OBJECT ORIENTED PROGRAMMING

FOR

SOFTWARE ENGINEERING SOPHOMORE

ADEKOLA OLUBUKOLA D. (PHD, FNCS, MCPN)

Object Oriented Programming for Software Engineering Sophomore

By

Adekola Olubukola D. ***(PhD, FNCS, MCPN)***

2021

FORWARD

The rapidly changing technological landscape is bringing to the fore, the need for the curriculum for computer science and especially software engineering students to introduce them early enough to the concepts of Object Orientation. It is described as “orientation” because Object Oriented Design and Programming or OOP simply put is not actually about a new language in town, but rather about a new way of thinking, a new way of solving problems such that tasks are accomplished quicker, smarter and also leaving behind reusable artefacts to solve future problems.

The book Object Oriented Programming for Software Engineering Sophomore takes a unique style in presenting the essential topics in OOP and its idiosyncrasies. A journey through this book would usher the reader from the basics to complex concepts of object technology with ease. It is advised that students pass through the solved examples, try them out on a compiler to see for themselves and also task themselves with the numerous “Tasks” in the pages in order to be well-grounded with the concepts. The use of Java programming language to teach this concept is second to none in that it exposes enough details of procedural and object oriented concepts and gives the reader leverage when it comes to undertaking a new language.

This is one of the well-written contents that would develop the capacity of our undergraduate students and spur them to changing our world.

Oludotun Oluyade,

Chief Software Architect,

PearlSoft Nig. Ltd (2008 – Till Date)

dotun.oluyade@pearlsoftng.com

www.pearlsoftng.com

WORDS FOR MEDITATION

- [even] to the time of the end: many shall run to and fro, and knowledge shall be increased- Daniel 12:4
- Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty- Donald Knuth
- What we learn with pleasure we never forget- Alfred Mercier
- Intelligence is the faculty of making artificial objects, especially tools to make tools – Henry Bergson

TABLE OF CONTENTS

General Objectives:

At the end of the course, students should be able to:

- understand and appreciate the concepts of structured and object-oriented techniques
- effectively use primitive data types and create their own data types
- gain mastery of program control structures
- Create their own reusable classes and methods
- Specifically solve problems using Java programming language tools

Specific Objectives: By the end of this course, you will be able to:

i. differentiate between Procedural and Object oriented programming (OOP)
ii. identify new features that made Java better
iii. understand the concept of OOP analysis, design and implementation
iv. optimise design and code reusability inherent in OOP (Java example)
v. design and develop functional, seamless and quick to deliver software

Expectations and Assumptions

i. Being students in their sophomore year (i.e. second year), prior knowledge of basic/elementary programming skills are required.
ii. There may be more than one solutions to a particular problem, however, students should always choose the optimal solution after considering known constraints and conditions.

Practical Tools:

i. JDK (To be used at the introductory level)

ii. Netbeans, eclipse or any good integrated development environment (IDE) software.

iii. Microsoft Word or Visio for drawing analysis.

Chapter 1
GETTING STARTED

Preamble:
This content introduces the reader to the idea behind object oriented programming techniques using Java programming language as implementation tool/example. This chapter will introduce students to Java language basics especially for those who are not familiar with the language syntax and peculiarity. Subsequent chapters will dwell more on the concept of object oriented programming.

1.1 The Origin and the Need for Java

Improvements in the art of programming and changes in the computing environment are the two common denominators behind computer language innovation and evolution. Java came to flourish upon the rich legacy inherited from C and C++, then added refinements and features that reflect the current state of the art in programming.

The original momentum for Java was not the internet but the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices, e.g. microwave ovens, hand-held devices etc. Also, the web required a portable program i.e. needed a new way to program and Java offers features that streamline programming for a highly distributed architecture; cross-platform, which goes a long way supporting our online community- Java became popular for meeting this need. The original idea; Java was the conception of James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991. It was initially called “Oak’s language” but was renamed “Java” in 1995.

Sun Microsystems was bought by Oracle in 2010.

The challenge was that most computer languages were designed to be compiled for a specific target. And again compilers are expensive and time-consuming to create. In an attempt to find a better solution, Gosling and others worked on a portable, cross-platform language that could produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the invention of Java. Java program can be written once and then run on many different devices, hence the slogan “write once, run anywhere” –WORA. Java’s exponential success began from the hunt for architecture-neutral programming language while the Internet crowned it all.

1.2 The World Wide Web (www)

www is a Graphical User Interface to the Internet. It is based on pages which can contain information of many different kinds as well as links to other pages. These information are viewed with programs called web browser e.g. Internet Explorer. The pages that you view with a web browser are files stored on some computers connected to the Internet. The web browser used contacts of the computer on which the page is stored and transfer it to your computer using a protocol called HyperText Transfer Protocol (HTTP). Special Java networked programs named applets are transmitted over the internet and displayed on web pages. A web browser that includes an interpreter for the Java can run applet right on the web page. With Java, a web page becomes more than just a passive display of information.

1.3 Java Rich Content

Java inherits the modern programming concepts from C as well as its syntax, and then its object model adapted from C++. Rather than reinventing the wheel, it further refined the already successful programming paradigm and added new feature required by the online environment.
Java and C++ have many similarities especially their support for object-oriented programming, but Java has significant practical and philosophical differences. It is neither upwardly nor downwardly compatible with C++. Java was designed to solve a certain set of problems.

1.4 Java Applet

An applet is a Java program designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. If a user opens a link that contains an applet, the applet will be automatically downloaded and run in the browser. They are that execute locally, rather than on the server. This means the applet allows some functionality to be moved from the server to the client. An applet is a dynamic, self-executing typically used to display data provided by the server, handle user input, or provide simple functions program. It is an active agent on the client computer, though initiated by the server.

1.5 Bytecode

Java was able to solve both the security and the portability problems because the output of a Java compiler is not executable code but *bytecode*. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). JVM is an abstract computer implemented in software.
JVM was originally designed as an interpreter for bytecode unlike many modern languages compiled into executable code. With JVM executing Java code solves the big challenge of web-based programs. The bytecode can run on a wide variety of environments and only the JVM needs to be implemented for each platform. That is, the details of the JVM will differ from one platform to another, but all understand the same Java bytecode. The execution of bytecode via JVM makes creating portable programs easy. Unlike other High Level languages that you write programs for different platforms e.g. Fortran for windows, Fortran for unix, there is no Java program for windows or unix or any device. Java program are written to run on a JVM.
Also, with JVM in charge, it can contain the program and prevent it from generating side effect outside of the system -achieving security. Remember that downloading a program comes with the risk of getting a virus or malicious code. Java achieved preventing applet from launching such an attack by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. Also, certain restrictions present in the Java language enforce safety.

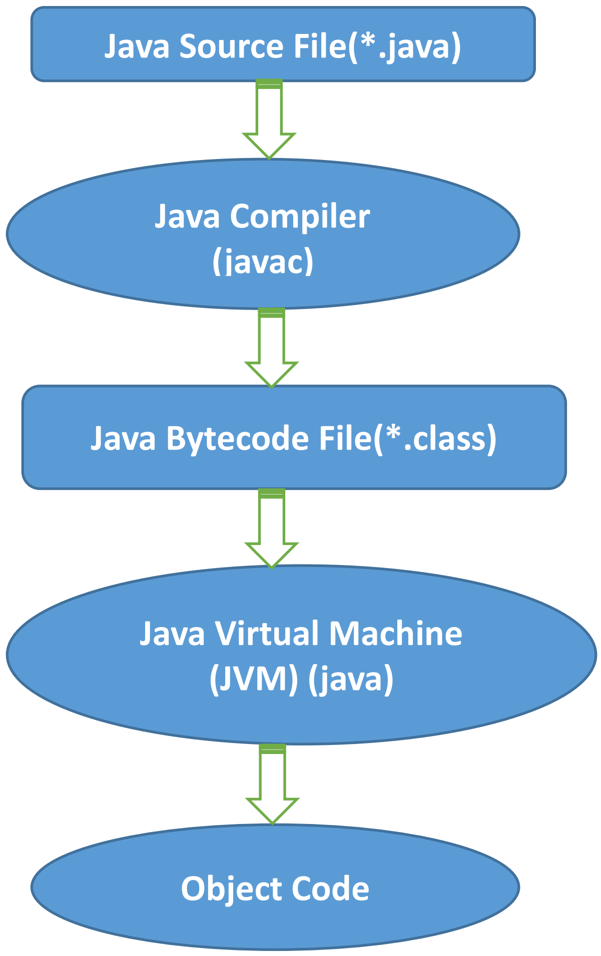

Java Source File(*.java)
Java Compiler (javac)
Java Bytecode File(*.class)
Java Virtual Machine (JVM) (java)
Object Code

Figure 1: Java Program Execution Phases

1.6 Java other features

Core assets in Java are basically portability and security. Other important features that made name for java language include:

- Simplicity – *easy to learn and use*
- Object-oriented philosophy- *a variation from procedural*
- Robust- *runtime checks, strongly typed*
- Multithreaded- *support for multithreaded programming*
- Architecture-neutral- *not tied to a specific OS or machine*
- Interpreted- *platform independent with bytecode*
- High performance- *bytecode is fast*
- Distributed – *support distributed network; Internet*
- Dynamic – *attend to runtime issues*

1.7 Java Editions

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions:

- Java *Standard Edition (Java SE) to develop client-side standalone applications* or applets.
- *Java Enterprise Edition (Java EE) to develop server- side applications, such as Java* servlets ,JavaServer Pages (JSP), and JavaServer Faces (JSF).
- *Java Micro Edition (Java ME) to develop applications for mobile devices, such as* cell phones.
- Java SE will be used to introduce Java programming.
- Java SE is the foundation upon which other technologies are based

1.8 Installing Java Compilers and Running Programs:

- To compile and run a Java program, you need to install a Java Development Kit, JDK. JDK comprises two major programs: the *javac* (i.e. the Java Compiler and the *java* (i.e. the standard Java interpreter, also called the *application launcher).* The JDK runs in the command prompt environment and uses command-line tools. It is not a window-based application.
- Besides the JDK described above, there are many Integrated Development Environment (IDE) that can be used to alternatively run our Java codes. Examples include NetBeans and Eclipse.
- For Java Development Kits, JDK(which is free), visit www.oracle.com/technetwork/java/javase/downloads/index.html
- An IDE can be very helpful when developing and deploying commercial applications. JDK is preferred for beginners because instructions for using the JDK are the same for all readers. While, each IDE has its own peculiar guide/ instructions.
- The Java program can be created using any text editor (textpad, notepad, Emacs, etc) and then compile your program with the Java Development Kit (JDK)
- To create the program:
  - *Open a text editor.*
  - *Type your program*
  - *Save the program in file with a java extension, e.g MyProgram.java*

1.9 Running Programs

To compile the java Program (in Windows Environment)

- Load command prompt:
- Click start
- Click run
- Type cmd, in the next box that appears
- Click ok
- Navigate to the directory where you have saved your program
- Then type the command: javac filename.java e.g javac Myprogram.java in the directory
- If the program has no error, the compiler creates a class file, using the filename but with the .class extension. E.g Myprogram.class
- The class file created contains the bytecode that is to be run on the JVM.

To Run the Java Program

to execute the class file, type the command:

```
java Myprogram
```

This will execute the bytecode on the JVM

Summary

To compile: javac Myprogram.java

To execute: java Myprogram

My First Program:

```
/*
This is my first attempt at coding in
Java. It is truly the first.
*/
class MyFirst {
// A Java program begins with a call to main()
    public static void main(String args[]) {
            System.out.println(“Can you see my first try?");
    }
}
```

Steps in running a program in Java

a) Enter the Program:

You need to type the program into your computer using a text editor, not a word processor. Word processors(e.g. MSWORD) typically store format information along with text. While a text editor like notepad/wordpad is unformatted / plain. This is because the format information will confuse the compiler.

Unlike most programming languages, the filename given to your program is NOT arbitrary (should not be anything) but must be the same as the **class** containing the **main** method.

Java source file is a *compilation unit that contains*

one or more class definitions plus other things as much as possible. Java compiler requires that a source file use the **.java filename** extension. Note, In Java, all code must reside inside a class. By *convention*, the name of the main class should match the name of the file that holds the program. Also ensure that the capitalization of the class name matches the filename. However, this convention makes it easier to maintain and organize your programs. Failure to follow this pattern will showcase trouble at compile time.

b) Compile the Program:

To compile the program(MyFirst.java) above, just type the following at the command-line:

```
javac MyFirst.java
```

This makes the java compiler(javac) to compile the program saved as MyFirst.java.

Javac generates a file called MyFirst.class that contains the bytecode version of the program.

c) Run the Program:

Recall that Bytecode is not executable code. It must be executed by a Java Virtual Machine, JVM. So to run it, use the Java interpreter, java. Therefore, pass the class name MyFirst (not with ".java") as a command-line argument, as follows:

```
java MyFirst
```

So if the program successfully executes, you will have the following output:

```
Can you see my first try?
```

At compilation, each individual class is put into its own output file named after the class and using the **.class extension**. This is why you need to give your Java source file the same name as its main class. When you execute the Java interpreter as just shown, you are actually specifying the name of the class that you want the interpreter to execute. It will automatically search for a file by that name that has the **.class** extension and execute it. But if your file name is arbitrary and different from its main class name the following illustration is your consequence:

Let's assume your source code is named/saved as **FirstAttempt.java** instead of **MyFirst.java** (as in our initial example). **Javac** compiles the code successfully and creates **MyFirst.class** aligning with the class name where the main methods resides. So when you type "java FirstAttempt", the JVM will not find FirstAttempt.class and so will flag error. This is because the bytecode created was **MyFirst.class** and NOT **FirstAttempt.class**. It means FirstAttempt has no bytecode generated and as such cannot run. If you must still run this program, though successfully compiled, then you must type: java MyFirst (which tallies with the class name.)

Setting the environment variables

Having installed the JDK correctly on your computer system, you may discover that your program(written error-free) would not compile. It is most likely that your computer cannot find javac i.e. the compiler. At this juncture, you may need to specify the path to the command-line tools. That is, from the directory where your source code is, you will type the following to fully specify where the computer should locate the java compiler and interpreter tools:

- C:\Program Files\Java\jdk1.7.0\bin\javac MyFirst.java
- C:\"Program Files"\Java\jdk1.7.0\bin\java MyFirst

Note that in the above directory(folder) and sub-directories, those tools actually reside in 'bin' sub-directory. Specifying path each time you want run a program could make you prone to errors. Therefore, if you set the environment variable, you do not need to specify the full path to the executable file every time. Having done this, your computer will see the "compile and execute" tools from wherever you want to run the program. Hence, there is no need to specify the path to these tools again. If you are using Windows, you will need to add the path to the command-line tools to the paths defined for the PATH environment variables.

Steps to setting the PATH variable:

- Open Control Panel -> System -> Advanced System Settings - >Advanced tab ->Environment variables

- Then select PATH, click Edit button.
- Append the java bin directory to the end of the existing path or paths e.g.
- C:\Program Files\Java\jdk1.7.0\bin.
- Note that one path is separated from the other on that same line with a semicolon (';') symbol. It means you will type semicolon before adding the path stated above. You may need to consult the documentation for your operating system on how to set the path, because of variation between Operating systems.

Setting the CLASSPATH:

- *Set the CLASSPATH environment variable value*
- *of CLASSPATH to be a single period (.)*
- *Also, multiple paths in CLASSPATHS are*
- *separated from each other with semicolons (;)*

JAVA_HOME environment variable:

- When required, JAVA_HOME should be set to the root directory of your Java installation, e.g.,
- C:\Program Files\Java\jdk1.7.0   or
- C:\jdk_1.5.0_08

It is strongly advised that you choose an installation directory that does not include spaces in the path name (e.g., C:\Program Files). Else, you may have issues if you set JAVA_HOME variable to a path that does not include spaces. This may result in throwing of exceptions by some programs that depend on the value of JAVA_HOME.
So, to compile your source code from "C:", all you need do is:

C:\Javac MyFirst.java
and no longer:
C:\"ProgramFiles\Java\jdk1.7.0\bin\javac" MyFirst.java

Note: With JDK 17 version, you do not need to set environment variable by yourself. All you need do is to just install the JDK and start to use.

1.10 Explaining My First Program:

The following /*…*/ is a multiline comment in Java. This is one of the syntax gained from C/C++. The compiler ignores anything written within /*…*/ because they are regarded as non-executable line/ lines. Note that this could be of several lines. While the // is a single line comment:

```
/*  This is my first attempt at coding in
       Java. It is truly the first.
*/
class MyFirst{
// A Java program begins with a call to main() method
    public static void main(String args[]) {
        System.out.println("Can you see my first try?");
    }
}
```

- The word **class** is a keyword used to define **MyFirst** as the newly created class. It simply means the classname is **MyFirst**. You cannot use the word class as your personal variable or method name because it has a predefined function in Java. A class is Java's basic unit of encapsulation. The class definition begins with the opening curly bracket ({) and ends with the closing curly bracket (}).
- The elements between the two braces are members of the class:

  class MyFirst{ …}

- The single line comment is used to present a line comment or line by line comment within the program. Each "//" effect ends at the end of each line. Remember that comment are only used as internal documentation, i.e. to explain codes to whoever is reading it:

  // A Java program begins with a call to main()

- As stated earlier, a Java program is made up of a class or several classes. A class contains member variables and methods. A program usually has one class consisting the main method. Note subroutine or function is called method in Java. Method **main** is where Java Program begin to execute:

  public static void main(String args[]) {…}

- The **public** keyword is an access modifier. An access modifier determines how other parts of the program can access the members of the class. When a class member is preceded by public, then that member can be accessed by code outside the class in which it is declared.
- (The opposite of public is **private**, which prevents a member from being used by code defined outside of its class.)
- Therefore, **main**( ) must be declared as public, because it has to be called by code outside of its class.
- The keyword **static** allows main() to be called before an object of the class is created. That is, declaring the *main*() method as **static** allows Java Virtual Machine (JVM) to invoke it without creating an object or an instance of the class. This is necessary because main() is called (when java application begins) by the JVM before any objects are made. The keyword **void** simply tells the compiler that main( ) does not return a value. As you have been thought that a method may return a value e.g. integer, float, char, string values etc.
- The variables declared within the parenthesis of a method name are called **Parameters**- as seen immediately after the method name.

Illustrating with the example above, any information you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method.

Consider the following typical C language code extract:

1. int score; // variable to be used is declared
2. void SumProg(int score); // function is declared
3. Scanf("%d",&score); /* value is read in and stored in variable named score */
4. SumProg(score); /* function is called to execute. Note the
5. value read in to variable score is now passed in to the function SumProg() as argument or parameter- which will be used by the function. The implementation of SumProg() is defined somewhere else */

Where no parameters are needed, the method is followed by empty parentheses. The only parameter in main() is String args[], which declares a parameter named args to be of type String. "args" is an array of objects of type String. As you know, elements like variable, function, method, array etc. must belong to one data type or the other. (An *Array is collection of data/objects of the same type). Objects* of type String store sequences of characters (e.g. "adekola"). In this case, **args receives any command-line** arguments present when the program is executed. We shall discuss in the future where this is very relevant.
The next line of code is inserted inside method main:

System.out.println(" Can you see my first try?");

This line displays/outputs the string "Can you see my first try?" followed by a new line on the screen. **println()** is a built-in method that displays the output. For now, System is a predefined class that provides access to the system, and out is the output stream connected to the console. Hence, ***System.out*** is an **object** that encapsulates console output.

We shall later discuss what is used in generating I/O instead of console I/O methods convenient for real-world Java applications.
Note, all technical statements in Java ends with a semicolon(;). So, you have (;) after println.
Finally, all open braces, {,must be closed/ended. Therefore, the first } is for the main and the second/last (}) ends the class.
Like C/C++, Java is case sensitive. Therefore, println is not Println, MyFirst is different from Myfirst. Violating this rule lands you a **syntax error**, hence, your code may not compile.

Generally, when a syntax error is reported, the compiler may not reveal to you expressly what the error is. Therefore, you may not have to take these reports at face value. You can, of course, have a clue of what the problem is but sometimes it may not be so. Hence, your experience in finding and fixing code errors(debugging) will go a long way for you. People who are familiar with C/C++ compilers experience similar situations.

Moreover, if you type Main() instead of main(), then the compiler will not find a main method in your program. Although the Java compiler *will compile* classes that do not contain a main( ) method, but it won't be able to execute/run them. So, if you had wrongly typed main, the compiler would still compile your program. However, the Java interpreter would flag an error because it would not find the main() method. Your mistake simply means the program has no “main() method”.

Task 1.1

a) Write a program in Java to find the area of a rectangle. Ensure you run the program using only JDK and notepad via the command prompt.

b) Write a Java program to find the square of a number.

Chapter 2

BASIC ELEMENTS OF JAVA

This chapter discusses the basic elements of Java program. The general common contents in Java program include the following namely: comment, header, keyword, delimiter, data type, method, variable, constant etc.

2.1 Comments in Code

When you buy a device, it comes with manuals explaining how to use. This is referred to as **external documentation**. Writing comments in code to explain a line of code or module or some logic is referred to as **internal documentation.** The main purpose of this is readability and understandability. Comments are generally non-executable statements which are skipped by the compiler during code execution. Comments put in the right points of a program enhance program readability both for the code creator or others. It is not surprising that you revisit a function written sometimes in the past only to discover that you find it difficult to understand the logic put in the module by yourself to solve a particular problem. How much more difficult that could be if the program was written by someone else called to maintain such code. Comment can be placed anywhere necessary in a program.

Two Common Types of Comments

1) **Single line comment**:

Compound symbol // (two forward slashes) –makes anything written after it becomes a comment till the end of the line. E.g.

```
// This module adds the numbers
```

2) **Multi-line comment:**

Begins with a compound symbol /* and ends with the compound symbol */

Anything written in between this symbol is regarded as comment and can span as many lines as possible. E.g.

```
/* The program output the sum of
   the square*/
```

2.2 Identifiers

Identifiers are unique names given to different entries or elements such as variables, constants, methods/functions, classes, etc. Some are pre-defined while some are named by the programmers following some rules.

Rules for Creating Valid Identifier in Java:

- Only alphabetic characters, digits and underscores(_) and dollar sign ($) are permitted.
- First letter must be an alphabet, underscore (_) dollar sign ($) .
- Identifiers are case sensitive.

- The name has no spaces in it
- Reserved keywords must not be used as an identifier (e.g. void).
- Functions with predefined meaning should be avoided (e.g. println)
- Identifier length can be as long as you prefer but too long may not look good.

a) **Variable Identifiers**: are names used to temporarily hold data on computer memory. The format for variable declaration is:

- *datatype variable-name;*

e.g. int ***number*** = 5; //***number*** is a variable initially set to 5 and can be changed or updated.

b) **Constant Identifiers**: Names used to permanently hold constant values or literals. These, once declared cannot be altered (i.e ***immutable)*** throughout the execution of the program.
You can define constants in Java by using **final** keyword:

E.g. **final** double PI = 3.142; PI -->the constant identifier

This means once the value is set (or initialized), the value of the data field cannot subsequently be changed. With this restriction, there is less need to hide access to a data value behind a method.

Valid Variable Identifiers in Java

i. unit_cost
ii. y
iii. totalCost
iv. Y2
v. $uxy
vi. _tape
vii. Bottle306

To improve code readability and maintainability, a good programming practice demands that you form identifier names that reflect the meaning or usage of the items being named.

Valid Variable Identifiers in Java

i. 2sum – numeric digit don't start an identifier
ii. println - word with predefined meaning is not used
iii. Total Cost – having a blank space between a compound name is invalid
iv. Y+2 – operator plus in between characters make is invalid
v. else – is a reserved word and so invalid

More on Variables in Java:

- In Java a variable has some attributes (names, type, value, lifetime, size etc)
- A program refers to a variable by its name and not by its address.
- In Java a variable must be declared before it can be used.

