STADIO

Object-Oriented Programming
OOP152

Note

The content of the STADIO Study Guides and teaching documents is not intended to be sold or used for commercial purposes. Such content is in essence part of tuition and constitutes an integral part of the learning experience, regardless of mode.

This Study Guide should be read in conjunction with prescribed readings that are downloadable for free from the STADIO Online library. You can find the list with specific instructions in the References section of this Study Guide. We suggest that you download these prescribed resources before you start reading this Study Guide. To access some of the eBooks you must also download and install the latest version of the Adobe Digital Editions eBook reader. This is a once-off process and will enable you to read all subsequent books requiring this reader without downloading it again. You will get clear instructions on how to download this software when this is necessary.

Links to websites and videos were active and functioning at the time of publication. We apologise in advance if there are instances where the owners of the sites or videos have terminated them. Please contact us in such cases.

A Glossary of terms is provided at the end of this study guide to clarify some important terms.

Any reference to the masculine gender may also imply the feminine. Similarly, singular may also refer to plural and vice versa.

Table of contents

Module purpose and outcomes

Good programming skills are one of the main competencies required for employment in any computing discipline. The object-oriented paradigm is extensively used in industry to develop large information systems.

This module covers the following topics:

1. Overview of imperative concepts in Java
2. More complex methods
3. Objects
4. Classes
5. Objects of objects

Module purpose

This module aims to impart both the theory and practice of this paradigm and further develop the students' overall programming skills by introducing a second programming language, Java.

You must have completed the "Computational Thinking and Introduction to Programming" module as a prerequisite before attempting this module. This module builds on the first principles of programming already covered. It focuses on designing and implementing larger, more complex programs through a widely used industry language. In addition to providing the corresponding Java syntax for the constructs previously covered in the Python programming language, this module introduces object-oriented programming concepts, such as encapsulation and information hiding, data objects and inheritance, as well as additional imperative programming concepts and constructs, such as garbage collection and reference types. Practical experience using an object-oriented language is included in computer-based programming assignments.

Module outcomes

Upon successful completion of this module, you should be able to:

1. Understand and apply basic programming constructs in the context of the Java programming language by converting hand-crafted algorithms into Java code.
2. Use an integrated development environment to write, compile, run, and test simple object-oriented Java programs.
3. Validate input in a Java program and design test procedures for these.

4. Identify and fix defects and common security issues in code.
5. Apply basic object-oriented programming constructs in Java to various problems.
6. Learn new programming languages in future modules more easily, given the programming skills developed in this module.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/

Topic 1
Overview of imperative concepts in Java

1.1 INTRODUCTION

This topic relates to the following module outcomes:

1. Understand and apply basic programming constructs in the context of the Java programming language by converting hand-crafted algorithms into Java code.
2. Use an integrated development environment to write, compile, run, and test simple object-oriented Java programs.
3. Validate input in a Java program and design test procedures for these.
4. Identify and fix defects and common security issues in code.

In this topic, you will gain knowledge in the following areas:

1. Displaying "Hello World" in Java
2. Using an IDE
3. Compilation and execution of Java programs
4. Variables and operators
5. Void methods
6. Conditional statements and logical expressions
7. Accepting (reading) input and displaying (writing) output

1.2 DISPLAYING "HELLO WORLD" IN JAVA

1.2.1 How to display messages in Java

Every computer programming language has a way or ways of to display messages onto the screen to enable interaction between your program and users. You may need to prompt users for action (such as asking them to enter input), display error messages, or even show them the output of some computation. Java supports two types of user interfaces – command line interface (CLI) and graphical user interface (GUI). The CLI is sometimes called the console user interface or text-based interface. However, the commonly used term is CLI. Java provides different methods (functions) to allow you to display messages. The

commonly used method to display messages on a CLI in Java is the System.out.println() or System.out.print(). Using these methods, you type the message you want displayed inside the brackets in double quotes. For example, System.out.println("Good morning") or System.out.print("Good morning". The difference between System.out.println() and System.out.print() is that System.out.println() moves the cursor to the next line after displaying the message while System.out.print() does not.

Example

For instance, the following lines will display the messages on different lines.

```
System.out.println("Good morning");
System.out.println("How are you?");
```

Output

```
Good morning
How are you?
```

On the other hand, using System.out.print() would display both messages on the same line.

```
System.out.print("Good morning");
System.out.print("How are you?");
```

Output

```
Good morningHow are you?
```

Note: If you use System.out.print() and want to insert space between the two messages, you can insert space after the first message before closing the double quotes, i.e. System.out.print("Good morning "); Alternatively, you can use the tab character (\t), i.e. System.out.print("Good morning\t");

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 3-4

1.2.2 Displaying Hello World: A complete example

This section teaches you how to display the "Hello World" message by writing a complete Java program. However, you need to know a few basics about Java before we can write the program. You must note the following key points about Java:

1. Java is a fully object-oriented programming (OOP) language. All Java code must be written inside a container called a class or interface (not a user interface). For now, know that before you can write any code to whatever you want in Java, you must create a class inside which you will write the code.
2. Java is case-sensitive. For instance, the words "System" and "system" are different in Java because the cases are different.
3. A terminator or semicolon signifies the end of a Java statement (sentence) (;).
4. We can create fields (variables) and behaviour (methods/ functions) inside a class.
5. You can only declare variables inside a class but outside of methods. All code to perform any action must be inside a method. (NB: The word **method** is synonymous with **function.** However, method is the preferred word in OOP.
6. The main method is a Java program's entry point. For methods defined outside of the main method to be executed, they must be invoked (called) inside the main method.
7. You can create classes that either contain a main method or not. We shall refer to a class that contains a main method as a **Driver** class. Henceforth, whenever you shall see the term "Driver class" in this study guide, take it to mean a class with a main method.

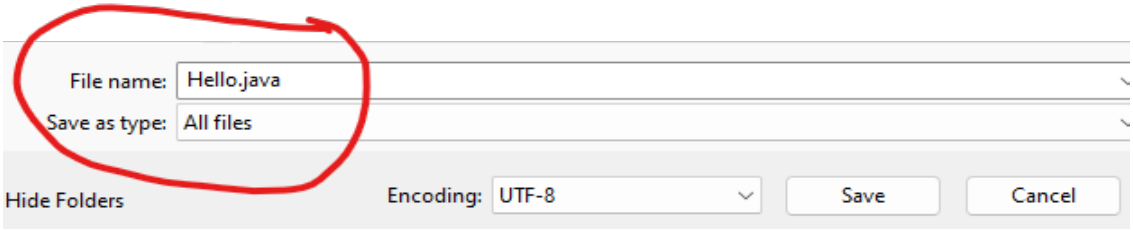
Having looked at these points, let us now turn to the example of a Java program that displays the "Hello world" message.

Example

Step 1: Open a text editor (e.g. Notepad) and type the lines below

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

Step 2: Create a folder on the Desktop and name it to Java (NB: The folder's name can be anything. However, for it to be easier for you to follow the instructions that will, I recommend you use the same name as used herein).
Step 3: Save the file as Hello.java in the Java folder you created in Step 2 above. (NB: Make sure you select All files on the Save as type option on the Save As dialog box, as shown below.)



Note: If you've done everything right, check the file in the folder and its file type should be JAVA file.

1.2.3 Executing Java programs on the command prompt in Windows

You can run a Java program on the command prompt. Let us see how you can run your Java program that you wrote to display "Hello world".

Step 1: First, run the command prompt

Step 2: Change the directory to the folder Java that you created on the Desktop. To change the directory, type the command cd Desktop\Java

Step 3: To compile your Java program, type the command javac Hello.java (Note: If it compiles successfully, you must not see any errors. Check in the Java folder on the Desktop and you will see another file created, named Hello.class).

Step 4: Now run the program to see the output by typing the command java Hello. You will see the output of your code. It will display the message "Hello world!" on the same command prompt screen. (Congratulations! You have successfully executed and run your first Java program.)

If you missed any steps, you can check the screenshot below:

```
C:\Users\HP>cd Desktop\Java

C:\Users\HP\Desktop\Java>javac Hello.java

C:\Users\HP\Desktop\Java>java Hello
Hello world!

C:\Users\HP\Desktop\Java>
```

Note

Below are some common mistakes that you need to avoid. Making any of these mistakes leads to a syntax error, and your code will not compile.

1. Forgetting to terminate a statement.
2. Forgetting that Java is case-sensitive. Remember the S in System.out.println() is uppercase.
3. Forgetting to close all open braces. For every opening brace { that you have, you need a corresponding closing brace }.

Activity

(a) Write a Java program that displays greetings in your native language. You should have at least two print statements.

(b) Execute and run your program on the command prompt.

1.3 USING AN IDE

1.3.1 What is an IDE?

IDE stands for Integrated Development Environment. It is software that integrates various developer tools (such as a source code editor and debugger) into one GUI. IDEs reduce development time by eliminating the need to configure utilities manually.

Recommended reading

Read more about IDEs on the link below.
https://www.redhat.com/en/topics/middleware/what-is-ide

Java has several IDEs, such as NetBeans, Eclipse, BlueJ, IntelliJ IDEA, JDeveloper, jGRASP, JCreator, DrJava, Greenfoot, Anjuta, Eclipse Che, JBuilder and MyEclipse. However, in this module, you will use NetBeans by Apache.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 5-6

Figure 1.1 This is a screenshot of the Apache NetBeans 12.6 IDE. As you shall soon see, executing and running programs is easier with an IDE than using the command prompt.

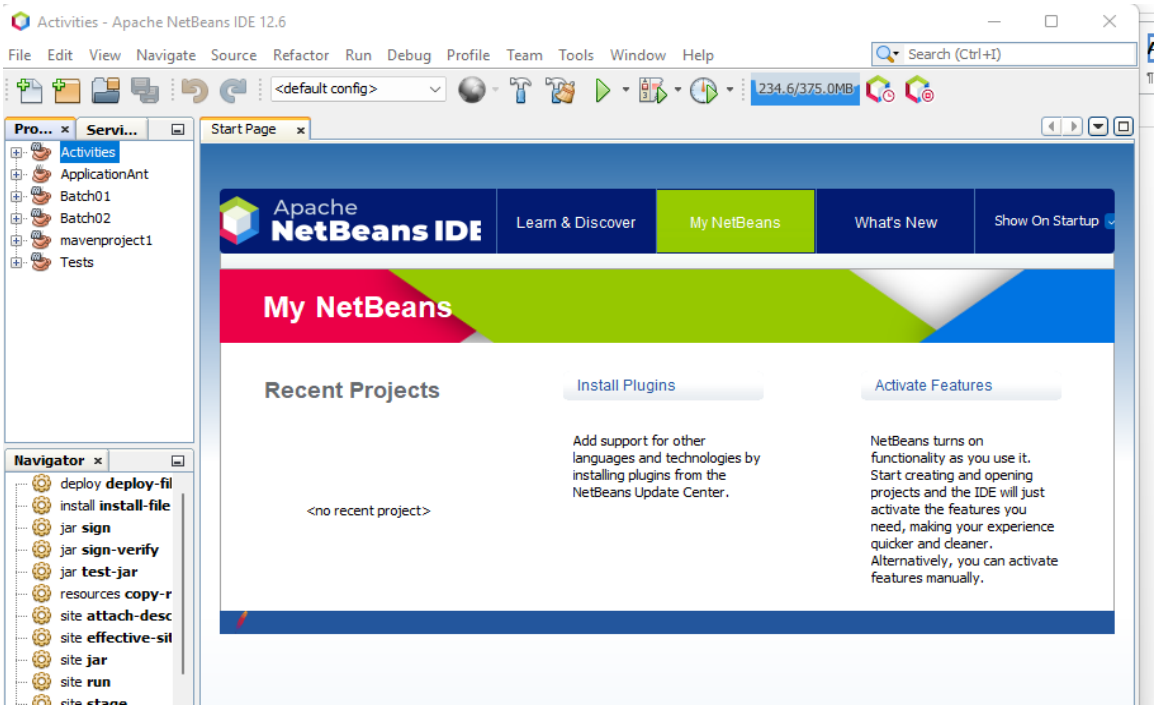


Figure 1.1 Screenshot of the Apache NetBeans 12.6 IDE

All the Java programs that you create in NetBeans exist inside a package. Think of a package as a folder containing Java classes and interfaces. If you do not create your own package, NetBeans creates a default package where your code will be stored. You must always create your own packages instead of using the default ones. Let us now see how we can create a Java project, package and class in NetBeans. Follow the steps below:

Step 1: Create a Java project by clicking **File** -> **New Project**

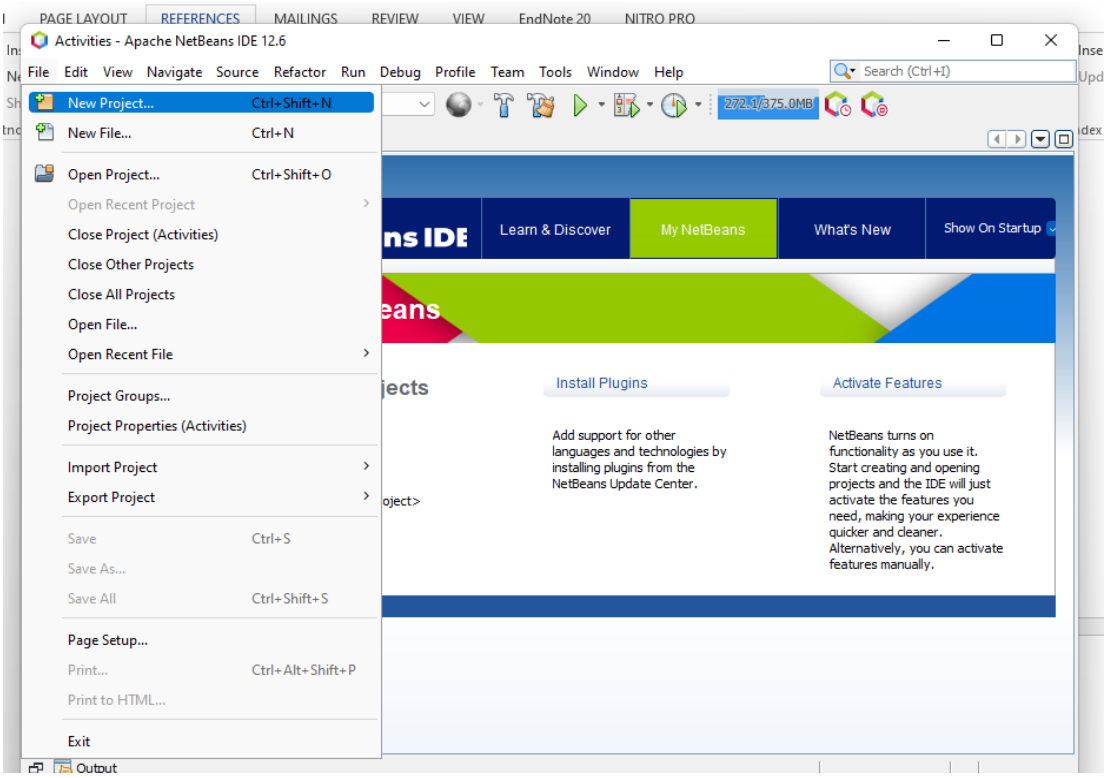


Figure 1.2 Creating a Java Project in NetBeans

Step 2: Select Project Type. You can select **Java with Maven** or **Java with Ant** under categories, and **Java Application** under Projects. For consistency, I suggest you choose Java with Ant and click **Next** as shown in Figure 1.3.

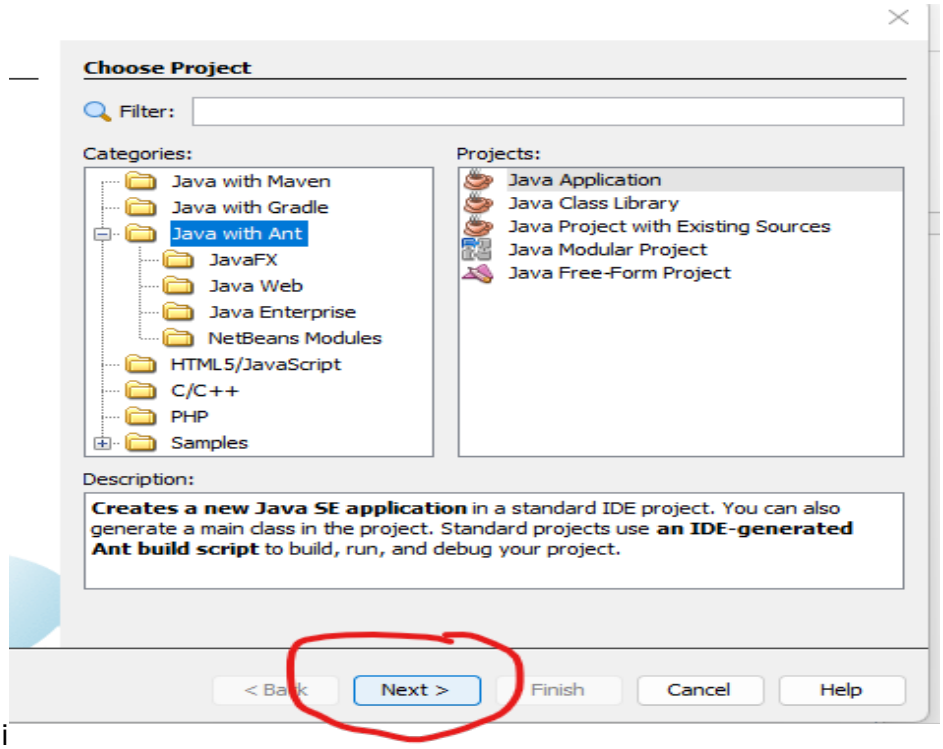
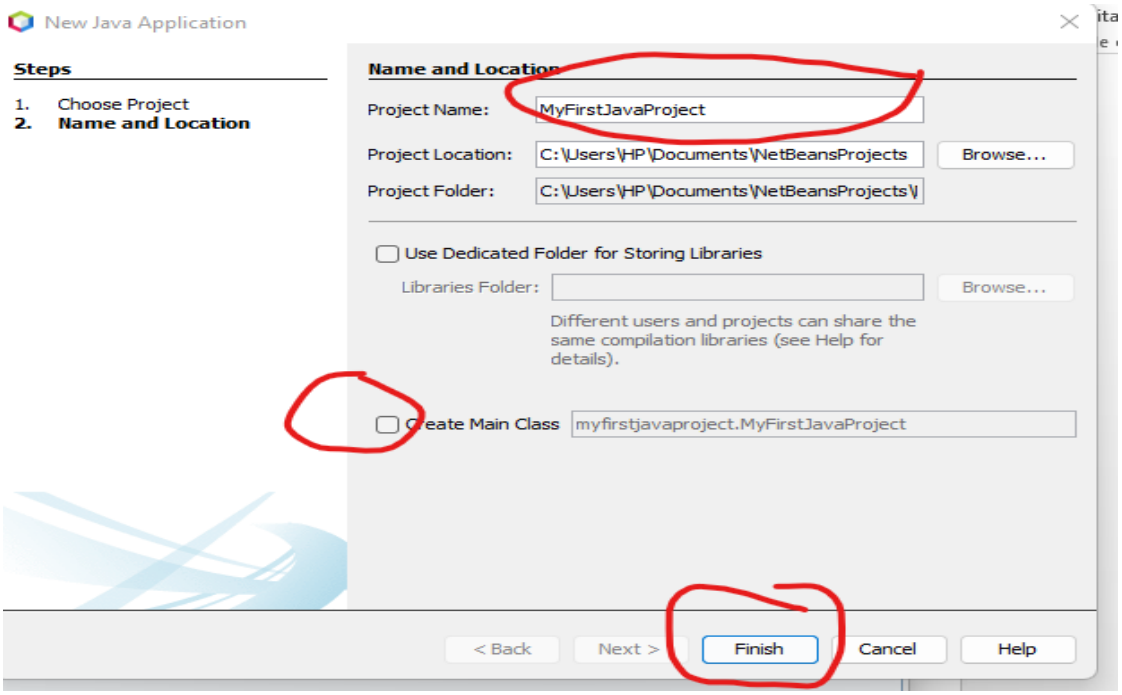


Figure 1.3 Selecting a Project Type

Step 3: Give your project a name (MyFirstJavaProject), uncheck the **Create Main Class** checkbox and click **Finish.** Note that your project name cannot contain the space character or any other special characters. You may change the location of your project as necessary by clicking on the **Browse** button.



