PROGRAM PRACTICALLY WITH

JAVA

(Eclipse IDE Version)

Build your programming muscle series

GERRY BYRNE

Program Practically

Java

Module 01

Introduction – what is a program?

Programming languages
A recipe
Input, output and process
Packages and classes
Variables
Console, form and web

Gerry Byrne

Program Practically – Introduction

Computer Program

We will be using Java to write computer programs, just like many programmers in companies around the world who use Java to write programs in the commercial environment. So, a very good starting point before we write programs (code) is to fully understand what a computer program is. We can think of a computer program as:

- a sequence of data instructions created by a programmer
- instructions that tell the computer what operations it should execute
- instructions that tell the computer how it should execute an operation
- instructions written in a special programming language e.g. Java, C#, C++, COBOL

Besides Java there are a large number of programming languages available to developers when creating their application. Each programming language will have particular advantages and disadvantages when compared to other programming languages, but they will all be useful for writing computer applications. It is important to understand that some programming languages are:

- more powerful than others e.g. Java
- better for developing applications requiring fast processing e.g. C
- better for developing web based software applications e.g. JavaScript
- better for developing computer games e.g. C++
- better for data analytics e.g. Python
- better for scripting e.g. Perl

The points above should help us understand that there are many programming languages available for software developers, but they all have concepts that can be applied across the majority of the languages. So, by the time we finish reading this book, entering and running all the example code and doing all the exercises, we will be in a strong position to recognise and apply constructs in the C# programming language or the C++ language and indeed other programming languages, as well as our main focus, the Java language.

Programming Languages

The language we will be using is Java but there are a large number of programming languages available for developers to use and create their application. It is certainly great to have a choice of programming languages but at times this makes it difficult to choose the correct one when writing a software application. In the list below, we can see some facts about programming languages:

- there are many different programming languages to choose from
- each language has its own set of very strict language rules
- Java is one such programming language
- other languages include C#, C++, Visual Basic, Python, JavaScript, Cobol, Swift Objective C, Ruby, Go
- programming languages such as Java, C#, C++ and Visual Basic are **high-level languages** since they have a high correlation with a spoken and written language
- assembly language is called a **low-level language** as it has a low correlation to a spoken and written language and is more like the language the computer can understand
- every computer program will need to be 'translated' into 'machine code' that the computer can understand e.g. byte code, object code and binary code
- the process of 'translation' is carried out by **compilers**, **interpreters** or **assemblers**.

A computer program – can we compare it to a recipe used for baking or cooking?

Let us think about a recipe that we might use in our kitchen to create the end product of **fifteens**. The information we need might be written in a book or on a website like this:

Ingredients

- 15 digestive biscuits
- 15 marshmallows
- 15 glacé cherries, cut into halves or smaller
- About 150ml of condensed milk
- 100g desiccated coconut

Instructions

- add 15 digestive biscuits to a bag and 'smash' the biscuits with a rolling pin until they are fine crumbs
- place the crumbs in a mixing bowl
- slice the 15 marshmallows into pieces, we decide how big the marshmallows should be
- slice the 15 cherries in half or smaller, we decide how big the cherries should be
- add the cherries and marshmallows to the digestive biscuit crumbs in the mixing bowl
- stir the mixture until the cherries and marshmallows are spread evenly around the biscuit crumbs
- pour the 150ml of condensed milk on top of the biscuit, glacé cherries and marshmallows mix
- mix the contents in the bowl and add more condensed milk if required, so that the mixture is not dry
- cut a large piece of tinfoil
- spread half of the coconut onto the tinfoil
- scoop the wet biscuit, glacé cherry and marshmallow mix onto the tinfoil
- add the other half of the coconut to the mixture
- roll the tinfoil over the mixture to create a sausage shape
- move the rolled mixture to the fridge
- leave in the fridge for 3 or 4 hours
- remove the roll from the fridge and cut it into 15 slices

As we can see, the recipe contains:

- a list of instructions (directions) written in a language (in this case it is English):
  - likewise a computer program contains a list of statements (directions) written in a programming language such as Java
- a list of ingredients. The ingredients are of various types e.g. biscuits, marshmallows, glacé cherries, condensed milk, desiccated coconut:
  - likewise a computer program contains a list of variables (ingredients). The variables will be of various types e.g. numbers, text

The following two code examples show the structure of code for Java and Python. Even at this early stage, by looking at the code examples, we should see some similarities between the two different programming languages, Java and Python. By the end of this course we will become more familiar with programming and other programming languages will be less 'daunting' to look at and to program with.

Example code

This example shows Java code which will ask the user to input two values and then totals the values. The program is like our recipe, it is a set of instructions.

```
int counter = 0;
int totalofallclaims = 0;
int inputnumber = 0;
```

variables (list of ingredients)

```
while (counter < 2)
{
   System.out.println("What is the value of the claim: -- ");
   inputnumber = myScanner.nextInt();

   // Add the number to the total
   totalofallclaims = totalofallclaims + inputnumber;
   counter = counter + 1;

   // Print out the total of the claims that have been entered
   System.out.println("The total of the claims that have been input is " + totalofallclaims);
}
```

statements (list of statements)

Example code

This example shows Python code which will ask the user to input two values and then totals the values. The program is like our recipe, it is a set of instructions.

```
counter = 0
totalofallclaims = 0

while counter < 2:

    #Input a number
    inputnumber = int(input("What is the value of the claim: -- "))

    #Add the number to the total
    totalofallclaims = totalofallclaims + inputnumber

    #Add one to the value of count
    counter = counter + 1

# Print out the total of the claims that have been entered
print("The total of the claims that have been input is ", totalofallclaims)
```

variables (list of ingredients)

statements (list of statements)

Module summary

In this module we have learnt about programming languages and some features that apply to Java. We have learnt that:

- a computer program is a set of instructions created by a programmer
- a computer program is like a cooking or baking recipe
- the computer can perform input, process and output with the help of a program
- Java programs can be written for **console**, **Window** or **Web** applications
- there is a structure to all Java programs which include the use of packages, classes and methods, including the main method
- the keyword import is used to 'import' the classes (methods and variables) contained within the named package
- classes contain variables (properties) and methods
- methods always have the () after them e.g. main()

This is only module 1 and yet we have made great progress in learning to code. **The fundamentals are so important** and are the foundation from which we build real applications. Now we have some fundamental concepts we can progress to other aspects of programming.

What a great achievement for us.

We have learnt so much about the terms and concepts used in Java programming. We will need all this information when we start writing and reading Java code. Everything we have learnt in this module will be reinforced throughout all the other modules. We have just picked up some of the building blocks necessary to allow us to be a Java programmer.

Program Practically

Java

Module 02

Introduction – Input and Output

Console input
Console output
System.out
Scanner class
Imports

Gerry Byrne

Program Practically - Writing to and reading from the console

We learnt in module 1 that:

Under the direction of a program, written in a programming language and converted to machine readable code, the computer can perform the following:

Input The computer can accept user input from the keyboard.

Process The computer can perform arithmetic calculations and other types of processing.

Output The computer can display a message or result on the screen or other device.

This module will concentrate on how to **output to the console**. We will also use a basic Java command to **read from the console**, which is an example of **input**. It is very important to understand that what we learn by completing the simple examples in this module will:

- help us build more complex code examples in future modules
- show us commands that are used in real world applications
- get us started with two important aspect of any programming language – **input** and **output**

Looking back at something that was shown in module 1:

> We can think of the console as a black and white screen (although the colours can be changed) where input from the user is accepted and output from the computer program is displayed.

So, our console will display data and in the Java programming language we can achieve this with the line of code:

```
System.out.println();
```

Analysing this line of code.

- **Fact 1**

  Here we can see the keyword **System.** System is a final class within the java.lang package and included in the class are facilities to handle standard input, standard output and error output streams. So, we can see that this means System allows us to interact with the console, yes, the 'screen' we mentioned earlier where input from the user is accepted and output from the computer program is displayed.

- **Fact 2**

  The second part is the full stop (or period as it is also known). In programming languages like Java, the full stop means that we want to use a part or element of the **object** that appears to the left of the full stop, in this case the **out**. The object will generally be a class, and we talked about classes in the previous module. Now, if we consider that System is a class and we once again go back to what we learnt in the previous module:

  > "As we go through the course modules, we will be reminded of the fact that **a class contains variables and methods**. This is a key concept and will be relevant when writing all code".

  So, if System is a class, it can contain variables and methods, therefore when we add the full stop after the class name, we are saying we want to use either a variable or a method that is inside the class. It was also said in the previous module:

  > "Likewise, we can create packages to hold our classes and we can use the packages created by Oracle in our code to get access to the **Oracle base classes**".

  Oracle base classes will be like the classes we write, they contain variables and methods which we as developers can use, without having to write them. System is one such base class and it therefore contains variables and methods that we can use.

out is a variable (member) of the System class. It is very special as it is of the data type **PrintStream,** but let's not concern ourselves with this now, we will just use the statement to perform input and output.

- **Fact 3**

  The third part is println()

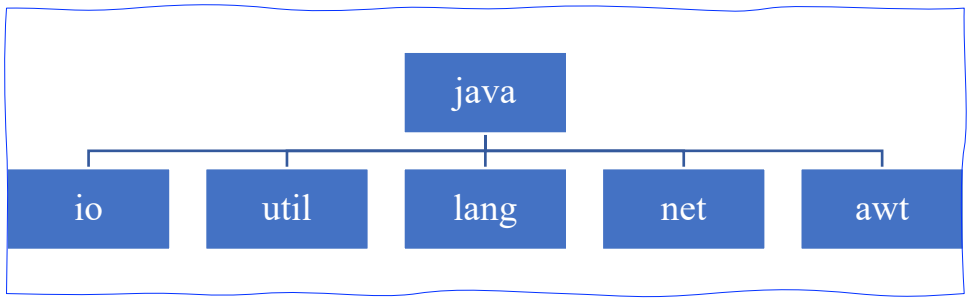
  Looking back to what we learnt in the previous module:

  > "So () means a method"

  We should now be able to recognise that println() is indeed a method and, as it has nothing between the brackets (), we should also be aware that this means the method takes in no value.

- **Fact 4**

  println() belongs to a class System which is contained in the java.lang package (we saw the java.lang connection in the diagram we saw earlier).



It is not obvious from the line of code that println() belongs to the System class, but it will become obvious as we start to write the code in our Integrated Development Environment (IDE). To explain this, we can think back to what we learnt in the previous module:

> "The lines of code at the start of the program code usually have a format that starts with the keyword **import**. The word **import**

refers to the fact that we wish to use classes (and ultimately the methods and variables in the classes) that are contained in the package that follows the word import".

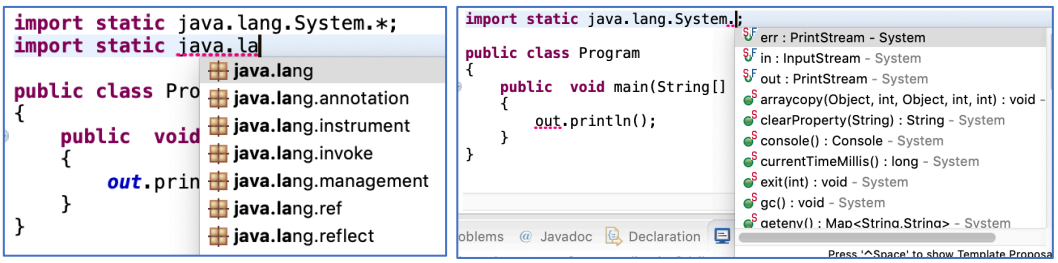
```
import static java.lang.System.*;
```

So, we can see a package called **java.lang** being used in our Java code and this illustrates another important concept to get used to when programming:

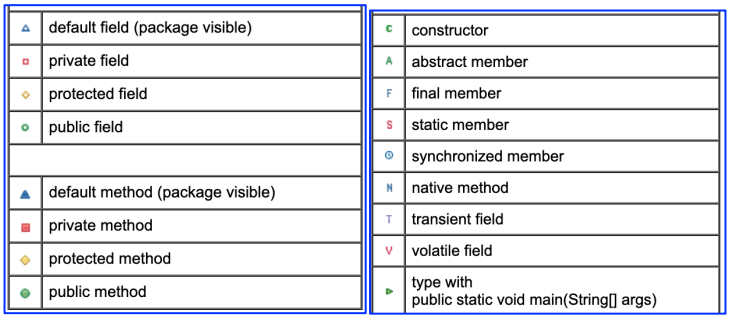
"we will use classes that already exist to help us build our own applications using Java code".