Example Declaration 1 (declaration then assignment):

int number;

number = 5;

Example Declaration 2 (declaration and assignment together):

int number = 5;

Description:

- *Data Type: int*
- *Name: number*
- *Value: 5*
- *Mem. Size: 4 bytes*

Usually, variable names start with lower case

- For declaration of many variables of the same type we can have:

  int i, j, count, total; //each separated by comma and ended by semicolon

- Good programming practice dictates that variable names should be reflect the meaning of the data they hold. CamelCase or hunchback notation is recommended. E.g. basicSalary, totalUnit, for compound names instead of just using a, b, c etc. This promotes readability and in fact, it is used in language processing techniques when developing program metrics.

Variable Type Casting:

Integer value can be assigned to variable of type double:

double number = 100; //implicit casting

Java first casts 100 (int) into double 100.0 before assigning it into number

Conversely, it is illegal to assign a floating point value to an integer variable:

int secondNumber = 67.5;

Furthermore, you need to know when to do **explicit type cast** to avoid incorrect or output error. This helps maintain data integrity:

E.g.:

int x = 18;

int y = 5;

double z;

z = x / y; //this means integer division though z is double

System.out.println("z is = " + z); // z is = 3.0

//we typecast into double to get the accurate result as follows:

*z = **(double) x** / y;*

System.out.println("z is = " + z); // z is = 3.6

Other examples:

(1) (int) 3.14592 casts a floating point number to integer value of 3

(2) (double) (5+6) = (double)(11) = 11.0

- Explicitly casting a variable does not change the content of that variable. E.g. Example; double y = 4.7;

  int x = (int) y;

This truncates 4.7 to 4 and gives 4 to x but leaves y as 4.7

  - *Smaller data types are promoted (cast) to larger data types.*
  - *Boolean values cannot be cast to other data types nor can other data types be cast to boolean*

2.3 Keywords

Java language keywords are reserved words that define the language and have special or predefined meaning to the compiler and, therefore, may not be used Otherwise.

Eg.

- ***Super*** - to call the constructor of the superclass
- ***If*** - to state an alternative course of action.
- ***private*** -: to make access to a property or method hidden.

The words **true**, **false** and **null** are three reserved words which technically are literal values and not keyword but they cannot be used as identifiers.

2.4 Other Program Elements

- **Literals:** Are values assigned to identifiers. It could be integer, floating points, Boolean, character, string etc. e.g. **5** is an integer literal in the following:

  int bus **= 5;**

- **Operators**: Are symbols used to perform mathematical or logical manipulations. These include:
  a) Arithmetic Operators: e.g. + (addition), % (modulus)
  b) Increment and Decrement Operators: e.g. ++ (increment)
  c) Relational Operators: e.g. > (greater than)
  d) Logical Operators: e.g. && (and) Gives true when both expression are true
  e) Bitwise Operators: e.g. & (binary and operator)
  f) Assignment Operators: e.g. = (assignment) right to left associativity
- **Terminators and other separators:** semi colon (;) (marks the end of a code)

2.5 Method

A Java class is essentially made up of two members, namely, ***member variables*** and ***member methods.*** A method is like a function or subroutine or program module designed to perform a specific or single well-defined task.

A class ***may*** contain two major types of method: the ***main method*** and ***others***. Remember that JAVA program execution starts from the **main** method.

A method consists of a statement/list of statements that a program (class) executes.

Every (running) application must have a main method. The structure of main method header is as follows:

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

The Structure of a *method* in Java

A java method consists of a

a. *Header (method signature) followed by*
b. *Method block (list of codes between open and close curly bracket { })*

Generally, a method header takes the form:

access modifier return-type name (parameter list)

Such that:

- ***Access*** : keywords e.g. public, private, protected or default access (obtained by leaving off the access modifier. public- allows anyone to invoke or call the method from anywhere, private access means no one else can invoke the method except member of its own class. protected access allows members of the same class and subclasses(used in implementing inheritance).
- **Modifier** (or specifier) – optional: e.g. static, final, abstract, native or synchronized. A method may not use none, one or more of these specifiers.
- **Return type**: e.g. int, double. This is the data type of the value that the method returns.
- Return type could be one of the **primitive** data types or a **reference**.
- **Method name:** Is any valid language identifier as given by the programmer.
- **Parameter list**: This is comma separated list of formal parameter that receive values from arguments passed to the method when a method is invoked. Parameters may be empty. A parameter is a **locally scoped** variable declared in a method header that temporarily comes into existence while a method is executing. A parameter consists of a data type and an identifier

//General format of a method:

```
Access modifier  return_type Method_name([Optional Parameter list])
{
        // list of code (executable statements)
}
```

Typical example:

```
public int periMtd(int para1, int para2)
  {
          int val;
          val = 2*(para1 + para2);
          return val;
  }
```

2.6 Class

In Java, the primary unit of encapsulation is the class. A class defines the form of an object. It specifies both the data and the code that will operate on that data. *Therefore, a class is* a set of plans where we specify how an object is built.

The code and data in a particular class are called members of that class. The data defined by the class are referred to as *member variables while the code* that operates on that data is referred to as *member methods*.

A class represents a set of objects that share a common structure and a common behavior. A class simply put, defines an object. Then an object has state, exhibits some well-defined behaviours and has a unique identity (Booch, 1995).

Java format example: (Access modifier is declared for individual member)

```
 Access Specifier class ClassName
{
        private //declaration of private member
        private //declaration of private member
         …
        public //declaration of public member
        public //declaration of public member
        …
}
```

Class Definition- Java Typical Example:

```
public class  Rectangle
//By convention, a class name begins with an uppercase letter.
{
        //the data attributes or instance variables
        private double length;
        private double height;

        // member method to find area
        public double computeArea()
        {
                return length * height;
        }
}
```

2.7 Escape Sequence:

The following are few escape sequences that Java shares with other languages as C/C++:

Escape Sequence	**Meaning**
\n	*newline*
\t	*tab*

\’	*single quote*
\”	*double quote*
\\	*Backslash*

These are used mostly in output statements to format display. E.g.:

```
System.out.println(“\” Hello There\” ”);
```

Yields as output in quote: ”Hello There”

2.8 Data Types

Variables are named memory locations to store values. This means that when you create a variable you reserve some space in memory. Hence, there is a need to know the type of data so as to know what appropriate memory to give to it. Data type of a variable indicates the type of data it can store and also the type of operations that can be performed on it.

There are basically two types:

1) Primitive/Basic/Fundamental data types (built-in)
2) Derived/User-defined/Reference types

Spaces on memory are allocated based on the type e.g. char is 1 byte

Primitive Data Types

A primitive – type variable can store exactly one value of its declared type at a time. E.g. if an integer variable **x** initially stores whole number **5** and it is later assigned whole number **9**, then **x** no longer carries 5 but **9**.

The **primitive** also called **simple** or **scalar data** types are initialized by default.

The primitive Data types are into four categories:

Integral – These are integers such as byte, short, int, and long. They are signed, that is, can store positive and negative numbers.

Floating Point- These are decimal numbers, i.e. numbers having fractional part- fixed point or floating point precision. E.g. float, double etc.

Boolean – These have to do with data with logical values (true or false).

Characters- These are symbols which include alphabets and numbers. When characters are combined we form a string.

The following table describes the ***8 primitive data*** types built into Java language (i.e. those that come with your compiler):

Table 2.1: The ***8 Java Language primitive data types***

S/N	Type	Type Represented	Range of Value	Default Initialization	Usage
1	byte	Very small integers	-128 to 127	0	byte v = 2
2	short	Small integers	-32768 to 32767	0	short v = 2
3	int	Big integers	-2,147,483,648 to 2,147,483,647	0	int v = 2
4	long	Very big integers	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0	long v = 2
5	float	Real numbers	$+/-1.4*10^{-45}$ to $3.4 * 10^{38}$	0	float v= 2.5
6	double	Very big real	$+/-4.9*10^{-324}$ to $1.8 * 10^{308}$	0	double v =2.5
7	char	Characters	Unicode character set	‘ ’	char v = ‘A’
8	boolean	True or False	Not applicable	False	boolean v = false

2.9 Operator and Precedence in Integer Expressions

Table2.2: **Java Basic Arithmetic Operators**

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

Remainder operator(%)

Operator %(Percent), returns remainder after division. It is referred to as ***modulus operator.***
E.g.
29 % 9 = 2

6 % 8 = 6
10 % 2 = 0

Other behaviours:

- y = x++; means assign the value of x to y, then increment x by 1
- y = ++x; means increment x by 1, then assign this new value to y
- y + = x means y = y +x
- ++x or x++ means increase x by 1, ie. x=x +1
- - -x or x - - means decrease x by 1, ie. x=x -1 (there is no absolute difference when this is used in control statement such as for…loop.

Operator Precedence in Integer Expressions

The following precedence or priority holds as far as the operators are concerned:

Table 2.3: Operator Associativity

Precedence	**Associativity**
* / %	Left to right
+ -	Left to right

↑ High

Low

Operators of equal precedence have left to right associativity (as in the one to encounter first from left to right).

E.g   9 – 5 + 2 = (9 - 5) + 2 = 4 + 2 = 6

Table 2.4: Relational Operators

Operator	**Meaning**
<	Less than
<=	Less than or equal
>=	Greater than
>=	Greater than or equal
==	Equal to
!=	Not equal

Example:

5>7 yields True

Operator Precedence:

Table 2.5: Operators associativity and type

PRECEDENCE	ASSOCIATIVITY	OPERATOR TYPE
!	Right to left	unary
* / %	Left to Right	Multiplicative
+ - +	Left to Right	Additive, concatenation
<, <=, >, >=	Left to Right	Relational
==, !=	Left to Right	Equality
&&	Left to Right	and (associative)
||	Left to Right	or (associative)
=, +=, -=, *=, /= , %=	Right to left	Short cut operators

High

Low

2.10 Derived/Reference types

All non-primitive types are reference types, so ***classes***, which specify the types of objects, are reference types. A class for example is a concrete data type that specifies an object state(s) and behavior(s).
Some Examples of Reference types are:

Table 2.6: List of some reference types

Reference Type	**Description**
Array	Collection of objects/data/items of the same type
String	Combination of series of characters
Classes	Binding of variables and methods
Enums	Used to define collections of constants
Interfaces	Abstract type used to specify behaviours a class must implement

Note on String Type:

String is a reference type. By definition, it is a sequence of characters (value is usually within a double quote) which could be a combination of alphanumeric or special characters. E.g. "John the Baptist".
Java provides a **String** class that allows us to use and manipulate strings.

String** name; // **a string declaration

name = “makinde”; ***// assigning an initial value***

Alternatively, you declare a String object by calling its constructor as follows:

String name = new String(“makinde”);

Exercise:

Make an attempt to implement the following methods in String Class:

length, charAt, substring, Concat, toUpperCase, toLowerCase, compareTo , equals, equalsIgnoreCase, startsWith, endsWith, trim.

Strings and Concatenation Operator(+)

- If “A” and “B” are Strings, then “A+B” evaluates to another String. This is concatenation (joining together).
- E.g. “Jesus is Lord” + “Everyday” results into another string “Jesus is Lord Everyday”

Task 2.1:

1) A shop keeper sold 10 bags at the rate of 500 naira each with a discount of 5% on the total sales. How much did the buyer pay. Write a program to compute this.

2) Write a code to show the result of the following:
   i. *y = x ++; means assign the value of x to y, then increment x by 1*
   ii. *y = ++x; means increment x by 1, then assign this new value to y*
   iii. *y + = x means y = y +x*
3) Write a program to manipulate string reference type. The program should accomplish the following:
   a) Declare/define two strings
   b) Assign values to the strings
   c) Concatenate the two string using a plus operator and show the output
   d) Print the second character of the first string
   e) Convert the first string into all uppercase

   Hint: Use any appropriate functions to aid your task.

5) Read functions used to manipulate strings and write a short one on some of them.
6) Write a program to Convert Time in Seconds to Hours, Minutes and Seconds.
7) Write a program to read in two values and print out the bigger of the two.

Chapter 3
USING SCANNER OBJECTS AND GUI

This chapter introduces how to do basic input and output (I/O) operations, reading values from the console, writing the output using methods available in Scanner class and some GUI (Graphical User Interface). Also, we shall discuss and implement how to process and manipulate strings.

3.1 Using Scanner Object

To read in input values from the keyboard as permitted by other programming languages, Java employs the service of a class called **Scanner** to accomplish the task. There are other tools used to achieve this but Scanner class is appropriate for the moment.
Scanner (class) is part of/inside a particular Java package named "util". A package in itself is a collection of pre-compiled classes used to achieve one task or the other. To access a package, we use "import" keyword at the beginning of a program. This opens the program up (grants access) to the classes in the package. The Scanner class has many input methods which shall be discussed.

Introducing the **Scanner** class, the following is written at the beginning of the program(before your class definition):

```
import java.util.Scanner;
```

Notice that the word "**Scanner**" has '**S**' as uppercase.
Then there is a need to instantiate its object (via an object variable) inside the method where the Scanner class is to be used. Therefore, by typing the following inside the method,

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

We have created an object "readVal" of the Scanner class. The word new is used to create the object variable of type Scanner. System.in represent the keyboard i.e. it should get input from keyboard but not from file or modem.

Code Example; Using Scanner class

```
1)  import java.util.Scanner;
2)  class Reader {
3)  public static void main(String args[]) {
4)     int n;
5)      /* note that variable name readVal is formed using the
6)       usual acceptable variable naming style */
7)      //instantiate Scanner object (via an object variable readVal)
8)      Scanner  readVal = new Scanner(System.in);
9)
10)      System.out.println("Enter your integer value");
```

```
11)   n = readVal.nextInt();
12)   System.out.print("The integer value keyed in is:" + n);
13   } // end of main method
14}//end of class
```

Note on the code written:

- If line 1 is alternatively written as :

  import java.util.*;

- This makes all classes in the **(util)** package accessible to the program. Scanner is used in the above program instead of * because the program needs only the Scanner class where method such as ***nextInt*()** exists.
- Line 11 shows how the object variable **readVal** is used to invoke **nextInt()** method, and the value read in is assigned to variable "n". Note that a dot operator is used with the method.

  n = readVal.nextInt();
- nextInt() is a method in Scanner class to read in integer.

Other methods in Scanner class include:

- double x = readVal.nextDouble();// to read floating point
- String y= readVal.nextLine();// to read a string value
- char c = readVal.next().charAt(0);/* to read a single character. This is because a Scanner object cannot read data of type char. So we use next() which reads string but charAt(0) makes it pick the first character alone. Remember first character of a string is accessed at index 0.*/
- **Some others** are: nextShort(), nextLong(), nextFLoat(), nextBoolaean().

Task 3.1

1) Test the following Scanner class methods in a simple program. nextShort(), nextLong(), nextFLoat(), nextBoolaean().
2) Consider the following 3 string values: "Good", "Old", "Days"
   Using **next()** method in Scanner Class, write a program to capture the first character in each of the first 2 strings and the entire string in the third value, concatenate them and print out the result.

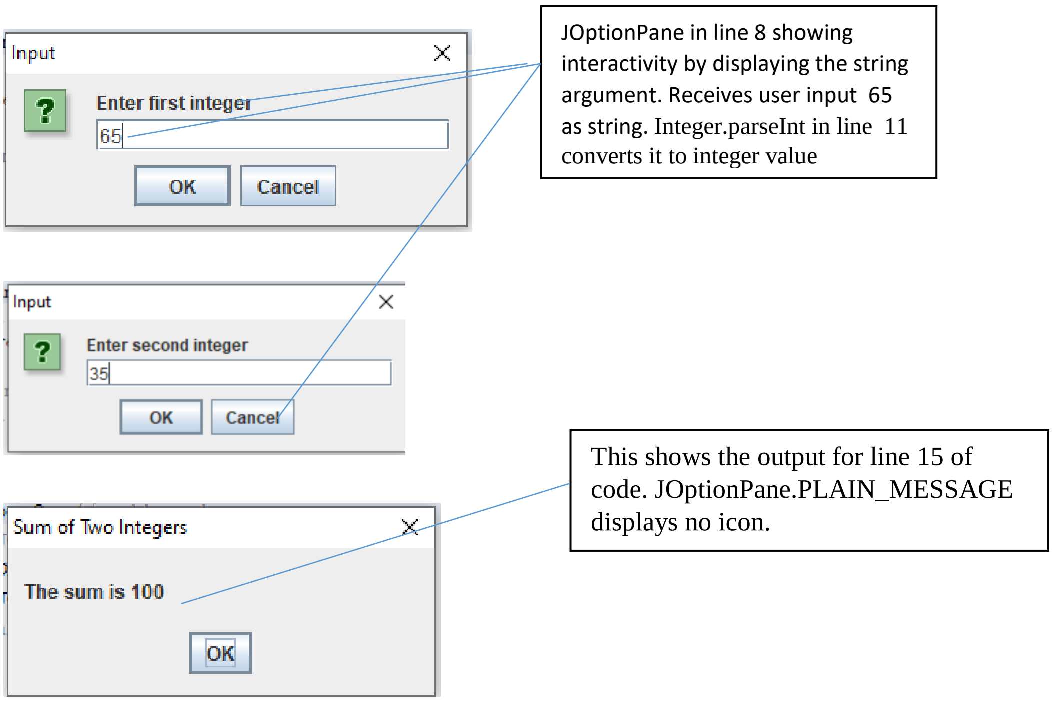
3.2 GUI-Based Input/Output Using JOptionPane:

Graphical User Interface (GUI) gives interactive programs a better look and feel or better visual effects. Most applications today use **windows** or **dialog boxes** to interact with the user. Dialog boxes are windows in which programs display important messages to the user or obtain information from the user. Java's **JOptionPane** class (package javax.swing) provides prebuilt dialog boxes for both input and output. These are displayed by invoking static **JOptionPane** methods.

The following shows an example of a program that uses two **input dialogs** to obtain integers from the user and a **message dialog** to display the sum of the integers.

Example 1: Program to add numbers

```
1)  // Ex 1: Addition program that uses JOptionPane for input and output.
2)  import javax.swing.JOptionPane; // program uses JOptionPane
3)  public class Addition
4)  {
5)  public static void main( String[] args )
6)  {
7)  // obtain user input from JOptionPane input dialogs
8)  String firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
9)  String secondNumber = JOptionPane.showInputDialog( "Enter second integer" );
10) // convert String inputs to int values for use in a calculation
11) int number1 = Integer.parseInt( firstNumber );
12) int number2 = Integer.parseInt( secondNumber );
13) int sum = number1 + number2; // add numbers
14) // display result in a JOptionPane message dialog
15) JOptionPane.showMessageDialog( null, "The sum is " + sum,
16) "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
17)       } // end method main
18) }// end class Addition
```



JOptionPane in line 8 showing interactivity by displaying the string argument. Receives user input 65 as string. Integer.parseInt in line 11 converts it to integer value

This shows the output for line 15 of code. JOptionPane.PLAIN_MESSAGE displays no icon.

Figure 3.1: Screen shots of GUI pop up interaction

Code Explanations:

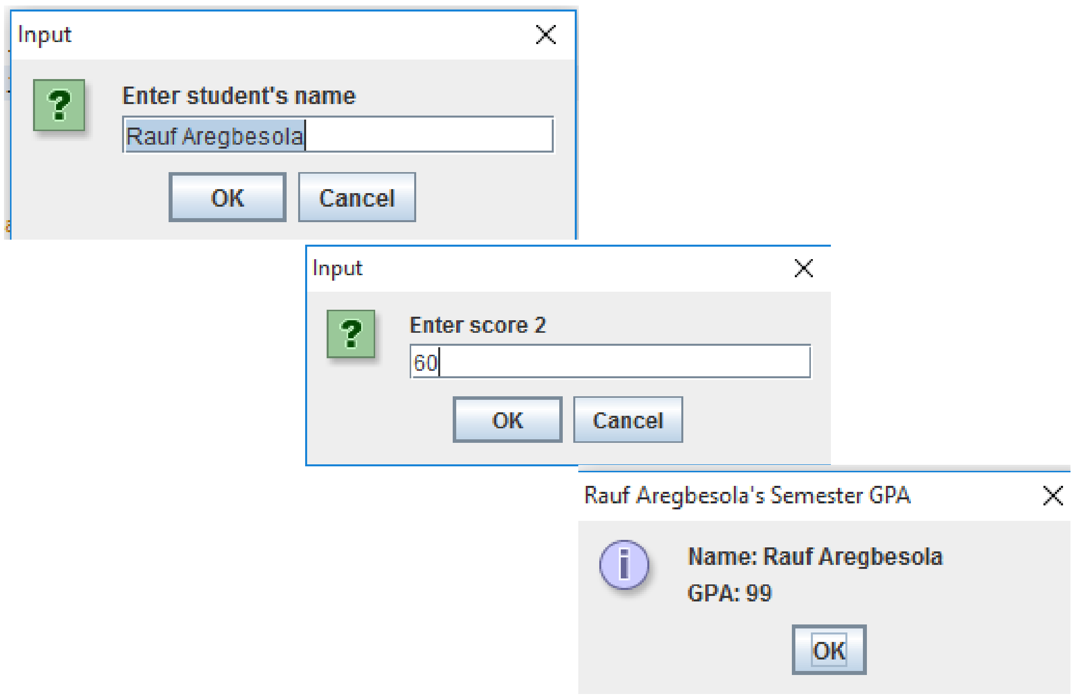
- Line 2 imports class JOptionPane.
- Line 8 assigns the result of the call to JOptionPane static method **showInputDialog** to string variable ***firstNumber***. This method displays an input dialog prompting the method's String argument "Enter First Integer". The input dialog prompt display capitalizes only the first letter of the first word i.e. performs sentence case on the output.
- An input dialog can input only Strings (this is common to most GUI components). If the user clicks **Cancel**, showInputDialog returns null.
- To perform the calculation, convert the Strings entered to ***int*** values using the Integer class's static method called **parseInt.**
- Lines 11-12 assign the converted values to local variables number1 and number2, and line 13 sums the values.
- Lines 15-16 use JOptionPane static method **showMessageDialog** to present the output.
- The first argument helps the Java application determine where to position the dialog box. If the first argument is null, the dialog box is displayed at the center of your screen.
- The second argument is the message to display—in this case, i.e. result of concatenating the String "The sum is " and the value of sum.
- The third argument—"Sum of Two Integers"—is the String that should appear in the *title bar* at the top of the dialog.
- The fourth argument—**JOptionPane.PLAIN_MESSAGE**—is the type of message dialog to display. A PLAIN_MESSAGE dialog does not display an icon to the left of the message.
- Class JOptionPane provides several overloaded versions of methods showInputDialog and showMessageDialog, as well as methods that display other dialog types. *The title bar of a window typically uses **book-title capitalization style.*** i.e. capitalizes the first letter of each significant word without appending any punctuation.

Example 2: Program to find the sum of 3 values

```
// Ex 2: Program to find the sum of three grades
import java.util.Scanner;
import javax.swing.JOptionPane;
public class GradeAverage {

   public static void main(String args[]) {
      Scanner in = new Scanner(System.in);
      int sum = 0;
      String name = JOptionPane.showInputDialog(null,
        "Enter student's name", "Rauf Aregbesola");
for (int i = 0; i < 3; i++) {
         int next = Integer.parseInt(JOptionPane.showInputDialog(null, "Enter score " + (i+1)));
         sum += next;
      }

     String res = String.format("Name: %s\nGPA: %d",name, sum);
```

```

    JOptionPane.showMessageDialog(null, res, name + "\'s Semester GPA", JOptionPane.INFORMATION_MESSAGE);
  }
}
```

GUI- Output:



Enter student's name
Rauf Aregbesola

Enter score 2
60

Rauf Aregbesola's Semester GPA
Name: Rauf Aregbesola
GPA: 99

Figure 3.2 Showing the screen shot of the interactive GUI output

JOptionPane Message Dialog Constants

The constants that represent the message dialog types are shown as follows. All message dialog types except **PLAIN_MESSAGE** display an icon to the left of the message. These icons provide a visual indication of the message’s importance to the user. A QUESTION_MESSAGE icon is the *default icon* for an input dialog box.

