

Java and Object-Oriented Programming

Course Notes for **CIS 1222**



These notes belong to

Name: _____

Mobile: _____

Prepared and compiled by

Matthew Xuereb

Contact details

Name: Matthew Xuereb

Email: matthew.xuereb@um.edu.mt

URL: www.matthewxuereb.com

Copyright

- These notes were prepared and compiled by Matthew Xuereb B.Sc. I.T. (Hons.), M.Sc, M.B.A, iPGCE.

Preface

Various books and websites were used for the compilation of these notes. The readers of these notes are encouraged to use other books and the web to find more Java and OOP examples.

Table of contents

PART 1	5
Programming Basics.....	5
The Java Programming Language	6
Programming Basics.....	7
Constants and variables.....	14
Getting input from console.....	16
Conditional statements.....	17
Looping.....	20
PART 2	22
Object Oriented Programming	22
Object Oriented Programming	23
Attributes	24
Methods.....	24
Main methods.....	26
Methods with parameters	27
Methods with return values	27
Encapsulation.....	28
Constructors.....	31
Inheritance.....	33
Method overriding.....	34
Constructors in inherited classes.....	35
Polymorphism (Part 1)	36
Arrays	38
ArrayLists.....	42
Polymorphism (Part 2)	43
PART 3	44
Important Programming Features	44
String handling	45
Exception handling.....	48
File handling.....	51
The Java Math Class	53
PART 4	54

User Interface Programming.....	54
GUI Programming	55
GUI Layout Managers	61
Listeners and Events – The ActionListener	66
Graphics	67
Listeners and Events – The MouseListener and MouseMotionListener	68
APPENDIX 1: for loops demos.....	69
APPENDIX 2: Arrays.....	71
APPENDIX 3: String Handling	74
Appendix 4: Helpers Class.....	75
APPENDIX 5: Inheritance & Polymorphism.....	79
APPENDIX 6: GUI using Swing.....	83
APPENDIX 7: GUIHelpers.....	92
APPENDIX 8: UML	93

PART 1

Programming

Basics

The Java Programming Language

The Java programming language was designed by James Arthur Gosling for the Sun Microsystems company. The latter is now owned by the Oracle Corporation. This programming language has been in use since 1995 and is currently one of the most popular programming languages, especially in the field of client-server web based applications.

Java comes in four different platforms:

1. Java SE (Standard Edition)
2. Java EE (Enterprise Edition)
3. Java ME (Micro Edition)
4. Java Card

In this course the Java SE (Standard Edition) is going to be covered. Java SE is the most important Java platform because it is the basis of all the other platforms. Java SE is used for the implementation of desktop based applications that is, applications that can run on traditional desktop/laptop computers.

A very popular feature of the Java Programming Language is its portability. Unlike many other programming languages, a program that is written in Java can be executed on computers having different Operating Systems and platforms. For example a program that is implemented on a computer with a Windows Operating System can be later on executed on computers with a Mac or Linux Operating System without the need of any re-compilation. For this reason, Java is referred to as a *WORA* Language: Write Once, Run An Timer anywhere.

The programming process and required software and tools

In order to implement a program, the procedure as depicted in Figure 1 is carried out. First the programmer writes the program's code (technically referred to as the *source code*). The code is then given to a programming language translator (usually referred to as the *compiler* or the *interpreter*) and if it is written correctly it will be translated to either an executable program or else to a running program. If the code is not correct then the programming language translator will output a list of possible errors that the source code might have.

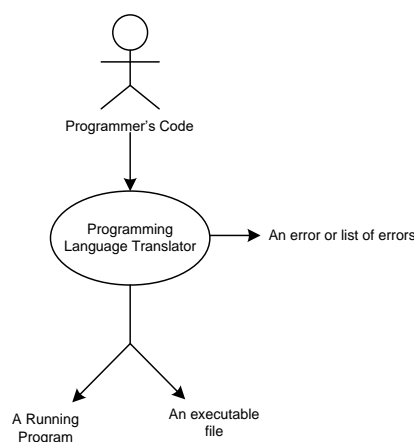


Figure 1

The Programming Language Translator for Java is called the Java Development Kit (JDK) and can be downloaded from the Oracle Corporation Website. In order to implement Java

programs, basic text editor programs such as *Notepad*, *TextPad* and *TextEdit* can be used. However it is suggested to avoid the use of such basic programs and instead use more user friendly programming tools (technically referred to as IDEs: *Integrated Software Environments*) for Java such as NetBeans, eclipse and BlueJ. The reason for this suggestion is that usually such Java IDEs include aides and tools that make the programmer's life easier by for instance highlighting any syntax errors. In these lessons, the NetBeans IDE is going to be used all the way through.

Installing the required software

Two different programs are required in order to program with Java. These are the Java Virtual Machine and an Integrated Development Environment (IDE) such as NetBeans. Since the NetBeans IDE is owned by Oracle, that is the same company that owns Java, a bundle that includes both programs is offered by Oracle for free download from their website.

To install this bundle, follow the following steps:

1. Browse to the following website or otherwise Google search the phrase 'NetBeans Bundle'.
<http://www.oracle.com/technetwork/articles/javase/jdk-netbeans-jsp-142931.html>
2. Accept the Licence Agreement and select your type of operating system.
3. Once the installation file is completely downloaded, run it to install the bundle. Note that the installation might take a few minutes.

Programming Basics

The HelloWorld Program

When learning a new programming language, the first program that is written is the famed *HelloWorld* program. This is a very simple program that all it does is displays the phrase "Hello

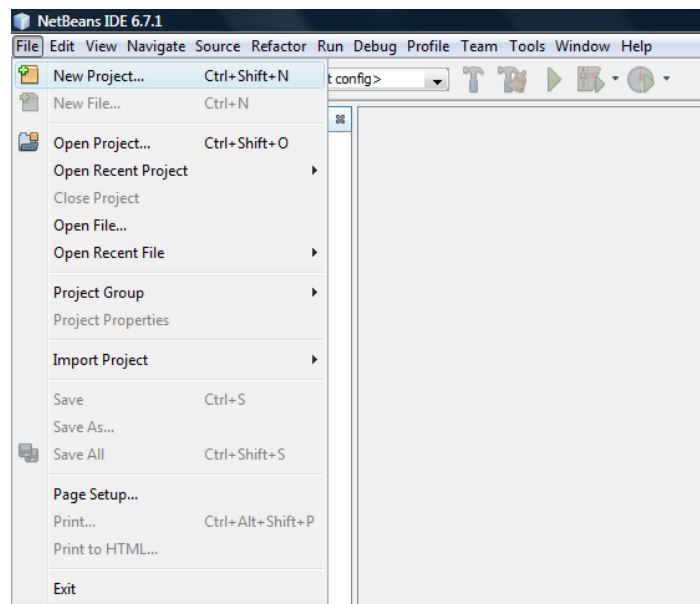
World!!" on the screen. Nevertheless this program is very important because with it one can confirm that his/her programming environment is working well.

The following is the source code for the *HelloWorld* program:

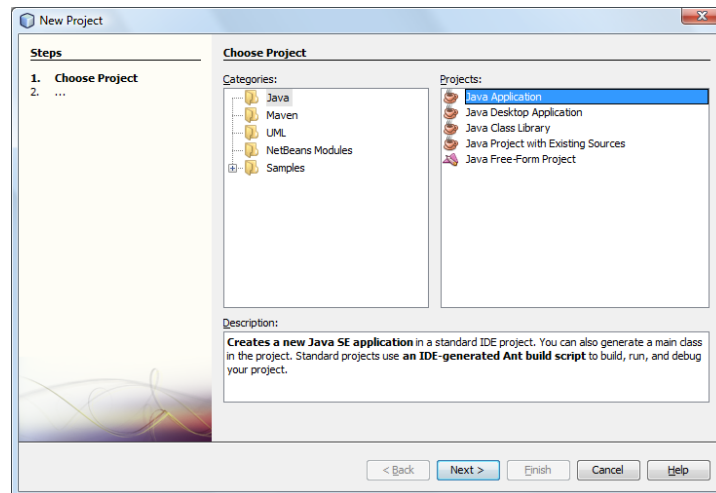
```
public class Main{  
    public static void main(String[] args){  
        System.out.println("Hello World!!");  
    }  
}
```

The code will be explained later on. The most important thing at this stage is to create and be able to successfully run this code. The following procedure should be carried out in order to write and run the *HelloWorld* program in NetBeans:

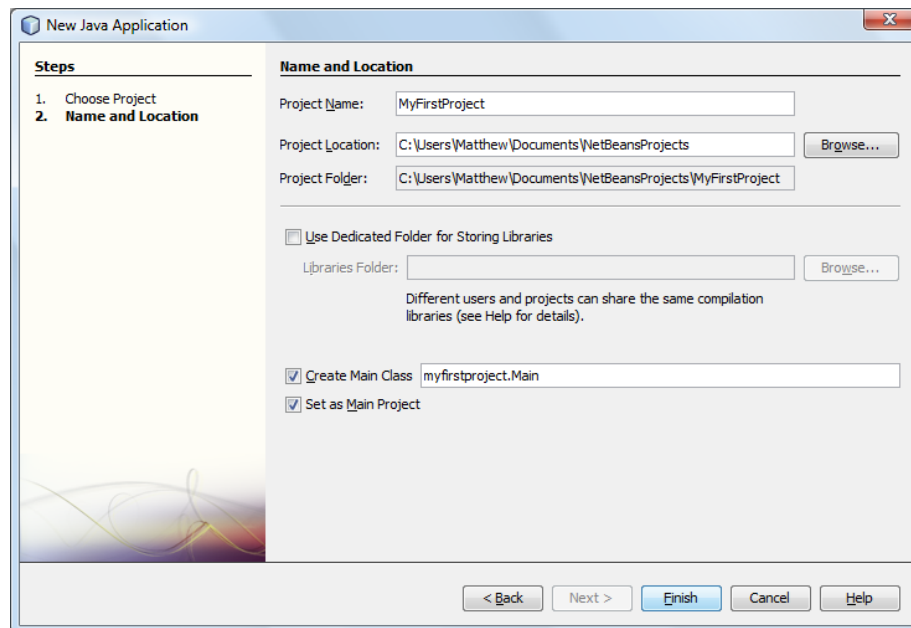
1. Open the NetBeans IDE and select File -> New Project... from the drop down menu as shown below:



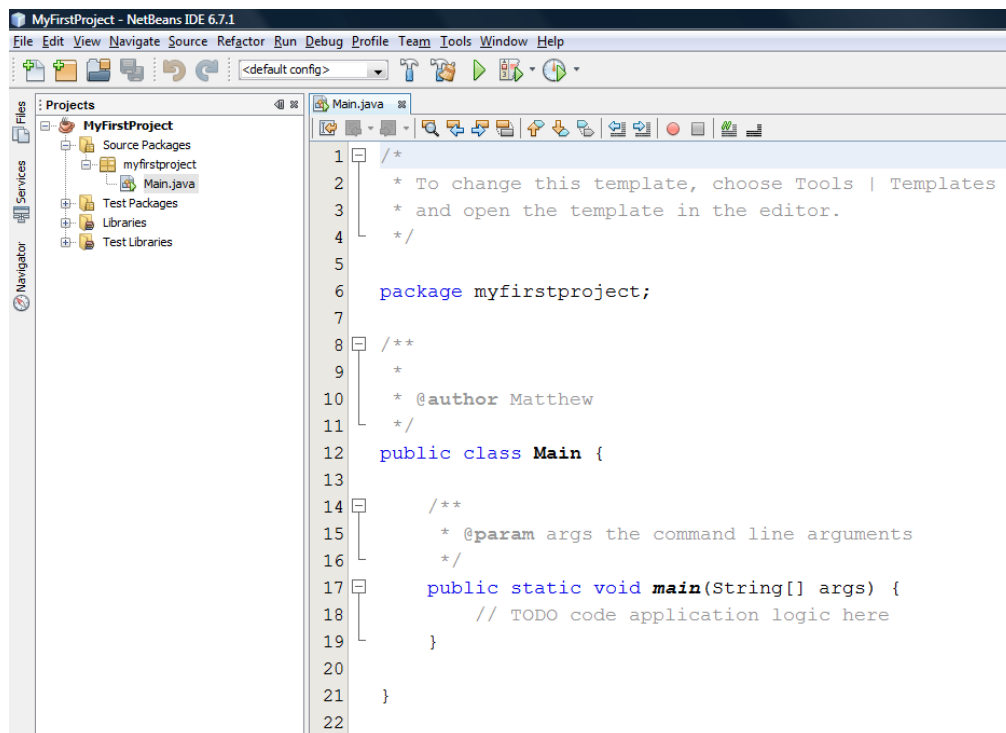
2. A *New Project* dialog box is displayed on the screen. Select *Java Application* from the *Java* Category and click on the *Next* button as shown in the screen shot below:



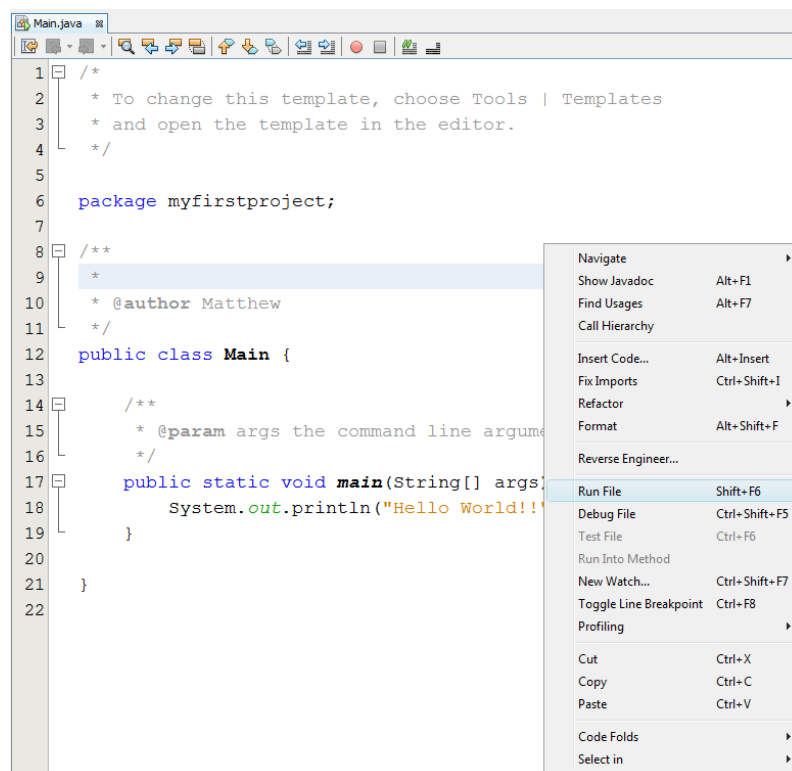
3. A *New Java Application* dialog box is displayed on the screen. Type the name of your project (e.g. *MyFirstProject*) in the *Project Name* text field and then click the *Finish* button.



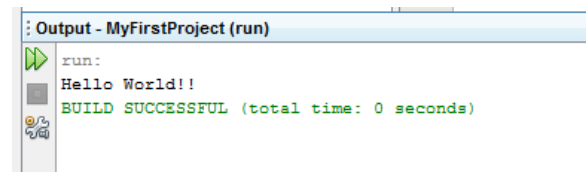
4. When the *Finish* button is clicked, the development environment is set up as shown in the screen shot below.



5. Type (or fill in) the source code and execute it by right clicking on the text editor panel and select the *Run File* option as shown in the screen shot below:



6. The program is now executed and the phrase *Hello World!!* is displayed at the bottom of the screen on the OutputWindow as shown in the screen shot below:



The HelloWorld program explained

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

Being an Object Oriented Programming Language, everything in Java has to be implemented in a Class. The meaning of Object Oriented Programming, Classes and other Object Oriented related things will be explained later on. However it is good to note that our first program was implemented in a class called Main. Java source code is stored in files that have the same name as the class that they are representing. If the name of the class is different from that of the file, then the code will give an error and will therefore not compile.

Java is a case sensitive language. This means that ‘class’ and ‘Class’ have a different meanings. If keywords or other phrases are not written according to the correct case, the program will not compile and instead a syntax error will be generated. The phrase ‘public static void main(String[] args)’ represents a main method. In Java a main method is the entry point of the program. In other words, the code that is written within the corresponding braces ({..} – known technically as a scope) following the main method signature is going to be invoked (executed) immediately. The meaning of the keywords public, static and void will also be explained in later on.

The phrase ‘System.out.println(“Hello World!!”);’ is used to output the text “Hello World!!” on the screen. Note the semicolon (;) at the end of the statement. In Java, all statements end with a semicolon.

Comments

When programming, sometimes it is required to write some notes or comments within the source code. In Java there are two types of comments, namely inline comments and multiline comments. Inline comments starts with a // and whatever is written following the // up till the end of the line is treated as a comment. A multiline comment starts with a /* and ends with a */. The contents written between these two symbols respectively, is treated as a comment. Note that all comments as ignored by the Java compiler and therefore will not affect the final program outcome.

