M301: Software Systems & their Development

Block 1: Introduction to Java

Unit 1: Introduction and the IDE

Section 1: Introduction to the course

Aims of the course:

- Enable you to read programs written in Java and to amend or extend existing software systems in response to new requirements, particularly in the area of networked systems;
- describe the major features of modern concurrent (distributed) systems;
- illustrate the use of the Unified Modelling Language in the analysis of new requirements in relation to existing solutions;
- analyse existing systems to identify their major architectural principles and software mechanisms, and to describe how they work both in isolation and together;
- introduce the development of software systems using an object-oriented approach with associated management practices.

Course structure:

Block 1 of the course concentrates on building software components. It shows you the fundamentals of creating objects and building systems from them. We shall introduce you to the basics of Java as a means of illustrating the concepts involved.

The existence of communicating distributed software implies that there are separate sequential programs executing simultaneously (we use the term *concurrent processes* to describe such a situation). In **Blocks 2 and 3** you will study the ramifications of concurrency in software systems. You can think of these blocks as providing the theoretical basis of distributed software.

Blocks 4 and 5 turn to the issues that arise in the process of developing software. These two blocks may be considered as providing the practice of software development.

Block 6 brings together the theory and practice, and provides you with the opportunity of putting what you have learned into practice via a small project based on a case study that you will have followed throughout the course.

Read Section 1.1 from Block 1

Block 1: Introduction to Java

Unit 1: Introduction and the IDE

Section 2: Introduction to the Integrated Development Environment (IDE)

Objectives:

- Set up a directory suitable for your practical work in Java;
- create a new *project* using the IDE;
- create a Java *source file* containing a small class;
- compile and execute a Java application;
- use the help facility in the IDE.

IMPORTANT: Read Section 1.2 from Block 1 and do the practical activities

Block 1: Introduction to Java

Unit 1: Introduction and the IDE

Section 3: First Program in Java

Book reference:

Chapter 4 of *Understanding Object-Oriented with JAVA* by Budd.

Objectives:

- understand some of the basic Java constructs needed to create a simple executable Java program, which includes the structure of a program, a class, a method, a data field, and the use of modifiers;
- describe the role of the main method when a Java program is executed;
- describe the model of execution of a Java program including the roles of variables and method invocation (or message passing);
- understand the need for, and how to write, declarations;
- understand the difference between types, abstract data types and classes.

Objects, classes and calling methods in Java

Consider the following simple example of a Java class named BankAccount:

```
Class BankAccount {

        private int accountBalance = 0;

        public void deposit (int amount) {
                accountBalance = accountBalance + amount;
        }

        public void withdrawal (int amount) {
                accountBalance = accountBalance - amount;
        }
  }
```

This class contains three *members*:

- A **data field** named accountBalance which is initialized to zero.
- Two **methods** named deposit and withdrawal.

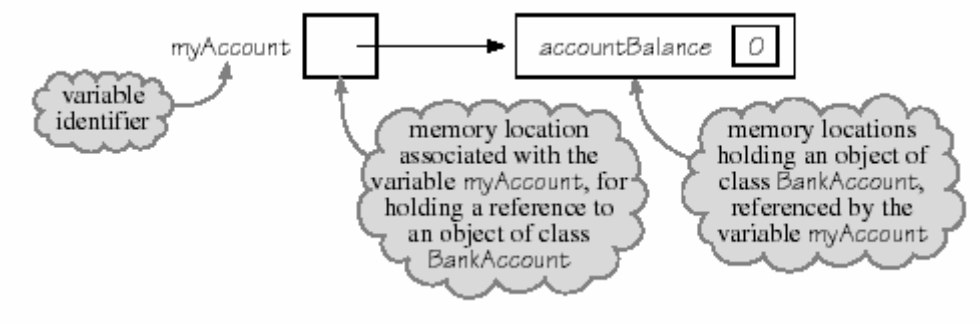
Sending messages to the object

In another class you could create a BankAccount object and call/invoke methods on it, for example:

```
BankAccount myAccount = new BankAccount();
```

Note: In Java = acts in the same way as **:=** in Smalltalk.

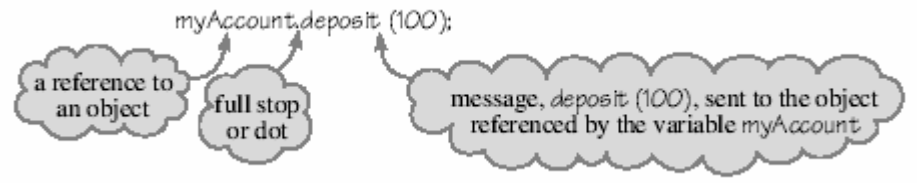
The result of the above Java statement is shown below.



Since every instance of the class BankAccount will have such a data field, it is known as an *instance variable*.