Table 3.1 Description of some dialog messages

Message Dialog	**Icon/Symbol**	**Meaning**
ERROR_MESSAGE	❌	Error
INFORMATION_MESSAGE	ℹ	To show its an information
WARNING_MESSAGE	⚠	Indicates a possibility of / or potential problem
QUESTION_MESSAGE	❓	Requires a user to take an action or a response
PLAIN_MESSAGE	No Icon/Symbol	There is a message but no icon

Task 3.2:
Consider a bank that offers four different types of account (‘A’, ‘B’, ‘C’ and ‘X’). The following table illustrates the annual rate of interest offered for each type of account.
Design and implement a program that allows the user to enter an amount of money and a type of bank account, before displaying the amount of money that can be earned in one year as interest on that money for the given type of bank account. Use any of the control statements you have learnt and make your program GUI based.

Account	Annual rate of interest (%)
A	1.5
B	2
C	1.5
X	5

Chapter 4
CONTROL STRUCTURES

This chapter re-emphasizes how to effectively use and apply program control flow constructs which basically include categories of Sequencing, Selection and Repetition.

4.1 Basic Control Structures:

There are 3 basic program flow constructs, namely:

1. Sequencing
2. Selection
3. Repetition

1. Sequencing:

By natural/default construct, a program proceeds execution from one statement to the next (orderly) till termination except interrupted/re-directed to flow in another direction.

```
E.g.    x =10;    //10 is assigned to x by right to left associativity
        x = x + 2;  //10 + 2 makes 12 and overrides x to hold 12
        x = x * 2;  //12 * 2 makes 24 and overrides x to hold 24
```

It executes in the above order, from first to last till value 24 is stored in x.

2. Selection:

This is a program control construct/structure used to alter the
natural/ sequencing construct. Based on the intention of the programmer, the following are some conditional statements:

a) if*(condition) statement;*

- ***If*** is the keyword
- ***(condition)*** is a boolean expression which results into true/false
- The ***statement*** that follows immediately is executed if the ***condition*** is *true (1)* else it is skipped without being executed, then the condition is terminated.

If Statement Example:

The **if** selection statement is a **single-branched selection statement** because it selects or ignores a single action (or a single group of actions –block of codes).

//Java implementation; code excerpt

```
Scanner input = new Scanner(System.in);
float score;
score = input.nextInt();
If(score > 79)
   {
System.out.println("Excellent Student");
   }
```

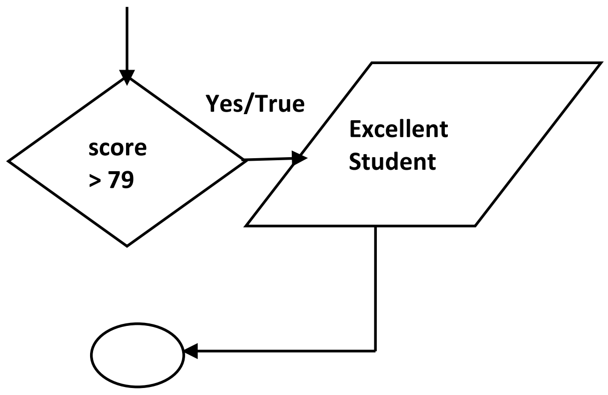


Figure 4.1: Structure of if statement

Exercise:

Considering the following lines of code snippet, what is the output of the code execution?

```
int a = 2;
int b = 3;
if(a == b) System.out.println("This cannot be seen");
….
 ….
```

Justify your claim.

b) If else *statement*

The if…else statement is called a **double-branched selection statement** because it selects between two different actions (or groups of actions). This has provision for what to do next in case the set condition is false instead of naturally skipping and moving to the next available statement at the end of the if statement.

It takes the form:

```
if(condition) statement; else statement;
```

e.g.

```
if(score >=50)
{  //open/close curly bracket here is optional if the line of executable code is only one
System.out.println("Pass");
}
else
{
System.out.println("Fail");
}
System.out.println("Bye"); //this line is not part of the if condition but rather sequence
```

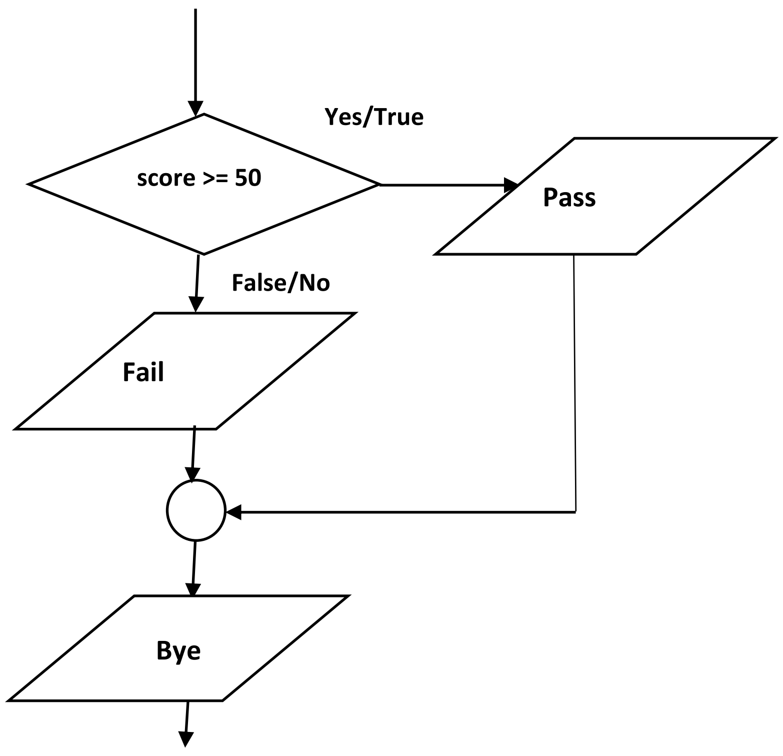


Figure 4.2: Pictorial view of ***if else*** construct.

Conditional Operator (?:) working as double-decision/selection statement:

Java provides the conditional operator (?:), which works like the if…else statement. This conditional operator is called ***ternary*** operator—one that takes three operands. The first operand is a conditional expression, the second operand the result you get if the conditional expression is true and the third operand is the value for a false result from the expression. Consider the following example:

```
//Displays the word “Pass” if grade is greater than 50 else “Fail”.
System.out.println( grade >= 50 ? "Pass" : "Fail" );
```

c) If else if else statements: compound if

An example is a code fragment to determine the rank of a student grade based on its score:

1) if (score > 79) Grade = “Excellent”;
2) else if(score > 59) Grade = “Good”;
3) else if (score > 49) Grade = “Pass”;
4) else Grade = “ Repeat”;
5) System.out.println(“Bye”); //this line is not part of the if condition but rather sequence

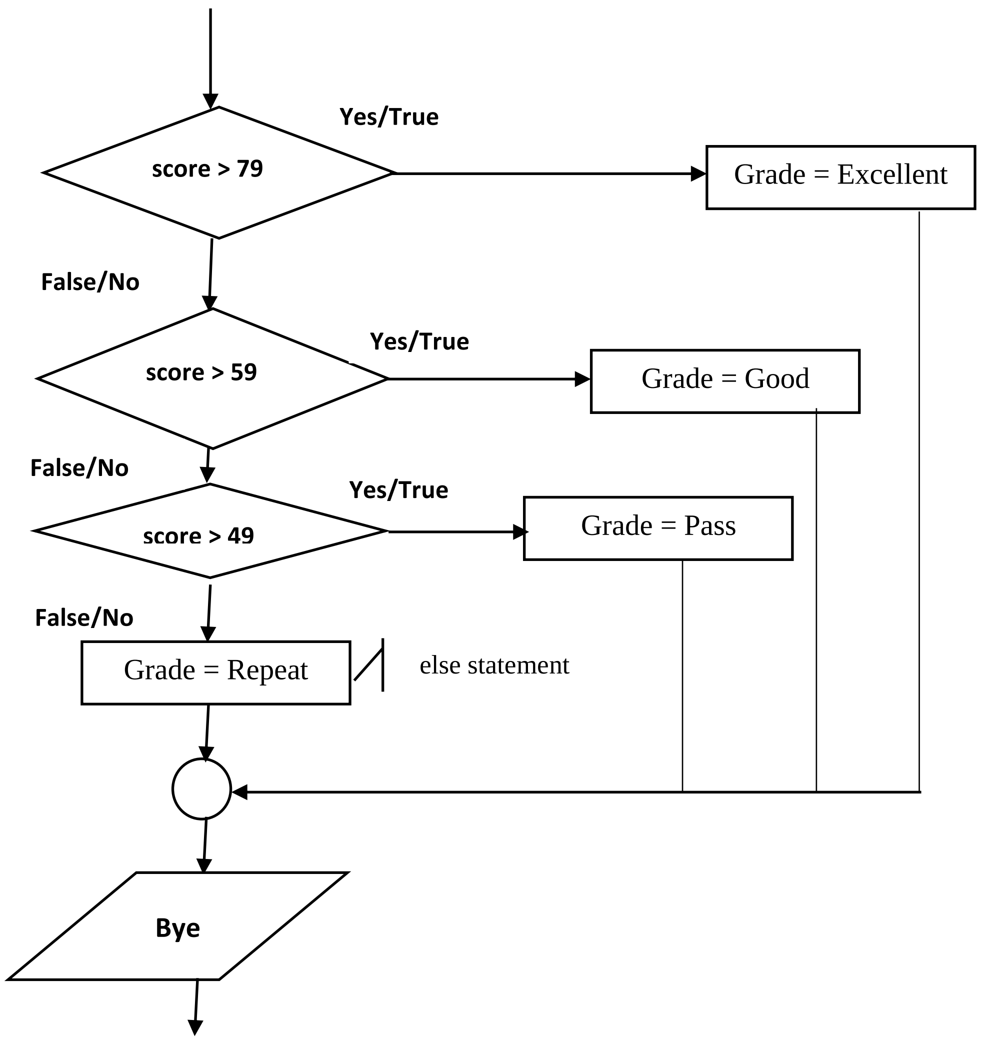


Figure 4.3: Pictorial view of ***compound if*** construct.

d) Nested IF:

NESTED IF statement is a multiple selection structure, i.e. when there is an IF … ELSE statement inside an IF statement or an IF THEN statement.

Nested if Example:

If Age < 16 years deny scholarship
else if sex = male grant scholarship in African Countries else grant scholarship in African and American Countries

//code snippet

```
1) int age;
2) // assume number takes a value here
3) if(age >= 16)  //outer if
4) {
5)     if (sex==“male”)  //inner if shown with code indentation
6)     {
```

```
            System.out.println("Grant Scholarship in African Countries");
7)        }
8)       else {System.out.println("Grant scholarship in American Countries:");}
9)  }
10) else {System.out.println("Grant No Scholarship");} //this else is for outer if
```

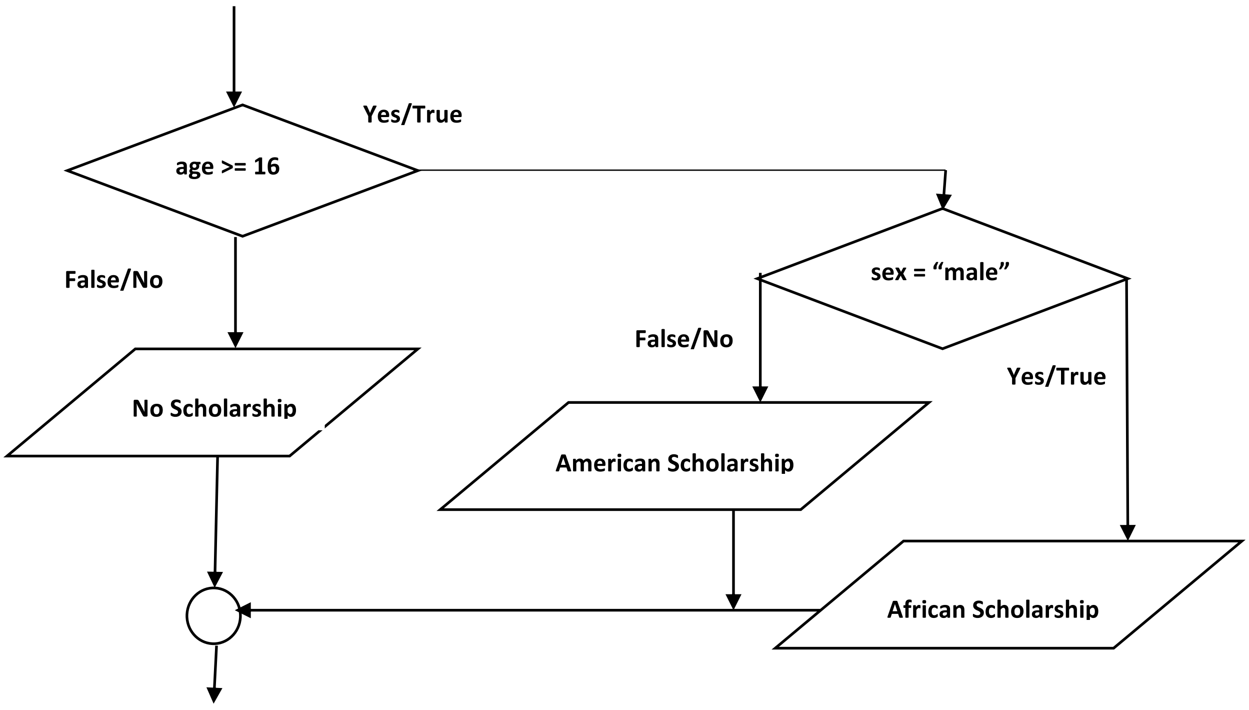


Figure 4.3: Pictorial view of ***Nested if*** construct.

e) Switch Statement

The switch selection statement is called a multiple-selection statement because it selects among many different actions (or groups of actions). It performs different actions based on the possible values of a ***constant integral expression*** of type byte, short, int or char:

The basic form of the switch:

```
switch(variable)
{
case value1:
        //statements 1 or method call
        break;
case value2:
        //statements 2 or method call
        break;
        :::::::::// all case statements before default
default:  /*focuses on the need to process exceptional conditions;
```

```
        statement to take care of what is not in earlier options*/
    }           // end of switch statement
```

//Switch Statement code example

```
Scanner input = new Scanner(System.in);
int group;
group = input.nextInt(); // read in group type from keyboard
switch(group){
        case 1: System.out.println("This is SS1 class");
          break;  // exit switch
        case 2: System.out.println("This is SS2 class");
          break; //what if the break statement is omitted here?
        case 3: System.out.println("This is SS3 class");
          break;
        default: System.out.println("This is not a group");
              break;//Optional - when placed as the last statement
} // end of switch statement
```

The statement is executed based on the particular input passed into variable **group**.
The **break** statement makes the compiler not to execute the subsequent statement which is what convention demands of program flow construct referred to as **sequencing** (i.e. execute the next code after the immediate). But what if the **break** statement is omitted anywhere for example in case 2? If the value in variable **group** is 2, the program jumps into case 2 according to the switch statement construct, then executes the next statement and so on until a break statement is encountered. For such error, our **output** from the above code chunk is as follows:

```
This is SS2 class
This is SS3 class
```

Figure 4.4: Typical output for omitting break statement

3) Repetition (Loop or Iteration Constructs)

This is another way to alter the natural sequencing process of program execution. When the program is expected to execute a particular instruction or a group of instructions in a number of times repeatedly. These are also referred to as loop/iterative/repetitive constructs.
The major variations include:

a) **for** loop [Pre-test loop- test first before loop]
b) **while** loop [Pre-test loop]
c) **do…while** loop [Post-test loop – loop first at least once then test]

a) for loop : general format

for*(initialization; condition; increment_step) statement;*

- ***for*** - This *is the keyword*

- ***Initialization*** *- sets a loop control variable to an initial value and it is logically executed once before the first iteration of the loop.*
- ***Condition*** *-The loop termination test, is executed before each iteration of the loop begins. The number of times the termination test is executed is one more than the number of times the loop body is executed (e.g. if number of loop is* ***n****, termination test goes in* ***n+1*** *times). This forces the termination to happen.*
- ***Increment_step*** *- the loop counter, "increment step", is executed once per loop iteration. It determines how the loop control variable is changed each time the loop iterates*

for loop Implementation Example:

```
for(int loop_count = 1; loop_count < 5; ++loop_count)
{
            System.out.println("loop_count: " + loop_count);
}
```

This loop iterates four times and gives the following output:

```
loop_count: 1
loop_count: 2
loop_count: 3
loop_count: 4 (because the loop is to terminate before the 5th iteration)
```

Nested For loops:
loops may be nested within another loop.
Example:

```
for( int i = 1;  i<= 4; i ++) //each of i runs all of j
 {
         for(int j = 21; j <=  23; j++ )
        {                      //the inner curly braces are important for multiple lines of code
              System.out.println( i +  " " + j);
        }//end of inner for loop
         System.out.println();
 }//end of outer for loop
```

b) While loop constructs:
The **while**(condition){…} is commonly referred to as the while - do construct. This is called a ***pretest*** loop. That is, test is carried out before the loop action takes place. It means that if the test condition is not true, then loop will not happen. That means, it is a loop useful if you want an iteration to happen at zero or more times. E.g. in input validation coding:

//Illustrative Code example

```
double mark;
mark = 10;
```

```
while(mark > 0)
{
  System.out.println("Printing as long as mark has not reached zero");
  mark- -;  // decrement the value of mark by one after each loop
}
```

c) do…while structure

This is called the ***post-test*** loop. That is, the test is carried out after the loop action. This means that the instruction in the loop will happen at least **once** before the test condition takes place. It then means the loop action happens at one or more times (1 or n times).

E.g. consider the following code fragment:

```
int studentCount = 1; //set initial value to 1
do {
      //line to read in score goes here;
      if (score>49) System.out.println("Pass");
      ++studentCount; // increment count
      }while(studentCount <=10);
```

Here, there is at least one student in a class for the course to exist.

Using for loop with break statement:

The **break** statement, when executed in a while, for, do…while or switch, causes immediate exit from that statement.

A good example is a program that allows you to make three guesses. Consider the following code chunk:

```
final int SECRET  =  10;
 boolean guess = false; // initialized to false(0)
for (int i = 1; i <= 3; ++i)
     {
        System.out.println("Enter  guess :" + i);
          int val = input.nextInt();
          if(val ==SECRET)
          {
       guess = true;
       break; // exits the for loop
          }
  }
  if (guess) {System.out.println("Guess is correct");}
  else {System.out.println("Guess is wrong ");}
```

Using Continue Statement:

A **break** statement forces a loop to terminate while a **continue** statement forces a loop to skip the remaining instructions in the body of the loop and continue to the next iteration. E.g.

```
for (int I = 10; i>= 1; i--)
{
        if (i%2 != 0)
            {
              continue; //if test is true, skip iteration and move to the next round of iteration
            }
  //executes when iteration is not skipped i.e. if test is false
         System.out.println( i + ”is even”);
}// end for statement
```

The following does the same task as the preceding one:

```
for (int i = 10; i<= 1; i- -)
{
        if (i%2 == 0)
             {
               System.out.println( i + ”is even”);
             }
}// end for statement
```

8. Note : the lines of code above is to display even integer values between 10 and 1.

Task 4.1:

1) What is the minimum number of times a while(condition){…} loop can execute
2) The second conditional statement in the for() loop is to be executed in how many times if n is the number of times for the loop
3) Write your own java function to test for prime number.
4) The expression ((x > y) && (a < b))is true if either x > y is true or a < b is true(T/F)
5) Maintan the following code by fixing the errors:

```
i=1;
while ( i <=10 ); ++i; }
```

6) Using iteration, find the factorial of n if the factorial is equal to the product of the positive integers from 1 to n.
7) Write a program that computes the sum of a list of integer numbers that is supplied by a user. The end of data is signaled by the value -9999 which is used only as a flag and is not included in the sum.
8) Write a code to test for palindrome. A palindrome is a word, sentence, verse, or even number that reads the same backward or forward e.g. madam, civic, level etc.
9) Imagine the various activities at an ATM (Automated Teller Machine) that can aid successful transaction between your Bank and you. Write a menu driven program to do

the following: (i) checking your balance (ii) making withdrawal and (iii) making deposit. (Note: amount to withdraw cannot be more than the balance.)

Chapter 5

PREDEFINED AND USER-DEFINED FUNCTIONS

This chapter presents how to create and apply user-defined methods and how to use predefined methods.

5.1 Methods

A ***method*** is like a function or subroutine or program module which is a set of instructions bundled together, like in a box, designed to perform a single well-defined task. This is a function or procedure associated with a class (or object type) invoked in a message-passing style. This is also referred to as the interpretation of the message (i.e. the method used to respond to the message or request).
As discussed earlier, a method has a name, a return type (void if it does not return anything), parameter list and method body (block or lines of codes).

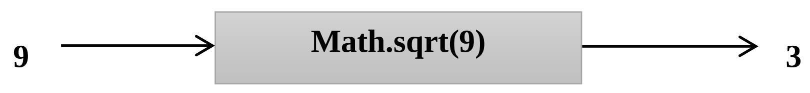
```
// Implementation Example
double addTax(double price, double taxPercent)
{
    return price + price * taxPercent
}
```

The following exemplified how ***addTax*** method is called:

double result = addTax(costPrice, tax); //

5.2 Illustrating a Method:

A method is very much like a black box that as you throw in input(s), it performs some computations and gives you output based on the purpose with which it is designed.
E.g. A method to find the **square root (named sqrt)** of an integer number behaves as follows:



The value passed is the argument while processed output is the returned value. Methods may also act without receiving any value or returning a value.
The behaviours of an object are represented as methods in a class.
Java comes with many predefined methods. You only need to know how to use them (e.g. knowing its name, return type and parameters) but not how they were written. That's the purpose of re-use.

5.3 A one Method(main()) simple Program:

The following program is to ***calculate*** the ***perimeter*** of a ***rectangle*** using the values of length and height.

//the following does one single task written as a plain code in function main

```
package javaapplication;
public class Rectangle {
   public static void main(String[] args)
   {
        int length  = 5;
        int height  = 6;

        int perimeter = 2 * (length + height);
        System.out.println("The area = " + perimeter);
    } // end of main method
} // end of Rectangle class
```

5.4 Creating/Making other Methods (User-defined):

If a program is to do many tasks, putting the various tasks into a single function such as **main()** might be nightmares. Therefore, a solution that is a combination of many tasks should be split into independent tasks tagged as methods. The above program could be seen in this light by breaking it into a main module/method and one other module/method to do the perimeter calculation. From this point, the program could be expanded easily to handle many more tasks added as modules or methods.

```
// Slitting into functions each handling independent tasks
package javaapplication;
public class Rectangle {
//main method setting values to variable, passing parameters and calling other method
    public static void main(String[] args)
    {
        int length  = 5;
        int height  = 6;

       perimeterMtd(length, height); // call by main() to perimeterMtd
} // end of main method
// Method definition for computing Perimeter
 static void perimeterMtd(int l, int h)
 {
        int perimeter = 2 * (l + h);
        System.out.println("The Perimeter = " + perimeter);
  }
} // end of Rectangle class
```

5.5 Method Basics

- A ***method*** is a complete section (block) of Java code with a definite start point and an end point and possibly its own set of variables.
- Methods should be designed so that they have one primary task to accomplish.
- Methods can be passed data values and they can return data.
- A method's set of variables are only know by the method and are called local variables – i.e. they are local to that method.
- Creating methods improves reusing code.

