NUEVA ECIJA UNIVERSITY OF SCIENCE AND TECHNOLOGY
COLLEGE OF INFORMATION AND COMMUNICATINS TECHNOLOGY

LEARNING MODULE

INTEGRATIVE PROGRAMMING AND TECHNOLOGIES

INTEGRATIVE PROGRAMMING AND TECHNOLOGIES

TABLE OF CONTENTS

INTEGRATIVE PROGRAMMING AND TECHNOLOGIES

FOREWORD

This course covers an overview of integrative programming as related to applications and systems. It tackles the concept of Java Programming essential for integrative programming and technologies. File input and output is also covered in this learning module to grasp relevant understanding before integrating it to programs utilizing graphical user interfaces such as Java Swing, Java AWT, and JavaFX. The Object-Oriented Application Development concepts are also covered in this learning module to provide you the information required when integrating programming concepts to different technologies. After successfully understanding OO Concepts, Unified Modeling Language will guide you to conceptualizing and developing your application. Also, an introduction to unified process is covered in this module to allow you to deepen your knowledge and understanding about software and application development.

At the end of this learning module, you are expected to exhibit understanding and competencies to advance Java Programming. Utilizing the language, you are expected to produce output that would apply the integrative programming and technologies concepts.

The Authors

Rachel T. Alegado
Leonylyn P. Bensi
Michael E. Bensi
Marcelino C. Collado, Jr.
Dr. Arnold P. Dela Cruz
Allen Paul Esteban
Marvin DG. Garcia
Cris Norman P. Olipas
Christian Peña

LESSON 1
JAVA FUNDAMENTALS

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. define what a program is and identify the different types of programming languages;
2. identify what is Java and understand its characteristics;
3. review how to write a Java program and understand its structure; and
4. show appreciation on the anatomy of programs written in Java.

READY, GET SET, DISCUSS!

What is a Program?

Programs are instructions or step-by-step procedures that programmers write using a programming language to command the computer about a series of activities that must be done. Programs are essential set of instructions allowing computers to become useful in performing its intended functions. When computers are not operated and instructed by set of program instructions, such technology is just a mere machine.

Programming Languages

Programming languages are written in different forms, in different formats, and in different syntax. Programming languages vary depending on how it was developed to make a computer run a set of instructions. They are classified into machine language, assembly language, and high-level language.

a. Machine languages are primitive commands assembled where binary codes are essential components in writing the program in order for the computer to understand it. Programs written in machine language are complex to modify, thus programmers without extensive exposure and training in the language might find machine languages difficult to recognize, identify, and understand.

For example: 1101101010011010

b. **Assembly languages** were developed to provide an easier way to write a program compared to machine languages. It was made to give additional means for programmers to increase productivity in writing programs. Compared to machine language, assembly languages used a program called assembler to convert the written program into machine code.

For example: ADDF3 R1, R2, R3

c. **High-level languages** are programming languages that are close to the English language. This means that programs written using high-level languages uses "English-like" words to represent executable commands converted into binary codes which the computer can understand through a compiler. High-level programming languages are simple, straightforward, and uncomplicated to write. Today, many high-level programming languages are available to provide programmers with a wide variety of options to choose from to efficiently and effective write powerful, sophisticated and relevant programs.

For example, you might want to write an instruction in high-level language that tells a program to compute the Area of a Circle with a Radius of 3.

```
Area = 3 * 3 * 3.1415;
```

ACTIVITY

1. Look for five definitions of a "Program" on the internet. Based on the five definitions that you will find, come-up with your own definition of what a program is based on your understanding. Include the sources of your definitions using the APA referencing format.
2. Search the internet and fill-up the matrix/table below.

Programming Language	Advantages	Disadvantages	Key Features
1. COBOL			
2. FORTRAN			
3. BASIC			

4. Pascal			
5. Ada			
6. C			
7. Visual Basic			
8. Delphi			
9. C++			
10. Java			

3. After completing the table/matrix, write a 300-word insight/reflection about comparing the different high-level programming languages.

SOURCE CODE COMPILING

The programs written using a high-level programming language is commonly known as a *source program*. A source program is translated to a machine language to become an object program using a compiler. These object programs are linked and executed on the machine. It is important to note that programs written using high-level programming language must go through a compiler to ensure its correctness and readiness before it can be converted as object programs.

JAVA SOURCE CODE COMPILING

Java was developed to execute object programs to any type of operating systems and considered as a high-level programming language. This means that programs written using Java regardless of the platform used can be executed provided a Java Virtual Machine (JVM) is present. Java Virtual Machine is a type of software used to interpret Java byte codes. When you write a Java program and compile it using a compiler, a source is produced. Source is a unique type of object code necessary for JVM when interpreting Java programs.

Why Java?

Today, many applications and programs are written in Java. It is a popular programming language used to write different kinds of applications. As a result, the demand for Java developers continue to increase in today's digital era. Since Java becomes a universal if not, common programming language, it is straightforward, uncomplicated to understand, and analyze, allowing more and more opportunities come to in for those who engage and devote time and effort in learning the language. Java enables user to build up and install applications on the internet for computer servers, desktop computers, and smart phones. The advent of internet-of-things allows more and more Java applications to be developed and utilized.

Characteristics of Java

1. ***Java is Simple***

   One of the characteristics of Java is Simplicity. It is partially based on the C++ programming language but more simplified and improved. One of the wrong notions about Java is that it is the same with C++ but by understanding the language, one would identify that Java has more functionality and fewer negative aspects. Since Java is a high-level programming language, unlike machine and assembly language, Java allows programmers to write and execute programs in a simpler and easier way.

2. ***Java is Object-oriented***

   Programming languages can be procedural, object-oriented, or event-driven language. The programming language is an object-oriented employing the concepts of encapsulation, inheritance, and polymorphism to achieve the purpose of a flexible, clear, modularize, and reusable programs. One of the common problems and challenges in software application development relates on how to reuse code, thus Java provides an easier means to achieve it by being an object-oriented driven language.

3. ***Java is distributed***

   Java programming language can be used to write programs to be deployed and utilized using a network. This language conforms to the requirements needed for distributed computing where a number of computers are connected and working together via a network. Since Java was designed to comply with the requirements needed for distributed computing, being capable to be distributed is one of the considered characteristics of the language.

4. ***Java is Interpreted***

   When writing a Java program, an interpreter is essential to compile a source code. A Java interpreter is a vital element of JVM. When programs are compiled in the JVM, a code is produced called byte code. These byte codes are machine-independent and can only run on machines that have JVM installed in it.

5. ***Java is Robust***

   Robustness is one of the characteristics of Java. Robustness refers to the capacity of the language to withstand difficult conditions. This means that through its compilers, Java has the ability to easily detect problems and errors that could possibly show up before and during the execution time. The runtime exception handling feature of the langue is one of the evident features of the language that deals with robustness.

6. ***Java is Secure***

   The Java programming language provides and implements security measures and ways to protect the system against harm caused by stray programs. This means that the programs can utilize the available security features of the language to ensure it is free from harmful programs which might affect the over-all performance of the program.

7. ***Java is Portable***

   The ability of the Java program to be executed in different platforms provided there is an installed JVM makes it architecture neutral. By being architecture neutral, Java programs become portable. Portability is the characteristic that pertains to the capability of a program to be run in any platform, and in any device.

8. ***Java's Performance***

   Since Java is secured, portable, robust, and easy to use, the quality of the output produced using the programming language is better, thus the performance is greatly affected.

9. ***Java is Multithreaded***

   Multithreading in Java is not specific to the Operating System (OS) unlike other programming languages that depend on the type of OS to enable multithreading. Since multithreading is integrated in Java, calling procedures and implementing them is easier and more accessible.

