих портів (параметр –n не визначає DNS-имя, а виводить тільки IP-адрес). З прикладу вище видно, що встановлених з’єднань немає, а всі відкриті порти знаходяться в стані "прослуховування", тобто до цього порту можна звернутися для установки з’єднання. TCP-порт 139 відповідає за установку Netbios-сесій (наприклад, для передачі даних через "мережеве оточення").



У ОС Solaris для отримання інформації про використовувані порти також застосовується утиліта netstat. Формат виводу практично однаковий.

Пакет java.net

Перейдемо до розгляду засобів Java для роботи з мережею.

Класи, що працюють з мережевими протоколами, розташовуються в пакетеjava.net, і простим з них є клас URL. З його допомогою можна сконструювати uniform resource locator (URL), який має наступний формат:

protocol://host:port/resource

Тут protocol – назва протоколу, використовуваного для зв'язку; host – IP-адрес, або DNS-имя сервера, до якого проводиться звернення; port – номер порту сервера (якщо порт не вказаний, то використовується значення за умовчанням для вказаного протоколу); resource – ім'я запрошуваного ресурсу, причому, воно може бути складеним, наприклад:

ftp://myserver.ru/pub/docs/Java/JavaCourse.txt

Потім можна скористатися методом openStream(), який повертає InputStream, що дозволяє рахувати вміст ресурсу. Наприклад, наступна програма за допомогою LineNumberReader прочитує першу сторінку сайту http://www.ru і виводить її на консоль.

```
import java.io.*;
import java.net.*;

public class Net {
  public static void main(String args[]){
    try {
      URL url = new URL("http://www.ru");
      LineNumberReader r =
        new LineNumberReader(new
        InputStreamReader(url.openStream()));
      String s = r.readLine();
      while (s!=null) {
        System.out.println(s);
        s = r.readLine();
      }
      System.out.println(r.getLineNumber());
      r.close();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

З прикладу ми бачимо, що робота з мережею, як і робота з потоками, вимагає додаткової роботи з винятковими ситуаціями. Помилка MalformedURLException з'являється у випадку, якщо рядок з URL містить помилки.

Більш функціональним класом є URLConnection, який можна отримати за допомогою методу класу URL.openConnection(). У цього класу є два методи – getInputStream() (саме з його допомогою працює URL.openStream()) і getOutputStream(), який можна використовувати для передачі даних на сервер, якщо він підтримує таку операцію (багато публічних web-серверы закрито для таких дій).

Клас URLConnection є абстрактним. Віртуальна машина надає реалізації цього класу для кожного протоколу, наприклад, в тому ж пакеті java.net визначений клас HttpURLConnection. Зрозуміло, що классыURL і URLConnection надають можливість роботи через мережу на прикладному рівні за допомогою високорівневих протоколів.

Пакет java.net також надає доступ до протоколів нижчого рівня – TCP і UDP. Для цього спочатку треба ознайомитися з класом InetAddress, який є Internet-адресом, або IP. Екземпляри цього класу створюються не за допомогою конструкторів, а за допомогою статичних методів:

```
InetAddress getLocalHost()
InetAddress getByName(String name)
InetAddress[] getAllByName(String name)
```

Перший метод повертає IP-адрес машини, на якій виконується Java- програма. Другий метод повертає адреса сервера, чиє ім'я передається як параметр. Це може бути як DNS-имя, так і числовий IP, записаний у вигляді тексту, наприклад, "67.11.12.101". Нарешті, третій метод визначає всі IP-адреса вказаного сервера.

Для роботи з TCP-протоколом використовуються класи Socket і ServerSocket. Першим створюється ServerSocket – сокет на стороні сервера. Його простий конструктор має тільки один параметр – номер порту, на якому прийматимуться вхідні запити. Після створення викликається метод accept(), який припиняє виконання програми і чекає, поки який-небудь клієнт не ініціює з'єднання. В цьому випадку робота сервера поновлюється, а метод повертає екземпляр класу Socket для взаємодії з клієнтом:

```
try {
  ServerSocket ss = new ServerSocket(3456);
  Socket client=ss.accept();
  // Метод не повертає
  // управління, поки не підключиться клієнт
} catch (IOException e) {
  e.printStackTrace();
}
```

Клієнт для підключення до сервера також використовує клас Socket. Його простий конструктор приймає два параметри - адреса сервера (у вигляді рядка, або екземпляра InetAddress) і номер порту. Якщо сервер прийняв запит, то сокет конструюється успішно і далі можна скористатися методами getInputStream() илиgetOutputStream().

```
try {
  Socket s = new Socket("localhost", 3456);
  InputStream is = s.getInputStream();
  is.read();
} catch (UnknownHostException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Звернете увагу на обробку виняткової ситуації UnknownHostException, яка генеруватиметься, якщо віртуальна машина за допомогою операційної системи не зможе розпізнати вказану адресу сервера у випадку, якщо він заданий рядком. Якщо ж він заданий екземпляром InetAddress, то цю помилку треба обробляти при виклику статичних методів даного класу.

На стороні сервера клас Socket використовується таким самим чином – через методи getInputStream() игetOutputStream(). Приведемо повніший приклад:

```
import java.io.*;
import java.net.*;
public class Server {
  public static void main(String args[]){
    try {
      ServerSocket ss = new ServerSocket(3456);
      System.out.println("Waiting...");
      Socket client=ss.accept();
      System.out.println("Connected");
      client.getOutputStream().write(10);
      client.close();
      ss.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Сервер по запиту клієнта відправляє число 10 і завершує роботу. Звернете увагу, що при завершенні викликаються методи close() для відкритих сокетів.

Клас клієнта:

```
import java.io.*;
import java.net.*;
public class Client {
  public static void main(String args[]){
    try {
      Socket s = new Socket("localhost", 3456);
      InputStream is = s.getInputStream();
      System.out.println("Read: "+is.read());
      s.close();
    } catch (UnknownHostException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Після запуску сервера, а потім клієнта, можна побачити результат – отримане число, 10, після чого обидві програми закриються.

Розглянемо ці класи детальніше. По-перше, клас ServerSocket має конструктор, в який передається, окрім номера порту, ще і адреса машини. Це може показатися дивним, адже сервер відкривається на тій же машині, де працює програма, навіщо спеціально

указувати її адресу? Проте, якщо комп'ютер має декілька мережевих інтерфейсів (мережевих карток), то він має і декілька мережевих адрес. За допомогою такого деталізованого конструктора можна вказати, по якому саме адресі чекати підключення. Це має бути саме локальна адреса машини, інакше виникне помилка.

Аналогічно, клас Socket має розширений конструктор для вказівки як локальної адреси, з якої встановлюватиметься з'єднання, так і локального порту (інакше операційна система виділяє довільний вільний порт).

По-друге, можна скористатися методом setSoTimeout(int timeout) класу ServerSocket, щоб вказати час в мілісекундах, впродовж якого потрібно чекати підключення клієнта. Це дозволяє серверу не "зависати", якщо ніхто не намагається почати з ним працювати. Тайм-аут задається в мілісекундах, нульове значення означає нескінченний час очікування.