Displaying text on the screen

The *System.out.println()* or *System.out.print()* are two popular statements that are used to output text in a Java program. The difference between these two methods is that the *System.out.println()* displays a phrase on the screen and then places the cursor in the next line while the *System.out.print()* method leaves the cursor on the same line. The following table illustrates some examples to explain this difference:

Source Code	Output
<code>System.out.println("Hello");</code> <code>System.out.println("My name is Matthew");</code> <code>System.out.println("What is your name?");</code>	Hello My name is Matthew What is your name?
<code>System.out.print("Hello");</code> <code>System.out.println("My name is Matthew");</code> <code>System.out.print("What is your name?");</code>	HelloMy name is Matthew What is your name?
<code>System.out.print("Hello");</code> <code>System.out.print("My name is Matthew");</code> <code>System.out.print("What is your name?");</code>	HelloMy name is MatthewWhat is your name?

Escape Sequences

In Java, the *backslash* (\) is called an *escape character*. It indicates that a “special character” is to be outputted. When a backslash appears in a string of characters, Java combines the next character or characters with the backslash to form an *escape sequence*. For example, the escape sequence `\n` displays a new line. When a newline character appears in a string being output with **System.out**, the newline character causes the screen’s output cursor to move to the beginning of the next line in the command window. Some common escape sequences are listed in the table below:

Escape sequence	Meaning
<code>\b</code>	<i>Backspace</i>
<code>\n</code>	<i>Newline</i>
<code>\t</code>	<i>Tab</i>
<code>\r</code>	<i>Carriage return</i>
<code>\\</code>	<i>Backslash</i>
<code>\"</code>	<i>Double quote</i>
<code>\u</code>	<i>Unicode Character</i>

The `\u` (Unicode character) should be followed by the Unicode number of the special character to be outputted. A complete table of the Unicode values can be accessed from <http://unicode-table.com/en/#0121>. The table below illustrates the Unicode for the Maltese characters.

Special Character	Unicode
Ċ	010A
ċ	010B
Ġ	0120
ġ	0121
Ħ	0126
ħ	0127
Ż	017B
ż	017C

The following table illustrates some examples to of print statements with escape characters:

Source Code	Output
<code>System.out.print("Hello\nMy name is Matthew");</code>	Hello My name is Matthew
<code>System.out.print("Joe has a height of 6\'4\'");</code>	Joe has a height of 6'4"
<code>System.out.print("Name\tSurname");</code>	Name Surname
<code>System.out.println("I\u017c-\u017Bebbu\u0121, G\u0127awdex");</code>	Iż-Żebbuġ, Ġhawdex

Constants and variables

A variable is the name for a place in the computer's memory where you store some data. In Java, a variable declaration includes the following:

- A data type that identifies the type of data that the variable will store.
- An identifier that is the variable's name.
- An optional assigned value, when you want a variable to contain an initial value (also known as *declare and assign*).
- An ending semicolon.

A constant is a meaningful name that represents a number, or string, that does not change. Constants are variables that remain the same, i.e. constant, throughout the execution of a program. In Java a constant is declared exactly as a variable however the keyword ***final*** is written before the data type.

Primitive data types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name. There are eight primitive data types and these are:

Data Type	Values
boolean	<i>true</i> or <i>false</i>
byte	-128..127 [8-bit signed]
short	-32,768..32,767 [16-bit signed]
int	-2,147,483,648..2,147,483,647 [32-bit signed]
long	-9,223,372,036,854,775,808..9,223,372,036,854,775,807 [64-bit signed]
float	1.4 E-45..3.4028235 E+38 [32-bit signed]
double	4.9 E-324..1.7976931348623157 E+308 [64-bit signed]
char	16-bit Unicode character

In Java, a string is not a primitive data type however Java has a special support for strings and many times they can be used just as primitive data types.

Example of how to declare variables in Java:

```
boolean flag = true;
int x, y , z;
double d = 4.346;
```

```
char c = 'x';  
String s = "Hello";
```

Example of how to declare constants in Java:

```
final double PI = 3.142;  
final String MESSAGE = "Hello!!";
```

Identifiers

- Must be composed of letters, numbers, the underscore (_) and the dollar sign (\$)
- May only begin with a letter, the underscore or a dollar sign
- Reserved words cannot be used as identifiers

Assignment statements

The equal sign (=) is used as the assignment statement as shown in the example below:

```
int x = 65;  
y = x - 9 + (x * 4); // Assuming that y is an integer  
String s = "Hello" + " World!!";
```

Arithmetic operators

Arithmetic operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	$F + 7$
Subtraction	-	$p - c$	$p - c$
Multiplication	*	$b \times m$ OR bm	$b * m$
Division	/	$x \div y$ OR x/y	x / y
Remainder	%	$r \bmod s$	$r \% s$

Rules of operator precedence:

- *, / and % operators are applied first. If an expression contains several such operations, the operators are applied from left to right.
- + and – operations are applied next. If an expression contains several such operations, the operators are applied from left to right.

Getting input from console

In order to be able to get input from console the class `Scanner` is used. An instance of this class is initiated since his class enables a program to read data for use in a program. The data can come from many sources such as file, disk or the user at the keyboard. The following examples illustrate how input from the keyboard can be entered by the user intervention.

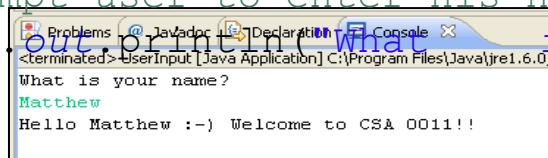
Syntax:

```
Scanner input = new Scanner(System.in);
int num1 = input.nextInt(); // read an integer from user
String str = input.nextLine(); // read a string from user
```

Note that in order to use this class, it should be imported from the *java.util* library. The following example illustrates an example program that asks the user to enter his name and then displays a welcome message on the screen.

```
import java.util.Scanner;
public class UserInput {

    public static void main(String[] args) {
        // create Scanner to get user input from
        console
        Scanner input = new Scanner(System.in);
        // Prompt user to enter his name
        System.out.println("What is your
        name?") :
```



The following `Scanner` class methods can be used in order to get various input types from the keyboard:

Method	Returns
<code>int nextInt()</code>	Returns the next token as an int. If the next token is not an integer, <code>InputMismatchException</code> is thrown.
<code>long nextLong()</code>	Returns the next token as a long. If the next token is not an integer, <code>InputMismatchException</code> is thrown.

float nextFloat()	Returns the next token as a float. If the next token is not a float or is out of range, InputMismatchException is thrown.
double nextDouble()	Returns the next token as a long. If the next token is not a float or is out of range, InputMismatchException is thrown.
String next()	Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If not token exists, NoSuchElementException is thrown.
String nextLine()	Returns the rest of the current line, excluding any line separator at the end.
void close()	Closes the scanner.

Conditional statements

In programming, the term conditional statements refer to programming instruction that has to do with decision making. Making a decision involves choosing between alternate courses of action based in some value within a program. The final value that the decision is based in is always boolean, i.e. **true** or **false**.

In Java an **if statement** structure or a **switch statement** structure are used to perform a decision making statement. The following is a table with the equality operators used by Java for decision making:

Algebraic equality or relational operator	Java equality or relational operator	Java example	Meaning
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
<i>Relational operators</i>			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Making decisions with the *if* and *if..else* structures

The **if** statement has this form, where *condition* is true or false.

```
if (condition) {
    ... // Do this clause if the condition is true.
}
```

----- OR -----

```
if (condition) {
    ... // Do this clause if the condition is true
} else {
    ... // Do this clause if the condition is false
}
```

For example, the following program, after asking the user to enter his age, checks whether the user can drive a car or not. The decision is based on the age of the user. If the user is 18 years old or older, then he can drive a car, otherwise he cannot drive a car.

```
public static void main(String[] args) {
    // create Scanner to get user input from console
    Scanner input = new Scanner(System.in);
    // Prompt user to enter his age
    System.out.println("What is your age?");
    int age = input.nextInt();

    if(age >= 18){
        System.out.println("You can drive a car!");
    }else{
        System.out.println("You cannot drive a car!");
    }
}
```

It is common to make a series of tests on a value, where the `else` part contains only another `if` statement as shown in the example below:

```
if (age >= 26) {
    System.out.println("You can drive a car, a truck and a bus");
} else if (score >= 22) {
    System.out.println("You can drive a car and a truck");
} else if (score >= 18) {
    System.out.println("You can drive a car");
} else {
    System.out.println("You cannot drive");
}
```

Making decisions with *switch* structure

The `if` statement allows you to select one of two sections of code to execute based on a boolean value (only two possible values). The `switch` statement allows you to choose from many statements based only on integers (including `char` and `enum` values). The syntax of a `switch` statement is as follows:

```
switch (expr) {
    case c1:
        statements // do these if expr == c1
        break;
    case c2:
        statements // do these if expr == c2
        break;
    case c2:
    case c3:
    case c4:           // Cases can simply fall thru.
        statements // do these if expr == any of c's
        break;
    . . .
    default:
        statements // do these if expr != any above
}
```

The `switch` keyword is followed by a parenthesized integer expression, which is followed by the `cases`, all enclosed in braces. The `switch` statement executes the case corresponding to the value of the expression. Normally the code in a case clause ends with a `break` statement,

which exits the switch statement and continues with the statement following the switch. If there is no corresponding case value, the `default` clause is executed. If no case matched and there is no default clause, execution continues after the end of the switch statement.

The `case` keyword is followed by an integer constant and a colon. This begins the statements that are executed when the switch expression has that case value.

If no case value matches the switch expression value, execution continues at the default clause. This is the equivalent of the *else* for the switch statement. It is written after the last case by convention, and typically isn't followed by `break` because execution just continues out the bottom of switch if this is the last clause.

The `break` statement causes execution to exit to the statement after the end of the switch. If there is no `break`, execution flows thru into the next case. Flowing directly into the next case is almost always an error.

Example of a switch statement:

```
switch (month) {
    case 1:  System.out.println("January");
            break;
    case 2:  System.out.println("February");
            break;
    ...
    ...
    case 12: System.out.println("December");
            break;
    default: System.out.println("Invalid month number");
}
```

Looping

A loop is a sequence of statements which is specified once but which may be carried out several times in succession. There are three main kinds of loop statements in Java. These are:

1. **for** loop
2. **while** loop
3. **do..while** loop

A **for** loop is a loop that repeats a group of statements for a defined number of times. Thus it can be defined as an iteration statement with a counter.

```
// A for loop to display the numbers 1-10
for(int i = 1; i <= 10; i++){
    System.out.println(i);
}
```

A **while** loop is a control flow statement that allows code to be executed repeatedly based on a given boolean condition. The condition is first evaluated - if the condition is true the code within the block is then executed.

```
Scanner input = new Scanner(System.in);

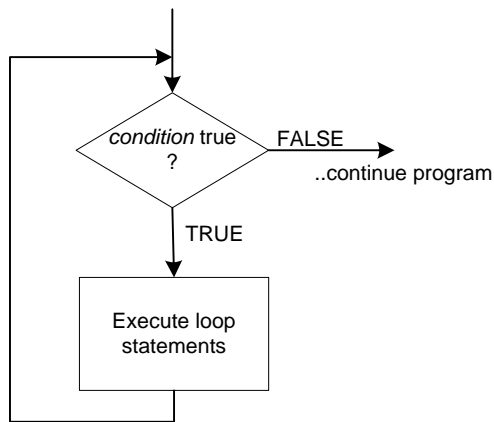
System.out.println("Enter an integer less than 100");
int num = input.nextInt();
while(num >= 100){
    System.out.println("Enter an integer less than 100");
    num = input.nextInt();
}
```

A **do..while** loop is a control flow statement that allows code to be executed repeatedly based on a given boolean condition. First, the code within the block is executed, and then the condition is evaluated. If the condition is true the code within the block is executed again.

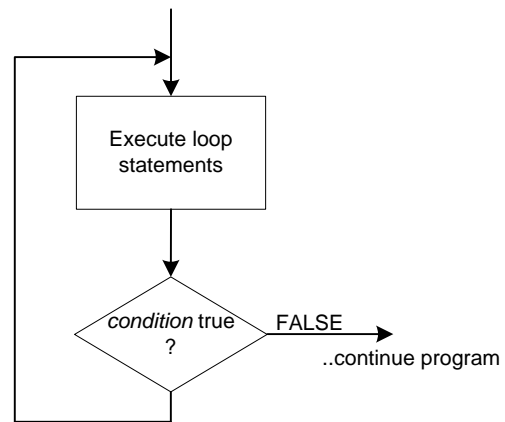
```
Scanner input = new Scanner(System.in);
int num;
do{
    System.out.println("Enter an integer less than 100");
    num = input.nextInt();
}while(num >= 100);
```

Comparison between the *while* loop and the *do..while* loop

The only difference between the *while* loop and the *do..while* loop is that the latter has the stopping condition at the end of the loop and therefore at least the looping content is executed once. This is graphically explained in the following two flow charts:



***while* loop** flow chart



***do..while* loop** flow chart

PART 2

Object Oriented Programming

Object Oriented Programming

Object oriented Programming is a type of programming in which programmers define not only the data type of a data structure, but also the operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

One of the principal advantages of object oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object oriented programs easier to modify and maintain.

Java is a native object oriented language, and one can say that in Java everything is defined as a class. Three fundamental principles of object oriented programming that will be covered in this section are:

- Encapsulation
- Inheritance
- Polymorphism

Classes and objects

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. On the other hand, a class is a blueprint or prototype from which objects are created. In short, **an object is an instance of a class**.

Declaring a class

A class is similar to a *record* in Pascal or a *struct* in the C programming language, in that it stores related data fields, where the data fields can be of different types. So you could for example store a test string in one field, and an integer in another field and a floating point in a third field. The difference between a class and a record (or *struct*) is that a class also defines methods to work on the data fields.

A Java class has the following structure:

```
public class <class_identifier>{  
    ...attributes...  
    ...constructors...  
    ...methods...  
}
```

The *class_identifier* represents the name of the class. It is very important to note that the class identifier has to be the same name that is given to the *.java* file that stores in it the class.

The body of the class consists of three types of elements:

- Variables (data fields or attributes)
- Constructors
- Methods

Note that these elements can be declared in any order. However in order to keep the code as readable as possible it is advised to keep the order as shown above, i.e. first the variables, then the constructors and then the methods. All these elements are optional, that is a class body does not need to have any variables or constructors or methods at all.

The following example illustrates a class that represents a Person:

```
public class Person{  
  
}
```

Attributes

Attributes are variables therefore each attribute has an identifier and an associated data type. Every time an object is created, memory is allocated for each attribute of the object. Attributes are defined in a class in a similar way that we define variables, with one difference – an attribute can have an access modifier. The example below defines some attributes for the class Person:

```
public class Person{  
  
    // Attributes  
    public String name;  
    public String surname;  
    public int age;  
  
}
```

Note the *public* access modifier in the above attributes declarations. In Java there are four types of access modifiers (*default*, *public*, *private* and *protected*). The access modifiers will be explained later. However note that if no access modifier is included with an attribute as shown in same example below, Java treats them as if they were defined as *default*.

```
public class Person{  
    // Attributes  
    String name;  
    String surname;  
    int age;  
  
}
```

Methods

Methods are the object-oriented language term for procedures, functions or subroutines in other types of programming languages. Methods are containers of a group of Java programming statements that perform a specific task and that can be called (invoked) from other parts of the program.

Each method has a name, a return data type, zero or more parameters and a body. The method body contains the statements that evaluate the input parameters, if any, and produces the return value. Methods that do not require a return value specify the **void** keyword instead of a return data type.

In the following example, a method called *displayMessage* is added to the class *Person*. This method is used to display a hello message on the screen.

```
public class Person{
    // Attributes
    public String name;
    public String surname;
    public int age;

    // A method used to display a hello message
    public void displayMessage(){
        System.out.println("Hello Person");
    }
}
```

As in the attribute declaration, the method declaration begins with the access modifier. In the above example it is *public*. Note that if the access modifier is omitted by default it is taken as *public*. After the access modifier, the return type is written. In the above example it is *void*, i.e. this method is not going to return anything. Following the return data type there is the method identifier (name) followed by a (). The () is an empty set of parameters. This means that this method does not take any parameters.

The body of the method contains statements that perform the method's task. In this case this method prints the name, surname and age of the person object on the screen. In the following example, another method *displayDetails* is added. This method is used to print the details of the person object on the screen.

```
public class Person{
    // Attributes
    public String name;
    public String surname;
    public int age;

    // A method used to display a hello message
    public void displayMessage(){
        System.out.println("Hello Person");
    }

    // A method used to display the details of the person
    public void displayDetails(){
        System.out.println("NAME      - " + name);
        System.out.println("SURNAME  - " + surname);
        System.out.println("AGE      - " + age);
    }
}
```

Main methods

A class can have many methods, and a program can be composed on many classes. However in a program there can only be one *main* method. The *main* method is the entry point of the program – it is the place from where the program starts its execution. In Java every program execution starts from the *main* method. The *main* method can be either written in a separate class or else in a class such as the person class written in this example.