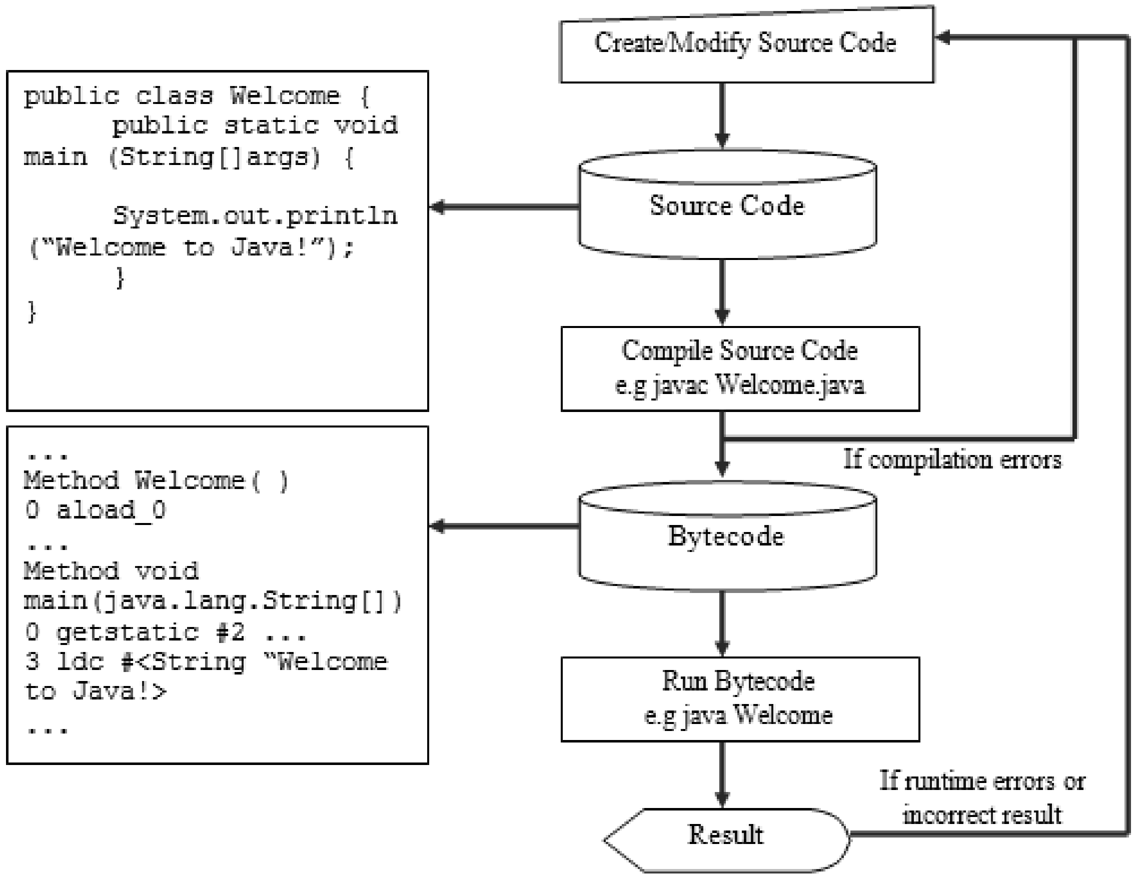
Java IDE Tools

a. Borland JBuilders
b. Oracle JDevelopers
c. JCreator
d. NetBeans
e. Eclipse

A SIMPLE JAVA PROGRAM

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

PROCESS TO CREATE, COMPILE, AND RUN A JAVA PROGRAM


Create/Modify Source Code
Source Code
Compile Source Code e.g javac Welcome.java
If compilation errors
Bytecode
Run Bytecode e.g java Welcome
If runtime errors or incorrect result
Result

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

     System.out.println
("Welcome to Java!");
     }
}
```

```
...
Method Welcome( )
0 aload_0
...
Method void
main(java.lang.String[])
0 getstatic #2 ...
3 ldc #<String "Welcome
to Java!>
...
```

Figure 1-1. Representations on how to create, compile, and run a Java program

ELEMENTS OF A JAVA PROGRAM

a. Comments

Comments are use in a Java program to indicate programmer-defined instructions and remarks. These are useful to allow other programmers reading the program to fully understand how a specific block of code works. Comments follows two slashes (//) in a line or enclosed between /* and */ to indicate multiple line of comments

b. Reserved Words

In writing Java programs, programmers must be mindful of the fact that the language has reserved words or keywords. These keywords contain built-in specific functions and capabilities in which programmers must not incorporate when not necessary. Oftentimes, for beginners, keywords are usually added in a program without knowing how it is use;

thus, it is important for new java programmer to be familiar with all the Java reserved keywords and their specific functions.

c. ***Modifiers***

Modifiers in java are certain keywords that specify the property of the data, the methods, the classes, and how they can be used in the java program. Examples of modifiers include public and static.

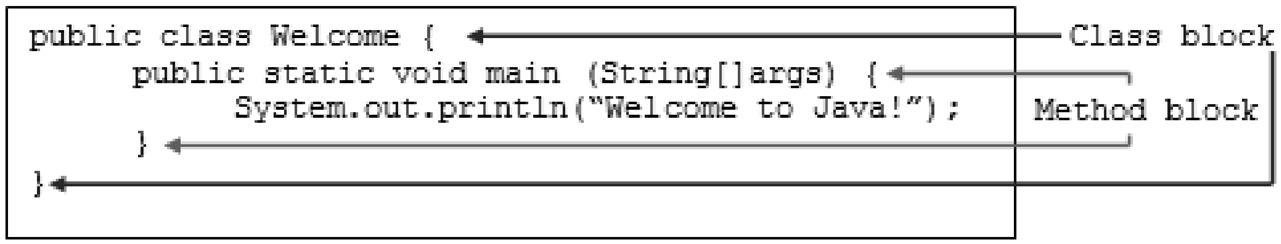
d. ***Statements***

To execute sequence of action/s, a statement is important. The statement System.out.println ("Welcome to Java!") is used to display the greetings "Welcome to Java!". When writing a java program, you need to remember that statements must end with a semicolon.

e. ***Blocks***

Curly braces { } are used to form a block in order to group line of code in a program.

Example:



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

f. ***Classes***

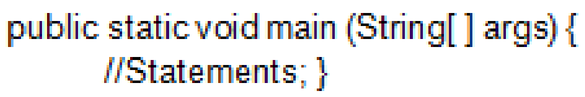
One of the essential Java constructs is the *class.* In order to successfully run a Java program, you must be able to properly understand the use of classes and how to write and use them properly. When you write a program, you can opt to write one or more classes.

g. ***Method***

A method is a collection of different statements or lines that specifically invoke to perform a series of operations in order to be able to display a message or an output on the console. Methods are typically used to invoke statements with string arguments. For example, System.out.println ("Welcome to Java") has arguments "Welcome to Java!".

h. ***The main Method***

The main method is invoked by the Java interpreter to execute the application. It looks like this:



```
public static void main (String[ ] args) {
        //Statements; }
```

Documentation

In every computer program, it is essential to write a well-written documentation. This documentation contains all the relevant information about the program, including the purpose, who wrote it, when was it written, and how to use it. Also, documentation may contain a brief or detailed explanation of the specific sections of the program code which might help a programmer understand the program months or years later when it requires modification. A program without a well-written documentation may result in difficulties and challenges in the future when the time comes that some modification is needed on it.

ACTIVITY

1. Write a java program to solve the machine problem.

Machine Problem

You are an owner of a meat shop selling beef, pork, and chicken. Each type of meat has a corresponding price per kilo and percentage value discount. Write a program that would ask the user for the

1. Price per kilo of Beef;
2. Percentage value for Beef
3. Price per kilo of Pork
4. Percentage value for Pork
5. Price per kilo of Chicken
6. Percentage value for Chicken

The program must be able to calculate and produce the result for each kilo of meat against their corresponding percentage value as shown in the sample output:

Sample Output:

```
Beef: 250
Percentage value: 10
Pork: 220
Percentage value: 15
Chicken: 140
Percentage value: 5