Always remember the key fact that **a class contains variables and methods**, so when we tell our code to use an existing class, which exists in a package, we are doing this to get access to variables and methods that already exist and will help us in building our application with Java code.

When we use an Integrated Development Environment like Eclipse or IntelliJ we will receive assistance when we type a class name (or package name) followed by the dot. We call this **dot notation**, and it presents us with a list of variables and methods that exist in the class, very handy for us as developers. The diagram below shows an example of the packages appearing as part of the Integrated Development Environment assistance and also the variables (properties) and methods that exist in the System class appearing, when we use the Eclipse Integrated Development Environment.



If we study the icons this will help when we are coding our applications. There are different icons representing aspects of the class. For now, simply get familiar with two of the icons that represent the variables (properties) and the methods:



The two terms and concepts, field and method, are highly important when we write programs in any programming language. It is essential that we become familiar with both terms as we will use the terms throughout the modules, and we will use them in every program we write during the modules. We will give a preliminary explanation of the two terms below:

The field

A **field** is represented in a few different ways depending on the type of field. A **field** is also called a property, variable or member of the class. Look back to what we learnt in previous module:

> "We will also see later that we use a different word from variable (property) when we talk about classes, but in our learning, they will be referred to as variables"

So, another name for a **variable** that we will use is **field** or **property.** We will also see the word **member.**

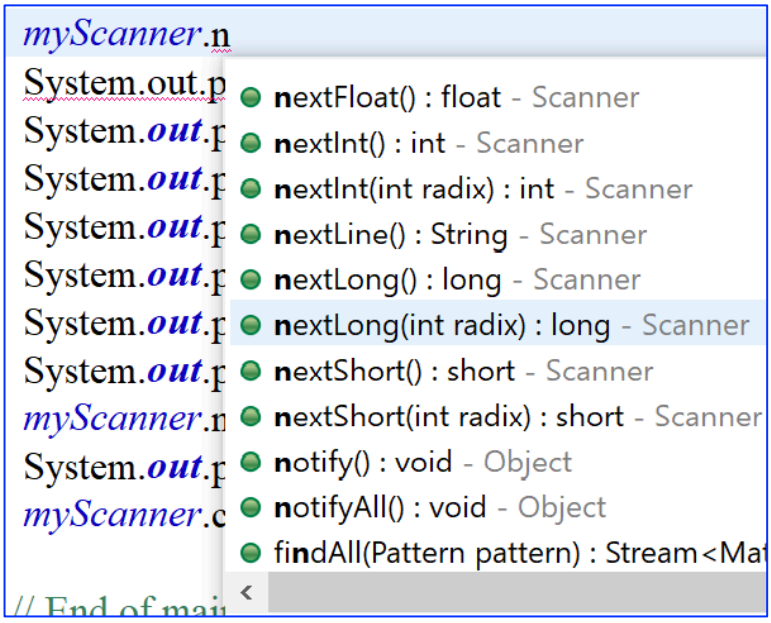
The Method

A **method** is represented in a few different ways depending on the type of method. A method is a block of code.

Let's code some Java

Now it is time for us to do some Java coding. The Java console application we will code will use the **System.out.println()** method to output data to the console window and then use some of the methods of the Scanner class to read keyboard input from the class.

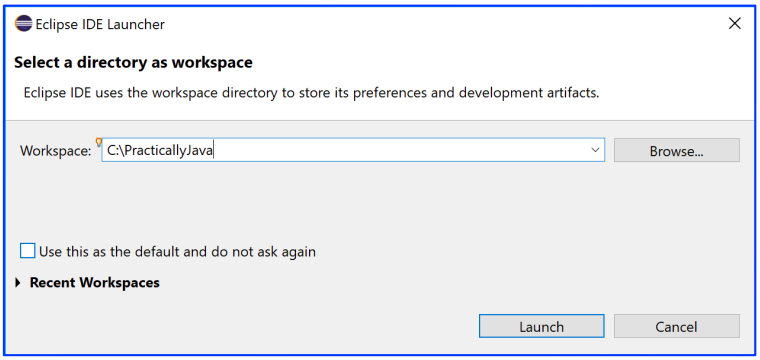
The diagram below shows some of the methods that exist in the Scanner class:



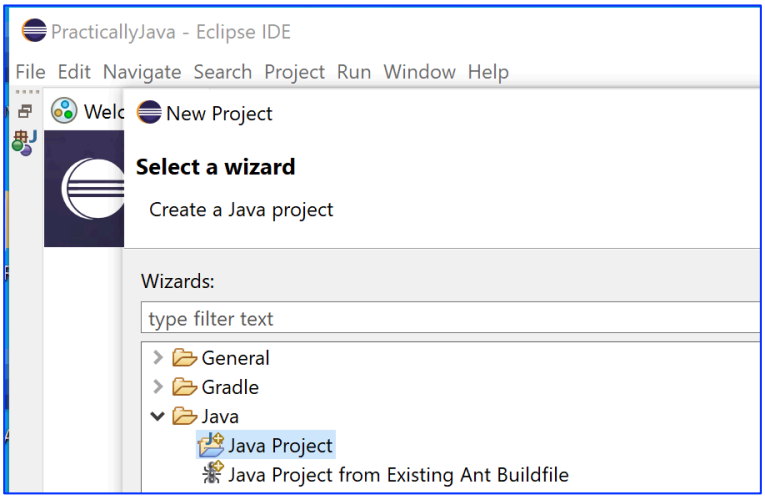
Create a workspace (a folder to hold one or more projects)

All our code will be saved in one location called a workspace. The workspace is a folder on our computer. Once we create the workspace, we will create projects within it and these projects are folders within the workspace. So, now we need to create a workspace at a location of our choice on our computer.

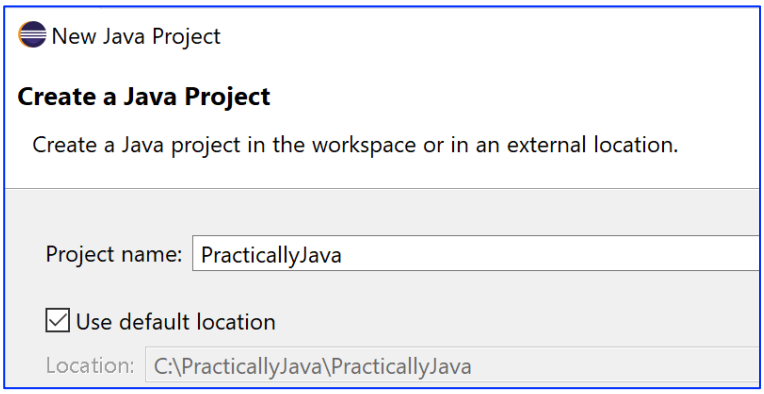
1. Open the Eclipse Integrated Development Environment.
2. At the opening screen use the Browse button to locate the directory or drive where we wish to create our code using the browse button.
3. Now we will add the name **PracticallyJava** to the end of the path location, as shown.



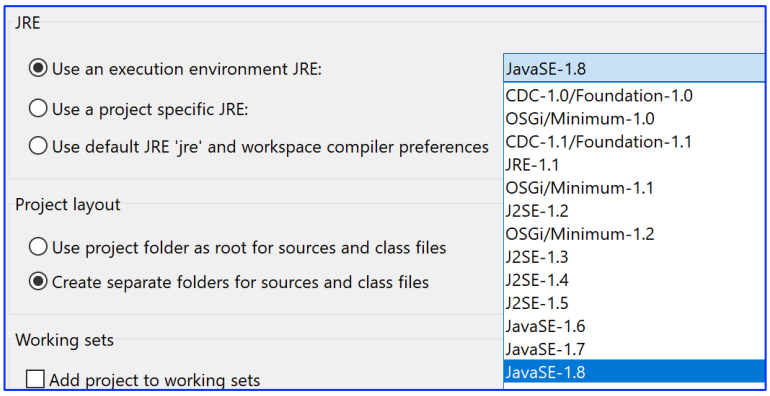
4. Click on the **Launch** button.
5. Click on the **File** menu.
6. Choose **New**.
7. Choose **Project**.
8. Expand the Java folder.
9. **Choose Java Project** as the project type.



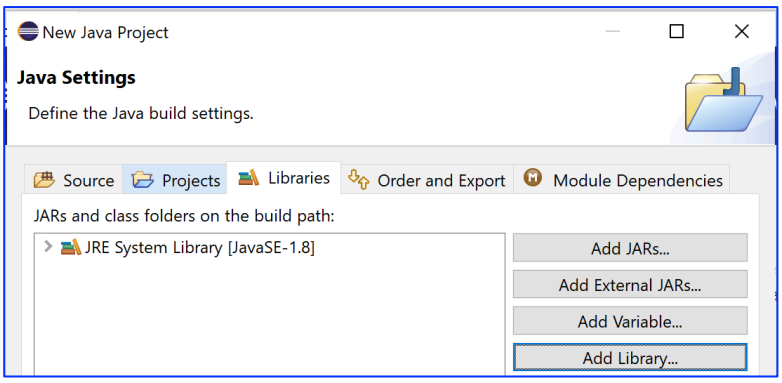
10. Click on the **Next** button.
11. In the name area type the name of the project – **PracticallyJava.**



12. In the Select a JRE drop down list choose our required version of Java (usually this will be the highest version we have). In the screenshot **version 1.8** has been selected. Version 1.8 is widely used in industry even though newer versions with some additional features are available. **When learning the fundamentals of Java version 1.8 is very suitable**.



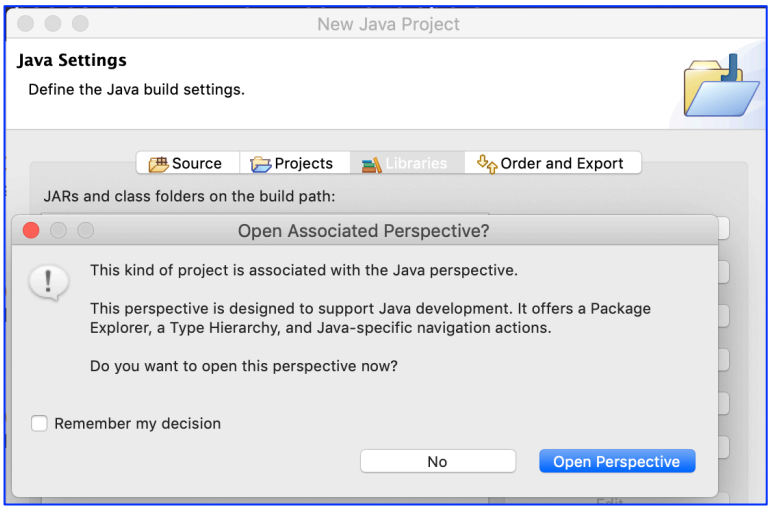
13. Leave the rest of this form as it is and then click the **Next** button.
14. Click on the **Libraries** tab.



We should see that Java 1.8, or whatever version we choose, will be shown as the JRE System Library.

