Додаток 4 ЕЛЕМЕНТИ МОВИ OBJECT PASCAL

1. МОДУЛЬ В OBJECT PASCAL

Мова об'єктно-орієнтованого програмування Object Pascal застосовується під час роботи у середовищі візуального програмування Delphi і Lazarus. Мова Object Pascal в основному включає "стару" мову Borland Pascal.

Програми мовою Object Pascal складаються з кількох файлів: файлу проекту (Delphi Project) з розширенням *.dpr, одного або кількох файлів модулів (Unit) з розширенням *.pas та файлів дизайнера екранних форм із розширенням *.dfm.

Файл проекту містить текст основної програми Program, з якої починається виконання всієї програми. Тексти підпрограм і об'єктів, що використовуються, знаходяться у файлах модулів.

Розглянемо організацію вихідного тексту модуля:

```
unitMyUnit1;

interface

uses
Unit1, Unit2, Unit3;

const
Pi =3.14;
type
MyType = . . .;
var
var1: MyType;

procedureProc1;
functionFunc: MyType;

implementation

uses
Unit4, Unit5, Unit6;

const
...;
type
...;
var
...;

procedureProc1;
begin
{ Операторі}
...
end;

functionFunc: MyType;
begin
{Операторі}
...
end;

initialization
{Операторі}
...

finalization
{Операторі}
...
end.
```

Модуль починається з описового оператора заголовка модуля:
```
unit MyUnit1;
```
Імена файлів MyUnit1.pas, MyUnit1.dfm повинні збігатися з ім'ям, описаним у заголовку модуля MyUnit1. Наявність файлу MyUnit1.dfm не є обов'язковою.
Між зарезервованими словами interface та implementation знаходяться описові оператори секції інтерфейсу. В інтерфейсній частині оголошуються константи, типи, змінні, прототипи процедур і функцій (тільки оператор заголовка без операторів), які повинні бути доступні для використання в інших модулях. Описи підключень інших модулів здійснюються за допомогою оператора uses, який може розташовуватися за оператором interface. Імена модулів, що підключаються, повинні бути розташовані в такому порядку, щоб забезпечити послідовний опис всіх потрібних типів в даній інтерфейсній секції і інтерфейсних секцій модулів, що підключаються.
За зарезервованим словом implementation знаходяться описові оператори секції реалізації. На відміну від описів секції інтерфейсу, описи з секції реалізації недоступні для інших модулів, але доступ до них можливий з даного модуля. Як і в секції інтерфейсу, може слідувати оператор uses, а за ним оголошення констант, типів, змінних. На відміну від секції інтерфейсу процедури та функції описуються разом з їх операторами. У секції повинні бути визначені як процедури та функції із секції реалізації, так і процедури та функції, прототипи яких були оголошені у секції інтерфейсу. Код процедур та функцій модуля, описаний у секції implementation, підключається (лінкується або зв'язується) до основної програми або точок виклику підпрограм та функцій в інших модулях (через механізм лінкера, робота якого визначається uses).
У необов'язковому розділі initialization розміщуються оператори, які виконуються відразу після запуску програми.
Розділ finalization не є обов'язковим, більше того, він може бути присутнім тільки в тому випадку, якщо в модулі була секція initialization. У секції розміщуються оператори, які виконуються безпосередньо до завершення програми.

2. ОБ'ЄКТИ І КЛАСИ В МОВІ OBJECT PASCAL

У звичайній мові Pascal існує тип-запис:
```
type
TmyRecord = record
MyField1: String;
MyField2: Integer;
end;
```
Тип-запис дозволяє описувати структуровані змінні, які містять кілька значень як одного, і різних типів. У наведеному прикладі запис TmyRecord містить поля MyField1 та MyField2, які відповідно мають типи String та Integer.
Тип-клас в Object Pascal на вигляд близький до запису, але відрізняється від запису можливістю успадкування від інших класів, а також можливістю опису методів класу. Класом в Object Pascal називається особливий тип запису, який може мати у своєму складі поля, методи та властивості. Такий тип також називатимемо об'єктним типом:
```
type
TMyObject = class(TObject)
MyField: Integer;
Procedure MyMethod1 (X: Real; var Y: Real);
function MyMethod2: Integer;
end;
```
У прикладі описаний клас TMyObject, який успадковується від класу Tobject. Поняття «спадкування класів» та поняття «властивості» будуть детально розглянуті далі. Поки можна визначити поняття властивості як поле, яке доступне не безпосередньо, а через надсилання повідомлень особливим методам.
Клас TMyObject має поле MyField та методи MyMethod1 та MyMethod2. Потрібно загострити увагу, що класи можуть бути описані або в секції інтерфейсу модуля Interface (під модулем тут розуміється файл з вихідним кодом виду Unit), або на верхньому рівні вкладеності секції реалізації Implementation. Не допускається опис класів усередині процедур та інших блоків коду.
Якщо клас включає поле з типом іншого класу, дозволено випереджаюче оголошення класу як у наступному прикладі:
```
type
```

```
TFirstObject = class;
TSecondObject = class (TObject)
Fist: TFirstObject;
{...}
end;
TFirstObject = class(TObject)
{...}
end;
```

Код методів описується нижче за текстом об'яв класів, наприклад:

```
Procedure TMyObject.MyMethod1(X: Real; var Y: Real);
begin
y := 5.0 * sin(X);
end;
function TMyObject.MyMethod2: Integer;
begin
{...}
MyMethod2: = MyField + 3;
end;
```