Final Price for Beef: 225
Final Price for Pork: 187
Final Price for Chicken: 133
```

2. Based on your java program answering the machine problem for question 1, take a screen shot of the source code and categorize which part adheres/complies to the basic anatomy of a java program. Note: You can put an arrow to easily label the following:
    a. Comments
    b. Reserved Words
    c. Modifiers
    d. Statements
    e. Blocks
    f. Classes
    g. Method
    h. The *main* Method

Pseudocode

A *pseudocode* which literally means a false code, is a text that closely similar to the syntax of a programming language. However, a pseudocode will not compile because it does not conform to a well-defined programming syntax. However, if you are planning to write a program, it will be beneficial for you to start writing a pseudocode to understand the entirety of the program without writing the actual syntax. Also, by writing pseudocode, you can easily construct the necessary algorithm needed to successfully write the program.

Example:

```
Start
Declare A, B, and C as integers
Get the value of A
Get the value of B
Let C = A + B
Print A, B, and C
Stop
```

If you are planning to write your program in Java, you might decide to write the pseudocode in a more Java-like type

Example:

```
class AddNumbers
    main method
        integers a, b, c
        read a
        read b
        c = a + b
        print c
    end main
end class
```

The form of pseudocode you use depends on your personal preference or the guidelines of a particular group you work on.

Flowcharts

Aside from writing pseudocode, developing a flowchart is an effective way to plan a program. Flowchart is a pictorial means to describe the basic structure of a program. Each symbol in a flowchart represents a specific purpose in the program. It is advisable that after writing your pseudocode, constructing the flowchart based on the pseudocode would help in visualizing how the actual program would work.

Example:

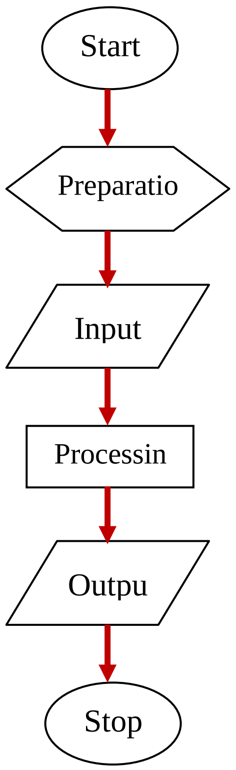


Figure 1-2. Sample Flowchart

QUIZ IS IT!

Instruction: Choose and identify the best answer from the choices given below.

1. Which among the following is NOT an element of a Java program?

a. Literals	c. Reserved Words
b. Comments	d. Blocks

2. This/These java element/s is/are used to provide programmer-defined instructions and remarks and is/are useful for other programmers so that the program can be easily understood.

a. Blocks c. Comments
b. Modifiers d. Method

3. Programs can be classified into procedural, ________________, and event-driven.
   a. Object-Orienting c. Oriented-Object
   b. Object-Oriented d. Object Type
4. Which among the following are used in creating blocks in java?
   a. [ ] c. / /
   b. { } d. ( )
5. Which part of the java program is invoked first when the program executes?
   a. Private Method c. Public Method
   b. Main Method d. Import Java
6. Which among the following is NOT a characteristic of Java?
   a. Java is Difficult c. Java is Simple
   b. Java is Robust d. Java is Distributed
7. Which among the following is NOT a high-level programming language?
   a. Foxtrot c. C++
   b. Java d. Fortran
8. Programs are set of instructions. Programs are not essential to a computer to run.
   a. Both Statement is True
   b. Both Statement is False
   c. First Statement is True, Second Statement is False
   d. First Statement is False, Second Statement is True
9. In the choices below, which is not a symbol in a flowchart?
   a. Square c. Parallelogram
   b. Diamond d. Heart
10. Pseudocode means ____________?
    a. False Hope c. False Code
    b. True Hope d. True Code

Enumerate what is being asked

1. Give at least five high level programming languages
2. Give at least five characteristics of Java

Short Essay

1. Draw an insight on the use of flowchart when writing a program. Explain its perceived importance to when writing your Java program.

LESSON 2
DATA IN JAVA PROGRAM

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. show appreciation on the use of literals, variables, identifiers, data types, assignment operator, type casting, and constants, and apply them in writing Java programs;
2. apply and use different arithmetic operators and arithmetic expressions in writing Java programs; and
3. apply String and its properties and features in writing Java Programs.

READY. GET SET. DISCUSS!

Literals

A *literal* is a number, single character, or string (a list of several characters) that is coded directly into a program.

Example: In a Java Program, a numeric literal may appear as 7 or 3.1415, and character literals are written in single quotes, such as 'a' or '$'. String literals are any character or string of characters in between the double quotation mark like "Hello World"

There will be no problem in assigning an integer literal into a variable as long as the size of the literal does not exceed to the capacity of the variable type hence it will show an error during program compilation.

Example: An error will be show in the statement byte = 1000 because the value 1000 is too large for a byte type.

A literal could be a number, text, or other value that directly appears in the program.

Example: 48, 10000 and 3.5 are literal values in the following statements:

```
int i = 48;
long x = 10000;
double d = 3.5;
```

Variables

A *variable* is a specific location or address in computer memory where they program can store data. The contents of a variable can change during program execution, hence the term variable. When you create or declare a variable in a program, you assign the memory address a name and specify the type of data the variable can hold. Two general data types in Java are numeric (for numbers) and string (for text).

Example:

```
int x;                    // x is declared as integer
double radius;            // radius is declared as double
char a;                   // a is declared as a character
```

Identifiers

In Java, an i*dentifier* can be a name given to variables, classes, methods that consist mostly of letters, numbers, dollar signs ($) and underscores (_).

The correct way of naming an identifier is by starting it with either a letter (a-z), a dollar sign (4), or a underscore (_) It is invalid, however, to use the following as the start of an identifier:

a. A digit.
b. A reserved word.
c. Values of true, false, or null.

An identifier can be of any length

Data Types

Type	Size
Boolean	1 bit
Char	16 bits or 2 bytes
Byte	8 bits or 1 byte
Short	16 bits or 2 bytes
Int	32 bits or 4 bytes
Long	64 bits or 8 bytes
Float	32 bits or 4 bytes
Double	64 bits or 8 bytes

Assigning Statements

Literals can be assigned as value of a variable. A *variable* can also be assigned to another variable. In Java, equal sign (=) is the assignment operator, a statement of action, not an algebraic statement of equality.

Example:

```
x = 1;          // Assign 1 to the variable x;
radius = 1.0;   // Assign 1.0 to radius;
a = 'A' ;       // Assign 'A' to the variable a;
```

Type Casting

Sometimes, you have a variable of one data type but you need to explicitly convert it to a different data type. To do this, Java requires a process called explicit type casting, which means casting or assigning a data type to a value.

Example:

```
double aDouble = 7.9;
int anInteger;
anInteger = (int) aDouble; //returns 7
```

If you do not use type cast operator, int, Java will produce a “possible loss of precision” error because you are trying to store more precise data type as a less precise data type. The operator (int) tells Java to truncate (cut off the decimal part of) the value in aDouble, assigning the result integer 7 to anInteger.

Named Constant

Named constant is a type of an identifier whose value is permanent. To create a named constant in Java, use constants “final” and “static final”. The final constant is for single instance while the static final is for all class instance.

Example:

```
final double TAX_RATE = .10;
final double PI_VAL = 3.14159;
final int SIZE_VAL = 3;
```

Java convention program for coding named constant is to use uppercase characters to underscore for separating individual words. This convention helps programmers remember which variables in a program are constants.

ARITHMETIC OPERATORS AND EXPRESSIONS

Arithmetic Operators

Arithmetic Operators in Java such as +, -, or / perform mathematical operation on one or two operands (numeric values). An operator that acts on a single operand, as in the case of -5 is called *unary operator*. The two unary operators in Java are plus (+) and minus (-), which simply assign a positive and negative sign (implying directions) to a number. An operator that acts on two operands such as with 1 + 2 is called a binary operator. The binary operators in Java are +, -, *, / and %.

Arithmetic Expressions

Arithmetic Expressions is a group of numeric literals (such as 7 or 2.75) and/or numeric variables (such as totalCost or temperature) combined with arithmetic operators and possible, parenthesis.

If all operands in an expression are integers, the expression is called *integral expressions* and produces an *integer result*.

If all the operands are floating-points, the expression is called *floating-point expression* and produces a *floating-point result.*

If an expression contains both integers and floating-points, it is called a *mixed-expression* (such as 5 + 2.5) and will result in a *floating-point*. In mixed expression, Java performs what is called *implicit type casting* - automatically converting the result of the expression to the most precise data type.

Unary Negation

The *unary negation operator* or *minus* (-) transforms the literal or variable that follows to its opposite sign, reversing the direction of the number

Example: oppositeValue = -value;

If the variable value holds the number 3, then the oppositeValue is assigned -3. If value holds the number -7, then opposieValue is assigned 7.

Addition and Subtraction

The + and – binary operators are used to add or subtract two numbers. The following contain examples of these operators:

```
totalWeight = myWeight + yourWeight;
myWeight = myWeight – 10;
```

Multiplication and Division

The *asterisk* (*) is the Java multiplication operators and the backslash (/) in Java is the operator for division.