Now your project is created, and you should see it under the **Projects** tab (Figure 1.4). It has two folders (Source Packages and Libraries). All your source code will be under Source Packages. Remember, you must not write your code in the default package! Always create your own packages.

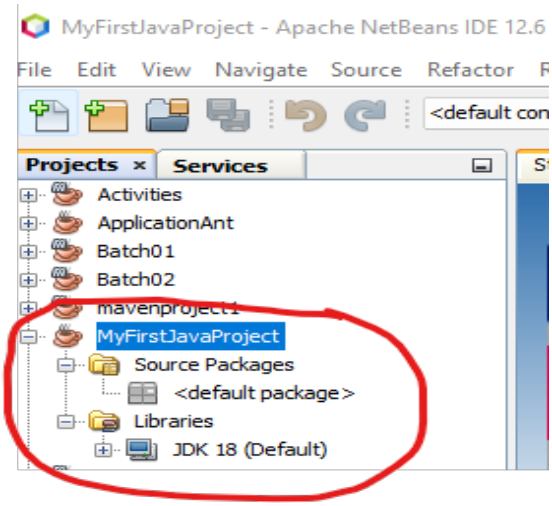


Figure 1.4 Project created

Step 4: Create a package named week1. To do so, right-click on **Source Packages -> New -> Java package.** Name your package and click **Finish.** Your package names should be in lowercase.

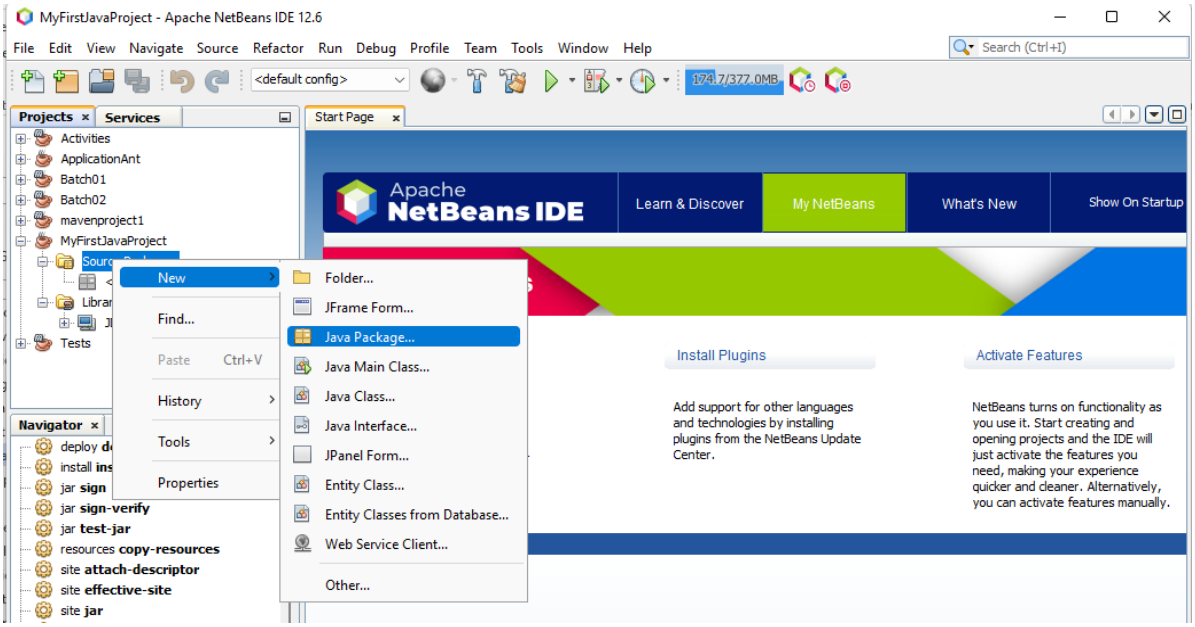


Figure 1.5 Creating a package

You should now see the package name under Source Packages, as shown Figure 1.6.

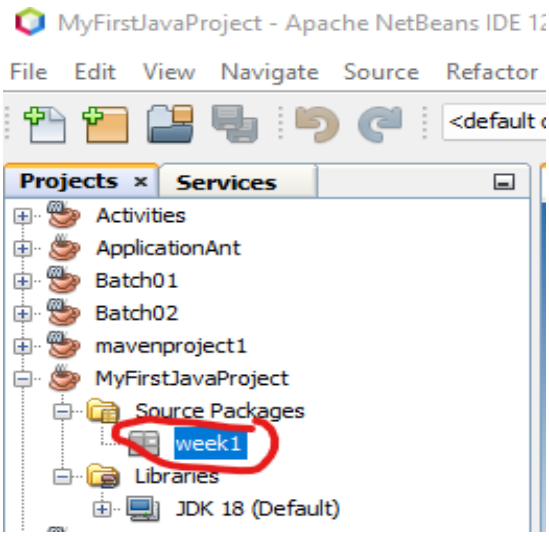
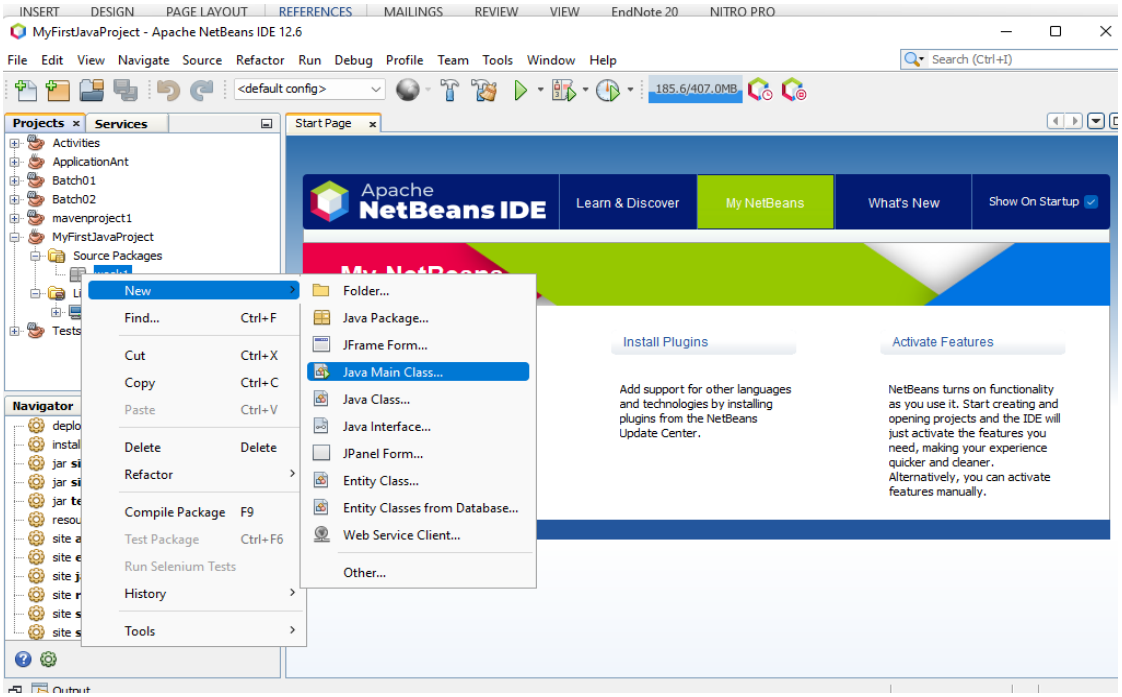
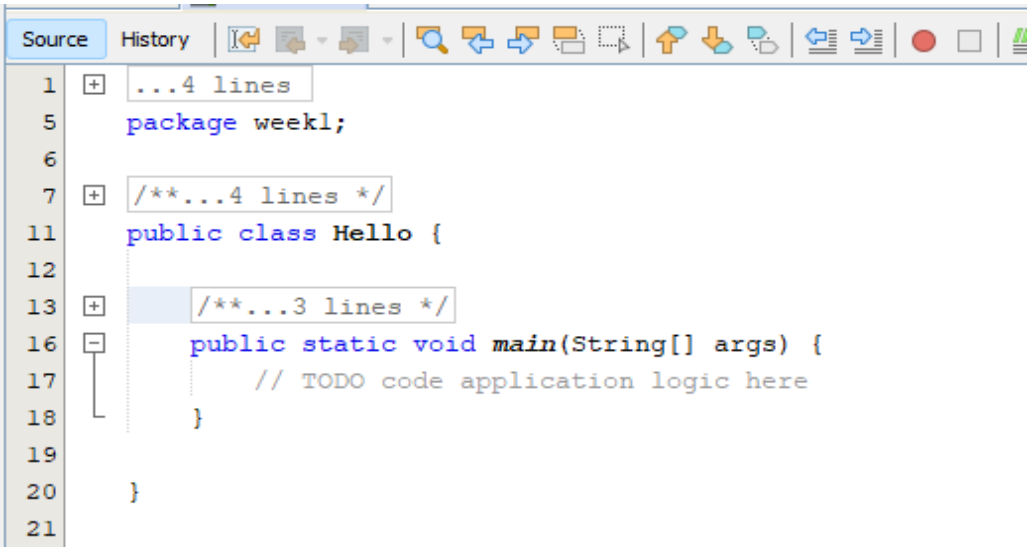


Figure 1.6 Package created

Step 5: Create a Java main class (hereafter, interchangeably referred to as Driver class) and name it Hello. To create the class, right-click the package week1, then **New -> Java Main Class** and click it. If you cannot see the Java Main Class, click Other and type main in the filter area, and it will appear. Name the class as Hello.



Step 6: Click **Finish.** The class is created along with the main method.



Inside the main method, type code to display the message "Hello world" as you did previously.

1.3.2 Compilation and execution of Java programs

To compile and execute a Java program in NetBeans, you can click the **Run Project** button, highlighted in Figure 1.7. You can also press the F6 key to run your project.

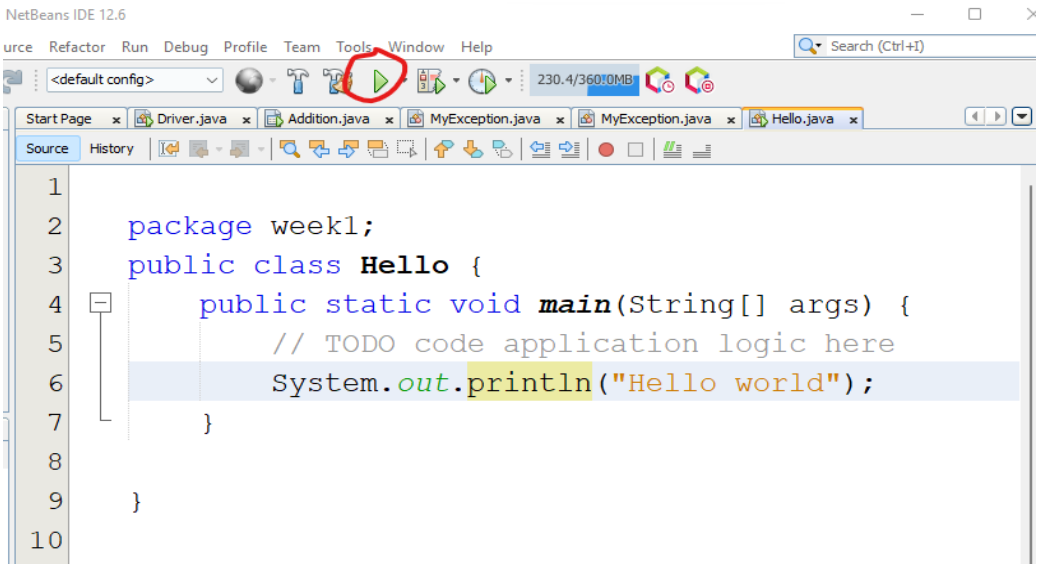


Figure 1.7 Compiling and executing a Java program

However, sometimes you could be having multiple Java main classes in your project, but you only want to run one file. In that case, you must right-click on any white space on the file and click **Run File** as indicated in Figure 1.8.

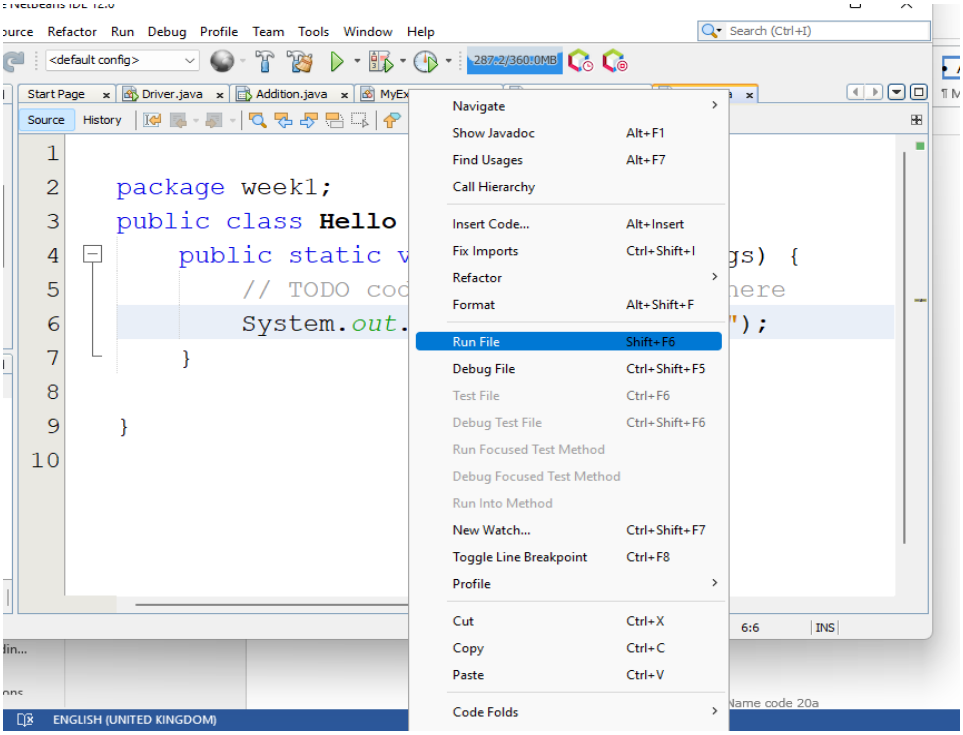


Figure 1.8 Compiling and executing a Java file

The output will be displayed on a console window, as shown in Figure 1.9.

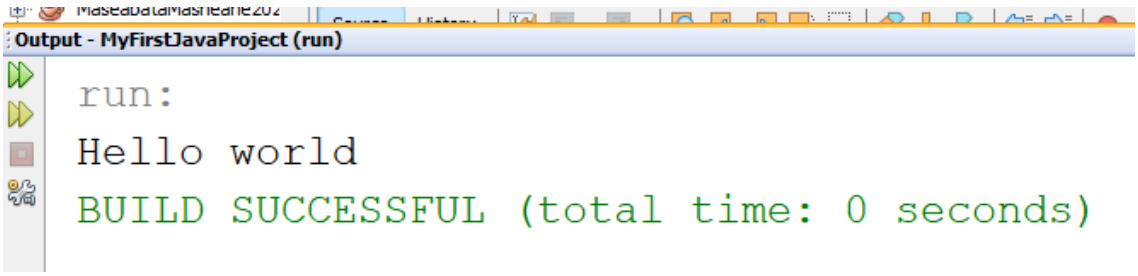


Figure 1.9 Output Window

1.4 VARIABLES AND OPERATORS

1.4.1 Variables

Useful computer programs often accept inputs and process them to generate outputs. Inputs could be entered by users or read from the environment using sensors. In some instances, inputs could come from other programs. A computer must store the data (inputs, outputs or intermediate results) to enable retrieval throughout the cycle of the program. A variable is a named memory location inside a computer that is used to store data. As the name suggests, the data stored can vary (change). A variable can store data of a particular type or subtype only. For instance, if a variable is meant to store numbers, you cannot use it to store names.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 60-61

In Java, all variables must be declared before use. Declaring a variable entails specifying the type of data that can be stored in the variable and the name of the variable. The general syntax of declaring variables in Java is:

Data_type identifier_name;

Java has primitive and non-primitive data types. Primitive data types are also called intrinsic data types in some languages. They are part of the Java programming language.