Для того щоб використовувати новий тип у програмі, потрібно, як мінімум, оголосити змінну цього типу, яка називається або змінною об'єктного типу, або екземпляром класу, або об'єктом:

```
var
AMyObject: TMyObject;
```

Відповідно до звичайної мови Borland Pascal, змінна AMyObject повинна містити в собі весь екземпляр об'єкта типу TMyObject (код та дані разом) – статичний об'єкт. Але в Delphi всі об'єкти динамічні, тому, не вдаючись до подробиць, виконаємо оператор:

```
{Дія зі створення екземпляра об'єкта}
AMyObject := TMyObject.Create;
```

Тепер інші об'єкти програми можуть надсилати повідомлення даному об'єкту. Надсилання повідомлень полягає у викликі методів потрібного об'єкта, наприклад:

```
var
До: Integer;
{...}
AMyObject.MyMethod1(2.3, Z);
До: = 6 + AMyObject.MyMethod2;
```

Методи- Це процедури та функції, описані всередині класу. Як видно, надсилання повідомлень у Object Pascal близьке до виклику процедур мови Pascal, але імені процедури, що викликається, або процедури-функції передує ім'я конкретного об'єкта, наприклад: AMyObject.

Опишемо два об'єкти AMyObject, BMyObject одного класу TMyObject:

```
var
AMyObject,
BMyObject: TmyObject;
```

При проектуванні програмісти вважають, кожен об'єкт (примірник класу) має власний внутрішній код методів і індивідуальну пам'ять, де розміщуються поля.

Насправді методи різних об'єктів одного класу загальні. Інакше кажучи, вони реалізуються загальним кодом, розташованим лише одному місці пам'яті. Це заощаджує пам'ять. У наведеному прикладі виклик методів:

```
AMyObject.MyMethod1(2.3, Z);
BMyObject.MyMethod1(0.7, Q)
```

насправді призведе до виконання одного й того ж коду при видимості, що моделюється для програміста, приналежності свого індивідуального коду різним об'єктам. Це зближує методи з процедурами та процедурами-функціями мови Pascal. Нагадаємо, що покажчик - це змінна, що містить у пам'яті адресу (номер осередку) іншої змінної, процедури або об'єкта. До складу класу входить покажчик на спеціальну таблицю, де міститься вся інформація, необхідна виклику методів. Від звичайних процедур та функцій методи відрізняються тим, що при виклику передається (неявно) покажчик на той об'єкт, який їх викликав. Усередині методів він доступний під зарезервованим Self.

Засилання та вилучення значень у поля, відповідно до прямого доступу, що не рекомендується в Object Pascal, практично не відрізняється від використання полів запису звичайної мови Pascal:

```
AMyObject.MyField := 3;
I := AMyObject.MyField + 5.
```

На відміну від методів поля об'єкта, це дані, унікальні для кожного об'єкта, що є екземпляром навіть одного класу. Поля AMyObject.MyField та BMyObject.MyField. є зовсім різними полями, оскільки вони розміщуються у різних об'єктах.

3. ОБЛАСТИ ВИДІМОСТІ

При описі нового класу важливим є розумний компроміс. З одного боку, потрібно приховати методи і поля, що є внутрішнім пристроєм класу. Незначні деталі на інших рівнях будуть марними і лише завадять цілісності сприйняття. Доступ до важливих деталей необхідно організувати через систему перевірок.

У мові Object Pascal введено механізм доступу до складових частин об'єкта, визначальний області, де можна використовувати (тобто області видимості). Поля та методи можуть належати до чотирьох груп, що відрізняються областями приховування інформації:

• *public* -загальні;
• *private* -особисті;
• *protected*-захищені;
• *published*- Опубліковані.

Розподіл на складові працює на рівні файлів модулів (Unit у сенсі мови Pascal). Якщо ви потребуєте спеціального захисту об'єкта або його частини, то для цього необхідно помістити його в окремий модуль, в якому є власні секції interface та implementation.

Поля, властивості та методи, що знаходяться у секції public, не мають обмежень на видимість. Вони доступні з інших функцій і методів об'єктів як в даному модулі, так і в інших, що посилаються на нього. Зазвичай методи цієї секції утворюють інтерфейс між об'єктним обміном повідомленнями під час виконання програми (run-time).

Поля, властивості та методи, що знаходяться в секції private, доступні тільки в методах класу та функціях, що містяться в тому ж модулі, що і клас, що описується. Така директива дозволяє приховати деталі внутрішньої реалізації класу всіх. Елементи з секції private можна змінювати, і це не позначатиметься на програмах, що працюють із об'єктами цього класу. Єдиний спосіб для когось іншого – звернутися до них – переписати заново створений вами модуль.

Розділ protected комбінує функціональне навантаження розділів private і public таким чином, що якщо ви хочете приховати внутрішні механізми вашого об'єкта від кінцевого користувача, цей користувач не зможе в run-time використовувати жодне з об'яв об'єкта з його protected-області. Але це не завадить розробнику нових компонентів використовувати ці механізми в інших успадкованих класах, тобто protected-оголошення доступні у будь-якого зі спадкоємців вашого класу.

