Вступ. Історія розвитку та сучасний стан мов програмування

Тема 1. Об'єктно-орієнтована методологія розробки програмних комплексів

ТЕМА 2. Об'єктна модель в Delphi Pascal

Класи та об'єкти

За визначенням Граді Буча ***об'єкт*** – відчутна сутність, що чітко виявляє свою поведінку. Кожний об'єкт характеризується властивостями (розмір, колір, смак і таке інше) та діями (ходить, їсть, сидить і таке інше). За збігом основних ознак ми можемо виділити групи об'єктів – класи. ***Клас*** – це сукупність об'єктів одного типу, тобто схожих за своїми властивостями та призначенням. Іншими словами, кожен об'єкт є екземпляром певного класу.

З точки зору програмної реалізації об'єкт – це сукупність полів даних та методів (процедур або функцій) їх обробки. Таким чином, при описі програмного об'єкту ми задаємо перелік полів та описуємо заголовки методів. В середовищі Delphi об'єкт описується так:

```
    type
TMyObj= class
            field 1: Type1;
            field 2: Type2;
            procedure M1 (P: Byte);
            function M2 (D:real): real;
    end;
```

де field 1, field 2 – поля заданого типу; M1, M2 – методи об'єкту.

Згідно принципу спадкування кожен об'єкт має бути нащадком якогось класу. За замовчанням в середовищі Delphi клас TObject - спільний предок для всіх об'єктів. За загальною нотацією опису класів після ключового слова **class** у дужках наводиться клас – предок. Опис

```
    type
        T MyObj= class (TObject)
```

```
    field1: Type1;
    field2: Type2;
    procedure M1 (P: Byte);
    function M2 (D:real): real;
end;
```

ідентичний опису, що був наведений вище.

Таким чином, всі описані в DelphiPascal класи успадковують всі поля і методи класу TObject (див. додаток 1).

В DelphiPascal всі об'єкти динамічні. Для їх створення та знищення використовуються конструктори та деструктори.

В деяких об'єктно-орієнтованих мовах оголошення змінної, що має тип класу, автоматично створює екземпляр цього класу. Object Pascal заміст цього використовує об'єктну модель посилань (object reference model). Ідея полягає в тому, що кожна змінна типу класу містить не значення об'єкту, а лише посилання (reference) або покажчик (pointer), що посилається на область пам'яті, в якій міститься об'єкт.

Єдина проблема цього підходу полягає в тому, що при оголошенні змінної в пам'яті не створюється сам об'єкт, а лише резервується місце для посилання на нього. Екземпляри об'єктів повинні створюватися вручну – по крайній мірі об'єкти тих класів, які визначаєте ви самі. Екземпляри компонента, що розташований на формі, автоматично створюється Delphi.[3]

Конструктор та деструктор

Для створення екземпляру об'єкту використовують спеціальний метод, який називається ***конструктором*** (constructor) і має ім'я Create. Призначення конструктора Create:

- виділення пам'яті для об'єкта шляхом виклику методу NewInstace класу TObject (див. додаток 1), який в свою чергу викликає метод InstanceSize для визначеного розміру пам'яті, потрібної для об'єкту.
- ініціалізація полів об'єкту нулями (метод InstInstance).

Конструктор Create не є віртуальним і в класі TObject не має параметрів. В інших стандартних класах (і в класах користувача) він може бути перевизначений. Наприклад, в класі T Object → Constructor Create, а в класі T Component → Constructor Create (A Owner: T Component); virtual;

При описі класу для завдання конструктора використовується спеціальне ключове слово – Constructor.

Таким чином, щоб створити екземпляр об'єкту треба викликати конструктор його класу (метод Create) – TMyObj.Create. Вас може здивувати, що при описі класу про конструктор і не згадувалось, але не слід забувати

про спадкування! В даному випадку буде викликаний конструктор предка – TObject.Create.

Всі, сподіваємося, пам'ятають жорстке правило роботи з динамічним розподілом пам'яті:

Взяв пам'ять – поклади на місце!

Отже, якщо існує спеціальний метод для виділення пам'яті під об'єкт, необхідно бути методу для її звільнення. Такий метод називається деструктором (destructor). Деструктор Destroy не має параметрів та виконує такі дії:

- виклик метода CleanupInstance.
- виклик InstanceSize для визначення розміру об'єкту, що видаляється.

Найчастіше для знищення об'єкта використовується не сам деструктор, а метод Free, який також спадкується об'єктами від TObject. Цей метод спочатку перевіряє, чи не дорівнює покажчик на об'єкт NIL, якщо ні, то викликає деструктор цього об'єкту.

ТЕМА 3. Класифікація методів об'єкту у середовищі Delphi

Статичні та заміщувані методи

Методи об'єктів можна розподілити на дві основні групи: *статичні* та *заміщувані*. Заміщувані методи розподіляються на: *віртуальні*, *динамічні* та *абстрактні*.

Статичні методи це регулярні звичайні методи об'єкту (В мові програмування C++ та Java статичними називаються класові методи).

Реалізація принципу поліморфізму надає можливість об'єктам – нащадкам перекривати (заміщати) методи об'єкта – предка. Це реалізується через введення при описі класу–нащадка одноіменних методів. Цей механізм називається перекриттям методу. Для чого ж тоді заміщувані методи?

Принципова різниця між статичними та заміщуваними методами полягає в тому, що:

- при описі у класі–предку заміщуваний метод описується з директивою virtual або dynamic. У класі–нащадку він перевизначається директивою override. Наприклад,

  procedure VirtProc; **virtual**;
  procedure DynamProc; **dynamic**;

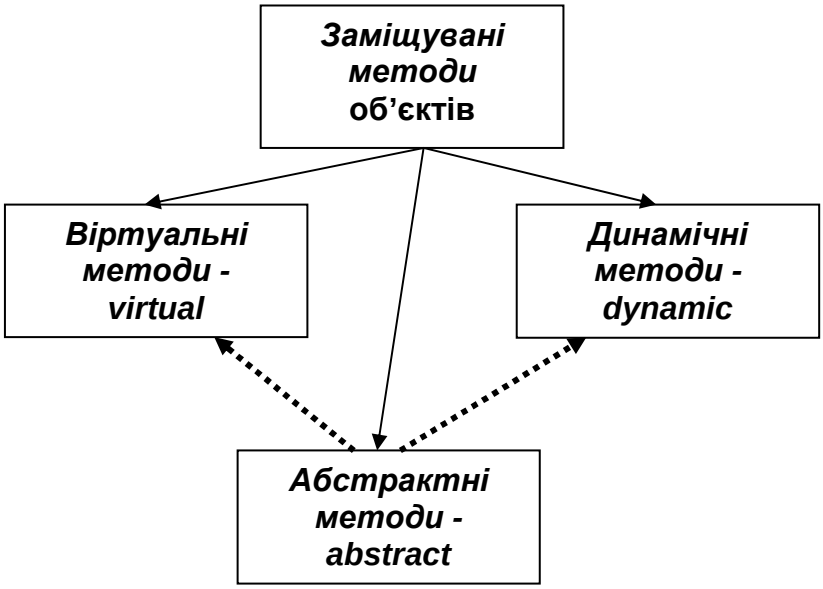


Рис. 1. Заміщувані методи об'єкта

function VirtFunc: type1; **virtual**;
function DynamFunc: type2; **dynamic**;

- для заміщуваних методів застосовується пізнє зв'язування з екземпляром класу, а для статичних – раннє;
- заміщувані методи дають можливість об'єктам – батькам викликати методи об'єктів - нащадків.

Для статичних методів компілятор визначає адресу і "передає" її об'єкту. Таким чином зв'язок об'єктів зі своїми статичними методами відбувається на етапі компіляції ("раннє зв'язування").