```
int numberOfCupCakesA = 10, numberOfCupCakesB = 5;
double averageOfCupCakes;
averageOfCupCakes = (numberOfCupCakesA + numberOfCupCakesB) / 2.0;
// returns 7.5 because of implicit type casting
```

Modulus Operator

The *modulus operator* (%) also called the *remainder operator*, only applies to integers. It returns the remainder in a division operation.

Example:

```
int numberOfCupCakes = 31, numberOfStudents = 10, cupcakesPerStudent;
cupcakesPerStudent = numberOfCupCakes / numberOfStudents; //returns 3
int cupcakesLeftOver = numberOfCupCakes % numberOfStudents; //returns 1
```

Operator Precedence

The precedence that Java gives arithmetic operators is the same that you would apply when evaluating algebraic expressions. Unary plus (+) and minus (-) have the highest precedence, then the *, / and percent sign (%) are the next lower point, and followed by the + and – binary operators.

Assignment Operators Shortcut

Operator	Example	Equivalent
+=	i + = 1	i = i + 1
-=	f -= 1.0	f = f – 1.0
*=	x *= 1	x = x * 1
/=	y /= 1	y = y / 1
%=	a %= 1	a = a % 1

Increment and Decrement Operators

Operators	Name	Description
Prefix ++ (++a)	Preincrement	The value will increment first then the incremented value will be used in the expression.
Postfix ++ (a++)	Postincrement	The value will be returned first then it will be incremented.
Prefix – (--a)	Predecrement	The value will decrement first then the incremented value will be used in the expression.
Postfix – (a--)	Postdecrement	The value will be returned first then it will be decremented.

Conversion Rules

Java uses the following conversion rules in performing binary operation between two operands:

a. If the type of one operand is double, the other will be changed into double type.
b. Then, if the type of one operand is float, the other operand will be converted into float.
c. Then, if the one of one operand is long, the other operand will be converted into long.
d. Then, both of the operands will be converted into an int type.

Character Escaping

To escape characters in Java, use a special symbol called *backslash* (\). Backslash together with the character that needs to be "escaped" is referred to as *control sequence*. Examples are \t for tab, \n for linefeed, \\ for backslash, \' for single quote and \" double quote.

The String type

The *char type* represents a single character. To represent a series of characters, use *String type.*

Example:

String message = “Welcome to Java”;

String Concatenation

Concatenation is used to “connect” strings using “+” operator

Example:

```
String message = “Welcome” + “to” + “Java”; //three strings are put together
String s = “Fundamental” + 1; //s becomes Fundamental1
String s1 = “Vitamin” + ‘B’; //s1 becomes VitaminB
```

Converting Strings to Integers

Java parseInt method can convert a string into an integer as per the following example:

```
int inString = “123”;
int intValue = Integer.parseInt (intString);
```

Programming Errors

1. Syntax Errors – An error detected by the compiler usually wrong use of syntax.
2. Runtime Errors – A type of error that causes the program to abort.
3. Logic Errors – produces different result from an intended output.

Debugging

Debugging is a process for the identification and correction of errors in a program. Different approaches can be used in dealing with errors in a program in Java. Typical approach can be done is by doing a hand-trace (or reading the entire program) or by using insert print statements to show the running values of a variable or the flow of the program. But this approach is only advantageous to small and simple program. A debugger utility is the most appropriate in large and complex programs.

LABORATORY EXERCISE

1. Plan the Java Program by illustrating its flowchart. Write the Java Program based on the flowchart that will solve the problem below.

<u>Weight Problem</u>

Introduction:

One of the many activities of the World Health Organization is to report on the “healthiness” of the world. This is done by conducting random sampling of the population and categorizing the people according to their Body Mass Index (BMI). A person's BMI is obtained using the height in meters and weight in kilograms using the formula

$$BMI = \frac{weight}{height^2}$$

This is then categorized according to the following table:

BMI	Category
below 16.5	Severely Underweight
16.5 to below 18.5	Underweight
18.5 to below 25	Normal
25 to below 30	Overweight
30 to below 35	Obese Class I
35 to below 40	Obese Class II
40 and above	Obese Class III

Of course, this is by no means a perfect measure since this does not take into consideration other factors like body type and race.

Problem:

Write a program that will determine an individual's "healthiness" category.

Input:

Input shall be sets of data containing the height in meters and weight in kilograms.

Program shall terminate when a height of 0 is received.

Output:

For each input set, the program shall output the computed BMI and it's corresponding category.

Input Validation:

No input validation is necessary.

Sample Run:

```
1.67 90↵
BMI = 32.27 (obese class I)
1.67 80↵
BMI = 28.69 (overweight)
2.0 80↵
BMI = 20.00 (normal)
1.5 40↵
BMI = 17.78 (underweight)
1.7 40↵
BMI = 13.84 (severely underweight)
```