Розділ published виявився необхідним при введенні в Object Pascal можливості встановлення властивостей та поведінки компонентів ще на етапі конструювання форм і самої програми (design-time) у середовищі візуальної розробки програм Delphi. Саме published-оголошення доступні через Object Inspector, чи це посилання на властивості або обробники подій. Під час виконання програми розділ об'єкта published повністю аналогічний public.

Слід зазначити той факт, що при породженні нового класу шляхом успадкування можливе перенесення оголошень з одного розділу в інший з єдиним обмеженням: якщо ви робите приховування оголошення за рахунок його перенесення в private, в подальшому його «витягування» у спадкоємця в більш доступний розділ в іншому модулі буде вже неможливо. Таке обмеження, на щастя, не поширюється на динамічні методи обробки повідомлень Windows.

Приклад опису класу із заданими областями видимості наведено нижче.

4. ІНКАПСУЛЯЦІЯ

Класичне правило об'єктно-орієнтованого програмування стверджує, що для забезпечення надійності небажаний прямий доступ з інших об'єктів до полів об'єкта: читання та оновлення їхнього вмісту повинно здійснюватися через виклик відповідних методів. Це і називається інкапсуляцією. Досі ідея інкапсуляції впроваджувалась у програмування лише за допомогою закликів та прикладів у документації, але в мові ж Object Pascal з'явилася відповідна конструкція. В об'єктах Object Pascal користувач об'єкта може бути повністю відгороджений з його полів за допомогою властивостей.

Роботу з властивостями розглянемо на прикладі. Нехай ми створили за допомогою дизайнера форм Delphi екранну форму Forml із двома елементами візуальних компонентів: Button 1 та Label 1 (рис. 1).

*Мал. 1.*Екранна форма прикладу

Натисніть кнопку Button1 і відредагуємо вихідний текст модуля до наступного тексту:

```
unittestir;
interface
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
type
TForm1 = class(TForm)
Button1: TButton;
Label1: TLabel;
procedureButton1Click(Sender: TObject);
private
{Private declarations}
public
{Public declarations}
end;
type
TSomeType = String;
type
TAnObject = class(TObject)
private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
end;
var
Form1: TForm1;
AnObject: TAnObject;
implementation
{$R *.DFM}
procedureTForm1.Button1Click(Sender: TObject);
begin
AnObject := TAnObject.Create;
AnObject.AProperty := 'Привіт!';
Label1.Caption := AnObject.AProperty;
end;
procedureTAnObject.SetAProperty(
ANewValue: TSomeType);
begin
FValue: = ANewValue; {Засилання значення у полі}
end;
functionTAnObject.GetAProperty: TSomeType;
begin
GetAProperty: = FValue; {Читання значення з поля}
end;
end.
```

Збережемо проект (Save Project As). При збереженні проекту вкажемо нове ім'я модуля – testir та нове ім'я проекту – PrTestir. Розглянемо текст файлу проекту, що вийшов (пункти меню View і далі Project Source):

```
programPrTestir;
uses
Forms,
testir in 'testir.pas' {Form1};
{$R *.RES}
begin
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.Run;
end.
```

Цей файл містить текст основної програми PrTestir. До основної програми підключаються модуль Forms (для роботи з формою) та вихідний код модуля testir. Три оператори основної програми, що виконуються, послідовно організують новий обчислювальний процес виконання написаної програми PrTestir, створять об'єкт форми Form1, здійснять запуск програми на виконання (Run).
У прикладі текст модуля містить згенерований текст оголошення об'єктного типу Tform1. Тип містить вказівки на агрегацію в даному класі об'єкта кнопки Button1: Tbutton і об'єкта Label1: Tlabel. Завдяки агрегації екземпляри об'єктів кнопки та написи будуть створюватися одночасно зі створенням екземпляра об'єкта форми, в результаті чого як би вийде об'єкт форми, що спільно працює з кнопкою і написом. У типі Tform1 Delphi по подвійному клацанню миші по кнопці <Button1> згенерувала прототип виклику методу:
```
procedureButton1Click(Sender: TObject).
```
Також Delphi автоматично згенерувала змінну об'єктного типу
```
var
Form1: TForm1;
```
і в секції реалізації вставила текст "порожній" процедури Button1Click відпрацювання дій із натискання кнопки Button1.
Розглянемо елементи, додані нами у текст модуля реалізації інкапсуляції. Отже, нами було набрано текст опису класу TAnObject та змінна даного об'єктного типу AnObject:
```
type
TAnObject = class(TObject) private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
end;
var
AnObject: TanObject.
```
Зазвичай властивість (property) визначається трьома своїми елементами: полем та двома методами, які здійснюють його запис/читання:
```
private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
...
procedureTAnObject.SetAProperty(ANewValue: TSomeType);
begin
FValue: = ANewValue; {Засилання значення у полі}
end;
functionTAnObject.GetAProperty: TSomeType;
begin
GetAProperty: = FValue; {Читання значення з поля}
end;
```
В даному прикладі властивість Aproperty виконує таку саму функцію, яку в попередньому прикладі виконувало поле MyField. Доступ до значення властивості Aproperty здійснюється через виклики методів GetAProperty та SetAProperty. Однак у зверненні до цих методів у явному вигляді немає необхідності (у прикладі вони навіть захищені — protected), достатньо написати:
```
AnObject.AProperty: = ...;
... := AnObject.Aproperty;
```
та компілятор відтранслює ці оператори на виклики методів.
Розглянемо три оператори, вписані в текст "порожній" процедури (метод) Button1Click:
```
AnObject := TAnObject.Create;
AnObject.AProperty := 'Привіт!';
Label1.Caption := AnObject.AProperty;
```