The following code snippet illustrates the class *Person* and another class *Test* with a main method that is instantiating an object of type *Person*, setting the person's fields and invoking the display details method:

```
public class Person{
    // Attributes
    public String name;
    public String surname;
    public int age;

    // A method used to display a hello message
    public void displayMessage(){
        System.out.println("Hello Person");
    }

    // A method used to display the details of the person
    public void displayDetails(){
        System.out.println("NAME      - " + name);
        System.out.println("SURNAME - " + surname);
        System.out.println("AGE       - " + age);
    }
}

~~~~~

public class Test{
    public static void main(String[] args){
        Person p1 = new Person(); // Creating an object of Person
        p1.name = "Joe";
        p1.surname = "Borg";
        p1.age = 43;
        p1.displayDetails();
    }
}
```

The following code snippet illustrates the same example as above however the main method is part of class *Person*:

```
public class Person{
    // Attributes
    public String name;
    public String surname;
    public int age;

    // A method used to display a hello message
    public void displayMessage(){
        System.out.println("Hello Person");
    }

    // A method used to display the details of the person
    public void displayDetails(){
        System.out.println("NAME      - " + name);
        System.out.println("SURNAME   - " + surname);
        System.out.println("AGE       - " + age);
    }

    public static void main(String[] args){
        Person p1 = new Person(); // Creating an object of Person
        p1.name = "Matthew";
        p1.surname = "Xuereb";
        p1.age = 27;
        p1.displayDetails();
    }
}
```

Methods with parameters

As outlined above, a method can have parameters. Parameters allow a method to accept values that can be used by that method. The following code snippet illustrates a method named *displayGreeting()* that accepts a parameter *greeting* and makes use of it as shown in the code snippet below.

```
public void displayGreeting(String greeting){
    System.out.println(greeting + " " + name + "!");
}
```

Note that in the above method, the class attribute *name* is also used to display the name of the person together with the greeting message that is passed as a parameter as shown in the example below:

Code that is calling method	Output
Person p = new Person(); p.name = "Matthew"; p.displayGreeting("Good Morning");	Good Morning Matthew!

Methods with return values

In the examples above, all the methods do not return any values, i.e. the return type is always set as *void*. In Java a method can return a value. In order to define a method that returns a value, the data type of the value to be returned by the method should be declared within the method signature instead of the *void* keyword. The *return* keyword is then used to instruct the method to return the actual value. The following code snippet illustrates a method that is used to return the name and surname as a single string:

```
public String getNameAndSurname(){
    String ans = name + " " + surname;
    return ans;
}
```

Note that the return data type can be either a primitive data type (such as *int*, *boolean*, ...) or a Java pre-defined object (such as *String*, *JButton*, ...) or else a programmer's defined object (such as *Person*).

Encapsulation

Encapsulation is the ability of an object to be a container for related properties and methods. Older languages did not enforce any property/method relationships. This often resulted in side effects where variables had their contents changed or reused in unexpected ways and spaghetti code that was difficult to unravel, understand and maintain.

Encapsulation is one of three fundamental principles in object oriented programming. Data hiding is the ability of objects to shield variables from external access. It is a useful consequence of the encapsulation principle. Those variables marked as *private* can only be seen or modified through the use of public *accessor* and *mutator* methods (*getters* and *setters* methods). This permits validity checking at run time. Access to other variables can be allowed but with tight control on how it is done. Methods can also be completely hidden from external use. Those that are made visible externally can only be called by using the object's front door.

Basically, when using the encapsulation technique, all the properties (data fields) are declared as **private** (and in some cases when there is inheritance as **protected**). Since these data fields are set to be private, they cannot be accessed for outside their class by invoking the using the dot notation (such as *p1.name*). Instead, getters and setters methods are used.

Getters (Accessors)

Getters are methods that are used to return a particular variable. For instance, suppose we have a class *Person* and this class consists of three data fields *name*, *surname* and *age* that are set as private as shown below.

```
public class Person{

    // Attributes
    private String name;
    private String surname;
    private int age;

}
```

Since the data fields are set as private their value cannot be accessed from outside that class. In order to be able to get the value of such data fields, getter methods are written as shown in the code snippet below. Note the naming convention used for the getter methods – every getter method starts with the word *get* followed by the data field identifier starting with an uppercase letter.

```
public class Person{

    // Attributes
    private String name;
    private String surname;
    private int age;

    public String getName(){
        return name;
    }

    public String getSurname(){
        return surname;
    }

    public int getAge(){
        return age;
    }
}
```

Setters (Mutators)

Setters are methods used to set a particular value to a particular data field. Consider the above class *Person* example, the example below continues to build on the previous one, this time adding also the setter methods.

```
public class Person{

    // Attributes
    private String name;
    private String surname;
    private int age;

    // Setters
    public void setName(String name){
        this.name = name;
    }

    public void setSurname(String s){
        surname = s;
    }

    public void setAge(int age){
        this.age = age;
    }

    // Getters
    public String getName(){
        return name;
    }

    public String getSurname(){
        return surname;
    }

    public int getAge(){
        return age;
    }
}
```

The *this* keyword

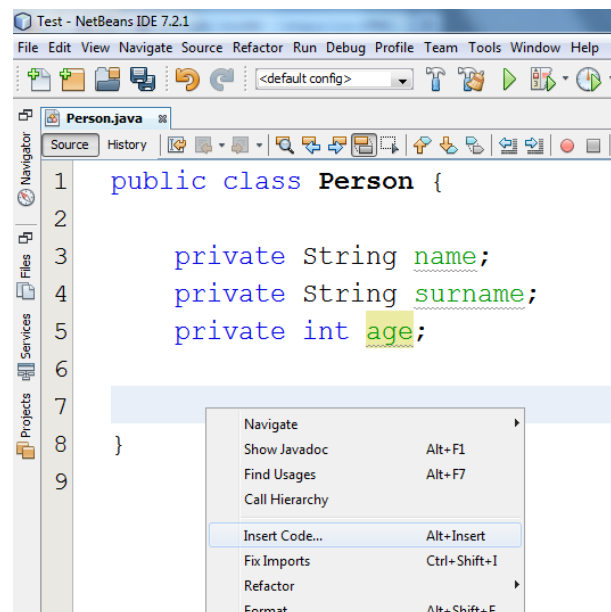
Notice in the above example the use of the **this** keyword. The **this** keyword is a reference to the current object — the object whose method is being called. You can refer to any member of the current object from within an instance method by using the keyword **this**.

Usually the **this** keyword is used when a parameter is passed to a method and this parameter has the same identifier as another data field in that class. The **this** is used to access the data field of that class and not the parameter or variable of a particular method.

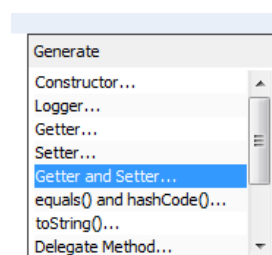
Automatically generating getters and setters

It is very tedious to write getters and setters for every attribute in a class. For this reason, professional IDEs such as NetBeans, offers the facility to automatically generate the getters and setters.

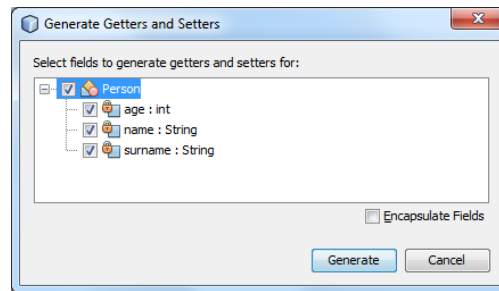
In NetBeans, first write the attributes (with private access modifier) and then right click on the text editor area and select the *Insert Code...* option as shown below:



An option list as shown below will be displayed. Select the option *Getter and Setter...*



A dialog box will be displayed as shown below. Select the fields to encapsulate (i.e. to generate the getters and the setters) and click the *Generate* button.



Constructors

A constructor is a special method for instantiating a new instance of a class. This special type of method is invoked as soon as an instance of a class is instantiated. The constructor method for a class will have the same name as the class. A class may have multiple constructors. In this case, each constructor will have the same name, but will have different parameters. A *parameter-less constructor* is a constructor which does not take any parameters.

Let's take as an example the above *Person* class. In order to create an instance of *Person*, first a *Person* has to be instantiated using the **new** keyword and then the *name*, *surname* and *age* of that instance of person has to be set individually using the setter. Constructors provide us with the capability to set such fields automatically during the initialisation of the object by passing the values of these fields as parameters to the object initialisation, for instance, consider the following code example with a constructor for class *Person*:

```
public class Person{

    // Attributes
    private String name;
    private String surname;
    private int age;

    // Constructor
    public Person(String name,String surname,int age){
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    // Setters
    public void setName(String name){
        this.name = name;
    }

    public void setSurname(String surname){
        this.surname = surname;
    }

    public void setAge(int age){
        this.age = age;
    }

    // Getters
    public String getName(){
        return name;
    }
}
```

```
        public String getSurname() {
            return surname;
        }

        public int getAge() {
            return age;
        }
    }
}
```

The above class can be initiated as follows:

```
Person p = new Person("Matthew", "Xuereb", 23);
```

Note that a class can have more than one constructor. This is called *Constructor Overloading*. For instance in the example below, class *Person* has three constructors, one of them empty, meaning that an instance of that class can be created without having to pass to it any parameters.

```
public class Person{

    // Attributes
    private String name;
    private String surname;
    private int age;

    // Empty constructor
    public Person() {
    }

    // Constructor
    public Person(String name, String surname, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    // Another Constructor
    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    // Setters
    public void setName(String name) {
        this.name = name;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public void setAge(int age) {
        this.age = age;
    }

    // Getters
    public String getName() {
        return name;
    }
}
```



```
        public String getSurname() {  
            return surname;  
        }  
  
        public int getAge() {  
            return age;  
        }  
    }  
}
```

Inheritance

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with a school records system. You could create a generic *Person* class with the states and actions that are common to all persons. Then more specific classes could be defined for students, teachers and administrative staff.

The generic class is known as the parent (or super class or base class) and the specific classes as children (or sub classes or derived classes). The concept of inheritance enhances significantly the ability to reuse code as well as making design a much simpler and cleaner process.

The extends keyword

Java uses the ***extends*** keyword to indicate an inheritance relationship between a parent class and a sub class. For example, using our *Person* class, let's create another class *Teacher* that inherits from class *Person*. Note that a teacher has all the properties of a *Person*, with some extra properties – in this case *subject* and *qualification*.

```
public class Teacher extends Person{  
  
    private String subject;  
    private String qualification;  
  
    public String getQualification() {  
        return qualification;  
    }  
  
    public void setQualification(String qualification) {  
        this.qualification = qualification;  
    }  
  
    public String getSubject() {  
        return subject;  
    }  
  
    public void setSubject(String subject) {  
        this.subject = subject;  
    }  
}
```

Since the above class extends from class *Person*, class *Teacher* is also a *Person*. Note that inheritance implies an ***“IS A” relationship between classes – A Teacher IS A Person***. Class teacher has all the methods and attributes of super class *Person*. That is, if an instance of

Teacher is created, the programmer can invoke also the public methods of class *Person* as shown in the code snippet below:

```
Teacher t1 = new Teacher();
t1.setName("Matthew");
t1.setSurname("Xuereb");
t1.setAge(27);
t1.setQualification("B.Sc. I.T. (Hons.), M.Sc.");
t1.setSubject("Computing");
```

The **protected** access modifier

Till now the only two access modifiers used were *public* and *private*. However, suppose that you want to be able to access the fields of a particular class from its sub classes however you don't want to allow access of these fields from other classes. This can be done by using the *protected* access modifier. In the code snippet below, all the fields of class *Person* has been changed to ***protected***.

```
public class Person {

    protected String name;
    protected String surname;
    protected int age;

    ...
    ...
    ...
}
```

The above *protected* fields can be accessed from subclasses of class *Person* as if they were fields of the same subclass.

Method overriding

Method overriding is a technique that allows a method that already exists in a super class, to be implemented again in a sub class using the same method signature. For example, in the code snippet below, a method *displayDetails()* in class *Teacher* is included using the same signature as in the super class *Person*.

```
public class Teacher extends Person{
    ...
    public void displayDetails(){
        System.out.println("DISPLAYING THE DETAILS OF A TEACHER:");
        System.out.println("NAME: " + name);
        System.out.println("Surname: " + surname);
        System.out.println("AGE: " + age);
        System.out.println("Qualification: " + qualification);
        System.out.println("Subject: " + subject);
    }
    ...
}
```

The method overriding technique is a very powerful technique especially when used in conjunction with polymorphism. More about overriding and polymorphism will be explained in the polymorphism section.

The super keyword

The ***super*** keyword is used to refer to a method, a constructor or an attribute in a super class from within a sub class. For instance, the above *displayDetails()* method can be better implemented by invoking the *displayDetails()* method of the super class (in this case class *Person*) and then add the required statements as shown in the code snippet below:

```
public class Teacher extends Person{
    ...
    ...
    public void displayDetails(){
        super.displayDetails();
        System.out.println("Qualification: " + qualification);
        System.out.println("Subject: " + subject);
    }
    ...
    ...
}
```

Constructors in inherited classes

Sometimes it is necessary to invoke the constructor of a super class from a derived class (sub class). This is possible by using the *super* keyword. For example, suppose that the constructor of the derived class *Teacher* takes as arguments the name, surname, age, qualification and subject. To set the name, surname and age without invoking the *Person* (super class) constructor, the *Teacher* constructor should be as shown in the code snippet below:

```
public Teacher(String name,String surname,int age, String quailification,String subject){
    this.name = name;
    this.surname = surname;
    this.age = age;
    this.qualification = qualification;
    this.subject = subject;
}
```

To make the constructor contents shorter, the constructor of the super class *Person* can be invoked using the *super* keyword as shown below:

```
public Teacher(String name,String surname,int age, String quailification,String subject){
    super(name,surname,age);
    this.qualification = qualification;
    this.subject = subject;
}
```

Polymorphism (Part 1)

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Any Java object that can pass more than an *IS A* test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the *IS A* test for their own type and for the class `Object`.

Consider classes *Person*, *Teacher* and *Student*. Both *Teacher* and *Student* are sub classes of class *Person* (recall that a *Teacher* IS A *Person* and a *Student* IS A *Person*). In other words, both *Teacher* and *Student* have all the attributes and functions of class *Person* and more. These types of classes can be instantiated using the sub class name but declared using the super class name. Consider the following code snippets that are declaring and instantiating both a *Teacher* and a *Student*:

```
Person p1 = new Teacher();
Person p2 = new Student();
```

In the above code snippet, although `p1` and `p2` are declared as classes of type *Person*, they are instantiated as objects of type *Teacher* and *Student* respectively.

The abstract keyword

Sometimes it is required to have two sub classes implementing a method with the same signature but with a different implementation that cannot be partially implemented in the super class. Consider the method *displaySalary()* for classes *Teacher* and *Student*. A teacher earns a salary however a student does not earn a salary.

In order to allow method overriding for these two classes, the method *displaySalary()* with no implementation has to be included in the super class *Person*. This can be achieved by writing an *abstract* method in class *Person* with the *displaySalary()* signature as shown in the code snippet below:

```
public abstract class Person {
    ...
    public abstract void displaySalary();
    ...
}
```

An abstract method indicates that the method is not being implemented in that class and an implementation for it will be provided using the *overriding* technique in its sub classes. Note that if a class contains an abstract method, the class itself has to be declared as abstract as well. When a class is declared as abstract it means that it cannot be instantiated as an object. Instead its subclasses that are implementing the abstract method(s) can be instantiated. The following code snippets illustrate three classes (*Person*, *Teacher* and *Student*) that are using this technique. Note that the getters, setters and constructors are omitted to keep the code as short as possible:

```
public abstract class Person {
    ...
    ...
    public abstract void displaySalary();
    ...
    ...
}

public class Teacher extends Person {
    ...
    ...
    public void displaySalary(){
        System.out.println("EUR 18,000");
    }
    ...
    ...
}

public class Student extends Person {
    ...
    ...
    public void displaySalary(){
        System.out.println("No Salary.");
    }
    ...
    ...
}
```

Arrays

Single-Dimensional Arrays

Single-Dimensional Arrays can be defined as a group of variables containing values that all have the same type, therefore a collection of related data items. The elements of an array are accessed by indexing.

Declaring and creating arrays:

```
int nums[];  
  
nums = new  
int[10];  
  
-- OR --  
  
int num[] = new  
int[10];
```

**Declaring
array**

**Creating
array**

**Declaring and
creating array**

Accessing arrays:

```
nums[2] = 36;  
  
System.out.println  
(nums[5]);
```

Note: An array of size 10 has an index range from 0 to 9

Examples:

```
public static void main(String[] args) {  
    Scanner input = new Scanner(System.in);  
    // Create an array of size 10 of type integer  
    int numbers[] = new int[10];  
    // Ask the user to enter 10 integers to fill the array  
    for(int i = 0; i <= 9; i++){  
        System.out.println("Enter an integer - ");  
        numbers[i] = input.nextInt();  
    }  
  
    // Now display the whole array  
    for(int i = 0; i <= 9; i++){  
        System.out.println(numbers[i]);  
    }  
}
```

Declaring and initialising arrays:

It is possible to declare and assign an array using only one Java instruction as shown below:

```
String[] days = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

int[] a = {1, 2, 4, 8, 16};
```

Getting the length of an array:

Each array has a property variable that has its length. You can find out how many elements an array can hold by writing the array name followed by `.length`. Remember that this is the number of elements in the array, i.e. one more than the maximum subscript.

The following are three examples of accessing arrays, the first one is accessing the array directly using the index number, the second example while the other example makes use of a *for* loop however the length of the array is hard coded. The third example uses a *for* loop however the length of the array is obtained by using the `.length` property provided by the array itself.

Example 1:

```
public class ArraysExample1 {

    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int[] marks = new int[10];

        System.out.print("Enter mark 1 - ");
        marks[0] = input.nextInt();

        System.out.print("Enter mark 2 - ");
        marks[1] = input.nextInt();

        System.out.print("Enter mark 3 - ");
        marks[2] = input.nextInt();

        System.out.print("Enter mark 4 - ");
        marks[3] = input.nextInt();

        System.out.print("Enter mark 5 - ");
        marks[4] = input.nextInt();

        System.out.print("Enter mark 6 - ");
        marks[5] = input.nextInt();

        System.out.print("Enter mark 7 - ");
        marks[6] = input.nextInt();

        System.out.print("Enter mark 8 - ");
        marks[7] = input.nextInt();

        System.out.print("Enter mark 9 - ");
        marks[8] = input.nextInt();

        System.out.print("Enter mark 10 - ");
        marks[9] = input.nextInt();
    }
}
```

Example 2:

```
public class ArraysExample2 {

    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int[] marks = new int[10];

        for(int i = 0;i < 10;i++){
            System.out.print("Enter mark " + (i + 1) + " - ");
            marks[i] = input.nextInt();
        }
    }
}
```

Example 3:

```
public class ArraysExample3 {

    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int[] marks = new int[10];

        for(int i = 0;i < marks.length;i++){
            System.out.print("Enter mark " + (i + 1) + " - ");
            marks[i] = input.nextInt();
        }
    }
}
```

Multi-Dimensional Arrays

Like many other programming languages, Java supports multi-dimensional arrays, i.e. 1-dimensional (as explained above) , 2-dimensional, 3-dimensional, ... In practice most arrays are one-dimensional. Two-dimensional arrays (rows and columns) are also quite common. However higher dimensional arrays are less common and they are only usually used in complex scientific programs.

Two-dimensional arrays are used whenever the model data is best represented with rows and columns, or has two varying aspects (e.g. gender and age, weight and height, etc...). A two dimensional array can also be called a matrix or a table.

Declaring and creating Two-Dimensional arrays:

```
int[][] a = new int[2][3]; // Two rows and three columns.
```

Declaring and initialising 2-D arrays:

You can assign initial values to an array when in a manner very similar to one-dimensional arrays, but with an extra level of braces. The dimension sizes are computed by the compiler from the number of values. The following example would allocate a 3x3 board:

```
int[][] board = new int[][] {{1,0,0},{0,1,0},{1,2,1}};
```


Accessing arrays:

Two-dimensional arrays are almost always processed with nested for loops as shown in the code snippet below:

```
final int ROWS = 2;
final int COLS = 4;

. . .
int[][] a2 = new int[ROWS][COLS];

. . .
//... Print array in rectangular form
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        System.out.print(" " + a2[i][j]);
    }
    System.out.println("");
}
```

ArrayLists

A problem that is encountered when developing programs using arrays is that the size of the array has to be fixed and predefined. An *ArrayList* can be vaguely and informally defined as an array which can dynamically change its size. This feature makes ArrayLists very popular with Java programmers.

Declaring and creating ArrayLists:

```
ArrayList<String> names;  
names = new ArrayList<String>();  
-- OR --  
ArrayList<String> names = new ArrayList<String>();
```

Declaring an ArrayList

Initialising and ArrayList

Declaring and initialising and ArrayList

Accessing ArrayLists:

```
names.add("Matthew");  
-- OR --  
String n = names.get(0);
```

Polymorphism (Part 2)

As outlined in the previous section, with polymorphism one can instantiate an object by declaring it using its super class. Recall the following example:

```
Person t = new Teacher();
```

The usefulness of this technique can be appreciated when arrays or ArrayLists are used. Consider the following scenario:

A school has three types of stakeholders: Students, Teachers and Administrative staff. Declare an array list that can hold the three types of persons, add each type of person to the list and then invoke the method getFixedSalary() on each item in the list.

```
ArrayList<Person> persons = new ArrayList<Person>();  
...  
...  
Person teacher = new Teacher();  
...  
persons.add(teacher);  
...  
Person student = new Student();  
...  
persons.add(student);  
...  
Person admin = new AdministrativeStaff();  
...  
persons.add(admin);  
...  
for(int i = 0; i < persons.size(); i++) {  
    System.out.println(  
        persons.get(i).getFixedSalary());  
}  
...
```

PART 3

Important Programming Features

String handling

Strings are sequences (or an array) of characters. The "+" operator is used for concatenation as shown below:

```
String name = "Matt" + "hew";
String fullName = name + " Xuereb";
```

All other operations on strings are done with methods available in the *java.lang.String* class as explained in this section.

Searching a String for a Character or a Substring

```
String string = "madam, i am Adam";

// Characters
// First occurrence of a c
int index = string.indexOf('a');    // 1

// Last occurrence
index = string.lastIndexOf('a');    // 14

// Not found
index = string.lastIndexOf('z');    // -1

// Substrings
// First occurrence
index = string.indexOf("dam");      // 1

// Last occurrence
index = string.lastIndexOf("dam");  // 13

// Not found
index = string.lastIndexOf("z");    // -1
```

Getting a Substring from a String

```
int start = 1;
int end = 4;
String substr = "aString".substring(start, end);    // Str
```

Replacing Characters in a String

Since strings are immutable, the `replace()` method creates a new string with the replaced characters.

```
// Replace all occurrences of 'a' with 'o'
String newString = string.replace('a', 'o');
```

Converting a String to Upper or Lower Case

```
// Convert to upper case
String upper = string.toUpperCase();

// Convert to lower case
String lower = string.toLowerCase();
```

Converting a String to a Number

```
byte b = Byte.parseByte("123");
short s = Short.parseShort("123");
int i = Integer.parseInt("123");
long l = Long.parseLong("123");
float f = Float.parseFloat("123.4");
double d = Double.parseDouble("123.4e10");
```

Breaking a String into Words

```
String aString = "word1 word2 word3";
StringTokenizer parser = new StringTokenizer(aString);
while (parser.hasMoreTokens()) {
    processWord(parser.nextToken());
}
```

Comparing Strings

```
String s1 = "a";
String s2 = "A";
String s3 = "B";

// Check if identical
boolean b = s1.equals(s2);           // false

// Check if identical ignoring case
b = s1.equalsIgnoreCase(s2);        // true

// Check order of two strings
int i = s1.compareTo(s2);            // 32; lowercase follows uppercase
if (i < 0) {
    // s1 precedes s2
} else if (i > 0) {
    // s1 follows s2
} else {
    // s1 equals s2
}

// Check order of two strings ignoring case
i = s1.compareToIgnoreCase(s3);      // -1
if (i < 0) {
    // s1 precedes s3
} else if (i > 0) {
    // s1 follows s3
} else {
    // s1 equals s3
}
```

Constructing a String using the *StringBuffer* class

If you are constructing a string with several appends, it may be more efficient to construct it using a `StringBuffer` and then convert it to an immutable `String` object.

```
StringBuffer buf = new StringBuffer("Java");

// Append
buf.append(" Almanac v1/");
buf.append(3);

// Set
int index = 15;
```

```
buf.setCharAt(index, '.');

// Insert
index = 5;
buf.insert(index, "Developers ");

// Replace
int start = 27;
int end = 28;
buf.replace(start, end, "4");

// Delete
start = 24;
end = 25;
buf.delete(start, end);

// Convert to string
String s = buf.toString();
```

Exception handling

The Java programming language uses *exceptions* to handle errors and other exceptional events. An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.

Creating an exception object and handing it to the runtime system is called *throwing an exception*. After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible “somethings” to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

try..catch statements

Put a *try..catch* statement around any section of code that might generate a user generated exception (e.g. taking an integer input using the *Scanner* class). The simplest form is:

```
try {  
    // Normal statements that might cause a problem  
} catch (exception-name parameter-name) {  
    // Statements to execute if exception-name occurred.  
}
```

Example

```
public static void main(String[] args) {  
    Scanner input = new Scanner(System.in);  
    try{  
        System.out.println("Enter an integer");  
        int x = input.nextInt();  
    }catch(Exception e){  
        System.out.println("ERROR!! Incorrect input");  
    }  
}
```


In the above example, a *try..catch* statement is surrounding a piece of code that is used to accept an integer input from the user. If the above code snippet is not surrounded within the above *try..catch* statement and the user types a string instead an integer and presses the enter key, the program will crash and the following exception is outputted:

```
Exception in thread "main" java.util.InputMismatchException
  at java.util.Scanner.throwFor(Scanner.java:840)
  at java.util.Scanner.next(Scanner.java:1461)
  at java.util.Scanner.nextInt(Scanner.java:2091)
  at java.util.Scanner.nextInt(Scanner.java:2050)
  at oopexamples.Main.main(Main.java:12)
Java Result: 1
```

However if the code that can throw an exception is surrounded with a *try..catch* statement and an error occurs, instead of crashing, the program will catch that exception, and continues with its execution. For instance, in the above example if an exception occurs, the flow of execution enters the *catch branch* and executes the instruction that outputs the message "Error!! Incorrect input".

Exceptions that can be caught

Note that in the *catch* block of a *try..catch* statement the exception name that is going to be caught is stated. If the exception name is *Exception* as shown in the above example, then each type of exception is caught since *Exception* is a super class and all other types of exceptions inherits from this class.

There are various types of exception that can be caught. It is suggested to view the whole list of exceptions from the Oracle website. In Java, it is possible to catch more than one exception as shown in the example below:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    try{
        System.out.println("Enter an integer");
        int x = input.nextInt();
        System.out.println("Enter a divider");
        int y = input.nextInt();
        int division = x / y;
        System.out.println(x + " / " + y + " = " + division);
    }catch(InputMismatchException e){
        System.out.println("ERROR!! Incorrect input");
    }catch(ArithmeticException e){
        System.out.println("ERROR!! An arithmetic error occurred");
    }
}
```

In the above example if the user enters text instead of integers, an *InputMismatchException* is thrown and the flow of the program enters the *InputMismatchException* branch. On the other hand, if for example the user enters the number 0 as the divider and a division by zero exception is thrown, the flow of the program enters the *ArithmeticException* branch.

Note that if multiple exceptions are caught, the exception with the highest priority should be written first.

The finally block

It is possible to add another block at the end of the *try..catch* statement as shown in the example below. This is the *finally* block and it always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    try{
        System.out.println("Enter an integer");
        int x = input.nextInt();
        System.out.println("Enter a divdor");
        int y = input.nextInt();
        int division = x / y;
        System.out.println(x + " / " + y + " = " + division);
    }catch(InputMismatchException e){
        System.out.println("ERROR!! Incorrect input");
    }catch(ArithmeticException e){
        System.out.println("ERRPR!! An arithmetic error occured");
    }finally{
        System.out.println("Thank you for using this program.");
    }
}
```

Throwing exceptions

It is possible to programmatically throw an exception. To throw an exception the *throw* keyword is used as shown in the code snippet below:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    try{
        System.out.println("Enter an integer between greater than 0");
        int x = input.nextInt();

        if(x > 0){
            throw new Exception("Number out of range");
        }
        System.out.println("Enter a divdor");
        int y = input.nextInt();
        int division = x / y;
        System.out.println(x + " / " + y + " = " + division);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

File handling

In Java there are various different ways to handle files. The basic functions that are required when working with files are to create a file, to check if a file exists or not, to delete a file, to read to a file, to write data to a file, and to append data to a file. These are explained below:

Creating a file

```
try {
    File file = new File("filename");

    // Create file if it does not exist
    boolean success = file.createNewFile();
    if (success) {
        // File did not exist and was created
    } else {
        // File already exists
    }
} catch (IOException e) {
    // Take care of exception
}
```

Checking if a file directory exists

```
boolean exists = (new File("filename")).exists();
if (exists) {
    // File or directory exists
} else {
    // File or directory does not exist
}
```

Deleting a file

```
boolean success = (new File("filename")).delete();
if (!success) {
    // Deletion failed
}
```

Reading text from a file

```
try {
    BufferedReader in = new BufferedReader(new FileReader("infilename"));
    String str;
    while ((str = in.readLine()) != null) {
        process(str);
    }
    in.close();
} catch (IOException e) {
    // Take care of exception
}
```

Writing text to a file

```
try {
    BufferedWriter out = new BufferedWriter(new FileWriter("outfilename"));
    out.write("aString");
    out.close();
} catch (IOException e) {
    // Take care of exception
}
```

Appending text to a file

```
try {
    BufferedWriter out = new BufferedWriter(new FileWriter("filename", true));
    out.write("aString");
    out.close();
} catch (IOException e) {
    // Take care of exception
}
```

Saving an object to a file (using *Serialisation*)

```
MyObject obj;
// Class MyObject should implements the interface Serializable
...
try {
    FileOutputStream fileOut = new FileOutputStream(FILE_PATH);
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(obj);
    out.close();
    fileOut.close();
} catch (Exception e) {
    // Take care of exception
}
```

Loading an object to a file (using *Serialisation*)

```
MyObject obj;
// Class MyObject should implements the interface Serializable
...
try {
    FileInputStream fileIn = new FileInputStream(FILE_PATH);
    ObjectInputStream in = new ObjectInputStream(fileIn);
    Obj = (MyObject) in.readObject();
    in.close();
    fileIn.close();
} catch (Exception e) {
    // Take care of exception
}
```

The Java Math Class

The Java Math class is a class that is used to perform a lot of mathematical operations. This class includes two constants that are used a lot in mathematics – the PI and the E (The base of the natural logarithm).

Displaying the PI and the E on the screen

```
System.out.println("PI: " + Math.PI);  
System.out.println("E : " + Math.E);
```

Displaying the absolute value of a double

```
System.out.println(Math.abs(56.34576));
```

Displaying the Tangent value of a number

```
System.out.println(Math.tan(0.3));
```

Converting and displaying an angle from radians to degrees

```
System.out.println(Math.toDegrees(3.0));
```

Generating and displaying a random number between 1 and 100

```
int randomNumber = (int)(Math.random() * 100) + 1;
```

Rounding a decimal number to two decimal places

```
double rounded = Math.round(num * 100.0) / 100.0;
```

Rounding a decimal number up to a particular number of decimal places (Using the DecimalFormat Class)

The Math class is not sufficient to round a decimal number to a particular number of decimal places. To do this the class *DecimalFormat* should be used. Note that in order to use this class it has to be imported and instantiated as shown below.