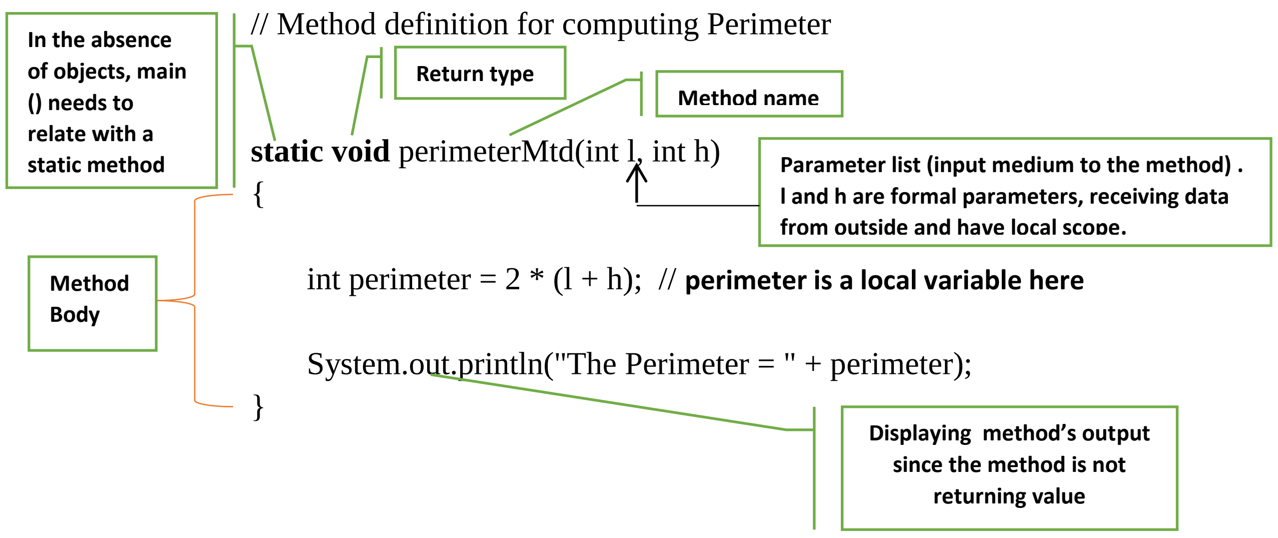
- **Basic Things To Know:**

i. How to write a method within a program
ii. How to call a method within a program or from another program
iii. How to send information/data to a method and how to get information back.

A Method General format:

1. ***method_return_type*** methodName([optional parameter list separated by comma])
2. { //method begins
3. …set of codes to perform the method task…//function body
4. }//method ends

A simple Example:



```
// Method definition for computing Perimeter

static void perimeterMtd(int l, int h)
{

        int perimeter = 2 * (l + h);  // perimeter is a local variable here

        System.out.println("The Perimeter = " + perimeter);
}
```

In the absence of objects, main () needs to relate with a static method
Return type
Method name
Parameter list (input medium to the method) . l and h are formal parameters, receiving data from outside and have local scope.
Method Body
Displaying method's output since the method is not returning value

5.6 Using Created Methods

To use a method, you need to invoke or call it.

A method is called using its **name** and if there are **parameters**, you have to supply values to the parameter variables in the **order** and **type**(datatype) with which they appear in the function definition.

Sending values is called **parameter passing**, which could be either **pass-by-value** or **pass-by-reference.** In Java, **variables** are **passed by value** value but **objects** by are **passed by reference**.

Therefore, in the previous program, the call perimeterMtd(length, height); by **main()** makes **length *and* height** the ***actual arguments*** (holding values) sent to the method, while **l** and **h** in: **void** perimeterMtd(int l, int h) are the **formal parameters** representing **length** and **height** respectively.

Important Note:

- No code should be written after the word ***return*** else it becomes unreachable.

- A void function needs no return statement, the function simply terminates after the last statement.
- In **pass-by-value**, a function does not change the original value of variable sent to it as a parameter because only a copy of the variable content is passed/ just a value.
- In **pass-by-reference**, the formal parameters are bound to the reference values of the actual parameters. A change in the value of a parameter inside the function changes the value contained in the variable to which that parameter refers. Objects variables and array are passed by reference in Java.

5.7 Scope of a variable:

This is all about where a variable is declared/defined - it is said to be local (visible) to that method. A variable declared within a particular method cannot be used elsewhere- they are not recognized there. Note, **parameters** also have local scope.

Variables are visible within the pair of curly brackets in which they have been declared. That is, if you refer to them elsewhere, you get a compile error. Consider the following:

```
1.  public static void main(String[] args)
2.  {
3.              int x = 1;    // x is only known in main method
4.              myMethod(x);
5.  }
6.  Static void myMethod(int   xformal)  //xformal is locally scoped to myMethod
7.  {
8.              int y;  /* variable y is local or known to myMethod and its existence starts
                        from line 8 */
9.              y = 20 *xformal;
10.             System.out.println("value = " + y);
11. } // y stops existing from line 11 (from the closing curly "}" of this method)
```

NOTE:
variable **y** cannot be used inside ***main*** and **x** cannot be used inside ***myMethod*** either.

- If **y** is referred to in the **main()** method, it would be out of scope. And also, if x is referred to in **myMethod()**, it would be out of scope. The visibility (scope) of y is limited to myMethod() in the program.
- Therefore, since a method does not know what is declared in another method and does not have anything to do with it, it means you could declare a variable with the same name inside different methods. The compiler sees them as different entities each within its scope in its own method. That is, a variable x declared and used in the main() method, is independent of x declared and used another method within the same program.
- Also, a **scanner** method declared in main() used to read in values may not cover the second method in case you need to read in values as well in the second method e.g. myMethod().
- Therefore, you declare Scanner input = new Scanner(System.in); in the other method.

5.8 Benefits of Methods

1) *Reusability*
2) *Information hiding*
3) *Reducing complexity*

5.9 Summarized rules for creating a Method:

- *The method name must follow standard Java naming conventions.*
- *There may be one return type.*
- *If there is no return type, the void data type is used.*
- *The input argument list must have data type and names (separated by commas if more than one, it must correspond when calling i.e. in type and in order)*
- *You may pass in as many arguments as you like*
- *If your list is empty (no arguments are passed), the parentheses will be empty ()*

5.10 Pre-Defined Methods

These are methods that come with the language compiler with predefined purposes available to programmers to solve common problems e.g. mathematical computations without the need to reinvent the wheel.
In java pre-defined methods are organised as a collection of classes, called class libraries (Packages) or Application Program Interface (API). e.g the class **Math** contains a collection of static methods that helps in performing common mathematical calculations. Math belongs to java.lang package. You can call any static method by specifying the name of the ***class*** in which the method is defined, followed by a dot(.) separator and the method ***name***. e.g. ClassName**.**MethodName([argument]). This class is implicitly imported by the compiler, and therefore, does not require any formal importation.
Therefore writing **Math.sqrt(81);** helps to find the square root of the value in the argument. Similarly, **Math.PI** is a call to one of the class variables. Math.PI is defined in class ***Math*** with modifiers public, final and static.

Table 5.1: Some Math Class methods – In package java.lang

S/N	Methods Expression	Return type	Description and Example
1	abs(double x) abs (int a)	double int	absolute value of double e.g Math.abs(-2.3)= 2.3 absolute value of integer e.g Math.abs(-7) = 7
2	ceil(x, double x)	double	Returns the smallest whole no > = x e.g Math.ceil(72.18) returns 73.0
3	exp (double x)	double	Returns e^x where e = 2.7182818284590455 Math.exp(3) returns the value 20.085536923187668
4	floor(double x)	double	Returns the largest whole number (as a double) e.g Math.floor(50.50) returns 50.0
5	pow(double x, double y)	double	Returns a value of x raised to the power of y e.g Math.pow(2.0, 3.0) = 8.0

5.11 Using Parse Methods

A Numeric String is a string consisting of only an integer or a floating point number which may be preceded by a minus sign. E.g.:

"2021" , "-128", "22.56", "-22.56".

Java has methods to convert numeric strings into their equivalent numeric form. Classes where these are implemented are referred to as wrapper classes. E.g.:

(1) Integer.parseInt(strExpression) converts a string consisting of an integer value to its actual integer equivalent:

Integer.parseInt("2021") = 2021

(2) Float.parseFloat(strExpression) converts a string consisting of a floating point number to its actual decimal equivalent:

Float.parseFloat("22.56") = 22.56

5.12 Method Overloading

Overloading is a feature in Object-Oriented Paradigm supported by Java. It is when a method is exhibiting many behaviours/forms the same method name but different set of parameter lists (i.e. parameter types or parameter quantity are different).

When an overloaded method is called, the compiler selects the proper method by examining the number, types and order of the arguments in the call.

E.g.:

int *findMax(int a, int b);*
int *findMax(int a, int b, int c);*
int *findMax(int a, int b, int c, int d);*

```
        double findMax(double a, double b);

//implementation example of overloading
int findMax(int a, int b)
{
   if (a>b) return a;
          else return b;
}

int findMax(int a, int b, int c)
{
    int biggest =  a;
    if (b> biggest) biggest = b;
    if (c > biggest) biggest = c;
   return biggest;
}
```

Methods sharing the same name but different parameter types or list are overloaded

Task 5.1:

1. Create a function to compute X^n such that the values of x and n are passed as parameters to the function. [Hint]: The main function should send input arguments and output the result from the function.
2. Create a function to implement the expression:

$$\sum_{i=1}^{n} i$$

n is expected to be the input value passed to the function.

3. Write a program having five methods, such that one of the functions is the **main** where others are called. Your **main** is to receive an input value **n** from the user and send to other functions where necessary. Let the program implement the following:

   a) Create a **detectPrime** function to test if the input received is prime or not
   b) Create a **detectOdd** function to test is the input is odd number
   c) Create a **factorial** function to compute the factorial value of the input
   d) Create a **summation** function to compute 1 + 2 + 3 + … + n.

Note: Use a **switch statement** to select which function to call at a point in time.

Chapter 6
OBJECT ORIENTED PROGRAMMING: CLASSES AND OBJECTS

This chapter is contains a proper introduction to object oriented programming (OOP) concepts. OOP is not about learning a new syntax but a new way of thinking as well as bringing solutions to software problems. Students will learn the key features of object oriented programming and its idiosyncrasies. For a start, in this chapter, you will learn how to create classes, instantiate and use objects.

6.1 Introduction to Object oriented Programming Techniques:

The field of systems analysis and design now incorporates object-oriented concepts and techniques, by which a system is viewed as a collection of self-contained objects that include both data and processes while traditional systems analysis and design methodologies are either data-centric or process-centric.

"Object think" is a more realistic way to think about the real world. Users typically do not think in terms of data or process; instead, they see their business as a collection of logical units that contain both—so communicating in terms of objects improves the interaction between the user and the analyst or developer.

Object oriented analysis and design is more than just learning new syntax or new language. It's a new way of thinking. In procedural concepts, we think in terms of data structures and algorithms but in Object Oriented Concept we think in terms of objects and their relationships. Object-orientation is introduced as a new programming concept which helps in developing high quality software at a much easier and faster pace.

Table 6.1: The Procedural Approach Vs Object Oriented Approach

The Procedural Approach	The Object-Oriented Approach
• System is organized around procedures. • Procedures send data to each other. • Procedures and data are clearly separated • Focus on data structures, algorithms and sequencing of steps. • Procedures are often hard to reuse (not so flexible for reuse). • Lack of expressive and powerful visual modeling techniques. • Transformation of concepts between analysis and implementation • This programming paradigm is	• System is organized around objects. • Objects send messages (procedure calls) to each other. • Related data and behavior are tied together in objects. • Modeling of the domain as objects so that the implementation naturally reflects the problem at hand. • Visual models are expressive and relatively easy to comprehend. • Focus on responsibilities and interfaces before implementation. • Powerful concepts: interfaces, abstraction, encapsulation, inheritance, delegation and polymorphism.

essentially an abstraction of machine/assembly language • Design models are far from implementation.	• Visual models of the problem evolve into models of the solution. • Design models are only a small step from implementation.

6.2 Object Concept:

The need to produce quick, economical and correct software solutions led to building essentially reusable software components called **objects**. Almost everything around us could be modeled as object. And these components could be easily linked together to form a functional unit or system. Objects undergo proper design and development process that enhance usability and reuse. This is simply what object-orientation stands for.
There are date objects, time objects, automobile objects, people objects, etc. Barely any noun can be reasonably represented as a software object in terms of ***attributes*** (e.g. name, size etc.) and ***behaviours*** (e.g. calculating, moving, etc.).
Object-oriented design and implementation approach is more productive than the earlier techniques such as structured / procedure-oriented techniques.

The Automobile Object

You can drive a car and increase its speed by pressing its accelerator pedal. The **blueprint** of a car is the design that describes the car. The accelerator pedal in the blueprint hides from the driver the complex mechanisms that actually increase the speed as the pedal is pressed. Hence the hiding of these detail enable people with little knowledge of how the mechanisms work to still drive easily as long as they know how to. Generally, objects can be used without knowing how they are made.

6.3 Basic Characteristics of Object-Oriented Systems

Object-oriented systems focus on capturing the structure and behavior of information systems in little modules that encompass both data and processes. These little modules are known as objects.

Classes and Objects

A class is the general template we use to define and create specific instances, or objects. For example, all of the objects that capture information about patients could fall into a class called Patient, because there are attributes (e.g., names, addresses, and birth dates) and methods (e.g., insert new instances, maintain information, and delete entries) that all patients share. ***An object is an instantiation of a class.*** In other words, an object is a person, place, event, or thing about which we want to capture information.
The state of an object is defined by the value of its attributes and its relationships with other objects at a particular point in time.

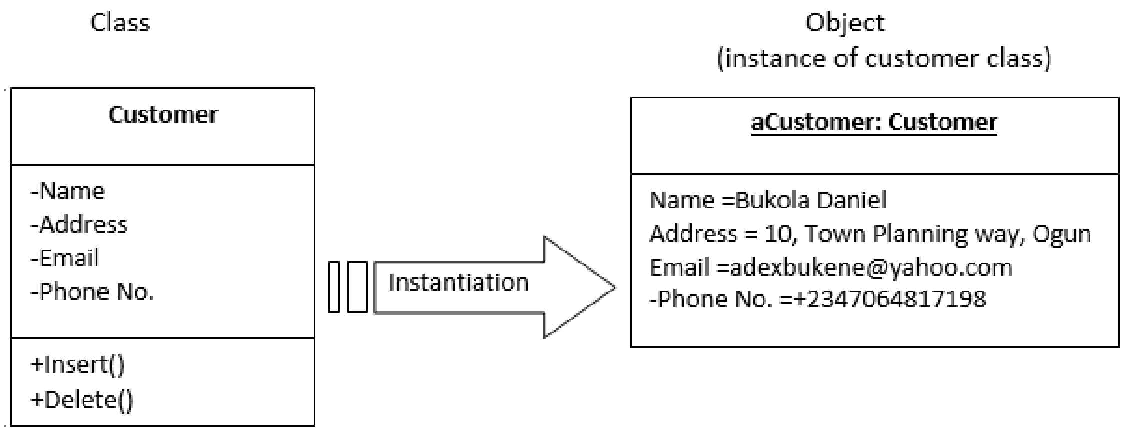

Class
Object
(instance of customer class)
Customer
-Name
-Address
-Email
-Phone No.
+Insert()
+Delete()
Instantiation
aCustomer: Customer
Name =Bukola Daniel
Address = 10, Town Planning way, Ogun
Email =adexbukene@yahoo.com
-Phone No. =+2347064817198

Figure 6.1: Class and Object Relationship

Members of a Class (Components)

A **class** is basically made up of two members namely: member data/attributes/variables and or member methods.

- (a) Attributes / Data Members:

A car has attributes such as colour, number of doors, current speed etc. the car attributes are part of the design in the engineering drawings. Every car maintains its own attributes but not that of others. So also an object has its attributes as it is used in a program. Attributes are specified by the class's data members. E.g. *account balance* as an attribute particular to each bank account object.

- (b) Member methods:

Using the car example to introduce some key OOP concepts, a member method performs a unique task in a program. It is what contains the statement(s) / codes that actually perform the task. It hides these statements from its user, just the same way the accelerator pedal hides the speed mechanism from the driver. A class is a program unit created to house the set of member methods that carries out the class's possibly many tasks. E.g. a bank class having ***withdraw*** as a method/operation.

Object Instantiation

To make a program perform the functions defined by its ***class's member methods,*** you must create an object of the class in your own program. This is called instantiation. An object is referred to as an instance of a class.

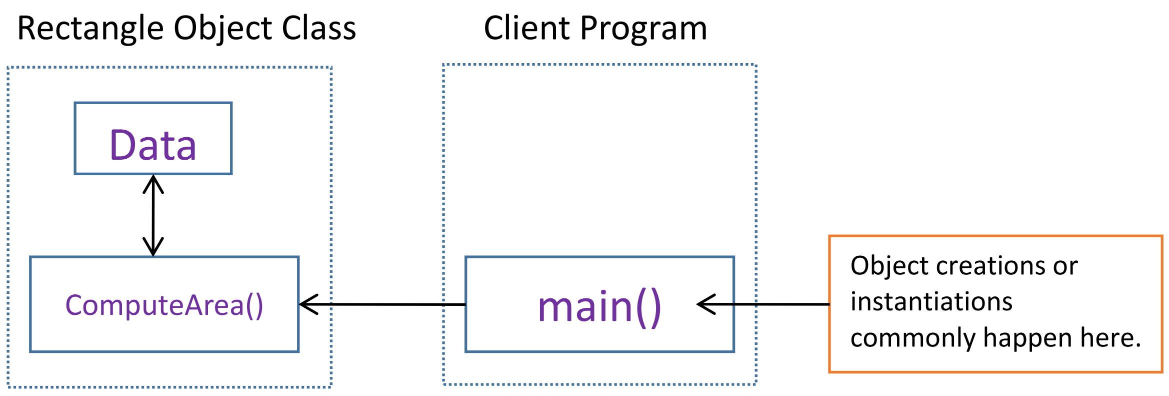


Figure 6.2: Programs interacting to do object instantiation

Methods and Messages

Methods implement an object’s behavior. Methods are very much like a function or procedure in a traditional programming language such as C. Messages are information sent to objects to trigger methods. A message is essentially a function or procedure call from one object to another object. Example is an insert message to the customer object to create a new customer.

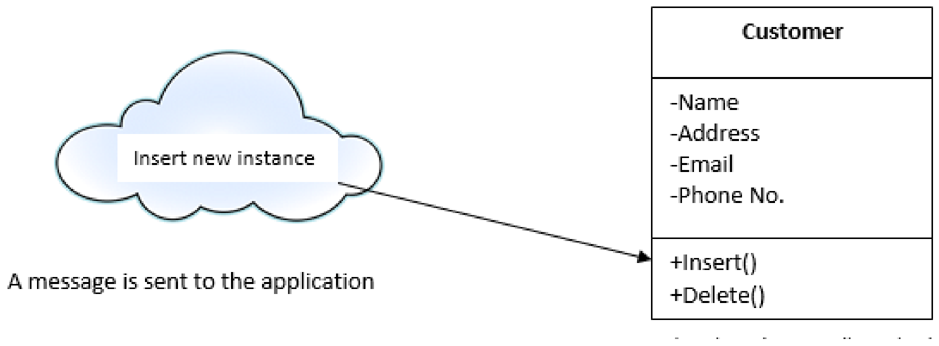


Figure 6.3: Message sending technique

6.4 Major Distinguishing Features of OOP:

There are three major features in object-oriented programming that makes them different from non-OOP languages namely:

(a) encapsulation,

(b) inheritance and

(c) polymorphism.

Encapsulation and Information Hiding:
Encapsulation in Object-oriented methodology is simply the combining of process and data into a single entity. Information hiding was first promoted in structured systems development. *The principle of information hiding suggests that only the information required to use a software module be published to the user of the module (abstraction).*

In object-oriented systems, combining encapsulation with the information hiding principle suggests that the information hiding principle be applied to objects instead of merely applying it to functions or processes. As such, objects are treated like black boxes. Ability to use an object by calling methods is key to reusability.

Classes encapsulate or bind **attributes** and **member methods** into objects. Objects communicate but do not have each other's implementation details(that are hidden within each object itself).

The principles of keeping variables/data private and restricting access to public methods is the basic idea of encapsulation. Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called "classes,"

Polymorphism
Polymorphism means that the same message can be interpreted differently by different classes of objects. For example, if we sent the message "Draw yourself " to a square object, a circle object, and a triangle object, the results would be very different, even though the message is the same. Polymorphism is either static or dynamic binding. **Static is demonstrated in method overloading. Dynamic or late *binding is a technique that delays identifying the type of object until run-time -demonstrated in method overriding.***
In a traditional programming language, instead of sending the message "Draw yourself " to the different types of graphical objects, you would have to write decision logic by using a case statement or a set of "if " statements to determine what kind of graphical object you wanted to draw, and you would have to name each draw function differently(e.g. drawSquare, drawCircle)

Inheritance
Inheritance entails making a new class object from an existing class either by customizing or adding new unique characteristics. Therefore, stabilized components are easily reused. E.g. an object of class "convertible" can be made from general "automobile".
Common sets of attributes and methods can be organized into superclasses. Typically, classes are arranged in a hierarchy whereby the superclasses, or general classes, are at the top, and the *subclasses*, or specific classes, are at the bottom.
Subclasses inherit the attributes and methods from the superclass above them. Instead of repeating the attributes and methods in the employee and customer classes separately, the attributes and methods that are common to both are placed in the person class and inherited by those classes below it.

Some classes do not produce instances, because they are used merely as templates for others that are more specific classes (especially those classes located high up in a hierarchy). They are called abstract classes.

Inheritance Class Hierarchy

In the following, **Person** is a superclass to the classes **Employee** and **Customer**.

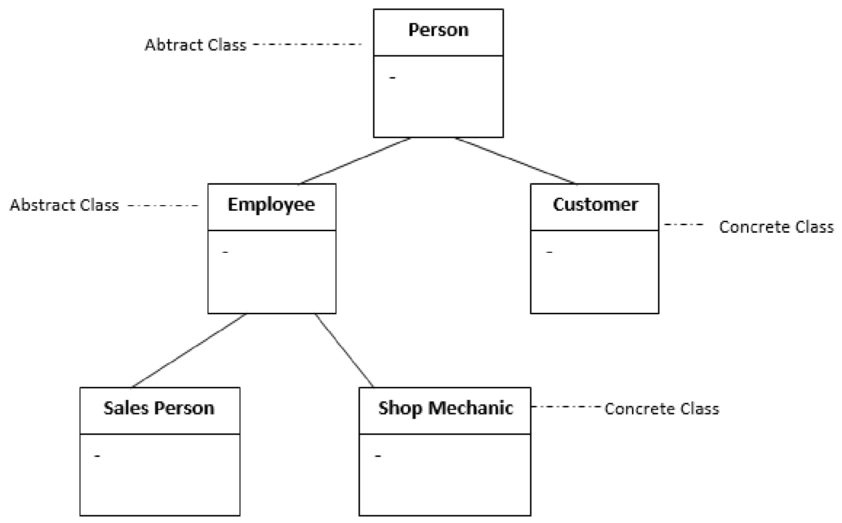


Figure 6.4: Showing **Person** as a superclass to E**mployee** and **Customer** classes.