Перший оператор створює екземпляр об'єкта. Другий оператор у недоступне за інтерфейсом (protected) поле Fvalue посилає за допомогою недоступної за інтерфейсом (protected) процедури AnObject.SetAProperty текст «Привіт!». Третій оператор за допомогою недоступної за інтерфейсом (protected) процедури АпОЬject.GetAProperty зчитує дані з недоступного за інтерфейсом поля Fvalue і посилає їх як Caption об'єкта написи Label1.
Сама властивість AProperty описано як загальнодоступне за інтерфейсом (public), тобто зовні властивість виглядає в точності, як доступ до звичайного поля, але за будь-яким зверненням до якості можуть стояти необхідні програмісту дії. Наприклад, якщо у нас є об'єкт, що є квадратом на екрані, і ми його властивості «колір» присвоюємо значення «білий», то відбудеться негайне перемальовування, що приводить реальний колір на екрані у відповідність до значення властивості.
У методах, що входять до складу властивостей, може здійснюватися перевірка величини, що встановлюється, на потрапляння в допустимий діапазон значень і виклик інших процедур, що залежать від внесених змін. Якщо ж потреби у спеціальних процедурах читання та/або запису немає, то, можливо, замість імен методів застосовувати імена полів.
Розглянемо таку конструкцію:

```
type
TPropObject = class(TObject)
FValue: TSomeType;
procedureDoSomething;
procedureCorrect(AValue: Integer);
procedureSetValue(NewValue: Integer);
procedureAValue: Integer read Fvalue
writeSetValue;
end;
...
procedureTPropObject.SetValue(NewValue: Integer);
begin
if(NewValue <> FValue) and Correct (NewValue)
then
FValue: = NewValue; {Засилання значення у полі}
DoSomething;
end;
```

У цьому прикладі читання значення властивості AValue означає просто читання поля FValue. Зате при присвоєння йому значення всередині методу SetValue викликається одразу два методи.
Якщо властивість повинна тільки читатися або записуватися, то в його описі може бути тільки відповідний метод:

```
type
TAnObject = class(TObject)
propertyAProperty: TSomeType read GetValue;
end;
```

У цьому прикладі поза об'єктом значення якості можна лише прочитати. Спроба встановити значення Aproperty викликає помилку компіляції.

5. ОБ'ЄКТИ ТА ЇХ ЖИТТЄВИЙ ЦИКЛ

Як створюються та знищуються об'єкти? У Object Pascal екземпляри об'єктів можуть бути лише динамічними! Це означає, що у наведеному на початку розділу фрагменті змінна AMyObject хоч і виглядає як статична змінна мови Pascal, насправді є вказівником, що містить адресу об'єкта. Будь-яка змінна об'єктного типу є покажчик (покажчик — змінна, що містить значення адреси оперативної пам'яті).
Пам'ять під конкретні об'єкти (динамічні екземпляри класів) виділяється диспетчером пам'яті в період виконання програми в особливій heap-області, де має бути вільне місце для розміщення нових об'єктів. Heap - "купа сміття". Диспетчер пам'яті може розміщувати в heap-області нові об'єкти, так і видаляти вже непотрібні, звільняючи пам'ять під всі нові об'єкти. Саме при розміщенні об'єкта в пам'яті ініціалізується значення змінної типу об'єкта об'єкта. При видаленні об'єкта змінна об'єктного типу ініціалізується значенням nil (порожній покажчик) - немає об'єкта пам'яті. При розміщенні нового об'єкта в пам'яті може виявитися, що загальна вільна пам'ять має великий обсяг, але будь-який з ділянок пам'яті, що звільнилися, займаних вже віддаленими об'єктами, виявляється менше обсягу нового великого об'єкта. Для уникнення такої ситуації при використанні об'єктних програм встановлюють в

ЕОМ оперативну пам'ять свідомо великого обсягу. Новий екземпляр об'єкта створюється особливим методом – конструктором, а знищується спеціальним методом – деструктором:

```
AMyObject := TMyObject.Create; {дії зі створеним об'єктом}
...
AMyObject.Destroy; {знищення об'єкта}
```

У Object Pascal конструкторів у класу може бути кілька. Загальноприйнято називати конструктор Create. Типова назва деструктора - Destroy. Рекомендується використовувати для знищення екземпляра об'єкта метод Free, який спочатку перевіряє покажчик (чи не дорівнює він nil) і потім викликають Destroy.
Для того щоб правильно проініціалізувати в об'єкті поля, що створюється, відносяться до класу-предка, потрібно відразу ж при вході в конструктор викликати конструктор предка:

```
constructorTmyObject.Create;
begin
inheritedCreate;
...
end;
```

Метод створення об'єкта MyObject типу TMyObject успадкований від класу предка TObject.
Взявши будь-який із прикладів, що постачаються разом із Delphi, ми виявимо, що там майже немає викликів конструкторів та деструкторів. Справа в тому, що будь-який компонент, що потрапив при візуальному проектуванні в програму з палітри компонентів, включається до певної ієрархії. Ієрархія ця замикається на формі (TForm): всім її складових частин конструктори і деструктори викликаються автоматично, незримо для програміста.
Хто створює та знищує форми? Це робить програму (глобальний об'єкт з ім'ям Application). У файлі проекту (.DPR) можна побачити виклик функції CreateForm, призначений для цієї мети. Що ж до об'єктів, створюваних динамічно (під час виконання докладання), тут потрібен явний виклик конструктора.