Настроювання об'єкта на заміщуваний метод класу відбувається під час ініціалізації екземпляра об'єкта. Коли компілятор зустрічає звернення до заміщуваного метода, він підставляє замість звернення до конкретної адреси код, який звертається до спеціальної таблиці, звідки потім витягається потрібна адреса. [Дарахвелідзе] Отже для заміщуваних методів зв'язування відбувається під час виконання (ініціалізації об'єкта) - "пізнє зв'язування" через спеціальні таблиці: таблиці віртуальних та динамічних методів (VMT та DMT).

Віртуальні та динамічні методи

Різниця між віртуальними (virtual) та динамічними (dynamic) методами полягає в структурі відповідних даним методам таблиць.

Для віртуальних методів в кодовому сегменті формується таблиця віртуальних методів (Virtual Method Table - VMT), яка містить посилання на

віртуальні методи. Така таблиця існує для кожного класу об'єктів. До неї заносяться адреси всіх віртуальних методів класу, не залежно від того успадковані ці методи від предка чи перекриті.

Кожному динамічному методу системою присвоюється унікальний індекс. В таблиці динамічних методів (Dynamic Method Table - DMT) класу зберігаються індекси та адреси тільки тих динамічних методів, які описані у даному класі. Під час виклику динамічного методі відбувається пошук в цій таблиці, в разі невдачі передивляються всі класи – предки за встановленою ієрархією та, наприкінці, клас TObject, де визначений стандартний оброблювач викликів динамічних методів. [Дарахвелідзе]

Тому диспетчеризація віртуальних методів відбувається швидше ніж динамічних (але повільніше за статичні), для зберігання покажчиків на віртуальні методи потребується більше пам'яті.

Диспетчеризація – порядок виклику застосуванням методів об'єктів, тобто визначення, який програмний код треба виконати під час виклику методів того чи іншого об'єкта.

Віртуальні методи у програмі визначаються директивою **virtual**, а динамічні – **dynamic**.

З погляду спадкування ці методи однакові. Для перевизначення функціональності одноіменних однотипних методів предка в класі – нащадку використовується директива **override**. Спроба застосувати директиву override для статичного метода викликає помилку. Спроба перевизначення віртуальних або динамічних методів не директивою override, а virtual або dynamic, призведе до утворення нового одноіменного методу, що порушить поліморфну ієрархію класів.

Розглянемо приклад. Декларуються два класи TParentClass та його нащадок TSonClass. Батьківський клас має два методи First та Second, при чому з першого метода викликається другий. У класі-нащадку другий метод перекривається (виконується повторна декларація цього метода).

```
Type
    TParentClass = class
        Procedure First;
        Procedure Second; virtual;
    End;
    TSonClass = class (TParentClass )
            Procedure Second; virtual;
    End;
```

В розділі implementation для цих методів задана така реалізація:

```
Procedure TParentClass. First;
  Begin
  Second;
  End;
Procedure TParentClass. Second;
  Begin
  Show Message('Procedure TParentClass.Second Done');
  End;
Procedure TSonClass. Second;
  Begin
  Show Message('Procedure TSonClass.Second Done');
End;
```

Для перевірки роботи цих класів на формі необхідно розташувати дві кнопки: одну – для виклику метода First батьківського класу, а другий для виклику цього ж метода з класу-нащадка. Після запуску застосування та почергового виклику цих методів з'являться, нажаль, два однакових повідомлення: 'Procedure **TParentClass**.Second Done'. А як же перекриття методів? Розглянемо схему виклику перекритих методів (рис. 2, 1). В класі TSonClass метод First спадкується від предка. Тому при його виклику буде викликаний метод предка. А предок, в свою чергу викличе свій метод Second. Отже перекриття віртуального (або динамічного метода) без використання директиви override не дає можливість класу-пердку викликати методи класів-нащадків.

Спробуємо для класу TSonClass змінити декларацію метода на:

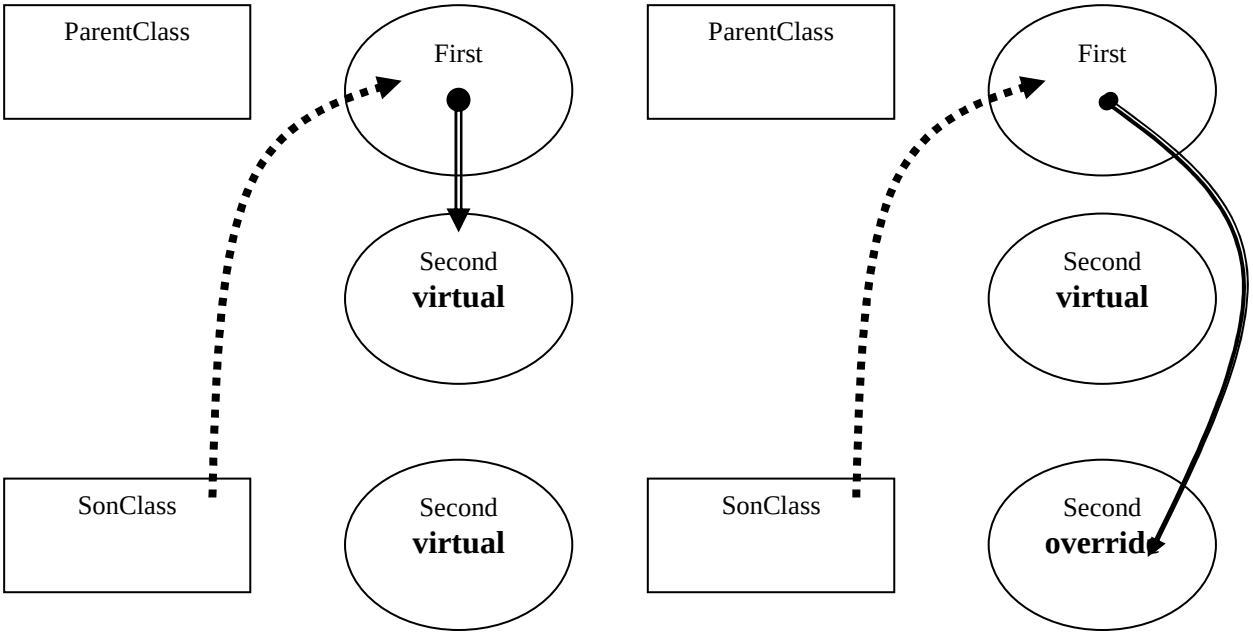
Procedure Second; **override**;

Після запуску застосування та виклику батьківського метода з'явиться повідомлення: 'Procedure **TParentClass**.Second Done'. Для класу-нащадку повідомлення буде: 'Procedure **TSonClass**.Second Done'. Схема виклику подана на рис.2, 2.

Директива inherited

Виникають ситуації, коли з класу-нащадку необхідно викликати одноіменний метод предка, що був заміщений (або перекритий) цим нащадком. Для цього використовується спеціальна директива ***inherited***. Загальний опис використання такий:

inherited < ім'я методу предка>;



1) Схема виклику перекритих методів

2) Схема виклику заміщуваних методів

Рис. 2. Схеми виклику перекритих та заміщуваних методів об’єктів

Абстрактні методи

Перший метод в ієрархії заміщуваних методів може бути абстрактним. ***Абстрактні методи*** – це методи, які не мають реалізації для даного класу об’єктів, але мають бути реалізовані в об’єктах нащадках. Абстрактні методи описуються директивою abstract та мають бути віртуальними або динамічними (дивися рис.1). Звернення (виклик) абстрактного для даного класу методу призводить до виникнення помилки. Метою введення абстрактних методів є надання родоначальнику ієрархії певних властивостей, що будуть успадковуватися, без впадання в подробиці реалізації (підтримка принципу абстрагування). Директива abstract може використовуватися тільки в класі, де метод об'являється перший раз.