```
1.5 90↵
BMI = 40.00 (obese class III)
1.6 90↵
BMI = 35.16 (obese class II)
0↵
```

2. Please take note of the following terminology and identify it based in the written Java program (Screen shot the source code and put an arrow the terminology is located).
   a. Literals
   b. variables,
   c. identifiers,
   d. data types,
   e. assignment operator,
   f. type casting, and
   g. constants
   h. arithmetic operators
   i. arithmetic expressions
   j. String and its properties and features

LESSON 3
FILE INPUT AND OUTPUT

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. identify what is a stream and its use;
2. compare and contrast stream byte and stream character ; and
3. write a java program applying the concepts of Streams.

READY. GET SET. DISCUSS!

File Input and Output

In Java, we can read file data, and write data in files as well. The java.io package holds almost every class that need to do input and output (I/O) in Java. All of these streams represent source for input and a destination for output. The java.io stream package supports a lot of data like primitives, localized characters, object, etc.

Stream

Java I/O stream is the flow of data from which you can either read, or write to. It is used to perform read and write file operations. In such tasks Java uses streams. Java I/O stream is also called *File Handling*, or *I/O File*. It can be found in java.io package.

Two (2) types of streams based on data:
• **Byte Stream**: used for the writing or reading byte data.
• **Character Stream**: used to write or read data about characters.

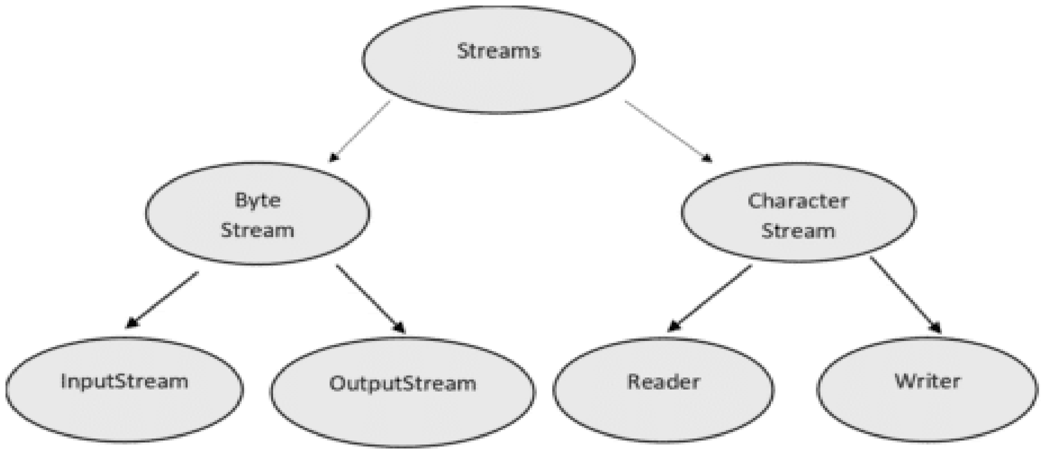


Figure 3-1. Hierarchy of Streams

Byte Streams

Byte Input Stream:

- These are used for reading byte data from different input devices.
- *InputStream* is an abstract class and all input byte are super class streams.

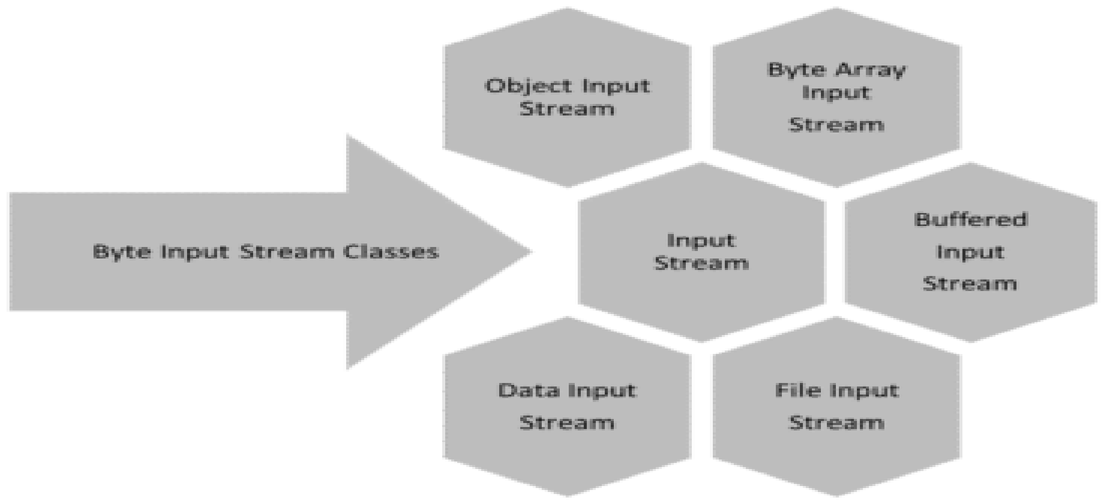


Figure 3-2. Byte Input Streams

Byte Output Stream:

- The *Output Stream* is an abstract class, and a superclass for all the byte output streams.
- Used to write data from bytes to different output devices.

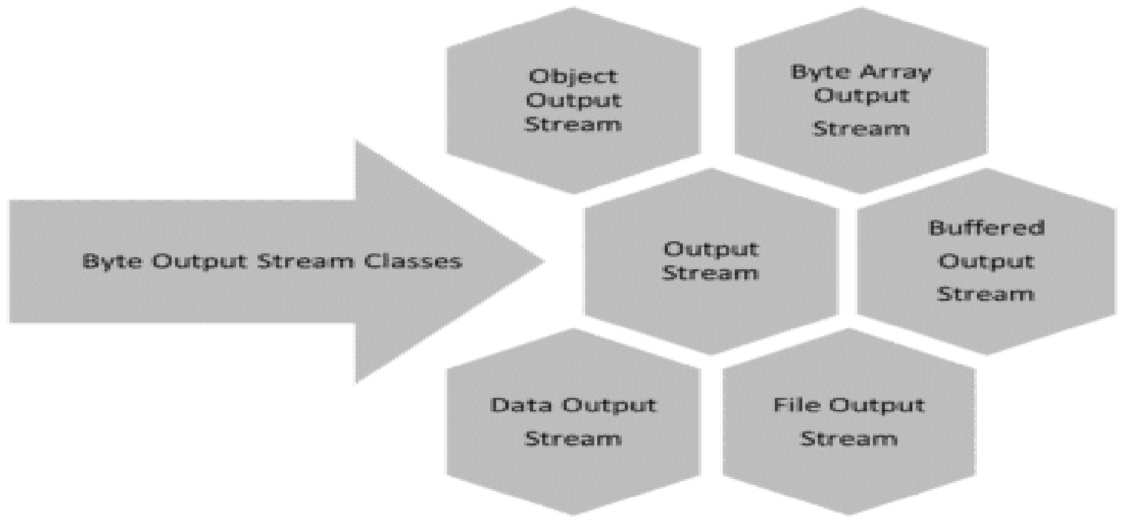
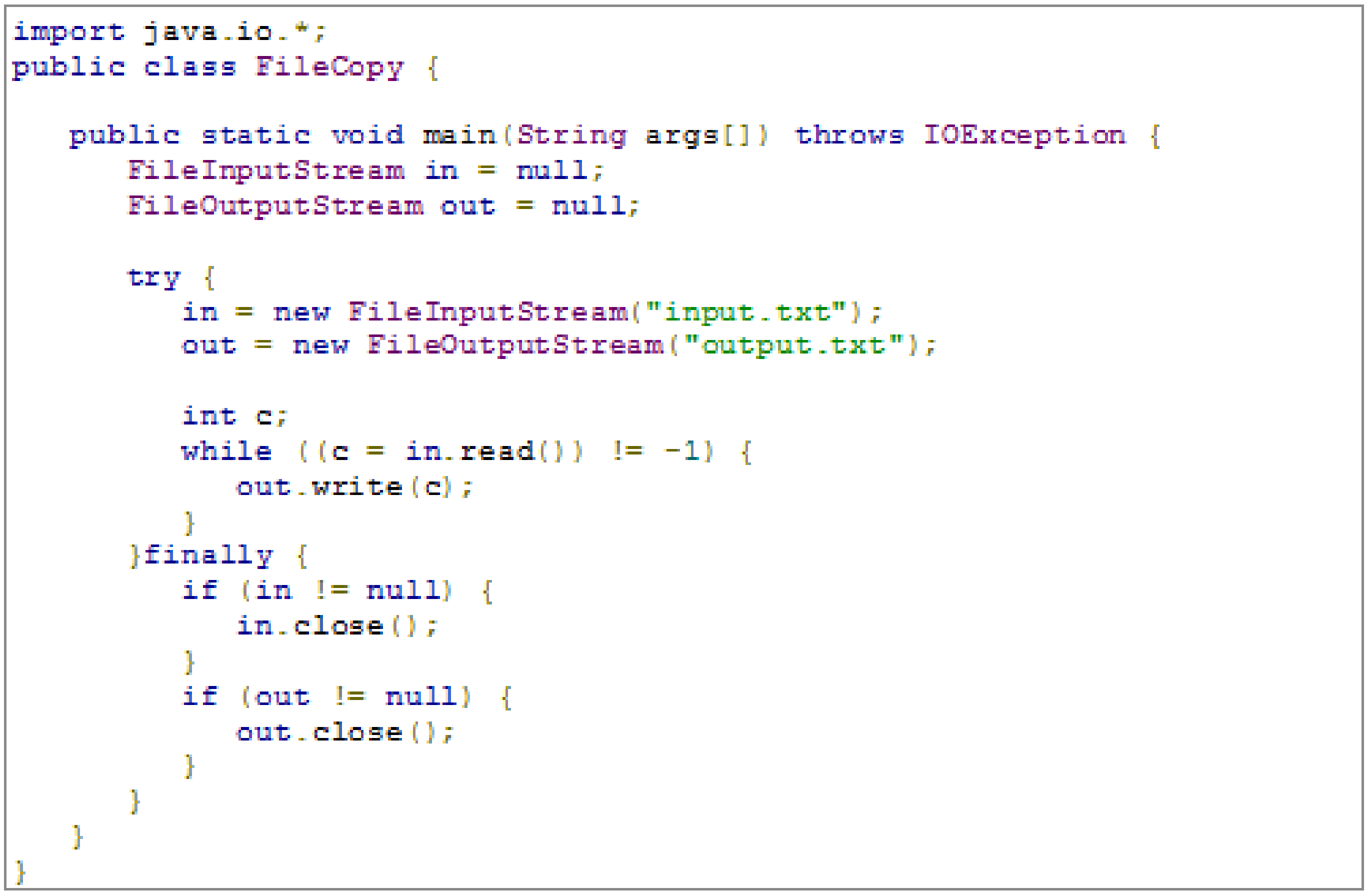


Figure 3-3. Byte Output Sreams

FileInputStream and FileOutputStream are the most frequently used classes. Example below "`FileCopy.Java`" uses these two classes to copy file input into file output.

Example. `FileCopy.Java`



```java
import java.io.*;
public class FileCopy {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have an input.txt file with the content below.