6. СПАДЧИНА

Другим «стовпом» ОВП, крім інкапсуляції, є успадкування. Цей простий принцип означає, що якщо потрібно створити новий клас, який лише трохи відрізняється від старого, то немає потреби в переписуванні заново вже існуючих полів і методів. Новий клас

```
TNewObject = class(TOldObject);
```

є нащадком, чи дочірнім класом старого класу, званого предком, чи батьківським класом. Додаються до нього лише нові поля, методи та властивості.
У Object Pascal усі класи є нащадками класу TObject. Тому, якщо будується дочірній клас прямо від TObject, то у визначенні TOject можна не згадувати. Наступні два вирази однаково вірні:

```
TMyObject = class(TObject);
TMyObject = class;
```

Використання останнього висловлювання виправдане, якщо розробник хоче показати, що, за його задумом, клас, що проектується, як би не має предків.
Наведемо оголошення базового всім об'єктних типів класу TObject:

```
TObject = class
constructorCreate;
destructorDestroy; virtual;
procedureFree;
class functionNewlnstance: TObject; virtual;
procedureFreelnstance; virtual;
class procedureInitlnstance(Instance: Pointer):
TObject;
functionClassType: TClass;
class functionClassName: string;
class functionClassParent: TClass;
class functionClassInfo: Pointer;
class functionInstanceSize: Word;
class functionInheritsFrom(AClass: TClass):
Boolean;
procedureDefaultHandler(var Message); virtual;
procedureDispatch (var Message);
class functionMethodAddress(const Name: string):
Pointer;
class functionMethodName(Address: Pointer):
string;
```

```
функція FieldAddress(const Name: string):
Pointer;
end;
```

Така архітектура можлива лише за наявності механізму підтримки інформації про типи – RTTI (RunTime Type Information). Основою такого механізму є внутрішня структура класів і, зокрема, можливість доступу до неї за рахунок використання методів класів, що описуються конструкцією class function...

Успадковані від предка поля та методи доступні у дочірньому класі; якщо має місце збіг імен методів, ці методи перекриваються.

По тому, які дії відбуваються під час виклику, методи поділяються на групи:

• статичні (static);
• віртуальні (virtual);
• динамічні (dynamic);
• абстрактні (abstract).

Статичні методи, а також поля в об'єктах-нащадках поводяться однаково: можна без обмежень перекривати старі імена і змінювати тип методів:

```
type
T1stObj = class
I: Real;
procedureSetData(Avalue: Real);
end;
T2ndObj = class(T1stObj)
I: Integer;
procedureSetData(Avalue: Integer);
end;
...
procedureT1stObj.SetData;
begin
i: = v;
end;
procedureT2nd0bj.SetData;
begin
i := 0;
inheritedSetData(0.99);
end;
```

У цьому прикладі різні методи з ім'ям SetData надають значення різним полям з ім'ям і. Перекрите поле предка недоступне нащадку. На відміну від поля всередині інших методів, перекритий метод доступний при вказівці зарезервованого слова bherited. Методи об'єктів за умовчанням статичні — їх адреса визначається ще на стадії компіляції проекту. Вони викликаються найшвидше.

Мова C++ дозволяє так зване множинне спадкування. У цьому випадку новий клас може успадковувати частину своїх елементів від одного батьківського класу, а частина від іншого, це поряд із зручностями часто призводить до проблем.

У Object Pascal поняття множинного спадкування відсутнє. Якщо необхідно, щоб новий клас поєднував властивості кількох, можна породити предки один від одного або включити в клас кілька полів, що відповідають цим бажаним класам.

Принципово відрізняються від статичних методів віртуальні та динамічні методи. Вони можуть бути оголошені шляхом додавання відповідної директиви virtual чи dynamic. Адреса таких методів визначається під час виконання програми за спеціальною таблицею. З погляду наслідування методи цих двох видів однакові: вони можуть бути перекриті в дочірньому класі лише однойменними методами, що мають той самий тип.

Спільним їм є те, що з їх виклику адреса визначається під час компіляції, а під час виконання шляхом пошуку у спеціальних таблицях. Такий спосіб ще називається пізнім зв'язуванням. Різниця між методами полягає особливо у пошуку адреси.

Коли компілятор зустрічає звернення до віртуального методу, він підставляє замість звернення до конкретної адреси код, який звертається до спеціальної таблиці та витягує звідти потрібну адресу. Ця таблиця називається таблицею віртуальних методів (Virtual Method Table, VMT) і є для кожного об'єктного типу. У ній зберігаються адреси всіх віртуальних методів класу незалежно від того, чи вони успадковані від предка чи перекриті. Звідси і переваги, і недоліки віртуальних методів: вони