15. Click the **Finish** button.

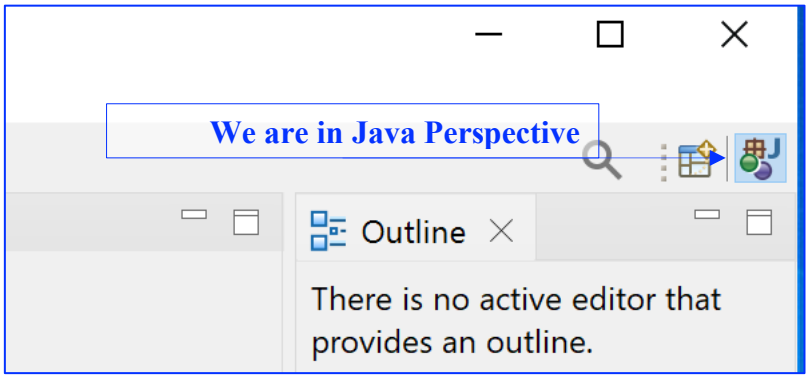
You may get a pop-up window appear and say that a Java project is associated with a Java perspective. This is telling us that the Eclipse IDE will show our Java project with tools and windows associated with a Java project. This makes it easier for us to find the required tools to create our Java code.



If this window appears click the **Open Perspective** button.

16. Close the welcome screen, if it appears, by clicking on the X on the Welcome screen tab at the top.

Look at the top right of the Eclipse screen. We should see something like the following:



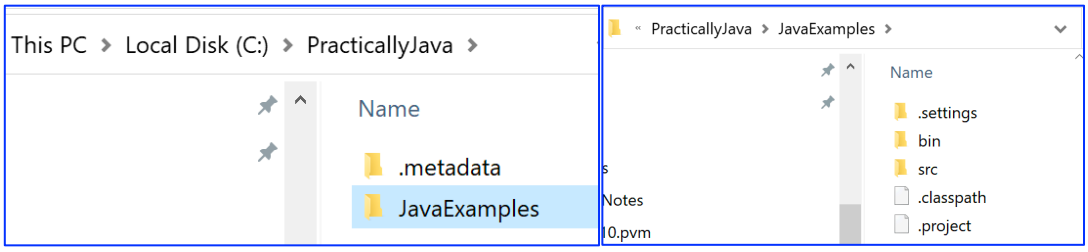
Notice the **J** in the top right corner – this indicates that we are in the Java perspective.

Now look in the **Package Explorer** panel on the left. We should see the name of our project, **JavaExamples**, beside the folder icon. Remember the project goes in a folder inside the namespace:

- our workspace is **LearnToCodeJava**
- our project is **JavaExamples**

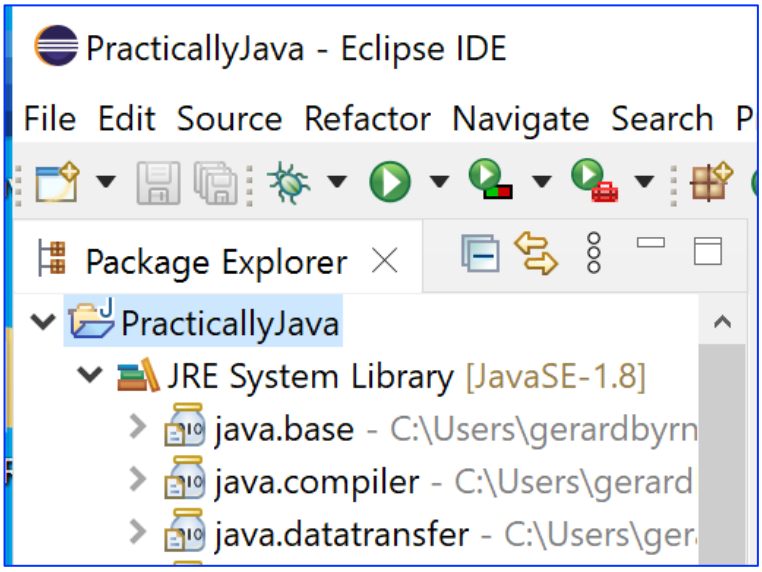
- the workspace and project are folders:
  - the top level folder is the **workspace** called **LearnToCodeJava**
  - the inner folder is the **project folder** called **JavaExamples**

The folder structure can also be seen in 'File Explorer' of our operating system:



17. Within Eclipse expand the JavaExamples project folder by clicking on the arrow to the left of the folder icon.
18. Expand the JRE System Library by clicking on the arrow to the left of the books icon.

We will see a series of **jar** files. These are **Java Archive (jar) files** and they hold packages which hold classes which hold variables and methods.



Now we will create a package that will hold our class(es), which will hold our variables and methods. Remember, classes in Java will be held within a package. In other words, these jar

Program Practically

Java

Module 03

Introduction – Commenting code

Why comment?
Single line comments
Inline comments
Multiple line comments (block comments)
Comments versus proper names
Clean code

Gerry Byrne

Program Practically - Commenting code for readability

We learnt in module 2 that:

Our application code can involve input and output and that the input and output is completed in the console window. We can say this is the visible part of our application for the user, and it is important they have a good experience when seeing the output from our application. The user experience is often referred to as the UX and can involve the use of colours, emphasis, layout etc by the developer to make the application readable and pleasing to look at.

This module will concentrate on how to create a good user experience **for the developer** when they are creating, reading and amending code. It is particularly important to understand that we as developers should be having a good user experience when we look at any application code, whether it is our code our someone else's.

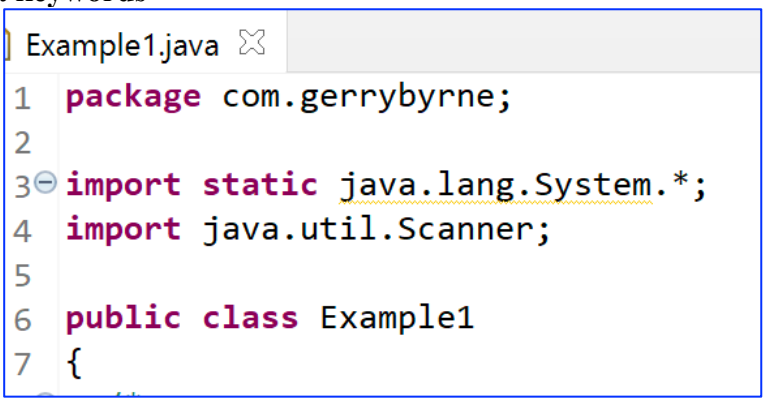
As a starting point we will say that one of the current themes in the world of programming in the commercial environment is to write code that is **self-documenting**. This is a great idea and one that we can achieve by writing Java code statements that will be easily understood by other developers who will read or use the code. In fact, it can even make the original writer of the code, us, understand it better when returning to it to make amendments.

Before we start to code, we should keep the strategy of **self-documenting code** foremost in our thoughts. Writing self-document code is easy to do and can involve:

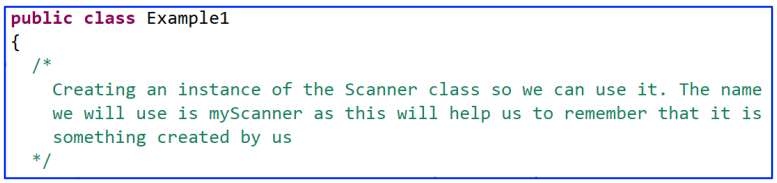
- adding **astutely placed comments** (explanations) in our code
- not overusing comments, not everything will need a comment if the code is written well, but comments can be a big help to the reader
- our **variable names** being such that they explain the purpose of the variable
- our **method names** being such that they explain the purpose of the method
- our **class names** being such that they explain the purpose of the class

Another strategy is to use colours in the coding statements, we should have already seen that the Eclipse Integrated Development Environment has coloured parts of our code. Examples of the code colouring are:

****Red - for Java keywords****



****Green - for Java comments****



This code colouring is one way in which we can help ourselves and other developers who might have to read our code.

Note

In the code examples we will use throughout the modules in this course, there will be lots of comments used to help us understand the code, but if these were commercial applications we would not have as many comments.

Make sure to read the code comments in the course code examples as they have invaluable information that adds to and supplements the text of this book.

Comments can be used to give information such as:

- a description indicating the purpose of the application
- information about the developer or developers
- the date on which the program was first create

Program Practically

Java

Module 04

Data types

Java primitive data types
Value types
char versus String
Data type conversion – implicit and explicit
Widening and narrowing conversions
Escape sequences \t and \n
Using the Scanner class to read console input as a specific data type

Gerry Byrne

Program Practically - Data types, variables and conversion

We learnt in module 3 that:

Whilst we can use single and multiple line comments, they should not be a replacement for self-documenting code. Comments are there to help the reader of the code but when the code is written expressively with proper class names, variable names etc. there is a limited need for comments and in reality, we should try for a zero need for comments approach.

In this module we will use code which is well documented for the purposes of helping us understand the code, it is not how we would do it in a real application.

We will learn from this module about the very important concepts of, **data types and variables**. We will use data types and variables in all the Java programs in this book, that is how crucial they are to Java programming. We should also be aware that data types and variables exist in all programming languages and are a fundamental building block for the code we will write.

Data Types

There are different data types in Java, but we will use the category called **Value Types**. **In Java there are 8 primitive (built in) data types**. Value types will contain data and we will hold data for value types such as:

boolean **char**

arithmetic integral types

byte **short** **int** **long**

arithmetic floating-point types

float **double**

Notice that **char** (**one character**) is a data type but there is **no String** (more than one character). Java still supports strings, but strings are a reference to an **instance of the class String.** Do not worry about this statement too much, we will use strings and characters in our

When we use narrowing conversions in our code, we will see that we must explicitly do the conversion by placing the new data type in parenthesis (), like a method. The data type to convert to, sits in front of the object to be converted. We will see this as we carry out the examples in this book, but here is an example of what the code lines might look like.

The code below shows an example of the conversion we have just talked about, int to byte, using **(byte).** The narrowing conversion is performed using **casting**.

```
package com.consoleexamples;
class Test {
    public static void main(String args[])
    {
        byte commissionfactor;
        byte commissionpremium;
        int commissionvalue = 257;
        double monthlyinsurancepremium = 296.99;
        System.out.println("Narrowing conversion from int to byte.");

        //(byte) means we wish to convert to byte the commissionvalue
        commissionfactor = (byte) commissionvalue;

        System.out.println("\nThe car emmision value is: "+ commissionvalue +
        but when converted to a byte the car emmission factor is: " + commissionfactor);
        System.out.println("\nConversion of double to byte.");
        /* (byte) means we wish to convert to byte the monthlyinsurancepremium
       So, we will now have 296.99 minus 256 which is 40 (forgetting the
       decimal places) */
        commissionpremium = (byte) monthlyinsurancepremium;
        System.out.println("\nThe monthly insurance premium is: " +
monthlyinsurancepremium + "\nand the car emmission premiumwhen is: "+
commissionpremium);
    }
}
```

More of this later.