Важливо підкреслити, що після встановлення з'єднання з клієнтом сервер виходить з методу accept(), тобто перестає бути готовим приймати нові запити. Проте, як правило, бажано, щоб сервер міг працювати з декількома клієнтами одночасно. Для цього необхідно при підключенні чергового користувача створювати новий потік виконання, який обслуговуватиме його, а основний потік знову увійде до методу accept(). Приведемо приклад такого рішення:

```
import java.io.*;
import java.net.*;

public class NetServer {
  public static final int PORT = 2500;
  private static final int TIME_SEND_SLEEP = 100;
  private static final int COUNT_TO_SEND = 10;
  private ServerSocket servSocket;

  public static void main(String[] args){
    NetServer server = new NetServer();
    server.go();
  }

  public NetServer() {
    try{
      servSocket = new ServerSocket(PORT);
    } catch(IOException e){
      System.err.println("Unable to open Server Socket : " + e.toString());
    }
  }

  public void go() {

    // Класс-поток для роботи з
    //клієнтом, що підключився
    class Listener implements Runnable {
      Socket socket;
      public Listener(Socket aSocket){
```

```
            socket = aSocket;
          }
          public void run() {
            try {
              System.out.println("Listener started");
              int count = 0;
              OutputStream out = socket.getOutputStream();
              OutputStreamWriter writer = new
                OutputStreamWriter(out);
              PrintWriter pWriter = new PrintWriter(writer);
              while (count<COUNT_TO_SEND) {
                count++;
                pWriter.print(((count>1)?",":"")+ "Say" + count);
                sleeps(TIME_SEND_SLEEP);
              }
              pWriter.close();
            } catch(IOException e){
              System.err.println("Exception : " + e.toString());
            }
          }
        }

        // Основний потік, що циклічно виконує метод accept()
        System.out.println("Server started");
        while (true) {
          try {
            Socket socket = servSocket.accept();
            Listener listener = new Listener(socket);
            Thread thread = new Thread(listener);
            thread.start();
          } catch(IOException e){
            System.err.println("IOException : " + e.toString());
          }
        }
      }

      public void sleeps(long time){
        try {
          Thread.sleep(time);
        } catch(InterruptedException e){
        }
      }
    }
```

Приклад 16.2. (html, txt)

Тепер оголосимо клієнта. Ця програма запускатиме декілька потоків, кожен з яких незалежно підключається до сервера, прочитує його відповідь і виводить на консоль.

```
import java.io.*;
```

```
import java.net.*;

public class NetClient implements Runnable {
  public static final int PORT = 2500;
  public static final String HOST = "localhost";
  public static final int CLIENTS_COUNT = 5;
  public static final int READ_BUFFER_SIZE = 10;

  private String name = null;

  public static void main(String[] args){
    String name = "name";
    for (int i=1; i<=CLIENTS_COUNT; i++) {
      NetClient client = new NetClient(name+i);
      Thread thread = new Thread(client);
      thread.start();
    }
  }

  public NetClient(String name){
    this.name = name;
  }

  public void run() {
    char[] readed = new char[READ_BUFFER_SIZE];
    StringBuffer strBuff = new StringBuffer();
    try {
      Socket socket = new Socket(HOST, PORT);
      InputStream in = socket.getInputStream();
      InputStreamReader reader = new InputStreamReader(in);
      while (true) {
        int count = reader.read(readed, 0
          READ_BUFFER_SIZE);
        if (count==-1) break;
        strBuff.append(readed, 0, count);
        Thread.yield();
      }
    } catch (UnknownHostException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    System.out.println("client " + name + " read : " + strBuff.toString());
  }
}
```

Приклад 16.3. (html, txt)

Тепер розглянемо UDP. Для роботи з цим протоколом і на стороні клієнта, і на стороні сервера використовується клас DatagramSocket. У нього є наступні конструктори:

```
DatagramSocket()
DatagramSocket(int port)
DatagramSocket(int port, InetAddress laddr)
```

При виклику першого конструктора сокет відкривається на довільному доступному порту, що доречно для клієнта. Конструктор з одним параметром, задаючим порт, як правило, застосовується на серверах, щоб клієнти знали, на якому порту їм потрібно намагатися встановлювати з'єднання. Нарешті, останній конструктор необхідний для машин, у яких присутньо декілька мережевих інтерфейсів.