викликаються порівняно швидко (але повільніше статичних), проте для зберігання покажчиків на них потрібна велика кількість пам'яті.
Динамічні методи викликаються повільніше, але дозволяють більш економно витрачати пам'ять. Кожному динамічному методу системою надається унікальний індекс. У таблиці динамічних методів (Dynamic Method Table, DMT) класу зберігаються індекси та адреси тільки тих динамічних методів, які описані в даному класі (базова інформація з обробки динамічних методів міститься в модулі x:\delphi\source\rtl\sys\dmth.asm). Під час виклику динамічного методу відбувається пошук у цій таблиці. У разі невдачі проглядаються всі класи-предки в порядку ієрархії і, нарешті, TObject, де є стандартний обробник виклику динамічних методів. Економія пам'яті очевидна.
Для перекриття і віртуальних, і динамічних методів служить нова директива override, за допомогою якої (і тільки з нею!) можна перевизначати обидва типи методів:

```
type
TFirstClass = class
FMyField1: Integer;
FMyField2: Longlnt;
procedureStatMethod1;
procedureVirtMethod1; virtual;
procedureVirtMethod2; virtual;
procedureDynaMethod1; dynamic;
procedureDynaMethod2; dynamic;
end;
TSecondClass = class(TFirstClass)
procedureStatMethod;
procedureVirtMethod; override;
procedureStatMethod; override;
end;
var
Obj2: TFirstClass;
Obj1: TSecondClass;
```

Перший із методів у прикладі створюється заново, решта — перекривається. Спроба застосувати override до статичного методу викликає помилку компіляції.
У Object Pascal абстрактними називаються методи, які визначені у класі, але не містять жодних дій, ніколи не викликаються і мають бути перевизначені у нащадках класу. Абстрактними можуть бути лише віртуальні та динамічні методи. Для цього використовується директива abstract, що вказується при описі методу:

```
procedureNeverCallMe; virtual; abstract;
```

При цьому жодного коду для цього писати не потрібно. Як і раніше, виклик методу NeverCallMe призведе до помилки часу виконання.

7. ПОЛІМОРФІЗМ

Розглянемо приклад, який пояснює, навіщо потрібне використання абстрактних методів. Нехай у нас є якесь узагальнене поле для зберігання даних — клас TField і три його нащадки — для зберігання рядків, цілих та дійсних чисел. У разі клас Tfield не використовується сам собою; його основне призначення - бути родоначальником ієрархії конкретних класів - «полів» і дати можливість абстрагуватися від частковостей. Хоча параметр у ShowData описаний як TField, але якщо помістити туди об'єкт цього класу, відбудеться помилка виклику абстрактного методу:

```
type
TField = class
functionGetData: string; virtual; abstract;
end;
TStringField = class(TField)
Data: string;
functionGetData: string; override;
end;
TIntegerField = class(TField)
Data: Integer;
functionGetData: string; override;
end;
TExtendedField = class(TField)
Data: Integer;
function GetData: string; override;
```

```
end;
functionTStringField.GetData;
begin
GetData: = Data;
end;
functionTIntegerField.GetData;
begin
GetData := IntToStr(Data);
end;
functionTExtendedField.GetData;
begin
GetData := IntToStrF(Data, ffFixed, 7, 2);
end;
...
procedureShowData(AField: TField);
begin
Form1.Label1.Caption := AField.GetData;
end;
```

У цьому прикладі класи містять різнотипні дані, які «вміють» лише повідомити про значення цих даних текстовим рядком (за допомогою методу GetData). Зовнішня щодо них процедура ShowData отримує об'єкт як параметр і показує цей рядок.

Правила контролю відповідності типів (typecasting) мови Pascal свідчать, що об'єкту як покажчику на екземпляр об'єктного типу може бути присвоєно адресу будь-якого екземпляра будь-якого з дочірніх типів. У процедурі ShowData параметр описаний як TField. Це означає, що в неї можна передавати об'єкти класів і TStringField, і TIntegerField, і TExtendedField, і будь-якого іншого нащадка TField.

Але який (точніше, чий) метод GetData буде викликаний? Той, що відповідає класу фактично переданого об'єкта. Цей принцип називається поліморфізмом, і він, мабуть, є найбільш важливим «козирем» ОВП. Припустимо, є справа з деякою сукупністю явищ чи процесів. Щоб змоделювати їх засобами ОВП, необхідно виділити їх найзагальніші, типові риси. Ті з них, які не змінюють свого змісту, мають бути реалізовані як статичні методи. Ті ж, які варіюють при переході від загального до приватного, краще надати форму віртуальних методів. Основні «родові» риси (методи) потрібно описати в класі-предку і потім перекрити їх у класах-нащадках. У попередньому прикладі програмісту, який пише процедуру на кшталт ShowData, важливим є лише одне: те, що будь-який об'єкт, переданий у неї, є нащадком TField, і він вміє повідомити про значення своїх даних (виконавши метод GetData). Якщо таку процедуру скомпілювати та помістити в динамічну бібліотеку, то цю бібліотеку можна буде раз і назавжди використовувати без змін, хоча з'являтимуться і нові, невідомі в момент її створення класи-нащадки TField!

Наочний приклад використання поліморфізму дає сама Delphi. У ній є клас TComponent, лише на рівні якого зосереджені певні «правила» того, як взаємодіяти з середовищем розробки та іншими компонентами. Дотримуючись цих правил, можна породжувати від TComponent свої компоненти, налаштовуючи Delphi рішення спеціальних завдань.

8. ОБРОБКА ПОВІДОМЛЕНЬ

Потреба динамічних методів особливо відчутна розробки об'єктів, відповідних елементам інтерфейсу Windows, коли кожен із великої ієрархії об'єктів містить обробники десятків різноманітних повідомлень.

