ЛАБОРАТОРНА РОБОТА

Тема: Багатопотокове програмування

В Java передбачена вбудована підтримка багатопотокового програмування (multithreaded programming). Багатопотокова програма містить декілька частин, які виконуються одночасно і називаються потоками. Потоки працюють незалежно один від одного. Застосування потоків дозволяє писати програми, які максимально використовують ресурси процесора. Це особливо важливо для інтерактивних і мережних середовищ, в яких використовується Java.

Підтримка багатопотокового програмування забезпечується класами Thread, ThreadGroup та інтерфейсом Runnable із пакету java.lang.

У табл.1 наведено методи класу Thread та їх опис.

Оголошення	**Опис**
Конструктори	
Thread()	Створює новий потік
Thread(Runnable target)	Створює новий потік, який використовує метод run() вказаного адресата target
Thread(ThreadGroup group, Runnable target)	Створює новий потік у групі потоків group, який використовує метод run()адресата target
Thread(String s)	Створює новий потік з іменем s
Thread(Runnable target, String s)	Створює новий потік з іменем s , який використовує метод run() вказаного адресата target
Thread(ThreadGroup group, String name)	Створює новий потік з іменем s в групі потоків group
Thread(ThreadGroup group, Runnable target, String name)	Створює потік з іменем s в групі потоків group, який використовує метод run()адресата target
Константи	
static final int MAX_PRIORITY=10	Найвищий пріоритет
static final int MIN_PRIORITY=1	Найнижчий пріоритет
static final int NORM_PRIORITY=5	Середній пріоритет
Методи класу	
int activeCount()	Повертає поточну кількість потоків у групі
Thread currentThread()	Повертає поточний об’єкт Thread
boolean interrupted()	Повертає true, якщо потік може бути перерваний
void sleep(long millis) throws InterruptedException	Переводить потік в стан очікування на millis мілісекунд
void yield()	Примушує потік поступитися процесором для іншого потоку
Методи екземпляру	
void checkAccess() throws SecurityException	Визначає, чи виконуваний потік має право модифікувати об’єкт Thread
final String getName()	Повертає ім’я потоку
final int getPriority()	Повертає пріоритет потоку
final ThreadGroup getThreadGroup()	Повертає групу потоків, до якої належить потік

void interrupt()	Перериває виконання потоку
final boolean isAlive()	Повертає true, якщо потік діючий
final boolean isDaemon()	Повертає true, якщо потік – демон
boolean isInterrupted()	Повертає true, якщо потік був перерваний
final synchronized void join(long millis) throws InterruptedException	Примушує потік, який його викликав, чекати завершення пов’язаного з цим методом об’єкта Thread протягом заданих millis мілісекунд
final void join() throws InterruptedException	Примушує потік, який його викликав, чекати завершення пов’язаного з цим методом об’єкта Thread
final void resume()	Відновлює роботу призупиненого потоку
void run()	Метод інтерфейсу Runnable. Запускається на виконання у процесі виклику методу start() потоку
final void setDaemon(boolean on)	Встановлює атрибути для потоку демону
final void setName(String name)	Встановлює ім’я потоку
final void setPriority (int newPriority)	Встановлює пріоритет потоку
synchronized native void start()	Запускає потік на виконання
final void stop()	Зупиняє виконання потоку
final void suspend()	Призупиняє виконання потоку
String toString()	Повертає рядкове подання об’єкта Thread

Під час запуску Java-програми створюється головний потік (main thread). Він створюється автоматично, але його роботою можна керувати через об’єкт Thread. Для цього необхідно отримати посилання на цей об’єкт за допомогою методу currentThread(), наприклад:

```
Thread t=Thread.currentThread();
t.setName(“MyThread”);
System.println(“Current thread is: ” + t);
```

Група потоків – структура даних, яка контролює стан усіх потоків.

Головний потік відіграє важливу роль, оскільки породжує дочірні потоки і повинен завершуватися останнім. У випадку зупинки головного потоку програма завершує свою роботу.