```
import java.text.DecimalFormat;  
  
public class DecimapPlacesRounding {  
    public static void main(String[] args) {  
        DecimalFormat twoDRounding = new DecimalFormat("#.##");  
        System.out.println(twoDRounding.format(4.147425624)); // 4.15  
    }  
}
```

PART 4

User Interface Programming

GUI Programming

A Graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. A GUI gives an application a distinctive *look and feel*. Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly and use it more productively.

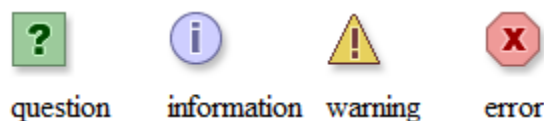
In Java, there are two ways to develop a program with a GUI. The GUI can either be fully coded by the programmer, or otherwise *GUI builder* tools can be used. Such tools provide a mechanism that allows the programmer to graphically build the GUI of their program by dragging and dropping components from a components palette to their actual program pane. The NetBeans IDE provides such a tool. Although ideally where possible, a GUI should be created via such tools, it is very important that Java programmers are capable to generate a GUI using coding.

Java swing components

Swing is a widget toolkit for Java. It is an API for providing a graphical user interface (GUI) for Java programs. Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit. Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. In this section, the most basic and important Swing components are listed and explained:

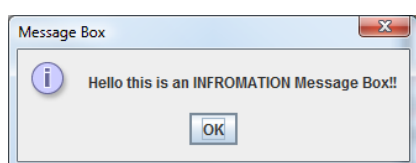
JOptionPane

Using `JOptionPane`, you can quickly create and customize several different kinds of dialogs. `JOptionPane` provides support for laying out standard dialogs, providing icons, specifying the dialog title and text, and customizing the button text. Other features allow you to customize the components the dialog displays and specify where the dialog should appear onscreen. `JOptionPane`'s icon support lets you easily specify which icon the dialog displays. You can use a custom icon, no icon at all, or any one of four standard `JOptionPane` icons (question, information, warning, and error) as shown below:

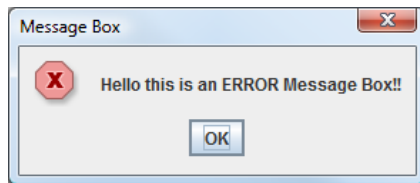


Message dialog box

The `showMessageDialog` method displays a simple dialog with one button, which is labeled "OK". You can easily specify the message, icon, and title that the dialog displays.



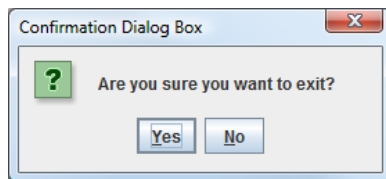
```
JOptionPane.showMessageDialog(myApp,  
    "Hello this is an INFROMATION Message Box!!",  
    "Message Box",  
    JOptionPane.INFORMATION_MESSAGE);
```



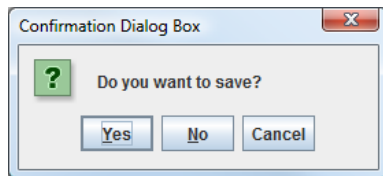
```
JOptionPane.showMessageDialog(
    myApp,
    "Hello this is an ERROR Message Box!!",
    "Message Box",
    JOptionPane.ERROR_MESSAGE);
```

Confirmation dialog box

The `showConfirmDialog` method displays a confirmation dialog box. It returns an integer indicating which option was selected.



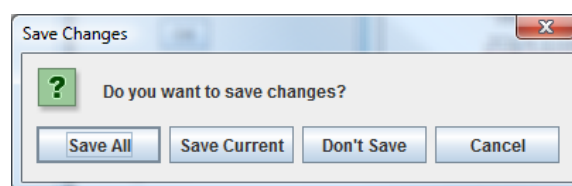
```
int n = JOptionPane.showConfirmDialog(myApp,
    "Are you sure you want to exit?",
    "Confirmation Dialog Box",
    JOptionPane.YES_NO_OPTION);
```



```
int n = JOptionPane.showConfirmDialog(myApp,
    "Do you want to save?",
    "Confirmation Dialog Box",
    JOptionPane.YES_NO_CANCEL_OPTION);
```

Option dialog box

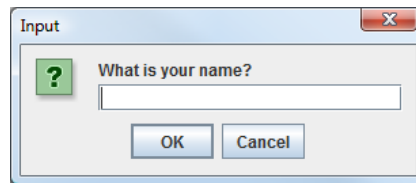
The `showOptionDialog` method displays a modal dialog with the specified buttons, icons, message, title, and so on. It returns an integer indicating which option was selected. With this method, you can change the text that appears on the buttons of standard dialogs. You can also perform many other kinds of customization.



```
String[] options = {"Save All", "Save Current", "Don't Save", "Cancel"};
int n = JOptionPane.showOptionDialog(myApp, "Do you want to save changes?",
    "Save Changes",
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE, null,
    options, options[0]);
```

Input dialog box

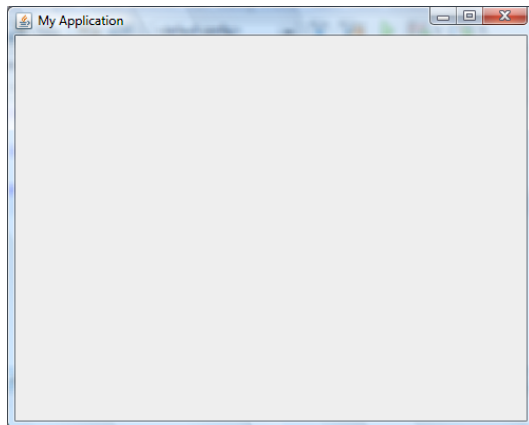
The `showInputDialog` method is used to get a text input from the user.



```
String ans = JOptionPane.showInputDialog(myApp, "What is your name?");
```

JFrame

A JFrame is the swing component that represents a window as shown in the screen shot below. It is the physical window that you'll be working with in the Swing API.



```
JFrame f = new JFrame();  
f.setSize(500,400);  
f.setVisible(true);  
f.setDefaultCloseOperation(  
    JFrame.EXIT_ON_CLOSE);  
f.setTitle("My Application");
```

The above code snippet illustrates how to create an instance of a JFrame. As usually, the JFrame is representing the application, many times it makes more sense to make the application class extending (inheriting) from class *JFrame* as shown in the screen shot below rather than creating an instance of it.

```
public class MyApplication extends JFrame{  
  
    public MyApplication(){  
        setSize(500,400);  
        setVisible(true);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setTitle("My Application");  
    }  
  
    public static void main(String[] args){  
        new MyApplication();  
    }  
}
```

JLabel

A JLabel component is used to display un-editable text or icons on the application pane. The screen shot below illustrates a JFrame with three different types of labels, namely a JLabel with a change in the default font type, a JLabel representing an image rather than text, and then a plain (default) JLabel.



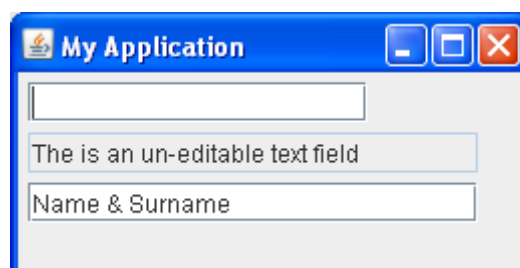
```
JLabel label1 = new JLabel("Traditional Boats");
label1.setFont(new Font("Serif", Font.BOLD, 20));

ImageIcon icon = new ImageIcon("c:\\luzzu.jpg", "Luzzu");
JLabel label2 = new JLabel(icon);

JLabel label3 = new JLabel("This is a Maltese Luzzu");
```

JTextField

A JTextField is another swing component that enables the users to enter text from the keyboard. It can also be used to display editable or un-editable text. The following screen shot illustrates three examples of JTextFields in an application:



```
JTextField textField1 = new JTextField(15);

JTextField textField2 = new JTextField("The is an un-editable text
field",20);
textField2.setEditable(false);

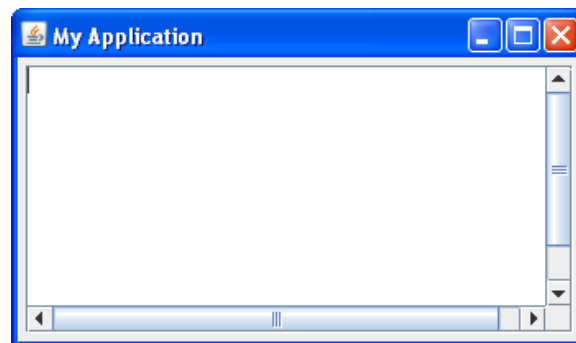
JTextField textField3 = new JTextField("Name & Surname",20);
```

In order to get the contents of a particular text field, the method *getText()* is used while to set the text of a particular text field, the method *setText(String s)* is used as shown below:

```
String s = textField1.getText();  
textField2.setText("Matthew Xuereb Computer Tuition");
```

JTextArea

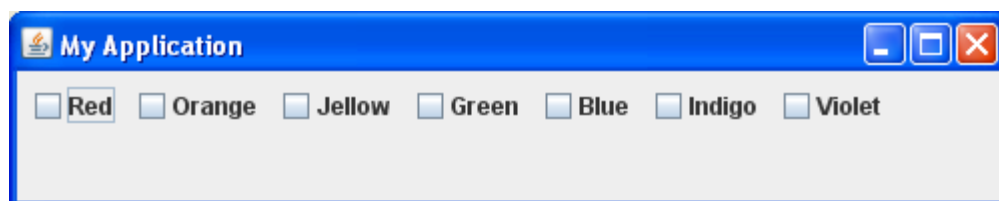
A *JTextArea* is a component that enables the user to do multiple lined text entry. The *setText(String s)* and *getText()* method are used to set and get the text from this component respectively. Many times this component is used in conjunction with the *JScrollPane* component so as to allow the user to scroll the contents of this component as shown below:



```
JTextArea textArea = new JTextArea(10,30);  
add(new JScrollPane(textArea));
```

JCheckBox

A *JCheckBox* is a component that specifies an option that can be selected or not selected as shown in the screen shot below:



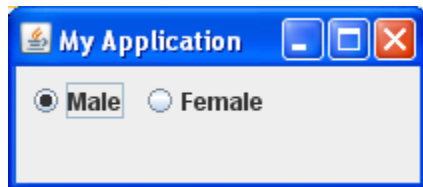
```
JCheckBox cBox1 = new JCheckBox("Red");  
JCheckBox cBox2 = new JCheckBox("Orange");  
JCheckBox cBox3 = new JCheckBox("Jellow");  
JCheckBox cBox4 = new JCheckBox("Green");  
JCheckBox cBox5 = new JCheckBox("Blue");  
JCheckBox cBox6 = new JCheckBox("Indigo");  
JCheckBox cBox7 = new JCheckBox("Violet");
```

To set a checkbox to be selected or not and to check whether a checkbox is selected or not, the methods *setSelected(boolean s)* and *isSelected()* are used respectively as shown in the code snippet below:

```
cBox1.setSelected(true);  
boolean selected = cBox2.isSelected();
```

JRadioButton

Radio buttons are similar to check boxes in that they have two states – selected and not selected. However radio buttons normally appear as a group in which only one button can be selected at a time. `JRadioButton` is the swing component for a radio button. An example of this component is shown below:

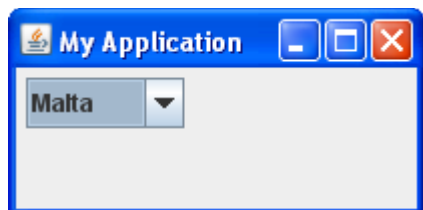


```
JRadioButton male = new JRadioButton("Male");  
JRadioButton female = new  
JRadioButton("Female");  
  
ButtonGroup buttGroup = new ButtonGroup();  
buttGroup.add(male);  
buttGroup.add(female);
```

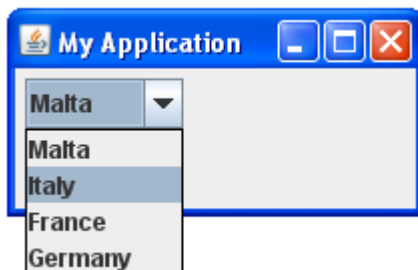
Similar the `JCheckBox`, the methods `setSelected(boolean s)` and `isSelected()` are used to set a radio button and to check whether a radio button is selected or not.

JComboBox

A `JComboBox` is a swing component that provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.



```
String[] items =  
    {"Malta" , "Italy" , "France" , "Germany"};  
JComboBox comboBox = new JComboBox(items);
```



GUI Layout Managers

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window.

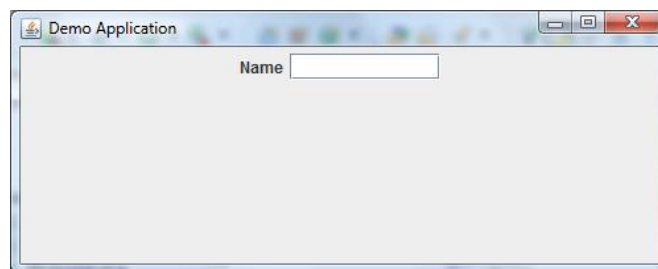
A container can contain another container. For example, a window can contain a panel, which is itself a container. The layout manager in effect for a frame determines how the two panels are sized and positioned inside the frame, and the layout manager in effect for each panel determines how components are sized and positioned inside the panels.

Every Swing container has a predefined layout manager as its default. It is easy to use the `.setLayout` method to change the layout manager, and you can define your own layout manager by implementing the `java.awt.LayoutManager` interface. The `java.awt` package provides the following predefined layout managers:

- `FlowLayout`
- `BorderLayout`
- `GridLayout`
- `BoxLayout`
- `CardLayout`

FlowLayout

The flow layout places the panel components in rows according to the width of the panel and the number and size of the components. The best way to understand flow layout is to resize the demonstration window and notice how the components flow from one row to the other as you make the window wide and narrow. See example below:



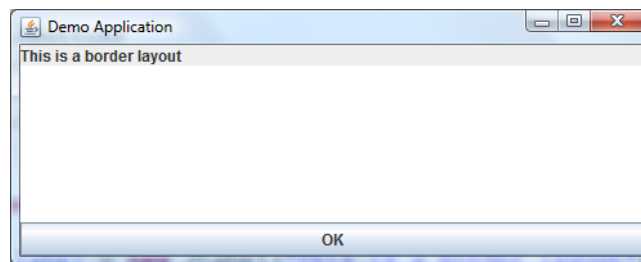
```
public class Demo extends JFrame{
    private JLabel label = new JLabel("Name");
    private JTextField nameField = new JTextField(10);

    public Demo(){
        super("Demo Application");
        setSize(500,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(label);
        add(nameField);
        setVisible(true);
    }
}
```

```
        public static void main(String[] args) {  
            new Demo();  
        }  
    }
```

BorderLayout

A border layout places components in up to five areas: North, South, West, East, and Center. All extra space is placed in the center area. You do not have to put a component in every area of the border layout. The code snippet below puts the label "Border Layout" in the North area, the OK button in the South area, and the text area in the Center area. The East (right) and West (left) areas are empty.

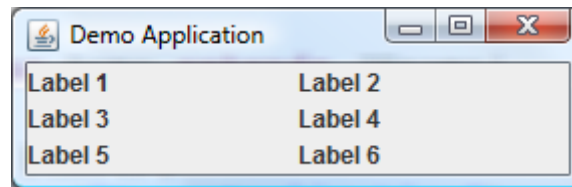


```
public class Demo extends JFrame{  
    private JLabel label = new JLabel("This is a border layout");  
    private JTextArea textArea = new JTextArea();  
    private JButton okButt = new JButton("OK");  
  
    public Demo(){  
        super("Demo Application");  
        setSize(500,200);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setLayout(new BorderLayout());  
        add(label,BorderLayout.NORTH);  
        add(textArea,BorderLayout.CENTER);  
        add(okButt,BorderLayout.SOUTH);  
        setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        new Demo();  
    }  
}
```

GridLayout

The Grid layout arranges components into a grid of rows and columns. You specify the number of rows and columns, the number of rows only and let the layout manager determine the number of columns, or the number of columns only and let the layout manager determine the number of rows.

The cells in the grid are equal size based on the largest component in the grid. Resize the window to see how the components are resized to fit cells as they get larger and smaller. See example below:

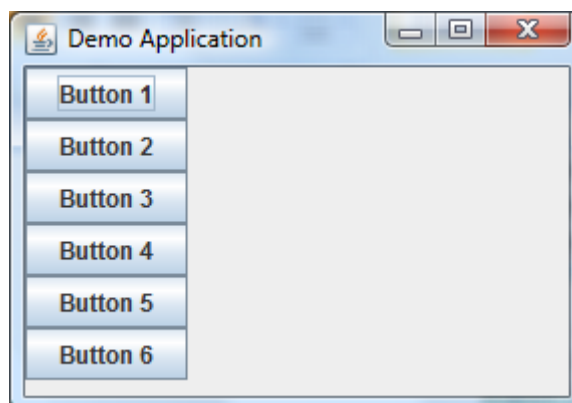


```
public class Demo extends JFrame{
    public Demo(){
        super("Demo Application");
        setSize(500,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(3,2));
        add(new JLabel("Label 1"));
        add(new JLabel("Label 2"));
        add(new JLabel("Label 3"));
        add(new JLabel("Label 4"));
        add(new JLabel("Label 5"));
        add(new JLabel("Label 6"));
        setVisible(true);
    }

    public static void main(String[] args) {
        new Demo();
    }
}
```

BoxLayout

The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components.



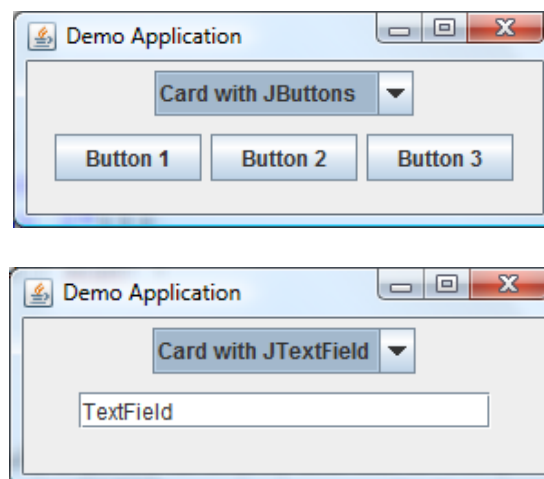
```
public class Demo extends JFrame{
    public Demo(){
        super("Demo Application");
        setSize(500,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BoxLayout(getContentPane(),
            BoxLayout.Y_AXIS));
        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JButton("Button 3"));
        add(new JButton("Button 4"));
        add(new JButton("Button 5"));
        add(new JButton("Button 6"));
        setVisible(true);
    }
}
```