Методи, призначені спеціально для дня обробки повідомлень Windows, становлять підмножину динамічних методів і оголошуються директивою message, за якою слідує індекс — ідентифікатор повідомлення. Вони повинні бути обов'язково описані як процедури, що мають один var-параметр, який може бути описаний довільно, наприклад:

```
type
TMyControl = class(TWinControl)
procedureWMSize (var Message: TWMSize);
messageWM_SIZE;
end;
type
TMyOtherControl = class(TMyControl)
procedureResize(var Info);
messageWM_SIZE;
end;
```

Для перекриття методів обробки повідомлень директива override не використовується. У цьому випадку слід зберегти в описі директиву message з індексом методу.
Необхідності винаходити власні структури, які по-своєму інтерпретують зміст того чи іншого повідомлення, немає: для більшості повідомлень Windows типи вже описані в модулі MESSAGES.
В обробниках повідомлень (і тільки в них) можна викликати метод-предок, просто вказавши ключове слово inherited без вказівки його імені та перетворення типу параметрів: предок буде знайдений за індексом. Слід нагадати, що система обробки повідомлень вбудована в Object Pascal на рівні моделі об'єктів, і найзагальніший обробник метод DefaultHandler описаний в класі TObject.

9. ПОДІЇ І ДЕЛЕГУВАННЯ

Працювати з великою кількістю повідомлень, навіть маючи під рукою довідник, нелегко, тому одним із великих досягнень Delphi є те, що програміст позбавлений необхідності працювати з повідомленнями Windows (хоча така можливість у нього є). Стандартних подій у Delphi не більше двох десятків, і всі вони мають просту інтерпретацію, яка не потребує глибоких знань середовища.
Розглянемо, як реалізовані події лише на рівні мови Object Pascal. Події - це властивості процедурного типу, призначені для створення користувальницької реакції на ті чи інші вхідні дії:

```
property OnMyEvent : TMyEvent read FonMyEvent
write FonMyEvent;
```

Привласнити таку властивість значення — це означає вказати об'єкту адресу методу, який буде викликатись у момент настання події. Такі методи назвемо обробниками подій. Наприклад:

```
Application.OnActive := MyActivatingMethod;
```

Це означає, що при кожній активізації Application (так називається об'єкт, що відповідає працюючому додатку) буде викликаний метод-обробник MyActivatingMethod.
Всередині бібліотеки часу виконання Delphi виклики обробників подій перебувають у методах, що обробляють повідомлення Windows. Виконавши принципово необхідні дії, цей метод перевіряє, чи відома адреса оброблювача, і якщо це так, викликає його:

```
if Assigned(FonMyEvent)
then
FonMyEvent(Self);
```

Залежно від походження та призначення події мають різні типи. Спільним для всіх є параметр Sender, що вказує на об'єкт-джерело події. Найпростіший тип - TNotifyEvent - не має інших параметрів:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Тип методу, призначений для сповіщення про натискання кнопки, передбачає передачу програмісту коду цієї кнопки, а пересування миші — її координат тощо.
Всі події в Delphi прийнято називати з On: OnCreate, OnMouseMove, OnPaint і т. д. Клацнувши в Інспекторі об'єктів на сторінці Events в полі будь-якої події, автоматично виходить заготівля методу потрібного типу. При цьому його ім'я складатиметься з імені поточного компонента та імені події (без On), а відноситися він буде до поточної форми. Нехай, наприклад, у формі Form1 є текст Label1. Тоді для обробки клацання мишею (подія OnClick) буде створено метод Tform1. Label1Click.
Оскільки події є властивостями об'єкта, їх значення можна змінювати під час виконання програми. Така чудова нагода називається делегуванням. Можна в будь-який момент взяти способи реакції на події одного об'єкта і делегувати їх іншому:

```
Object.OnMouseMove := Object2.OnMouseMove;
```

Але який механізм дозволяє замінювати обробники, адже це не просто процедури, а методи? Тут до речі доводиться введене в Object Pascal поняття покажчика на метод. Крім явно описаних параметрів методу передається ще й покажчик на екземпляр (Self), що його викликав. Ви можете описати тип процедури, яка буде сумісна з присвоєння методом (тобто передбачати отримання Self). Для цього до її опису потрібно додати зарезервовані слова of Object. Покажчик на метод – це покажчик на таку процедуру:

```
type
TmyEvent = procedure(Sender: TObject;
varAvalue: Integer) of object;
T1stObject = class;
FOnMyEvent: TMyEvent;
propertyOnMyEvent: TMyEvent read FonMyEvent
writeFonMyEvent;
end;
T2ndObject = class;
procedureSetValue1(Sender: TObject;
```

```
varAvalue: Integer);
procedureSetValue2(Sender: TObject;
varAvalue: Integer);
end;
...
var
Obj1: T1stObject;
Obj2: T2ndObject;
begin
Obj1: = T1stObject.Create;
Obj2: = T2ndtObject.Create;
Obj1.OnMyEvent := Obj2.SetValue1;
Obj2.OnMyEvent := Obj2.SetValue2;
...
end;
```

Як у цьому прикладі, так і всюди Delphi за властивостями-подіями стоять поля, що є покажчиками на метод. Таким чином, при делегуванні можна надавати методи інших класів. Тут обробником події OnMyEvent об'єкта Obj1 по черзі виступають методи SetValue1 та SetValue2 об'єкта Obj2.

