7. Programming Paradigms

A programming paradigm is a set of principles that describe the structure of a language. There are various programming paradigms and for each paradigm there are various languages that follow the model (paradigm). For example the languages BASIC, C, FORTRAN, COBOL and Pascal all follow the Imperative Paradigm. Java, C++ and Smalltalk are three languages built on the Object-Oriented Paradigm.

7.1 Natural Languages and Formal Languages

Natural languages are the ones spoken between people like Maltese, English and Italian while formal languages are the ones constructed by academics for a specific purpose. Programming languages are all formal languages.

7.1.1. Natural Languages

'Natural language' refers to languages like English, French, German, etc. They are called natural because we do not (consciously) invent them; we naturally acquire them.

7.1.1.1. Syntax and Semantics

Excluding properties of sound (phonology), the properties of a natural language are generally understood to fall into three areas: syntax (or grammar), semantics, and pragmatics.

'Syntax' refers to the way in which words may be combined to form phrases and sentences. We may think of syntax as a set of rules telling us how to form sentences. However the syntax of natural languages is irregular. Apart from this, a fact can be expressed in more than one way e.g. "Bill kicked the ball" and "The ball was kicked by Bill".

'Semantics' is the study of the meaning of words, expressions, sentences etc. The semantics of a natural language is an exceptionally complex and confusing area.

Natural language is riddled with ambiguity. This is where a single word, phrase of sentence has multiple meanings independent of context. For example "Bob went to the bank" can mean (i) Bob visited a financial institution or (ii) Bob visited the side of a river.

Here is another example: "Mad dogs and men go out in the mid-day sun." Is 'mad' referring to 'dogs' or 'dogs and men'?

7.1.2. Formal Languages

A formal language is one we invent for some purpose or other. A programming language such as Java, for instance, is a formal language invented for the purpose of programming computers. A programming language cannot be ambiguous.

A formal language consists of:

- an alphabet of symbols
- rules (syntax) for their combination.

7.2 Context-Free and Non-Context Free Grammars

A context-free grammar is one where the syntax of each constituent is independent of the symbols occurring before and after it in a sentence.

Non-context-free grammars cannot identify the meaning of a term without considering the terms before and/or after the term. Natural languages have non-context-free grammars. For example consider the following two sentences.

- He was hit in his shoulder.
- He was hit in Rome.

The first meaning of 'hit' refers to a part of the body while the second refers to a place where an accident happened.

7.3 Imperative (Procedural) Paradigm

Imperative languages are also called 'procedural'. In these languages the programmer writes a sequence of statements (algorithm) and these express to the computer how a problem should be solved. Examples of procedural languages are BASIC, C/C++, FORTRAN, Java, COBOL and Pascal. Most computer languages are imperative.

The following dBASE examples show procedural and non-procedural ways to list a file. Procedural and non-procedural languages are also considered third and fourth-generation languages.

Procedural (3GL)	Non-Procedural (4GL)

USE FILEX
DO WHILE .NOT. EOF
  ? NAME, AMOUNTDUE
  SKIP
ENDDO
	USE FILEX
	LIST NAME, AMOUNTDUE

7.4 Declarative Paradigm

Declarative programming describes what the program should accomplish, rather than describing how to go about accomplishing it. This is in contrast with imperative programming, which requires an explicit algorithm that describes all the steps that a program should follow to solve a particular problem.

Declarative programming is often defined as any style of programming that is not imperative. Declarative paradigm is an umbrella term that includes a number of programming paradigms. Examples include:

- Functional paradigm
  - A program in such a language consists of a set of function definitions and an expression whose value is output as the program's result.
  - Haskell is an example of a purely functional language.
  - Lisp is not purely functional since it has some functional and some non-functional constructs.
- Logic programming languages
  - Such languages make use of mathematical logic.
  - The most famous language of this type, developed in 1972, is Prolog.
  - See example below.