```
    }  
  
    public static void main(String[] args) {  
        new Demo();  
    }  
}
```

CardLayout

The card layout lets you use one container (usually a panel) to display one out of many possible component children (like flipping cards on a table). A program can use this layout to show a different child component to different users. For example, the interface shown to an administrator might have additional functionality from the interface shown to a regular user. With card layout, your program can show the appropriate interface depending on the type of user using the program.

You can also use card layout to let your end user toggle among different displays and choose the one they prefer. In this case, the program must provide a GUI for the user to make the selection. The Choice component is commonly used for this case and the following card layout example shows how to use a Choice component to set up a card layout that lets the end user toggle between two interfaces. See example below:



```
public class Demo implements ItemListener {  
    private JPanel cards;  
    private String BUTTONPANEL = "Card with JButtons";  
    private String TEXTPANEL = "Card with JTextField";  
  
    public void addComponentToPane(Container pane) {  
        JPanel comboBoxPane = new JPanel();  
        String comboBoxItems[] = { BUTTONPANEL, TEXTPANEL };  
        JComboBox cb = new JComboBox(comboBoxItems);  
        cb.setEditable(false);  
        cb.addItemListener(this);  
        comboBoxPane.add(cb);  
  
        //Create the "cards".  
        JPanel card1 = new JPanel();  
        card1.add(new JButton("Button 1"));  
        card1.add(new JButton("Button 2"));  
        card1.add(new JButton("Button 3"));  
        JPanel card2 = new JPanel();
```



```
card2.add(new JTextField("TextField", 20));

//Create the panel that contains the "cards".
cards = new JPanel(new CardLayout());
cards.add(card1, BUTTONPANEL);
cards.add(card2, TEXTPANEL);

pane.add(comboBoxPane, BorderLayout.PAGE_START);
pane.add(cards, BorderLayout.CENTER);
}

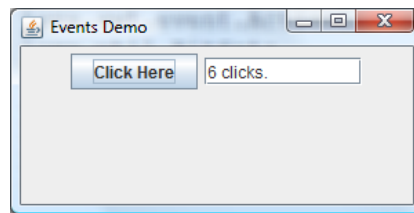
public void itemStateChanged(ItemEvent evt) {
    CardLayout cl = (CardLayout) (cards.getLayout());
    cl.show(cards, (String)evt.getItem());
}

public static void main(String[] args) {
    JFrame f = new JFrame("Demo Application");
    f.setSize(300,120);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Demo demo = new Demo();
    demo.addComponentToPane(f.getContentPane());
    f.setVisible(true);
}
}
```

Listeners and Events – The ActionListener

In Java, events represent all activity that goes on between the user and the application. Java's AWT library communicates these actions to the programs using events. When the user interacts with a program let us say by clicking a command button, the system creates an event representing the action and delegates it to the event handling code within the program.

There are a number of different events however the principle is the same for all of them. The following example illustrates an example of an event for a button. This is handled by the **ActionListener** event handler. When the user clicks on the button, a message is displayed counter is incremented and displayed in the text field:



```
public class EventDemo extends JFrame implements ActionListener{
    private JButton butt = new JButton("Click Here");
    private JTextField textField = new JTextField(10);
    private int buttonClicks = 0;

    public EventDemo(){
        super("Events Demo");
        setSize(300,150);
        setLayout(new FlowLayout());
        add(butt);
        add(textField);
        textField.setText(buttonClicks + " clicks.");
        butt.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e){
        buttonClicks++;
        textField.setText(buttonClicks + " clicks.");
    }

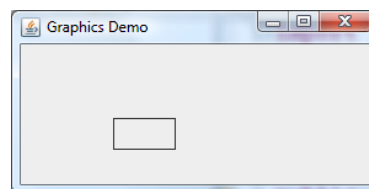
    public static void main(String[] args) {
        new EventDemo();
    }
}
```

Graphics

The Graphics class provides the framework for all graphics operations. It plays two different, but related, roles. First, it is the graphics context. The graphics context is information that will affect drawing operations. This includes the background and foreground colours, the font, and the location and dimensions of the clipping rectangle (the region of a component in which graphics can be drawn). It even includes information about the eventual destination of the graphics operations themselves (screen or image).

Second, the Graphics class provides methods for drawing simple geometric shapes, text, and images to the graphics destination. All output to the graphics destination occurs via an invocation of one of these methods. In order to draw, a program requires a valid graphics context (represented by an instance of the Graphics class). Because the Graphics class is an abstract base class, it cannot be instantiated directly. An instance is typically created by a component, and handed to the program as an argument to a component's ***update()*** and ***paint()*** methods.

The following example illustrates an example of a program that displays a rectangle on the window panel:



```
public class Rectangle extends JPanel{
    private int LENGTH = 50;
    private int BREDTH = 25;

    private int xPos;
    private int yPos;

    public Rectangle(int xPos,int yPos){
        this.xPos = xPos;
        this.yPos = yPos;
    }

    public void paint(Graphics g){
        g.drawRect(xPos,yPos,LENGTH,BREDTH);
    }

    public static void main(String[] args) {
        JFrame f = new JFrame("Graphics Demo");
        f.setSize(300,150);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new Rectangle(75,60));
        f.setVisible(true);
    }
}
```

Listeners and Events – The MouseListener and MouseMotionListener

The ***MouseListener*** and the ***MouseMotionListener*** are event handlers that are used to take care of mouse event and mouse movement events respectively.

The following methods should be implemented for the *MouseListener* event handler:

```
public void mousePressed(MouseEvent e) {  
    ...  
}  
  
public void mouseReleased(MouseEvent e) {  
    ...  
}  
  
public void mouseEntered(MouseEvent e) {  
    ...  
}  
  
public void mouseExited(MouseEvent e) {  
    ...  
}  
  
public void mouseClicked(MouseEvent e) {  
    ...  
}
```

The following methods should be implemented for the *MouseMotionListener* event handler:

```
public void mouseDragged(MouseEvent e) {  
    ...  
}  
  
public void mouseMoved(MouseEvent e) {  
    ...  
}
```

APPENDIX 1: for loops demos

Output 10 "Hello" on the screen

```
public class Test {  
  
    public static void main(String args[]){  
        for(int i = 1; i <= 10;i++){  
            System.out.println("Hello");  
        }  
    }  
}
```

Output the numbers 1..10

```
public class Test {  
  
    public static void main(String args[]){  
        for(int i = 1; i <= 10;i++){  
            System.out.println(i);  
        }  
    }  
}
```

Output the odd numbers 1..10

```
public class Test {  
  
    public static void main(String args[]){  
        for(int i = 1; i <= 10;i+=2){  
            System.out.println(i);  
        }  
    }  
}
```

Output the numbers 10..1

```
public class Test {  
  
    public static void main(String args[]){  
        for(int i = 10; i >= 1;i--){  
            System.out.println(i);  
        }  
    }  
}
```

Average, Minimum & Maximum

```
import java.util.Scanner;

public class Test {

    public static void main(String args[]){
        Scanner input = new Scanner(System.in);
        int total = 0;
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;

        for(int i = 1; i <= 5;i++){
            System.out.print("Please enter a number: ");
            int num = input.nextInt();
            total = total + num;

            if(num > max){
                max = num;
            }

            if(num < min){
                min = num;
            }
        }

        int average = total / 5;
        System.out.println("AVERAGE = " + average);
        System.out.println("MAXIMUM = " + max);
        System.out.println("MINIMUM = " + min);
    }
}
```

APPENDIX 2: Arrays

Creating an array of integers, populating it with numbers and displaying the contents of the array:

```
public static void main(String[] args){
    int nums[] = new int[5];
    nums[0] = 50;
    nums[1] = 12;
    nums[2] = 17;
    nums[3] = 29;
    nums[4] = 3;

    System.out.println(nums[0]);
    System.out.println(nums[1]);
    System.out.println(nums[2]);
    System.out.println(nums[3]);
    System.out.println(nums[4]);
}
```

Declaring and assigning an array of integers:

```
public static void main(String[] args){
    int nums[] = {50 , 12 , 17 , 29 , 3};

    System.out.println(nums[0]);
    System.out.println(nums[1]);
    System.out.println(nums[2]);
    System.out.println(nums[3]);
    System.out.println(nums[4]);
}
```

Using a *for loop* to display the contents of the array:

```
public static void main(String[] args){
    int nums[] = {50 , 12 , 17 , 29 , 3};
    for(int i = 0;i < 5;i++){
        System.out.println(nums[i]);
    }
}
```

It is suggested to use the *.length* method to indicate the size of the array instead of writing the size manually:

```
public static void main(String[] args){
    int nums[] = {50 , 12 , 17 , 29 , 3 , 23 , 56 , 86};
    for(int i = 0;i < nums.length;i++){
        System.out.println(nums[i]);
    }
}
```

Accepting inputs from the user and saving them in an array:

```
public static void main(String[] args){
    Scanner input= new Scanner(System.in);
    int nums[] = new int[5];

    for(int i = 0;i < nums.length;i++){
        System.out.print("Enter a number: ");
        nums[i] = input.nextInt();
    }
}
```

Write a program that asks the user to input 10 integers, stores them in an array and then calculates and outputs the average.

```
import java.util.Scanner;

public class Test {

    public static void main(String[] args){
        Scanner input= new Scanner(System.in);
        int nums[] = new int[10];

        int total = 0;

        for(int i = 0;i < nums.length;i++){
            System.out.print("Enter a number: ");
            nums[i] = input.nextInt();
            total = total + nums[i];
        }

        int average = total / 10;
        System.out.println("The average is " + average);
    }
}
```

Array of Objects Example

```
public class Test {

    public static void main(String[] args){
        Person myFamily[] = new Person[3];
        myFamily[0] = new Person("Matthew", "Xuereb", 29);
        myFamily[1] = new Person("Michaela", "Xuereb", 27);
        myFamily[2] = new Person("Maria", "Xuereb", 1);

        for(int i = 0;i< myFamily.length;i++){
            myFamily[i].displayDetails();
            System.out.println();
        }

    }
}
```


ArrayList of Objects Example

```
import java.util.ArrayList;
```

```
public class Test {
```

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

```
        ArrayList<Person> myFamily = new ArrayList<>();
```

```
        myFamily.add(new Person("Matthew","Xuereb",29));
```

```
        myFamily.add(new Person("Michaela","Xuereb",27));
```

```
        myFamily.add(new Person("Maria","Xuereb",1));
```

```
        for(int i = 0;i < myFamily.size();i++){
```

```
            myFamily.get(i).displayDetails();
```

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

```
        }
```

```
    }
```

```
}
```

APPENDIX 3: String Handling

Replacing a substring in a string

```
public static void main(String[] args) {
    String s1 = "My name is Matthew Xuereb. I am 26 years old and I am Matthew.";
    String s2 = s1.replaceAll("Matthew", "Joe");
    System.out.println(s2);
}
```

Counting the number of occurrences of a character in a string

```
public static void main(String[] args) {
    String s1 = "My name is Matthew Xuereb. I am 26 years old and I am Matthew.";
    int counter = 0;
    for(int i = 0; i < s1.length(); i++){
        if(s1.charAt(i) == '.'){
            counter++;
        }
    }
    System.out.println("There are " + counter + " sentences.");
}
```

Getting a substring from a string

```
public static void main(String[] args) {
    String s1 = "My name is Matthew Xuereb. I am 26 years old and I am Matthew.";
    String s2 = s1.substring(0, s1.indexOf(".") + 1);
    System.out.println(s2);
}
```

Splitting an array using a delimiter

```
public static void main(String[] args) {
    String names = "Joe,Paul,Matthew,John";
    String[] namesArray = names.split(",");
    for(int i = 0; i < namesArray.length; i++){
        System.out.println(namesArray[i]);
    }
}
```

Appendix 4: Helpers Class

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

public class Helpers {

    public static String getDate() {
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
        return dateFormat.format(new Date());
    }

    public static boolean isDateAndTimeValid(String date) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy hh:mm aa");
        dateFormat.setLenient(false);
        boolean dateValid;
        dateValid = true;
        try {
            Date d = dateFormat.parse(date);
        } catch (Exception e) {
            dateValid = false;
        }
        return dateValid;
    }

    public static boolean isDateValid(String date) {
        try {
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
            sdf.setLenient(false);
            sdf.parse(date);
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    public static boolean isEmailAddressValid(String email) {
        String ePattern =
            "^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\)|((\\[a-zA-Z\\-0-9\\]+\\.)+[a-zA-Z]{2,}))$";
        java.util.regex.Pattern p = java.util.regex.Pattern.compile(ePattern);
        java.util.regex.Matcher m = p.matcher(email);
        return m.matches();
    }

    public static String getDateAndTimeInput(String prompt) {
        Scanner input = new Scanner(System.in);
        String ans;
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy hh:mm aa");
        dateFormat.setLenient(false);
        boolean dateValid;
        do {
            System.out.print(prompt);
            ans = input.nextLine();
            ans = ans.trim();
            dateValid = true;
            try {
                Date d = dateFormat.parse(ans);
            } catch (Exception e) {

```

```
        dateValid = false;
    }
} while (!dateValid);
return ans;
}

public static String getDateInput2(String prompt) {
    Scanner input = new Scanner(System.in);
    String ans;
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
    dateFormat.setLenient(false);
    boolean dateValid;
    do {
        System.out.print(prompt);
        ans = input.nextLine();
        ans = ans.trim();
        dateValid = true;
        try {
            Date d = dateFormat.parse(ans);
        } catch (Exception e) {
            dateValid = false;
        }
    } while (!dateValid);
    return ans;
}

public static String getDateInput(String prompt) {
    Scanner input = new Scanner(System.in);
    String ans;
    SimpleDateFormat dateFormat = new SimpleDateFormat("MM yyyy");
    dateFormat.setLenient(false);
    boolean dateValid;
    do {
        System.out.print(prompt);
        ans = input.nextLine();
        ans = ans.trim();
        dateValid = true;
        try {
            Date d = dateFormat.parse(ans);
        } catch (Exception e) {
            dateValid = false;
        }
    } while (!dateValid);
    return ans;
}

public static String getTimeInput(String prompt) {
    Scanner input = new Scanner(System.in);
    String ans;
    SimpleDateFormat dateFormat = new SimpleDateFormat("hh:mm aa");
    dateFormat.setLenient(false);
    boolean timeValid;
    do {
        System.out.print(prompt);
        ans = input.nextLine();
        ans = ans.trim();
        timeValid = true;
        try {
            Date d = dateFormat.parse(ans);
        } catch (Exception e) {
            timeValid = false;
        }
    }
```

```
        }
    } while (!timeValid);
    return ans;
}

public static String getStringInput(String prompt) {
    Scanner input = new Scanner(System.in);
    String ans;

    do {
        System.out.print(prompt);
        ans = input.nextLine();
        ans = ans.trim();
    } while (ans.length() == 0);
    return ans;
}

public static String getTelInput(String prompt) {
    Scanner input = new Scanner(System.in);
    String ans;
    boolean inputValid;
    do {
        System.out.print(prompt);
        ans = input.nextLine();
        ans = ans.trim();
        inputValid = true;
        if (ans.length() != 8) {
            inputValid = false;
        } else {
            for (int i = 0; i < ans.length(); i++) {
                if (Character.isDigit(ans.charAt(i)) == false) {
                    inputValid = false;
                    break;
                }
            }
        }
    } while (!inputValid);
    return ans;
}

public static double getDoubleInput(String prompt) {
    Scanner input = new Scanner(System.in);
    double ans = 0;
    boolean inputValid;
    do {
        System.out.print(prompt);
        String s = input.nextLine();
        // Convert string input to integer
        try {
            ans = Double.parseDouble(s);
            inputValid = true;
        } catch (Exception e) {
            inputValid = false;
        }
    } while (!inputValid);
    return ans;
}

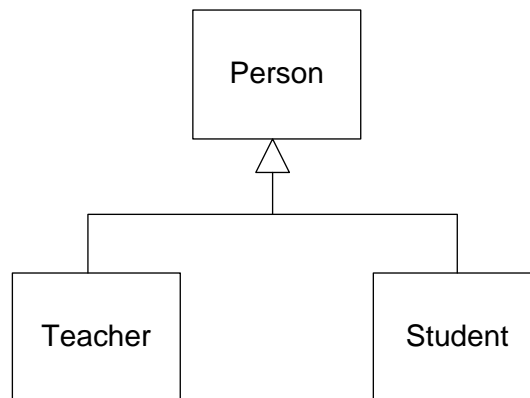
public static int getIntegerInput(String prompt) {
    Scanner input = new Scanner(System.in);
    int ans = 0;
```

```
        boolean inputValid;
        do {
            System.out.print(prompt);
            String s = input.nextLine();
            // Convert string input to integer
            try {
                ans = Integer.parseInt(s);
                inputValid = true;
            } catch (Exception e) {
                inputValid = false;
            }
        } while (!inputValid);
        return ans;
    }

    public static char getCharInput(String prompt, char[] possibilities) {
        while (true) {
            String input = getStringInput(prompt).toUpperCase();
            for (int i = 0; i < possibilities.length; i++) {
                if (input.charAt(0) == possibilities[i]) {
                    return input.charAt(0);
                }
            }
        }
    }

    public static int getIntegerInput(String prompt, int lowerBound, int upperBound) {
        Scanner input = new Scanner(System.in);
        int ans = 0;
        boolean inputValid;
        do {
            System.out.print(prompt);
            String s = input.nextLine();
            // Convert string input to integer
            try {
                ans = Integer.parseInt(s);
                if (ans >= lowerBound && ans <= upperBound) {
                    inputValid = true;
                } else {
                    inputValid = false;
                }
            } catch (Exception e) {
                inputValid = false;
            }
        } while (!inputValid);
        return ans;
    }
}
```