Data Type	Length/ size (bits)	Minimum	Maximum	Description
byte	8	-2^7	2^7-1	It is used to store whole numbers that are at most 8 bits long, that is, between -128 and 127
short	16	-2^{15}	$2^{15}-1$	It is used to store whole numbers between -32 768 and 32 767
int	32	-2^{31}	$2^{31}-1$	Used to store integers (whole numbers) between -2 147 483 648 and 2 147 483 647
long	64	-2^{63}	$2^{63}-1$	Used to store whole numbers between -2^{63} and $2^{63}-1$
float	32	3.4e-038	3.4e+038	It is used to store fractional numbers. It has a low level of precision compared to double. **Note: All fractional literals/ constants in Java are used to be double. If you want to indicate a literal as float, suffix it with F or f, e.g. 2.3f, 0.5F**
double	64	1.7e-308	1.7e+308	It is used to store fractional numbers. It has a high level of precision compared to float.
char	16	0	65,535	It is 16-bit Unicode character. This data type is used to store characters. A character literal must be enclosed in single quotes, e.g. 'a', '3', 'W'.
boolean				It is used to store either **true** or **false.**

In Java, a String is a class and is used to store a sequence of characters. In addition to the primitive data type, you can also create your own data types. For instance, if you define a class or an interface, it is a user-defined data type, and you can declare variables of that type.

Example

```
int firstNumber;
float salary;
double fees;
char myChar;
```

boolean result;
String name;

Rules for naming identifiers

1. It cannot start with a digit
2. It cannot contain any special characters or space
3. It must not be a Java reserved word/ keyword
4. It can start with a $, _ , or a letter of the English alphabet.

Activity

For each of the following, state whether or not it is a valid identifier in Java. Give at least one reason for your answer

(a) floAt
(b) 2num
(c) num2
(d) _____myName__
(e) myFees%balance

Initialising Variables

After declaring a variable, you can set a value to it. The two can be split or combined. The modern version of Java allows you to use underscores to group digits for ease of reading. However, the underscores must not be next to a period (dot).

Example

Examples of variable initialisations:

(a) int a = 1_000_234;
(b) float x = 0.031_345f;
(c) double y = 234.900_345_874;
(d) int number = 3452345;
(e) String name = "Stadio University";
(f) char grade = 'A';
(g) boolean pass = true;
(h) short size = 6;
(i) byte z = 6;

Note

NB: A float literal must always be suffixed with an f or F, otherwise, it is assumed to be a float. This means that any number with a decimal point in Java is assumed to be a double, unless it is suffixed with an F or f to show that it is a float.

While in everyday life we group numbers into threes for ease of reading, Java does not allow that. You have to type the number without a space in between. If you want to group the digits, use underscores instead.

NB: When using underscores to group digits, make sure the underscore is NOT next to a dot (.), otherwise the compiler will give an error. For example, the following is incorrect:
double myNum = 92._094_465;
float someFloat = 123_.093_23;

1.4.2 Casting

Converting data from one type to the other is called casting. For instance, data of type int could be converted to float or double any other numeric type. Casting can also be called "type conversion". There are two forms of casting – implicit and explicit. Implicit casting occurs automatically as it is done by the compiler. It occurs when you are promoting data from a narrower type to a wider one.

For instance:
short s = 3; //variable declaration and initialisation
int i = s; //s is short and is being stored in int, hence, it is implicitly cast to int
However, the following statement will give an error:
s=i; //error

This is because int is broader than short and storing an int in a short could lead to data loss. Consequently, the compiler will not perform automatic conversion. Instead, you have to perform **explicit casting** by indicating the data type that you want to convert to.

The syntax for explicit casting is as follows:
Variable1 = (datatype) variable2;
Using the above example of short s and int i, we would perform explicit casting as follows:
s=(short)i; //explicit casting of int to short. It may lead to data loss

Rules for casting

1. The data types must be compatible. E.g. any numeric type could be cast to any other numeric type.
2. If the type you are casting to is wider, the casting occurs automatically (implicit casting).
3. If the type you are casting to is narrower, you have to perform explicit casting. Explicit casting may lead to data loss if the actual data cannot fit in the destination variable.
4. Float is implicitly cast to double; while double can only be explicitly cast to float.

1.4.3 Operators

Java provides a rich set of operators for variable manipulation. These operators are grouped into arithmetic, relational, logical, assignment, bitwise and miscellaneous operators.

Arithmetic Operators

Arithmetic operators are used in the same way as in algebra.

Table 1.1 Arithmetic Operators

Operator	Description	Examples
+ (Addition)	Used for addition. However, if used with strings, it concatenates the strings.	3+5; Num1+num2;
- (Subtraction)	Subtracts the right operand from the left one	3-5; Num1-num2;
* (Multiplication)	Multiplies values on either side of the operator	3*5; Num1*num2
/	Divides the left operand with the right operand	3/5
% (Modulus)	Returns the remainder of dividing the left operand with the right operand	5%2 returns 1

++ (increment)	Increases the operand's value by 1.	int a=3;
		a++; //post
		++a; //pre

-- (decrement)	Reduces the operand's value by 1	int a=3; a--; --a;

Relational Operators

Relational operators determine the relationship between two operands or values. They return a Boolean value (true or false) and are crucial in conditional execution (as you shall see later).

Table 1.2 Relational Operators

Operator	Description	Examples
== (equal to)	The equality sign returns true if both operands are equal; otherwise it returns false	4==5 3==3 x==y
!= (not equal to)	Returns true if the two operands are not equal; otherwise, it returns false	3 != 4 (true) 4 != 4 (false)
> (greater than)	Checks whether the left operand is greater than the right one. It returns true if the left operand is greater	3 > 5 (false) 7 > 4 (true)
>= (greater than or equal to)	Checks whether the left operand is greater than or equal to the right one and returns true if that is the case.	5 >= 3 (true) 3 >= 3 (true) 2 >= 3 (false)
< (less than)	Checks whether the left operand is less than the right one. It returns true if the left operand is lesser.	2 < 3 (true) 3 < 2 (false)
<= (less than or equal to)	Checks whether the left operand is less than or equal to the right one and returns true if that is the case.	2 <= 3 (true) 3 <= 2 (false)

Logical Operators

Logical operators work on Boolean operands.

Table 1.3 Relational Operators

Operator	Description	Examples
&	The logical AND operator evaluates the truth value of two Boolean operands. Returns true if both operands are true.	(3 > 2) & (2 != 3) returns true
|	The logical OR returns true if at least one of the operands returns true.	(3<2) | (5 >2) returns true

^ (logical XOR or exclusive OR)	Returns true if both operands have different Boolean values, otherwise it returns false.	(3 >2) ^ (4 < 2) returns true (3 >2) ^ (4 > 2) returns false
||	The short-circuit OR, unlike the logical OR, does not evaluate the whole expression if it realises that the truth value is already determined. For instance, if the first operand is true, it returns true since at least one operand must be true for the truth value of an OR to be true.	(3 != 4) || (3 == 2)
&&	The short-circuit AND works in the same way the short-circuit OR in that, if it can determine the truth value before completing the evaluation of the entire expression, it stops. For instance, if the first operand is false, it returns false for the entire expression.	(3 != 4) && (3 == 2)
! (logical Unary NOT)	As explained earlier	
== (equal to)	As explained earlier	
!= (Not equal to)	As explained earlier	

Assignment Operators

Table 1.4 Assignment Operators

Operator	Description	Examples
=	Assigns (stores) the value on the right in the variable specified on the left	C=x+y; D = 6;
+=	Takes the value on the right, adds it to the contents of the variable on the left and updates the value in the variable on the left	C += D; takes the value in D, adds it to the value in C and stores the new value in C. C += D + 6;

-=	Takes the value on the right, subtracts it from the contents of the variable on the left and updates the value in the variable on the left	C -= B is equivalent to C = C - B
*=	C *= C * B is equivalent to C = C*B	X *= 5
%=	A %= 3 is equivalent to A = A % 3	

Miscellaneous Operators

The ternary operators operates on three operands. Its syntax is:

dataType variable = (expression) ? Value if true: value if false.

This is a short form of the if statement (covered in section 1.6). The statement

int i = (3 > 5)? 1:0;

will store 1 in i if the condition (3 > 5) is true, otherwise it stores 0.

1.5 VOID METHODS

1.5.1 The structure of a Java program

First, let us look at the structure of a Java program before looking at void methods. Java code must exist in a package. You can think of a **package** as a folder in Java. Optionally, you can have **import** statements to shorten your code. Java is a fully object-oriented programming (OOP) language; hence, you can only write code inside containers called classes or interfaces. The general structure of a Java program is as follows:

```
package statement //mandatory
import statement(s) //optional
class ClassName{
        instance & static variables //optional
        instance and static methods //optional
}
```

Note

Unlike other languages like C and C++, Java does not have "global" variables that are declared outside classes. All variables and methods must be encapsulated inside classes or interfaces.

1.5.2 Methods in Java

Classes are the basic building blocks of Java. A class has **state** and **behaviour**. Variables represent the state of a class, while the behaviour is represented by methods. Methods are called functions in other languages. The general syntax of a method signature in Java is as follows:

```
Modifier returnType methodName (parameter list)
{
//method body
}
```

Modifier

A modifier could be access or nonaccess. Access modifiers specify which classes can access that method. Examples include public, protected, private or default package-level access. Nonaccess modifiers include static and final. A static method is a class method while a final method cannot be **overridden**. A modifier is optional in Java.

returnType

Ideally, a method returns a value as its output. A return type indicates that the data type of the returned value. If a method does not return a value, its return type is void. A method with a return type of void does not need a return statement. A return type is mandatory.

methodName

A method name must meet the same rules for naming identifiers/ variables. Every method must have a name.

Parentheses ()

The parentheses/ brackets () are mandatory, regardless of whether the method has parameters or not.

Parameter list

Parameters are optional. They represent inputs to a method.

Braces { }

Braces are mandatory. All the code associated with a method must be written inside the curly braces. If a method returns a value, the last statement in that method must be a return statement.

1.5.3 void methods

A void method is a method that does not return any value, not even zero. The commonest void method is the main method, which is the entry-point of program execution. The main method looks like this:

```
public static void main(String[] args) {
//source code
    }
```

Example

```
package week1;
public class Example {
  void display()
  {
     System.out.println("Hello there");
     System.out.println("How are you today?");
  }
}
```

Example

```
package week1;
public class Example {
  void display(String name)
  {
     System.out.println("Hello "+name);
     System.out.println("How are you today?");
  }
}
```

Note: This is an example of a parameterised method. Whenever this method is invoked, a string must passed as an argument. That string represents a name that will be displayed along with the "Hello" message. For instance, if John is passed, the message will appear as "Hello John"

Activity

(a) Create a class called Person with a method that displays the message "This is a Person class".

(b) In the same class, create another method with the same name as the one you created above, but this one should have a parameter of type string, and should display a customised message.
(c) Visit the URL
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html to read more about access modifiers in Java. Which of the access modifiers is the most restrictive? Write down your answer.

1.6 CONDITIONAL STATEMENTS AND LOGICAL EXPRESSIONS

As you program, you might want some code to be executed based on certain conditions. For instance, you might want to display an appropriate message to a user based on whether they are below 18 years or not, or you might want to determine a student's grade based on their mark and the university's grading scale. Java provides constructs for conditional execution. Let us learn about these.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 71-88

1.6.1 The if statement

The if statement executes a piece of code only when the specified condition is met. The condition must be a logical expression that evaluates to either true or false but not both. The syntax of an if statement is as follows:

```
if(condition)
{
//statement(s) to be executed
}
```

Example

```
int age = 18;
if(age > 18)
   System.out.print("You are eligible to vote");
```

```
if(age < 18)
{
System.out.print("Sorry, you are too young to vote");
System.out.print("Please try in the next time");
}
```

Note: The {} are optional if you only have one line to be executed as the body of your if statement, otherwise, the {} are mandatory. However, you are advised to always use curly braces to avoid mistakes.

Note

A common mistake made by students is using a = in an if statement to check for equality, instead of ==. Always be on the lookout to ensure you use the right operator when checking for equality.

1.6.2 The if…else statement

The if..else statement is used to specify what should happen if the condition does not evaluate to true. Note that you cannot have an else without a preceding if, and an else cannot have a condition.

Syntax:

```
if(condition)
{
//one more statements
}
else{
//one more statements
}
```

1.6.3 The if …else if… statement

Sometimes you might have multiple conditions to check for. In that case, you cannot use if..else. The first condition should be associated with an if, then the other conditions with else if.

Syntax:

```
if(condition){
//statement(s)
}
else if (condition){
//statement(s)
}
else if (condition){
//statement(s)
```

```
        }
else{
//statement(s)
}
```

Example

The following program displays the day of the week based on the day number.

```
package week1;
public class Example {
  void weekDay()
  {
     int day=6;
     if(day==1)
        System.out.println("Day 1 is Sunday");
     else  if(day==2)
        System.out.println("Day 2 is Monday");
     else if(day==3)
        System.out.println("Day 3 is Tuesday");
     else if(day > 3)
        System.out.println("It is a day after Tueday");
  }
}
```

1.6.4 Nested if statements

Nesting if statements means writing an if another if or else statement. The syntax is:

```
        if(condition)
        {
                if(condition)
                {
                //statement(s)
                }
                else{
                //statement(s)
                }
                //statement(s)
```

```
        }
else{
        //statement(s)
}
```

An inner if will only be executed if the outer if's condition evaluated to true; thus, nesting if statements is similar to having multiple conditions joined using & or &&.

You could combine conditions using the logical operators. For instance, where you want a bounded range like 12<age<18. You cannot write a condition like this in Java; rather you write it 12 < age && age <18. The following example illustrates this.

```
int day=6;
    if(day > 1 && day <6)
       System.out.println("This is a week day");
```

The above could be written as

```
int day=6;
    if(day > 1){
         if(day < 6){
                System.out.println("This is a week day");
        }
}
```

Note

You are advised to visit the following link for supplementary material on conditional statements in Java:
https://www.w3schools.com/java/java_conditions.asp

Activity

Write a program that displays the month of a year based on the month number. For instance, if the month number is 1, it should display that the month is January. If the month is outside the range 1 to 12, display the message "Sorry, the month number is invalid".

1.6.5 The switch statement

The switch statement is a compact way of representing multiple if…else if statements; thus, you can replace if …else if statements with a switch statement.

However, you can only replace an if…else if statement with a switch statement if the conditions are not ranges. A switch statement works with **cases,** where a case is a match. In other words, a case is similar to ==. The syntax of a switch statement is as follows:

```
//declare a variable and store input.
switch(variableName)
{
        case value1:
                //code to be executed
                break;
        case value2:
                //code to be executed
                break;
        case value3:
                //code to be executed
                break;
        .
        .
         case valueN:
                //code to be executed
                break;

        default:
                //code to be executed
                break;
}
```

For each case, you write the code that should be executed. The **break** keyword is optional. However, if you do not include it and the first case is matched, it will execute all the code till the end of the switch statement. Therefore, the purpose of **break** is to ensure that only the code for that case is executed. Likewise, the **default** case is optional. It is used to execute code where the value in the variable does not match any of the specified cases. You can think of it as the **else** statement in an if or if…else if statement.

Example

The following example displays messages based on the day of the week. The example uses strings. In Java, a string literal is enclosed in double quotation marks.

```
package stadio;
public class SwitchExample {
  public static void main(String[] args) {
     String day = "Monday";
     switch(day)
     {
        case "Monday":
```

```
            System.out.println("The day is Monday");
            System.out.println("It is the second day of the week");
              break;
         case "Tuesday":
            System.out.println("The day is Tuesday");
            System.out.println("It is the third day of the week");
            break;
     }
   }
 }
```

When you are dealing with numbers, you do not enclosed the value in double quotation marks. Check out the example below:

```
package stadio;
public class SwitchExample {
  public static void main(String[] args) {
     int day = 2;
     switch(day)
     {
        case 1:
           System.out.println("The day is Sunday");
           System.out.println("It is the first day of the week");
           break;
        case 2:
           System.out.println("The day is Monday");
           System.out.println("It is the second day of the week");
             break;
        case 3:
           System.out.println("The day is Tuesday");
           System.out.println("It is the third day of the week");
           break;
     }
   }

}
```

NB: After running the code above, remove the **break** keyword and run it again to notice the difference.

1.7 ACCEPTING (READING) INPUT AND DISPLAYING (WRITING) OUTPUT

Useful computer programs ideally accept input, e.g., from users, the environment, or other programs. We will focus on how to accept input from users. Java provides various classes and methods for accepting input from the keyboard.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 11, 17-28, 33-45

1.7.1 Accepting input

The Scanner class

The Scanner class is found in the java.util package and is used to accept input of different data types. You must import this package if you are going to use the Scanner class. The steps involved to use the Scanner class are follows:

1. Import the Scanner class (import java.util.Scanner;)
2. Create an object of type Scanner (Scanner sc = new Scanner(System.in;)
3. Prompt a user to enter input
4. Accept the input and store it in a relevant variable.

There are various methods used to accept the input, depending on the expected data type of the input. These methods are as follows:

Table 1.5 Scanner class methods

Method	Description
nextBoolean()	Used to read a Boolean value from the keyboard.
nextByte()	Used to read byte values from the keyboard
nextDouble()	Used to read double values from the keyboard
nextFloat()	Used to read float values from the keyboard
nextInt()	Used to read int values from the keyboard
next()	Used to read String values from the keyboard. It reads the first word only.
nextLine()	Used to read String values from the keyboard. It reads the whole line until a user presses the Enter key.

nextLong()	Used to read long values from the keyboard
nextShort()	Used to read short values from the keyboard

The BufferedReader class

The BufferedReader class reads text from a character-based input stream and inherits the Reader class. You have to import java.io.BufferedReader to use the BufferedReader class.