Now we will look at adding code that will help in building an application to simulate a car insurance quotation application. Firstly, we will create a String variable called

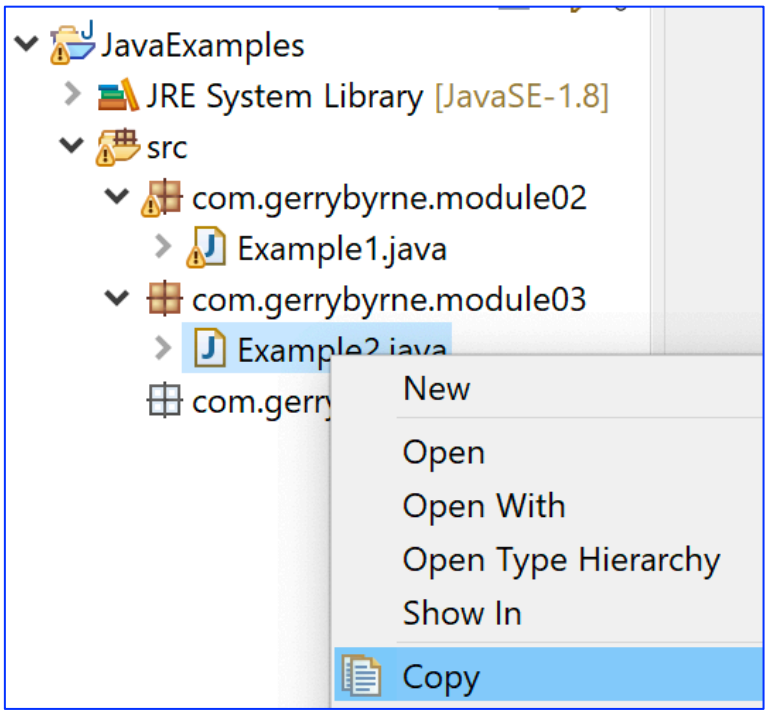
vehicleManufacturer that will hold the value typed in by the user at the console. Remember to read the comments carefully as they fully explain what we are doing.

Create a new package

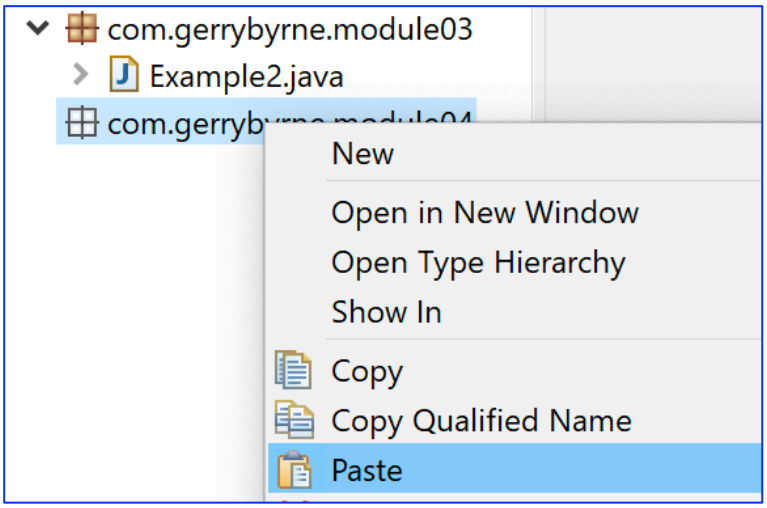
1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Enter the new name for the package, we will call it **com.gerrybyrne.module04.**
5. Click the **Finish** button.

Copy and paste a Java file from one package to another package

6. In the Package Explorer window right click on the **Example2** file in the com.gerrybyrne.module03 package.
7. Choose **Copy**.



8. In the Package Explorer window right click on the com.gerrybyrne.module04 package.
9. Choose **Paste**.



10. Expand the com.gerrybyrne.module04 package by clicking on the > symbol beside the name.

The Java file will be located in this package and it is obviously called Example2, but we will rename it to call it Example3. In module 2 we renamed the package by using the refactor option followed by the rename option, now we will do the same for the class. Once again, the package name statement which is the first line of code in the class, should automatically be amended.

Rename the Java file

11. Right click on the **Example2** file in the com.gerrybyrne.module04 package.
12. Choose **Refactor**.
13. Choose **Rename**.

Program Practically

Java

Module 05

Conversion – Casting and Parsing

Casting from one numeric data type to another

Parsing a String to a numeric data type

Wrapper classes

Gerry Byrne

Program Practically - Data types, variables, casting and parsing

We learnt in module 4 that:

We can declare variables which are of a particular data type and then assign values to them in our code. We can accept user input and assign the value input by the user to the variable. We also saw that there are times when we will need to convert a variable from one data type to another. We learnt about narrowing and widening conversions and in particular we used a conversion from an int to a byte which involved casting, (byte).

Conversions using casting and parsing

In this module we will continue the theme of conversion and look at both casting and parsing as forms of conversion from one data type to another. First let us look at the difference between casting and parsing.

Casting

In Java, casting is a method used to **convert one numeric data type to another**. Casting is used as an explicit conversion telling the compiler what to do, and to be aware that there may be a loss of data. So, we use casting to achieve a **numeric conversion** where the destination data type we are assigning the value to is of a lesser precision. **Casting is a conversion from one numeric data type to another numeric data type**.

Think back to what we have read about the numeric data types. We start with a less precise data type, byte and move to the most precise long data type. Data types in Java are shown below along with their size and default values:

Java data type	default	Size	Description
boolean	false	1 byte	Contains either true or false
char	\u0000	2 bytes	Contains a single character

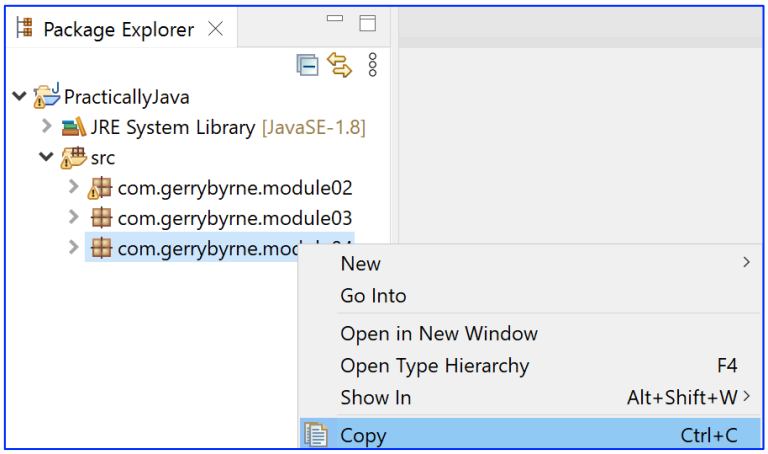
Let's code some Java

Now we will look at using casting in different ways. Remember to read the comments carefully as they fully explain what we are doing.

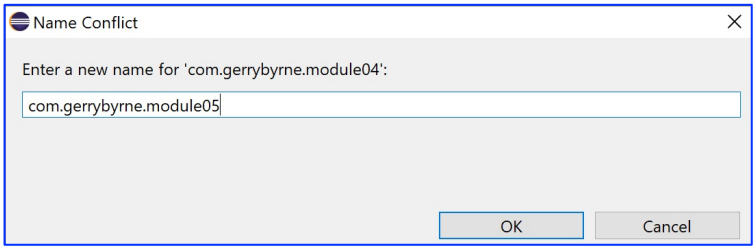
Create a new package

In the last module we saw how to create a new package, then copy and paste a Java class file from another package to the new package. Now we will do something different, this time by copying the package com.gerrybyrne.module04 and renaming it com.gerrybyrne.module05. By copying the package we will also get the Java files that are within it. This is just what we want as it will mean we do not have to copy and paste the files.

1. Right click on the package **com.gerrybyrne.module04**.



2. Choose **Copy**.
3. Right click on the **src** folder.
4. Choose **Paste**.
5. Rename the package as **com.gerrybyrne.module05.**



6. Choose the **OK** button.
7. Expand the com.gerrybyrne.module05 package by clicking on the > symbol beside the name.
8. Right click on the **Example3** file.
9. Choose **Refactor**.
10. Choose **Rename**.
11. Enter the new name for the file, we will call it **Example4**.
12. Click the **Finish** button.
13. Click the **Finish** button.

Amend the Java code to use a casting from int to short

14. Amend the existing code by adding two new variables, one of data type int and called maximumAmountForRepairCosts and the other of data type short and called minmumAmountForRepairCosts

```
int vehicleCurrentMileage;
String dateOfBirthOfMainDriver;

// max value of short is 2,147,483,647
int maximumAmountForRepairCosts = 32767;

// max value of short is 32,767
short maximumAmountForCarHire = 0;
```

15. Click on the **File** menu.
16. Choose **Save All**.
17. Amend the code to assign the value of the variable maximumAmountForRepairCosts to the variable called maximumAmountForCarHire:

```
    catch (ParseException e)
    {
        e.printStackTrace();
    }
/*
```

Program Practically

Java

Module 06

Arithmetic

Arithmetic using standard operators, + - * /

Arithmetic using non-standard operators, % += -= *= /=

Square root as a special method from the Math class

Formatting output to 2 decimal places

Gerry Byrne

Program Practically - Arithmetic Operations

We learnt in module 5 that:

Variables can be 'converted' from one data type to another which is referred to as **casting** or converting a String value to a numeric value which is called **parsing**. Parsing uses methods from a wrapper class to convert the String data to a numeric data type value. These are important concepts and widely used in all programming languages by professional developers. As we develop our Java skills throughout the modules, we will use these concepts, so it is worthwhile constantly reminding ourselves of the differences between casting and parsing and how they are used within Java code.

Arithmetic in our business logic

The code, or business logic as it is often called, of many applications will have some degree of computation or calculation. In Java it is possible to perform operations, on integers and other numerical data types, that we can perform in normal mathematics.

We will probably be aware from our mathematics lessons at school, that mathematical operations are performed in a specific order, and therefore we need to ensure that formulae are written in such a way that the mathematical operators work in the correct order. Based on this knowledge we should recognise that calculations involving combinations of the mathematical operators such as add (+), subtract (-), multiply (*) and divide (/) can return a different value (answer) when the order is changed. The normal algebraic rules of **precedence** (**priority**) apply in every programming language, including Java, and need to be thoroughly understood and applied. The precedence can be understood using the acronym BODMAS which means:

- **B**rackets
- p**O**wers
- **D**ivision These two operations have the same priority

1. Choose **Package**.
2. Name the package **com.gerrybyrne.module06**.
3. Click the **Finish** button.
4. Right click on the package icon.
5. Choose **New**.
6. Choose **Class**.
7. Name the class **Arithmetic**.
8. Put a tick in the checkbox beside the public static void main(String[] args) box.
9. Click on the Finish button.