Є два способи створення потоків: реалізація так званого виконавчого інтерфейсу Runnable та розширення класу Thread.

У процесі реалізації інтерфейсу Runnable необхідно:

- створити клас, який реалізує інтерфейс Runnable. Цей інтерфейс містить єдиний метод run(), який необхідно перевизначити. У цьому разі створюється точка входу іншого, конкуруючого потоку, який завершується при завершенні методу run();
- оголосити змінну, яка буде містити об’єкт класу Thread, за допомогою одного з конструкторів;
- викликати метод run () для запуску потоку.

Приклад використання інтерфейсу Runnable:

```
class TNewRunnable implements Runnable
{
	static final short kLimit = 1000;
	private short fLimit = 0;
	public void run()
	{
		while (fLimit++ < kLimit)
			{
			System.out.println("Виконується потік " +
							Thread.currentThread().toString());
			try
			{
				Thread.currentThread().sleep(10);
				} catch (InterruptedException e) {
					System.err.println("Очікування "+ "дочірнього процесу
									перервано.");
					System.exit(1);
					}
			}
		}
	}

public class Example
{
	public static void main(String argv[])
	{
		TNewRunnable aNewRunnable = new TNewRunnable();
		aNewRunnable.run();
		}
	}
```

У процесі розширення класу Thread слід:

- створити клас, який розширює клас Thread. У розширеному класі необхідно перекрити метод run();
- оголосити змінну, яка буде містити об’єкт нового класу;
- викликати метод start() для запуску потоку.

Приклад використання класу Thread:

```
class TNewThread extends Thread{
	static final short kLimit = 1000;
	private short fLimit = 0;
	public TNewThread(String aName) {
		super(aName);
		}

	public void run(){
		while (fLimit++ < kLimit)
			{
			System.out.println("Виконується потік " +
							Thread.currentThread().toString());
			try{
				Thread.currentThread().sleep(10);
				} catch (InterruptedException e) {
```

```
                                    System.err.println("Очікування "+ "дочірнього процесу
                                                                        перервано");
                                    System.exit(1);}
                        }
                  }
            }

public class Example{
      public static void main(String argv[]) {
            TNewThread aNewThread = new TNewThread("Новий потік");
            aNewThread.start();
      }
}
```

Щоб ефективно використовувати потоки, необхідно розуміти їхні різноманітні аспекти й особливості роботи виконавчої системи Java, а саме:

- тіло потоку – як створити тіло потоку;
- стан потоку – життєвий цикл потоку;
- пріоритет потоку – як виконавча система планує виконання потоків;
- потоки-демони – як вони створюються;
- групи потоків – всі потоки повинні міститися в деякій групі потоків.

Тіло потоку реалізується в його методі run(). Тіло потоку можна створити шляхом породження похідного класу від Thread і перевизначення методу run(). Або ж створити клас, який реалізує інтерфейс Runnable. В цьому випадку необхідно створити об’єкт класу Thread і передати йому об’єкт виконавчого інтерфейсу адресата:

updateThread = new Thread(this);

Потік може перебувати в одному із чотирьох станів. Новий потік – стан, у який переходить потік під час створення екземпляру потоку:

Thread myThread=new Thread(this);

Під час цього відбувається розподіл системних ресурсів. Наразі – це порожній об’єкт. Його можна запустити на виконання чи зупинити. Інші методи генерують виняткову ситуацію IllegalThreadStateException.

Виконуваний потік – стан, у який переходить потік під час виклику методу start(). Це означає, що процес може бути виконаний, якщо планувальник надасть йому час процесора. На цей момент він може і не виконуватися, але ніщо не заважає йому виконатися, тобто він не заблокований і не завершений.