Other kinds of paradigms that fall under the broader title of Declarative paradigms are Constraint Paradigm, Domain-Specific Paradigm and the Hybrid Paradigm.

Prolog

Prolog is a high-level programming language based on formal logic. It is based on defining and then solving logical formulas. Its programs consist of a list of facts and rules. Prolog is used widely for artificial intelligence applications, particularly expert systems.

facts	female (amy). female (sarah). male (anthony). male (bruce). male (joseph).

	parentof (amy,sarah). //i.e. amy is a parent of sarah parentof (amy,anthony). parentof (amy,bruce). parentof (joseph,sarah). parentof (joseph,anthony). parentof (joseph,bruce).
rules	siblingof (X,Y) :- parentof (Z,X), parentof (Z,Y), X Y. //i.e. X and Y must be different brotherof (X,Y) :- parentof (Z,X), male (X), parentof(Z,Y), X Y.
queries	? siblingof (sarah, bruce) Yes ?

7.5 Functional Paradigm

Functional programming is a style of programming that emphasizes the evaluation of expressions rather than the execution of commands. The expressions make use of functions (possibly nested).

The relations between functions are very simple: one function can call another, or the result of one function can be used as an argument of another function.

A program in functional language states "what" is to be done rather than "how" it is to be done. These languages are therefore declarative. These languages are at a higher level than imperative languages. This makes it easier to reason about the program.

However programs written in a functional language, although compact and elegant, run slowly and require a lot of memory. Examples of purely functional languages are Clean, FP, Haskell, Hope, Joy, LML, Miranda and SML. Many other languages such as Lisp have a subset which is purely functional but also contain non-functional constructs.

The following are two implementations of the Quicksort algorithm. One is written in Haskell and the other in C.

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

Quicksort in C

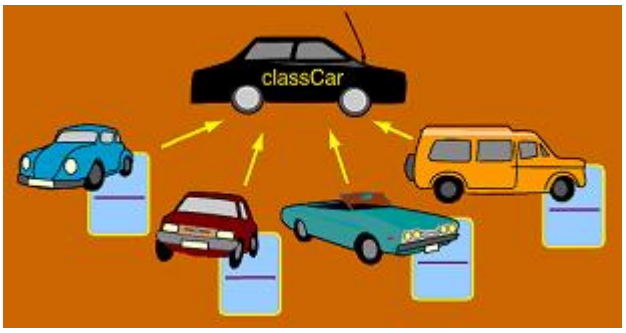
```
// To sort array a[] of size n: qsort(a,0,n-1)

void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

7.6 Object-Oriented Paradigm

This type of programming supports a model wherein the 'Data' and their associated 'Processing' (called 'Methods') are defined as self-contained entities called 'objects'. Object-oriented programming (OOP) languages, such as C++ and Java, provide a formal set of rules for creating and managing objects.

7.6.1. Objects

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.



Real-world objects share two characteristics: They all have state and behaviour. Dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behaviour (changing gear, changing pedal cadence, applying brakes). Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in 'fields' ('variables' in some programming languages) and exposes its behaviour through 'methods' ('functions' in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. An 'object' is also called an 'instance of a class'. In the program shown in the next section bike1 and bike2 are two objects. They are both instances of the class Bicycle.

7.6.2. Classes

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an Instance of the Class of Objects known as Bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeCadence(int newValue) {
             cadence = newValue;
        }
```

```
        void changeGear(int newValue) {
             gear = newValue;
        }

        void speedUp(int increment) {
             speed = speed + increment;
        }

        void applyBrakes(int decrement) {
             speed = speed - decrement;
        }

        void printStates() {
             System.out.println("cadence:"+cadence+"
                    speed:"+speed+" gear:"+gear);
        }
    }
```

The fields (variables) 'cadence', 'speed', and 'gear' represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world. (Cadence is the speed at which you turn the cranks, measured in revolutions per minute (rpm)).