Test for file copy.

Next step, compile and execute the above example "`FileCopy.Java`", resulting in the construction of output.txt file with the similar content as the one we have in input.txt. So let us put the code above into the "FileCopy.java" file and do the following:

```
$javac FileCopy.java
$java FileCopy
```

Character Streams

Character Input Stream:

• These are used for reading the char data from different input devices.

• *Reader* is an abstract class, and a super class for all input streams of character.

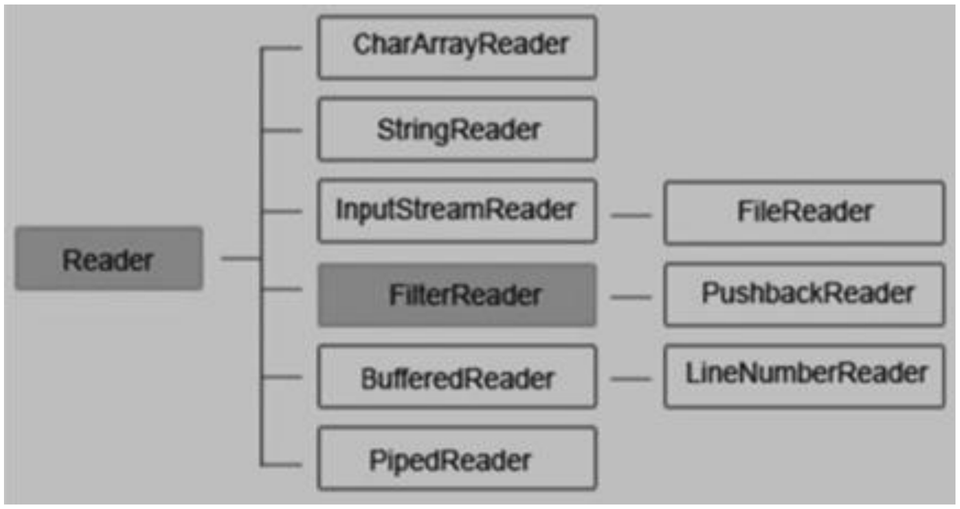


Figure 3-4. Input Streams

Character Output Stream:

- Using these to write the char data to different output devices.
- Writer is an abstract class and is the super class of all the character output streams.

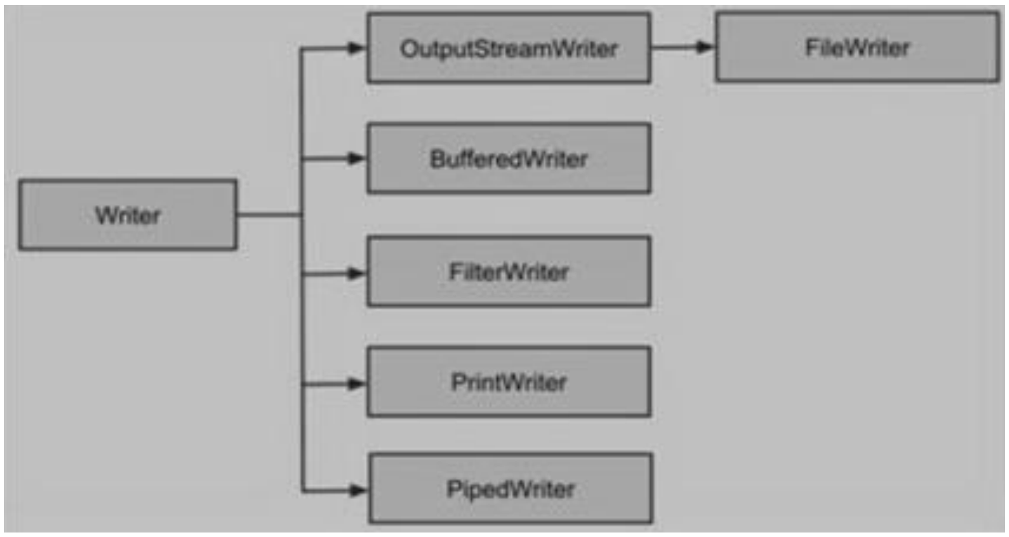
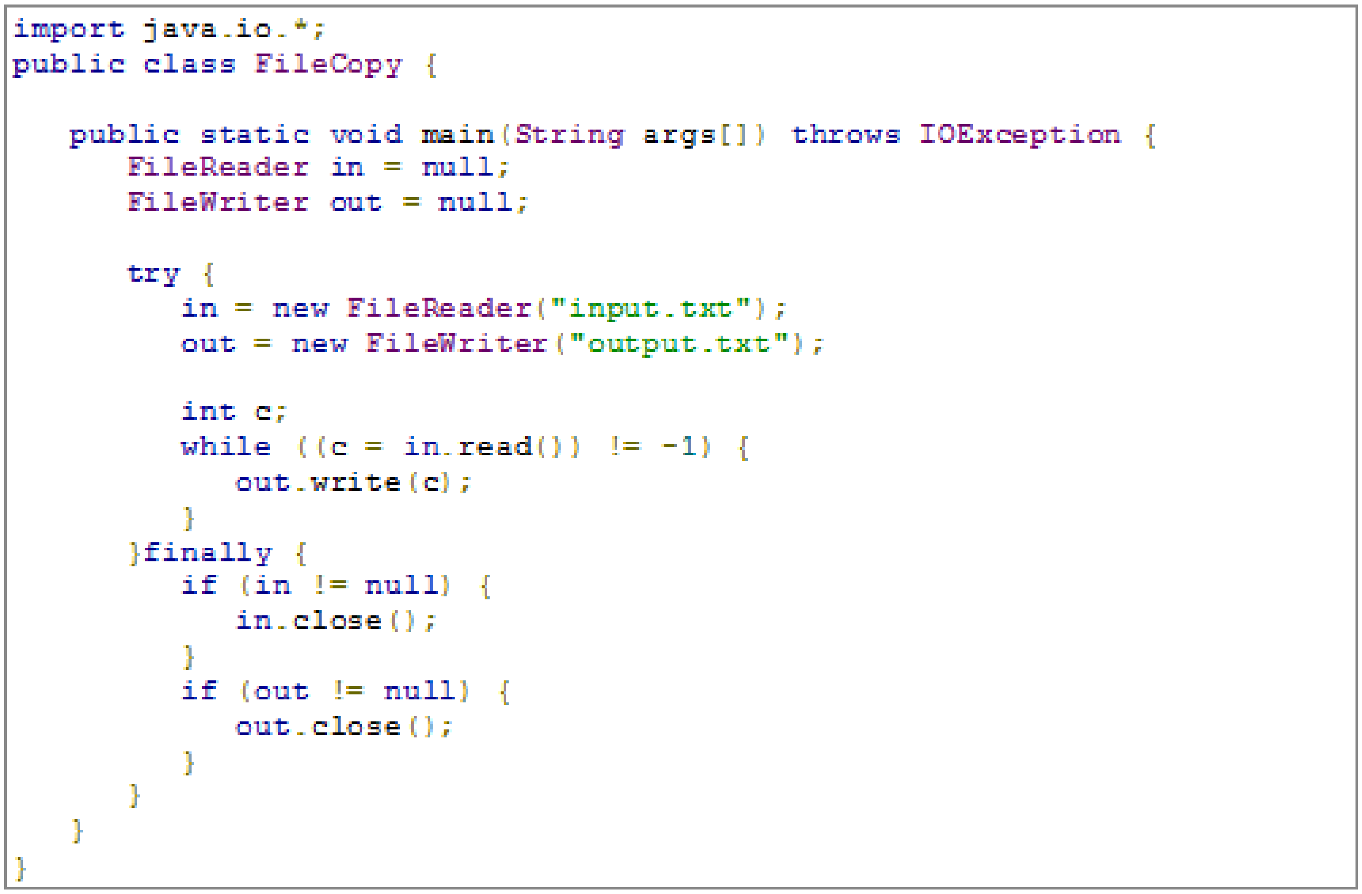


Figure 3-5. Output Streams

FileReader and *FileWriter* are the most usually used classes. Because FileReader uses FileInputStream internally, and FileWriter uses FileOutputStream, the key difference here is that FileReader reads two (2) bytes and FileWriter writes two (2) bytes at a time.