Заблокований потік – стан, коли процес може бути запущеним, але не виконується. Допоки процес заблокований, планувальник просто пропускає його і не виділяє для нього час процесора. Цей стан можливий у таких випадках:

- потік був призупинений за допомогою методу suspend(). В цьому випадку він переходить у забуття. Повернути його у виконуваний стан можна за допомогою методу resume(). Використання цих методів не рекомендується, оскільки метод suspend() захоплює блокування об’єкта і можлива ситуація взаємного блокування. Отже, можлива ситуація, коли декілька об’єктів очікують один одного, що викликає зависання програми;
- потік очікує деякий заданий проміжок часу (метод sleep()). Метод sleep() не звільняє блокування;
- потік призупинений за допомогою методу wait(). Вихід із цього стану здійснюється за допомогою методів notify(), notifyAll(), або якщо мине вказаний у методі wait() час очікування. Метод звільняє блокування, а тому під час очікування можуть викликатися інші синхронізовані методи цього об’єкта. Зазначимо, що метод wait()може бути викликаний тільки із синхронізованого методу. В іншому випадку буде згенерований виняток IllegalMonitorStateException;
- потік заблокований іншим потоком (наприклад, потоком, пов’язаним із операцією введення-виведення). У цьому випадку потік вважається невиконуваним, навіть якщо він повністю готовий до виконання.
- потік робить спробу викликати синхронізований метод іншого об’єкта, і блокування цього об’єкта неможливе.

Завершений потік – стан, в який переходить потік під час виклику методу stop() потоку, або у випадку завершенні методу run(). Метод isAlive() дає змогу визначити стан потоку. Якщо він повертає true, то потік був створений і запущений. Наразі він може бути у стані виконання або заблокований. Якщо результат виклику методу false – то потік зупинений або незапущений на виконання.

Щоб призупинити, відновити чи зупинити виконання потоку, необхідно спроектувати його так, щоб метод run() періодично перевіряв, у який стан потік повинен перейти. Цього можна досягти залученням змінної прапорця, яка б вказувала на стан потоку.

Використовує диспетчер потоків для визначення моментів перемикання між потоками. Теоретично потік із вищим пріоритетом отримує більше часу процесора. Однак на практиці все залежить ще й від методу реалізації багатопотоковості в ОС. Крім цього, потоки з вищим пріоритетом можуть призупинити потоки з нижчим пріоритетом. Для потоків з однаковим пріоритетом усе залежить від реалізації багатопотоковості в ОС. Тому потоки з однаковим пріоритетом час від часу повинні віддавати управління головному потоку, щоб дати можливість запуску інших потоків в будь-якій ОС. Для цього застосовується метод yield().

Пріоритети потоків встановлюються за допомогою методу setPriority() у межах MIN_PRIORITY і MAX_PRIORITY:

```
MyThread.setRriority(MAX_PRIORUTY);
```

Пріоритети потоків використовуються для визначення моменту перемикання між потоками за такими правилами:

- потік може передати управління із власної ініціативи. Це відбувається у процесі переходу до стану очікування чи блокування. Тоді опитуються інші потоки, готові до виконання, і потік з найвищим пріоритетом отримує процесор у своє розпорядження. Якщо два потоки мають однаковий пріоритет, то планувальник виконує їх за круговою схемою;
- потік може бути призупинений іншим потоком із вищим пріоритетом (пріоритетна схема з витісненням).

Потоки-демони - це потоки, які підтримують інші потоки. В тілі потоку-демона часто міститься нескінченний цикл, в якому очікується запит від об’єкта чи іншого потоку. Коли приходить запит, потік-демон його обробляє. Щоб зробити потік демоном застосовується метод setDeamon(true). Для визначення, чи є потік демоном, застосовуємо метод isDeamon(), який повертає true, якщо потік – демон.

Всі потоки в Java повинні входити до деякої групи. Під час створення нового потоку можна вказати групу потоків, до якої ввійде створений потік. Для цього передбачено три конструктори класу Thread. Якщо під час створенні потоку не вказується група потоків, то створений потік ввійде до так званої групи main. Групи потоків особливо корисні, оскільки дають змогу керувати цілою групою потоків, тобто призупиняти чи запускати всі потоки групи одночасно.