Після відкриття сокетів починається обмін датаграмами. Вони представляються екземплярами классаDatagramPacket. При відсиланні повідомлення застосовується наступний конструктор:

```
DatagramPacket(byte[] buf, int length
        InetAddress address, int port)
```

Масив містить дані для відправки (створений пакет матиме довжину, рівну length), а адреса і порт указують одержувача пакету. Після цього викликається метод send() класу DatagramSocket.

```
try {
  DatagramSocket s = new DatagramSocket();
  byte data[]={1, 2, 3};
  InetAddress addr =
    InetAddress.getByName("localhost");
  DatagramPacket p =
    new DatagramPacket(data, 3, addr, 3456);
  s.send(p);
  System.out.println("Datagram sent");
  s.close();
} catch (SocketException e) {
  e.printStackTrace();
} catch (UnknownHostException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Для отримання датаграми також створюється екземпляр класу DatagramPacket, але в конструктор передається лише масив, в який будуть записані отримані дані (також указується очікувана довжина пакету). Сокет необхідно створити з вказівкою порту, інакше, швидше за все, повідомлення просто не дійде до адресата. Використовується метод receive() класу DatagramSocket (аналогічно методу ServerSocket.accept(), цей метод також перериває виконання потоку, поки не прийде запит від клієнта). Приклад реалізації одержувача:

```
try {
  DatagramSocket s =
    new DatagramSocket(3456);
  byte data[]=new byte[3];
  DatagramPacket p =
    new DatagramPacket(data, 3);
  System.out.println("Waiting...");
  s.receive(p);
  System.out.println("Datagram received: "+
    data[0]+", "+data[1]+", "+data[2]);
  s.close();
} catch (SocketException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Якщо запустити спочатку одержувача, а потім відправника, то можна побачити, що перший надрукує вміст отриманої датаграми, а потім програми завершать свою роботу.

На закінчення приведемо приклад сервера, який отримує датаграми і відправляє їх назад, дописавши до них слово received.

```
import java.io.*;
import java.net.*;

public class DatagramDemoServer {
  public static final int PORT = 2000;
  private static final int LENGTH_RECEIVE = 1;
  private static final byte[] answer = ("received").getBytes();
  private DatagramSocket servSocket = null;
  private boolean keepRunning = true;
  public static void main(String[] args){
    DatagramDemoServer server = new DatagramDemoServer();
    server.service();
  }

  public DatagramDemoServer() {
    try {
      servSocket = new DatagramSocket(PORT);
    } catch(SocketException e){
      System.err.println("Unable to open socket : " + e.toString());
    }
  }
  protected void service() {
    DatagramPacket datagram;
    InetAddress clientAddr;
    int clientPort;
    byte[] data;
```

```
      while (keepRunning) {
        try {
          data = new byte[LENGTH_RECEIVE];
          datagram = new DatagramPacket(data, data.length);
          servSocket.receive(datagram);
          clientAddr = datagram.getAddress();
          clientPort = datagram.getPort();
          data = getSendData(datagram.getData());
          datagram = new DatagramPacket(data, data.length,
            clientAddr, clientPort);
          servSocket.send(datagram);
        } catch(IOException e){
          System.err.println("I/O Exception : " + e.toString());
        }
      }
    }
    protected byte[] getSendData(byte b[]){
      byte[] result = new byte[b.length+answer.length];
      System.arraycopy(b, 0, result, 0, b.length);
      System.arraycopy(answer, 0, result, b.length, answer.length);
      return result;
    }
}
```

Приклад 16.4. (html, txt)

Висновок

У даному розділі були розглянуті теоретичні основи мережі як однієї великої взаємодіючої системи. Були описані всі рівні моделі OSI і їх функціональні призначення. Також були представлені основні утиліти, використовувані для налаштування і виявлення несправностей в мережі. Потім були розглянуті засоби Java для роботи з найбільш поширеними мережевими протоколами. Приведений докладний приклад і для складнішого випадку – сервер, обслуговуючий декілька клієнтів одночасно.