APPENDIX 5: Inheritance & Polymorphism



```
public abstract class Person {

    private String name;
    private String surname;
    private int age;

    public Person(String name,String surname,int age){
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public Person(String name,String surname){
        this.name = name;
        this.surname = surname;
    }

    public Person(){
        // Empty constructor
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }
}
```

```
public void setSurname(String surname) {
    this.surname = surname;
}

public void displayDetails(){
    System.out.println("NAME    : " + name);
    System.out.println("SURNAME: " + surname);
    System.out.println("AGE     : " + age);
}

public void greetingMessage(String message){
    System.out.println(message + " " + name +
        ". Welcome to Java programming :)");
}

public void greetingMessage(String message,String subject){
    System.out.println(message + " " + name + " . Welcome to " + subject);
}

public abstract void displaySalary();
}

.....

public class Teacher extends Person{

    private String qualification;
    private String subject;

    public Teacher(String name, String surname, int age, String qualification,
        String subject) {
        super(name, surname, age);
        this.qualification = qualification;
        this.subject = subject;
    }

    public Teacher(String name,String surname,String qualification,String subject){
        super(name, surname);
        this.qualification = qualification;
        this.subject = subject;
    }

    public String getQualification() {
        return qualification;
    }

    public void setQualification(String qualification) {
        this.qualification = qualification;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }
}
```



```
    }

    @Override
    public void displayDetails(){
        super.displayDetails();
        System.out.println("QUALIFICATIONS: " + qualification);
        System.out.println("SUBJECT: " + subject);
    }

    public void displaySalary(){
        System.out.println("EUR 18,000");
    }
}

.....

public class Student extends Person{

    private String className;
    private String optionalSubject;
    private String optionalLanguage;

    public Student(String name, String surname, int age, String className,
                   String optionalSubject, String optionalLanguage) {
        super(name, surname, age);
        this.className = className;
        this.optionalSubject = optionalSubject;
        this.optionalLanguage = optionalLanguage;
    }

    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }

    public String getOptionalLanguage() {
        return optionalLanguage;
    }

    public void setOptionalLanguage(String optionalLanguage) {
        this.optionalLanguage = optionalLanguage;
    }

    public String getOptionalSubject() {
        return optionalSubject;
    }

    public void setOptionalSubject(String optionalSubject) {
        this.optionalSubject = optionalSubject;
    }

    @Override
    public void displayDetails(){
        super.displayDetails();
        System.out.println("CLASS NAME: " + className);
        System.out.println("LANGUAGE: " + optionalLanguage);
        System.out.println("SUBJECT: " + optionalSubject);
    }
}
```

```
    }

    public void displaySalary(){
        System.out.println("No Salary.");
    }

}
```

```
.....

import java.util.ArrayList;

public class Test {

    public static void main(String[] args) {
        ArrayList<Person> school = new ArrayList<Person>();

        Teacher t = new Teacher("Matthew", "Xuereb", 26, "IT", "Computing");
        Student s1 = new Student("Joe", "Borg", 13, "3B",
                                "Computer Studies", "Italian");
        Student s2 = new Student("Mario", "Grech", 14, "4B",
                                "Business Studies", "French");

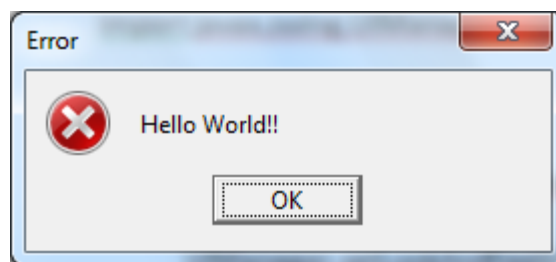
        school.add(t);
        school.add(s1);
        school.add(s2);

        for(int i = 0; i < school.size(); i++){
            school.get(i).displaySalary();
            System.out.println();
        }
    }
}
```

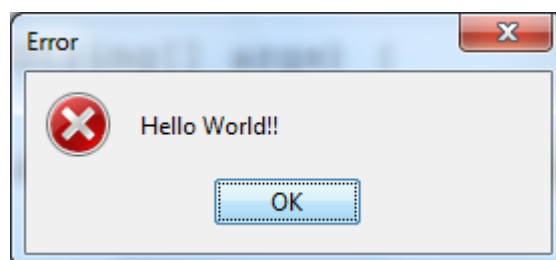
APPENDIX 6: GUI using Swing

Java Look and Feel Demo

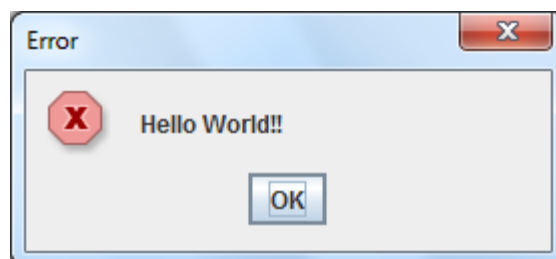
```
import com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel;  
import javax.swing.JOptionPane;  
import javax.swing.UIManager;  
  
public class Test {  
  
    public static void main(String[] args) {  
        try{  
            UIManager.setLookAndFeel(new WindowsClassicLookAndFeel());  
        }catch(Exception e){  
  
        }  
  
        JOptionPane.showMessageDialog(null,"Hello World!!",  
                                     "Error",JOptionPane.ERROR_MESSAGE);  
    }  
}
```



```
UIManager.setLookAndFeel(new WindowsLookAndFeel());
```



```
UIManager.setLookAndFeel(new MetalLookAndFeel());
```



```
UIManager.setLookAndFeel(new MotifLookAndFeel());
```



```
UIManager.setLookAndFeel(new NimbusLookAndFeel());
```



Option Dialog

```
import javax.swing.JOptionPane;
import javax.swing.UIManager;

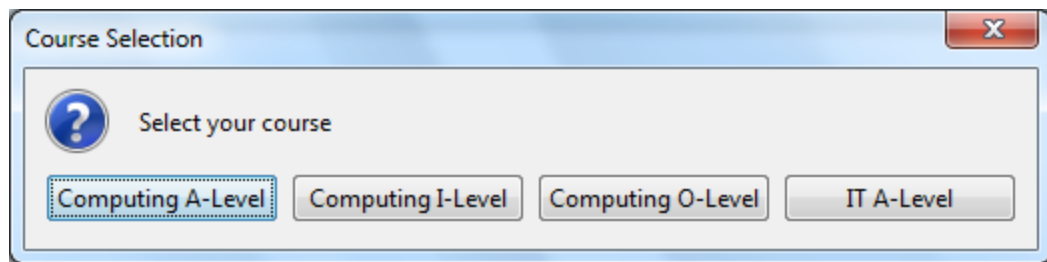
public class Test {

    public static void main(String[] args) {
        try{
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }catch(Exception e){
            // Do nothing
        }

        String options[] = {"Computing A-Level" , "Computing I-Level" ,
                            "Computing O-Level" , "IT A-Level"};

        int x = JOptionPane.showOptionDialog(null,"Select your course",
            "Course Selection" , JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,null,options,options[0]);

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



Input Dialog

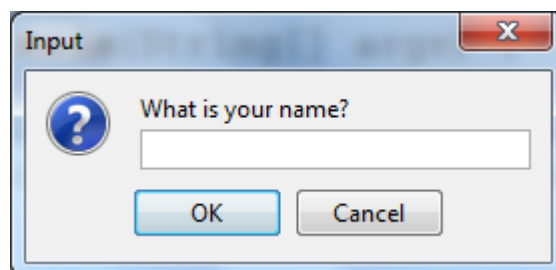
```
import javax.swing.JOptionPane;
import javax.swing.UIManager;

public class Test {

    public static void main(String[] args) {
        try{
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }catch(Exception e){
            // Do nothing
        }

        String x = JOptionPane.showInputDialog("What is your name?");

        System.out.println("Hello " + x);
    }
}
```



GUI Form Demo (Writing Code)

```
import java.awt.FlowLayout;
import javax.swing.BoxLayout;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JSeparator;
import javax.swing.JTextField;
import javax.swing.UIManager;
import javax.swing.plaf.basic.BasicArrowButton;

public class Test extends JFrame {
    private String NATIONALITIES[] = {"Maltese" , "Italian" , "French" , "English"};

    private JTextField nameField = new JTextField(20);
    private JTextField surnameField = new JTextField(20);
    private JComboBox nationalitiesField = new JComboBox(NATIONALITIES);
    private JRadioButton maleRB = new JRadioButton("Male");
    private JRadioButton femaleRB = new JRadioButton("Female");
    private JButton okButton = new JButton("Ok");
    private JButton cancelButton = new JButton("Cancel");

    public Test(){
        setTitle("My First GUI Application");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BoxLayout(getContentPane(),BoxLayout.Y_AXIS));
        add(getFormLine(nameField,"Name"));
        add(getFormLine(surnameField,"Surname"));
        add(getFormLine(nationalitiesField,"Nationalities"));
        add(getFormLine(maleRB,femaleRB,"Gender"));
        add(new JSeparator());
        add(getFormLine(okButton, cancelButton));

        maleRB.setSelected(true);
        ButtonGroup bg = new ButtonGroup();
        bg.add(maleRB); bg.add(femaleRB);

        pack();
    }

    public JPanel getFormLine(JComponent component, String labelText) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.LEFT));
        JLabel lab = new JLabel(labelText);
        p.add(lab);
        p.add(component);
        return p;
    }

    public JPanel getFormLine(JComponent component1,JComponent component2, String
labelText) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.LEFT));
        JLabel lab = new JLabel(labelText);
        p.add(lab);
        p.add(component1);
        p.add(component2);
    }
}
```

```

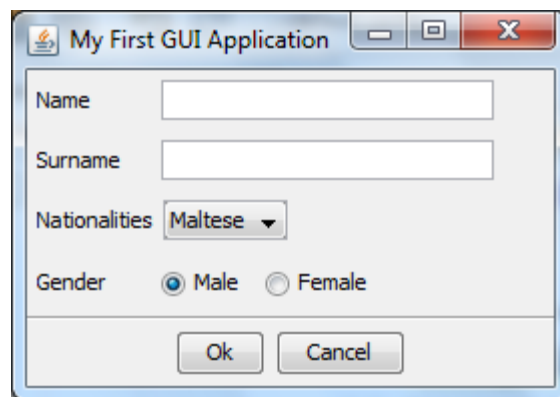
        return p;
    }

    public JPanel getFormLine(JButton button1, JButton button2) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.CENTER));
        p.add(button1);
        p.add(button2);
        return p;
    }

    public static void main(String[] args) {
        try{
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }catch(Exception e){
            // Do nothing
        }

        Test t = new Test();
        t.setVisible(true);
    }
}

```



GUI Form Demo 2 (Drag & Drop Using NetBeans)

```

public class Test2 extends javax.swing.JFrame {

    public Test2() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        buttonGroup1 = new javax.swing.ButtonGroup();
        jLabel1 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jLabel2 = new javax.swing.JLabel();
        jTextField2 = new javax.swing.JTextField();
        jLabel3 = new javax.swing.JLabel();
        jComboBox1 = new javax.swing.JComboBox();
        jLabel4 = new javax.swing.JLabel();
        jRadioButton1 = new javax.swing.JRadioButton();
        jRadioButton2 = new javax.swing.JRadioButton();
        jButton1 = new javax.swing.JButton();
    }
}

```

```

jButton2 = new javax.swing.JButton();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

jLabel1.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
jLabel1.setText("Name");

jLabel2.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
jLabel2.setText("Surname");

jLabel3.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
jLabel3.setText("Nationality");

jComboBox1.setModel(new javax.swing.DefaultComboBoxModel(new String[] {
"Maltese", "Italian", "French", "English" }));

jLabel4.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
jLabel4.setText("Gender");

buttonGroup1.add(jRadioButton1);
jRadioButton1.setSelected(true);
jRadioButton1.setText("Male");

buttonGroup1.add(jRadioButton2);
jRadioButton2.setText("Female");

jButton1.setText("Ok");

jButton2.setText("Cancel");

javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .add(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .add(layout.createSequentialGroup()
                    .add(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                        .add(jLabel1)
                        .add(jLabel2)
                        .add(jLabel3)
                        .add(jLabel4)
                        .add(jComboBox1)
                        .add(jButton1)
                        .add(jButton2)
                    )
                )
            )
        )
    );

```



```

        .addComponent(jLabel1))
        .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addComponent(jTextField2)
        .addComponent(jTextField1)))
        .addContainerGap()
        .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
        .addContainerGap(48, Short.MAX_VALUE)
        .addComponent(jButton1)
        .addGap(49, 49, 49)
        .addComponent(jButton2)
        .addGap(43, 43, 43))
);
layout.setVerticalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel1)
            .addComponent(jTextField1,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel2)
            .addComponent(jTextField2,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel3)
            .addComponent(jComboBox1,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel4)
            .addComponent(jRadioButton1)
            .addComponent(jRadioButton2))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(jButton2)
            .addComponent(jButton1))
            .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
);

pack();
} // </editor-fold>

/**
 * @param args the command line arguments
 */

```

```

    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
        //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code
(optional) ">
        /* If Nimbus (introduced in Java SE 6) is not available, stay with the default
look and feel.
        *
        * For details see
http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
        */
        try {
            for (javax.swing.UIManager.LookAndFeelInfo info :
javax.swing.UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
        } catch (ClassNotFoundException ex) {

java.util.logging.Logger.getLogger(Test2.class.getName()).log(java.util.logging.Level
1.SEVERE, null, ex);
        } catch (InstantiationException ex) {

java.util.logging.Logger.getLogger(Test2.class.getName()).log(java.util.logging.Level
1.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {

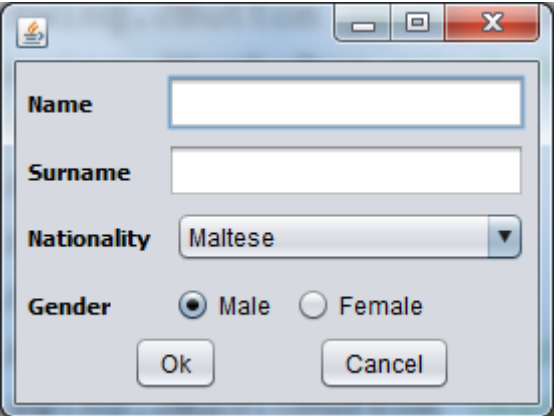
java.util.logging.Logger.getLogger(Test2.class.getName()).log(java.util.logging.Level
1.SEVERE, null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {

java.util.logging.Logger.getLogger(Test2.class.getName()).log(java.util.logging.Level
1.SEVERE, null, ex);
        }
    }
    //</editor-fold>

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Test2().setVisible(true);
        }
    });
}

// Variables declaration - do not modify
private javax.swing.ButtonGroup buttonGroup1;
private javax.swing.JButton jButton1;
private javax.swing.JButton jButton2;
private javax.swing.JComboBox jComboBox1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JRadioButton jRadioButton1;
private javax.swing.JRadioButton jRadioButton2;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
// End of variables declaration
}

```



A Java Swing dialog box with a light blue title bar and standard window controls (minimize, maximize, close). The dialog contains the following fields and controls:

- Name**: A text input field.
- Surname**: A text input field.
- Nationality**: A dropdown menu with "Maltese" selected.
- Gender**: Two radio buttons, "Male" (selected) and "Female".
- Buttons**: "Ok" and "Cancel" buttons at the bottom.

APPENDIX 7: GUIHelpers