Цей формальний опис робить незаконним визначення методу у даному класі.

```
type
 TSample=class
              ...
      procedure SomeMethod; virtual; abstract;
 end;
```

Класові методи

Класові методи можна викликати без створення екземпляру класу, а через посилання на клас. Хоча класові методи можна викликати також з екземпляра об'єкту, але реалізація класових методів не може посилатися на конкретне поле, або інший некласовий метод. На конструктор та на інші класові методи посилатися можна. Класові методи, зазвичай, модифікують глобальні дані або видають інформацію про клас.

Опис класових методів здійснюється за допомогою ключового слова class, що ставиться перед заголовком процедури або функції:

```
Type
    TSample = class
      Class Function GetClassName:string;
    End;
Var
    Obj:TSample;
    S1, S2:string;
Class Function TSample.GetClassname:string;
Begin
  Result:='The Sample Class';
End;
```

Виклик класового методу може здійснюватися двома способами:

```
S1:=TSample.GetClassName; //без створення екземпляру класу
```

або

```
Obj:= TSample.Create; //зі створенням екземпляру класу
S2:=Obj.GetClassName;
Obj.Free;
```

Синтаксис класового методу накладає обмеження на виклик методів предка. Можливо викликати тільки метод безпосередньо предка, використовуючи Inherited.

Тема 4. Внутрішня структура об'єкта

Реалізація TVM та TDM

Розглянемо приклад. [Дарахвелідзе]

```
Type
    TFirstClass= class
        FmyField1: integer;
        FmyField2: Longint;
```

```
        Procedure StatMethod;
        Procedure VirtMethod1; virtual;
        Procedure VirtMethod2; virtual;
        Procedure DynaMethod1; dynamic;
        Procedure DynaMethod2; dynamic;
    End;
    TSecondClass= class
        Procedure StatMethod;
        Procedure VirtMethod1; override;
        Procedure DynaMethod1; override;
    End;
Var
    Obj1: TFirstClass;
    Obj2 : TSecondClass;
```

Перший клас TFirstClass за замовчанням є нащадком від TObject. Він включає два поля та п’ять методів: один статичний, два віртуальних і два динамічних. У другому класі TSecondClass перевизначається статичний метод, перекриваються один віртуальний та один динамічний. Що це означає і як це відобразиться на структурі об’єкта? Розглянемо рис. 3.

Об’єкти (або екземпляри класів TFirstClass та TSecondClass) Obj1 та Obj2 є покажчиками на область пам’яті, яка розподілена для збереження полів об’єкту. Як видно з рисунка, ця область містить копії полів класу. Крім того кожний екземпляр класу містить покажчик на його клас (розмір – 32 байта), де сконцентрована вся інформація безпосередньо про клас та посилання на методи. Інформація про клас це спеціальна 32 – байтна структура, яка інакше називається ***інформація про тип під час виконання (runtime type information – RTTI)***. Вона містить всі дані про клас: його ім’я, розмір екземпляру, покажчик на клас – предок, адресу таблиці динамічних методів та інше (детальніше див. розділ “Інформація про тип під час виконання (RTTI). Оператори *is* та *as* ”).

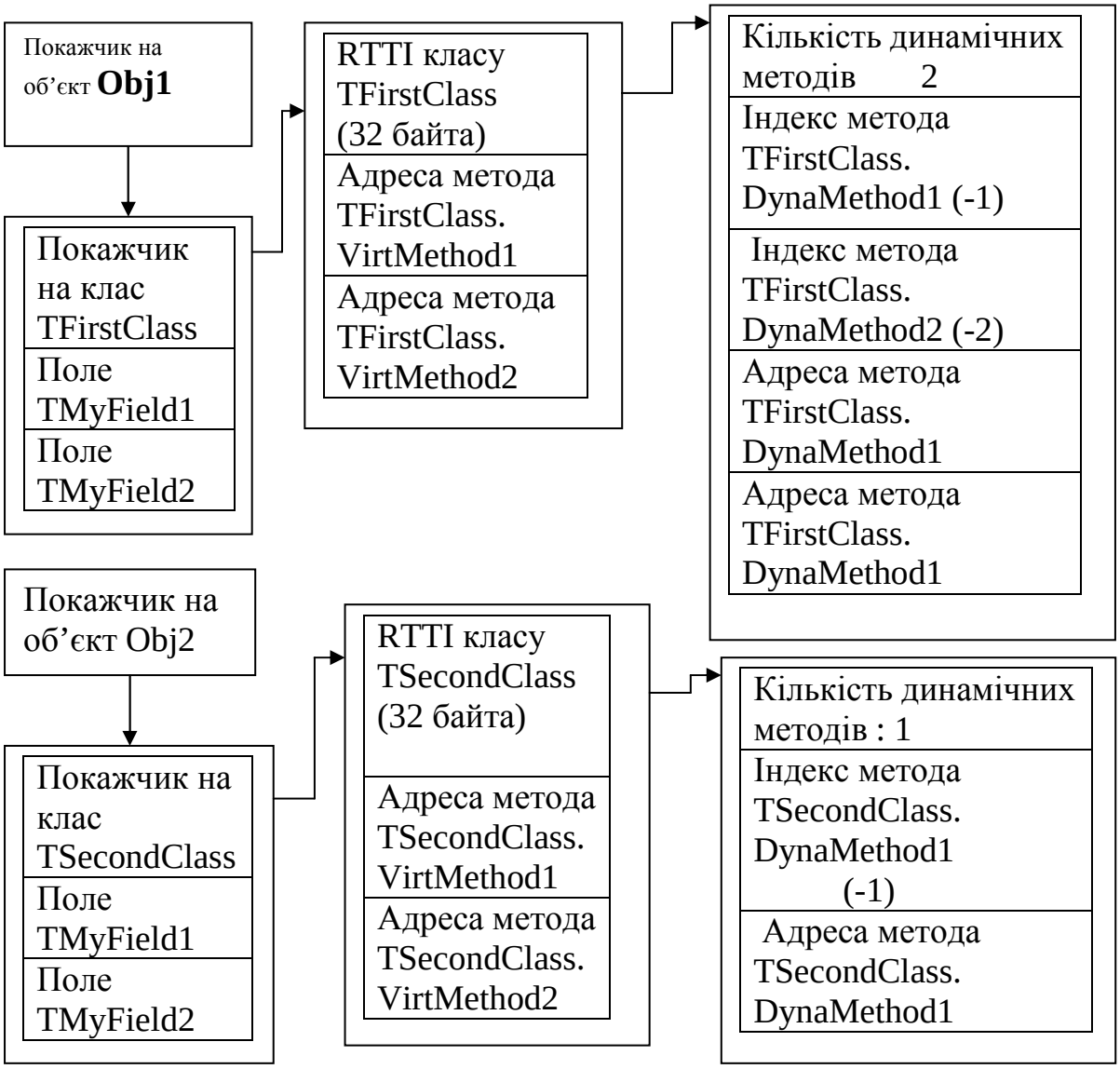


Рис. 3. Внутрішня структура методів Obj1 та Obj2

Далі розташована таблиця віртуальних методів (ТВМ) класу, яка містить адреси всіх віртуальних методів для даного класу. Для виклику віртуального метода компілятор генерує код, що виконує безумовний перехід за адресою, що вказана у відповідній комірці цієї таблиці. ТВМ дає можливість швидкого виклику метода, але недоліком є те, що для кожного дочірнього класу в його ТВМ копіюються посилання на всі віртуальні методи предків, навіть якщо вони не були перевизначені, що значно збільшує об'єм пам'яті для збереження ТВМ. На рисунку 3 структура ТВМ для TFirstClass та TSecondClass однакова. Адреси VirtMethod2 для цих класів будуть однакові, але адреси для VirtMethod1 будуть різні.