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application (i.e. program); it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

```
class BicycleDemo {
     public static void main(String[] args) {

          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();
```

```
                bike2.changeCadence(50);
                bike2.speedUp(10);
                bike2.changeGear(2);
                bike2.changeCadence(40);
                bike2.speedUp(10);
                bike2.changeGear(3);
                bike2.printStates();
            }
        }
```

The output of this program (i.e. the class BicycleDemo, that makes use of the class Bicycle) prints the final values of pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

7.6.3. Methods

In other languages methods are called 'functions', 'subroutines' or 'procedures'. A method may take 0 or more parameters and may or may not produce a result. Java methods never return more than one result. All methods in Java are part of some class.

The following is an example of a method.

```
boolean evenOrNot (int x)
{
    if ( (x%2) == 0) return true;
        else return false;
}
```

A value passed to a method is called an 'argument'. The variable that receives the argument is called a 'parameter'.

So x is a parameter. Now if evenOrNot is prompted by an object called nu as in nu.evenOrNot(9) then the number 9 is an argument.

However in many cases the terms 'parameter' and 'argument' are considered as synonymous.

7.6.4. Encapsulation

Encapsulation is one of the three major features in object-oriented programming. The other two are Inheritance and Polymorphism.

Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called 'classes', and

one instance of a class is an 'object'. For example, in a payroll system, a class could be Manager, and Pat and Jan could be two instances (two objects) of the Manager class. Encapsulation ensures good code modularity.

Example: The following diagram depicts a class called 'Shape'.

Class 'Shape'

DATA Variable 'colour' of type 'string'
METHODS moveDown(...) moveUp(...) moveLeft(...) moveRight(...) makeInvisible() makeVisible()

From the class 'Shape' one can create as many objects as one requires in one's program. Let us say that three objects are created (i.e. three instances of the class 'Shape'). Let these objects be called 'Shp1', 'Shp2' and 'Shp3'. These are depicted in the following diagram.

Object 'Shp1'	Object 'Shp2'	Object 'Shp3'
DATA colour = "green"	DATA colour = "yellow"	DATA colour = "violet"
METHODS (as in class)	METHODS (as in class)	METHODS (as in class)

7.6.4.1. Information Hiding

Encapsulation causes information hiding. When creating a class for use by other programmers the way it is implemented is hidden to these programmers. The creator of the class can change its implementation without causing any disturbance to the programmers that use the class.