The above example can be written in two classes, which allows to copy a file input file into a file output.

Example. `FileCopy.java`



```
import java.io.*;
public class FileCopy {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Let us get a file called **"input.txt"** with the following details:

Test for file copy.

The next step, "FileCopy.java" program will be compiled and executed, which will result in making "output.txt" file with the similar information/data as we have in "input.txt". So let us put the following code into FileCopy.java" file and do the following:

```
$javac FileCopy.java
$java FileCopy
```

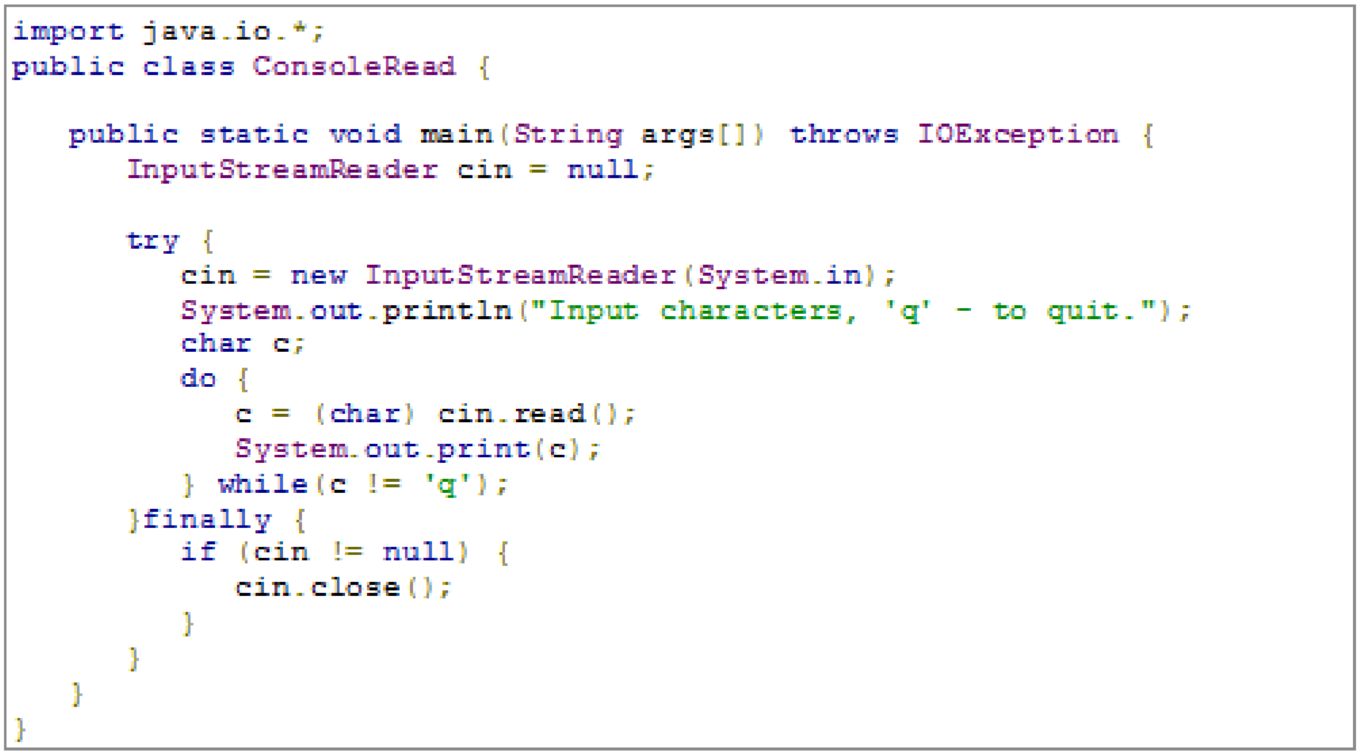
Standard Streams

The idea of standard I/O streams is a C library model that has been integrates into the environment of Java. There are three (3) standard streams namely *STDOUT, STDIN*, and *STDERR* tools. All of these are managed by the java.lang.System class.

- The **Standard Input** (for STDIN). It can be represented by System.in that used for program input, normally reads input entered by the user.
- The **Standard Output** (for STDOUT). It is the PrintStream class that used for program output, typically output/displays information to the user.
- The **Standard Error** (for STDERR). It is a PrintStream just like System.out that used to output/display error messages to the user.

A simple program follows that creates InputStreamReader to read the regular input stream before the user types a "q":

Example. `ConsoleRead.java`



```java
import java.io.*;
public class ConsoleRead {

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Input characters, 'q' - to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

The above program "**`ConsoleRead.java`**" file, compile and execute it as displayed in the program. The program keeps reading and displaying the equal character until you press the letter 'q' :

```
$javac ConsoleRead.java
$java ConsoleRead
Input characters  'q' – to quit
1
1
q
q
```

QUIZ IS IT!

1. Explain the use of streams when applied to a Java program.
2. Give the different types of Character and Byte Streams.
3. Write a Java program to modify the source code written in Example "**`ReadConsole.java`**" of this lesson. The program should ask for your *Student Number, Last Name, First Name, and Course*. When the program accepts a *null* input, the program should terminate.

LESSON 4
READING AND WRITING FILES

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. recognize the hierarchy of classes that deals with Input and Output Streams;
2. apply the methods used in FileInputStream and FileOutputStream; and
3. write a java program that applies FileInputStream and FileOutputStream