На відміну від віртуальних посилання на динамічні методи більш економно розташовуються в пам'яті. Для кожного класу у таблиці динамічних методів (ТДМ) зберігається посилання тільки на методі

перевизначені для даного класу. Крім того виклик цих методів здійснюється за допомогою унікального номера (індексу), що визначає метод. Відповідно до цього пошук динамічного методу здійснюється довше, ніж віртуального. Але для глибоких ієрархій класів використання динамічних методів замість віртуальних призводить до значної економії пам'яті при невеликій втраті у швидкості.

Перед програмістом постає проблема вибору: що краще віртуальний або динамічний метод? Ось декілька правил [М. Кенту]:

- якщо метод скоріше за всього буде перевизначений майже всіма нащадками, він має бути віртуальним;
- якщо метод, буде перекриватися не дуже часто, але вимагає пізнього зв'язування для більшої гнучкості, зробіть його динамічним, особливо якщо для класу планується значна кількість нащадків;
- якщо метод буде викликатися дуже часто, багато разів на секунду, зробіть його віртуальним. В інших випадках відмінність в швидкості виклику віртуальних та динамічних методів не має принципового значення.

Інформація про тип під час виконання (RTTI)

Розглянемо структуру RTTI:

RTTI= record

VtTypeInfo : word	короткий покажчик на додаткову інформацію про клас
VtFieldTable : word	короткий покажчик на таблицю інформації про поля
VtMetodTable : word	короткий покажчик на таблицю інформацію про методи класу
VtDynamicTable: word	короткий покажчик на таблицю динамічних методів
VtClassName : word	короткий покажчик на рядок з ім'ям класу
VtInstSize :word	Розмір екземпляра класу
VtParent : Pointer	Покажчик на клас – предок
VtDefault : Pointer	Покажчик на метод обробки повідомлень Windows
VtNewIst : Pointer	Покажчик на конструктор
VtFreeInst : Pointer	Покажчик на метод знищення екземпляра класу
VtDestroy : Pointer	Покажчик на деструктор

End;

Примітка ! Поняття короткого покажчика пов’язане з тим, що додаткова інформація про клас має бути розташована в одному з ним сегменті. Покажчик типа Pointer містить адресу сегмента та зсув від його початку.

В Delphi інформація RTTI відіграє важливу роль і може використовуватися програмістом явно чи неявно.

Маючи справу з поліморфізмом та ієрархією класів, часто виникає необхідність визначити тип об'єкту, на який вказує покажчик об'єкту. [Конопка]

В BP7 існує функція TypeOf, яка повертає покажчик на ТВМ об'єкту, значення якого можна порівняти з тим, що нас цікавить. Однак, сумісність класів по оператору присвоєння розповсюджується на нащадки. Якщо необхідно перевірити тільки сумісність по присвоєнню, то необхідно перевірити усі нащадки.

Оператори is та as

В Delphi оператор ***is*** забезпечує доступ до RTTI для визначення того, чи є тип об'єкту типом даного класу, або одним з його нащадків.

is - логічна операція, операндами якої є екземпляр об'єкту і класовий тип. Результат операції булового типу.

Наприклад:

```
Procedure Tform1.SpeedButtonClick (Sender : TObject);
Begin
  If  Sender is TCastomEdit Then
            TCustomEdit  (Sender ).CopyToClipBoard;
End;
```

У прикладі перевіряється чи є поточний елемент управління нащадком TCastomEdit. Якщо він сумісний (наприклад TEdit, TMemo), то викликається метод CopyToClipboard.

Delphi використовує оператор ***as*** для приведення типу. Напис TCastomEdit (ActiveControl) аналогічний напису ActiveControl ***as*** TCastomEdit, але на відміну від простого перетворення типів оператор ***as*** на початку перевіряє чи сумісні ці типи під час виконання. Якщо перетворення зробити не можливо, то збуджується особлива ситуація EInvalidCast. Після застосування оператора ***as*** самий об'єкт не змінюється, але викликаються ті його методи, як якби він належав до приведеного класу.

[Дарахвелидзе] Дуже корисно може бути використання оператора ***as*** у методах обробки подій. Для забезпечення сумісності джерело події *Sender* має тип TObject, хоча реально ними можуть бути інші компоненти. Тому,

щоб мати можливість користуватися їхніми методами застосовують ***as***:

(Sender as TControl).Caption := ‘Thanks’;

Недоліком цих операторів є те, що присвоюваний тип має бути відомим вже на етапі компіляції.

Інформація RTTI може використовуватися незалежно від того створено екземпляр класу чи ні. В такому випадку доступ до RTTI здійснюється через покажчик на клас або виклик класових методів.

Тема 5. Регламентація доступу до полів та методів класу

Директиви видимості

При описі класів використовуються таки директиви видимості, які регламентують доступ до полів та методів об’єкту з зовні (з інших модулів, класів, класів - нащадків):

- private – “особисті” поля, властивості та методи є доступними тільки в методах класу і в функціях, що описані в одному з ним модулі. Ця директива дозволяє приховати деталі внутрішньої реалізації об’єкту. Доцільно вносити в область видимості private поля, властивості та методи, які ніколи не будуть використовуватись або модифікуватись користувачами або нащадками, а тільки забезпечують функціонування класу. Це суто внутрішні поля, властивості та методи.
- protected – “захищені” поля, властивості та методи є доступними тільки з класів, які є нащадками даного класу, в тому числі і з інших модулів.
- public – “загальнодоступні” поля, властивості та методи не мають обмежень на видимість. Вони є доступними з інших модулів.
- published – “опубліковані” властивості використовуються при розробці компонентів. Це ті властивості, що ми бачимо в Object Inspector, вони є загальнодоступними властивостями.

Загальний опис класів набуває такого вигляду:

Type
<Ім’я класу> = **class** (<клас – предок>)
private
приватні поля;
приватні методи;
приватні властивості;
protected

захищені поля;
захищені методи;
захищені властивості;

public

загальнодоступні поля;
загальнодоступні методи;
загальнодоступні властивості;

published

опубліковані поля ;
опубліковані методи ;
опубліковані властивості;

end;

Одним з найважливіших принципів об’єктно-орієнтованого підходу є принцип інкапсуляції. Отже введення директив видимості для полів та методів об’єкту дозволяє розробнику реалізовувати цей теоретичний принцип на практиці. В розширеному розумінні принцип інкапсуляції полягає в тому, що для забезпечення надійного функціонування об’єкту небажаний прямий доступ до полів даних: запис і оновлення їх змісту повинні виконуватися через відповідні методи. Розширене розуміння інкапсуляції знайшло своє відображення в понятті властивості.

Властивості об’єкту

Розглянемо приклад. Визначити клас для цілочисельної змінної. Задати метод для зведення значення змінної у квадрат.

Розпочнемо з аналізу полів та методів майбутнього класу змінної TVariable. Одне поле необхідне для зберігання цілочисельного значення змінної, тому його можна описати типом integer:

Value : integer;

В іншому полі зберігатиметься ім’я змінної. Згідно з правилами мови програмування назва змінної – це послідовність символів, яка не може починатися з цифри та включати спеціальні символи. Довжина ідентифікатора не може перевищувати 63 символи, тому для опису поля обирається тип string [63].

