Chapter 3
THE DELPHI OBJECT-ORIENTED PROGRAMMING

Professor Zhaoyun Sun

3.1 Overview

The following topics are covered in this chapter:

- Classes and objects
- Encapsulation: private and public
- Using properties
- Constructors
- Objects and memory
- Inheritance
- Virtual methods and polymorphism
- Working with exceptions

3.2 Core Language Feature

- The Delphi language is an OOP extension of the classic Pascal language.
- The syntax of the Pascal language is known to be quite verbose and more readable than the C language.
- Its OOP Extension follows the same approach, delivering the same power of the recent breed of OOP language, from Java to C#.

3.3 Classes and Objects

- Delphi is based on OOP concepts, and in class types.

- The use of OOP is partially enforced by the visual development environment, because for every new form defined at design time, Delphi automatically defines a new class.

3.3 Classes and Objects

- In Delphi a class-type variable doesn't provide the storage for the object, but is only a **pointer** or reference to the object in memory.

- Before you use the object, you must allocate memory for it by creating a new instance or by assigning an existing instance to the variable:

```
var
Obj1, Obj2: TMyClass;
begin
// assign a newly created
//object
Obj1 := TMyClass.Create;
// assign to an existing
 //object
Obj2 := ExistingObject;
```

3.3 Classes and Objects

- A method is defined with the function or procedure keyword, depending on whether it has a return value.

3.3 Classes and Objects

- Inside the class definition, methods can only be declared; they must be then defined in the implementation portion of the same unit.

- In this case, you prefix each method name with the name of the class it belongs to, using dot notation:

3.3 Classes and Objects

```
procedure TDate.SetValue (m, d, y: Integer);
begin
Month := m; Day := d; Year := y;
end;
function TDate.LeapYear: Boolean;
begin
// call IsLeapYear in SysUtils.pas
Result := IsLeapYear (Year);
end;
```


3.3 Classes and Objects

- This is how you can use an object of the previously defined class:

```
var
   ADay: TDate;
begin

// create an object
 ADay := TDate.Create;
try
  // use the object
  ADay.SetValue (1, 1, 2000);
  if ADay.LeapYear then
     ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
finally
   // destroy the object
 ADay.Free;
 end;
```

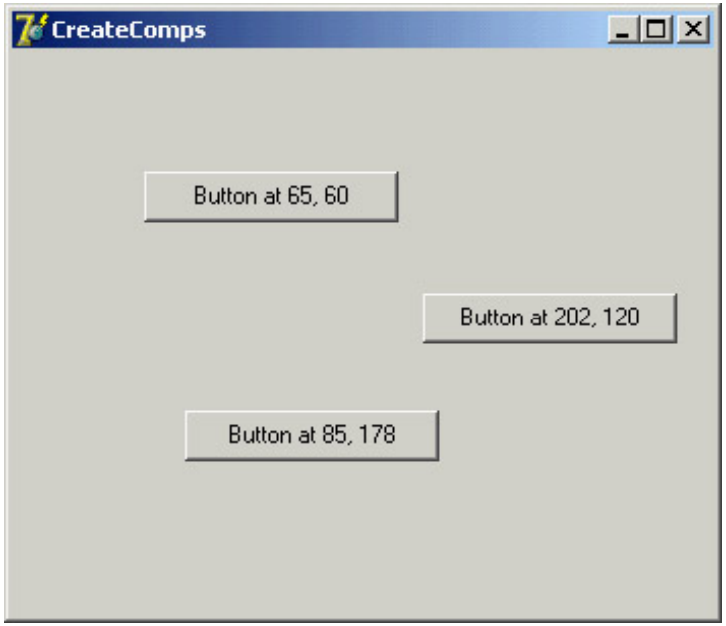
3.4 Creating Components Dynamically

- Delphi components aren't much different from other objects. This program has a form with no components and a handler for its OnMouseDown event. Here is the method's code:

```
procedure TForm1.FormMouseDown (Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
Var
  Btn: TButton;
begin
  Btn := TButton.Create (Self); Btn.Parent := Self;
  Btn.Left := X;
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button at %d, %d', [X, Y]);
end;
```


3.4 Creating Components Dynamically

- ***The effect of this code is to create buttons at mouse-click positions***



The Output Of The Createcomps Example, Which Creates Button Components At Run Time

3.5 Encapsulation

- A class can have any amount of data and any number of methods.

However, for a good object-oriented approach, data should be hidden, or encapsulated, inside the class using it.

3.5 Encapsulation

☐ Private, Protected, and Public

- For class-based encapsulation, the Delphi language has three access specifiers: private, protected, and public.

