

## **Programming is a process of problem solving**

- **Problem solving techniques**
  - Analyze the problem
  - Outline the problem requirements
  - Design steps (algorithm) to solve the problem

### **Algorithm:**

- Step-by-step problem-solving process
- Solution achieved in finite amount of time

## **Problem Solving Process**

- **Step 1 - Analyze the problem**
  - Outline the problem and its requirements

### **Thoroughly understand the problem**

#### **Understand problem requirements**

Does program require user interaction?

Does program manipulate data?

What is the output?

#### **If the problem is complex, divide it into subproblems**

Analyze each subproblem as above

- Design steps (algorithm) to solve the problem

- **Step 2 - Implement the algorithm**

- Implement the algorithm in code
- Verify that the algorithm works

- **Step 3 - Maintenance**

- Use and modify the program if the problem domain changes

## **What is an algorithm?**

- **The idea behind the computer program**
- **Stays the same independent of**
  - Which kind of hardware it is running on
  - Which programming language it is written in
- **Solves a well-specified problem in a general way**

- **Is specified by**
  - Describing the set of instances (input) it must work on
  - Describing the desired properties of the output
- **Before a computer can perform a task, it must have an algorithm that tells it what to do.**
- **Informally: “An algorithm is a set of steps that define how a task is performed.”**
- **Formally: “An algorithm is an ordered set of unambiguous executable steps, defining a terminating process.”**
  - Ordered set of steps: structure!
  - Executable steps: doable!
  - Unambiguous steps: follow the directions!
  - Terminating: must have an end!

## Important Properties of Algorithms

- **Correct**
  - always returns the desired output for all legal instances of the problem.
- **Accurate**
  - It shouldn't be ambiguous (each step in the algorithm should exactly determine the action to be done).
- **Finite**
  - has to end up at a point (after finite amount of steps) at which the task is complete
- **Efficient**
  - Can be measured in terms of time, space
  - Time tends to be more important

## Representation of Algorithms




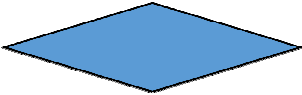
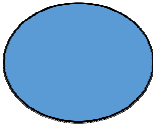

- **A single algorithm can be represented in many ways:**
  - Formulas:  $F = (9/5)C + 32$
  - Words: Multiply the Celsius by 9/5 and add 32.
  - Flow Charts.
  - Pseudo-code. (Pseudocode is like a programming language but its rules are less stringent.)
- **In each case, the algorithm stays the same; the implementation differs!**
  - A program is a representation of an algorithm designed for computer applications
  - Process: Activity of executing a program, or execute the algorithm represented by the program

## Flowchart

- It is another way to display the algorithm.
- It is composed of special geometric symbols connected by lines and contain the instructions.

You can follow the lines from one symbol to another, executing the instructions inside it

## Flowchart Symbols

- Start / End symbol 
- Input/Output symbol 
- Processing symbol 
- Condition & decision symbol 
- Continuation (connection symbol) 
- Links 

The algorithm would consist of at least the following tasks:

- 1- Input (Read the data)
- 2- Processing (Perform the computation)
- 3- Output (Display the results)

## Procedural vs object-oriented

### Procedural

- Early high-level languages
- Contain functions (or sub-routines) written and used inside the main program
- Cannot use external functions easily

Procedural : Pascal

### Object-oriented (OO)

- Later high-level languages
- Contain methods (or functions) and variables that can be written in main or external programs
- Can call external functions or variables easily

Object -oriented: Delphi, C++, Java

In old style programming, you had:

- data, which was completely passive
- functions, which could manipulate any data

**Object-oriented programming (OOP)** is a programming paradigm where an object contains both data and methods that manipulate that data

- Objects are not passive. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- The data that represent the object are organized into a set of **properties**.
- The values stored in an object's properties at any one time form the **state** of an object.

In object-oriented programming, the programs that manipulate the properties of an object are the object's **methods**. *Methods* are functions and procedures that apply only to objects of a particular class and its descendants.

- A **class** is a group of objects with the same properties and the same methods.
- Each copy of an object from a particular class is called an **instance** of the object.
- The act of creating a new instance of an object is called **instantiation**.