Таблиця 2. Методи класу ThreadGroup

Оголошення	**Опис**
Конструктори	
ThreadGroup(String name)	Створює групу потоків із заданим іменем name у тій самій групі потоків, що й поточний потік
ThreadGroup(ThreadGroup parent, String name)	Створює групу потоків із заданим іменем name у вказаній батьківській parent групі потоків
Методи екземпляру	
int activeCount()	Повертає приблизну кількість потоків, які належать групі потоків та будь-яким її дочірнім групам потоків
int activeGroupCount()	Повертає приблизну кількість дочірніх груп потоків, які належать групі потоків
boolean allowThreadSuspension(boolean b)	Повертає true, якщо віртуальна машина Java дає змогу призупиняти потоки завдяки низькорівневій пам’яті
final void checkAccess()	Метод закінчується, якщо виконуваний у цей момент потік має дозвіл модифікувати групу потоків
final void destroy()	Руйнує групу потоків та будь-які дочірні групи потоків. У цьому випадку група потоків не повинна містити жодного потоку. Цей метод також вилучає групу потоків із його батьківської групи потоків. Генерує вийняткову ситуацію IllegalThreadStateException, якщо група потоків непорожня або вже знищена
int enumerate(Thread list[])	Зберігає посилання на активні потоки, що належать певній групі потоків або будь-яким дочірнім групам, у масив. Для визначення розміру масиву можна скористатись методом activeCount(). Повертає кількість потоків, які зберігаються у масиві потоків

int enumerate(Thread list[], boolean rec)	Аналогічний до попереднього методу. Змінна rec вказує, чи поміщати потоки із дочірніх груп у масив.
int enumerate(ThreadGroup list[])	Зберігає посилання на активні групи потоків, що належать певній групі потоків або будь-яким дочірнім групам, у масив. Для визначення розміру масиву використовують метод activeGroupCount(). Повертає кількість груп потоків, які зберігаються в масиві.
int enumerate(ThreadGroup list[], boolean rec)	Аналогічний до попереднього. Змінна rec вказує, чи поміщати групи потоки із дочірніх груп у масив.
final int getMaxPriority()	Повертає максимальне значення пріоритету, який може бути призначений потоку, що належить до групи потоків
final String getName()	Повертає ім’я групи потоків
final ThreadGroup getParent()	Повертає групу потоків – батьківську – для певної групи потоків. Якщо ця група потоків міститься у вершині групової ієрархії потоків, то метод повертає null
final boolean isDaemon()	Повертає true, якщо група потоків – демон
synchronized boolean isDestroyed()	Повертає true, якщо група потоків уже знищена
void list()	Виводить список потоків групи
final boolean parentOf (ThreadGroup g)	Повертає true, якщо група потоків є прямим чи непрямим предком вказаної групи g або збігається з нею
final void resume()	Відновлює виконання всіх потоків в певній групі потоків
final void setDaemon(boolean daem)	Змінює стан демона групи потоків
final void setMaxPriority(int pri)	Встановлює максимальне значення пріоритету групі
final void stop()	Зупиняє всі потоки групи потоків та її дочірніх груп
final void suspend()	Призупиняє всі потоки групи потоків та всіх її дочірніх груп потоків
String toString()	Повертає рядкове подання групи потоків

Коли двом чи більше потокам потрібен доступ до ресурсів, необхідно забезпечити доступ не більше ніж одного потоку в кожен момент часу. Процес, за допомогою якого це досягається, називається синхронізацією. Ключем до синхронізації є концепція монітора.

Монітор – це об’єкт, який використовується як взаємно-виключне блокування. Тільки один потік може володіти монітором у певний час. Коли потік блокується, то говорять, що він увійшов в монітор. Всі інші потоки, які захочуть увійти в заблокований монітор, будуть призупинені.

Java забезпечує синхронізацію на рівні мови. Є два способи синхронізації потоків: застосування методів синхронізації та синхронізуючий блок. В Java синхронізація програмується легко, оскільки всі об’єкти мають свої неявні монітори. Щоб зайти в монітор об’єкта, необхідно викликати метод, доповнений ключовим словом synchronized. Для виходу з монітора потік – власник монітора – просто повертається із синхронізованого методу.