Note

You are advised to the visit the following link for supplementary material on the BufferedReader class in Java:
https://www.javatpoint.com/java-bufferedreader-class#:~:text=Java%20BufferedReader%20class%20is%20used,It%20makes%20the%20performance%20fast.

JOptionPane input dialog

You can also use a JOptionPane input dialog to accept input. The JOptionPane is a graphical dialog. It appears like the one below:

However, you need to import javax.swing.JOptionPane for you to be able to use the JOptionPane. The JOptionPane input dialog returns a String. If you want to accept any data type other than string, you have to perform explicit casting.

Example

The following example illustrates how you can read user input using the Scanner class.

```
package stadio;
//needed to work with the Scanner class
import java.util.Scanner;
public class SwitchExample {
  public static void main(String[] args) {
     //declare variables
```

```
    float height;
    double price;
    String itemName;
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter height ");
    height = sc.nextFloat();
     System.out.println("Enter price ");
    price = sc.nextDouble();
     System.out.println("Enter item name ");
    itemName = sc.nextLine();
  }
 }
```

Example

The following example illustrates how you can read user input using the JOptionPane class.

```
package stadio;
//needed to work with the JOptionPane class
import javax.swing.JOptionPane;
public class SwitchExample {
  public static void main(String[] args) {
    //declare variables
    float height;
    double price;
    String itemName;
    itemName = JOptionPane.showInputDialog("Enter item name");
    height = Float.parseFloat(JOptionPane.showInputDialog("Enter height "));
    price = Double.parseDouble(JOptionPane.showInputDialog("Enter price "));
  }
 }
```

Example

The following example illustrates how you can read user input using the BufferedReader class.

```
package stadio;
//needed to work with the BufferedReader class
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class SwitchExample {
  public static void main(String[] args) throws IOException {
      //declare variables
      String itemName;
     //create an InputStreamReader object
     InputStreamReader reader = new InputStreamReader(System.in);
   //create a BufferedReader object
   BufferedReader br = new BufferedReader(reader);
  //prompt user for input
   System.out.println("Enter item name");
  //read the input and store it in variable itemName
   itemName=br.readLine();
  }
}
```

1.7.2 Displaying output

Displaying output is critical to programming as it allows interaction between users and your program. You have already seen how to display messages when we displayed the "Hello world" message. There are various approaches to displaying messages, but we will focus on a few in this section.

System.out.println() – this is used to display output on a console. The argument you pass to this method will be displayed on one line after which the cursor moves to the next line.

System.out.print() – this is used to display a message and the cursor remains on the same line.

JOptionPane.showMessageDialog – this is used to display a message on a graphical message dialog. There are various overloaded versions of this message. The simplest of them takes a string argument and has the syntax:

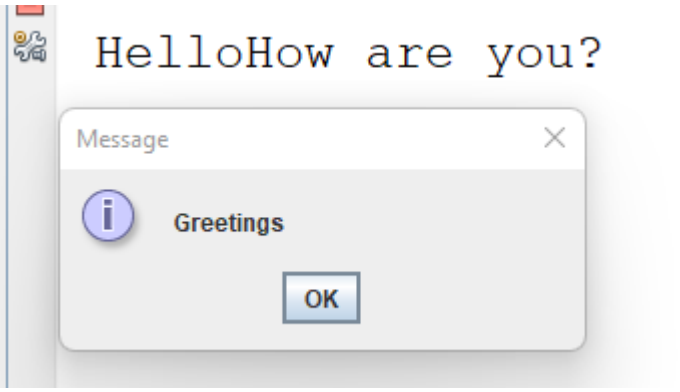
JOptionPane.showMessageDialog(null, "message to be displayed");

Example

The following examples illustrate how to display messages using the three methods discussed above and show the output.

```
package stadio;
import javax.swing.JOptionPane;
public class SwitchExample {
  public static void main(String[] args)
  {   System.out.print("Hello");
      System.out.println("How are you?");
      JOptionPane.showMessageDialog(null, "Greetings");
  }
}
```

Output



Summary

This topic covered how to display the "Hello world" message, how to execute Java programs using the Java IDE, variables and operators, void methods, the structure of a Java program, conditional statements, accepting input and displaying output. Java programs are organised as classes, inside which you can have variables and methods. There are different primitive data types in Java, include float, int, double, char byte, long and short. All variables must be declared before use. Declaring a variable tells a computer two things – the data type of the expected data and the name to use to refer to the memory location. You can use several operators to manipulate operands in Java. These include arithmetic,

assignment and relational operators. Methods are used to specify the behaviour of an object. Every method must have a return type, which is the data type of the returned value. A method that returns nothing has the 'void' return type. Conditional statements are used to execute code on condition that a certain condition evaluates to true. You use the if, if..else, if..else if and switch statements for conditional execution. A condition is an expression that evaluates to true or false, or it could be Boolean value itself. To accept user input, you can use the Scanner, BufferedReader or JOptionPane classes. However, you import appropriate packages. You can use the System.out.print(), System.out.println() and the JOptionPane to display messages.

Self-Assessment Questions

1. Write a program that accepts a user's full name and displays a custom welcome message. E.g. if the person enters **Stadio University** as their name, display the message **Welcome Stadio University.**

2. Write a program that accepts a mark from a user and displays the grade using the grading scale below:
   Table 1.6 Grading scale

Marks range	Grade
75 – 100	A
60 – 74	B
50 – 59	C
Below 50	F

[Hint: Use an if…else if statement. If a user enters a mark below 0 or above 100, display an error message that the mark is invalid. Declare a char variable to store the grade.]

3. Write a program that accepts one's sex (Female or Male) and stores it in a string variable. If the sex is male, display the message 'Greetings sir', if it is female, display the message 'Greetings madam', otherwise display "Sorry, the system could not identify this sex". [Hint: Use a switch statement].

4. Identify and correct any bugs in the following program:
   package stadio;

```
public class Test {
   public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     int age;
     String name;
     System.out.println("Enter your name ");
     name = sc.nextDouble();
     System.out.println("Enter your age ");
     age = sc.nextInt();
     System.out.println("Welcome " + name);
     if(age > OR = 18)
        System.out.print("You are an adult");
     else if (age < 0)
        System.out.println("Invalid input. Age can't be negative");
     else
        System.out.println("You are still a minor");
         }
   }
```

Topic 2
More complex methods

2.1 INTRODUCTION

This topic relates to the following module outcome/s:

1. Understand and apply basic programming constructs in the context of the Java programming language by converting hand-crafted algorithms into Java code.
2. Use an integrated development environment to write, compile, run, and test simple object-oriented Java programs.
3. Validate input in a Java program and design test procedures for these.
4. Identify and fix defects and common security issues in code.

In this topic, you will gain knowledge in the following areas:

1. Recursive methods
2. Value methods
3. Control flow constructs
4. Arrays and strings
5. Collection processing

2.2 RECURSIVE METHODS

2.2.1 Introduction

Java provides looping constructs for handling repetitive statements. These are the while, do … while, for and the enhanced for loops. A method that is defined to perform repetitive tasks using loops is called an **iterative** method. Normally, methods are invoked by other methods. However, a method can invoke itself, in which case it becomes a **recursive method.** Similar to loops, a recursive method must have a base or stopping condition specifying when to stop the recursion. This condition is specified in the form of an 'if' statement.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 127 - 141

2.2.2 Recursive void methods

A recursive method is a method that invokes itself. If that method returns void, then it is a recursive void method. Thus, a recursive void method is a method that calls itself and has a void return type. Recursive methods can return a value, just like any other method.

Example

```
package stadio;
public class Topic2 {
   public static void example(int x)
   {
      if(x==0) /*base or stopping condition*/
         System.out.println("end of recursion");
      else
      {
         System.out.println("The current value of x is :"+x);
         example(x-1); /*self-invocation of the method example - recursion*/
      }
   }
    public static void main(String[] args) {
     example(3); /*invocation of example in the main method*/
   }
   }
```

In the above example, the recursive method example has one parameter of type of int. The method checks if the input (x) is 0, in which case it displays "end of recursion", otherwise it displays the current value of x and invokes the same method passing a new value of

2.2.3 Value-returning methods

Apart from returning void, recursive methods can also return values of different types, similar to any other method in Java. The data type of the value being returned determines the return type of the method; that is, the return type of a method must be the same data type (or a subtype) of the returned value. Recursion is used to define several mathematical functions, the commonest ones being the factorial and Fibonacci series. The factorial of a non-negative integer, n, is defined as: $n! = n * (n-1)!$ This can be implemented as following.

Structure of a recursive method

```
Access-modifier non-access-modifier return-type methodName(parameters)
{
//base case
if(condition)
{
//statement(s)
}
//recursive case
Else
{
//statement(s) to be executed
}
}
```

Example

```
package stadio;
public class Topic2 {
  /*method name is factorial, return type is int,  access modifier is default, static means you can access the method via class name
  without creating an object
  */
  static int factorial(int n)
   {
      //base case, or stopping condition
     if(n == 0)
                return 1;
     /*this condition is optional. You can exclude it along with its return statement and you still get the same output*/
       else if(n ==1 )
                    return 1;
```

```
      else
          return n*factorial(n-1);
   }
    public static void main(String[] args) {
   /*invoking the recursive method without creating an object since it's static*/
     System.out.println(factorial(4));
   }
 }
```

The Fibonacci series begins with the numbers 0 1 and each successive number is the sum of its previous two numbers, leading to the sequence 0 1 1 2 3 5 8 13 ….. The base cases are in this example are n=0 and n=1. Defining this series recursively, $fib(n) = fib(n-1) + fib(n-2)$ and this function can be implemented as a recursive method as shown below.

Example

```
package stadio;
public class Topic2 {
  static int fib(int n){
      if(n==0)
          return 0;
      if(n==1)
          return 1;
      else
          return fib(n-1)+fib(n-2);
   }
    public static void main(String[] args) {
   /*invoking the recursive method without creating an object since it's static*/
     System.out.println(fib(6));
   }
}
```

2.3 VALUE METHODS

Value methods are methods that return a value. That is, they are methods whose return type is not **void.** The value returned by a value method must be utilised by the method invoking it, otherwise, there will be no point in having a value method if the returned value is not being used. Refer to section 2.2.3 for more information on value methods.

Methods have parameters. A parameter is an input to a method. Oftentimes, the words parameter and arguments are used interchangeably. However, an

argument is the actual value or variable passed to a method when it is invoked, whereas a parameter is a placeholder specified when defining a method.

2.4 CONTROL FLOW CONSTRUCTS

2.4.1 Introduction

Java supports three types of control flow statements, namely conditional statement (if and switch statements), looping statements (do, do…while, while, for and enhanced for (for-each) loop) and jump (break and continue) statements). We will cover the enhanced for loop after looking at arrays. Conditional statements were covered in section 1.6. Looping constructs allow you to write code that does a repetitive task in a smarter way that reduces the number of lines of code. A loop can be *finite* or *infinite*. Every finite loop must have a starting value, a stopping condition and an operation (increment/decrement) that modifies the starting value to approach the stopping value.

Suppose you want to display all the integers between 1 and 10. You could write 10 statements, each for displaying each number. However, suppose you wanted to display all the integers between 1 and 10 000. You could easily do this using a loop since the task involves repetitively displaying an integer, all that changes is the value to be displayed.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 89-97

2.4.2 The while loop

The while loop checks if the condition is true first before executing anything; thus, it might never execute if the initial value does meet the condition.
The syntax of the while loop is as follows:

```
while(condition)
{
//statements to be executed
}
```

A condition could be any logical expression that can evaluate to true or false. However, true and false could also be used as conditions inside the while loop. The following example illustrates a while loop to display all the multiples of 3 between 1 and 100.

Example

```
package stadio;
public class Loops {
    public static void main(String[] args) {
       int i =1; //initialisation -- starting value
       while(i<=100)
       {
          if(i%3==0) //if i is a multiple of 3
          {
             System.out.println(i);
          }
          i++; /*increment. If you remove statement, the value of i remains 1, making it an infinite loop*/
       }
    }
 }
```

The condition could be compound, such operators like &, &&, | and ||, among others. The following example illustrates a compound condition.

Example

```
package stadio;
public class Loops {
    public static void main(String[] args) {
       int i =1; //initialisation -- starting value
       while(i>=1 && i <5)
       {
          if(i%3==0) //if i is a multiple of 3
          {
             System.out.println(i);
          }
          i++; /*increment. If you remove statement, the value of i remains 1, making it an infinite loop*/
       }
    }

}
```

2.4.3 The do…while loop

It is similar to the while loop. However, it is always guaranteed to execute at least once even if the initial value does not meet the condition. This is so because the do…while loop checks for the condition at the end. The syntax is as follows:

```
do
{
        //statements to be executed
} while(condition);
```

Note that there is a terminator after the condition.

The following example displays all the multiples of 3 between 1 and 100.

Example

```
package stadio;
public class Loops {
   public static void main(String[] args) {
      int i =1; //initialisation -- starting value
      do
      {
         if(i%3==0) //if i is a multiple of 3
         {
            System.out.println(i);
         }
         i++; /*increment. If you remove statement, the value of i remains 1, making it an infinite loop*/
      }while(i <= 100);
   }
 }
```

Note

The initialisation for both the while and do…while loops must be done inside the loop, otherwise the modified value will be reset and it will never approach the stopping value.

2.4.4 The for loop

The for loop is a compact way of combining the initialisation, condition and increment/decrement operation. Its syntax is as follows:

```
for(initialisation; stopping condition; increment/decrement)
    {
       //statements to be executed
    }
```

The following example is similar to the previous ones, though it uses a for loop.

Example

```
package stadio;
public class Loops {
   public static void main(String[] args) {
      for(int i=1; i<=100;i++)
      {
         if(i%3==0) //if i is a multiple of 3
         {
            System.out.println(i);
         }
      }
   }
}
```

2.4.5 Infinite loops

An infinite loop is a loop that execute endlessly or forever. You can create one using any of the looping constructs we have looked at so far (while, do..while and for). Creating an infinite loop can be done by omitting one of the three components of a loop (initialisation, stopping condition or increment/decrement operation. If you omit an increment/decrement operation from any loop, it means the initial value will never change and therefore, it will never approach the stopping condition. Again, where the initial value is less than the stopping value, a decrement operation will move the initial value further away from the stopping condition, creating an infinite loop. Likewise, if the initial value is greater than the stopping value, an increment operation will create an infinite loop as the initial value will further drift away from the stopping value.

For the for loop, you can create an infinite loop by omitting the initialisation, stopping condition and increment/decrement operation. However, the semi-colons are still required.

```
for( ; ;)
      {
         //statements to be executed
      }
```

2.4.6 Nested loops

Nesting loops means putting a loop inside another loop. Where a loop is nested inside another, it executes until it reaches the stopping criteria before it returns control to the outer loop. For instance, suppose you want to display the half pyramid:

```
*
**
***
****
```

You can notice that there are rows and columns, and as each increase, the other also increases. Displaying output like this requires you to use two loops, one to control the row number and the other the column number. The following example illustrates nested loops to display that half pyramid.

Example

```
package stadio;
public class Loops {
   public static void main(String[] args) {
         //outer loop
      for(int row=1; row <=4;row++)
      {
         //inner loop
         for(int column=1; column<=row;column++)
         {
            //display * on the same line
            System.out.print("*");
         }
         //display a blank line-same as moving the cursor to the next line
         System.out.println();
      }
   }
}
```

2.5 ARRAYS AND STRINGS

2.5.1 Introduction

An array is a fixed-size collection of data items (elements) of the same data type that are referenced by a common name. You can create an array of any data type (primitive or non-primitive). Using arrays over multiple variables makes it easier to manipulate the data.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 98-103, 107-122

2.5.2 Creating arrays

Declaring an array entails specifying the data type, the array name and the size. The syntax for creating an array is as follows:

```
Data-type[] arrayName; //this does not reserve space
arrayName = new Data-type[size]; //reserves memory for the array.
```

The above steps could be combined as follows:

```
Data-type[] arrayName = new Data-type[size];
```

You can also create an array as follows:

```
Data-type[] arrayName = {elements-separated-by-commas};
```

The size of an array is a non-zero integer, ideally two or above. You can create a an array with more than one dimension. Most commonly, you will encounter two-dimensional (2D) arrays. The number of square brackets indicates the number of dimensions. Thus, the syntax for declaring a 2D array is as follows:
Data-type[][]ar = new Data-type[number-of-rows][number-of-columns];

Example

```
int[][]ar = new int[3][4];

String[]days={"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday
","Saturday"};
```

2.5.3 Accessing array elements

Array elements are accessed through indices/ indexes. Array indexes are zero-based, that is, the first index of an array is always zero (0). Thus, the largest index of an array in 1 less than the size of the array. The size of an array is the number of elements an array can accommodate. To access an array's elements, you write the name of the array followed by a pair of square brackets, inside which you write the index number.

Example

```
package stadio;
public class ArraysExample {
   void exampleMethod()  {
       int[] marks = new int[3];
//store elements in array using indexes
marks[0] = 80;
marks[1] = 70;
marks[2] = 95;
//read marks from array
    for(int i=0;i<marks.length;i++)    {
        System.out.println(marks[i]);
   }
   }
 }
```

2.5.4 Displaying array elements

Displaying array elements requires you to access the individual elements through their indexes. Moreover, you can also use the enhanced-for loop to access array elements. Using the enhanced-for loop, you do not use indexes, instead, the loop iterates through the array until it gets to the end. The syntax of an enhanced-for loop is as follows:

```
for(datatype variableName: ArrayName)
{
//code to execute
}
```

Example

```
package stadio;
```

```
public class ArraysExample {
   void exampleMethod()  {
       int[] marks = new int[3];
//store elements in array using indexes
marks[0] = 80;
marks[1] = 70;
marks[2] = 95;
//read marks from array
    for(int i=0;i<marks.length;i++)    {
        System.out.println(marks[i]);
   }
   }
 }
```

2.5.5 Strings

Strings in Java are objects, which can be created by invoking the class's constructor. Because a string is a class, its name begins with an uppercase letter S, as is the class naming convention in Java. Strings are immutable objects, meaning, once an object of String is created, it cannot be modified. When you create a new string, it is actually a new object. There are different ways of declaring and initialising strings in Java. These include the following:

```
String t = new String("Test"); //creates a String object referenced by t
String name = "Stadio Institution of Higher Learning";
```

Now suppose you write the statement: t= t + name; This statement concatenates (joins) the two string to form a new string object that is referenced by the variable t. This means t is no longer referencing to the string object "Test" and therefore "Test" is now unreachable and will be automatically deleted from memory (automatic garbage collection). By concatenating t and name to form a new string, the old string that was referenced by t also goes out of scope and a new string is created that t now references.