Once the object referenced by myAccount has been created, messages can be sent to it.

For example, to add 100 to the accountBalance; of the object referred to by myAccount, you would write:



The full stop between the variable identifier and the method name is known as the **dot notation** which is used to denote sending a message to an object. When a message is sent to an object, the corresponding method is executed (or called).

Note: In Java, we usually say a method is called rather that a message is sent to an object.

Exercise:

After executing the following statements:

```
myAccount.deposit(100);
myAccount.withdraw(30);
myAccount.withdraw(26);
```

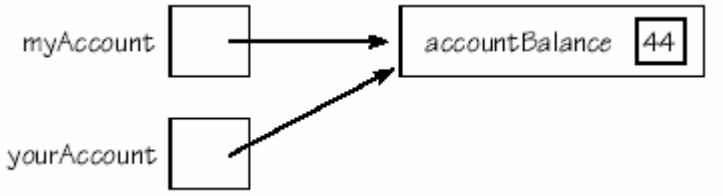
What is the value of the balance in myAccount?

The three lines in the above exercise are called *statements*. Each statement terminated with a semicolon.

myAccount is known as a variable because the object to which it refers can change as the program is executed. For example, suppose that there were two BankAccount objects, referred to by the variables myAccount and yourAccount. The following statement, an example of an assignment statement,

```
yourAccount = myAccount;
```

would result in the variable yourAccount referring to the same object as myAccount as illustrated in the following figure.



This implies that the object would be updated in exactly the same way by either

```
myAccount.deposit(50);
```

or

```
yourAccount.deposit(50);
```

Having two (or more) variables reference the same object is known as *aliasing* and must be used with care.

Primitive data types and classes in Java

Primitive data types are:

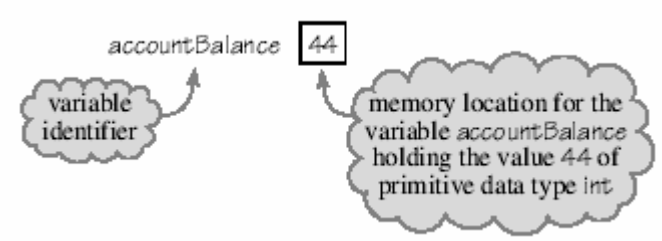
- int (for integer)
- float (for real numbers)
- double (double precision real)

- `boolean` (either true or false)
- `char` (short for character)

Classes are program constructs containing methods and data fields whose definitions can be inspected by the programmer.

Primitive data types vs. Classes

In the case of objects, variables are used to refer to objects as illustrated in the previous figure. Java does not use the reference mechanism for variables of primitive data type. Instead, the values are stored in the variable itself. This is illustrated in the following figure.



Now to demonstrate the difference further, suppose `i` = 1 and `j` = 4, then the assignment statement
`i = j;`
Results in `i` = `j` = 4. However, both variable (`i` and `j`) are independent. So, if we apply the following statement
`i = 2;`
Then `j` will remain 4.

Variables and declarations

Java is a **strongly typed language** in the sense that every variable must be *declared* as having a particular type. That is, every variable must appear in a *declaration* in which the class of the objects to which it may refer or the primitive data type of the values it contains must be stated explicitly. In a strongly typed language, a given variable is allowed only to refer to an object or value of its declared type. Examples of Java declarations are:

```
private int accountBalance = 0;
int amount;
BankAccount myAccount = new BankAccount();
String [] arges;
```

Conceptual vs. implementation view of a solution

Thinking about how to solve a problem that involves understanding the problem (or application) domain which has nothing to do with programming languages, a process that leads to what we describe as the **conceptual view** of the solution.

Implementing the solution to a problem in a particular programming language that requires a knowledge of the syntax and semantics of that language, a process that leads to what we describe as the **implementational view** of the solution.

Types, abstract data types and classes

Type is a word given to a named set of items having some property in common.

An **abstract data type (ADT)** is a set of items defined by the collection of operations that can be carried out on them. The elements of such a set are usually referred to as **values**.

Example of an ADT is a set of bank accounts. All the instances of this set will respond to the same operations (e.g. `getBalance`, `setBalance`, `withdrawAmount`, `transferAmount`).

A **class** is a collection of objects and the objects are defined by a set of methods that correspond to the operations of the ADT. These methods will, in general, be implemented in terms of data fields and other methods contained within the class. The methods that implement the operations will be *public* (and therefore available for use in other classes) whereas the other methods and data fields will be *private* (available for use only within the class).

Therefore, a type can stands to either the name of a primitive data type or the name of a class.