The **Arithmetic** class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module06;

public class Arithmetic
{

    public static void main(String[] args)
    {
       // TODO Auto-generated method stub

    } // End of main method()
} // End of Arithmetic class
```

Look at the structure of the Solution in the Package Explorer window:

- the com.gerrybyrne.module06 package is now included in the workspace alongside the other packages we have created for the other modules
- the com.gerrybyrne.module06 package has the Arithmetic Java class with some template code and it should be open in the editor window. If it is not opened, double click on the Arithmetic file in the project window

We should now be getting a better understanding of the structure for projects and packages and, as we learn more, we will see how to add different classes to our projects and packages, with only one class containing the main() method. Obviously, we will need to have different

names for each class in a project, so, once we have named classes it is still possible to rename them. We can start by doing this in our project and will rename the file, Arithmetic, and call it **QuoteArithmetic**.

10. Right click on the file **Arithmetic** file in the Package Explorer window.
11. Choose **Refactor** from the menu that appears.
12. Choose **Rename** from the next menu that appears.
13. Type **QuoteArithmetic** as the new file name.
14. Click the **Finish** button.
15. Click the second **Finish** button.

The file name is amended, as can be seen in the Package Explorer window.

Now we will add the variables that will be used in our code. In the code below there are detailed comments to help us get a full understanding of the code.

Amend the Java project to add the variables we require

16. Amend the template code as shown below by adding the variables we will use in our insurance quotation application code:

```
// Program Description:     A Java program to perform arithmetical operations
// Author:                  Gerry Byrne
// Date of creation:        01/10/2021

package com.gerrybyrne.module06;
public class QuoteArithmetic {
public static void main(String[] args)
{
 /*
   We will setup our variables that will be used in the mathematical
   calculation used to produce an insurance quotation for a vehicle.
   First we will setup the variables that will hold the user input and
   that will be used in calculating the quote
 */
```

```
int vehicleAgeInYears;
int vehicleCurrentMileage;

/*
 For the quotation we will use 10000 kilometres as a base line for
 calculating a mileage factor. If the average kilomteres travelled
 per year is above the       base mileage of 10000 the mileage factor will
 be above 1, if the average kilomteres travelled per year is the lower
 than the base mileage of 10000 the mileage factor will be below 1
*/
double quoteAverageExpectedKilometres = 10000;

/*
 For the quotation we will use £100 as a base figure (this is just
 an example) and this figure will be multiplied by the mileage and age factors
*/
double quoteBaseRate = 100.00;

/*
 For the quotation we will use 10 as a base figure for the age of
 the vehicle (this is just an example). If the vehicle is older than
 10 years, the age factor will be above 1. If the vehicle is younger
 than 10 years the age factor will be below 1
*/
int quoteBaseAge = 10;

/* This variable will be used to hold the value of the age factor */
double quoteAgeFactor;

/*
 This variable holds the quote amount based on the age factor and
 the base rate
*/
double quoteAgeFactorPremium;

/*
 This variable holds the quote mileage factor based on the number of
 kilometres travelled each year and how the kilometres per year is a
 ratio of the average expected 10000 kilometres as decided by the
 insurance company
*
```

Program Practically

Java

Module 07

Selection

Selection using if construct

Selection using if else construct

Selection using if else if construct

Selection using switch construct

Switch on numeric and String values

Logical operators AND, OR and NOT

ToUppercase() and ToLowercase()

Gerry Byrne

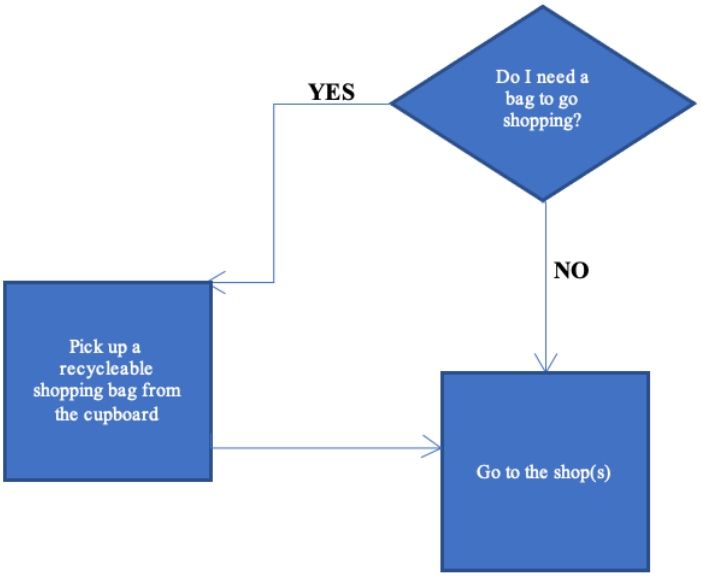
Program Practically - Selection

We learnt in module 6 that:

We could apply arithmetic operations on some variables or values and we could use the printf() method to specify the number of decimal places required in the output. We also investigated the use of less familiar arithmetic operators such as +=, -=, *=, /=.

Selection

In this module we will learn about the very important of concept selection and its use within an application. However, the concept of selection should be familiar to us through our everyday life. Many of the things we do in everyday life require us to make decisions. When making decisions in everyday life we will be directed down one path or another, as shown in the diagram below.



In a similar manner the programs we, and every developer, write normally will require us to make decisions. Decisions in our code will change the flow of execution depending on the decision made. This can be clearly seen in the diagram above where:

- a **yes** decision changes execution down the yes path and
- a **no** decision changes execution down the no path

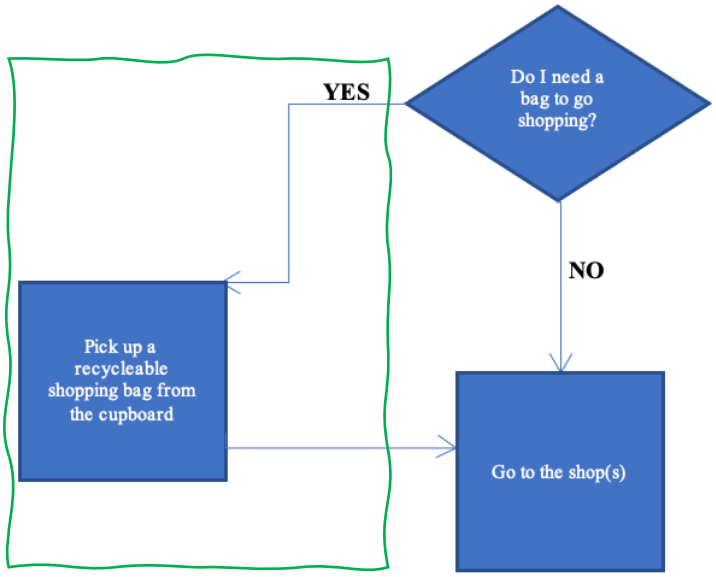
In this example the execution eventually returns to a common path (Go to the shop(s)).

```
case 5:
{
      discount = 5.00;
          break;
}
case 10:
{
      discount = 10.00;
          break;
}
default:
{
      discount = 0.00;
          break;
}
} // End of switch statement
```

Let's code some Java

The if construct

We will now use the **if construct** which has one block of code, between the curly braces, that is executed if the condition inside the brackets evaluates as boolean true. If the condition evaluates to boolean false then, with the if construct, there is no other block of code associated with it so, the next line in the program after the close curly brace, is executed. The evaluation to boolean true would be equivalent to the area highlighted by the green rectangle in the diagram below (the pathway to the left).



We should use the same workspace that we created for the earlier modules, as this will mean we will still be able to see the code we have written for the previous modules. The approach of keeping all our separate projects in one workspace is a good idea while studying this book and coding the examples.

This module will concentrate on selection, use an insurance quote example and build on our learning from the previous modules.

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **com.gerrybyrne.module07**.
5. Click the **Finish** button.
6. Right click on the package icon.
7. Choose **New**.
8. Choose **Class**.
9. Name the class **Selection**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

Program Practically

Java

Module 08

Iteration

Iteration using for construct

Iteration using while construct

Iteration using do while construct

Iteration using for each construct

Using the break and continue statements

Gerry Byrne

Program Practically - Iteration (Looping)

We learnt in module 7 that:

Selection is a particularly important programming concept in all programming languages. To use selection in our Java code we have several options and the best option to choose will depend on the particular task the code has to perform. The different formats for the selection construct are the if construct, the if else construct, the if else-if construct and the switch construct with its case label. The switch construct can use numeric or string data types and when we use strings it is case sensitive. To help in using strings with the switch construct we can make use of the ToUpperCase() or ToLowerCase() methods. We also learnt that displaying data to the console could be achieved using the format() method and 'placeholders' (%).

In terms of the project structure, we learnt that not only can we have multiple packages but within a package we can have multiple classes, each having to have unique name.

Introduction to Iteration

Many of the things we do in everyday life require iteration. Think about making a number of slices of toast in a toaster. The instructions could be:

- take a slice of bread from the recycleable packaging
- put the slice of bread in the toaster
- pull the toaster lever down to start the heating process
- when the toast pops up remove the slice of toast from the toaster
- put the slice of toast on a plate
- **repeat** the process the required number of times

Think about brushing our teeth – move the toothbrush left and right the required number of times.

The concept of iteration is important in programming and the Java language offers us a number of different structures to perform iteration. In this module we will look at the Java iteration (loops) constructs and concepts including:

- the **for** loop
- the **while** loop
- the **do** loop
- the **foreach** loop
- the **break** statement
- the **continue** statement

The principle of iteration is to repeat a sequence of Java instructions (a block of code) a number of times. The number of times is determined by the type of loop structure, as we will see when we code each type of loop structure.

For Loop

The first structure we will look at is the **For Loop** which will allow us to repeat a sequence of instructions a set number of times. The for statement will repeat the block of code, a number of lines of code, while a Boolean expression evaluates to true.

The format of the for loop is shown below:

```
for(<Start value>; <Condition>; <increment value>)
{
   <statements>
 }
```

There are 3 parts to the for construct, the:

- **start value – which will be of data type int**
- **condition – which will equate to true or false**
- **increment – which will change the start value by a specified amount**

Example:

```
for (int counter = 0; counter < 2; counter++)
{
        block of code statements
}
```

In the example code:

- a local variable called **counter** is set up inside the brackets ()
- the **variable** will be used as the loop counter and help to decide how many times the block of code is executed
- the variable is created as an integer and set to have an **initial value of 0** (it does not have to be 0 as the starting point). **This is the first part of the for loop, the start value**
- the loop counter is **compared** with the value 2, and if it less than 2, the execution of the block of code continues. **This is the second part of the for loop, the condition**
- the loop counter is **incremented** (increased) by 1. **This is the third part of the for loop, the increment, we could also decrement**
- each section is separated by a semi-colon ;
- all of this is enclosed in the brackets **()**
- the block of code to be executed the required number of times is enclosed between curly braces {}

The for statement can be exited early, if this is required, by using the **break** statement. It is also possible to have the code move to the next iteration in the loop by using the keyword **continue**.

This module will concentrate on selection using an insurance quote example and will build on our learning from the previous modules.

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **com.gerrybyrne.module08.**
5. Click the **Finish** button.
6. Right click on the package icon.
7. Choose **New**.
8. Choose **Class**.
9. Name the class **Iteration**.

10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The **Iteration** class code will appear in the editor window and will be similar to the following code:

```
package com.gerrybyrne.module08;

public class Iteration
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	} // End of main method()
} // End of Iteration class
```

When a vehicle is involved in an accident and requires repair, it could go to a repair centre which has been nominated by the insurance company. When the repairs are completed the repair centre will recoup their costs from the insurance company. We will now develop a program that will ask the user from the repair shop to enter the details required by the insurance company. The details will be:

- the repair shop unique id (String)
- the vehicle insurance policy number (String)
- the claim amount and (double)
- the date of the claim (date)

Now we will add the variables that will be used in our code. In the code below there are detailed comments to help us get a full understanding of the code.

Let's code some Java

12. Amend the code to create an instance of the Scanner class, calling the instance myScanner and importing the Scanner class

```
package com.gerrybyrne.module08;

import java.util.Scanner;

public class Iteration {

 public static void main(String[] args)
 {
        Scanner myScanner = new Scanner(System.in);
```

13. Amend the code to add the variables we will require:

```
 public static void main(String[] args)
 {
     Scanner myScanner = new Scanner(System.in);
      /*
     We will set up the variables to be used in the quote application
     The details will be:
        - the repair shop unique id                (String)
        - the vehicle insurance policy number      (String)
        - the claim amount and                     (double)
        - the date of the claim                    (date)
    */
     String repairShopID;
     String vehiclePolicyNumber;
     String claimDate;
     double claimAmount;
 } // End of main method()
} // End of Iteration class
```

14. Amend the code to include a for construct which will iterate twice:

```
double claimAmount;

  for (int claimsCounter= 0; claimsCounter < 2; claimsCounter ++)
  {
  } // End of for loop
 } // End of main method()
} // End of Iteration class
```

Program Practically

Java

Module 09

Arrays

Data structure

'Collection'

Homogeneous data types

Fixed size

Declare and create an array

Gerry Byrne

Program Practically - Arrays

We learnt in module 8 that:

Iteration is a very important programming concept in all programming languages. To use iteration in our Java code we have a number of options and the best option to choose will depend on the particular task the code has to perform. The different formats for the iteration construct are the for construct, the while construct, the do while construct and the foreach loop. Within the constructs there are options to break out of the iterations completely or to break out of a particular iteration using the continue keyword. In terms of the project structure we once again used the ability to have multiple classes within a package where each class has to have a unique name.

Introduction to single dimensional array

An array is a list of data items all of the same type. We could also describe it as a **collection** of data items all of the same type. We could have an array which contains a:

- list of integers
- list of real numbers
- list of characters
- list of strings

If we think about a Java application (program) which is applicable to a business that sells food products it may contain arrays for:

- vegetables - this could be list of strings
- cheeses - this could be list of strings
- product codes - this could be list of integers

Insurance cost single dimensional array

insurancepremium [0] = 104.99	The first item in the array is indexed as 0
insurancepremium [1] = 105.99;	The second item in the array is indexed as 1
insurancepremium [2] = 106.99;	The third item in the array is indexed as 2
insurancepremium [3] = 107.99;	The fourth item in the array is indexed as 3
insurancepremium [4] = 108.99;	The fifth item in the array is indexed as 4
insurancepremium [5] = 109.99;	The sixth item in the array is indexed as 5

Exercise One - Declare and create string arrays in 2 stages with no initialisation

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **com.gerrybyrne.module09**.
5. Click the **Finish** button.
6. Right click on the package icon.
7. Choose **New**.
8. Choose **Class**.
9. Name the class **Arrays**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The **Arrays** class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module09;

public class Arrays
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
	} // End of main method()
} // End of DoWhileIteration class
```

As we have seen earlier and have coded as an example - when a vehicle is involved in an accident and has to be repaired, the repair shop has to supply specific details to the insurance company so they can be reimbursed for the costs. The details required are:

- the repair shop unique id (String)
- the vehicle insurance policy number (String)
- the claim amount and (double)
- the date of the claim (date)

When we coded this program as part of the last module on iteration, we were aware that any data entered was not stored by the program code. We were made aware that this 'flaw' would be rectified using an array. So, now the time has come to amend the last program so that the data entered by the repair shop will be stored, for the duration that the program runs.

To do this we will need to:

- **declare an array** – we need to decide what data type the array will hold

  Remember from the information at the start of this module, an array can only hold variables of the **same data type**, an array is **homogenous**. We have Strings, a double and a Date (as a String), so what data type will we us? Well, the answer is the String data type. This will mean that the double and the Date will have to be converted to a string value.

- use a name for the array. We will use **repairShopClaims** for the array name

- **create the array** – using the new keyword and stating the size of the array

  Remember from the information at the start of this module, an array has a fixed size and the compiler needs to know the size of the array at compile time.

- add the values to the array in correct position

Let's code some Java

1. Amend the code to create an instance of the Scanner class, so we can accept use input, calling the instance myScanner and importing the Scanner class.

```
package com.gerrybyrne.module09;

import java.util.Scanner;

public class Arrays
{
   public static void main(String[] args)
   {
      Scanner myScanner = new Scanner(System.in);

   } // End of main method()
} // End of DoWhileIteration class
```

2. Amend the code to declare and create the array that will hold the 8 items of data input by the user:

```
   public static void main(String[] args)
   {
      Scanner myScanner = new Scanner(System.in);
   /*
    The array is going to hold the data for 2 claims. Each claim has
    four pieces of information. The number of data items is therefore
      2 multiplied by 4 = 8. So, we will make the array for this example
      of size 8. Not the best way to do things, but fine for now.
      */
      String[] repairShopClaims = new String[8];
   } // End of main method()
} // End of DoWhileIteration class
```

3. Amend the code to add the variables to be used, some variables are initialised:

```
      String[] repairShopClaims = new String[8];
```

Program Practically

Java

Module 10

Methods

Methods and modularisation

Void methods

Value methods

Parameter methods

Method overloading

User designed methods and Java class methods

Gerry Byrne

Program Practically - Methods

We learnt in module 9 that:

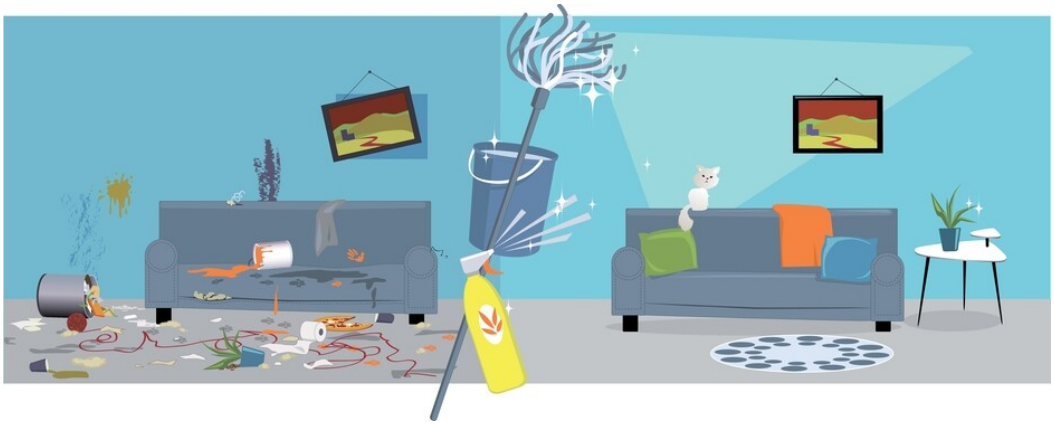
Arrays are a very important programming structure when we need to 'store' a collection of data, variables of the same data type. We saw that arrays in Java are of a fixed size and once we declare the size of the array its size cannot be altered. Each item in an array can be referenced using its index, which is also called its subscript and we can use the for each loop to iterate the array items. With the for each iteration, we do not need to use a counter as the for each construct handles the indexing for us. If we wish to reference an index in an iteration, we can use the more traditional for, while or do while iteration. We also learnt that we could cause an IndexOutOfBounds Exception if we are not careful in our coding.

Methods - concepts of methods and functions

Most commercial programs will involve large amounts of code and from a maintenance and testing perspective it is essential that the program has a good structure. Professionally written and organised programs allow those who maintain or test them to:

- follow the code and the flow of events easier
- find things quicker

Look at this image and think which side fits with a sense of being organised:



Exercise One – Create and use void methods

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **com.gerrybyrne.module10**.
5. Click the **Finish** button.
6. Right click on the **com.gerrybyrne.module10** package icon.
7. Choose **New**.
8. Choose **Class**.
9. Name the class **MethodsVoid**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The **MethodsVoid** class code will appear in the editor window and will be similar to the following:

```
package gerrybyrne.module10;
public class MethodsVoid
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
	} // End of main method()
} // End of MethodsVoid class
```

We are now going to use the same code that we created for the Arrays1D program but we will make the code more maintainable by creating multiple methods. Shown below is the code that we will work with and the outlined rectangular shapes represent the methods we will create and then call as required.

DO NOT TYPE THE CODE BELOW, IT IS FOR REFERENCE ONLY AND IS THE SAME AS WE CODED IN THE LAST MODULE.

```
package gerrybyrne.module09;
import java.util.Scanner;

public class Arrays
{
 public static void main(String[] args)
 {
        Scanner myScanner = new Scanner(System.in);

   /*
     The array is going to hold the data for 2 claims. Each claim has
     four pieces of information. The number of data items is therefore
     2 multiplied by 4 = 8. So we will make the array for this example
     of size 8.
     Not the best way to do things but fine for now.
   */
        String[] repairShopClaims = new String[8];

     /*
     We will setup our variables that will be used in the quote application
     The details will be:
         - the repair shop unique id                    (String)
         - the vehicle insurance policy number          (String)
         - the claim amount and                         (double)
         - the date of the claim                        (String)
   */

        String repairShopID;
        String vehiclePolicyNumber;
        double claimAmount;
        String claimDate;
        int numberOfClaimsBeingMade;
        int numberOfClaimsEntered = 0;
        int arrayPositionCounter = 0;

     /* Read the user input for the number of claims being made and convert
        the string value to an integer data type*/
     System.out.println("How many claims are we wishing to make?\n");
     numberOfClaimsBeingMade = myScanner.nextInt();
```

1

```
/* As we are using a variable in the loop our code is flexible and can be used
   for any number of claims. An ideal situation and good code.
*/
 do
 {
   System.out.println("The current value of the counter is :" +
         numberOfClaimsEntered + "\n");                                    2

/*
Read the user input for the repair shop id and keep it as a string
*/

   System.out.println("What is our repair shop id?\n");
   repairShopID = myScanner.next();                                        3

/*
Write the first input value to the array and then increment the
   value of the arrayPositionCounter by 1.
*/

   repairShopClaims[arrayPositionCounter] = repairShopID;
   arrayPositionCounter++;                                                 4

/*
Read the user input for the vehicle policy number and keep it as a string
*/

   System.out.println("What is the vehicle policy number?\n");
   vehiclePolicyNumber = myScanner.next();                                 5

/*
Write the second input value to the array and then increment the
 * value of the arrayPositionCounter by 1
*/

repairShopClaims[arrayPositionCounter] = vehiclePolicyNumber;
arrayPositionCounter++;                                                    6

/*
Read the user input for the repair amount and convert it to a double
*/
```

Program Practically

Java

Module 11

Classes and objects

Classes as data structures

Classes as a template for objects

Classes and the main() method

Class methods and properties (fields, members)

Class constructors

Classes getters and setters

Gerry Byrne

Program Practically - Classes and Objects

We learnt in module 10 that:

Methods belong inside classes and classes consist of variables and methods. We saw that there are a number of different method types that can be used in our code. The method types we can create or use are:

- void methods that return no value and they simply execute code
- value methods that return a value of a specific data type having executed code
- parameter methods that accept actual values as their parameters and which may or may not return a value of a specific data type having executed code
- overloaded methods which are methods with the same name but different parameters.

There are many methods used in our code which are not written by us, examples include println(), print(), nextInt(), next(), nextDouble, toString().

The methods we have created and the methods we have not created, but have used, all have one thing in common:

They all live inside a class, they are part of a class.

It is the commonality of classes that this module will be concentrating on.

The crucial takeaway from the last module and a vital thing to remember in this module is that a class contains:

- **methods** and
- **variables**

As we have just seen in the methods module, all of what we have achieved so far in our code has used our own class, where we created our own variables (properties) and methods. We have also used methods that are provided as part of the Java Framework or through the imports and this has saved us much coding and effort.

All roles have separate concerns, but all serve one purpose, to keep the hospital functioning.

Let's code some Java

Now it is time for us to code some classes with methods and show the separation of concern working in our application. We will be programming the same application that we have just completed in the methods module, so we can choose to copy and paste as required, but the instructions below assume that we are starting again with no code.

Exercise One – Create a class without a main() method

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package **com.gerrybyrne.module11**.
5. Click the **Finish** button.
6. Right click on the **com.gerrybyrne.module11** package icon.
7. Choose **New**.
8. Choose **Class**.
9. Name the class **ClaimApplication**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the Finish button.

The **ClaimApplication** class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module11;

public class ClaimApplication
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
	} // End of main method()
} // End of ClaimApplication class
```

12. Right click on the **com.gerrybyrne.module11** package icon.
13. Choose **New**.
14. Choose **Class**.
15. Name the class **ClaimDetails**.
16. **DO NOT** put a tick in the checkbox beside the public static void main(String[] args) box.
17. Click on the Finish button.

The **ClaimDetails** class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module11;

public class ClaimDetails
{
} // End of ClaimDetails class
```

We are now going to use the same code, with some small changes, that we created for the MethodsValue program, but the methods will be contained within the ClaimDetails class and will be called from within the ClaimApplication class which contains the main() method. This will now ensure that we have some degree of separation.

Remember that the methods in the MethodsValue program were numbered, so the instructions below will reference the methods by their number. We will then add further classes to enforce the concept of classes and objects.