Remember a String is a class in Java a class, thus, it has several methods defined in it, such as to find the length of the string, a character at a certain index, and converting the string to uppercase, among others.

Activity

Write short notes on the methods of the String class, including the purpose of each.

Write a Java program to use those methods.

2.6 COLLECTION PROCESSING

A collection is a framework that provides an architecture for storing and manipulating groups of objects, such as lists, queues and linked lists. It is a dynamic data structure, that is, its size is grows and shrinks as necessary unlike with fixed-size (static) data structures like arrays. Besides being dynamic, collections also contain data of different data types unlike arrays that can only store data of the same type. In other words, collections store heterogeneous data.

The Java Collection Framework contains interfaces, classes (both abstract and concrete) and methods. The super interface of the Java Collection Framework is called Collection. There are numerous other sub-interfaces, including List, Set, Queue and Iterator. The concrete classes implementing these interfaces include ArryList, HashSet, LinkedList and PriorityQueue. Various operations can be performed on collections, but the commonest ones include iteration, insertion and deletion.

Example

```
package stadio;
/*ArrayList and List are both in the same package so you could write only import statement: import java.util.* and it will import all the classes and interfaces in the java.util package*/
import java.util.ArrayList;
import java.util.List;
public class CollectionsExample {
    public static void main(String[] args) {
        int choice =1;
        /*declare collections variables*/
      ArrayList al = new ArrayList();
      /*You can include the data type when declaring a collection object
      but this is unnecessary since collections store heterogeneous data*/
      List<Integer> ls = new ArrayList<Integer>();
      /*Note that List is an interface. You can create a reference variable of its type but not an object. The variable can reference any object of classes that implement it*/
      List names = new ArrayList();
      /*Populate the lists*/
      al.add("Monday");
      al.add(2);//notice we are storing both integers and strings in one list
      System.out.println("The list current has the elements : "+al);
      /*you can insert an element at a particular index. This does not replace the element but at that index but pushes it to the next index*/
      al.add(0, "Sunday");
```

```
        System.out.println("The list current has the elements : "+al);
        /*using the set method replaces (does not push it to the next index) an element at that index*/
        al.set(1, "Replaced Monday at index 1");
        System.out.println("The list current has the elements : "+al);
        /*accept names using a loop*/
        ls.add(5);
        ls.add(choice); //store the current value in variable choice, i.e 1
        System.out.println("Elements in ls : "+ls);
    }
  }
```

Output

```
The list current has the elements : [Monday, 2]
The list current has the elements : [Sunday, Monday, 2]
The list current has the elements : [Sunday, Replaced Monday at index 1, 2]
Elements in ls : [5, 1]
```

Activity

Compare and contrast arrays and array lists.

If you had to choose between arrays and array lists, which one would you choose and why?

SUMMARY

This topic covered concepts on more complex methods like recursive methods with and without return values. A recursive method is one that calls itself. It must always have a base case which tells the method when to stop the call. Methods can be parameterised or not. A parameterised method is one that takes one or more inputs from the method that invokes it.

Moreover, control flow constructs were covered, specifically looping constructs. The loops in Java are the while, do..while, for and enhanced-for loops. Each of these allow you to write code that repeatedly executes for as long as a certain condition is true. A finite loop consists of three parts – initialisation, stopping condition and an operation that modifies the initial value to approach the stopping one. A loop that executes endlessly is called an infinite loop. Loops can be nested. Nesting loops means putting a loop inside the other. Note that the inner loops always execute to completion before control is returned to the outer loop, and when control returns to the inner loop the values are set to the initial ones.

You also learnt about Strings, that they are not a primitive data type but a class. The default value of a string is null. Strings are immutable objects, meaning once created they cannot be modified. When you try to modify a string, you actually create a new one instead. There are several string methods as you have seen and used. Finally, you learnt about collections, their use and advantages over static data structures like arrays.

Self-Assessment Questions

1. Given the sequence 5,8,11,14, write a recursive function to determine the n^{th} number in this sequence.

2. Write a program that allows a user to enter the number of rows and displays an inverted half pyramid. The number of rows must be between 5 and 9. If a user enters any number outside of this range, display an error message and ask them if they want to enter another number. If yes, allow them to, otherwise, display a goodbye message. For instance, if a user enters 5 for number of rows, the pyramid should look like the one below:

   55555
   4444
   333
   22
   1

   Hint: Use nested loops

3. Write a program that allows a user to create an array of integers of their preferred size and asks them to populate it. Your program should display the highest and smallest elements, as well as the sum and average.

4. Write a program that allows a user to enter any number of names into an array list and allows them to search for any names. Your program should also allow users to delete and replace any names as they so wish.

5. Write a Java program that allows a user to create a string array of their preferred size, populate it and search for any elements. The search must be case insensitive, that is, if the name is John and the user enters jOhn, it should still state that the name exists.

Topic 3
Objects

3.1 INTRODUCTION

This topic relates to the following module outcomes:

5. Apply basic object-oriented programming constructs in Java to various problems.
6. Learn new programming languages in future modules more easily, given the programming skills developed in this module

In this topic, you will gain knowledge in the following areas:

1. Point objects
2. Attributes
3. Objects as parameters and return types
4. Null objects
5. Garbage collection
6. Class diagrams
7. Using library classes

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 167-170

3.2 POINT OBJECTS

In two-dimensional (2D) mathematics, we can represent a point on a line using a coordinate, which is basically a point that shows the x and y values of that point. Each coordinate is represented as (x,y). While you can create your own

class for this purpose, Java defines a class called Point in the package java.awt. You just have to import the package and start using the class. The point class has attributes x and y and methods getX and getY, among other methods. You can create objects of the class Point and use them. The class has overloaded constructors. The Point class is a subclass of Point2D, hence, it is used for 2D only.

Example

Question: Write a program that calculates the gradient of a straight using the Point class of java.awt. Use the points (3,8) and (6,-4).

Note: The gradient of a straight line, m, is calculated as: $m = \frac{y_1 - y_2}{x_1 - x_2}$

```
/*program to calculate the gradient of a straight line with given points*/
package stadio;
import java.awt.Point;//required to use the Point class
public class Driver {
    public static void main(String[] args) {
       //first coordinate
        Point p1 = new Point(3,8);
        //second coordinate
        Point p2 = new Point(6,-4);
        /*casting is required since the gradient is likely to having a
        floating point, yet the points are integers*/
        double m = (double)(p1.x - p2.x)/(double)(p1.y-p2.y);
        System.out.println("Gradient = "+m);
    }
    }
```

Output

Gradient = -0.25

Activity

Write a program to calculate the gradient of a straight line for any coordinates entered by a user.

Note:

- Your program must accept inputs from users.

- Use getters for x and y (getX and getY) instead of the attributes x and y in your computation.

3.3 ATTRIBUTES

3.3.1 Introduction

You probably recall that a class contains attributes (fields/ state) and methods (behaviour). These fields could static or not. Let us look at each of these.

3.3.2 Instance variables

An instance variable is a variable declared inside a class, outside any methods and not marked as static. Each instance (object) will have its own value for each instance variable. For instance, think of a class called Person with the variables name and address. Each person will have their own name and address, thus, there is a different value to each of these variables for each instance (object), and hence, they are called instance variables. Any variables declared inside any methods are local to that method, not instance variables. In the case of the Point class, it has two instance variables – x and y. To access these attributes from another class, you must first create an object of that class and access them using a **dot operator**. In the case of the Point class, you can access the attributes x and y as follows:

```
Point pt = new Point(3,5);
int x = pt.x;
int y = pt.y;
```

3.3.2 Static variables

A static variable is declared inside of a class, outside of any methods and is marked static. The static modifier can be placed anywhere before the data type. To access static variables, you use the class name and not an object. You still use the dot operator.

3.4 OBJECTS AS PARAMETERS

You can pass an object as an argument to a method; thus, you can create methods with objects as parameters. Once you pass an object as an argument to a method, you can access its accessible methods and attributes. Remember a

class is a user-defined data type. Since we can declare variables of that type, we can also create parameterised methods with those classes as parameters, allowing us to pass their objects as arguments.

Example

Question: Write a program that calculates the gradient of a straight using the Point class of java.awt. Use the points (3,8) and (6,-4).

Note: The gradient of a straight line, m, is calculated as: $m = \frac{y_1 - y_2}{x_1 - x_2}$

```
package stadio;
import java.awt.Point;//required to use the Point class
public class Driver {
   //a method that takes objects as parameters
   public static void example(Point p1, Point p2)
   {
 /*casting is required since the gradient is likely to having a floating point, yet the points are integers*/
       double m = (double)(p1.x - p2.x)/(double)(p1.y-p2.y);
       System.out.println("Gradient = "+m);
   }
   public static void main(String[] args) {
       Point p1 = new Point(3,8);
       Point p2 = new Point(6,-4);
       example(p1,p2);
   }
   }
```

Output

Gradient = -0.25

3.5 NULL OBJECTS

A null object is one without reference or defined with null behaviour or functionality. Accessing the properties and methods of a null object will throw an exception because the object essentially does not exist. Instead of writing code always to check if an object is null, you can use null objects patterns to cater for that. The following example illustrates this concept.

Example

```
//File name: Employee.java
package stadio;
//abstract class with abstract methods
public abstract class Employee {
   public abstract String getName();
   public abstract boolean isExists();
}
//file name: RealEmployee.java
package stadio;
public class RealEmployee extends Employee{
   private String name;
   public RealEmployee(String name)
   {
      this.name = name;
   }
   @Override
   public String getName()
   {
      return this.name;
   }
   @Override
   public boolean isExists()
   {
      return true;
   }
}
/*file name: NoEmployee.java*/
package stadio;
//class to deal with null objects of Employee
public class NoEmployee extends Employee{
    @Override
   public String getName()
 {
      return null;
   }
   @Override
   public boolean isExists()
  {
      return false;
   }
}

/*File name: Driver.java*/
package stadio;
import java.util.ArrayList; //required for ArrayList
import java.util.List;//required for List
public class Driver {
   public static void main(String[] args) {
//get the list of Employees and store it in a local variable list
List<Employee> list = Driver.getEmployee();
      for(Employee emp:list)
```

```
    {
       System.out.println("Object is real : "+emp.isExists());
    }
 }
//method to create a list of Employee and return the list
   public static List<Employee> getEmployee()
    {
       List<Employee> list = new ArrayList<Employee>();
       list.add(new RealEmployee("John"));
       list.add(new RealEmployee("Nyabiko"));
       list.add(new NoEmployee());// a null object
       list.add(new RealEmployee("Batani"));
       list.add(new RealEmployee("McJohn"));
       list.add(new RealEmployee("Memory"));
       list.add(new NoEmployee());// a null object
       return list;
    }
    }
```

Output

```
Object is real : true
Object is real : true
Object is real : false
Object is real : true
Object is real : true
Object is real : true
Object is real : false
```

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 176

3.6 GARBAGE COLLECTION

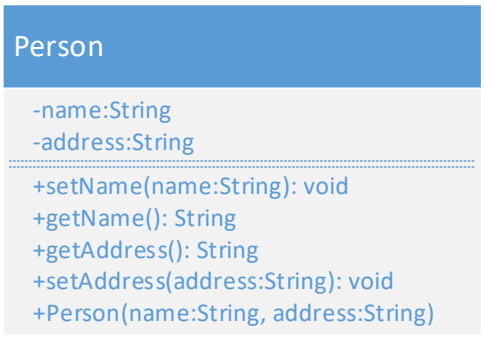
Computer memory is one of the resources that need to be efficiently used. When you create an object, it gets allocated memory inside the computer. There is a need to constantly monitor if the object is still needed, otherwise, the memory allocated to it must be freed up. Unlike in other languages like C++ where you have to create a destructor, Java performs automatic garbage collection. It

automatically checks for objects that are no longer reachable and deletes them from memory.

3.7 CLASS DIAGRAMS

Remember a class contains attributes and methods. The Unified Modelling Language (UML) provides a graphical means to design classes by indicating the attributes and methods, along with their data types and return types. Consider the following example:

Example

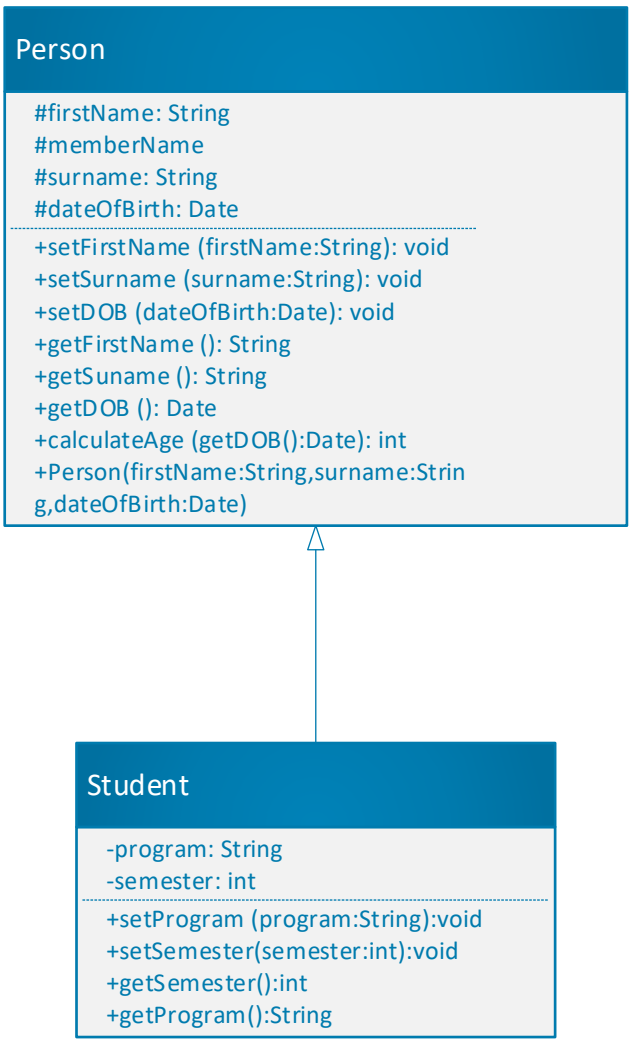


The above example is a class diagram for the class Person with the attributes name and address and the setters and getters as methods. Note that the – and + are access modifiers private and public, respectively. Thus, all the attributes are private while all the methods are public. Can you identify the constructor in that diagram? Pause for a moment and check if there is a constructor. Yes! The method Person is the constructor because its name is the same as that of the class and it has no return type. Each method has optional parameters, the data type of which is indicated after the parameter name. Outside of the parenthesis after a colon (:), what follows is the return type of the method.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 174-180

Example



The above example shows two related classes – Person and Student. In this example, Student is a subclass of Person. The attributes of Person are not private but protected (#).

Generally, the following symbols are used for access modifiers in class diagrams.

Table 3.1 Representation of visibility of class members

Modifier	Symbol
private	-
protected	#
default	~
public	+

Activity

(a) Draw a class diagram for any of the classes you have created before. Also include the ones you did under inheritance.
(b) Research how to represent relationships between classes and interfaces in an UML diagram.

3.8 USING LIBRARY CLASSES

Java has numerous classes for different purposes. For instance, the Point class that you will learn about in topic 5, or the Date class that you can use for date manipulation. The classes are grouped into folders called packages, where related classes are stored in the same package. To access the classes, you should import them. For instance, the statement: import java.util.Date imports the Date class from the package java.util. As you can create a folder inside another folder, you can also create a package inside another package. The package java.util means the top-level package is java and inside it is util. See the example below.

Example

```
package stadio;
//import the necessary classes
import java.time.LocalDate;
import java.util.Date;
public class LibrariesExample {
    public static void main(String[] args) {
       /*retrieve current system date*/
       Date dt = new Date();
       //display the retrieved date
        System.out.println("Today is : "+dt);
        /*retrieve current data using the LocalDate class*/
        LocalDate dat = LocalDate.now();
        //display the local date
        System.out.println("Today's date is :"+dat);
   }
  }
```

Output

(Your today's date. In my case, the output was as follows)

Today is : Mon Nov 07 08:30:20 SAST 2022

Today's date is : 2022-11-07

Where you want to import multiple or all classes from a package, you can use a wildcard (*). For instance, import java.util.* imports all the classes in the package java.util.
Import statements may create ambiguity where you import all classes from different packages that have classes with the same name. For instance, the packages java.util and java.sql both contain a class called Date. Thus, the following code will cause an error:

```
package stadio;
//import the necessary classes
import java.time.LocalDate;
import java.sql.*;
import java.util.*;
public class LibrariesExample {
    public static void main(String[] args) {
       /*retrieve current system date*/
       Date dt = new Date(); /*causes error due to ambiguity. The compiler won't
know which Date class are you invoking, the one in java.util or java.sql*/
}
}
```

In such cases, you have to use a fully qualified name, like below:

```
package stadio;
//import the necessary classes
import java.time.LocalDate;
import java.sql.*;
import java.util.*;
public class LibrariesExample {
    public static void main(String[] args) {
       /*retrieve current system date*/
       java.util.Date dt = new java.util.Date();
       java.sql.Date dt1 = new java.sql.Date(dt.getTime());
       //display the retrieved date
        System.out.println("Today's java.util date is : "+dt);
         //display the sql date
        System.out.println("Today's java.sql date is :"+dt1);
    }
 }
```

Note

Import statements help shorten your code by avoiding typing fully qualified names, hence, you can write code without them though it a good practice to have them. Import statements are not like #include statements in C/C++.

Summary

This topic covered concepts related to objects by using the Point class of the java.awt package. The other concepts covered in this topic include attributes, passing objects as parameters to methods, null objects, garbage collection, class diagrams and using library classes.