Designs with and without Inheritance – Spotting the difference

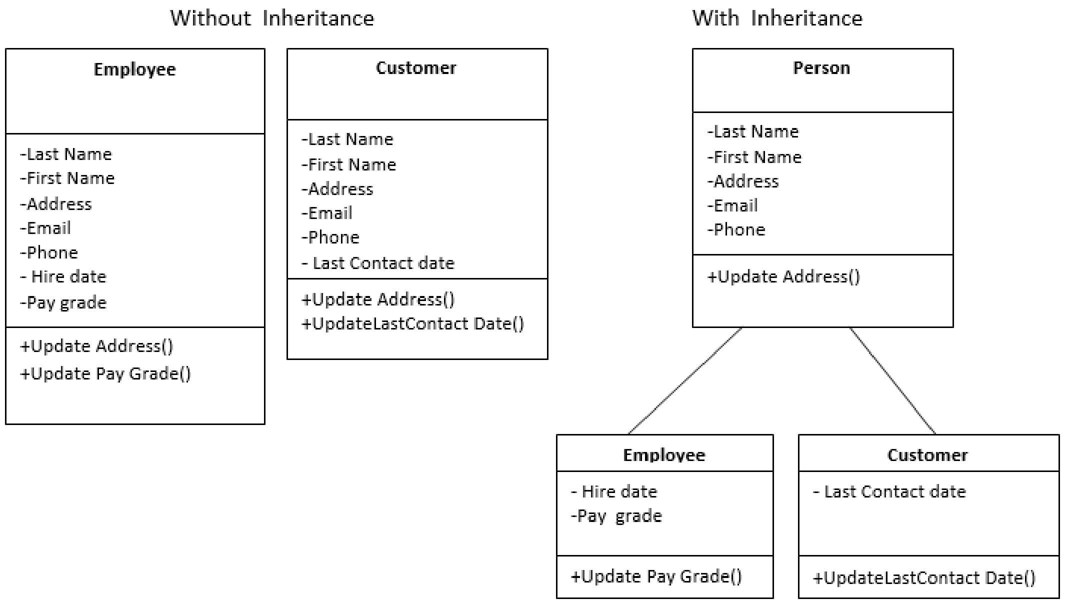


Figure 6.5: Code repetition difference in no inheritance and with inheritance.

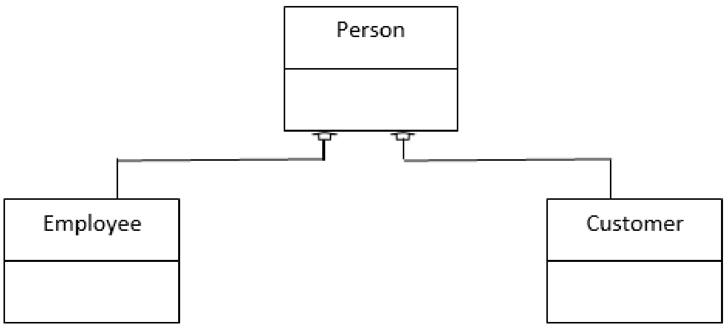


Figure 6.6: A summarized view of inheritance:

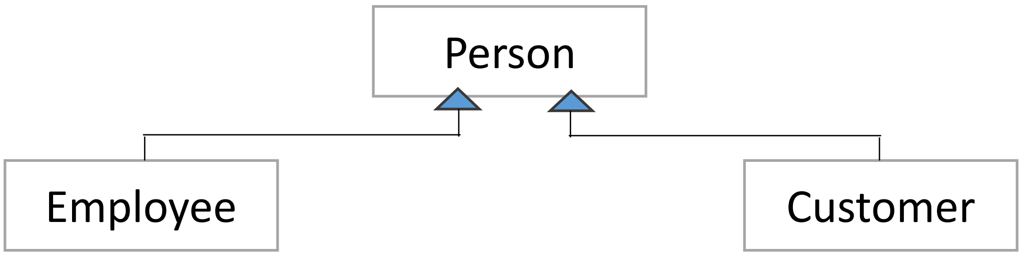


Figure 6.7: A more summarized view of class relationship showing only the name compartment

6.5 Generalisation and Specialisation: Term Usage

Generalization is the term that we use to denote abstraction of common properties into a base class in UML. The UML diagram's Generalization association is also known as Inheritance. When we implement Generalization in a programming language, it is often called Inheritance instead. Going specific into subclasses is therefore called specialization.

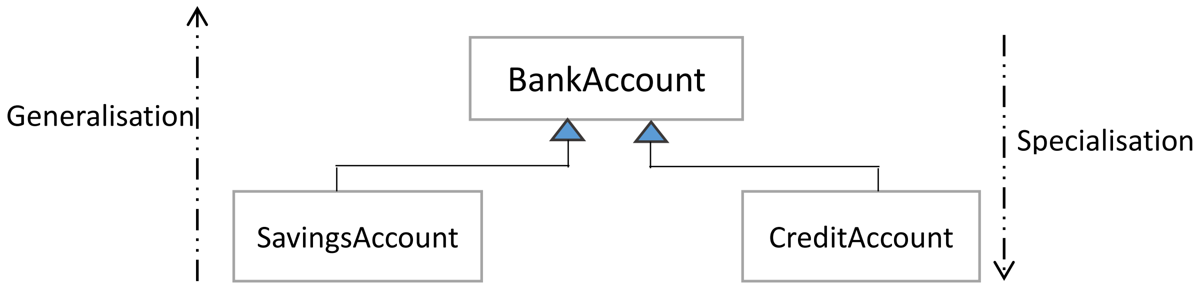


Figure 6.8: Illustration of Generalisation and Specialisation Terms

6.6 Benefits of Object-Oriented Systems Analysis and Design (OOAD)

Inheritance, encapsulation etc allow analysts to break a complex system into smaller, more manageable components, to work on the components individually, and to more easily piece the components back together to form a system.

Also easier to communicate to users to provide requirements and confirm how well the system meets the requirements.

6.7 Classes and objects (Deeper Explanation)

Objects and classes are the basic building blocks of OO systems.

What is an object? "A discreet entity with a well-defined boundary that encapsulates state and behavior; an instance of a class." Objects hide data behind a layer of functions/methods. Hiding the data part of an object behind this layer of functions is known as encapsulation or data-hiding.

What is a class? A class defines the common set of features (attributes, and operations or methods) that are shared by the objects of the class. A class defines an object.

An object has as properties:

a) Identity (unique existence in time and space),
b) State (determined by the attribute value of an object at a point in time) and
c) Behaviour (what object can do e.g. a printer object can ***printDocument***())

6.8 A class as a Data type: User-Defined Type

An Abstract Data Type (ADT) is the mathematical model (composition or makeup) of a data type/object. This composition means the set of values involved and the set of operations that can manipulate them.

A Data type is a computer representation of an ADT (the implementation of the mathematical model specified by an ADT). Eg. int x = 2;

Programming Languages usually have built-in data types that have their own ADT(mathematical implementation). e.g. INTEGER ADT in C language declared as "**int**" defines the set of numbers given by the union of set (-1, -2, -3, - - - , -∞) and the set of whole numbers (0, 1, 2, - - - , + ∞). The INTEGER ADT also specifies the operations that can be performed on integer numbers (e.g. addition, subtraction, multiplication and division). When a design or a program requires data types that are not in the programming language, we must construct the necessary data types by using built-in data types. When constructed, they are called **user-defined data types**.

The implementation of an ADT involves a translation of the ADT's specifications into the syntax of a particular programming language.

Data types like char, int etc. are basic/primitive data types that hold a single piece of information. If there is a need to use a single variable to represent quantity or item such as a book, a student, simple data type might not be adequate. Why? A book might have a title, an author, an ISBN, a price etc, where a Student might also have a name, matriculation number, programme of study, scores for various subjects and so on. An array will also not be appropriate because the attributes in each of these data items or quantities might not be the same as required by an array structure. . E.g student age might be ***int*** while student sex might be ***char*** or ***string***. Earlier procedural programming language such as **C** allows us to create a data type that can hold more than one piece/type of information – it is called **structure** implemented using a keyword named "**struct".** A single **structure** might contain integer elements, floating-point elements or character elements or more complex types (e.g. array or other structures). The individual structure elements are referred to as ***members.***

6.9 A typical structure declaration (struct):

Defining a Structure: Structure is defined in terms of its individual members as follows:

```
struct tag {
                member 1;
                member 2;
                …
                member n;
         };
```

Figure 6.9: using **struct** as derived type in C language.

Such that: **struct** is a required keyword; ***tag*** is a name that identifies structures **of** this type (having this composition); ***member 1 , member 2,*** . . . , ***member n*** are individual member declarations (which could be variables of different data types/structures).

The following structure example is named **account** (i.e., the tag is **account).** It contains four members: an integer quantity **(acct_no),** a single character **(acct_type),** an 80-element character array **(name[80]),** and a floating-point quantity **(balance).**

```
struct account {
                int acct_no;
                char acct_type;
                char name[80];
                float balance;
           };
```

Figure 6.10: **struct** example implementing account record structure

We can now declare the structure variables **oldcustomer** and **newcustomer as** follows:

struct account oldcustomer, newcustomer;

Therefore**, oldcustomer** and **newcustomer** are variables of type **account.**
It is possible to combine the declaration of the structure composition with that of the structure variables, as follows:

```
struct tag {
            member 1;
            member 2;
            …
            member n;

        }variable1, variable2,…, variablen;
```

tag (e.g. account) is optional here

Figure 6.11: Making many variables of type account.

```
struct account {
            int acct_no;
            char acct_type;
            char name[80];
            float balance;
        }oldcustomer, newcustomer;
```

OR

```
struct {
            int acct_no;
            char acct_type;
            char name[80];
            float balance;
        } oldcustomer, newcustomer;
```

Figure 6.12: Multiple variable declarations of type account

Processing a structure

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by:

variable.member

Variable is the name of a structure-type variable, and ***member*** refers to the name of a member within the structure. The dot (.) operator is inserted between variable and member. The dot (.) is a member of the highest precedence group, and its associativity is left to right.

To access customer account number in this example, we write: ***oldcustomer.acct_no***;
Also customer's name can be accessed by: **old*customer.name***

Important Note: ***In procedure oriented programming, data is separated from process. Therefore, to work with or use the member in the struct type example, functions would have to be written independently of the data (e.g. balance) and then programmed to work together as opposed to object oriented concept where data are process (i.e. the functions) are bound together in a single self-contained unit called class.***

6.10 Data Centric and Process Centric nature of C:

The problem with procedural languages such as Pascal or C include:

a) data does not have an owner and there is difficulty in maintaining data integrity. For example, many functions are using shared data.
b) there is insufficient support for abstraction

Consider the following examples:

```
struct eagle {
                    int weight;
                    float age;
               }
 fly();
 hunt();
```

```
struct horse {
                    int color;   //struct combines data
                    float age;   //of different types
                  } //end of struct definition

gallop();  // functions to use the data are separated
canter();
```

Figure 6.13: Data Centric or Process Centric nature of procedural programming

In POP (procedure oriented programming), data are handled separately from operations/code/process. Functions are global and therefore, the compiler has no means to know that horses cannot fly.

6.11 Defining a Class: Creating User-defined type

In the following example, the compiler knows that the two methods (fly and hunt) bound with the data belong explicitly to **eagle** ADT (Abstract Data Type).

In OOP, codes are bound with data as a single entity name **class**. Here, there is better abstraction by focusing on essential characteristics of an object.

```
class eagle {
                    int weight;
                    float age;
                    fly();
                    hunt();
               }
```

```
class horse {
              int color;   // member data
              float age;   //another member data
              gallop();  //member method now in the same
              canter();  //container (class) with data
 } //data and operations are bound together in a box
called class
```

Figure 6.14: Data and operations/processes are bound together (encapsulation)

Class = state (data members defined e.g. weight) + behavior (methods defined e.g. hunt())

OOP languages extended the **structure type** capability by not only allowing us to create types that store different pieces of data, but also to define within this type the ***methods*** by which we could process these data. E.g. a book type might have a method that computes and adds **tax** to the sale price; a student type might have a method to calculate an average score.

Furthermore, a Book in a library would be many but we would not need to define a book many time but once in a class and then generate many objects as we want from this blueprint, each representing an individual book.

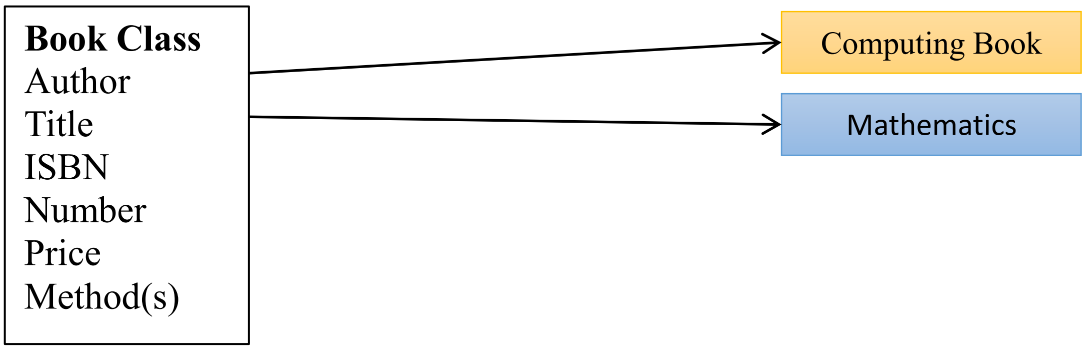


Figure 6.15: Book Class example instantiating different book objects

6.12 Encapsulation and OOP Law:

a) The first law of OOP is that data must be hidden(make it private)
b) let read access be done through read functions
c) let write access be done through write functions
d) for every piece of data, there are four possibilities:
    i. read and write permission
    ii. read only
    iii. write only
    iv. no access

6.13 Anatomy of a class:

The following reveals the structure of a class examining its parts. These parts shall be discussed and implemented in details in the subsequent sections.

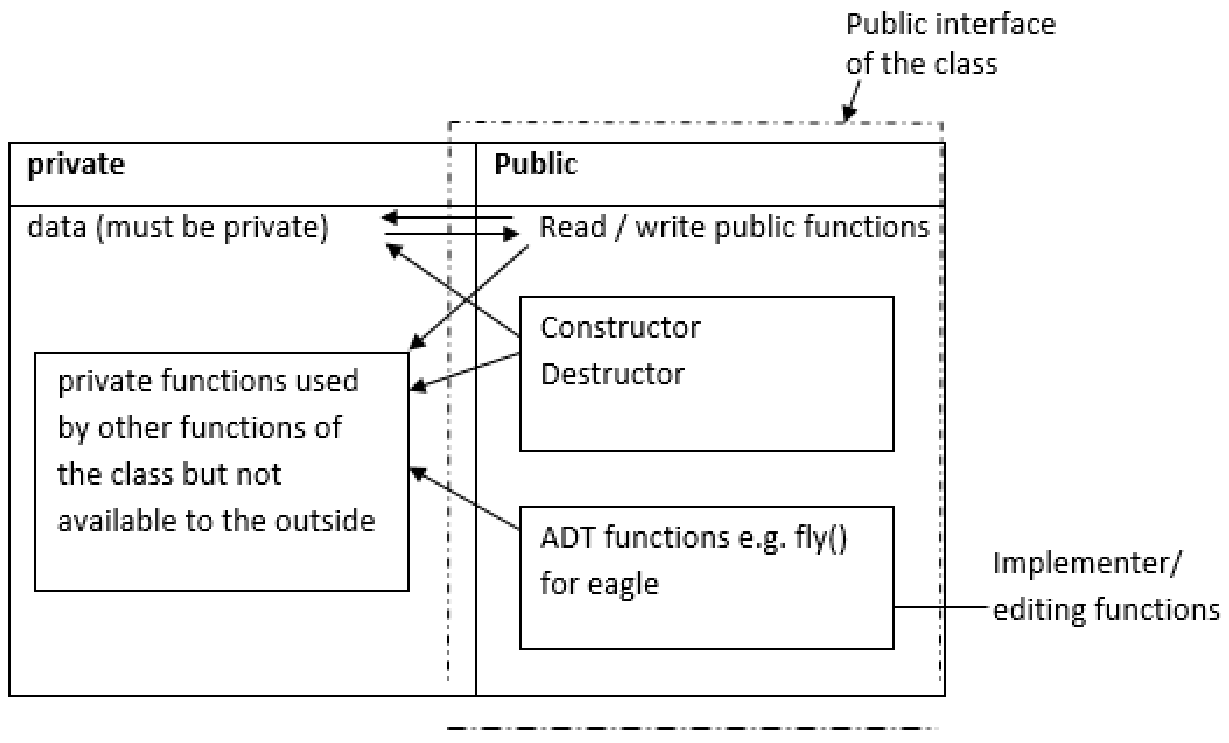

Public interface of the class
private
Public
data (must be private)
Read / write public functions
Constructor Destructor
private functions used by other functions of the class but not available to the outside
ADT functions e.g. fly() for eagle
Implementer/ editing functions

Figure 6.16: The internal structure of a class and its parts.

6.14 UML class notation

A graphical language for communicating the results of any object oriented analysis and design, OOAD process is the Unified Modeling Language, UML. Object Management Group (OMG) defines a set of 14 diagramming techniques for modeling a system, one of which is the UML class diagram. Class diagram is used for modeling the static structure of a system during analysis and design. It illustrates the relationships between classes modeled in the system.

Each class is drawn by using three part-rectangles with the class's name at the top, attributes in the middle, and methods (also called operations/ behaviours) at the bottom.

The only mandatory part of the visual UML syntax is the name compartment with the class name in it(i.e. the name compartment is enough to show relationship between various classes).

Class Diagrams:

Attribute visibility relates to the level of information hiding to be enforced for the attribute. The visibility of an attribute can either be public (+), protected (#), or private (-).

A ***public*** attribute is one that is not hidden from any other object.

A ***protected*** attribute is hidden from all other classes except its immediate subclasses.

A ***private*** attribute is one that is hidden from all other classes.

The default visibility for an attribute is usually private.

Operations are actions or functions that a class can perform. ***The default visibility for an operation is usually public.***

There are three kinds of operations that a class can contain: constructor, query, and update.

6.15 Summary of Major Access Specifiers:

Table 6.2 Access specifiers and their definitions

Specifier	**Description**
Private	Private members (variables or methods) in a class can be seen, used, assigned or changed only by other members of the same class. Private members are neither accessible by the object nor inherited by the derived class.
Protected	Protected members in a class can be seen, used, assigned or changed by other members of the class. They can be inherited by the derived/subclass. Protected members are also not accessible by the object.
Public	Public members may be seen, used, assigned, or changed by all class members and also accessible by objects. Any part of the program that has access to the class object can access public members (variables or methods) via the object.

6.16 UML class Diagram

The following are examples of a detailed view:

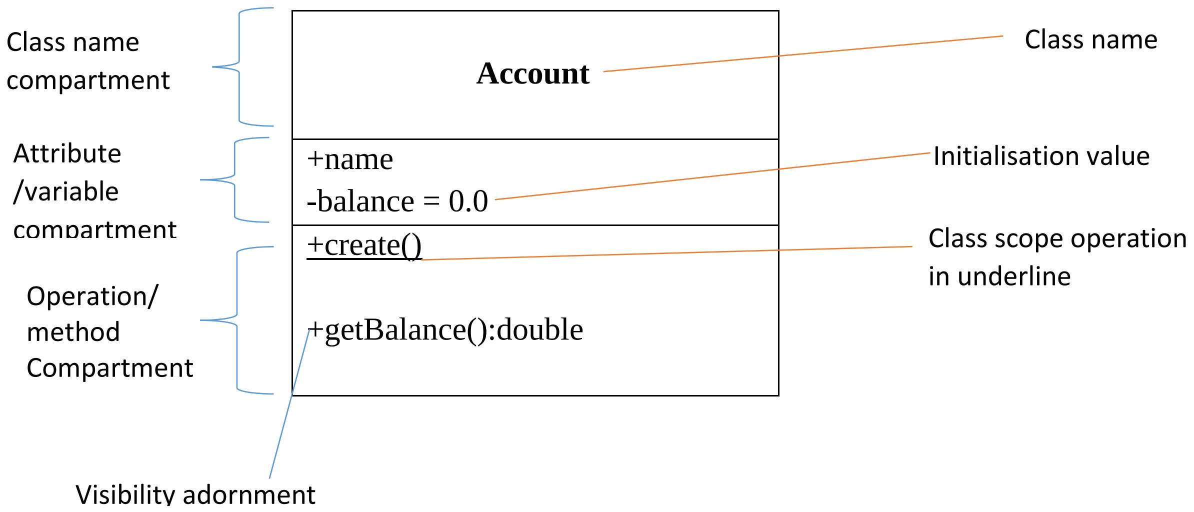
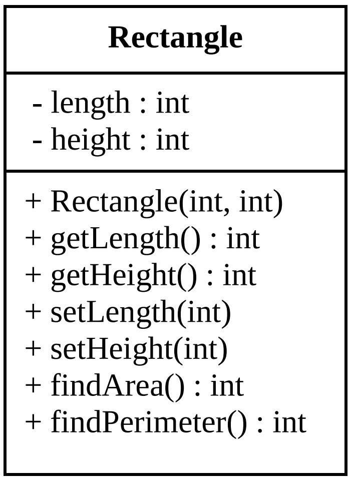


Figure 6.17: Class components illustration (The underlined denotes a class method or static variable or method)

Class Diagram: Rectangle Class Example



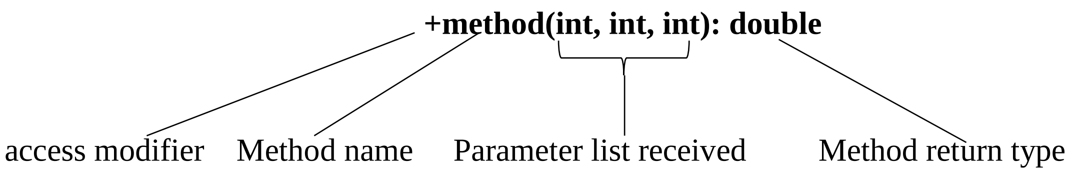


Figure 6.18: Illustration of method declaration in the UML class diagram;
In Java, constructor method answers the same name as the class.

6.17 Implementing Rectangle Class

We will show a progression of different versions of Rectangle class implementations (Rectangle1, Rectangle2, etc.). The first will use implicit constructor then gradually others will use user-defined/parameterized constructor. An implicit default constructor is used by Java program when the programmer does not specify any constructor.

Rectangle1 is the class template where **area method** is implemented. To explain in bits, the data **attributes** are directly assigned values in the class. ***Rectangle1Use*** having main() method is a separate class where the Rectangle1 class object is instantiated and used. The keyword ***new*** is to create memory for object x. Rectangle1() is the default constructor method used to create object x.

```
public class Rectangle1 {
   //instance variables
    public int length = 6;  //value or state initialized or set
    public int height = 5;

public int findArea()  // method
       {
          int area = length * height;
          return area;
       }
}
```

Figure 6.19: Rectangle1 class implementation showing code and variables

```
public class Rectangle1Use {

   public static void main(String[] args)  //method
      {
         Rectangle1  x = new Rectangle1();// instantiation
         System.out.println("The length = " +x.length);
             System.out.println("The height = " +x.height);
         System.out.println("The Area = " + x.findArea());
      }
}

```

Figure 6.20: Rectangle1Use is another program where Rectangle1 object is instantiated

Rectangle1 x = new Rectangle1(); This line means Rectangle1 object named **x** has been instantiated. Therefore, object x can access all public variables and public methods defined in its class(Rectangle1) from its present scope(main method of ***Rectangle1Use*** class). This is why we can write: x.length and x.findArea().

But remember we earlier said that "***The principles of keeping variables/data private and restricting access to public methods is the basic idea of encapsulation***." This implies that variables (instance variables) ***length*** and ***height*** are better defined ***private*** for **security** and **data integrity.** This promotes the benefit of encapsulation. If the variables remain public, any program can inadvertently access them and thereby defeat the purpose of encapsulation. Therefore, good software engineering practice demands that you make these data/variables private and then create public methods to access them instead of leaving them to possible unintentional access and modification. This is done by creating ***setter(mutator***) and ***getter(accessor)*** methods.