18. Amend the ClaimDetails code to add an instance of the Scanner class so input can be read from the keyboard:

```
package com.gerrybyrne.module11;
import java.util.Scanner;

public class ClaimDetails
{
   Scanner myScanner = new Scanner(System.in);
} // End of ClaimDetails class
```

Program Practically

Java

Module 12

String Handling

String literal

Strings as immutable objects

Strings and the new keyword

String methods

Gerry Byrne

Program Practically - String Handling

We learnt from all previous modules about the core constructs of a Java program including the structure of a Java program with the concept of a class containing methods and properties and inherent in this was the concept of data types for the properties. We learnt how to structure code in the form of methods within classes and the importance of the main() method to start the programming running. We looked at the important constructs of selection to make choices, iteration to repeat blocks of code and we saw how to temporarily store data in a data structure called an Array. Now we will take a closer look at Strings which have been part of many of our code examples.

String Handling

Throughout our modules we used Strings and in many of our coding examples we saw how we could concatenate a String to another String or to a non-String data type which has been converted to a String. When we look back to the data types module we can see that String is not one of the primitive data types whereas char is. So, we know that String as a data type is special, just look at the fact that we use a capital S when coding it. The capital S might then suggest to us that it is a class, as classes by convention start with a capital letter. Indeed, we are correct in this assumption and we would say that String is a class.

We said that a character is represented by the char primitive data type, but a String is a sequence of characters or a character array, char[], and it can be used for the purpose of representing a character sequence in a convenient way. We read in a previous module that:

> One important thing to note is the capital letter of the 'data type'. When we see the capital letter, we are not using a data type the way we did when we were defining a variable, we are using **an object**, **a class**. A **Wrapper class** is the fancy name for it.

However, this comment applied to the primitive data types and String is not a primitive data type. So, let us be clear, the String class is not a wrapper class. To reinforce this the table below shows the Java wrapper classes.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

As we have said a **String** is a sequence of characters e.g. "Home Insurance" is a string of 14 characters. A String in Java is an **immutable object,** it is constant and therefore cannot be changed once it has been created. The String class offers us many String methods and properties that we can use to manipulate our Strings and we will now look at arrange of these.

Creating a String

We can create a String in one of two ways in Java

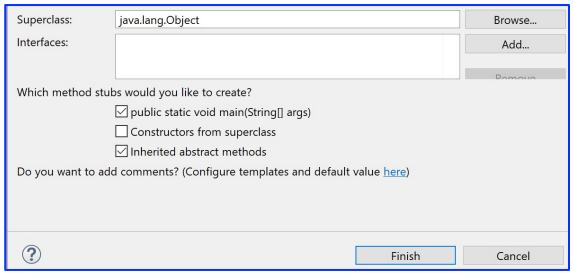
1. as a **String literal.**
2. using the **new** keyword.

String literal

Create a new package

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package as **com.gerrybyrne.module12.**
5. Choose the **Finish** button.
6. Right click on the **com.gerrybyrne.module12** package.

7. Choose **New**.
8. Choose **Class**.
9. Enter the new name for the file, we will call it **Strings**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.



11. Click on the **Finish** button.

The Strings class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module12;

public class Strings
{
        public static void main(String[] args)
        {
                // TODO Auto-generated method stub
        } // End of main method()
} // End of Strings class
```

12. Amend the code to create three string literals, 2 of which will have the same content:

```
package com.gerrybyrne.module12;

public class Strings
{
  public static void main(String[] args)
  {
    /*
     A String is an object in Java and when we created classes we saw that we
     instantiated the class we used the new keyword. Here we have not created any
     string object using the new keyword. However the compiler will perform the
     task for us and it will create a String object from the String literal.
```

Program Practically

Java

Module 13

File Handling

File streaming with Java I/O class

Write to a text file, FileWriter and BufferedWriter

Read from a text file, FileReader and BufferedReader

Try catch exception handling when using File Handling

Gerry Byrne

Program Practically - File Handling

We learnt throughout all the previous modules about the core constructs of a Java program and saw how we can write data to an Array. We also read that an array is used to temporarily store data in a data structure. Now we will look at how to store the data in the more permanent form of a text file. Once we have seen how to write to a text file, we should easily be capable of writing the data to a database, but for this course we will not get into the setting up of a database.

It is common for developers to need to interact with files within their applications. Within the Java framework we are provided with many interfaces, classes and methods to help us interact with the file system. Files provide a means by which our programs can store and access data. Within Java the file I/O is a subset of Java's overall I/O system. The core of Java's I/O system is packaged in java.io.

An Overview of File Handling

In Java, file handling is taken care of by file I/O which is just part of Java's overall I/O system. Key points in respect of the I/O system are:

- the Java I/O system is built on interrelated classes which are organised in a hierarchy
- the most important classes are abstract classes that define much of the basic functionality shared by all specific concrete subclasses
- the stream concept ties together the file system because all I/O operations occur through a stream

Streaming – Old Java versus New Java

When we discuss file handling, we will inevitably come across the term **streaming** and we need to be sure what streaming in Java means. In versions of Java before 1.7 we had streams and they were associated with I/O (input and output). Streams were therefore used to read the contents of a file or to write the contents of a file.

Then came Java 1.8 and we met the Stream, yes, the same name, but used for a completely different purpose than the traditional, read or write the contents of a file. The modern Stream is used to **manipulate** a collection of data, a data structure. A Stream does not store data and, in that sense, is not a data structure, and it will never alter the underlying data source.

So, we have streams and Streams, the old and still used stream to read or to write the contents of a file and the new Streams that are used to manipulate a collection of data.

Writing to a file

Create a new package

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package as **com.gerrybyrne.module13**.
5. Choose the **Finish** button.
6. Right click on the **com.gerrybyrne.module13** package.
7. Choose **New**.
8. Choose **Class**.
9. Enter the new name for the file, we will call it **WriteFile**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The WriteFile class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module13;
public class WriteFile
{
    public static void main(String[] args)
    {
        // TODO Auto-generated method stub
    } // End of main method()
} // End of WriteFile class
```

12. Amend the code to add the required imports:

```
package com.gerrybyrne.module13;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile
{
   public static void main(String[] args)
   {
         // TODO Auto-generated method stub
   } // End of main method()
} // End of WriteFile class
```

13. Amend the code to add a String variable that will hold the pathname of the text file:

```
   public static void main(String[] args)
   {
     // Assign the name of the file to be used to a variable.
     String filePath = "txtOutputFile.txt";

   } // End of main method()
} // End of WriteFile class
```

This sets up a variable to hold the file path and name of the text file that is to be written. The text file will be created in the same directory as the bin folder and therefore we can simply type in the file name. If the file was to be stored in a directory above the bin folder the pathname would be set to "../txtOutputFile.txt".

14. Amend the code to add an instance of the FileWriter class, inside a try catch construct, passing it the name of the file which we have already set up in a variable called filePath. Do not worry about any error messages, the code is incomplete:

```
   // Assign the name of the file to be used to a variable.
   String filePath = "txtOutputFile.txt";
```

Program Practically

Java

Module 14

Serialisation of an object (class entity)

Serialisation

Deserialisation

Transient keyword

Gerry Byrne

Program Practically - Serialisation

In a previous module we gained knowledge of classes and objects. This module will extend our knowledge and explain how we can save an object so it can be recreated when required. The processes we will investigate are called serialisation and deserialisation.

Serialisation

Serialisation is a process to convert an object into a stream of bytes so that the bytes can be written into a file. We will normally do this so the serialised data can be used to store the data in a database or for sending it across a network e.g. to a message queue to form part of a transaction process. The byte stream created is platform independent, it is an object serialised on one platform that can be deserialised on a different platform.

A class must implement the **java.io.Serializable** interface to be able to serialise data. The Java Serialisation API provides the features to perform serialisation ad de-serialisation.

Example

```
public class Customer implements java.io.Serializable
{
}
```

De-serialisation

De-serialisation is the process of taking the serialised data (file) and returning it to an object as defined by the class.

Access modifier - transient

The keyword **transient** is a variable modifier which can be used in serialisation. When we serialise, there may be some values we do not want to save to the file. These values may contain sensitive data or data that can be calculated again. Changing the variable modifier to transient means that during the serialisation process the Java Virtual Machine (JVM) will

ignore the transient variable value and save the default value of the variable data type.
The transient keyword assists us with the important role of meeting security constraints when we do not want to save private data in a file. It is important to use the transient keyword with private confidential fields (members) of a class during serialisation.

Let's code some Java

Serialisation is about objects and an object, as we know, is an instance of a class. So, let's create the class first with its properties, methods, constructor, getters and setters. The class will be called **Customer**.

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package as **com.gerrybyrne.module14**.
5. Choose the **Finish** button.
6. Right click on the **com.gerrybyrne.module14** package.
7. Choose **New**.
8. Choose **Class**.
9. Enter the new name for the file, we will call it **Customer**.
10. **Do not put a tick in the checkbox** beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The Customer class code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module14;

public class Customer
{

} // End of Customer class
```

12. Amend the code so that the class implements java.io.Serializable:

```
package com.gerrybyrne.module14;

public class Customer implements java.io.Serializable
{
} // End of Customer class
```

13. Amend the code to add the class properties (members, fields, variables):

```
package com.gerrybyrne.module14;

public class Customer implements java.io.Serializable
{
 /*****************************************************************
  transient is a variable modifier which can be used in serialization.
  When we need to serialise there may be some values we do not want to save to
  the file. These may be sensitive data or data that can be  calculated again.
  Changing the variable modifier to transient means  that during the serialisation
  process the Java Virtual Machine (JVM) will ignore the transient variable value
  and save the default value of the variable data type.

  The transient keyword assists us with the important role of meeting security
  constraints e.g. when we do not want to save private data     in file. It is important
  to use the transient keyword with private confidential fields (members) of a class
  during serialization.
  *****************************************************************/
  private  int          customerAccountNumber;
  private  int          customerAge;
  private  String       customerName;
  private   String      customerAddress;
  private  int          customerYearsWithCompany;

} // End of Customer class
```

14. Amend the code to add a constructor for the class:

```
  private  String       customerName;
  private   String      customerAddress;
  private  int          customerYearsWithCompany;
```

Program Practically

Java

Module 15

Common programming routines

Gerry Byrne

Program Practically - Applying your knowledge to some common theoretical routines

Linear search

A Linear search is used to search a series of elements in a data structure for a specified a key element. Whilst a linear search can be used successfully it will generally be slower than a binary search. It can be thought of as a 'brute-force' algorithm as it simply compares each item in the data structure with the element being searched for. It does not need to change the state of the data structure before it begins e.g. it does not need to sort the elements to have them in a chronological or alphabetic order. We can think of a linear search as undertaking the steps below or following the algorithm:

- navigate through the data structure one element at a time
- check if the element being searched for matches the current element of the data structure
- if there is a match, the element is found and we return the index (position) of the current data structure element
- if there is no match, the element has not been found and we return the value -1

Let's create an application that will implement a linear search.

1. Right click on the **src** folder.
2. Choose **New**.
3. Choose **Package**.
4. Name the package as **com.gerrybyrne.module15**.
5. Choose the **Finish** button.
6. Right click on the **com.gerrybyrne.module15** package.
7. Choose **New**.
8. Choose **Class**.
9. Enter the new name for the file, we will call it **LinearSearch**.
10. Put a tick in the checkbox beside the public static void main(String[] args) box.
11. Click on the **Finish** button.

The LinearSearch code will appear in the editor window and will be similar to the following:

Binary search (Iterative Binary Search)

A Binary search is used to search a series of elements (the data structure) for a specified a key. Unlike the linear search the binary search only works when the array is sorted. The binary search starts with the whole sorted array and checks if the value of our search key is less than the item in the middle of the array and if it:

- is, the search is narrowed to the lower (left) half of the array
- is not, then we use the upper (right) half and we repeat the process until the value is found or there are no elements to half

1. Right click on the **com.gerrybyrne.module15** package.
2. Choose **New**.
3. Choose **Class**.
4. Enter the new name for the file, we will call it **BinarySearch**.
5. Put a tick in the checkbox beside the public static void main(String[] args) box.
6. Click on the **Finish** button.

The BinarySearch code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module15;

public class BinarySearch
{
   public static void main(String[] args)
   {
       // TODO Auto-generated method stub
   } // End of main() method

} // End of BinarySearch class
```

7. Amend the code to add a comment block, create an array and initialise the values:

```
package com.gerrybyrne.module15;

/*
With a binary search we must first ensure the array is sorted.
The binary search starts with the whole array and checks if the
```

Bubble sort

A Bubble sort is a simple algorithm which compares two adjacent elements of the array. If the first element is numerically greater than the next one, the elements are swapped. The process is then repeated to move across all the elements of the array.