An object is an instance of a class and is created by invoking one of the class's constructors, prefixing it with the **new** keyword. An object must always be referenced using a variable so that it will be accessible after creation. However, if you create an object and immediately invoke a method (e.g. new Person().display()), you may not have to reference it with any variables. A class can contain instance and/or static variables. An instance variable has a different value for each object while a static variable is a class variable and ideally contains the same value for all objects of that class. A null object is a non-existent object, hence, it cannot be used for anything. Trying to use a null object throws a NullPointerException. Instead of writing code to always check if an object is null or not, you can employ the concept of null object design pattern, where you write code with null behaviour to handle such cases.

Java performs automatic garbage collection – there is no need to create a destructor (like in C++). Objects that are out of reach and scope get automatically destroyed. Before writing code, it is important to design your software. One way of design in OOP is by using class diagrams to show what each class or interface will contain and it relates to others in the system. A class diagram contains three sections – name, attributes and behaviour sections. Each attribute must have visibility level, name and type. Methods must also have visibility level, return type, name and optional parameters. To use classes defined in Java libraries, it is advisable to use import statements to make your code shorter by avoiding using fully qualified class names. Where multiple packages

contain classes with the same name and you have imported them, you must use fully qualified class names to avoid ambiguity that would lead to errors.

Self-Assessment Questions

1. Dr John Nyabiko is a farmer in Honde Valley, a beautiful Lowveld area in Manicaland Province, Zimbabwe. He wants a Java application that can help him keep track of his animals. He has dogs, fish and cattle. For each animal, he wants to keep track of its type, number of legs, colour and health status. Moreover, he wants to track whether each dog is vaccinated or not, each fish has scales or not, each cow's number of calves and whether it is due for dip or not. Draw the class diagram for this scenario.

2. Create a class called NoAnimal that inherits Animal and handles the default behaviour if the Animal object is null.

3. A square is a 2D shape whose sides are equal. Write a Java program that contains a class called Square with the relevant instance variable(s). Include a method called printDetails that takes a parameter of type Square and displays its instance variable(s).

Topic 4
Classes

4.1 INTRODUCTION

This topic relates to the following module outcomes:

5. Apply basic object-oriented programming constructs in Java to a variety of problems.
6. Learn new programming languages in future modules more easily given the programming skills developed in this module

In this topic, you will gain knowledge in the following areas:

1. Basic class
2. Constructors
3. Getters and setters
4. Displaying objects
5. Representations of arrays of objects and objects of arrays

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 183-197

4.2 BASIC CLASS

4.2.1 Classes

The object-oriented programming (OOP) paradigm views software as a system of interacting objects. Objects of the same category (class) will always have the same attributes/ features. However, these attributes have different values. For

instance, you can think of a person as a class, and you and your friends as objects (instances of the class person). Each of you as a person has a name, height, address and sex (gender). These are called the features/ attributes of the class person. Thus, a class is blueprint or template upon which objects are created. It is a user-defined data type. The syntax for creating classes is as follows:

```
access-modifier non-access-modifier class ClassName
{
//attributes
//methods or behaviour
}
```

An access-modifier could be public or default, whereas a non-modifier could be final. A top-level class cannot be marked as private or protected, but a nested inner class can. Everything that is marked in blue in the above syntax is optional. The class naming convention is that class names should be in upper. Let us look at an example.

Example

```
package stadio;
public class Person {
  /*attributes. Note that variables of the same type can be declared in one line*/
   private String fullName;
   private String address;
   private float height;
   //behaviour or methods
   public void eat()
   {
     //code for the method comes here
   }
   public void sleep()
   {
       //code for the method comes here
   }
}
```

Your project will ideally have multiple classes but only one should have the main method. A Java class with the main method is called a Java main class. It is inside the Java main class where we ideally create objects of other classes and invoke their methods. In this module, we shall call the Java main class the Driver class.

Note

You must create only one class on each source file (.java file) since you cannot have multiple public classes on one file. Where multiple classes have been created on one source file, only one public file must exist and such a class must have the same name as the file name.

4.2.2 Objects

An object is an instance of a class. You are an instance of a person, with your own values for the person's features specified above. In Java, a class is created by using the keyword **new** followed by invocation of the class's constructor. An object of the class Person above is created by writing new Person(); The computer reserves memory to store the object, but you will not be able to access this object later if you do create a reference variable to refer to it. We declare the variable by writing the class's name (user-defined data type) followed by variable name. For example:

```
Person p = new Person();
Person person = new Person();
```

You can now access the accessible features and methods of the class Person through the reference variable, such as p.fullName.

Note

Note that when you write the statement Person p = new Person(), p is not the object. Instead p is storing the object. The object is created when write new Person();

4.3 CONSTRUCTORS

A constructor is a special kind of a method that has the same name as the class's name and does not have a return type (not even void!). A constructor is used to create (construct) 0bjects and initialise instance variables. A constructor can be parameterised or not. If you do not create any constructors in your class, Java will insert a default no-argument (no-args) constructor for you. However, if you create any constructors, the no-args constructor ceases to exist and if you want one, you have to create it yourself.

If you create a parameterised constructor, it is advisable to use the same parameter names as the instance variables, and then use the 'this' keyword to differentiate the instance and local variables. Parameters to any method are local variables. Constructors can be overloaded, that is, you can have more than one constructor in a class with different parameter lists. This is called constructor overloading.

Example

```
package stadio;
public class Person {
  /*attributes*/
   private String fullName;
   private String address;
   private float height;
   //no-args constructor
   Person(){
      //initialise instance variables to default values
      fullName = null;
      address = null;
      height =0.0f;
   }
   /*parameterised constructor. The parameters are local variables and have the same names as the instance variables. The 'this' keyword is associated with instance variables*/
   Person(String fullName, String address, float height){
      this.fullName = fullName;
      this.address = address;
      this.height = height;
   }
}
```

4.4 GETTERS AND SETTERS

The encapsulation feature of OOP requires class attributes to be marked private unless they are inherited, in which case they must be marked protected. Private class members can only be accessed with the same class. Accessing them from a different class requires you define getters and setters for those attributes. A

method that is used to access (read) a value of an attribute is called a getter or accessor. A getter must always have a return type other than void.

On the other hand, a method that is used to set or modify the value of a class attribute is called a setter or mutator method. A setter must always have a parameter representing the value to be set to the variable. The return type of a setter is also void since it does not return a value. Setters and getters are used to provide access control. Since a getter is used for reading and a setter is used for writing, if you do not want to provide a read access to other classes, you just do not create a getter for that value. Likewise, if you do not want to provide write access, you simply do not provide a setter for the concerned attribute. The getters and setters are marked public. The naming convention for setters is setXxxx and for getters is getXxxxx, that is, setters names begin with set while getters' names start with get.

Example

```
package stadio;
public class Person {
/*attributes*/
    private String fullName;
    private String address;
    private float height;
     /*getter for fullName*/
    public String getFullName()
    {/*using the keyword this on the getter's return statement is optional*/
        return this.fullName;
    }
    //setter for fullName
    public void setFullName(String fullName)
    {
        this.fullName = fullName;
    }
    //getter for address
    public String getAddress()
    {
        return this.address;
    }
      public void setAddress(String address)
    {
       this.address = address;
    }
    public void setHeight(float height)
    {
        this.height = height;
    }
   public float getHeight()
    {
        return height;
```

```
    }
    //no-args constructor
    Person(){
        //initialise instance variables to default values
        fullName = null;
        address = null;
        height =0.0f;
    }
    /*parameterised constructor. The parameters are local
    variables and have the same names as the instance variables.*/
    Person(String fullName, String address, float height){
        this.fullName = fullName;
        this.address = address;
        this.height = height;
    }
}
```

4.5 DISPLAYING OBJECTS

An object is an instance of a class. We can display the properties of an object by creating a method within its class to display the properties, or we can do so from another class. Where displaying is done from another class, the attributes should be accessed through getters since attributes are private.

Example

```
package stadio;
public class Person {
   /*attributes*/
    private String fullName;
    private String address;
    private float height;
      /*method for displaying. You could still maintain the setters and getters and
also constructors. They have been removed here to shorten the code*/
    // method for displaying an object p
public void display(Person p)
    {
        System.out.println("Full name :"+p.fullName);
        System.out.println("Address :"+p.address);
        System.out.println("Height :"+p.height);
    }
/*another display method. Having more than one method with the same name
in the same class is called method overloading. You will learn this later*/
  public void display()
    {
        System.out.println("Full name :"+fullName);
        System.out.println("Address :"+address);
        System.out.println("Height :"+height);
    }
```

```
}
```

4.6 REPRESENTATIONS OF ARRAYS OF OBJECTS AND OBJECTS OF ARRAYS

Remember a class is a data type, meaning we can create arrays of objects of any class. Creating an array of objects follows the syntax:

ClassName[] variableName = new ClassName[size];

This creates an array of reference variables, but the actual object referred by each of those must be created separately by passing the right arguments to the construct. You can use loops, just like with ordinary variables. Study the example below.

Example

```
/*File Name: Person*/
package stadio;
public class Person {
    private String name;
    int yearOfBirth;
    Person(String name) {
        this.name = name;
    }
   public void setName(String name)
    {
        this.name = name;
    }
  public String getName()
  {
     return this.name;
   }
}
/*File name: Driver*/
package stadio;
import java.util.Scanner;
public class Driver {
    public static void main(String[] args) {
            /*create an array of objects of Person with size 3*/
        Person[] pe = new Person[3];
/*create a Scanner object to accept input from users*/
         Scanner sc = new Scanner(System.in);
        for(int i=0;i<pe.length;i++)
        {
            System.out.println("Enter name ");//prompt for input
            pe[i] = new Person(sc.nextLine());//create new object
```

```
        }
        System.out.println("******OUTPUT**********");
/*use an enhanced for loop to display the names*/
        for(Person p:pe)
        {
           p.display();
        }
  }
 }
```

Recommended Reading

https://docs.oracle.com/javase/specs/jls/se8/html/jls-10.html

Summary

This topic covered the basics of classes, constructors, getters and setters, displaying objects and representations of arrays of objects and objects of arrays. Classes are the basic building blocks of OOP. They are templates for creating objects and contain attributes and/or methods. A constructor is a special method without a return type and with the same name as that of its class. Constructors can be overloaded but not overridden. Ideally, attributes of a class must be marked private unless they are inherited, in which case they should be marked protected. Getters and setters can be defined to provide access to instance variables by the external world. These getters and setters are normally public. Getters and setters provide a read or write access control mechanism. You can display objects by displaying their attributes. In the next topic, you will learn about objects of objects.

Self-Assessment Questions

1. Dr John Nyabiko is a farmer in Honde Valley, a beautiful Lowveld area in Manicaland Province, Zimbabwe. He wants a Java application that can help him keep track of his animals. He has dogs, fish and cattle. For each animal, he wants to keep track of its type, number of legs, colour and

health status. Moreover, he wants to track whether each dog is vaccinated or not, each fish has scales or not, each cow's number of calves and whether it is due for dip or not. Suppose you have been tasked with developing this system, draw the system's class diagram. [Note: Include an interface called IAnimal with setters and getters for all the properties of the class Animal and a display method to display the animal's attribute values].

2. Implement the system you designed above. It should be a complete system and must allow users to choose the animal they want to add (dog, fish or cow). After adding the animal, ask users if they want to continue using the system or not and your system should respond accordingly.

3. Create a class called Address with the instance variables for house number, street name and city name. Create two constructors- a parameterised constructor to initialise the address to any user-entered values and a no-args constructor to initialise the address to default values. Include a Java main class to make the program complete.

Topic 5
Objects of Objects

5.1 INTRODUCTION

This topic relates to the following module outcomes:

5. Apply basic object-oriented programming constructs in Java to a variety of problems.
6. Learn new programming languages in future modules more given the programming skills developed in this module

In this topic, you will gain knowledge in the following areas:

1. Abstraction, modularization, encapsulation
2. Subclasses
3. Inheritance
4. Method overriding
5. Class relationships
6. Polymorphism
7. Encapsulation: privacy, visibility and interfaces

5.2 ABSTRACTION, MODULARISATION, ENCAPSULATION

5.2.1 Introduction

This topic discusses the distinguishing features of OOP, including abstraction, encapsulation, inheritance, and polymorphism (including method overriding and overloading).

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 273-275

5.2.2 Abstraction

Abstraction is a feature that hides unnecessary details from users, only showing essential ones. Abstract classes and interfaces provide abstraction mechanism in Java. You will learn about interfaces later in this topic. An abstract class is a class marked as abstract, where abstract is a non-access modifier. An abstract class can have abstract method(s). An abstract method is a method marked as abstract. An abstract method is a method prototype, it does not contain a body and can only be declared inside an abstract class or interface. Abstract classes cannot be instantiated. This means, even though you can create a constructor for an abstract class, you cannot create an object of the class. An abstract class can also contain non-abstract methods. A non-abstract class is called a concrete class.

Example

```
package stadio;
/*an example of an abstract class. The keyword abstract can be placed before or after public, but not after class*/
public abstract class AbstractionExample {
   /*abstract classes can have variables too*/
   protected int someInt;
   /*abstract methods do not contain a body*/
   public abstract void test();
   public abstract int getAge();
   /*abstract classes can have non-abstract methods*/
   public void show()
   {
      System.out.println("This method is not abstract");
   }
     /*abstract classes can have constructors. These are used to initialise instance variables but not to create objects*/
   public AbstractionExample()
   {
      //code comes here
   }
}
```

5.2.3 Modularisation

Modularisation is design concept that breaks down a piece of software into small subcomponents called modules. Modularisation simplifies debugging and maintainability.

5.2.4 Encapsulation

The bundling together of attributes (data/ state) and the methods (behaviour) that act upon them is called encapsulation. We encapsulate fields and methods

inside a container called a class or interface. When writing Java code, you are writing classes and/or interfaces. You cannot write any code outside of a class or interface. Even inside classes and interfaces, all code is written inside methods, except variable declaration. An attempt to assign a value to a variable outside of a method after declaration will throw an error. Besides bundling attributes and methods, encapsulation also includes data hiding, which is achieved by marking instance variables as private.

5.3 SUBCLASSES

You have learnt that a class is a blueprint or template for creating objects. Class names are nouns, like Person, Vehicle, Student, Lecturer, among others. When you consider the classes Person and Student, you can realise that they are related in that every Student is a Person. In other words, a Student is a subtype of Person. In OOP terms, a Student is a subclass of Person. Thus, Person is a superclass of Student. Use the IS-A relation to establish whether one class is a subclass of another. For instance, Person IS-A student is not always correct since not every person is a student; thus, Person cannot be a subclass of Student. However, Lecturer IS-A Person is a valid relationship; thus, Lecturer is a subclass of Person, and Person is a superclass of Lecturer.

5.4 INHERITANCE

5.4.1 What is inheritance?

The concept of subclass and superclass falls under inheritance. Inheritance is an OOP feature where a subclass acquires the non-private members of its superclass, including attributes and methods. Private members of a class are not inherited in Java. Think of it this: You inherited the surname that you are using from your parent. However, you cannot inherit your parent's academic qualifications because they are private to him. There are two keywords used for inheritance in Java programs – extends and implements. The **extends** keyword is used when one class inherits another, or when an interface inherits another interface. On the other hand, **implements** is used when a class inherits an interface. An interface never inherits a class! Where a superclass's attributes are to be inherited by a subclass, mark them as **protected** and not **private.** A Java class cannot have multiple super classes, thus, Java does not support multiple inheritance. However, a Java class can implement multiple interfaces.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 235-249

Example

```
/*this code should be in a file named Person.java*/
package stadio;
public class Person {
  /*attributes*/
   protected String fullName;
   protected String address;
   protected float height;
   /*parameterised constructor*/
   public Person(String fullName, String address, float height)
   {
      this.fullName = fullName;
      this.address = address;
      this.height = height;
   }
   /*method for displaying*/
   public void display()
   {
      System.out.println("Full name :"+fullName);
      System.out.println("Address :"+address);
      System.out.println("Height :"+height);
   }
 }

/*this code should be in a file named Student.java*/
package stadio;
/*Student extends Person means Student is a subclass of Person*/
public class Student extends Person{
/*since this class is not meant to be inherited, we will mark its fields as private*/
private String programme;
/*create a constructor. The first statement in the constructor should be an
invocation of the superclass's constructor to initialise the inherited instance
variable*/
public Student(String fullName, String address, float height, String programme)
{
   //invoke the superclass constructor
   super(fullName, address, height);
   //initialise this class' unique variables
   this.programme = programme;
}
```

```
}
```

5.4.2 Inheritance rules

The following rules apply to inheritance in Java:

1. A class can only extend one class.
2. A class can implement multiple interfaces. A class does not extend but implements an interface.
3. An interface can extend multiple interfaces.
4. Only non-private members of classes are inherited.
5. Constructors are not inherited.
6. Inherited attributes are initialised by the constructor of the superclass from which they are being inherited.
7. The first statement in a subclass's constructor must be an invocation of the superclass's constructor using the **super** keyword, passing the arguments to it – super(argument-list).
8. A class marked final cannot be inherited.

5.5 METHOD OVERRIDING

Suppose you have two classes – TwoDShape and Triangle, where Triangle IS-A TwoDShape. In the TwoDShape, you could have a method that calculates the area as the product of the two sides. However, for a Triangle, the area has to be half of the product of the two sides. Java provides a mechanism to override the behaviour of the superclass's method by providing a different implementation unique to the subclass. This is called method overriding. When you override a method and invoke it on an object of the subclass, the superclass's version gets overshadowed and inaccessible.

Prescribed reading

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/ pages 248, 290-291, 288,297, 238

Example

```
/*a program to calculate the area of a two dimensional shape*/
package stadio;
public class TwoDShape {
    /*sides of a two dimensional shape*/
    protected int length, width;
    /*parameterised constructor*/
    public TwoDShape(int length, int width)   {
        this.length = length;
        this.width = width;
    }
    /*method to calculate the area of a 2D shape*/
    public int calcArea()  {
        return this.length * this.width;
    }
    public void display()    {
        System.out.println("This is a 2D shape");
        System.out.println("The area = "+calcArea());
    }
}
/*the following code must be in a file named Driver.java*/
package stadio;
import java.util.Scanner;
public class Driver {
    public static void main(String[] args) {
        int l,w;
        Scanner sc = new Scanner(System.in);
        /*accept dimensions*/
        System.out.println("Enter the length ");
       l = sc.nextInt();
       System.out.println("Enter the width ");
       w = sc.nextInt();
        //create an object of TwoDShape
        TwoDShape td = new TwoDShape(l,w);
        /*invoke display. Note, display invokes calcArea so there is no
        need to invoke calcArea in this method*/
        td.display();
     }
    }
```

Below is the output of the above code.

```
run:
Enter the length
4
Enter the width
5
This is a 2D shape
The area = 20
```

The above example does not involve method overriding. We are now going to create the class Triangle and override the method calcArea.