Name : string [63];

Згідно з завданням для класу необхідно створити метод, який зводить значення змінної у квадрат.

procedure Square;

Крім цього необхідно описати конструктор, в якому буде проводитись ініціалізація полів класу:

constructor Create (V:integer; N:string);

Опис класу набуває такий вигляд:

```
Type
TVariable = class
   Value: integer;
   Name:string [63];
   constructor Create (V:integer; N:Name);
   procedure Square;
end;
```

Отже отримуємо клас, в якому два поля і два методи. Як кожний митець програміст завжди переймається за судьбу свого створіння. А що трапиться, якщо користувач введе некоректні дані? Наприклад, задасть ім'я змінної, що суперечить правилам формування ідентифікаторів. Для того щоб уникнути цих неприємностей можливо додати до класу спеціальний метод, назвемо його SetName, в якому будуть виконуватися необхідні перевірки. Аналогічний метод можна придумати і для іншого поля.

DelphiPascal надає для захисту полів спеціальний програмний механізм, що називається ***властивістю*** (property). Загальний синтаксис опису властивості такий:

```
Property <ім'я_властивості>: <тип_значення>
        read <ім'я_методу_для_зчитування>
              write <ім'я_методу_для_запису>
                    default <значення_за_замовчанням>;
```

Властивість – це зовнішній шар для обробки поля (рис. 4). Властивість має ім'я та тип. Тип властивості може збігатися з типом поля, з яким вона пов'язана. Для організації зв'язку властивості з полем використовуються два методи: метод для зчитування значення поля та метод для запису значення поля.

Метод для зчитування повинен бути функцією без параметрів. Тип результату функції повинен співпадати з типом властивості.

Метод для запису повинен бути процедурою з одним параметром. Тип параметра повинен співпадати з типом властивості.

Опишемо поля класу приватними та зробити властивості для доступу до них. Тоді опис класу буде:

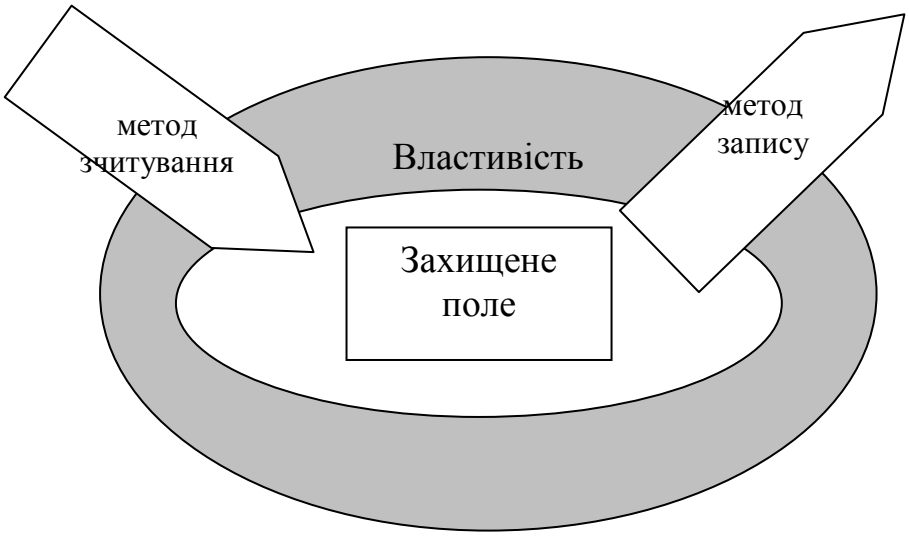


Рис. 4. Механізм доступу до поля через властивість

```
Type
TVariable = class
private
   FValue: integer;
   FName:string [63];
protected
//методи для зчитування полів FName та FValue
function GetName :string;
function GetValue :integer;
//методи для запису полів FName та FValue
procedure SetValue (V:integer);
procedure SetName (N:string);
public
//властивості для доступу до  полів FName та FValue
property Value:integer read GetValue write SetValue;
property Name:string read GetName write SetName;

   constructor Create (V:integer; N:Name);
   procedure Square;
end;
```

Розглянемо реалізацію методів класу.

```
function TVariable.GetName: string;
begin
  Result:= FName;
end;
```

```
function TVariable.GetValue: integer;
begin
  Result := FValue;
end;

procedure TVariable.SetName(N: string);
 var
  i:byte;
  Digit:set of '0'..'9';
  Good:boolean;
begin
Good:=false;
if not (N[1] in Digit) then
 for i:=1 to length (N) do
 begin
  if not((N[i] >= 'a') and (N[i] <= 'z') or  // не буква
         (N[i] >= 'A') and (N[i] <= 'Z') or
          (N[1] in Digit) or (N[i]='_') ) then //цифра або знак підкреслення
  begin
   Good:=false;
   break;
  end
  else Good:=true;
 end;
if Good then   FName:=N else FName:=' ';
end;

procedure TVariable.SetValue(V: integer);
begin
  if FValue <> V then   FValue:=V;
end;

constructor TVariable.Create(V: integer; N: string);
begin
  Value:=V;
  Name:=N;
end;

procedure TVariable.Square;
```

```
begin
  if sqr(Value)> 0  then Value:=sqr(Value)
    else ShowMessage ('Результат перевищує діапазон integer!');
end;
```

Зверніть увагу на те, що в методі запису SetName проводиться перевірка на правильність формування ідентифікатора.

Розглянемо детальніше метод Square. В ньому ми тричі звертаємося до властивості Value. Вперше, після **if** значення Value зводиться у квадрат. З контексту зрозуміло, що для виконання цієї дії необхідно прочитати значення пооля FValue. Властивість налаштована таким чином, що в цьому випадку буде викликаний метод для зчитування GetValue. Далі при виконанні правої частини оператора

Value:=sqr(Value)

буде знов викликаний метод зчитування GetValue, а при виконанні операції присвоєнні (операція присвоєння ініціює запис значення) буде викликаний метод запису SetValue. Отже, ззовні використання властивостей у програмі майже нічим не відрізняється від використання простих полів.

Як вже було сказано, методи, що забезпечують реалізацію властивостей, можуть виконувати перевірку коректності встановлювальних значень або певні перетворення при зчитування значень полів. Якщо ж потреби в спеціальних методах зчитування /запису полів немає, то можна використовувати замість імен методів імена полів. Наприклад, для зчитування значення змінної в класі TVariable властивість Value може бути описана таким чином :

```
property  Value : integer read  FValue   write FValue ;
```

Можуть бути властивості, які можна тільки зчитувати, або тільки записувати:

```
type
TMyOBject = class (TObject)
……
property SomeProperty : TSomeType  read  GetValue;
end;
```

Для цього прикладу оператор MyObject.SomeProperty := SomeData викличе помилку при помилку компіляції.

Для підтвердження важливості використання властивостей досить навести такий факт: в стандартних класах Delphi 100% полів недоступні для користувачів і замінені на відповідні їм властивості. Більш ніж доцільно дотримуватись цього правила і при розробці власних об’єктів.

Але слід пам’ятати одне обмеження на використання властивостей:

Властивості не можуть бути параметри процедури чи функції.

Виникає запитання, як запобігти випадковому зверненню користувача до якого-небудь поля. На допомогу приходять модифікатори доступу, що дозволяють “сховати” певні дані і методи від користувача.

Масив властивостей

В Delphi допускаються векторні властивості або **масив властивостей**, Опис таких властивостей має вигляд :

Property <ім’я_властивості> [Index : integer]: <тип_значення>
read <ім’я_методу_для_зчитування>
write <ім’я_методу_для_запису> ;

Доступ до векторних властивостей забезпечується тільки за допомогою методів, причому методи повинні обов’язково мати в якості першого формального параметра параметр-індекс типу integer:

Наприклад, в деякому класі описане поле:

```
Field:array [1..N] of real;
```

Для зчитування та запису цього поля необхідно задати два методи:

```
function GetItem  (Index : integer) : real;
procedure SetItem(Index : integer; NewPoint : real);
```

Тоді властивість для доступу до цього поля буде:

```
property Some_Array  : real  read  GetItem  write  SetItem;
```

Реалізація цих методів така:

```
function GetItem  (Index : integer) : real;
  begin
        Result:=Field[Index];
  end;
procedure SetItem (Index : integer; NewValue : real);
  begin
    if  Field[Index] <> NewValue then Field[Index]:=NewValue;
  end;
```

Як стає зрозумілим з наведеного програмного коду перший параметр цих методів Index використовується для індексації елементів поля-масиву Field.