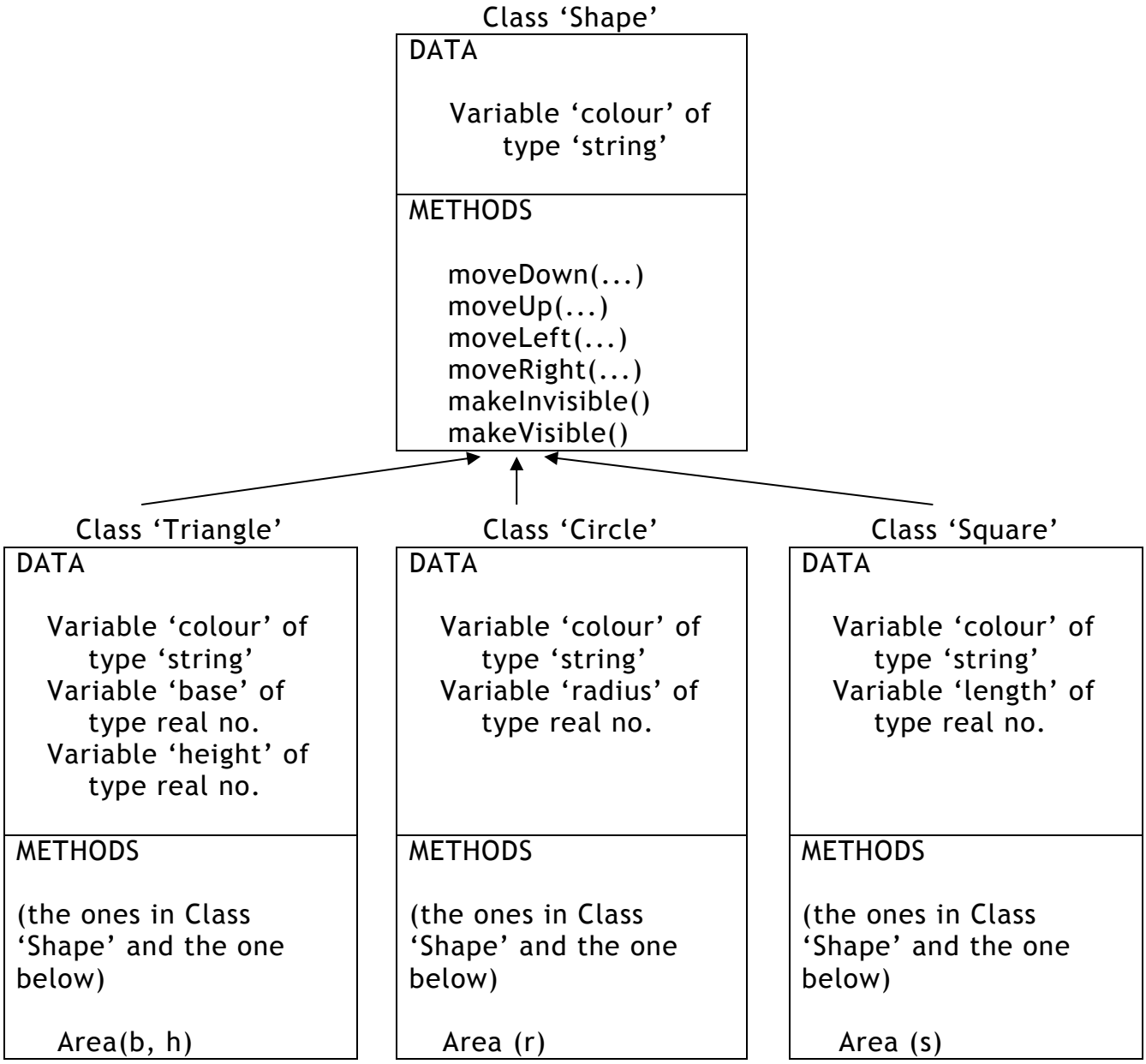
7.6.5. Inheritance

"Someday, son, none of all this will be yours."

Classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is

inherited.

Example: Let us consider the class 'Shape' and from it we create three other classes called 'Triangle', 'Circle' and 'Square'. The following diagram depicts this procedure.



The class 'Shape' in this case is called a 'superclass' while the classes 'Triangle', 'Circle' and 'Square' are all 'subclasses' of the class 'Shape'.

7.6.6. Polymorphism

Object-oriented programming allows procedures about objects to be created whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement would be

written for "cursor," and polymorphism allows that cursor to take on whatever shape is required at runtime. It also allows new shapes to be easily integrated.

In the above example one can apply the method Area to any object that is an instance of one of the classes Triangle, Circle and Square.

7.6.7. Abstraction

Abstraction is a term that refers to a simplified representation that contains only those features that are relevant for a particular task.

7.6.8. Message Passing

Message passing is a way to express how objects interact in an object-oriented system. Objects send each other messages to request services, to request information or to supply information. Since objects interact only through the messages they exchange, their internal details can remain hidden from each other.

7.6.9. Benefits of Object-Oriented Programming

The concepts and rules used in object-oriented programming provide these important benefits:

1. The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Inheritance forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
2. The definition of a class is re-useable not only by the program for which it is initially created but also by other object-oriented programs.
3. The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.
4. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
5. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
6. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
7. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous

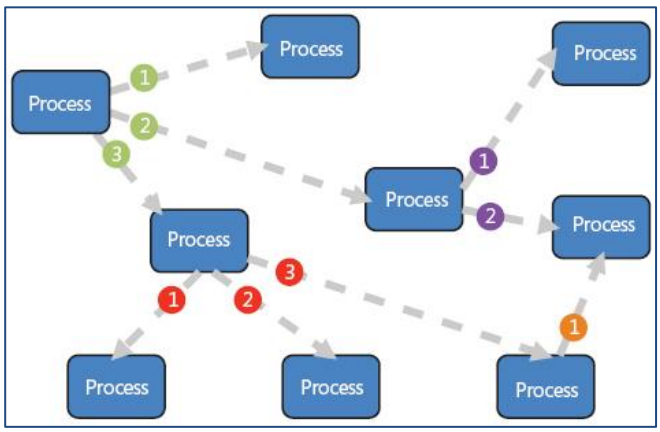
to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

7.6.10. The Advantages of Object-Oriented Paradigm over the Traditional Imperative Paradigm

a) The real world can be better (and easier) modelled by using the OO paradigm than the imperative paradigm especially if the real world situation to be modelled is complex.
b) Encapsulation, inheritance and polymorphism are features that are not present in the imperative paradigm.
c) UML (Unified Modelling Language) diagrams can represent the model of the real-world very clearly before coding can start. There are various types of UML diagram-types.

7.7 Event-Driven Paradigm

These kinds of languages build applications where objects (like an icon) wait for an event (like mouse click, mouse double-click, mouse right-click, pressing 'enter', etc) to happen. Once that event happens, a program called an 'event-handler' is triggered and starts being executed. Most Visual Basic application programs are event-driven. The best example of an application built by an event-driven language is the GUI mouse-and-windows-driven user interface.



These kinds of languages are imperative. The event-handlers are themselves formed by a sequence of commands. A GUI event-driven application can however be written also by means of an imperative language like Java or Pascal.

7.8 Domain Relevance of Each Paradigm

Natural languages have always been a challenge in the history of computer science particularly in the field of artificial intelligence. Programs were often requested that could translate from one natural language to another. Today in quite a number of applications use is made of natural languages e.g. one can write in the help text-window of a word processor "I want to know how to build a macro". The word processor's analyser is capable of focusing on the words 'macro' and 'build' while rejecting the others. Then it presents help options involving both or one of the words. For a user it would have the same result if in the text-window he/she writes "build macro" but writing a whole sentence can make one feel one is communicating with an intelligent being. The same can be said about search engines.

Formal languages have a wide domain of applications in computer programs. They are however not used in computers alone. They are also used in areas such as mathematics and logic.

All computer languages are formal languages. Also, object-oriented languages are also imperative languages, yet not all imperative languages are object-oriented. These languages are general purpose and can be used to program any kind of problem. However for particular areas such as mathematics specialised languages like FORTRAN are better suited. The same can be said for databases where specialised languages exist that focus on database requirements.

Object-oriented languages are chronologically an improvement on traditional imperative languages for example the language C (developed in 1972 by Dennis Ritchie) was 'upgraded' to C++ by adding to it object-oriented facilities. It was developed by Bjarne Stroustrup in 1979 as an enhancement to the C and originally named "C with Classes". It was renamed C++ in 1983.

Complex programs are more manageable when designed and coded with an object-oriented language. Today these languages have a vast library of ready-made classes including GUI objects.

Hardware is moving to multi-core, multi-processor machines. In addition to multi-core, high processing and complex algorithm environments are moving toward leveraging graphic processing units or GPU for highly parallel processing. The sum of all of this from a developer's standpoint is concurrency.

Most of our programming languages make concurrency hard. Functional languages already aid in the development of constructs that ease concurrent development, through the benefit of functions which do not share memory. Many of the functional languages implement a pattern of concurrent development. Another value add of functional languages is conciseness (see the quicksort examples).

An important observation is that functional languages are not a replacement for procedural or object-oriented programming language. Many of the newer functional languages are multi-paradigm languages that run on virtual machines and bridge other object-oriented and imperative languages.

Event-driven programming languages are extremely important for GUIs. This makes the human-computer interface more user-friendly.