Example

```
package stadio;
/*Triangle extends TwoDShape so it's a subclass of TwoDShape*/
public class Triangle extends TwoDShape{
    /*this clas does not have any unique attributes. It inherits length and width
    as base and height. However, it must have a parameterised constructor to
initialise those attributes*/
    public Triangle(int l,int w)
    {
        super(l,w);
    }
    /*This class inherits calcMethod but the formula is different.  So let's override
that method*/
    @Override
    public int calcArea()
    {
        return this.length*this.width/2;
    }
    /*Because we want to maintain the behaviour of the method display, we are
not overriding it*/
}
/*file name is Driver.java*/
package stadio;
import java.util.Scanner;
public class Driver {
    public static void main(String[] args) {
        int l,w;
        Scanner sc = new Scanner(System.in);
        /*accept dimensions*/
        System.out.println("Enter the length ");
       l = sc.nextInt();
       System.out.println("Enter the width ");
       w = sc.nextInt();
        //create an object of Triangle
        Triangle td = new Triangle(l,w);
        /*invoke display. Note, display invokes calcArea so there is no need to
invoke calcArea in this method*/
        td.display();
        }
  }
```

Output of the above code:

```
run:
Enter the length
4
Enter the width
4
This is a 2D shape
The area = 8
```

Sometimes you may want to override a method but retain its functionality and add more functionality specific to the subclass. To achieve this, you use the **super** keyword to invoke the superclass's version in the overriding method. In our example, suppose we want to override the display method to include the string "This is a triangle" in addition to retaining what the method is currently doing. The following example illustrates that.

Example

```
package stadio;
/*Triangle extends TwoDShape so it's a subclass of TwoDShape*/
public class Triangle extends TwoDShape{
    /*this clas does not have any unique attributes. It inherits length and width as base and height. However, it must have a parameterised constructor to initialise  those attributes*/
    public Triangle(int l,int w)  {
        super(l,w);
    }
    /*This class inherits calcMethod but the formula is different.
    So let's override that method*/
    @Override
    public int calcArea()  {
        return this.length*this.width/2;
    }
    /*override display*/
    @Override
    public void display()  {
       super.display();/*if you do not do this, only the next line will be displayed*/
       System.out.println("This 2D shape is a Triangle");
    }
}
package stadio;
import java.util.Scanner;
public class Driver {
    public static void main(String[] args) {
        int l,w;
        Scanner sc = new Scanner(System.in);
        /*accept dimensions*/
        System.out.println("Enter the length ");
```

```
        l = sc.nextInt();
        System.out.println("Enter the width ");
        w = sc.nextInt();
         //create an object of Triangle
         Triangle td = new Triangle(l,w);
 /*invoke display. Note, display invokes calcArea so there is no need to invoke
calcArea in this method*/
         td.display();
         }
    }
```

Output:

```
run:
Enter the length
4
Enter the width
5
This is a 2D shape
The area = 10
This 2D shape is a Triangle
```

Note

A method can be marked as **final**, in which case it cannot be overridden.
A class marked as **final** cannot be inherited/ extended.
A variable marked as **final** cannot be reassigned. In other words, it is a constant and must be initialised on declaration.

Activity

Write the rules of method overriding in Java in your notebooks. Discuss with your friend your understanding of each rule.

5.6 CLASS RELATIONSHIPS

5.6.1 Introduction

There are basically two types of class relationships that you will learn in this section – inheritance and composition. Let us look at each of them.

5.6.1 Inheritance (IS-A relationship)

If one class is of a type of another class, such that their relationship can be represented by IS-A, then that class is a subclass. The subclass can inherit all the non-private members of its superclass. This kind of relationship is called inheritance. For instance, a Dog is-a Mammal means that Dog and Mammal are related in that Dog is a subclass of Mammal.

5.6.2 Composition (HAS-A relationship)

Suppose we have two classes – Person and Eye. The relationship between these classes is such that Person HAS-A Eye. This means that while Eye is a class, it is also an attribute of the class Person. This type of a relationship is called a composition.

Example

```
public class Person {
   /*attributes*/
    protected String fullName;
    protected String address;
    protected float height;
    protected Eye eye; //Person has Eye as its attribute, yet Eye is a class
   }
public class Eye {
//attributes of Eye
    String color;
   }
```

5.7 POLYMORPHISM

5.7.1 Introduction

Polymorphism is a feature of OOP languages. The word poly means many, and polymorphism simply means "having many forms". We are going to cover method overload and reference variables to clarify this concept.

5.7.2 Method overloading

Suppose we create a class called TwoDShape in which we will have a method calcArea to compute the area of the shape. Ordinarily, you would expect such a method to return the product of width and length. However, there are other 2D shapes, like a circle, whose area is calculated differently. Java provides a

mechanism to define multiple methods with the same name (but different parameters) in the same class. This is called overloading a method, or method overloading. The compiler will determine which version of that method to invoke at runtime based on the passed arguments. Thus, the method will take different forms. Let us look at an example.

Example

```
/*a program to calculate the area of a two dimensional shape using overloaded methods*/
package stadio;
public class TwoDShape {
   /*sides of a two dimensional shape*/
   protected int length, width;

   /*overloaded constructors*/
   public TwoDShape(int length, int width)
   {
      this.length = length;
      this.width = width;
   }
    public TwoDShape(int length)
   {
      this.length = length;

   }
   public void display(float area)
   {
      System.out.println("This is a 2D shape");
      System.out.println("The area = "+area);
   }
   //area of a circle
   public int calcArea(int r)
   {
      return r*r*22/7;
   }
   //area of a rectangle and square
   public int calcArea(int length, int width)
   {
      return length * width;
   }
   //area of a triangle
   public float calcArea(float temp,float base, float height)
   {
      return temp * base * height;
   }

}
```

```
/*file name: Driver.java*
package stadio;
import java.util.Scanner;
public class Driver {
    public static void main(String[] args) {
        int l,w,shape;
        Scanner sc = new Scanner(System.in);
        System.out.println("**********SELECT A SHAPE ********");
        System.out.println("1: Circle\n2 Square or Rectangle\n3 Triangle");
        shape = sc.nextInt();
        switch(shape)
        {
            case 1:
                System.out.println("Enter the radius of the circle");
                l = sc.nextInt();
                TwoDShape circle = new TwoDShape(l);
                circle.display(circle.calcArea(l));
                break;
            case 2:
                /*accept dimensions*/
                System.out.println("Enter the length ");
                l = sc.nextInt();
                System.out.println("Enter the width ");
                w = sc.nextInt();
                 TwoDShape rectangle = new TwoDShape(l,w);
                rectangle.display(rectangle.calcArea(l, w));
                break;
            case 3:
                 /*accept dimensions*/
                System.out.println("Enter the base ");
                l = sc.nextInt();
                System.out.println("Enter the height ");
                w = sc.nextInt();
                //create an object of Triangle
               TwoDShape td = new Triangle(l,w);
               td.display(td.calcArea(0.5f,l, w));
               break;
            default:
                System.out.println("Sorry, invalid input. Goodbye");
        }
      }
   }
```

Output

```
Do you want to continue? Press 1 for yes or 0 to exit
1
**********SELECT A SHAPE ********
1: Circle
2 Square or Rectangle
3 Triangle
2
Enter the length
4
Enter the width
3
This is a 2D shape
The area = 12.0
Do you want to continue? Press 1 for yes or 0 to exit
```

In this example, the method calcArea takes different forms depending on the arguments passed to it.

Activity

Write the rules of method overloading in Java in your notebooks. Discuss with your friend your understanding of each rule.

How is method overloading different from method overriding?

5.7.3 Reference variables

A class is a user-defined data type, thus, you can declare variables of that type. The variable you declare is **not** an object, rather it references an object. For instance, using Example 5.2 where we have two classes – Person and Student, the statement Person p does not create an object but a reference variable. An object is created when we write new Person(). The object that is created by this statement will never be accessible again unless we create a variable to reference to it, which we will then use whenever we want to access that object. This variable is called a reference variable. As you might recall, variable declaration and initialisation can be combined to reduce the lines of code. The following statement declares a reference variable and assigns an object to it.

Person p = new Person();

However, p can be reassigned a different value (object). Since p is of type Person and Student is a Person, we can assign an object of type Student to p. That is, we can write something like:

```
Person p;
p=new Person();
p=new Student();
```

Clearly, p is taking the form of a Person and also that of a Student, thus, p is taking many forms (polymorphism).

Note

A variable of a superclass type can store any objects of that class's subclasses. However, you cannot store an object of a superclass in a variable of a subclass. Doing so would require you to perform explicit casting. For example:

```
Student s;
s=new Person(); //error
s=(Student) new Person(); // works, explicit casting
```

5.8 ENCAPSULATION: PRIVACY, VISIBILITY AND INTERFACES

5.8.1 Privacy and Visibility

A principle of OOP is to make data (variables) private or encapsulated and methods public. Thus, where variables are not inherited, they must always be marked private. Private class members are only visible within the class they have been declared. If class attributes are being inherited, mark them protected. It is a bad OOP practice to mark variables as public! As such, you must never mark class attributes as public. You probably still recall this concept from the section on getters and setters. The following table summarises the visibility of class members based on the various access modifiers.

Table 5.1 Visibility of class members

Modifier	Within the same class	Same package, subclass	Different packages, subclass	Same package, non-subclass	Different package, non-subclass
private	Yes	No	No	No	No
protected	Yes	Yes	Yes	No	No
default	Yes	Yes	No	Yes	No
public	Yes	Yes	Yes	Yes	Yes

5.8.2 Interfaces

Java does not support multiple inheritance due to the complexities it involves. However, it provides interfaces which can be used to support multiple inheritance. You would recall that a class can only extend one class (single inheritance) but can implement multiple interfaces (multiple inheritance). An interface is a container like a class but has specific rules that may not apply to classes. The syntax of an interfaces is:

```
access-modifier interface InterfaceName {
  //optional code goes here
}
```

An access modifier could be public or default. The keyword interface is required, followed by the name of the interface.

Example

```
package stadio;
public interface IPerson {
  void setName(String name);
  public String getName();
  default void display()
  {
     System.out.println("Name: "+getName());
  }
}
```

Note

The following rules apply to interfaces:

1. All the members of an interface are implied to be public, whether or not you mark them as such.
2. All the methods of an interface are implicitly public and abstract (only default methods are not abstract).
3. An interface can contain default methods beginning from Java 8. A default method is marked as such and must be implemented in the interface.
4. Default methods do not have to be overridden.
5. All the non-default methods of an interface must be overridden (implemented) by every non-abstract class that inherits (implements) the interface.
6. An interface cannot be instantiated, nor does it contain a constructor.
7. An interface cannot contain instance variables. All variables in an interface are implied to be static, public and final.
8. All variables declared in an interface must be initialised as they are constants.
9. An interface can extend but not implement another interface.
10. An interface can extend multiple interfaces.
11. An interface cannot inherit a class, it is a class that inherits an interface.
12. A non-abstract class that implements an interface must implement all the non-default methods in that interface and all the other interfaces inherited by that interface.
13. You can create a reference variable of type interface and use it to reference objects of any class that implements that interface.

Summary

This topic covered some important concepts of OOP – abstraction, inheritance, method overloading and overriding, class relationships, polymorphism, encapsulation and interfaces. You have learnt that abstraction is implemented through abstract classes and interfaces.

An abstract class is a class marked as such and can contain zero or more abstract methods, where an abstract method is a method prototype. One class can acquire the non-private members of another class and this feature is called

inheritance. Only non-final classes can be inherited. Java allows you to override and overload methods. While method overloading can happen in inheritance, where a subclass can redefine an inherited method but provide a different parameter lists or return type, overriding only happens in a subclass. Thus, method overloading can happen in the same class but method overriding always happens in a subclass.

There are basically two class relationships – inheritance and composition. These are represented by the "IS-A" and "HAS-A" relationships, respectively. The two Java keywords associated with inheritance are "extends" and "implements", where a class extends another class and an interface extends another interface but a class implements an interface. Interfaces cannot be instantiated, meaning you cannot create an object of an interface.

Self-Assessment Questions

1. Create:
   (a) an interface called ISavingAccount with the method printAccountInfo that returns nothing and has no parameters.
   (b) an interface called ICurrentAccount with the method printAccountInfo that returns nothing and has no parameters.
   (c) a concrete class Account that implements ISavingAccount and ICurrentAccount. This account must have the fields to store the account number, account name and accountType. All the fields must be encapsulated. Provide getters (accessors) and setters (mutators) for all fields.
   (d) a Driver (main) class in which you will accept the Account details from a user, set them using properties and invoke the right method depending on account type entered by a user.

2. Create a class called StudentAccount that inherits the class Account in 1 above. The class must override the method printAccountInfo and add the message "This is a student account".

Glossary of terms

Array – a fixed-size data structure containing data items of the same type.
Attribute – a feature of an object representing data about the object.
Class – a user-defined data type containing attributes and methods. It is the backbone of OOP.
Casting – the conversion of data from one compatible type to another. There are two types – explicit and implicit casting.
Constructor – a special method that has the same name as the class name and has no return type, not even void.
Encapsulation – the grouping together of data and behaviour inside a class.
Inheritance – a feature that allows one class or interface to acquire the methods and properties of another.
Loop – a repetitive statement or group of statements. Looping constructs include the while, do..while, for and enhanced for loop.
Method – it is called a function, subroutine, or sub-procedure in other languages. It is a group of statements grouped together to achieve a specific task.
Object – an instance of a class.
Method overloading – having multiple methods with the same name but different parameters and/or return types in the same class or subclasses.
Method overriding – the redefinition of an inherited method in the subclass.
Recursive method/function – a method/function that invokes itself.
Variable – a named memory location inside a computer that stores a single piece of data of a particular type. It can be reassigned, hence, the actual value stored varies.

References

Downey, A.B. and Mayfield, C. (2019) Think Java: How to think like a Computer Scientist, Version 6.1.3, Green Tea Press, MA, USA. Available at: https://greenteapress.com/wp/think-java/

Answers to Self-Assessment Questions

TOPIC 1 SELF-ASSESSMENT ANSWERS

1.

```
package assessment1;
import javax.swing.JOptionPane;//import statement for JOptionPane
/*1.     Write a program that accepts a user's full name and displays a custom
         welcome message. E.g. if the person enters Stadio University as their
         name,display the message Welcome Stadio University. */
public class Welcome {
   public static void main(String[] args) {
       /*note that you could still use the Scanner class instead of JOptionPane*/
      String fullName = JOptionPane.showInputDialog("Enter your full name");
   System.out.println("Welcome "+ fullName);
   }
 }
```

2.

```
package assessment1;
import java.util.Scanner;//needed for the Scanner class
public class Question2 {
   public static void main(String[] args) {
      //declare local variables
       int mark;
      char grade;
      String choice = "yes";
      Scanner sc = new Scanner(System.in);
      /*equalsIgnoreCase() is a method that compares two strings regardless of
          the case*/
      while(choice.equalsIgnoreCase("yes"))
      {
         System.out.println("Enter mark ");
         mark = sc.nextInt();//accept mark
         if(mark < 0 || mark >100)
         {
```

```
            System.out.println("Invalid. Mark is out of range 0 - 100");
            grade = 'F';
            continue;
        }
        else if(mark >=75)
            grade='A';
        else if (mark >=60)
            grade ='B';
        else if (mark >=50)
            grade ='C';
        else
            grade='F';
        System.out.println("Mark = "+mark+".\nGrade = "+grade);
        System.out.println("Do you want to continue? Type Yes or No");
        sc.nextLine();
        choice = sc.nextLine();
    }
    System.out.println("You have opted not to continue. Thank you for using
        my system");
  }
 }
```

3.

```
package assessment1;
import java.util.Scanner;
public class Question3 {
   public static void main(String[] args) {
       String sex;
       Scanner sc = new Scanner(System.in);
       System.out.println("Enter your sex, e.g. male");
       sex = sc.nextLine();
       /*you cannot know in advance the case to be used by a user when typing
          their sex.
       This makes it hard to write the code to check the sex. Let's convert to
          lowercase
       whatever the user input and then compare in lowercase too*/
       switch(sex.toLowerCase())
       {
           case "male":
               System.out.println("Greetings sir");
               break;
```

```
            case "female":
                System.out.println("Greetings madam");
                break;
            default:
                System.out.println("Sorry, the system could not identify this sex");
        }
    }
}
```

4.

Identified errors:

- missing import statement - import java.util.Scanner
- name is String, it cannot store a double
- the word OR is not a Java keyword, syntax error

Corrected version:

```
package stadio;
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int age;
        String name;
        System.out.println("Enter your name ");
        name = sc.nextLine();
        System.out.println("Enter your age ");
        age = sc.nextInt();
        System.out.println("Welcome " + name);
            if(age >= 18)
            System.out.print("You are an adult");
        else if (age < 0)
            System.out.println("Invalid input. Age can't be negative");
        else
            System.out.println("You are still a minor");
            }
    }
```

TOPIC 2 SELF-ASSESSMENT ANSWERS

1. Given the sequence 5,8,11,14, write a recursive function to determine the n^{th} number in this sequence.

Taking 5 as the base case, each successive number is 3 more than the previous.