## OOP – main principles

### Inheritance

- Inheritance allows child classes inherit the characteristics of existing parent class
  - Attributes (fields and properties)
  - Operations (methods)
- Child class can extend the parent class
  - Add new fields and methods
  - Redefine methods (modify existing behavior)
- A class can implement an interface by providing implementation for all its methods

### Abstraction

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones - relevant to the given project (with an eye to future reuse in similar projects)
- Abstraction = managing complexity

### Encapsulation

- Encapsulation hides the implementation details
- Class announces some operations (methods) available for its clients – its public interface
- All data members (fields) of a class should be hidden
  - Accessed via properties (read-only and read-write)

- No interface members should be hidden
- Data fields are private
- Constructors and accessors are defined (getters and setters)

## Polymorphism

- Polymorphism = ability to take more than one form (objects have more than one type)
  - A class can be used through its parent interface
  - A child class may override some of the behaviors of the parent class
- Polymorphism allows abstract operations to be defined and used
  - Abstract operations are defined in the base class' interface and implemented in the child classes
  - For example, in a graphics application, we may have circle, square and triangle objects, each of which could have a Draw method. This would look the same to the caller, but each object would draw in a different way of course.

## Integrated development environment

- An **integrated development environment (IDE)** or **interactive development environment** is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger. Most modern IDEs offer Intelligent code completion feature.
- Integrated development environments are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. IDEs present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software.

## Computer graphics

- The interaction and understanding of computers and interpretation of data has been made easier because of computer graphics. Computer graphic development has had a significant impact on many types of media and have revolutionized animation, movies and the video game industry.
- Object-oriented concepts are particularly applicable to computer graphics – for example animations, simulations etc.

## Delphi

- [Borland Delphi](#) is a sophisticated Windows programming environment, suitable for beginners and professional programmers alike. Using Delphi you can easily create self-contained, user friendly, highly efficient Windows applications in a very short time - with a minimum of manual coding
- Delphi provides all the tools you need to develop, test and deploy Windows applications, including a large number of so-called reusable components.  
Borland Delphi, in it's latest version, provides a cross platform solution when used with Borland Kylix - Borland's RAD tool for the Linux platform.

- Delphi's roots lie in Borland's Turbo Pascal, introduced in the mid-1980s. Object Pascal, the object-oriented extensions to Pascal, is the underlying language of Delphi. The Visual Component Library, or VCL, is a hierarchy of Object Pascal objects that allow you to design applications. A better way of describing Delphi is an Object Pascal-based visual development environment.
- eleventh version, Delphi 2007 supports development in Object Pascal and C ++ for the Win32 platform; Object Pascal and C # for .NET.

## Object Pascal, C++

- **Object Pascal** refers to a branch of object-oriented derivatives of Pascal, mostly known as the primary programming language of Embarcadero Delphi.
- **C++** is a general purpose programming language. It has imperative, object-oriented and generic programming features, while also providing the facilities for low level memory manipulation.

## Program structure and syntax

- A complete, executable Delphi application consists of multiple unit modules, all tied together by a single source code module called a project file. In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use the file extension .dpr to designate the main program source module, while most other source code resides in unit files having the traditional .pas extension. To build a project, the compiler needs the project source file, and either a source file or a compiled unit file for each unit.

The source code file for an executable Delphi application contains:

- program heading
- a uses clause (optional)
- block of declarations and executable statements.

The first line contains the program heading.

The \$R compiler directive links the project's resource file into the program. Finally, the block of statements between the begin and end keywords are executed when the program runs. The project file, like all Delphi source files, ends with a period (not a semicolon).

### The Program Heading

The program heading specifies a name for the executable program. It consists of the reserved word program, followed by a valid identifier, followed by a semicolon. The identifier must match the project source file name.

The following example shows the project source file for a program called greeting.

```
program greeting;
{$APPTYPE CONSOLE}
var MyMessage : string;
begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

## The Program Uses Clause

The uses clause lists those units that are incorporated into the program. These units may in turn have uses clauses of their own.

The uses clause consists of the keyword `uses`, followed by a comma delimited list of units the project file directly depends on.

## Declarations

Here we declare all variables ( key word `var`) and constants (key word `const`) used in the program.

- A **variable** is an identifier whose value can change at run time. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold. Example:

- **`var`**

```
X, Y, Z: Double;  
I, J, K: Integer;  
Digit: 0..9;  
Okay: Boolean;
```

- **Right syntax for the variable names**

Rules	Wrong identifiers	Right identifiers
Must not start with a number	1name	name1
Dots are not allowed	name.2	name_2
Dashes are not allowed	-name-3	_name_3
Spaces are not allowed	Variable name	Variable_name
Accented characters are not allowed	déjà_vu	deja_vu

- The **constants** are similar to variables, except one point: they can't change their value during the execution.

## The Block

The block contains a simple or structured statement that is executed when the program runs. In most program files, the block consists of a compound statement bracketed between the reserved words `begin` and `end`, whose component statements are simply method calls to the project's Application object. Most projects have a global Application variable that holds an instance of `TApplication`, `TWebApplication`, or `TServiceApplication`. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

## Components of the language

- Compiler directives
- Directives
- Keywords – reserved words
- Types
- Variables and Constants
- Functions
- Procedures

## Types and declaration

- For those new to computer programming, data and code go hand in hand. You cannot write a program of any real value without lines of code, or without data. A Word Processor program has logic that takes what the user types and stores it in data. It also uses data to control how it stores and formats what the user types and clicks.
- How to define a variable to Delphi:

```
var           // This starts a section of variables
LineTotal : Integer; // This defines an Integer variable called LineTotal
First,Second : String; // This defines two variables to hold strings of text
```

Each variable starts with the name you choose, followed by a : and then the variable type. As with all Delphi statements, a ; terminates the line. As you can see, you can define multiple variables in one line if they are of the same type.

### Number types

Delphi provides many different data types for storing numbers. Your choice depends on the data you want to handle. Our Word Processor line count is an unsigned Integer, so we might choose **Word** which can hold values up to 65,535. Financial or mathematical calculations may require numbers with decimal places - floating point numbers.

```
var
// Integer data types :
Int1 : Byte;           // 0 to 255
Int2 : ShortInt;       // -127 to 127
Int3 : Word;           // 0 to 65,535
Int4 : SmallInt;       // -32,768 to 32,767
Int5 : LongWord;       // 0 to 4,294,967,295
Int6 : Cardinal;       // 0 to 4,294,967,295
Int7 : LongInt;        // -2,147,483,648 to 2,147,483,647
Int8 : Integer;        // -2,147,483,648 to 2,147,483,647
Int9 : Int64;          // -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807

// Decimal data types :
Dec1 : Single;         // 7 significant digits, exponent -38 to +38
Dec2 : Currency;       // 50+ significant digits, fixed 4 decimal places
```



```
Dec3 : Double;    // 15  significant digits, exponent  -308 to +308
Dec4 : Extended; // 19  significant digits, exponent  -4932 to +4932
```

## Text types

Like many other languages, Delphi allows you to store letters, words, and sentences in single variables. These can be used to display, to hold user details and so on. A letter is stored in a single character variable type, such as **Char**, and words and sentences stored in string types, such as **String**.

```
var
  Str1 : Char;           // Holds a single character, small alphabet
  Str2 : WideChar;       // Holds a single character, International alphabet
  Str3 : AnsiChar;       // Holds a single character, small alphabet
  Str4 : ShortString;    // Holds a string of up to 255 Char's
  Str5 : String;         // Holds strings of Char's of any size desired
  Str6 : AnsiString;     // Holds strings of AnsiChar's any size desired
  Str7 : WideString;    // Holds strings of WideChar's of any size desired
```

## Logical data types

These are used in conjunction with programming logic. They are very simple:

```
var
  Log1 : Boolean;        // Can be 'True' or 'False'
```

Boolean variables are a form of **enumerated** type. This means that they can hold one of a fixed number of values, designated by name. Here, the values can be **True** or **False**.

## Sets, enumerations and subtypes

Delphi excels in this area. Using sets and enumerations makes your code both easier to use and more reliable. They are used when categories of data are used. For example, you may have an enumeration of playing card suits. You literally enumerate the suit names. Before we can have an enumerated variable, we must define the enumeration values. This is done in a **type** section.

```
type
  TSuit = (Hearts, Diamonds, Clubs, Spades); // Defines the enumeration
var
  suit : TSuit;                             // An enumeration variable
```

Sets are often confused with enumerations. The difference is tricky to understand. An enumeration variable can have only one of the enumerated values. A set can have none, 1, some, or all of the set values. Here, the set values are not named - they are simply indexed slots in a numeric range. Here is an example to try to help you out. It will introduce a bit of code a bit early, but it is important to understand.

```
type
  TWeek = Set of 1..7; // Set comprising the days of the
week, by number
var
```

```

week : TWeek;
begin
    week := [1,2,3,4,5];      // Switch on the first 5 days of the week
end;

```

### Assigning to and from variables

Variables can be assigned from constant values, such as **23** and **'My Name'**, and also from other variables. The code below illustrates this assignment, and also introduces a further section of a Delphi program : the **const** (constants) section. This allows the programmer to give names to constant values. This is useful where the same constant is used throughout a program - a change where the constant is defined can have a global effect on the program.

Note that we use upper case letters to identify constants. This is just a convention, since Delphi is not case sensitive with names (it is with strings). Note also that we use **=** to define a constant value.