- A fourth, published, controls run-time type information (RTTI) and design-time information, but it gives the same programmatic accessibility as public. Here are the three classic access specifiers:

3.5 Encapsulation

☐ Private, Protected, and Public

- The private directive denotes fields and methods of a class that are not accessible outside the unit that declares the class.

- The protected directive is used to indicate methods and fields with limited visibility. Only the current class and its inherited classes can access protected elements. More precisely, only the class, subclasses, and any code in the same unit as the class can access protected members.

3.5 Encapsulation

The public directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

3.6 Constructors

- a constructor is a special method that you can apply to a class to allocate memory for an instance of that class.

- The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later.

3.6 Constructors

- All the data of the new instance is set to zero.

- If you want your instance data to start out with specific values, then you need to write a custom constructor to do that.

3.6 Constructors

Destructors and the Free Method

- In the same way that a class can have a custom constructor, it can have a custom destructor—a method declared with the destructor keyword and called Destroy.

3.6 Constructors

☐ Destructors and the Free Method

- Just as a constructor call allocates memory for the object, a destructor call frees the memory. Destructors are needed only for objects that acquire external resources in their constructors or during their lifetime.

3.7 Inheriting from Existing Types

- To inherit from an existing class in Delphi, you only need to indicate that class at the beginning of the declaration of the new class. For example, this is done each time you create a new form:

```
type
   TForm1 = class(TForm)
end;
```

3.7 Inheriting from Existing Types

- This definition indicates that the TForm1 class inherits all the methods, fields, properties, and events of the TForm class.

- You can call any public method of the TForm class for an object of the TForm1 type. TForm, in turn, inherits some of its methods from another class, and so on, up to the TObject base class.

3.7 Inheriting from Existing Types

☐ Inheritance and Type Compatibility

- Pascal is a strictly typed language. This means that cannot, for example, assign an integer value are type-compatible only if they are of the same data type, or(to be more precise) if their data type refers to single type definition.

3.8 Working with Exceptions

- Another key feature of Delphi is its support for exceptions. Exceptions make programs more robust by providing a standard way for notifying and handling errors and unexpected conditions.

- Exceptions make programs easier to write, read, and debug because they allow you to separate the error-handling code from your normal code, instead of intertwining the two.

3.8 Working with Exceptions

- Enforcing a logical split between code and error handling and branching to the error handler automatically makes the actual logic cleaner and clearer.

- You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

3.8 Working with Exceptions

- At run time, Delphi libraries raise exceptions when something goes wrong (in the run-time code, in a component, or in the operating system).

- From the point in the code at which it is raised, the exception is passed to its calling code, and so on.

3.8 Working with Exceptions

- Ultimately, if no part of your code handles the exception, the VCL handles it, by displaying a standard error message and then trying to continue the program by handling the next system message or user request.

3.8 Working with Exceptions

- The whole mechanism is based on four keywords:
  - *try* Delimits the beginning of a protected block of code.
  - except Delimits the end of a protected block of code and introduces the exception-handling statements.

3.8 Working with Exceptions

- **finally Specifies blocks of code that must always be executed, even when exceptions occur. This block is generally used to perform cleanup operations that should always be executed, such as closing files or database tables, freeing objects, and releasing memory and other resources acquired in the same program block.**

3.8 Working with Exceptions

- **raise Generates an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also raise exceptions in your own code when it discovers invalid or inconsistent data at run time.**

The raise keyword can also be used inside a handler to re-raise an exception; that is, to propagate it to the next handler.

3.8 Working with Exceptions

- ***Exception Classes*** **In the exception-handling statements, you caught the EDivByZero exception, which is defined by Delphi's RTL. Other such exceptions refer to run-time problems (such as a wrong dynamic cast), Windows resource problems (such as out-of-memory errors), or component errors (such as a wrong index).**

3.8 Working with Exceptions

- Programmers can also define their own exceptions; you can create a new inherited class of the default exception class or one of its inherited classes:

```
type
        EArrayFull = class (Exception);
```


3.8 Working with Exceptions

- When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
      raise EArrayFull.Create ('Array full');
```


3.8 Working with Exceptions

- This Create constructor (inherited from the Exception class) has a string parameter to describe the exception to the user.

- You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

3.8 Working with Exceptions

- The code presented in the previous excerpts is part of a sample program called **Exception1**.

- Some of the routines have been slightly modified, as in the following DivideTwicePlusOne function:

3.8 Working with Exceptions

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin try
// error if B equals 0
Result := A div B;
// do something else... skip if exception is raised
Result := Result div B; Result := Result + 1;
except
on EDivByZero do begin
Result := 0;
MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);
end;
on E: Exception do begin
Result := 0;
MessageDlg (E.Message, mtError, [mbOK], 0);
end;
end; // end except
end;
```