Thus, the function could summarised as $f(x) = \begin{cases} 5 & x = 0 \\ f(x-1)+3 & , x \geq 1 \end{cases}$

This can be implemented as a complete program as below, where the recursive function's name is f.

```
package assessment2;
import java.util.Scanner;
public class Question1 {
    static int f(int x)
    {
        if(x==0)
            return 5;
        else
            return f(x-1)+3;
    }
    public static void main(String[] args) {
        int num;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a position of the term you want,
            it must be an integer greater than zero");
       num=sc.nextInt();
        System.out.println(f(num));
    }
}
```

2.

```
package assessment2;
import java.util.Scanner;
public class Question2 {
   public static void main(String[] args) {
      int rows;
      String choice;
      Scanner sc = new Scanner(System.in);
      while(true)
      {
         System.out.println("Enter number of rows between 5 and 9 inclusive");
         rows =sc.nextInt();
```

```
        if(rows < 5 || rows > 9)
        {
           System.out.println("Invalid input. Do you want to continue?\nType
         yes or no");
           sc.nextLine();
           choice = sc.nextLine();
           if(choice.equalsIgnoreCase("no"))
              break;
                   }
        else{
           for(int i=rows;i>=1;i--)
           {
              for(int cols=1;cols<=i;cols++)
              {
                 System.out.print(i);
              }
               System.out.println();
           }
        }
     }

   }
   }
```

3.

```
package assessment2;
import java.util.Scanner;
public class Question3 {
/*Write a program that allows a user to create an array
   of integers of their preferred size and asks them to populate it.
   Your program should display the highest
   and smallest elements, as well as the sum and average. */
   public static void main(String[] args) {
      int size, max=0,min=0,sum=0;
      int[] arr;
      float average;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter the size of the array. It must be an integer
          greater than 1");
      size = sc.nextInt();
      if(size < 1)
         System.out.println("Invalid input");
      else{
         //create array of entered size
         arr = new int[size];
         //populate array
         for(int i=0;i<arr.length;i++)
         {
            System.out.print("Enter element : ");
            arr[i]=sc.nextInt();
            //for the first number, take it as min and max
            if(i==0){
```

```
            //take first number as lowest
            min = arr[0];
            //take first number as max
            max = arr[0];
            }
            //calculate running sum
            sum = sum + arr[i];
            if(arr[i]<min)
               min = arr[i];
            if(arr[i]>max)
               max = arr[i];
         }
                  //calculate average
         average = (float)sum/size;
         System.out.println("Sum = "+sum+"\nAverage= "+average+"\nHighest
          number = "+max+"\nLowest number = "+min);
      }

   }

}
```

4.

```
package assessment2;
import java.util.ArrayList;
import java.util.Scanner;
/*Write a program that allows a user to enter any number of names into
an array list and allows them to search for any names. Your program
should also allow users to delete and replace any names as they so wish. */
public class Question4 {
  public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       int choice;
       ArrayList names = new ArrayList();
       while(true){
       System.out.println("**********Main Menu*********");
       System.out.println("1. Add name\n2. Print names\n3. delete name\n4.
          Search");
       choice = sc.nextInt();
       if(choice==1){
       while(true)
       {
          System.out.print("Enter a name : ");
          sc.nextLine();//include this line if a user is not getting a chance to enter
          name
          names.add(sc.nextLine());
```

```
    System.out.println("Do you want to enter another name? 1. Yes\t2.
    No");
    choice = sc.nextInt();
    if(choice==2)
        break;
}
}
else if(choice==2){
    if(!names.isEmpty()){
    for(Object name:names)
        {
        System.out.println(name);
        }
    }
    else{
        System.out.println("Sorry, there are no names to print");
    System.out.println("Do you want to enter another name? 1. Yes\t2. No");
    choice = sc.nextInt();
    if(choice==2)
        break;
    }
}
else if(choice==3){
    if(!names.isEmpty()){
         System.out.println("Enter name to delete");
        sc.nextLine();
        String search=sc.nextLine();
        if(names.contains(search))
        {
           names.remove(search);
        }
        System.out.println("Names after deleting: "+search);
    for(Object name:names)
        {
        System.out.println(name);
        }
    }
    else{
        System.out.println("Sorry, there are no names to delete");
    System.out.println("Do you want to enter another name? 1. Yes\t2.
    No");
    choice = sc.nextInt();
```

```
            if(choice==2)
                break;
                  }
        }
        else if(choice==4)
        {
          if(!names.isEmpty()){
                System.out.println("Enter name to search");
                sc.nextLine();
                String search=sc.nextLine();
                if(names.contains(search))
                {
                   System.out.println("Name found at index :
          "+names.indexOf(search));
                }
                else
                  System.out.println("Sorry, name not found");
                   }
            else{
                System.out.println("Sorry, there are no names to search");
            System.out.println("Do you want to enter another name? 1. Yes\t2.
            No");
            choice = sc.nextInt();
            if(choice==2)
                break;

            }
        }
    }
    }
}
```

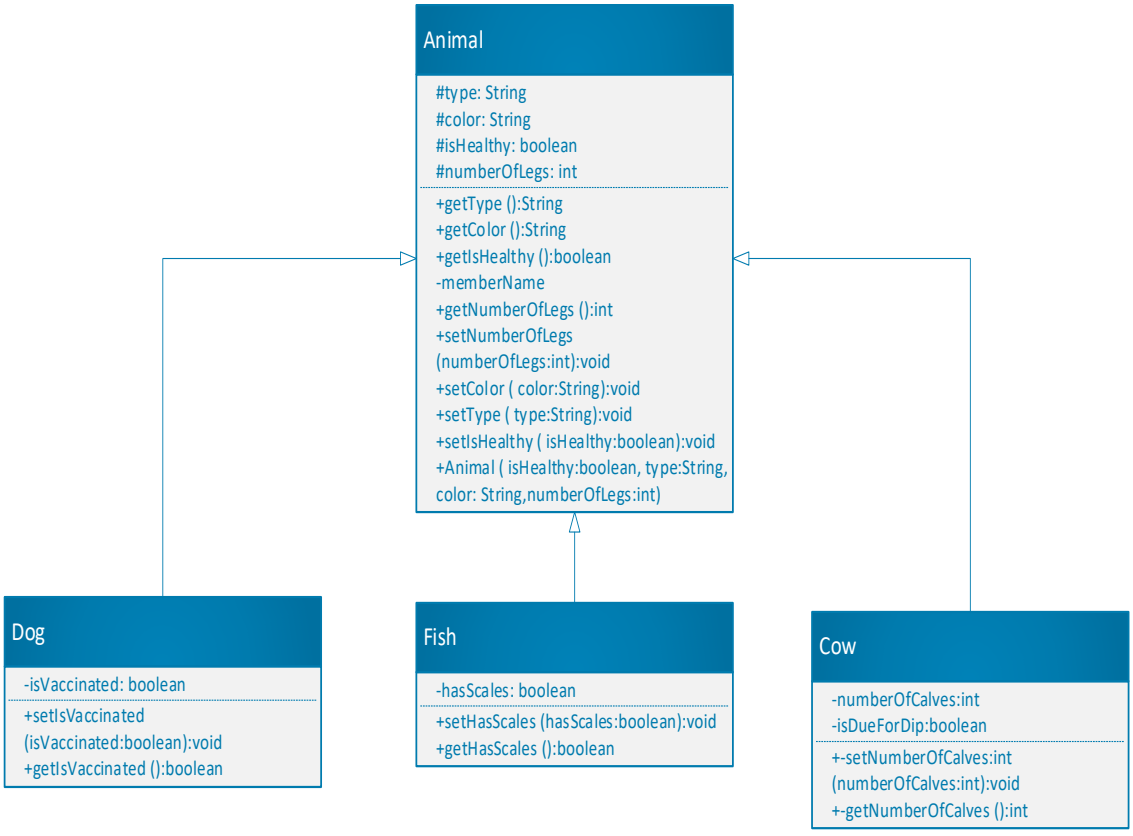
5.

```
package assessment2;
import java.util.Scanner;
/*Write a Java program that allows a user to create a string array of their preferred size, populate it and search for any elements. The search must be case insensitive, that is, if the name is John and the user enters jOhn, it should still state that the name exists. */
public class Question5 {
    public static void main(String[] args) {
        int size;
        String[] names;
        String  search;
```

```
        Scanner sc = new Scanner(System.in);
        System.out.println("How many names do you want to enter?");
         size = sc.nextInt();
         names = new String[size];
         for(int i=0;i<size;i++)
         {
            System.out.println("Enter name ");
            sc.nextLine();
            names[i]=sc.nextLine();
         }
         System.out.println("Enter name to search");
         search = sc.nextLine();
         for(String name:names)
         {
            if(name.equalsIgnoreCase(search))
               System.out.println("Name "+search+" exists");

          }

    }

}
```

TOPIC 3 SELF-ASSESSMENT ANSWERS

1.



2.

```
//class to deal with null objects of Animal
public class NoAnimal extends Animal{
    @Override
   public String getType()
 {
       return null;
   }

    @Override
   public String getColor()
 {
       return null;
   }

   public boolean isExists()
  {
       return false;
   }
}
```

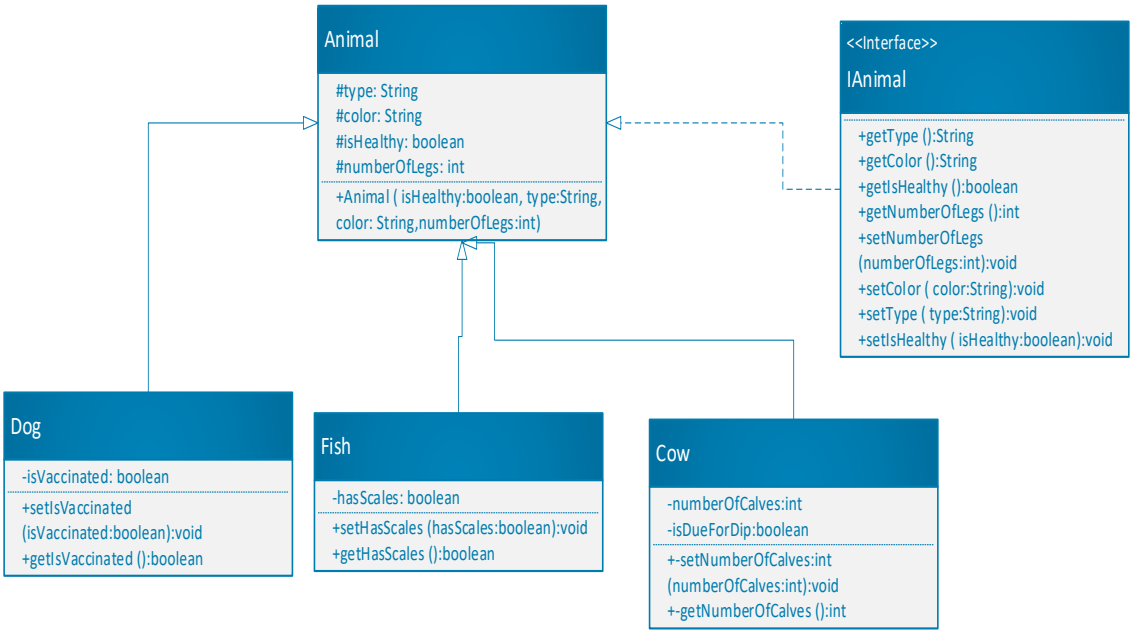
3.

```
package assessment3;
public class Square {
   private float side;
   public void setSide(float side)
   {
      this.side=side;
   }
   public float getSide()
   {
      return this.side;
   }
   public Square(float side)
   {
      this.side=side;
   }
   public float area()
   {
      return side*side;
   }
   public void display(Square s)
   {
      System.out.println("Lenght of each side = "+side);
      System.out.println("Area = "+area());
```

```
    }
}
```

TOPIC 4 SELF-ASSESSMENT ANSWERS

1.



2.

```
package assessment4;
public class Animal implements IAnimal{
    protected String color, type;
    protected int numberOfLegs;
    protected boolean isHealthy;
     public String getType(){
         return type;
     }
     public Animal(String color, String type,int numberOfLegs,boolean isHealthy)
     {
         this.type=type;
         this.color = color;
         this.numberOfLegs=numberOfLegs;
         this.isHealthy=isHealthy;
     }
    @Override
   public String getColor()
   {
      return this.color;
   }
```

```
    @Override
   public boolean getIsHealthy()
   {
      return isHealthy;
   }
    @Override
   public int getNumberOfLegs()
   {
      return numberOfLegs;
   }
    @Override
  public void setType(String type)
   {
      this.type=type;
   }
    @Override
   public void setColor(String color)
   {
      this.color=color;
   }
   public void setIsHealthy(boolean isHealthy)
   {
      this.isHealthy=isHealthy;
   }
    @Override
   public void setNumberOfLegs(int legs)
   {
      this.numberOfLegs=legs;
   }
}
```

```
package assessment4;
public class Dog extends Animal{
    private boolean isVaccinated;
    public    Dog(String    color,    String    type,int    numberOfLegs,boolean
          isHealthy,boolean isVaccinated)
    {
       super(color,type,numberOfLegs,isHealthy);
       this.isVaccinated=isVaccinated;
    }
}
```

```
package assessment4;
public class Fish extends Animal{
    private boolean hasScales;
    public    Fish(String    color,    String    type,int    numberOfLegs,boolean
          isHealthy,boolean hasScales)
    {
       super(color,type,numberOfLegs,isHealthy);
       this.hasScales=hasScales;
    }
}
```

```
package assessment4;
public class Cow extends Animal{
   private boolean isDueForDip;
   private int numberOfCalves;
   public Cow(String color, String type,int numberOfLegs,boolean
         isHealthy,boolean isDueForDip,int numberOfCalves)
   {
      super(color,type,numberOfLegs,isHealthy);
      this.isDueForDip=isDueForDip;
      this.numberOfCalves=numberOfCalves;
   }
}

package assessment4;
import java.util.Scanner;
public class Driver {
   public static void main(String[] args) {
      String color, type;
      int numberOfLegs,numberOfCalves;
      boolean isHealthy, isDueForDip,isVaccinated,hasScales;
      int choice;
      Animal dog,cow,fish;
      Scanner sc = new Scanner(System.in);
      while(true) //infinite loop
      {
         System.err.println("Select type of animal.\nEnter:\n1 for dog\n2 for
         cow\n3 for fish");
         choice = sc.nextInt();

         if(choice==1)
         {
            System.out.println("Enter color");
            sc.nextLine();
            color = sc.nextLine();
            type = "dog";
            System.out.println("Enter number of legs");
            numberOfLegs = sc.nextInt();
             System.out.println("Is the dog healthy? Enter true or false in
         lowercase");
            isHealthy = sc.nextBoolean();
            System.out.println("Is the dog vaccinated? Enter true or false in
         lowercase");
            isVaccinated = sc.nextBoolean();
            dog = new Dog(color,type,numberOfLegs,isHealthy,isVaccinated);
         }
         else if (choice==2)
         {
            System.out.println("Enter color");
            sc.nextLine();
            color = sc.nextLine();
            type = "cow";
            System.out.println("Enter number of legs");
            numberOfLegs = sc.nextInt();
```

```
            System.out.println("Is the cow healthy? Enter true or false in
        lowercase");
            isHealthy = sc.nextBoolean();
            System.out.println("Is the cow vaccinated? Enter true or false in
        lowercase");
            isVaccinated = sc.nextBoolean();
             System.out.println("Enter number of calves");
            numberOfCalves = sc.nextInt();
            System.out.println("Is the cow ready for dip? Enter true or false in
        lowercase");
            isDueForDip = sc.nextBoolean();
            cow                                =                                new
        Cow(color,type,numberOfLegs,isHealthy,isDueForDip,numberOfCalves);
        }
        else if(choice==3)
        {
            System.out.println("Enter color");
            sc.nextLine();
            color = sc.nextLine();
            type = "fish";
            numberOfLegs = 0;

             System.out.println("Is the fish healthy? Enter true or false in
        lowercase");
            isHealthy = sc.nextBoolean();

            System.out.println("Does the fish has scales? Enter true or false in
        lowercase");
            hasScales = sc.nextBoolean();
            fish = new Fish(color,type,numberOfLegs,isHealthy,hasScales);
        }

        System.out.println("Do you want to continue? Type 1 for yes, 2 for no");
        choice = sc.nextInt();
        if(choice==2)
            break;
        System.out.println("Thank you for using our system");
    }
  }

}
```

TOPIC 5 SELF-ASSESSMENT ANSWERS

1.

```
package assessment5;
public interface ISavingsAccount {
    void printAccountInfo();
}
```

```
package assessment5;
public interface ICurrentAccount {
   public abstract void printAccountInfo();
}

package assessment5;
import java.util.Scanner;
public class Account implements ICurrentAccount,ISavingsAccount{
   private long accountNumber;
   private String accountName,accountType;
 /*a concrete class Account that implements ISavingAccount and
   ICurrentAccount. This account must have the fields to store the
   account number, account name and accountType. All the fields must be
         encapsulated.
   Provide getters (accessors) and setters (mutators) for all fields. */
  public void setAccountNumber(long accountNumber)
  {
     this.accountNumber=accountNumber;
  }
   public long getAccountNumber()
  {
     return this.accountNumber;
  }
  public void setAccountName(String accountName)
  {
     this.accountName=accountName;
  }
  public String getAccountName()
  {
     return this.accountName;
  }
  public void setAccountType(String accountType)
  {
     this.accountType=accountType;
  }
  public String getAccountType()
  {
     return this.accountType;
  }
   @Override
 public  void printAccountInfo(){
    System.out.println("Account number : "+accountNumber);
    System.out.println("Account name : "+accountName);
    System.out.println("Account type : "+accountType);
 }
public void getInput()
{
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter Account type : ");
   this.setAccountType(sc.nextLine());
   System.out.println("Enter Account name : ");
   this.setAccountName(sc.nextLine());
   System.out.println("Enter Account number : ");
```

```
    this.setAccountNumber(sc.nextLong());
}
}
package assessment5;
public class Driver {
    public static void main(String[] args) {
        Account acc = new Account();
        acc.getInput();
        acc.printAccountInfo();
    }
 }
```

2.

```
package assessment5;
public class StudentAccount extends Account{
    @Override
    public void printAccountInfo()
    {
        super.printAccountInfo();
        System.out.println("This is a student account");
    }
}

package assessment5;
public class Driver {
    public static void main(String[] args) {
        StudentAccount acc = new StudentAccount();
        acc.getInput();
        acc.printAccountInfo();
    }

}
```