Багатомірні масиви властивостей

Для доступу до багатомірного масивів можна описати відповідну властивість. Розглянемо опис на прикладі двовимірного масиву:

Property <ім’я_властивості> [Index1, Index2: integer]: <тип_значення>
read <ім’я_методу_для_зчитування>

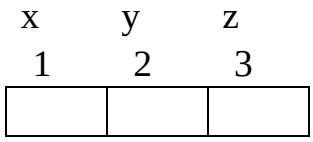
write <ім'я_методу_для_запису> ;

При описі методів доступу до їх списку формальних параметрів також додаються додаткові параметри-індекси.

Індексовані властивості

Інколи для доступу до поля-масиву зручніше описати не властивість-масив, а індексовану властивість, яка дозволяє звертатися до елементу поля-масива за певним ідентифікатором. Наприклад, в класі описане поле-масив, в якому зберігаються координати точки:

Coord : array [1..3] of real;



Методи доступу до елементів цього масиву будуть такі ж як у попередньому прикладі:

function GetItem (Index : integer) : real;

procedure SetItem(Index : integer; NewPoint : real);

Але для доступу до елементів цього масиву описуються три властивості:

property X : Integer **index** 1 **read** GetItem **write** SetItem;

property Y : Integer **index** 2 **read** GetItem **write** SetItem;

property Z : Integer **index** 3 **read** GetItem **write** SetItem;

Тепер при виконанні, наприклад, такого оператору:

X:=10;

буде визначений індекс цієї властивості (для X індекс 1) та буде здійснено присвоєння значення 10 елементу Coord [1].

Тема 6. Події та обробники подій

Основні обробники подій

Як було вже сказано, починаючи з класу TControl, елементи управління спроможні обробляти події. Давайте розглянемо їх за категоріями.

В класі TControl вводяться такі групи обробників[*]:

- події від миші:

OnClick, **OnDblClick** викликаються, коли відбулося клацання, або подвійне клацання мишею;

OnMouseDown, **OnMouseUp** викликаються, коли кнопка миші натиснута (MouseDown) або відпущена (MouseUp). Параметрами передаються:

[*] спільним параметром для всіх подій завжди є Sender: TObject.

– Button: TMouseButton, який приймає одне із значень mbLeft, mbRight, mbMiddle в залежності від того , яка кнопка миші була натиснута.
– Shift: TShiftState, який відстежує стан клавіатури і миші. Приймає значення ssShift, ssAlt, ssCtrl, якщо були натиснуті відповідні кнопки на клавіатурі, ssLeft, ssRight, ssMiddle додатково натиснуті кнопки на миші, або ssDouble – обидві кнопки на миші.
– X, Y: Integer – координати клацання.

OnMouseMove викликаються, коли відбувається просування покажчика миші над елементом. Параметрами є Shift: TShiftState і X, Y: Integer.

OnContextPopup викликаються, коли відбулося клацання правою кнопкою миші на елементі управління і дозволяє змінювати дії при виведенні спливаючого меню. Параметрами є MousePos: TPoint – координати миші, Handled: Boolean. По замовчанню параметр Handled встановлений False, якщо його встановити в True, це припинить обробку події.

- події на зміну розміру елементу управління:

OnResize викликається при завершенні операції зміни розміру елементу управління;

OnCanResize викликається перед операцією зміни розміру. Параметрами є NewWidth, NewHeight: Integer – нова ширина та висота елемента; Resize: Boolean, що дозволяє взагалі зміну або ні.

OnConstrainedResize викликається одразу після OnCanResize. Параметрами є MinWidth, MinHeight, MaxWidth, MaxHeight: Integer – мінімально та максимально припустимі розміри елементу.

- на події при перетаскуванні елемента управління:

OnDragOver виникає при перетаскуванні елемента управління над даним. Параметри:

– Sender : TObject – елемент, над яким відбувається перетаскування;
– Source : TObject – елемент, який перетаскують;
– X, Y: Integer – координати курсору миші
– State: TDragState визначає стан перетаскування: dsDragEnter при вході покажчика миші на елемент, dsDragLeave – при виході або якщо кнопка миші була відпущена, dsDragMove – пересування всередині.
– Accept: Boolean – повідомляє, чи дозволяє прийняти (Accept:=true), елемент, що перетаскується. Якщо Accept:=false, то на цей елемент неможна перетаскувати інші.
– **OnDragDrop** виникає при перетаскуванні елемента управління (тим, що викликав цій обробник) над даним, коли вже відпущена ліва кнопка миші. Параметрами є вище означені Sender Source : TObject ,X, Y: Integer.

OnStartDrag, OnEndDrag виникає при початку та завершенні перетаскування. Викликається елементом, що перетаскують.

Для того щоб перетаскування взагалі було можливе для всіх елементів треба встановити властивість **DragMode** := dmAutomatic. Властивість **DragCursor** визначають тип курсору при перетаскуванні. Обидві властивості встановлюються через Object Inspector або програмно.

- події на стикування елементів управління.

Найчастіше стикуються вікна. Наприклад, на рис.1 вікна Code Explore об'єднане з вікном редагування програмного коду. Механізм об'єднання закладений всередині елементів управління. Для того щоб їм скористатися необхідно для приймача (стиковочна станція – docking site) встановити властивість **DockSite** := True. Приймачем може бути будь-який елемент управління, що має цю властивість (Form, Panel, ToolBar і інші).

Елементи, що стикуються повинні мати властивості DragKind:= dkDock, DragMode:=dmAutomatic.

Нащадки TControl можуть бути тільки елементами, що стикуються. Стиковочними станціями можуть бути тільки нащадки TWinControl.

Обробники **OnStartDock** і **OnEndDock** викликаються для стикуємих елементів під час початку за закінчення стикування.

В класі TWinControl вводяться такі групи обробників:

- події на стикування елемента управління:

Обробники **OnDockDrop** і **OnDockOver** викликаються для стиковочних станцій під час стикування, а обробник **OnUnDock**при роз'єднанні.

OnGetSiteInfo (Sender: TObject; DockClient: TControl; var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean)

- події на передачу фокуса введення:

OnEnter, **OnExit** викликається, коли елемент отримує або втрачає фокус введення.

- події від клавіатури:

OnKeyPress викликаються, коли була натиснута клавіша. Параметр Key: Char – символ, що відповідає клавіші.

OnKeyDown, **OnKeyUp** викликаються в момент натискання або відпущення кнопки на клавіатурі. Параметри:

  – Key: Word – код клавіші, що була натиснута:
  – Shift: TShiftState – стан функціональних клавіш на клавіатурі (дивися обробники подій від миші).

- додаткові події від миші:

OnMouseWheel, OnMouseWheelDown, OnMouseWheelUp викликаються, коли обертається коліща миші або при клацанні коліщам як кнопкою.

Подія як властивість процедурного типу

Типова Windows-програма це програма, що керується подіями. Події виникають в результаті дій користувача, надходження сигналів від апаратури, операційної системи чи інших програм. Звістка про настання події – це повідомлення Windows, що одержується певною функцією вікна. Таких повідомлень є сотні, тому однією з задач програміста при розробці Windows-програми є необхідність призначити і описати реакцію на деякі з них.