Кожного разу, коли застосовується метод чи група методів, які обробляють внутрішній вміст об’єкта в багатопоточній ситуації, необхідно використовувати

ключове слово synchronized. Як тільки потік входить у синхронізований метод деякого екземпляра, жоден інший потік не може ввійти в будь-який інший синхронізований метод цього екземпляра. Однак будь-який інший потік може отримати доступ до несинхронізованих методів цього екземпляра.

Наприклад, програма здійснює правильний вивід завдяки тому, що метод call синхронізований. В іншому випадку вивід програми непрогнозований.

```
	class Callme {
	synchronized void call(String msg) {
		System.out.print("[" + msg);
		try {
			Thread.sleep(1000);
			} catch(InterruptedException e) {
				System.out.println("Interrupted");
				}
		System.out.println("]");
		}
	}

class Caller implements Runnable {
	String msg;
	Callme target;
	Thread t;
	public Caller(Callme targ, String s) {
		target = targ;
		msg = s;
		t = new Thread(this);
		t.start();
		}
	public void run() {
		target.call(msg);
		}
	}

public class Example {
	public static void main(String args[]) {
		Callme target = new Callme();
		Caller ob1 = new Caller(target, "Hello");
		Caller ob2 = new Caller(target,"Synchronized");
		Caller ob3 = new Caller(target, "World");
//очікування завершення потоків
		try {
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
			} catch(InterruptedException e) {
				System.out.println("Перервано");
				}
		}
	}
```

Якщо необхідно синхронізувати доступ до об’єктів класу, який не розроблений для багатопотоковості, або не маємо доступу до коду класу, необхідно використовувати синхронізуючий блок із викликами методів класу:

synchronized (об’єкт){оператори, які необхідно синхронізувати}

де об’єкт – посилання на об’єкт, який синхронізуємо.

Синхронізуючий блок призначений для того, щоб виклик методів об’єкта класу відбувався тільки після того, як потік успішно увійшов у монітор об’єкта.

У процесі синхронізації використовувалось безумовне блокування потоків від асинхронного доступу до деяких методів. Проте є можливість більш тонко керувати процесом з використанням механізму міжпотокових зв’язків.

Для взаємодії між потоками застосовуються методи wait(), notify() і notifyAll() класу Object. Ці методи можна викликати тільки із синхронізованих методів.

- wait() – наказує потоку, який викликав цей метод, віддати монітор і перейти в стан очікування, поки інший потік не ввійде в монітор і не викличе метод notify().
- notify() – активізує перший потік, який викликав метод wait() на тому ж об’єкті.
- notifyAll() – активізує всі потоки, які викликали wait() одного і того ж об’єкта. Першим запускається потік з найвищим пріоритетом.

Існують перевантажені версії методу wait(), які дають змогу вказати максимальний період часу очікування.

Помилка взаємного блокування виникає у випадку, коли два потоки мають циклічну залежність від пари синхронізованих методів. Нехай один потік входить у синхронізований метод об’єкта X, а інший – у синхронізований метод об’єкта У. Якщо потік у методі X викликає будь-який синхронізований метод об’єкта У, то він буде заблокований. Якщо ж потік у методі об’єкта У викликає синхронізований метод об’єкта X, то він нескінченно чекатиме.

Практичні завдання

1. Виконайте наведені у теоретичній частині приклади, а їх результати занесіть до звіту.
2. Напишіть послідовну програму додавання двох матриць розміром 1000x1000 з випадковими значеннями. Дослідить можливість виконання цієї задачі з використанням паралельної парадигми Java. Порівняйте час виконання послідовної та паралельної реалізації.
3. Напишіть послідовну програму множення двох матриць розміром 1000x1000 з випадковими значеннями. Дослідить можливість виконання цієї задачі з використанням паралельної парадигми Java. Порівняйте час виконання послідовної та паралельної реалізації.
4. Підготуйте звіт з виконання завдань лабораторної роботи.