```

types
    TWeek = 1..7;                // Set comprising the days of the week, by
number
    TSuit = (Hearts, Diamonds, Clubs, Spades);    // Defines an enumeration

const
    FRED          = 'Fred';      // String constant
    YOUNG_AGE     = 23;          // Integer constant
    TALL : Single = 196.9;       // Decimal constant
    NO            = False;       // Boolean constant

var
    FirstName, SecondName : String;    // String variables
    Age                   : Byte;       // Integer variable
    Height                : Single;     // Decimal variable
    IsTall                : Boolean;    // Boolean variable
    OtherName             : String;     // String variable
    Week                  : TWeek;      // A set variable
    Suit                  : TSuit;      // An enumeration variable

begin    // Begin starts a block of code statements
    FirstName := FRED;              // Assign from predefined constant
    SecondName := 'Bloggs';         // Assign from a literal constant
    Age       := YOUNG_AGE;         // Assign from predefined constant
    Age       := 55;                // Assign from constant - overrides
YOUNG_AGE
    Height    := TALL - 5.5;         // Assign from a mix of constants
    IsTall    := NO;                // Assign from predefined constant
    OtherName := FirstName;         // Assign from another variable
    Week      := [1,2,3,4,5];       // Switch on the first 5 days of the week
    Suit      := Diamonds;          // Assign to an enumerated variable
end;    // End finishes a block of code statements

```

```

FirstName is now set to 'Fred'
SecondName is now set to 'Bloggs'
Age        is now set to 55
Height     is now set to 191.4
IsTall     is now set to False

```

```
OtherName  is now set to 'Fred'
Week       is now set to 1,2,3,4,5
Suit       is now set to Diamonds (Notice no quotes)
```

Note that the third constant, **TALL**, is defined as a Single type. This is called a **typed constant**. It allows you to force Delphi to use a type for the constant that suits your need. Otherwise, it will make the decision itself.

## Compound data types

The simple data types are like single elements. Delphi provides compound data types, comprising collections of simple data types.

These allow programmers to group together variables, and treat this group as a single variable. When we discuss programming logic, you will see how useful this can be.

### Arrays

Array collections are accessed by index. An array holds data in indexed 'slots'. Each slot holds one variable of data. You can visualise them as lists. For example:

```
var
  Suits : array[1..4] of String;    // A list of 4 playing card suit
names

begin
  Suits[1] := 'Hearts';    // Assigning to array index 1
  Suits[2] := 'Diamonds'; // Assigning to array index 2
  Suits[3] := 'Clubs';    // Assigning to array index 3
  Suits[4] := 'Spades';   // Assigning to array index 4
end;
```

The array defined above has indexes 1 to 4 (1..4). The two dots indicate a range. We have told Delphi that the array elements will be string variables. We could equally have defined integers or decimals.

### Records

Records are like arrays in that they hold collections of data. However, records can hold a mixture of data types. They are a very powerful and useful feature of Delphi, and one that distinguishes Delphi from many other languages.

Normally, you will define your own record structure. This definition is not itself a variable. It is called a data **type**. It is defined in a **type** data section. By convention, the record type starts with a **T** to indicate that it is a type not real data (types are like templates). Let us define a customer record:

```
type
  TCustomer Record
    firstName : string[20];
    lastName  : string[20];
    age       : byte;
```

```
end;
```

Note that the strings are suffixed with **[20]**. This tells Delphi to make a fixed space for them. Since strings can be a variable length, we must tell Delphi so that it can make a record of known size. Records of one type always take up the same memory space.

a record variable from this record type and assign to it:

```
var
    customer : TCustomer;           // Our customer variable
begin
    customer.firstName := 'Fred';    // Assigning to the customer record
    customer.lastName  := 'Bloggs';
    customer.age        := 55;
end;
```

```
customer.firstName is now set to 'Fred'
customer.lastName  is now set to 'Bloggs'
customer.age       is now set to 55
```

Notice how we do not use an index to refer to the record elements. Records are very friendly - we use the record element by its name, separated from the record name by a qualifying dot.

## Objects

Objects are collections of both data and logic. They are like programs, but also like data structures. They are the key part of the **Object oriented** nature of Delphi.

## Other data types

The remaining main object types in Delphi are a mixed bunch:

### Files

File variables represent computer disk files. You can read from and write to these files using file access routines.

### Pointers

They allow variables to be indirectly referenced. The **Pointer** type provides a general use pointer to any memory based variable. That is, one that is accessed by reference.

### Variants

They allow the normal Delphi rigid type handling to be avoided. Use with care!