Працювати з сотнями повідомлень Windows, навіть маючи довідник і електронну допомогу, досить важко. Однією з переваг Delphi є те, що програміст може уникнути безпосередньої роботи з ними. В Delphi більшість повідомлень Windows подані як стандартні події (біля двох десятків), використання яких не вимагає глибокого знання самих повідомлень Windows.

Наприклад, для інтерфейсних об'єктів визначені події OnClick, OnDblClick, OnMouseDown, OnMouseUp, OnMouseMove і т.п., з якими ви знайомилися у попередніх темах.

В мові Object Pascal подія (Event) - це властивість процедурного типу, призначена для створення реакції об'єкту на ті чи інші дії користувача або системи:

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

Події мають різні процедури, типи і параметри, залежно від походження і призначення. Як і для опису процедурного типу для опису події існує спеціальний синтаксис:

```
Type
 TMyEvent =
            procedure   (Sender:TObject; < список параметрів >)  of object;
```

Для всіх подій першим параметром має бути параметр Sender типу TObject. Під час виклику обробника він буде вказувати на об'єкт - джерело події. Присвоїти значення такій властивості – значить вказати об'єкту адресу методу, що буде викликатися в момент настання події. Такі методи називаються ***обробниками подій*** (Event Handlers).

Найпростіший тип подій не має більше ніяких параметрів крім Sender і називається TNotifyEvent:

```
type
TNotifyEvent = procedure ( Sender : TObject ) of Object;
```

Наприклад, подія клацання кнопкою миші :

property OnClick : TNotifyEvent;

Наприклад, подія, що виникає при натисканні клавіші на клавіатурі відповідає повідомленню Windows WM_CHAR, забезпечує передачу програмісту коду натиснутої клавіші.

type

TKeyPressEvent = **procedure** (Sender : TObject ; **var** Key : Char) **of Object**;

property OnKeyPress : TKeyPressEvent ;

Всі події в Delphi прийнято іменувати, починаючи з префікса “On”: OnCreate, OnMouseMove і т. д.

Події, які визначені для компонентів з VCL Delphi, знаходяться на сторінці Events Інспектора Об'єктів. Якщо двічі клацнути мишею в полі події, то в тексті програми генерується заготовка методу відповідного процедурного типу, причому ім'я методу буде складатися з імені об'єкту і імені події (але без префіксу “On”). Наприклад:

TForm1.Label1Click ;

У програміста є дві можливості :

- проігнорувати подію (тоді поведінка об'єкту визначається за замовчанням методом обробки повідомлень Windows) ;
- обробити подію, тобто назначити власний обробник подій, який дозволить змінити поведінку об'єкту, визначену за замовчанням методом.

Поняття делегування подій

Оскільки подія визначається як властивість об'єкту, то її значення можна змінювати не тільки на етапі розробки програми, а і в процесі її виконання.

Така можливість зміни обробника подій називається делегуванням. Можливо в будь-який момент часу зв’язати обробник події одного об'єкту з обробником події іншого, тим самим призначити (делегувати) її іншому. Наприклад :

Button1.OnClick : = Label1.OnClick ;

Принцип делегування є найбільш яскравим прикладом інкапсуляції методів у властивостях процедурного типу. Це дозволяє розширювати функціональність об’єкту не шляхом спадкування та перевизначення методів, а через делегування подій. Розглянемо приклад:

type

TMyEvent = **procedure** (Sender : TObject ; MyParameters) **of Object** ;

TMyObject = **class** (.....)

private

```
    FOnMyEvent : TMyEvent ;
protected
  procedure  MyEvent  ( Sender : TObject ; MyParameters ); dynamic;
published
   property  OnMyEvent:TMyEvent  read  FOnMyEvent  write  MyEvent ;
end ;
. . . . . .
procedure  TMyObject . MyEvent ( Sender : TObject ; MyParameters)
  begin
   if  Assigned  ( OnMyEvent )  then OnMyEvent (Sender, MyParameters)
 else
. . . . . .                          // обробка події  за замовчанням
end ;
```

Конструкція **if** Assigned (OnMyEvent) **then** є еквівалентною перевірці:

```
if  OnMyEvent  <> nil   then …
```

СПИСОК ЛІТЕРАТУРИ

1. Буч Г. , Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-ое изд. \ Пер. с англ.- М.: “Издательство Бином”, СПб: “Невский диалект”, 1998 г. – 560 с. ил.
2. Дарахвелидзе П.Г., Марков Е.П. Delphi – среда визуального программирования: СПб.:BHV – Санкт Петербург, 1996. – 352 с.
3. Кенту М. Delphi 5 для профессионалов. – СПб: Питер, 2001. – 944 с.
4. Конопка Рэй. Создание оригинальных компонент в Delphi. Пер. с англ. – К.: ДиаСофт, 1996-.512 с., ил.
5. Сван Том, Основы программирования Delphi для WINDOWS 95. Пер. с англ. – К.: Диалектика, 1996. – 480 с., ил.

Додаток 1. Клас TObject

В класі TObject введені основні методи, без яких неможливе використання жодного об’єкта. Дамо коротку характеристику кожного з методів базового класу TObject, поділивши їх за категоріями застосування.[Орлик]

Таблиця 1

Методи TObject, що стосуються екземпляру класу

Метод	Призначення
Constructor **Create;**	Утворює екземпляр класу (розподіляє пам’ять під об’єкт)
Class function **NewInstanse**: TObject;	Виділяє пам’ять розміром InstanceSize для екземпляра об’єкта (викликається з конструктора).
Class procedure **InitInstanse** (Instance: Pointer);	Заповнює 0 пам’ять, яка виділена методом NewInstanse (викликається з конструктора).
Procedure **Free**;	Викликає деструктор, якщо екземпляр об’єкта не пустий.
Destructor **Destroy**;	Деструктор непустого екземпляра об’єкта.
Procedure **FreeInstanse**;	Звільнює пам’ять розміром InstanceSize (викликається з деструктора).

Таблиця 2

Методи TObject, які повертають інформацію про клас

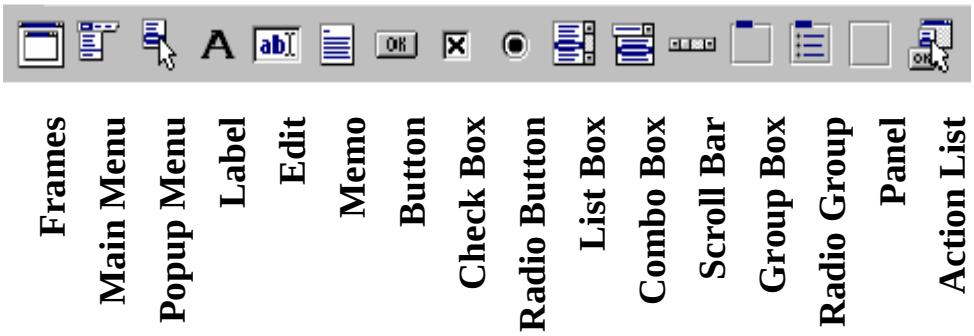
Метод	Призначення
Class function **InstanseSize**: word;	Повертає розмір даних об’єкта
Function **ClassType**: TClass;	Повертає об’єктне посилання, використовується для заміни as та is.
Class function **ClassName**: string;	Повертає ім’я класу
Class function **ClassParent**: TClass;	Повертає об’єктне посилання на предка
Class function **ClassInfo**: Pointer;	Повертає покажчик на структуру RTTI класу
Class function **InheritsFrom** (AClass: TClass): boolean;	Перевіряє чи спадкується поточний клас від заданого AClass.