Rectangle2 Class: Introducing Setter and Getter Methods

```
public class Rectangle2 {

/*attributes or instance variables declared private – meaning
restrict the accessibility of the attributes to methods of this
class only. Instance Variables are NOT declared inside any
method*/
    private int length;
    private int height;

  //this method is used to read the length attribute -getter
  public int getLength()
  {
    return length;
  }

  //this method is used to read the height attribute - getter
  public  int getHeight()
  {
     return  height;
  }
//this method allows us to write the length attribute
  public void setLength( double l)
  {
       length =  l;
  }
//this method allows writing to the height attribute
  public void setHeight( double h)
  {
       height =  h;
  }
  // this method returns the area of the rectangle
  public int findArea()
  {
       return length * height;
  }
```

Figure 6.21: Rectangle2 class implementing set and get methods

Rectangle2Use Class: Using Rectangle2 class object:

Direct access to length from external program like (Rectangle2Use) is not possible any longer since attribute length is now hidden as private in Rectangle2 class:

```
System.out.println("The length = " +x.length);       // is no longer possible
```

```
public class Rectangle2Use {

  public static void main(String[] args) {
       Rectangle2  x = new Rectangle2();// instantiation

                //the mutator methods gives value to the variables
                x.setHeight(3);
                x.setLength(4);
       //accessing the variable values through "." operator
       System.out.println("The value of length = " +x.getLength());
       System.out.println("The values of height = " +x.getHeight());
       //Displaying the result of area computation
       System.out.println("The Area = " +x.findArea());

  }
}
```

Figure 6.22: Using Rectangle2 class object set and get methods

6.18 Introducing Constructor Methods

When the new Rectangle object is created using the keyword "new", some space is reserved in memory. The following line of code in the earlier example (Rectangle1Use class):

Rectangle1 x = new Rectangle1(); // instantiation using default Constructor

Rectangle1() in the instantiation represents the implicit default constructor provided automatically by the Java program. This means that one can refer to the implicit default constructor even when the programmer does not specify any constructor. The default constructor takes no parameters and does only the task of reserving memory for the new object. Then once you have defined your own constructors, this default constructor is no longer automatically available. But if you want it to be available then we have to re-define it explicitly like this as an example:

```
public Rectangle1()
{
}
```

Restricting our method definitions to essential functions that defines the class (avoiding I/O statements e.g. using the ***System.out.println*** statement as experimented in **findArea()** method in Rectangle2 class) will make your classes universal and reusable elsewhere with little or no modifications.

In Java, any method(s) that bears the same name with its class are the constructor(s).

What are user-defined constructor methods?

A user-defined constructor method is one that, in addition to reserving memory for the existence of a new object, also takes parameter(s) and sets initial value(s) for the new object. i.e. sets the new object's state(s). This means Constructor parameters are used to initialize attribute values at the point of object construction.

E.g.

```
public Rectangle3( double l, double  h)
{
        length = l;
        height = h;
}
```

We will write the **Rectangle3 Class** containing **Rectangle3()** constructor method that will receive values from an external code via its parameters **l** and **h** during object instantiation. The method Rectangle() assigns values (or sets initial values) to instance variables **length** and **height** via its **l** and **h** parameters respectively.

So if you are using a user-defined constructor, each time you create a Rectangle object, you may have to specify what will go into the **length** and **height** attributes simultaneously. The following is an example usage:

// creating memory for object x and initializing instance variables

1) ***Rectangle3   x;***
2) ***x = new Rectangle3(8, 5);***

or the two steps above re-written as one statement as follows:

1) ***Rectangle3   x = new Rectangle3(8, 5);***

In standard implementation, a constructor does not have a return type and does not even use the word "**void**" otherwise the compiler would think it is a regular method.

In point of fact, it is very logical to find a means of giving values to the **length** and **height** of the rectangle. Defining a constructor method will help do that initially. Then adding methods such as **setLength** and **setHeight** will allow us to change these values during the course of a program as required by the user. In case we need to fetch and use these values, methods named **getLength** and **getHeight** can be created and used.

So far, all of these are presented in the following table before we show a complete implementation of Rectangle3 class and how it is used.

Recangle3 class Table: Explaining all class members

Table 6.3: Method members of Rectangle3 class example

Method	Description	Inputs	Output
Rectangle3	The Constructor	Two data items, type int, representing the length and height of rectangle respectively	Not Applicable
setLength	Set or assigns the length of the rectangle	A data item, type int	None
setHeight	Set or assigns the height of the rectangle	A data item, type int	None
getLength	Returns the length of the rectangle	None	A data item, type int
getHeight	Returns the height of the rectangle	None	A data item, type int
findArea	Calculates and returns the area of the rectangle	None	A data item, type int

//Rectangle3 Class: Implementation

```
public class Rectangle3 {
    //the attributes
    private int length;
    private int height;

  /*the constructor method; has no type at all; bears same name as class name*/
     public Rectangle3()  // default constructor defined
    {
    }
    // user-defined constructor defined; more than one constructors are allowed
```

```
 public Rectangle3( int l, int h)
 {
   length = l;
   height = h;
 }
 //this method is used to read the length attribute
 public int getLength()
 {
   return length;
 }
 //this method is used to read the height attribute
 public  int getHeight()
 {
    return  height;
 }
//this method allows us to write the length attribute
public void setLength( int l)
 {
        length =  l;
 }
//this method is used to write the height attribute
public void setHeight( int h)
 {
        height =  h;
 }
// this method returns the area of the rectangle
public int findArea()
 {
        return length * height;
 }
// this method returns the perimeter of the rectangle
public int findPerimeter()
 {
        return  2 * (length + height);
 }
}// end of class
```

Figure 6.23: Recatngle3 class implementation

//Rectangle3Use Class: Using the Rectangle 3 Class Template

```
import java.util.Scanner;
public class Rectangle3Use {

    public static void main(String[] args)
    {  // Instantiating Scanner class object input
          Scanner input = new Scanner(System.in);
         //Instantiating x using default constructor
          Rectangle3 x = new Rectangle3();
      /* x object is set to default of zero until set
       functions are called to assign values */
          x.setLength(9);
          x.setHeight(5);
 // print the value of area from x object
System.out.println(“Area for x object = " +x.findArea());
//Reading in values of sides from keyboard
 System.out.println("The Height = " + y.getHeight(
 Int side1 = input.nextInt();
    Int side2 = input.nextInt();
// Instantiating y using user-defined constructor
          Rectangle3 y = new Rectangle3(side1, side2);
//print the values of sides in Rectangle object y
    System.out.println("The Length = " + y.getLength());
    System.out.println("The Height = " + y.getHeight());

 //print the Area from y object
   System.out.println(" Area for y object = " + y.findArea());

// change the state of object y variables again
        y.setLength(4);
                  y.setHeight(7);

//prints the values of sides in Rectangle object y
  System.out.println("The Length= " + y.getLength());
  System.out.println("The Height = " + y.getHeight());

//prints the Area from y object using current state
 System.out.println(" Area for y object = " + y.findArea());
 }
}
```

Figure 6.24: Using Rectangle3 class objects in Rectangle3Use client code

6.19 Manipulating Objects

In Java, when a reference is first created without assigning it to a new object in the same line, it is assigned a special value of NULL; meaning no storage is allocated. We can also personally assign a NULL value to a reference at any point in the program, e.g.:

```
Rectangle x; // here x has a null value
x =  new Rectangle(3, 5);//new rectangle object is created and set to initial values 3 and 5
// after some lines of codes here
x =  null;
If(x == null)
{
     System.out.println("No Storage is allocated to this object");
}
```

6.20 Scope: (Variables and Methods)

Objects have their own copies of the attributes defined in their class, so different objects have different attribute values. This is the normal case, and we say that these attributes and operations have ***instance*** scope.

e.g. **private int length;** // this variable has instance scope (instance variable)

However, sometimes it is useful to define attributes that have a single, shared value for every object of the class, and to have operations (like object creation and destruction operations) that don't operate on any particular class instance(object). So these attributes and operations have **class scope**. So, they are referred to as either class variables or class methods. Class scope features provide a set of global features for an entire class of objects. So the constructor methods or variables and methods declared **static** in Java have class scope.

The notation for instance scope and class scope attributes and operations:

Constructors are special methods(operations) that create new objects. They have class scope. If they were instance scope, you obviously couldn't invoke them to create the first instance as you would not yet have created any instances(object). Generic way is to call the constructor ***create(…).***

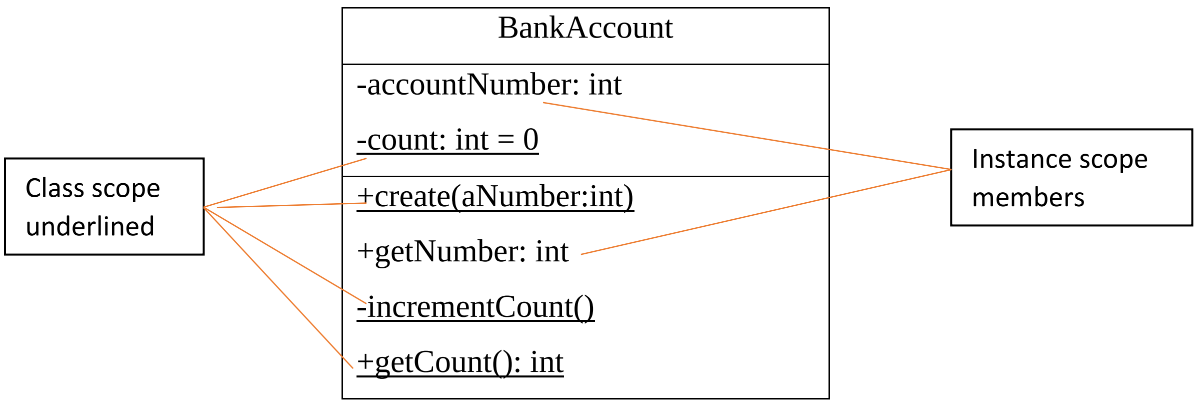


Figure 6.25: Instance scope and class scope attributes and operations

BankAccount **Constructor Example**

Every time you create a new ***BankAccount*** object, you have to pass in ***number*** as a parameter to the constructor to set the ***accountNumber*** attribute. This ensures that every BankAccount object has the ***accountNumber*** attribute value set at the point of creation- splendid practice. Different OO languages have different semantics for object destruction. *Java's garbage collector* automatically destroys objects when they're no longer used.

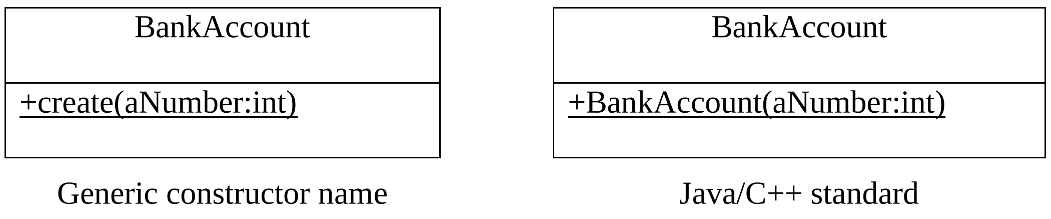


Figure 6.26: Constructor representation on UML.

Table 6.4: The semantics for instance scope and class scope attributes and operations:

	Instance Scope	**Class Scope**
Attributes	By default, attributes have instance scope Each object of the class gets its own copy of the instance attributes/variables Each object may have different instance scope attribute values	Attributes/variables may be defined as class scope Each object of the class shares the same, single copy of the class scope attributes Each object will have the same class scope attribute values
Operations	By default. Operations have instance scope Every invocation of an instance scope operation/method applies to a specific instance of the class One cannot invoke and instance scope operation unless there is an instance or object of the class in existence.	Operations may be defined as class scope Invocation of a class scope operation does not apply to any specific instance of the class. A class scope operation applies to the class itself. One can invoke a class scope operation even if there is no instance or object of the class in existence. This is used for object creation operation.

Scope determines access:

Whether an operation can access another feature of the class or not is determined by the scope of the operation and the scope of the feature it is trying to access.

Instance scope operations can access other instance scope attributes and operations, and also all of the class scope attributes and operations.

Class scope operations may *only* access other class scope operations and attributes. Class scope operations can't access instance scope operations because:

i. there might not be any class instances created yet;
ii. even if class instances exist, you don't know which one to use.

Scope Definitions in Java Example:

Instance Variables (Non-Static Fields): Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the ***static*** keyword. Non-static fields are also known as instance variables because their values are unique to each instance of a class (to each object, in other words); i.e. the ***current Speed*** of one ***Bicycle*** object is independent of the ***current Speed*** of another.

Example:

```
Class Rectangle{
        private int length; // this is an instance variable
       private int height;
      …
Rectangle obj1 = new Rectangle(3, 5); //obj1 and obj2 are two instances
Rectangle obj2 = new Rectangle(7, 9);   //having different states
```

6.21 Parameters and instance variables usage

Parameters: The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter accepting constructs as well (such as constructors). When the context calls for a distinction, we will use specific terms (static field, local variables, etc.) as appropriate.

Avoid method-parameter names or local-variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs. Using parameter names that are identical to the class's instance-variable names is not a recommended practice.

- E.g. is the following constructor method where this style is applied:

```
class Rectangle{
 private int length;
 private int height;
public Rectangle( int length, int height )
 {
        this.length = length;       // set "this" object's length
        this.height = height ; // set "this" object's height
 }
```

Figure 6.27 Implementation style Not Recommended

```
class Rectangle{
 private int length;
 private int height;
public Rectangle(int l, int h)
 {
        length = l; //parameter l sets the value of length
        height = h;
 }
```

Figure 6.28 Implementation style Recommended

6.22 Class scope: Using static keyword in Java

Class scope attributes and operations in Java example are referred to as class fields/ variables and class methods respectively. A ***class variable*** is any field declared with the ***static*** modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. This is used when only one copy of a particular variable should be shared by all objects of a class (as opposed to every object having its own copy of all instance variable of the class). A field defining the number of gears for a particular kind of bicycle could be marked as ***static*** since conceptually the same number of gears will apply to all instances. The code ***static int numGears = 6***; would create such a static field.

Similarly, a method preceded with the keyword *static* is tagged a ***class method***. To use this method, object creation is not needed.

A static method is invoked as follows:

ClassName.MethodName([argument])

```
public class BodyIndex
{     //index is a class method i.e. has class scope
public static void index(double weight, double height)
{
            double bmi = weight/ (height * height);
            System.out.println("The BMI is"+ bmi);
}
}
```

Figure 6.29: Coding Example: Body Mass Index class

```
public class UseBodyIndex {

public static void main(String[]args)
{

            Scanner sc = new Scanner(System.in);
            System.out.println("Enter weight and height");
            double w = sc.nextDouble();
            double h = sc.nextDouble();

            BodyIndex.index(w, h); // use via class name and method name only
}
}
```

Figure 6.30: Using Body index class object in a separate class

Further Note on Static keyword Implementation in Java

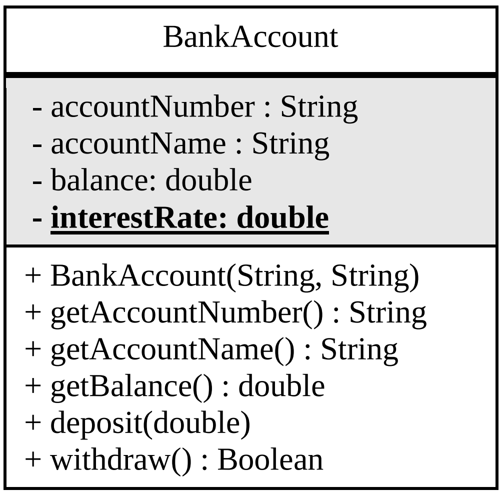
A non-static method can call any method of the same class directly (i.e. using the method name by itself) and can manipulate any of the class's field directly. Conversely, a static method can call only other static methods of the same class and can manipulate only static fields in the same class directly. *To access a class's non-static members, a static method must use a reference to an object of the class*. Recall that static methods relate to a class as a whole, whereas non-static methods are associated with a specific class instance (object) and may manipulate the object's instance variables.

Suppose a static method was to invoke a non-static method directly, how would the method know which object's instance variables to manipulate? What would happen if no objects of the class exist when the non-static method was invoked? This could pose a problem. *Therefore, Java does not allow a static method to access a non-static member of the same class directly*. Just as you have been informed, a static method can be called even when there are no objects of its class in memory. This is why "this" reference cannot be used in a static method. Because '***this'*** reference must refer to a specific object of the class.

More Examples: Implementing Class Scope

Consider the following UML notation for ***BankAccount*** Class. Note that ***interestRate*** variable has class scope- as it is underlined. So also ***setInterestRate***() and ***getInterestRate***() i.e. supporting operations have class scope.

A static(class) method can call only other static methods of the same class and can manipulate only static fields in the same class directly. Recall that static methods relate to a class as a whole, whereas non-static methods are associated with a specific class instance (object). A static method can be called even when there are no objects of its class in memory.



+ setInterestRate(double)
+ getInterestRate(): double

Figure 6.31: Detailed UML class diagram for BankAccount class

We include an attribute say ***interestRate*** to hold the current rate of interest. Now, interest rate should be the same for any customer, and any change should be the same for all. It implies that the change must be the same for every object of the class. This makes a good example of a shared data/class variable. Therefore, we can achieve this by declaring variable ***interestRate*** as ***static***. Any change made to it affects all objects in the class. Accessing this attribute without reference to a specific object should make sense and should be possible. All we need do is to create and declare methods such as ***setInterestRate*** and ***getInterestRate*** as ***static*** (to manipulate the variable). Thus, they are made class methods by this keyword; and they do not refer to any specific object. Then we can call these methods by using the class name instead of the object name. Then the third method ***addInterest*** will add interest to the customer's balance.

```
Public class BankAccount
{
Private String acctNum;
Private String acctName;
Private double balance;
Private static double interestRate;/*this is accessed by client code  only through
methods of the class. */
Public BankAccount(string num, String name) //constructor
{
acctNum = num;
acctName = name;
balance = 0;
}
/* static class member exist even when no object of the class exist. To access private static
member(e.g. interestRate) when no object exist, create a public static method, call through that
name and the class name using a dot(.) separator */
```

```
Public static void setInterestRate(double rate)
{
interestRate = rate;
}
Public static double getInterestRate()
{
return interestRate;
}
//operation to fetch balance
Public double getBalance()
{
    return balance;
}
// method to make deposit
Public void deposit(double amt)
{
balance = balance + amt;
}
```

Figure 6.32: BankAccount Class, implementing Class Scope

Implementation example: The *Tester/Client Program*

The following Figure 6.33 is the client code to demonstrate the use of the BankAccount class created:

```
Public class BankAccountTester
{
Public static void main(String [] args)
{
// create a bank account
BankAccount account1  = new BankAccount("1021030001", "Owolabi Ireti");
// create another bank account
BankAccount account2  = new BankAccount("1021030002","Donnet Cyndy");
// make a deposit into the first account
account1.deposit(1000);
// set the interest rate
BankAccount.setInterestRate(10);
//add interest to accounts
account1.addInterest();
// display account details
System.out.println(“Account number:” + account1.getAccountNumber());
System.out.println(“Account name: “ + account1.getAccountName());
System.out.println(“Interest Rate: “ + BankAccount.getInterestRate());
System.out.println(“current Balance: “ + account1.getBalance());
}
}
```

Figure 6.33: Client code to demonstrate the use of instance scope

Task 6.1

a) In structured programming paradigm, the basic building blocks are procedures and data structures but in Object Oriented Programming (OOP) the basic building block are……………….. and is implemented in ……………

b) Look around you, objects are everywhere! Barely any noun can be reasonably represented as a software object in terms of ***attributes*** (e.g. name, size etc.) and ***behaviours*** (e.g. calculate, move, fly, etc.). Identify 3 objects where you are (or imagine yourself somewhere else), name the objects and state their likely attributes and behaviours.

Object	**Property**	**1**	**2**	**3**
1)	Attribute			
	Behaviour			
2)	Attribute			
	Behaviour			
3)	Attribute			
	Behaviour			

Task 6.2

1) Briefly explain the idea of class creation in line with making a datatype.
2) Why do we characteristically make the data members private:
3) Draw the UML class diagram for a named class having two methods in which one is public and the other is void
4) State the implications of the following symbols in a UML class diagram:
    a) *minus sign (-)*
    b) *hash sign (#)*
5) The **set** method could also be used to validate data before assigning it to the data member. Write a **set** method that validates user's input for a clock object such that "**26**:50:12" would be alarmed invalid for **hh:mm:ss** format (hours can't be up to 26).
6) Imagine the various activities at an ATM (Automated Teller Machine) that can aid successful transaction between your Bank and you. Using object orientation concept, implement operations that would do the following: (i) checking your balance (ii) making withdrawal and (iii) making deposit.

Task 6.3

1. Mention at least 3 important characteristics of a constructor method.
2. What is the scope of a constructor method?
3. What is the scope of a static method and how is it used?
4. What is the difference between a local variable and an instance variable?
5. Create a class called **Employee** that includes three pieces of information as data members—a first name (type string), a last name (type string) and a monthly salary (type **int**). Your class should have a constructor that initializes the three data members. Provide a ***set*** and a ***get***

methods for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class Employee's capabilities. Create two Employee objects and display each object's *yearly* salary. Then give each Employee a 10 % raise and display each Employee's yearly salary again.

Note: Encapsulation requires you to hide your data, while you expose the behaviours

Task 6.4

1) What are the advantages and advantages of inheritance (if any).
2) Explain ***Overloading*** and ***Overriding*** as characteristic features of polymorphism in Object Oriented Programming.
3) Using Object Oriented Programming, implement a Class for Calculator. Four methods called Add, Multiply, Subtract and Divide should be created within the Calculator class to implement addition, multiplication, subtraction and division. System accepts two inputs at a time, and the output is displayed after correct input. A loop keeps the system running until user choses to exit.

Task 6.5

1) Define the following terms: (i) Object (ii) Attribute (iii) Behaviour
2) Define the following terms: (i) Mutator (ii) Accessor (iii) Constructor
3) State the difference between a class variable and an instance variable.
4) A class keeps track of a pressure **Sensor** in a laboratory. When a Sensor object is created using the first constructor, the initial pressure is set to zero. When it is created using the second constructor it is set to the value of the parameter. The pressure should not be set to a value less than zero. Therefore, if the input parameter to the ***setPressure*** method is a negative number, the pressure should not be changed and a value of false should be returned. If the pressure is set successfully, a value of true should be returned.

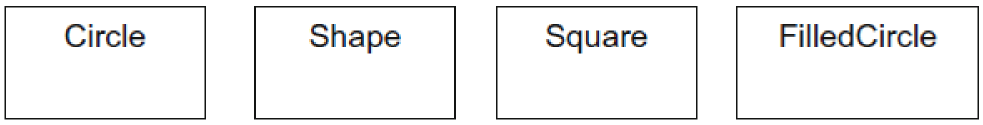
   (i) Draw a UML class diagram for this design

   (ii) Write the code for the Sensor class.

   (iii) Develop a **SensorTester** *program to test the Sensor class.*

Task 6.6

1) Consider the following classes and arrange them into an inheritance hierarchy:



2) Create a class called **Date** that includes three pieces of information as data members—a month (type int), a day (type int) and a year (type int). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. Assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn't, set the month to 1. Provide a member method **displayDate** that displays the month, day and year separated by forward slashes (/).Create a Date object in a main method and invoke **displayDate** method.

Solution to Task 6.3

1. Mention at least 3 important characteristics of a constructor method.

Solution:

a) It is used to create an object

b) It can be used to set initial values to required member variables

c) It is a public method that is not void but still not written to return any value

2. What is the scope of a constructor method?

Solution: A constructor method has class scope

3. What is the scope of a static method and how is it used?

Solution: A static method has class scope. The class name, a dot(.) operator and the method name. No instantiation is required to use a static method.

E.g. ClassName.Methodname() as in : Math.sqrt(9);

4. What is the difference between a local variable and an instance variable?

Solution:

Instance Variables: are "non-static fields" where objects store their individual states. Their values are unique to each instance of a class (to each object); i.e. the *current Speed* of one *Bicycle* object is independent of the *current Speed* of another.