Methods

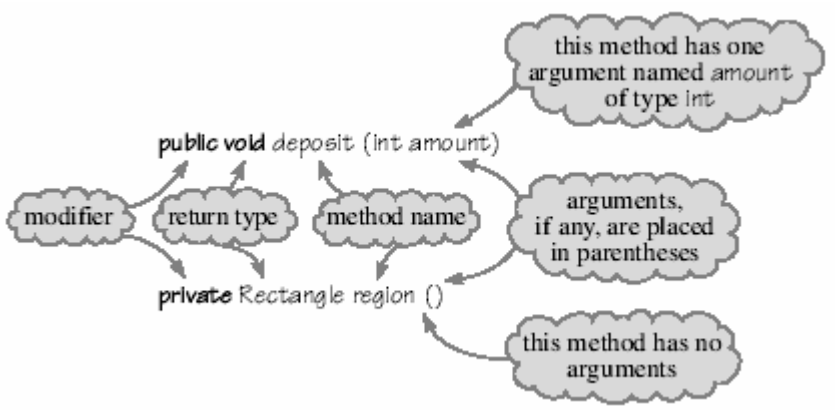
A method is a small part of a program that is supplied with some objects (or values of primitive data type) called **arguments**, and whose purpose is to perform a computation, possibly producing an object (or a value of a primitive data type) as its result.

A method in Java is composed of two parts: a *heading* and a *body*.

The heading consists of four elements:

- its name;
- the name and types of its arguments;
- the type of the object or primitive data type it returns as output (if any);
- the modifier(s) (access and/or lifetime).

This is illustrated in the following figure.



The keyword void is used to state that a method does not return a result.
In Java, a method always has parentheses even if there are no arguments (this is a good way to distinguish a data field from a method).

Examples of methods are:

```
myAccount.deposit(100);
myBall.region();
```

In the message deposit(100), the value 100 is known as an **actual argument** which, when the method deposit is invoked, is assigned to the **formal argument** , amount, appearing in the method heading. During the execution of the body of deposit, the formal argument amount acts as a local variable.

A method **signature** defines everything you need to know about a method to call it, i.e. its modifiers, return type, name, and parameters. In Java, methods whose signatures differ only in their return types are not allowed.

Read Budd, Chapter 4 then solve the SAQs of section 1.3 of Block 1

Summary of Chapter 4

Consider the following Java program

```
import java.lang.*;

public class FirstProgram {

        public static void main (String [] args) {
                System.out.println("Hello World!");
        }
}
```

The Java universe is a community populated mainly by *objects*. Objects are all instances of *classes*. Hence, the overall structure of a java program is simply a series of class descriptions.

A class consists of a class *header* and a class *body*. The header provides the name of the class. The class body begins with a curly bracket and terminates with a matching curly bracket.

The class body consists of a series of *members*. A member can be either a *data field* or a *method*.

A data field characterizes the internal data being held by an object.

A method defines the behaviors an instance of the class can perform.

A method consists of a header and a body. It follows the following form:

modifiers return-type method-name (arguments) {
sequence-of-statements
}

Method bodies must always be properly nested within a class description.

The import statement makes a portion (or *package*) of the Java library (knows as the Java Application Programming Interface, or API) visible to the class description that follows it.

The statement

```
System.out.println("Hello World!");
```

Print a single line of output on a standard output (*output console*). In the above statement, System is a class, out is a data field and println is a method.

Strings

Consider the following program

```
import java.lang.*;

public class SecondProgram {

	public static void main (String [] args) {
		if (args.length > 0)
			System.out.println("Hello " + args[0]);
		else
			System.out.println("Hello World!");
	}
}
```

Here, the data member length is being used to determine the number of values held by the array named args.

If the user entered a command line argument, such as the string "Khalid", the output will be "Hello Khalid".

The index values for an array in Java begin with zero and extend to the value one smaller than the number of elements in the array.

The operator + is used for string concatenation when at least one argument is a String.

Access Modifiers

The three access modifier keywords and their purposes are:

public: the data field, method or class to which it is applied is visible to, i.e. available to be used by, all objects of all other classes in addition to the class in which it is defined;

private: the data field, or method to which it is applied is not visible to objects of classes outside the class in which it is defined, including subclasses;

protected: the data field, or method to which it is applied is visible only to those objects outside the class in which it is defined that are heirs, i.e. subclasses, of that class. It makes sense to allow subclasses to access the private members of its parent class even though client classes cannot do so. A protected member is also visible to all classes in its package (if it has been defined as part of one).

Lifetime Modifiers

When applied to a data field, static signifies that the data field is shared by all the objects of the class. A static variable is also known as a class variable to distinguish it from a non-static variable known as an instance variable because each object has its own copy of the instance variable.

Note that a static member, be it a method or a data field, exists even if no objects of its class have been created. Consequently, like the method main, which is always declared static, all static members are available for use when a program starts executing, and before any objects have been created.

Exercise:

Why do you think that the main method should always be public and static? (read Budd, pp. 62).