Таблиця 3

Методи TObject, які є необхідними для доступу до методів класів та даних об’єкта

Метод	Призначення
Class function **MethodAddress** (const Name: string): Pointer;	Повертає адресу метода за його ім’ям Name.
Class function **MethodName** (Address: Pointer): string;	Повертає ім’я метода за його адресою.
Function **FieldAddress** (const Name: string): Pointer;	Повертає адресу поля за його ім’ям.
Procedure **DefaultHandler** (var Message); virtual;	Обробник подій за замовчанням.
Procedure **Dispatch** (var Message);	Викликає обробник подій відповідно до вмісту отриманого нетипізованого параметра Message. Зазвичай в Dispatch передається TMessage або одне з повідомлень TWMXxx .

Додаток 2. Бібліотека візуальних компонентів (палітри Standard, Additional, System, Dialogs)



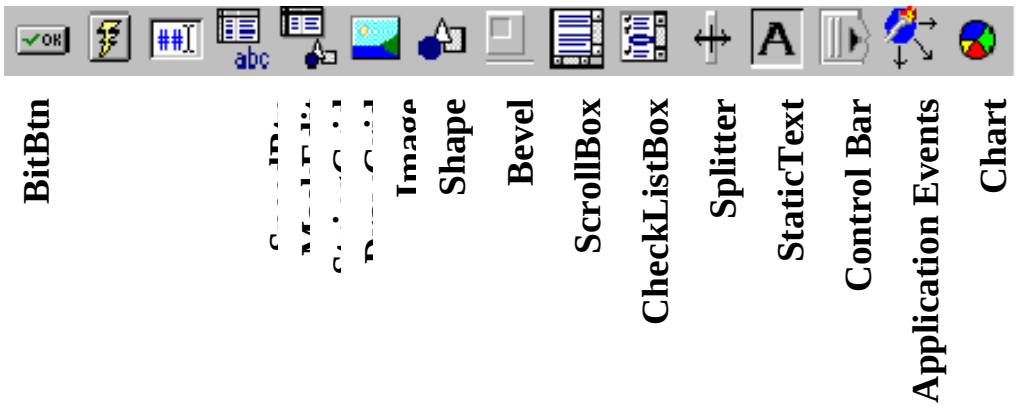
Компоненти категорії Standard

Frames	–	об’єкт-контейнер для інших компонентів (аналог форми), який може бути збереженим в VCL та використовуватися як один компонент в інших застосуваннях.
MainMenu	–	меню, яке випадає вниз з лінійки меню вікна і включає сукупність команд, з яких можна зробити вибір.
PopUpMenu	–	спливаюче меню, яке з’являється поверх об’єкту при кліку правою кнопкою миші.
Label	–	статичний текст, що виводиться на форму.
Edit	–	поле для введення текстової інформації та її редагування.
Memo	–	область введення/ виведення та редагування багаторядкового тексту.
Button	–	звичайна кнопка з надписом.
CheckBox	–	кнопка – прапорець, яка визначає, чи обрана подана опція.
RadioButton	–	взаємно виключаюча кнопка, вибір однієї з множини операцій.
ListBox	–	список елементів зі стрічкою прокрутки та курсором.
ComboBox	–	вікно редагування зі списком елементів, що розкривається.
ScrollBar	–	стрічки прокрутки.
GroupBox	–	прямокутний елемент управління з заголовком, який групує інші елементи управління.
RadioGroup	–	об’єкт – контейнер, який відображає групу залежних перемикачів.

Panel – панель, що використовується для створення панелі інструментів. Віконний елемент, може бути контейнером для інших. Використовується для створення панелей інструментів та панелей станів.

Action List – клас для створення компонентів, які підтримують списки дій, що використовуються компонентами та елементами управління, таких як пункти меню і кнопки.

Компоненти категорії Additional



BitBtn кнопка з надписом та піктограмою.

SpeedBtn кнопка з піктограмою, але без надпису. Використовується для створення панелей інструментів. Не можуть отримати фокус введення та потребують менше ресурсів.

MaskEdit використовується для введення та редагування даних. на відміну від Edit надає можливість роботи з маскованими даними такими як поштовий індекс, номер телефону, складені коди та інші.

StringGrid двовимірний масив рядків.

DrawGrid двовимірний масив рядків з можливістю формування графічних зображень у комірках.

Image поле виведення графічних зображення.

Shape призначений для виведення на форму найпростіших геометричних фігур: прямокутник, коло, еліпс та інші.

Bevel фаска. Може бути прямокутної форми або лінією. Використовується як контейнер для інших об’єктів. Не має обробників подій.

ScrollBox визначає прямокутну область зі стрічками прокрутки.

CheckListBox список елементів з прапорцями Check Box поряд, стрічкою прокрутки та курсором.

Splitter роздільник елементів – контейнерів, наприклад панелей. Використовується для зміни їх розміру під час роботи застосування.

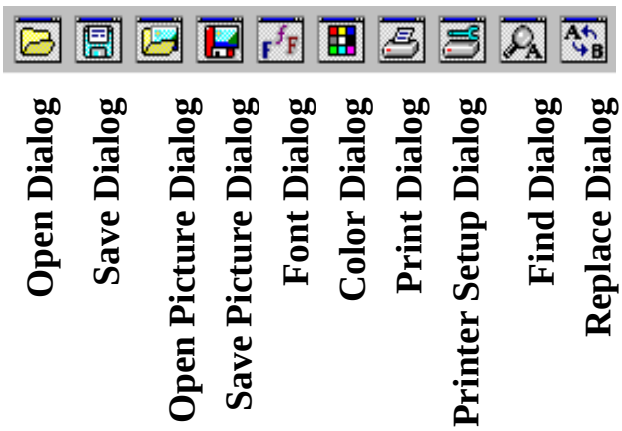
Static Text призначений для виведення на форму статичного тексту як і

	компонент Label, але може отримати фокус введення, тобто є віконним елементом.
Control Bar	контейнер для панелей інструментів. використовується для їх стикування (зв'язування) та управління.
Application Events	перехоплює і передає формі події об'єкту Application.
Chart	призначений для формування діаграм. Надає для формування діаграм редактор.

Категорія компонентів System

Timer	невізуальний компонент, який використовується для генерації під час роботи застосування сигналів через задані інтервали часу. З кожним сигналом викликається обробник подій OnTimer.
PaintBox	задає прямокутну область на формі для створення графічних зображень.
MediaPlayer	надає пульт управління для програвання аудіо- та відео-файлів.
OLEConteiner	контейнер OLE – об'єктів на формі застосування – клієнта.
DdeClientCovn	використовується для встановлення сеансу зв'язку клієнта (на боці клієнта) з сервером під час DDE - обміну
DdeClientItem	використовується на боці клієнта як контейнер для даних, що були отримані з серверу під час DDE - обміну.
DdeServerCovn	використовується для встановлення сеансу зв'язку клієнта з сервером під час DDE - обміну (на боці серверу).
DdeServerItem	використовується на боці серверу для передачі даних клієнту під час DDE - обміну.

Компоненти категорії Dialogs



OpenDialog	стандартне діалогове вікно Windows для відкриття файла.
SaveDialog	стандартне діалогове вікно Windows для збереження файла.
OpenPictureDialog	стандартне діалогове вікно Windows для відкриття файла з графічним зображенням.
SavePictureDialog	стандартне діалогове вікно Windows для збереження файла з графічним зображенням.
FontDialog	стандартне діалогове вікно Windows для вибору шрифту та його стилю.
ColorDialog	стандартне діалогове вікно Windows для вибору кольору .
PrintDialog	стандартне діалогове вікно Windows для завдання параметрів друку.
PrinterSetupDialog	стандартне діалогове вікно Windows для встановлення властивостей принтера, макету сторінки і та інших.
FindDialog	стандартне діалогове вікно для пошуку заданого тексту у файлі.
ReplaceDialog	стандартне діалогове вікно Windows для заміни у файлі певного тексту на іншій.