10. ФУНКЦІЇ КЛАСУ

Object Pascal має можливість визначення полів процедурного типу. Очевидно, що в тілі функцій, що прив'язуються до цих полів, розробнику необхідний доступ до інших полів об'єкта, методів і т.п. . Такі функції називаються функціями класів. Для оголошення функцій класів необхідно використовувати спеціальну конструкцію function... of object.

11. ПРИВЕДЕННЯ ТИПІВ

На операції зі змінною певного типу компілятор зазвичай накладає обмеження, дозволяючи виконання лише операцій, характерні для зазначеного типу даних. Іноді компілятор здійснює автоматичне наведення типу, наприклад, при присвоєнні цілого значення дійсної змінної.

У мові Pascal є механізм очевидного приведення типів.

В операції is визначається, чи належить цей об'єкт зазначеному типу чи одному з його нащадків. Вираз, наведений у наступному прикладі, повертає True, якщо змінна AnObject посилається на зразок об'єктного типу TMyClass або одного з його нащадків.

```
AnObject is TmyClass
```

Сама собою операція is не є операцією завдання типу. У ній лише перевіряється сумісність об'єктних типів. Для коректного приведення типу об'єкта застосовується операція as:

```
WithAnObject як TmyClass do ...
```

Можливий такий спосіб приведення типу без явного вказівки as.

```
WithTMyClass(AnObject)do ...
```

У програмах перед операцією as перевіряють сумісність типів з допомогою операції is. Якщо типи несумісні, запускається обробник виключної ситуації EinvalidCast.

Таким чином, у конструкції as операція явного приведення типу виявляється укладеною в безпечну оболонку:

```
IfAnObject is TobjectType then
withTobjectType(AnObject) do ...
else
raiseEinvalidCast.Create('Неправильне приведення типу');
```

12. ОБ'ЄКТНЕ ПОСИЛАННЯ

Delphiдозволяє створити спеціальний описник об'єктного типу (саме тип, а чи не на екземпляр!), відомий як object reference — об'єктна посилання.

Об'єктні посилання використовуються у таких випадках:

- Тип створюваного об'єкта не відомий на етапі компіляції;
- Необхідний виклик методу класу, чий тип не відомий на етапі компіляції;
- В якості правого операнда в операціях перевірки та приведення типів з використанням is і as.

Об'єктне посилання визначається з використанням конструкції class of... . Наведемо приклад оголошення та використання class reference:

```
type
TMyObject = class (TObject)
MyField:TMyObject;
constructorCreate;
end;
```

```
TObjectRef = class of TObject;
...
var
ObjectRef:TObjectRef;
s:string;
begin
ObjectRef:=TMyObject; {Привласнюємо тип, а не екземпляр!}
s:=ObjectRef.ClassName; {рядок s містить 'TMyObject'}
end;
```

Таким чином, Delphi визначено спеціальне посилання TClass, сумісна з присвоєння з будь-яким спадкоємцем TObject. Аналогічно оголошені класи: TPersistentClass і TCотponentClass.

13. СТРУКТУРНА ОБРОБКА ВИКЛЮЧНИХ СИТУАЦІЙ

Під винятковою ситуацією (raise) тут розуміється ситуація, яка дозволяє без особливих додаткових заходів продовжити виконання програми, наприклад розподілу на нуль, переповнення розрядної сітки, вилучення квадратного кореня з негативного числа тощо.

При традиційній обробці помилок, помилки, виявлені в процедурі, зазвичай передаються назовні (в викликав процедуру) у вигляді значення функції, параметрів або глобальних змінних (прапорів), що повертається. Кожна процедура, що викликає, повинна перевіряти результат виклику на наявність помилки і виконувати відповідні дії. Часто це просто вихід у більш верхню процедуру, що викликає, і т.д.

Структурна обробка виняткових ситуацій — це програмний механізм, що дозволяє програмісту у разі виникнення помилки (виключної ситуації — exception) зв'язатися з кодом програми, підготовленим для обробки такої помилки. У Delphi система називається структурною, оскільки обробка помилок визначається областю «захищеного» коду. Такі області можуть бути вкладені. Виконання програми не може перейти на довільну ділянку коду. Виконання програми може перейти лише на обробник виняткової ситуації активної програми.

Модель виняткових ситуацій в Object Pascal є невідновлюваною (non-resumable). При виникненні виняткової ситуації ви вже не зможете повернутися в точку, де вона виникла, для продовження виконання програми (це дозволяє зробити лише відновлювану модель).

Для обробки виняткових ситуацій до мови Object Pascal додано нове ключове слово «try», яке використовується для позначення першої частини захищеної ділянки коду. Існують два типи захищених ділянок:

1) try..except;
2) try..finally.

Перший тип використовується для обробки виняткових ситуацій. Його синтаксис:

```
try
Statement1;
Statement2;
...
except
on Exception1 do Statement;
on Exception2 do Statement;
...
else
Statements; {default exception-handler}
end;
```

Для впевненості в тому, що ресурси, зайняті вашою програмою, звільняться в будь-якому випадку, можете використовувати конструкцію другого типу. Код, розташований у частині finally, виконується у будь-якому разі, навіть якщо виникає виняткова ситуація. Відповідний синтаксис:

```
try
Statement1;
Statement2;
finally
Statements; {These statements always execute}
end;
```