```
import java.awt.FlowLayout;
import java.text.SimpleDateFormat;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GUIHelpers {

    public static JPanel getFormLine(JComponent component, String labelText) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.LEFT));
        JLabel lab = new JLabel(labelText);
        p.add(lab);
        p.add(component);
        return p;
    }

    public static JPanel getFormLine(JComponent component1, JComponent component2, String
labelText) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.LEFT));
        JLabel lab = new JLabel(labelText);
        p.add(lab);
        p.add(component1);
        p.add(component2);
        return p;
    }

    public static JPanel getFormLine(JButton button1, JButton button2) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.CENTER));
        p.add(button1);
        p.add(button2);
        return p;
    }

    public static JPanel getFormLine(JButton button1, JButton button2, JButton button3) {
        JPanel p = new JPanel(new FlowLayout(FlowLayout.CENTER));
        p.add(button1);
        p.add(button2);
        p.add(button3);
        return p;
    }
}
```

APPENDIX 8: UML

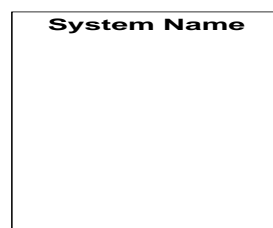
UML Use Case Diagrams

Overview

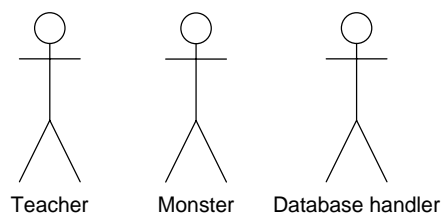
- Use case diagrams are the starting point when designing a new system using UML.
- A use case diagram is the highest form of detail about a system.
- It is an excellent way of communication to management, customers, and other non development people what a system will do when it is completed.
- Does not go into the detail of how a system will do something.
- They are not technical.
- It illustrates who will use the system and what will be able to do with it.

Components

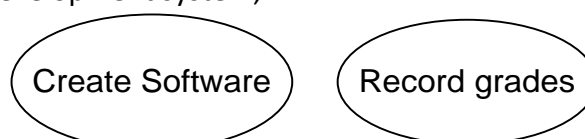
- 4 basic components – systems, actors, use cases, relationships.
- A **system** is something that performs a function
 - Actors and use cases are part of a single system.
 - A system can have subsystems.



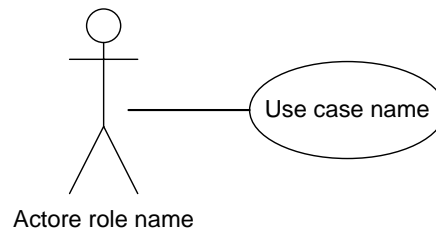
- An **actor** is used to represent something that uses a system. (Can be a person or another system).



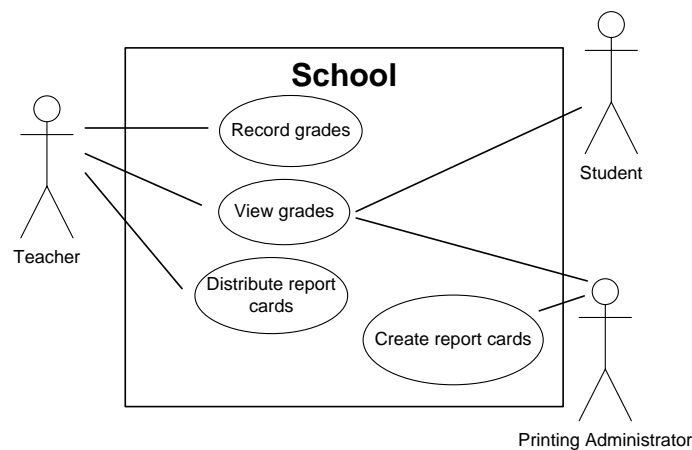
- **Use cases** are the actions that a user takes on a system.
- When naming a use case use active verbs combined with nouns that describe the functionality
- E.g. A teacher would record grades with a grading system, a developer would create software with a development system, ...



- **Relationships** are illustrated with a line connecting actors to use cases.

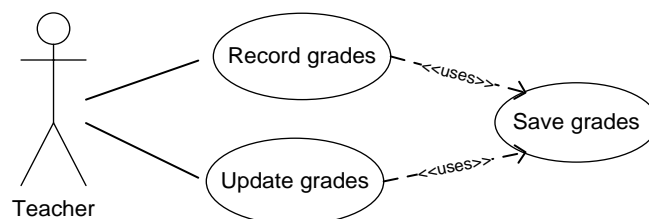


- Actors can relate to multiple use cases and use cases can relate to multiple actors.

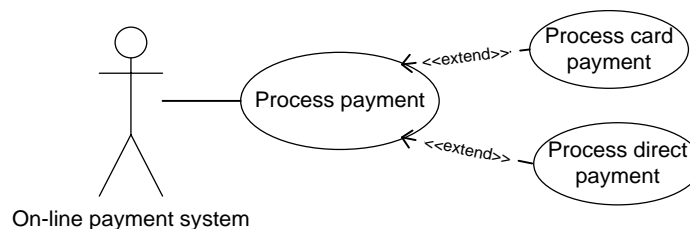


The Uses and Extend relationships

- Two ways of relating use cases with each other when the two are highly related.
- Uses** (or include) is used to indicate that a use case will use functionality from an additional use case to perform its function.

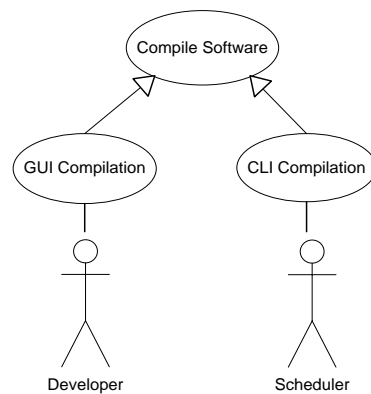
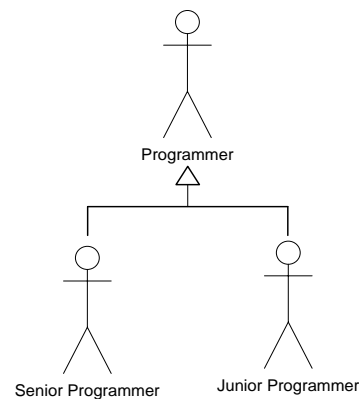


- Extend** relationship indicates that a use case may be extended by another use case.



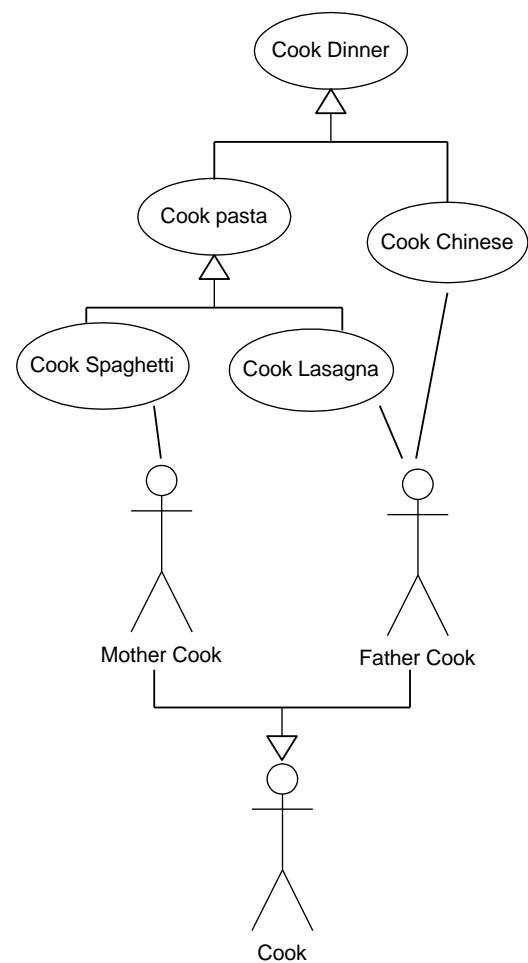
The generalisation technique

- Used to indicate inheritance of an item in UML.
- Can be applied to both actors and use cases.

**Use Case to use case****Actor to actor****Example 1**

Mother and father can both *Cook* for dinner. *Mother Cook* knows how to *Cook Spaghetti* while *Father Cook* knows how to *Cook Lasagna* and *Cook Chinese* food. Sketch a Use Case diagram to depict this scenario. Hint: Make use of the generalisation technique.

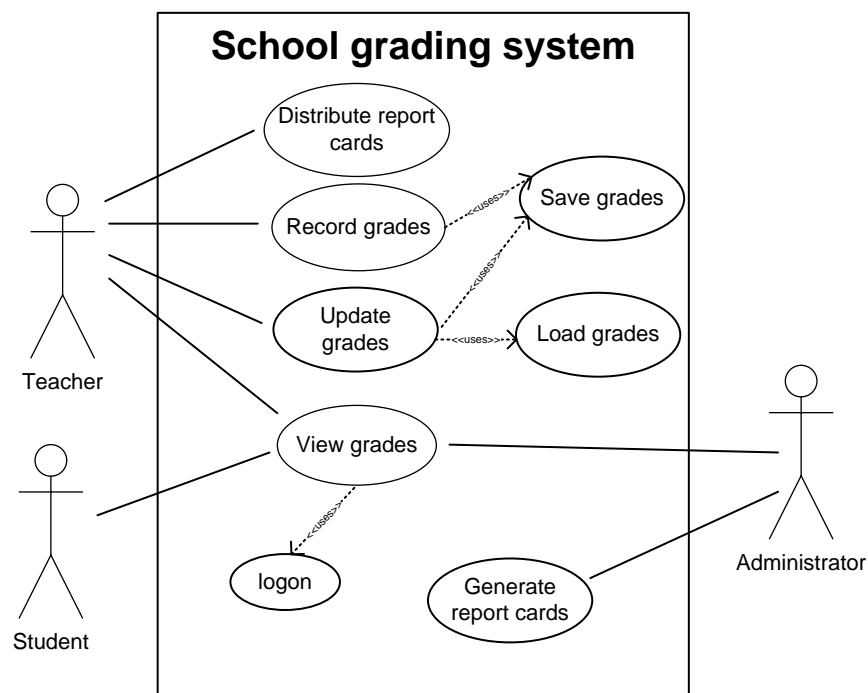
[Question adapted from: Jason T. Roff, *UML A Beginner's Guide*, McGraw Hill, 2003]



Example 2

In a school grading system, a teacher can record and update grades. Recording grades includes saving them while updating grades includes loading and saving the grades. The system administrator can generate report cards however these are distributed by a teacher. A teacher, and administrator and a student can view grades, which includes logging in the system. Sketch a Use Case diagram to depict this scenario. Hint: Make use of the *uses* relationship.

[Question adapted from: Jason T. Roff, *UML A Beginner's Guide*, McGraw Hill, 2003]



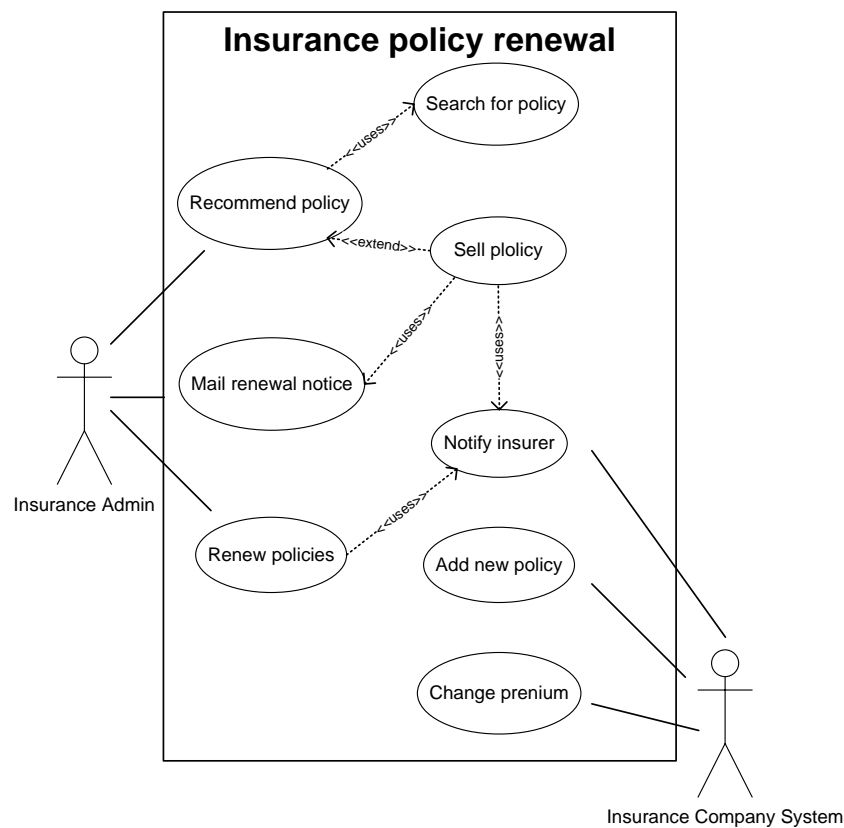
Example 4

One of the tasks that an insurance company is required to do is that of notifying their clients to either renew a policy or else to sell new policies. This can be seen as a subsystem of an insurance company. Draw a use case diagram to include the following requirements of this insurance sub system:

- To notify the insurance company system of all sales.
- To receive notifications of new policies and the criteria that enable them to be matched to members' needs from the insurance company system.
- To receive notifications of premium changes from the insurance company system.
- To generate on a weekly basis renewal notices for all policies one month before they are due for renewal. (These will be mailed out to the policy holders).
- To renew a policy.
- To notify the insurance company system of all renewals.
- To calculate the insurance premium for the recommendations, sales and renewal notices.

Hint: Make use of both the *uses* and *extend* relationships.

[Question adapted from: Bennett S, Skelton J, Lunn K., *UML*, Schaum's Outlines, 2005]



Class diagrams

Overview

- A class diagrams consists of classes and their relationships to each other.
- Classes can contain:
 - Attributes
 - Operations (methods)...or else consists of the class name only

A very simple example of a basic class diagram

```
public class Person {  
    private String name;  
    private String surname;  
    private int age;  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getSurname() {  
        return surname;  
    }  
    public void setSurname(String surname) {  
        this.surname = surname;  
    }  
    public void displayDetails() {  
        System.out.println("NAME      - " + name);  
        System.out.println("SURNAME - " + surname);  
        System.out.println("AGE      - " + age);  
    }  
}
```

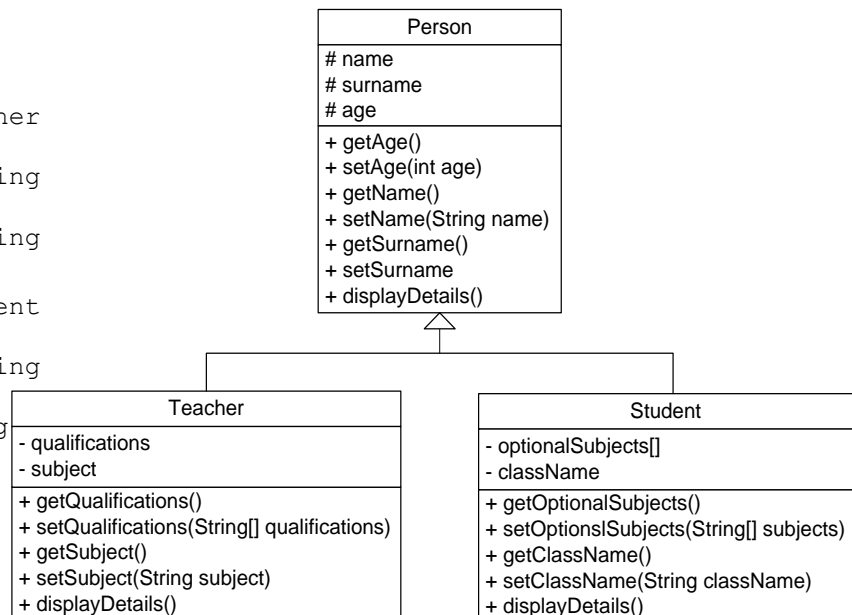
Person
- name - surname - age
+ getAge() + setAge(int age) + getName() + setName(String name) + getSurname() + setSurname + displayDetails()

Generalisation (Inheritance)

```
public class Person {
    protected String name;
    protected String surname;
    protected int age;
    .
    .
    .
}

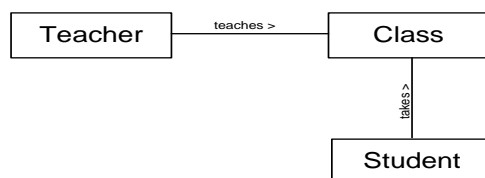
public class Teacher
extends Person {
    private String
    qualification;
    private String
    Subject;
}

public class Student
extends Person {
    private String
    className;
    private String
    optionalSubjects;
    .
    .
    .
}
```



Relationships

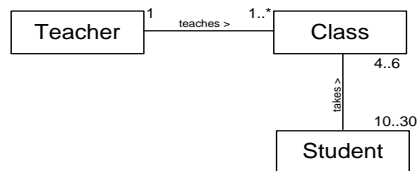
- Classes can relate to each other with a line and an association name.



- A Teacher teaches a Class.**
- A Student takes a Class.**

Multiplicity

- Indicate how many instances of one class can relate to one other instance of another class.
- Multiplicity is a single number or a range of numbers



- A **Teacher** *teaches* a 1 or more **Classes**.
- A **Student** *takes* 4 to 6 **Classes** and each **Class** has 10 to 30 students.