1. Right click on the **com.gerrybyrne.module15** package.
2. Choose **New**.
3. Choose **Class**.
4. Enter the new name for the file, we will call it **BubbleSort**.
5. Put a tick in the checkbox beside the public static void main(String[] args) box.
6. Click on the **Finish** button.

The BubbleSort code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module15;

public class BubbleSort
{
   public static void main(String[] args)
   {
     // TODO Auto-generated method stub

   } // End of main() method

} // End of LinearSearch class
```

7. Amend the code to add a comment block, create an array and initialise the values:

```
package com.gerrybyrne.module15;

/*
A Bubble sort is a simple algorithm which compares two adjacent elements
of the array. If the first element is numerically greater than the next one, the
elements are swapped. The process is then repeated to move across all the
elements of the array.
 */
```

Insertion sort

An Insertion Sort is similar to a Bubble sort, however, it is a more efficient sort. We should think about using the Insertion sort when we have a small number of elements to sort. Larger data sets will take more time.

1. Right click on the **com.gerrybyrne.module15** package.
2. Choose **New**.
3. Choose **Class**.
4. Enter the new name for the file, we will call it **InsertionSort**.
5. Put a tick in the checkbox beside the public static void main(String[] args) box.
6. Click on the **Finish** button.

The InsertionSort code will appear in the editor window and will be similar to the following:

```
package com.gerrybyrne.module15;
public class InsertionSort
{
   public static void main(String[] args)
   {
            // TODO Auto-generated method stub
   } // End of main() method
} // End of InsertionSort class
```

7. Amend the code to add a comment block, create an array and initialise the values:

```
package com.gerrybyrne.module15;

/*
An Insertion Sort is similar to a Bubble sort, however, it is a more efficient
sort. We should think about using the Insertion sort when we have a small
number of elements to sort. Larger data sets will take more time.
*/

public class InsertionSort
{
```

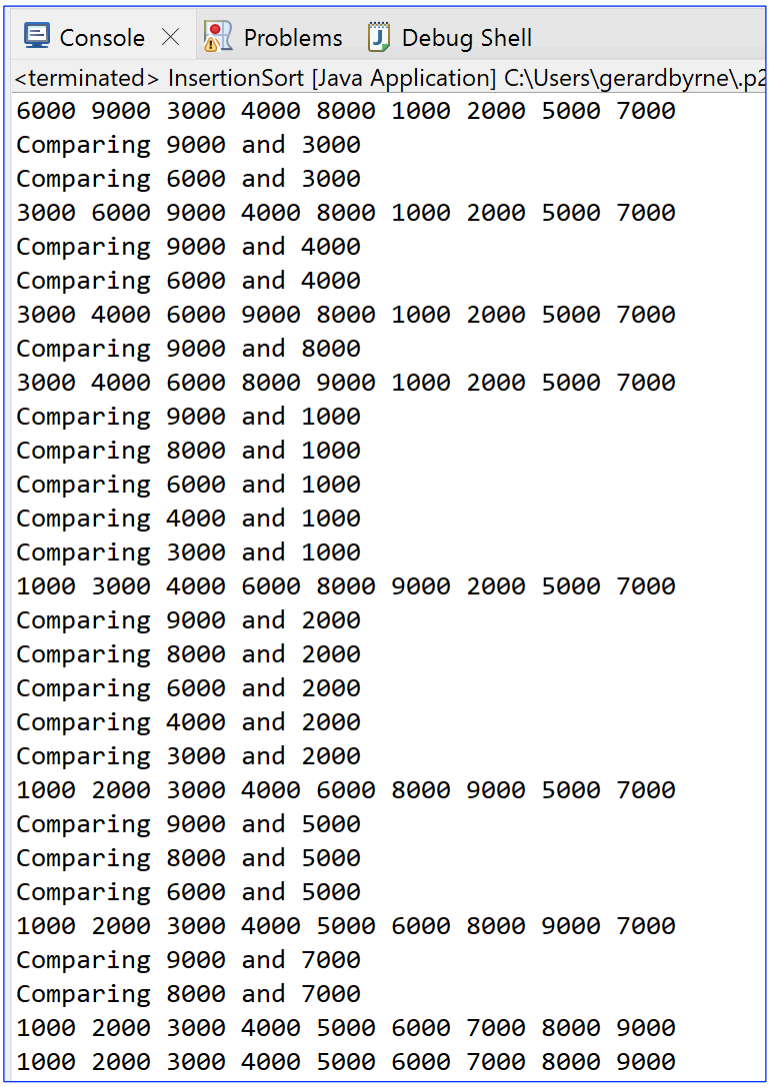
Original values

6000 9000 3000 4000 8000 1000 2000 5000 7000

Resulting iterations – highlighted values are the ones being compared

6000	9000	3000	4000	8000	1000	2000	5000	7000
6000	9000	3000	4000	8000	1000	2000	5000	7000
6000	3000	9000	4000	8000	1000	2000	5000	7000
3000	6000	9000	4000	8000	1000	2000	5000	7000
3000	6000	4000	9000	8000	1000	2000	5000	7000
3000	4000	6000	9000	8000	1000	2000	5000	7000
3000	4000	6000	8000	9000	1000	2000	5000	7000
3000	4000	6000	8000	1000	9000	2000	5000	7000
3000	4000	6000	1000	8000	9000	2000	5000	7000
3000	4000	1000	6000	8000	9000	2000	5000	7000
3000	1000	4000	6000	8000	9000	2000	5000	7000
1000	3000	4000	6000	8000	9000	2000	5000	7000
1000	3000	4000	6000	8000	2000	9000	5000	7000
1000	3000	4000	6000	2000	8000	9000	5000	7000
1000	3000	4000	2000	6000	8000	9000	5000	7000
1000	3000	2000	4000	6000	8000	9000	5000	7000
1000	2000	3000	4000	6000	8000	9000	5000	7000
1000	2000	3000	4000	6000	8000	5000	9000	7000
1000	2000	3000	4000	6000	5000	8000	9000	7000
1000	2000	3000	4000	5000	6000	8000	9000	7000
1000	2000	3000	4000	5000	6000	8000	7000	9000
1000	2000	3000	4000	5000	6000	7000	8000	9000

Output


Console
Problems
Debug Shell
<terminated> InsertionSort [Java Application] C:\Users\gerardbyrne\.p2
6000 9000 3000 4000 8000 1000 2000 5000 7000
Comparing 9000 and 3000
Comparing 6000 and 3000
3000 6000 9000 4000 8000 1000 2000 5000 7000
Comparing 9000 and 4000
Comparing 6000 and 4000
3000 4000 6000 9000 8000 1000 2000 5000 7000
Comparing 9000 and 8000
3000 4000 6000 8000 9000 1000 2000 5000 7000
Comparing 9000 and 1000
Comparing 8000 and 1000
Comparing 6000 and 1000
Comparing 4000 and 1000
Comparing 3000 and 1000
1000 3000 4000 6000 8000 9000 2000 5000 7000
Comparing 9000 and 2000
Comparing 8000 and 2000
Comparing 6000 and 2000
Comparing 4000 and 2000
Comparing 3000 and 2000
1000 2000 3000 4000 6000 8000 9000 5000 7000
Comparing 9000 and 5000
Comparing 8000 and 5000
Comparing 6000 and 5000
1000 2000 3000 4000 5000 6000 8000 9000 7000
Comparing 9000 and 7000
Comparing 8000 and 7000
1000 2000 3000 4000 5000 6000 7000 8000 9000
1000 2000 3000 4000 5000 6000 7000 8000 9000

Program Practically

Java

Labs

It's time to reinforce our learning

Gerry Byrne

Program Practically - Module Labs

This is our opportunity to practise what we have learnt. We should complete the labs by referring to the book modules when we are unsure about how to do something, but more importantly we should look at the previous code we have written.

The code we have written should be an invaluable source of working code and it is important not to 'reinvent the wheel', use the code, copy, paste and amend it if required. Reuse the code, that is what the professional developer would do and is expected to do. Professional software developers are expected to create applications as fast and accurately as possible and reusing existing code is one technique they apply, so, why should we be any different.

If we really get stuck there are sample solutions following the labs and we can refer to these, but it is important we understand any code that we copy and paste.

It is important we enjoy the challenge of developing solutions for each lab. We will apply the things we have learnt in the modules but more importantly we will develop our own techniques and styles for coding, debugging and problem solving.

Think about the saying:

"Life begins at the edge of our comfort zone"

We will inevitably feel at the edge of our programming ability but every new thing we learn in completing each lab should make us feel better and encourage us to learn more. Whilst we may be 'frightened' and 'uncomfortable' completing the coding labs the process will lead us to grow and develop our coding skills. We might find it 'painful' at times but that is the reality of programming. We will find it exciting and challenging as we are stretched and brought to a place we have not been to before.

Module 2 Labs – Println()

Lab One

Write a Java console application, using the println() command, that will display the letter E using * to form the shape e.g. one line could be System.out.println("*******");

Lab Two

Write a Java console application, using the println() command, that will display the letter A using * to form the shape e.g. one line could be System.out.println(" *");

Lab Three

Write a Java console application that will display your name and address in a format that might look like a label for an envelope.

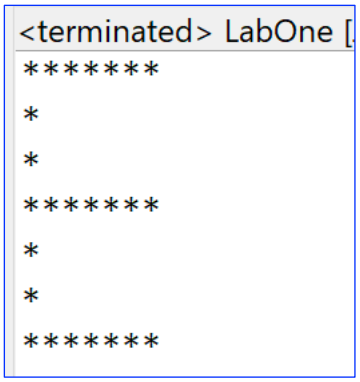
Lab Four

Using the same code that you developed for Lab Three, the name and address label, add a statement between each of the name and address lines that will require the user to press a key on the keyboard before the display moves to the next line.

Module 2 Labs – Possible solutions

Lab One

```
package labs.module02;
/*
 Write a Java console application, using the println()
 command, that will display the letter E using * to form
 the shape e.g. one line could be System.out.println("*******");
*/
public class LabOne
{
    public static void main(String[] args)
    {
            System.out.println("********");
            System.out.println("*");
            System.out.println("*");
            System.out.println("********");
            System.out.println("*");
            System.out.println("*");
            System.out.println("********");
    } // This is the end of the method
} // This is the end of the class
```



Lab Two

```
package labs.module02;
/*
 Write a Java console application, using the println()
 command, that will display the letter A using * to form
 the shape e.g. one line could be System.out.println("      *");
*/
public class LabTwo
{
    public static void main(String[] args)
    {
      System.out.println("      *");
      System.out.println("     * *");
      System.out.println("    *   *");
      System.out.println("   *******");
      System.out.println("  *       *");
      System.out.println(" *         *");
      System.out.println("*           *");
    } // This is the end of the method
} // This is the end of the class
```