E.g.

public int length;   //this is an instance variable

Class Variables (Static Fields): This is declared with the *static* modifier. It is used when only one copy of a particular variable should be shared by all objects of a class. E.g. public *static int numGears = 6*;

5. Create a class called **Employee** that includes three pieces of information as data members—a first name (type string), a last name (type string) and a monthly salary (type **int**). Your class should have a constructor that initializes the three data members. Provide a ***set*** and a ***get*** methods for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class Employee's capabilities. Create two Employee objects and display each object's *yearly* salary. Then give each Employee a 10 % raise and display each Employee's yearly salary again.

Note: Encapsulation requires you to hide your data, while you expose the behaviours

Solution:

```
//This is the implementation of the Employee class
package javaapplication1;
public class Employee {
    private String firstname;
    private String lastname;
    private int monthlySalary;
    //default constructor
    public Employee()
    { }
 // user-defined contructor
    public Employee(String firstN, String lastN, int salary)
    {
            firstname = firstN;
            lastname = lastN;
            if(salary<0)
             {
                     monthlySalary = 0;
             }
             else
             {
                     monthlySalary = salary;
             }
    }
//get Methods
    public String getFirstname()
    {
        return firstname;
    }

    public String getLastname()
    {
        return lastname;
    }
     public int getMonthlySalary()
    {
        return monthlySalary;
    }
// set methods
    public void setFirstname( String firstN)
    {
        firstname = firstN;

    }
     public void setLastname( String lastN)
    {
        lastname = lastN;

    }
public void setMontlySalary( int salary)
    {
        if(salary<0)
        {
            monthlySalary = 0;
        }
        else
        {
             monthlySalary = salary;
        }

    }
//yearly salary calculation
    public int yearlySalary()
    {
        return 12*monthlySalary;
     }
    //pay raise calculation method
    public int salaryRaise(int raise)
    {
         return monthlySalary + monthlySalary * raise/100;
    }
    }
```

```
//+++++++ The Client Programme ++++++++++++++++++
 // This is where the Employee class objects are instantiated
package javaapplication1;

public class EmployeeTest {

public static void main( String[] args )
{
   //user-defined constructor
    Employee x = new Employee("Adekola", "Dan", 1000);

    System.out.println("First Name =" + x.getFirstname());
    System.out.println("Last Name =" + x.getLastname());
    System.out.println("Salary =" + x.getMonthlySalary());
    System.out.println("Yearly Salary =" + x.yearlySalary());
    System.out.println();
Employee y = new Employee();  // using default constructor

    y.setFirstname("David");
    y.setLastname("Home");
    y.setMontlySalary(6000);

    System.out.println("First Name =" + y.getFirstname());
    System.out.println("Last Name =" + y.getLastname());
    System.out.println("Salary =" + y.getMonthlySalary());
    System.out.println("Yearly Salary =" + y.yearlySalary());
    System.out.println("Raise =" + y.salaryRaise(10));

}//end of main method

}//end of class
```

Chapter 7
ARRAYS AND LIST

In this chapter, a student is expected to understand well the following:

- *Definition of an array*
- *How to create an array (Linear Array)*
- *Arrays as a method input and output*
- *Manipulating arrays*
- *Multidimensional arrays*
- *Creating ragged arrays*
- *Passing Array as object*

7.1 Definition of Array

Array is a data structure that collects data or elements of the same type. Elements of a particular array must be of the same type but not a mixture e.g. collection of integer data as a named array. Array is a container that stores collection of items (called elements). Each array element is referenced or accessed through its index.

Briefly, a data structure is the arrangement of data in the computer memory e.g. array, stack, tree, queue etc.

Array could be one-dimensional or multi-dimensional.

Illustrative Example:

An example of where an array can be used is in an attempt to process the temperature of an environment for a week. These kind of data is the same from day 1 to day 7. Your option is to declare seven variables of the same type as:

```
double t1, t2, t3, t4, t5, t6, t7;
```

There are difficulties working with this for example, you cannot use a loop construct to manipulate the content of variables. The way out is to declare an array and allocated memory to store the elements.

7.2 Steps in creating an array:

1). Declare an array variable

2). Allocate memory to store the array elements

1) Declare an array variable in Java (One- Dimensional Array):

data type [ ] variable;

double [ ] t;

Variable t (for temperature) above is a **one-dimensional array also called a list or linear array** *which holds a reference to the array elements.* A reference is a variable that holds a location in the computer memory(known as memory address) where data is stored, rather than the data itself. By declaring an array reference, you want to point to the memory address where the data is stored.

2) Allocate memory to store the array elements:

t = new double[7];

The above code means: create space in memory for an array of 7 elements and type double.

Or once or together as follows:

double [ ] t = new double[7];

To allocate memory is to state and carve out size and space for the array(i.e. the maximum number of elements expected to fill the array). The array type and size are then put together with a special **new** operator.

Note: The keyword **new** creates space in memory for an array of the stated size and element type.

Note on Array Size:

Note that the size needed must be known and set before you can start using the array. And once the size is set, it cannot be changed. It is wise to allocate space larger than needed when one is not sure of the actual space needed. Though this is responsible for memory wastage attributed to array use but it's a safer option. If you allocate less than needed, we may have problems with our program such as inability to capture some data. Also note that size should be positive value.

7.3 Indexing an Array:

Array uses same variable **name** but individual element is uniquely identified by an **index**. Array name such as **'t'** above is like a street name while array index represents each house number. Array indices (plural of index) are usually contiguous integers and Java language specifically starts indexing from zero (0).

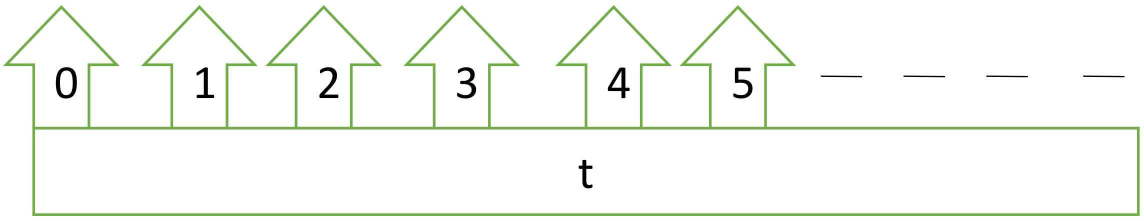
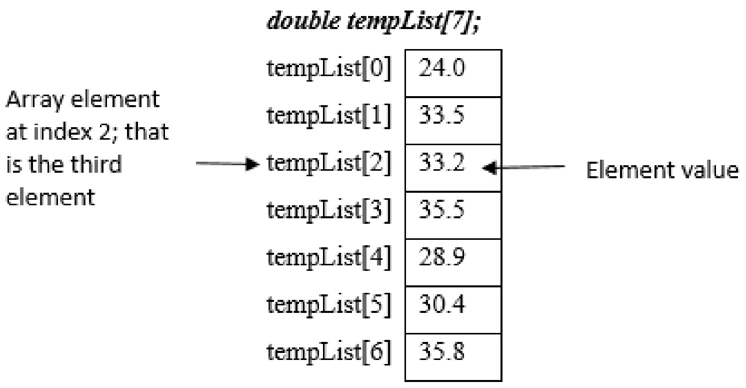


Figure 7.1 Array indexing

For one-dimensional array, there will be a single *subscript (**index**) whose* value refers to individual array elements. If the array contains n elements, the subscript will be an integer quantity whose values range from 0 to n-1 .

One-Dimensional Array Structure: Example

Consider the following example array named **tempList** having 7 data of the same type double defined as:



tempList[0]	24.0
tempList[1]	33.5
tempList[2]	33.2
tempList[3]	35.5
tempList[4]	28.9
tempList[5]	30.4
tempList[6]	35.8

Figure 7.2: Linear Array named ***tempList*** having 7 elements of type double

Therefore:

tempList[0] //implies the first element; indexing starts from zero (0)

tempList[6] //implies the seventh / last element

tempList[7] //is invalid; there is no such element; if you attempt to reference this, //you get an exception error: java.lang.Array.IndexOutOfBoundsException. Exception error results in program terminations.

Therefore, for an **n**-element array, the subscripts always range from 0 to n-1. The elements an array contains determine (inform) its type e.g integer, character type.

7.4 Assigning elements:

If all elements are known before hand, then **new** operator is not needed to allocate memory. Initialization and allocation can be done as follows with each element separated by comma as seen in the following example:

double [ ] t = {1.0, 3.4, 2.5, 9.0, 7.5, 5.0, 4.0 }

In this case, you need not include the number of elements; the compiler automatically determines the size of the array by itself. Once an array is created, you can access the elements individually. This is the term called indexing the array. Then, t[0] has 1.0, t[1] has 3.4, t[6] has 4.0 by the array named t above.

Whether initialized or not, values can be placed into individual array element. We can assign values into the array with "="(assignment) operator using for example direct assignment e.g.

```
t[0] = 1;
t[1] = 3.4;
```

Or by reading it in through the keyboard with:

```
t[0] = input.nextDouble();
```

Also, System.out.println(t[5]); prints the sixth element on the screen.

You can use an array element the same way you use other variables e.g.:

```
t[4] = t[4] + 3; //simply means, add the value in t[4] to 3 and store the result in t[4].
```

Using loop on Array Elements:

A loop can be used to assign elements to an array:

```
1)  for(int i = 0; i < 7; ++i)
2)      {
3)          System.out.println("Enter the day's Temperature:");
4)          t[i] = input.nextDouble();
```

```
}
```

There is a method (called **length**) in java that returns the value of the size of the array. This could be used instead of stating 7 as seen in the loop above. It is written by combining the name of the array with keyword **length** using a dot **(.)** operator.

// Using t.length . Note that the square bracket is not required.

Therefore, to display the elements in the array we write:

```
for(int i = 0; i < t.length; ++i)
{
   System.out.println("Element " +(i+1) +"is  "+ t[i]);
}
```

The following output is received assuming we are referencing the initialized array t:

```
Element  1 is 1.0
Element  2 is 3.4
Element  3 is 2.5
Element  4 is 9.0
Element  5 is 7.5
Element  6 is 5.0
Element  7 is 4.0
```

7.5 Passing Array as object:

Array can be used both as parameter to a method and as return values. Array is passed to a method as object. Parameter is declared as an array type without having to include the size of the array in the method header:

- **e.g.**

```
static void readT(double [ ] tmp ) //note that tmp is a formal parameter standing for t
{
  Scanner input = new Scanner(Sysem.in);
  for (int i  = 0; I < tmp.length; ++i)
  {
    System.out.println("Enter Temperature for day: "+ i+1);
    tmp[i] = input.NextDouble();
  }
}
```

Array pass by reference explained:

The following shows how the array method can be called from another method like the main():

```
class Temperature {
   public static void main(String args[]) {
       double [ ] t;
        t = new double[7];
        readT( t ); // t is passed to the method without any square bracket
   }
//this method fills in the content of the array t
Static void readT(double [ ] tmp ) // note that tmp is a formal
{                                              parameter standing for t */
  Scanner input = new Scanner(Sysem.in);
  for (int i  = 0; i < tmp.length; ++i)
  {
     System.out.println("Enter Temperature for day: "+ i+1);
     tmp[i] = input.NextDouble();
  }
}
```

When an ordinary variable e.g. x = 10 is passed to a method, it is only a copy of that value that is passed, then any change made to x in that method does not affect x outside that method. This is the pass by value we have earlier discussed.

Conversely, when an array is passed a copy of the array reference is passed. This is not a copy of each array element or value but the location / memory address of the array that is sent. This is why the receiving method could fill the array with values and as such change the content of the array permanently. Therefore, a change inside this method affects the content of the array. So when this happens, we call it pass by reference. It does not matter if a different variable name is used, as tmp (formal parameter) is used for t (actual parameter) in the above experiment, the change in the method is felt outside the method. This method points to the memory address /location and writes the result there.

Returning an array from a method:

We can create an array within a method instead of sending it to it. We can also fill the content there and then return/send the array.

- e.g.

```
static double [ ] readT( ) /* [ ] after double means this is an array method */
{
  Scanner input = new Scanner(Sysem.in);
  double [ ] tmp = new double[7];
  for (int i  = 0; i < tmp.length; ++i)
  {
    System.out.println("Enter Temperature for day: "+ i+1);
    tmp[i] = input.NextDouble();
  }
  return tmp; //only the name needed to return the array
}
```

Returning an array from a method (Complete Program):

```
class Temperature {
  public static void main(String args[]) {
double [ ] t;
      t = readT( );// this method fills the array and return it.
 displayT(t);//this displays the content filled by readT()
 }
 Static double [ ] readT( )
 {
     Scanner input = new Scanner(Sysem.in);
     double [ ] tmp = new double[7];
     for (int i  = 0; i < tmp.length; ++i)
     {
        System.out.println("Enter Temperature for day: "+ i+1);
        tmp[i] = input.NextDouble();
     }
     return tmp;
   } // end of readT method

 //beginning of the method to display result
   Static void displayT(double []  tp)
   {
     for(int i = 0; i < tp.length; ++i)
   {
        System.out.println("Element " +(i+1) +"is  "+ t[i]);
    }
    }
}
```

- Note that the calling pattern has changed from something like readT(t); to readT(); – does not need any parameter. This is because the first program declared and allocated the memory to the array then sends the array to another method to fill the content. But the second method declared and allocated the array by itself and then sent the array to another array t which automatically assigns all the contents of tp to t at point: t = readT( ); By this, array t becomes everything tp is, both in size and in content. Note that we only stated what t is in terms of type, the assignment operation makes both same in terms of size and elements.
- displayT(t); this is another method in which the newly filled array t is passed to, so that we can finally print and see our array contents / elements.

7.6 Assigning array size through input:

As far as array is concerned, the size must be predetermined before it can be used. That is the compiler would have to allocate the memory needed first before it can be used. We can work around assigning this as the program executes, then it is supplied by the user. So instead of being confined to the following all the time:

```
int [ ] array1 = new int[7];
```

We can re-write our memory allocation as follows:

```
1)  Static double [] readT( )
2)   {
3)           Scanner input = new Scanner(Sysem.in);
4)           int [ ] array1;
5)           int size = input.nextInt(); // size set as program runs
6)           array1 =  new int [size];
7)           …
8)  }
```

Finding the largest element for an array:

The algorithm is as follows:

```
1) Set result to first value in the array
2) Loop from the second element to the last
   2.1) If current-element > result
        2.1.1) Set result to current-element
   End the loop
```

3) *Return result*

Corresponding code method to solve the problem:

```
static int maxElt( int [ ]  array1 )
 {
        int result  =  array1[ 0 ] ;
        for(int i = 1; i < array1.length; ++i)
     {
         if (array1 [i] > result)
                    {   result = array1[ i ];  }
        return result;
}
```

Searching if an element exists in an array:

Here, we introduce the **linear search** in which you search the array through from the beginning to the end, then the search method returns the index of where the element found otherwise it means the element does not exist in the array, hence, a **dummy index** is returned. A dummy is usually a value that cannot be available in a particular operation. E.g. since an array index cannot negative we can use – 99 as our dummy index.

//The following method solves the problem:

```
static int searchArray( int [ ]  array1 , int val)
{
for(int i = 0; i < array1.length; ++i)
{
if (array1[ i ] == val)
  {
     return i;
  }
}
return – 99 ; // this is a test criterion; a method receiving this knows that
}                           //the  values searched for is not available in  the array
```

7.7 Multi-Dimensional Arrays:

In our previous example, we used array named "[]t" to hold the values of seven different temperature values in one week. The array index allows us to loop through the elements of the array(as in t[i]) and reference each element via its index (e.g. t[2]; pointing to the third element. Assuming we want to process temperatures for four weeks of a month. The declaration and memory allocation below could be used:

```
double[] t1 = new double [7];  // for week one temperatures
double[] t2 = new double [7];  // for week two temperatures
double[] t3 = new double [7];  // for week three temperatures
double[] t4 = new double [7];  // for week four temperatures
```

- The above means that four different arrays were created bearing different array names; t1, t2, t3 and t4.

Since there are four different arrays with different names, it is required to process the array differently. For example, to enter values into them will require that we run each through an independent loop. Therefore, it means four different loops is needed to fill all the arrays with elements. This could be made less tedious by creating a ***multi-dimensional*** array.

A multi-dimensional array is an array that has more than one index. The arrays used so far has one index and are therefore called one-dimensional array. If we require to process temperatures for each month of the year we may require 3 indexes (one for the month, one for the week and one for the day). Since the number of indexes is what is used to name the dimension, then, we say we have a 3-dimensional array.

For the problem at hand, all we need is to process daily temperature readings in four weeks. Therefore, we only need to cater for the weeks and the days. Hence, two indexes (indices) are required and such array is termed two-dimensional.

Creating a two-dimensional array:

We need to know the size of both indexes e.g. 4 weeks and 7 days.

```
        double [ ][ ] t;      / / declaration of a two dimensional array
        t = new double [4] [7];  // creation of memory (4 by 7 array)
   or
        double [ ][ ] t = new double [4] [7];
```

Here we use a two pairs of square brackets. The pairs of brackets increases accordingly as array dimension increases. In the above illustration, based on our choice, we use the first square brackets to represent the number of weeks and the second to represent the number of days.
While a one-dimensional array is regarded as a list, a 2-D is seen as a table with rows and columns (though implemented as array of arrays in Java). The name of each item in a two-dimensional array is the array name, plus the row and column index.

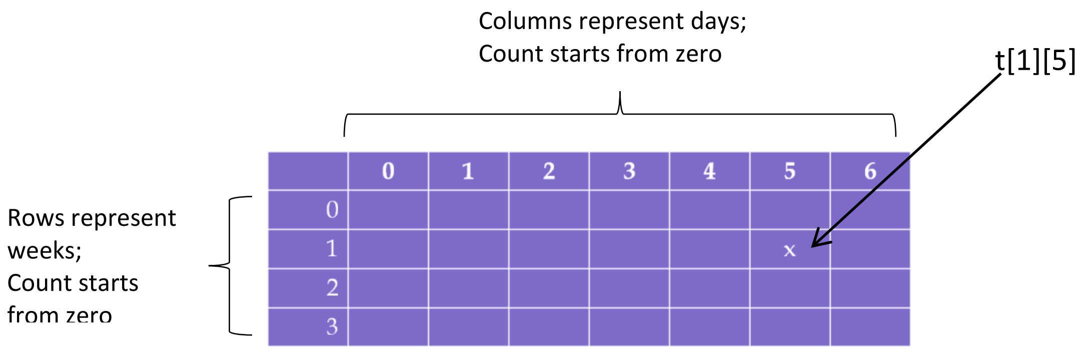


Figure 7.3: t[1][5] marked x represents 2nd week 6th day

Multidimensional array Implementation:
In the Java, a multidimensional array is an array whose components are themselves arrays. A consequence of this is that the rows are allowed to vary in length, as shown in the following ***MultiDimArrayDemo*** program:

```
class MultiDimArrayDemo {
public static void main(String[] args) {
String[][] names = {
                        {"Mr. ", "Mrs. ", "Ms. "},
                        {"Smith", "Jones"}
};

// Mr. Smith
System.out.println(names[0][0] + names[1][0]);
// Ms. Jones
System.out.println(names[0][2] + names[1][1]);
}
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

7.8 Collections and Class Arraylist

Collection is about making data structures with many capabilities. We can create our own collections but first it is good to mention that Java API provides several predefined data structures, called **collections,** used to store groups of related objects. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This saves program development time.

The conventional array definition does not automatically change size at execution time to accommodate additional elements. The collection class **ArrayList<T> (belonging to package java.util)** offers a convenient way out. It can *dynamically change its size to accommodate more elements. The T (by convention) is* a ***placeholder****—when declaring a new ArrayList, replace it with the type of elements that* you want the ArrayList to hold. Just like specifying type when declaring an array. But *only* ***non-primitive*** *types can be used with these collection classes.*

- **ArrayList< String > list;**

The above statement declares variable ***list*** as ArrayList collection to store strictly string values. Classes with this kind of placeholder that can be used with any type are called **generic classes. The following is the table of some common methods of ArrayList<T> collection:**

Table 7.4: Methods in ArrayList Collection class

Method	Description
add	Adds an element to the end of the ArrayList.
clear	Removes all the elements from the ArrayList.
contains	Returns true if the ArrayList contains the specified element; otherwise, returns false.
get	Returns the element at the specified index.
indexOf	Returns the index of the first occurrence of the specified element in the ArrayList.
remove	Overloaded. Removes the first occurrence of the specified value or the element at the specified index
size	Returns the number of elements stored in the ArrayList.
trimToSize	Trims the capacity of the ArrayList to current number of elements

//Implementation Example:

```
import java.util.ArrayList;
public class ArrayListCollection
{
public static void main( String[] args )
{
// create a new ArrayList of Strings with an initial capacity of 10
J:    ArrayList< String > items = new ArrayList< String >();
K:   items.add( "red" ); // append a string value “red” to the list, list is initially empty
L:   items.add( 0, "yellow" ); // insert the value at index 0; perform a shift where needed
// display the colors in the list
M:     for ( int i = 0; i < items.size(); i++ )
N:           System.out.printf( " %s", items.get( i ) );
// display colors using method call
    display( items, "\nDisplay list contents with enhanced for statement:" );

O:     items.add( "green" ); // add "green" to the end of the list
P:     items.add( "yellow" ); // add "yellow" to the end of the list

      display( items, "List with two new elements:" );

Q:  items.remove( "yellow" ); // remove the first "yellow“
display( items, "Remove first instance of yellow:" );

R:  items.remove( 1 ); // remove item at index 1
display( items, "Remove second list element (green):" );

// check if a value is in the List
System.out.printf( "\"red\" is %sin the list\n", items.contains( "red" ) ? "": "not " );
// display number of elements in the List
System.out.printf( "Size: %s\n", items.size() );

} // end main

// display method definition
      public static void display( ArrayList< String > items, String header )
     {
      System.out.print( header );
      for ( String item : items )
         System.out.printf( " %s", item );

     System.out.println(); // display end of line
  } // end method display

} // end of class
```

Note: The alphabets (J, K, L,…) used are not part of the code but are only used to tag some lines in need of more explanations.

Output:

```
The first loop Display of content gives: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Important Note:

The code above utilized some ***ArrayList*** capabilities. ***Line J*** creates a new empty ***ArrayList*** of type ***String*** with a default initial capacity of ***10*** elements. This is the minimum number of elements ***ArrayList*** can hold without growing. When the ***ArrayList*** grows, it will create a larger internal array and copy each element to the new array. For efficiency, ArrayList grows only when an element is added *and the number of elements is* equal to the capacity—i.e., when there's no space for the new element.

Illustration on the methods utilized:

- The ***add*** method with one argument appends its argument (i.e. the value) to the end of the ArrayList(line K). The add method with two arguments inserts a new element at the specified position (line L). The first argument is an index. Collection indices also start at zero. The second argument is the value to insert at that index. Inserting an element is usually slower than adding an element to the end of the ArrayList.

- The ***size*** method (line M) returns the number of elements currently in the ***ArrayList***.
- The ***get*** method (line N) obtains the element at a specified index. Lines O and P add two more elements to the end of the collection.
- The ***remove*** method is to remove an element with a specific value; its first occurrence (line Q). If such element is does not exist, no operation is done. An overloaded version of the method (line R) removes the element at the specified index. When an element is removed, the indices of all elements after the removed element decrease by one (like a shift operation).
- The ***contain*** method is a **boolean** type that checks if an element exist in the ***ArrayList***. The method compares its argument to each element of the ArrayList in order, *so using contains on a large ArrayList can be inefficient.*

Task 7.1

A magic word square is a square where a word can be formed from reading each row and each column. For example, the following is a 4 by 4 magic word square:

‘P’	**‘R’**	**‘E’**	**‘Y’**
‘L’	**‘A’**	**‘V’**	**‘A’**
‘O’	**‘V’**	**‘E’**	**‘R’**
‘T’	**‘E’**	**‘N’**	**‘D’**

a) Declare and initialize a 2D array, magicSquare, to hold the words illustrated above

b) Write a method, displayRow, that accepts the magicSquare array and a row number and displays the word in that row.

c) Write a method, displayColumn, that accepts the magicSquare array and column number and displays the word in that column

d) In a two dimensional array named truthTable, what is the difference between these two references: truthTable.length and truthTable[0].length

Task 7.2

Malam Gambo was offered a two-week contractual terminal appointment to supply oil (in litre) to Eko PLC. She made different supplies per day. Write a Java Class using different named **methods** to accomplish the following:

a) Read in daily supplies for the two weeks (use a multi-dimensional array; there are 7 days in a week)
b) Compute the cost of daily supply stored in another array if oil is 155 naira per litre.
c) Display the cost of all the daily supply in naira
d) Display the total cost of supplies for the entire period

Task 7.3

Write a program to sum all the elements of a 2-dimensional array and print the total summation.

Summing all elements in a 2-D: Guide:

```
int total = 0;
for ( int row = 0; row < a.length; row++ )
{
for ( int column = 0; column < a[ row ].length; column++ )
      total += a[ row ][ column ];
} // end outer for
```

Chapter 8
OBJECT ORIENTED PROGRAMMING: INHERITANCE

This chapter demonstrates design and code reuse offered by inheritance- a unique feature of object oriented programming. Other features as overriding and overloading which are polymorphic characteristics are also demonstrated.

8.1 Inheritance

This introduces one of the primary capabilities of Object-Oriented Programming Paradigm based on organizing classes into a hierarchical structure described as the concept of ***Inheritance***. It is a form of software reuse in which a new class is created by absorbing (inheriting from) an existing class's members and embellishing them with new or modified capabilities. This earns us ***time*** and ***quality*** advantages. The existing class is called the ***superclass,*** and the new class is the ***subclass*** *(also called derived or child class)*. Each subclass can become a superclass for future subclasses. A subclass can add its own fields and methods. Therefore, a subclass is *more* ***specific*** than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass. The **direct superclass** is the superclass from which the subclass explicitly inherits. An **indirect** superclass is any class ***above*** the direct superclass in the class hierarchy, which defines the inheritance relationships between classes.

In Java, the class hierarchy begins with class **Object** (in package java.lang), which *every class in Java directly or indirectly extends* (or "inherits from"). ***Methods*** in ***Object*** are inherited by all other Java classes. Java supports only ***single inheritance***, in which each class is derived from exactly *one direct superclass. Unlike C++, Java does not support* ***multiple inheritance*** *(which* occurs when a class is derived from more than one direct superclass).
Inheritance is described as ***"Is-a" relationship.*** In an *is-a relationship, an object of a subclass can also be treated as an object of its superclass —e.g., a car is a vehicle.*

Superclasses and Subclasses:

Superclasses tend to be "more general" and subclasses "more specific." For example, a **CarLoan** *is a* **Loan**. Thus, class CarLoan can be said to inherit from class Loan. In this context, class Loan is a superclass and class CarLoan is a subclass. A CarLoan *is a specific type of Loan,* *but it's incorrect to claim that every Loan is a CarLoan—the Loan* could be any type of loan. Because every subclass object *is an object of its superclass, and one superclass can have* many subclasses, ***the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses.*** For example, the superclass Vehicle represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass Car represents a smaller, more specific subset of vehicles.

Inheritance hierarchy for university Community Members:

A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members. Faculty members are either administrators (e.g., deans and head of departments) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors or seniors. Each arrow in the hierarchy represents an ***is-a*** *relationship. As we follow the arrows* upward in this class hierarchy, we can state, for instance, that "an Employee *is a Community*-Member" and "a Teacher *is a Faculty member." Community Member is the **direct** superclass* of Employee, Student and Alumnus is an ***indirect*** superclass of all the other classes in the diagram. Starting from the bottom, you can follow the arrows and apply the *is-a relationship* up to the topmost superclass. For example, an Administrator *is a Faculty* member, *is an Employee, is a CommunityMember and, of course, is an **Object** (the root class).*

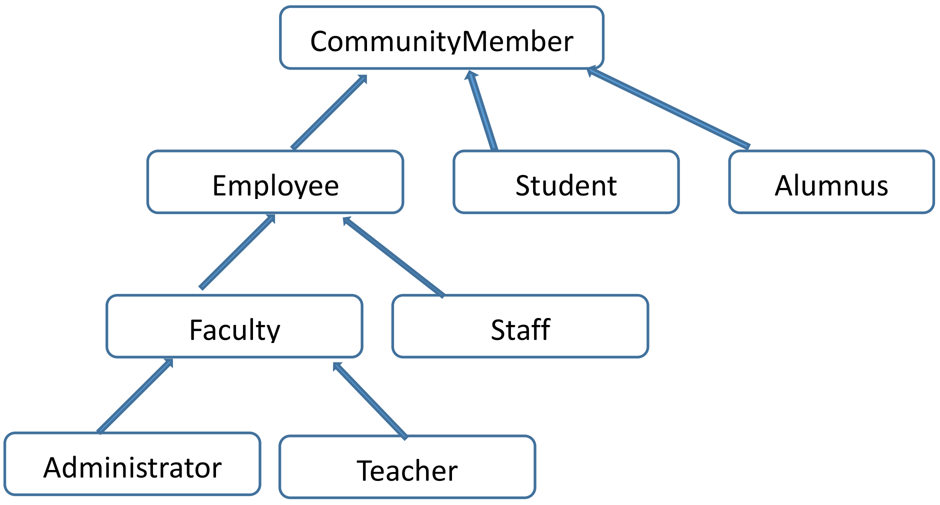


Figure 8.1 Inheritance relationships form treelike hierarchical structures

As mentioned earlier, Inheritance entails making a new class object from an existing class either by customizing or adding new unique characteristics. Therefore, stabilized components are easily reused. E.g. an object of class "convertible" can be made from general "automobile".
A ***subclass*** can ***customize*** methods that it inherits from its ***superclass***. To do this, the subclass **overrides** (redefines) the superclass method with an appropriate implementation (This is a feature in OOP called **Polymorphism-** this type of polymorphism is specifically called **overriding**).

8.2 Implementation example –Inheritance

One major advantage of object oriented approach to software development is the privilege of re-using already written classes (either by yourself or others). Take for example, if you have a need to write an ***Employee class*** and there exist a ***pre-written*** class called Employee that is purposed to do the similar task you are embarking on. Suppose the existing class does not do all that you intended, for example if your employees are part-time and you want ***attributes*** like ***hourlypay,*** or ***methods*** like ***calculateWeeklyPay*** and ***setHourlyPa****y* which do not exist in the first class.
There won't be any need to go and modify the old/existing class. All you need do is to exploit the OOPL feature called Inheritance which provides ability to ***extend*** existing classes by adding attributes and methods to them.
Inheritance is the ***sharing*** of attributes and methods among classes. We define a new class based on the existing/old one. The new class inherits all the attributes and methods of the old/existing, but also has attributes and methods of its own.

Using Employee class as an illustration, assuming the first/existing ***Employee*** class has 2 attributes, ***number*** and ***name***, a ***user-defined constructor*** method, and ***get***- and ***set***- methods for the attributes. Then, we define our new class called ***PartTimeEmployee*** class; this will inherit the attributes and methods of the Employee class but is also free to have attributes and methods of its own e.g. attribute - ***hourlyPay***, and method - ***calculateweeklyPay***.
The following Figure 8.2 shows an inheritance relationship between ***Employee*** and ***PartTimeEmployee***:

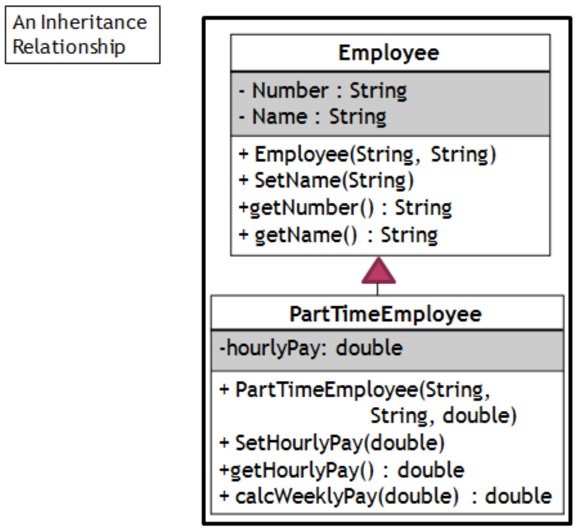


Figure 8.2: A UML notation for Inheritance indicated with a triangle in the diagram

// Implementation example of inheritance

```
//super class implementation
public class Employee {
    private String number;
    private String name;

    public Employee(String num, String nam)
    {
        number = num;
        name = nam;
    }

    public void setName(String nam)
    {
        name = nam;
    }
    public String getNumber()
    {
        return number;
    }
    public String getName()
    {
        return name;
    }

} // end of superclass
```

// Derived or subclass implementation

```
//subclass implemetation
public class EmployeePT  extends Employee {
    private double hourlyPay;
    public EmployeePT (String num, String nam, double hrPay)
    {
        super(num, nam);
        hourlyPay =  hrPay;
    }
    public double getHourlyPay()
    {
        return hourlyPay;
    }
   public void setHourlyPay(double hrPay)
    {
        hourlyPay =  hrPay;
    }
   public double calcWeeklyPay(int noOfhr)
    {
        return noOfhr * hourlyPay;
    }
}   //end of subclass
```

// Client / driver code containing the main method

```
//client / main program implementation
import java.util.Scanner;
    public class EmployeePTTest {
    public static void main(String[] args)
    {
        Scanner keyIn = new Scanner(System.in);
        Scanner keyStr = new Scanner(System.in);
        String number, name;
        double pay;
        double hours;
        EmployeePT emp;

      // read in values
        System.out.println("Emp no");
        number = keyStr.nextLine();
        System.out.println("Name");
        name = keyStr.nextLine();
        System.out.println("Pay");
        pay = keyIn.nextDouble();
        System.out.println("Hr worked");
        hours = keyIn.nextDouble();
//instantiating with user-defined constructor  method
        emp = new EmployeePT(number, name, pay);

        System.out.println(emp.getName());
        System.out.println(emp.getNumber());
        System.out.println(emp.calcWeeklyPay(hours));
    }
}
```

8.3 Writing Classes– Good Design Hints

- ***Always keep data private*** – a programmer may need to write an accessor or mutator method occasionally, but are still better off keeping the instance fields private. When data are kept private, changes in their representation do not affect the user of the class, and bugs are easier to detect. Doing anything else violates encapsulation.
- ***Always initialize data*** – Java will not initialize local variables, it will initialize instance of objects. Initialize the variables explicitly, either by supplying a default or by setting defaults in all constructors
- ***Don't use too many basic types in a class*** – The idea is to replace multiple related uses of basic types with other classes
- ***Not all fields need individual field accessors and mutators*** – a programmer may need to do a set or get, objects have instance variables that programmers may not want others to set or get.
- ***Use a standard form for class definitions*** – the users of the class are more interested in the public interface than in the details of the private implementation; and they are more interested in the methods than in data
- ***Break up classes with too many responsibilities*** – avoid the extremes (one complicated class (called God class) or one simple class with just one method).
- ***Make the names of your classes and methods reflect their responsibilities* –** just as variables should have meaningful name that reflect what they represent, so should classes. A good convention is that a class should be a noun or a noun preceded by an adjective or a gerund (an "ing") and methods standard convention is begin with a lowercase followed by an uppercase name.

8.4 More on Classes and Inheritance

Key Points:

- A class's private members are accessible only within the class itself.
- A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package—protected members also have package access.
- public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass.

■ When a subclass method overrides an inherited superclass method, the *superclass* method can be accessed from the *subclass* by preceding the superclass method name with keyword **super** and a dot (.) separator.

8.5 Implementation Example:

BasePlusCommissionEmployee is-a **CommissionEmployee**, where *firstName, lastName, socialSecurityNumber, grossSales* and *commissionRate* are attributes to be inherited, while *baseSalary* is the only attribute specific to **BasePlusCommissionEmployee** subclass.

a) If you have to restrict the security and the integrity of these inherited attributes to its super class and still make them accessible for the subclass, state what to do.

b) Show implementation of **CommissionEmployee** existing already as a superclass, implement the subclass **BasePlusCommissionEmployee** depicting inheritance relationship clearly defining *baseSalary* attribute and the class's ***constructor*** method.

c) Implement an override of the method called **earnings** which simply adds *baseSalary* to the super class ***earnings*** method implementation.

Note: **earnings** method returns the product of *commissionRate* and *grossSales* in the super class.

a) To promote data security and integrity, define the attributes to be inherited as private then create public methods to access them (getter and setter methods).

b) The following implements the **CommissionEmployee** as a superclass:

```
//CommissionEmployee extends Object //(i.e., inherits from) class Object (from
//package java.lang). Meanwhile, including this root base class is optional.
public class CommissionEmployee extends Object
{
 private String firstName;
private String lastName;
private String socialSecurityNumber;
private double grossSales; // gross weekly sales
private double commissionRate; // commission percentage
// five-argument constructor
public CommissionEmployee( String first, String last, String ssn,
double sales, double rate )
{
```

```
// implicit call to Object constructor occurs here
firstName = first;
lastName = last;
socialSecurityNumber = ssn;
setGrossSales( sales ); // validate and store gross sales
setCommissionRate( rate ); // validate and store commission rate
} // end five-argument CommissionEmployee constructor
// set first name
public void setFirstName( String first )
{
firstName = first; // should validate
} // end method setFirstName
// return first name
public String getFirstName()
{
return firstName;
} // end method getFirstName
// set last name
public void setLastName( String last )
{
lastName = last; // should validate
} // end method setLastName
// return last name
public String getLastName()
{
    return lastName;
} // end method getLastName
// set social security number
public void setSocialSecurityNumber( String ssn )
{
socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber
// return social security number
public String getSocialSecurityNumber()
```

```
{
return socialSecurityNumber;
} // end method getSocialSecurityNumber
// set gross sales amount
public void setGrossSales( double sales )
{
if ( sales >= 0.0 )
grossSales = sales;
else
throw new IllegalArgumentException(
"Gross sales must be >= 0.0" );
} // end method setGrossSales
// return gross sales amount
public double getGrossSales()
{
return grossSales;
} // end method getGrossSales
// set commission rate
public void setCommissionRate( double rate )
{
if ( rate > 0.0 && rate < 1.0 )
commissionRate = rate;
else
throw new IllegalArgumentException(
"Commission rate must be > 0.0 and < 1.0" );
} // end method setCommissionRate
// return commission rate
public double getCommissionRate()
{
return commissionRate;
} // end method getCommissionRate
// calculate earnings
public double earnings()
{
```

```
return commissionRate * grossSales;
} // end method earnings
// return String representation of CommissionEmployee object
@Override // indicates that this method overrides a superclass method
public String toString()
{
return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
"commission employee", firstName, lastName,
"social security number", socialSecurityNumber,
"gross sales", grossSales,
"commission rate", commissionRate );
} // end method toString
} // end class CommissionEmployee
```

- Constructors are *not* inherited However, a superclass's constructors are still available to subclasses. In fact, *the first task of any subclass constructor is to call its direct superclass's constructor*, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly.
- The **@Override annotation** indicates that method toString should override a superclass method.
- common errors include naming the subclass method incorrectly, or using the wrong number or types of parameters in the parameter list. Each of these problems creates an *unintentional overload* of the superclass method.
- *It's a syntax error to override a method with a more restricted access modifier—a public method of the superclass cannot become a protected or private method in the subclass; a protected method of the superclass cannot become a private method in the subclass.*

//CommissionEmployee as Super Class

```
public class CommissionEmployee extends Object
{
private String firstName;
private String lastName;
private String socialSecurityNumber;
private double grossSales; // gross weekly sales
private double commissionRate;
// constructors go here
// calculate earnings
public double earnings()
{
return commissionRate * grossSales;
}
```

// sub class creation using “extends”

```
public class BasePlusCommissionEmployee extends CommissionEmployee
{
private double baseSalary; // base salary per week

// six-argument constructor
public BasePlusCommissionEmployee( String first, String last,  String ssn, double sales, double rate, double salary )
{
// explicit call to superclass CommissionEmployee constructor
super( first, last, ssn, sales, rate );
setBaseSalary( salary ); // validate and store base salary
}//end of constructor
// set base salary
public void setBaseSalary( double salary )
{
if ( salary >= 0.0 )
baseSalary = salary;
else
throw new IllegalArgumentException(
"Base salary must be >= 0.0" );
} // end method setBaseSalary
// return base salary
public double getBaseSalary()
{
return baseSalary;
} // end method getBaseSalary
@Override // indicates that this method overrides a superclass method
public double earnings()
{
// not allowed: commissionRate and grossSales private in superclass
return baseSalary + ( commissionRate * grossSales );
}
// return String representation of BasePlusCommissionEmployee
    @Override // indicates that this method overrides a superclass method
public String toString()
{
// not allowed: attempts to access private superclass members
return String.format(
"%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
"base-salaried commission employee", firstName, lastName,
"social security number", socialSecurityNumber,
"gross sales", grossSales, "commission rate", commissionRate,
"base salary", baseSalary );
} // end method toString
} // end class BasePlusCommissionEmployee
```

- A way out is to declare superclass instance variables as protected so that subclasses could access them. Inheriting protected instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a *set* or *get* method call. But a subclass object can assign an invalid value to the variable, possibly leaving the object in an inconsistent state.
- Another problem with using protected instance variables is that subclass methods are more likely to be written so that they depend on the superclass's data implementation. In practice, subclasses should depend only on the superclass services (i.e., non-private methods) and not on the superclass data implementation.
- With protected instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. For example, if

for some reason we were to change the names of instance variables firstName and lastName to first and last, then we would have to do so for all occurrences in which a subclass directly references superclass instance variables firstName and lastName. In such a case, the software is said to be **fragile** or **brittle**, because a small change in the superclass can “break” subclass implementation. You should be able to change the superclass implementation while still providing the same services to the subclasses.

- A third problem is that a class’s protected members are visible to all classes in the same package as the class containing the protected members—this is not always desirable.
- In most cases, however, it’s better to use **private** instance variables, build **public** set/get methods to assess them -this is to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

// update on super class

```
public class CommissionEmployee
{
private String firstName;
private String lastName;
private String socialSecurityNumber;
private double grossSales; // gross weekly sales
private double commissionRate;
// write get methods for the instance variables
public double getCommissionRate()
{
return commissionRate;
}
public double getGrossSales()
{
return grossSales;
}
public double earnings()
{
return  getCommissionRate()*   getGrossSales();
}
@Override // indicates that this method overrides a superclass method
 public String toString()
 {
  return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
  "commission employee", , , getFirstName() getLastName()
  "social security number", , getSocialSecurityNumber()
  "gross sales", , getGrossSales()
  "commission rate",getCommissionRate() );
  } // end method toString
  } // end class
// Modification on the subclass
public class BasePlusCommissionEmployee extends CommissionEmployee
 {
@Override // indicates that this method overrides a superclass method
 public double earnings()
 {
return getBaseSalary() + super.earnings();
} //
// return String representation of BasePlusCommissionEmployee
@Override // indicates that this method overrides a superclass method
 public String toString()
 {
   return String.format( "%s %s\n%s: %.2f", "base-salaried", super.toString(), "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

- Note the syntax used to invoke an overridden superclass method from a subclass—place the keyword super and a dot (.) separator before the superclass method name. This method invocation is a good software engineering practice—if a method performs all or some of the actions needed by another method, call that method rather than duplicate its code. If we did not use “super.” then BasePlusCommissionEmployee’s earnings method would *call itself* rather than the superclass version. This would result in a phenomenon called *infinite recursion*, which would eventually cause the method-call stack to overflow—a fatal runtime error.

- When a program creates a BasePlusCommissionEmployee object, its constructor is called. That constructor calls CommissionEmployee's constructor which in turn calls Object's constructor. Class Object's constructor has an empty body, so it immediately returns control to CommissionEmployee's constructor, which then initializes the CommissionEmployee private instance variables that are part of the Base-PlusCommissionEmployee object. When CommissionEmployee's constructor completes execution, it returns control to BasePlusCommissionEmployee's constructor, which initializes the BasePlusCommissionEmployee object's baseSalary.

8.6 Software Engineering Practice with Inheritance

When you extend a class, the new class inherits the superclass's members—though the private superclass members are *hidden* in the new class. You can *customize* the new class to meet your needs by *including additional members* and by *overriding* superclass members. Doing this does not require the subclass programmer to change (or even have access to) the superclass's source code. Java simply requires access to the superclass's .class file so it can compile and execute any program that uses or extends the superclass. This powerful capability is attractive to independent software vendors (ISVs), who can develop proprietary classes for sale or license and make them available to users in bytecode format. Users then can derive new classes from these library classes rapidly and without accessing the ISVs' proprietary source code.
People experienced with such projects say that effective software reuse improves the software-development process and Object-oriented programming facilitates software reuse.

Task 8.1

a) Draw a Unified Modelling Language (UML) diagram of a **Rectangle** class showing the following class members: length, breadth, constructor and area.
b) Define RectangleX class as a child class of Rectangle. Create a Rectangle object and a RectangleX object in the main method and output perimeter properties of the two objects created. Override the RectangleX perimeter() method using the following formula:

perimeter = length + xlength + breadth.

Also, output the perimeter property of the overridden method.

Task 8.2

A ***MountainBike*** is-a ***Bicycle***, such that ***Cadence***, ***Speed*** and ***Gear*** are properties of Bicycle. ***MountainBike*** has one specific attribute known as ***seatHeight***. MountainBike inherits all the fields and methods of Bicycle and adds the field ***seatHeight*** and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands)

(i) Draw the UML(Unified Modeling Language) class diagram showing the inheritance relationship
(ii) Implement the ***Bicycle*** class showcasing its properties, constructor method and the method to increase speed (note the increment value must be passed to this method).
(iii) Implement its subclass showcasing its property, its constructor and the ***seatHeight*** method.

Task 8.3

The following behavior in inheritance is referred to as:

```
class Parent {
    public void example (int x) {...}

}
```

```
class Child extends Parent {

        public void example (int x) {...}

}
```

Task 8.4

Differentiate between overloading and overriding:

Solution of Task 8.4

Overloading: Overloading is static binding. Occurs when several methods have same names with:

i. Different method signature and different number or type of parameters.
ii. Same method signature but different number of parameters.
iii. Same method signature and same number of parameters but of different type

Overriding: This is dynamic binding. Overriding is the same method names with same arguments and return types associate with the class and its child class. This is dynamic or run time polymorphism in which call to an overridden function is resolved during run time, not at the compile time. It means having two or more methods with the same name, same signature but with different implementations

BIBLIOGRAPHY AND REFERENCES

Bravaco, R. and Simonson, S. (2010). Java Programming from the Ground Up. McGraw-Hill international Edition: Singapore

Deitel, P. & Deitel, H. (2012). ***Java How To Program (***9th International Edition). New Jersey: Pearson Education, Inc.

Dennis, A., Wixom, B. H., & Roth, R. M. (2012). Systems analysis and design, (5th ed.). USA: John Wiley & Sons, Inc.

Dennis, A., Wixom, B. H., & Tegarden, D. (2009). Systems analysis and design with UML version 2.0: An Object-Oriented Approach, (3rd ed.). USA: John Wiley & Sons, Inc.

Herbert Schildt (2011). ***Java A Beginner's Guide*** (Fifth Edition). New York: McGraw-Hill/Osborne

Martin Fowler, (2004), UML Distilled, Third Edition, Addison Wesley

Okolie, S. O., Ayankoya F. Y., Adesegun O. A. & Nzewata J. U. Java Fundamentals, Nigeria: Franco-Ola Printers.

Quentin Charatan and Aaron Kans (2019). ***Java in Two Semesters*** (4th Edition). London : Mc Graw-Hill Higher Education.