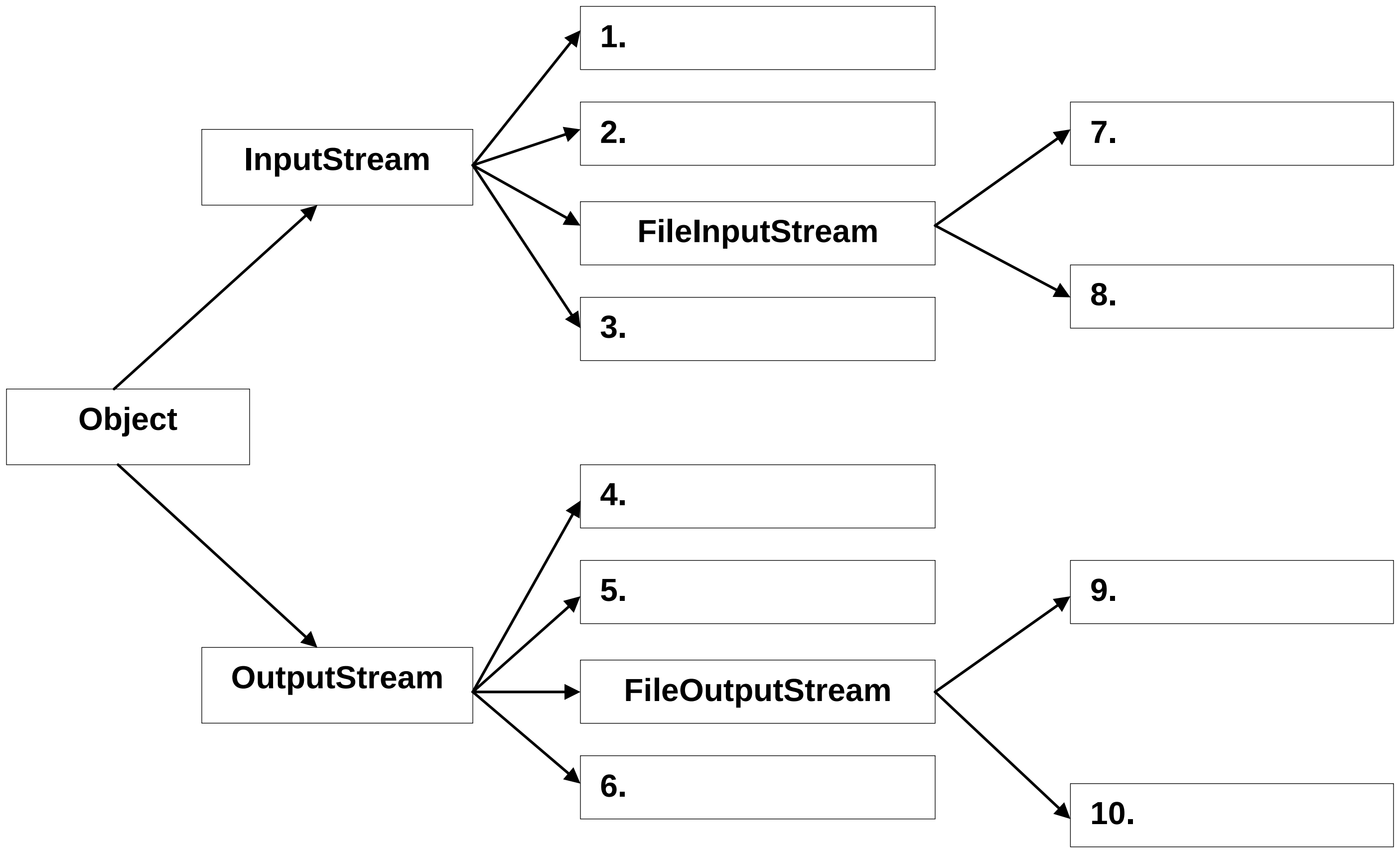
READY. GET SET. DISCUSS!

Reading and Writing Files

Stream is classified into **InputStream** and **OutputStream**. **InputStream** is for reading data from a source while writing data to an endpoint is an **OutputStream**.

ACTIVITY

As per discussed earlier about Input and Output Stream, complete the corresponding hierarchy of classes. You may research the internet to find the different classes for Input and Output Streams.



FileInputStream

This type of stream is meant to read contents of a file in a stream of bytes. To create an instance of an *FileInputStream,* there are many possible constructors that can be. One way is by having a **String** as parameter. This **String** contains the path where the file is located. The below code is an example:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Another way is to take a **File** object as parameter. The **File** object should point to the file to read. An example code is:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Reading data from a stream is possible using some of the following methods:

- **void close()**. To close a stream, this method is called.
- **protected void finalize.** This method is called *ensures* that the stream is closed and no reference is active in relation to the stream.
- **read(int r).** Returns an int value, next byte of data or -1.
- **read(byte [] r).** Will read from the input stream the length of bytes.
- **available().** Will return the available bytes that can be returned.

FileOutputStream

Creating and writing data into a file is possible in Java using FileOutputStream. Possible constructors associated with FileOutputStream that can be used is, one, take a String containing the path of a file to write to. The following code is an example:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Another is by taking a Java File object that points to a file. The following code is an example:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Methods that can be use with regards to FileOutputStream:

- **void close().** To close the output stream, this method is called.

- **protected void finalize().** This will ensure that connection is closed with no reference is active within the stream.
- **void write(int x).** A specified byte will be written into the stream.
- **void write(byte[] x).** A length of byte (x) will be written in a mentioned byte array.

Example. `fileStreamTest.java`

Code example of using InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

   try{
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("C:/test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("C:/test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

File Navigation in Java and I/O

In Java, there are many classes in relation to File Navigation and I/O. Some of the class are the following:

- File Class

In Java, a representation of the files and the directory pathnames is called a *File Class.* Some uses of this class is for file and directory formation, file search, deletion of file, and so on. Below is a list of constructors in creating a File Object.

- **File(File parent, String child).** From the specified parent abstract pathname and child string, a new instance will be created.
- **File(String pathname).** From the conversion of string (pathname) into an abstract pathname, a new instance of a file will be created.
- **File(String parent, String child).** From the pathname string of both parent and child, a new instance will be created.
- **File(URI uri).** A new instance of a file from conversion of URI to abstract pathname.

- FileReader Class

FileReader is used to print character streams. There are several constructors in this class to create the required objects. The following list are constructors provided by the class FileReader:

- **FileReader(File file).** From the given File, this will create a new FileReader.
- **FileReader(FileDescriptor fd).** From the given FileDescriptor, this constructor will create a new FileReader.
- **FileReader(String fileName).** From the given name of the File, this constructor will create a new FileReader.

The following are FileReader object methods that can be utilized to manipulate files:

- **public int read().** Will return an int (represent the read character) from reading single character.
- **public int read(char [] c, int offset, int len).** Will return the number of character based on the read character array.

Example. `FileRead.java`

```java
import java.io.*;
public class FileRead {

   public static void main(String args[])throws IOException {
      File file = new File("Hello1.txt");

      // creates the file
      file.createNewFile();

      // creates a FileWriter Object
      FileWriter writer = new FileWriter(file);

      // Writes the content to the file
      writer.write("This\n is\n an\n example\n");
      writer.flush();
      writer.close();

      // Creates a FileReader Object
      FileReader fr = new FileReader(file);
      char [] a = new char[50];
      fr.read(a);    // reads the content to the array

      for(char c : a)
         System.out.print(c);    // prints the characters one by one
      fr.close();
   }
}
```

The above code will have an output of:

```
This
is
an
example
```

- FileWriter Class

Inherited from OutputStreamWriter, this class is used for writing character streams. List of constructors are as follows:

- **FileWriter(File file).** From the given File object, this will create a FileWriter object.
- **FileWriter(File file, boolean append).** From the given File object and boolean value (to append or not), this will create a FileWriter object.
- **FileWriter(FileDescriptor fd).** From the given file descriptor, this will create a FileWriter object.
- **FileWriter(String fileName).** From the given file name, this constructor will create a FileWriter object.
- **FileWriter(String fileName, boolean append).** From the given file name and Boolean value (to append or not), this will create a FileWriter object.

Some methods, of FileWriter object, that can be used:

- **public void write(int *variable name*).** For one character.
- **public void write(char [] *variable name*, int offset, int *variable name*).** Will write from an array portion from offset with length.
- **public void write(String *variable name*, int offset, int *variable name*).** Will write from a string portion from offset with length.

Example. `FileRead.java`

```java
import java.io.*;
public class FileRead {

   public static void main(String args[])throws IOException {
      File file = new File("Hello1.txt");

      // creates the file
      file.createNewFile();

      // creates a FileWriter Object
      FileWriter writer = new FileWriter(file);

      // Writes the content to the file
      writer.write("This\n is\n an\n example\n");
      writer.flush();
      writer.close();

      // Creates a FileReader Object
      FileReader fr = new FileReader(file);
      char [] a = new char[50];
      fr.read(a);    // reads the content to the array

      for(char c : a)
         System.out.print(c);    // prints the characters one by one
      fr.close();
   }
}
```

The previous code will come up with the following output:

```
This
is
an
example
```

Directories using Java

Other files and directories is a list that can be contained in a directory File. To create directories, use *File object.* This method will allow in listing down all files available in a specified directory. Several methods can be utilized to call a File object and their relationship to directories.

Creating Directories

The following are useful methods to create a directory:

- The **mkdir( )** will create a directory provided by a path. Please note that directory can be created only if the parent directory is present.
- The **mkdirs()** will create a directory including the parent directory if it does not exists.

Example. `CreateDir.java`

```java
import java.io.File;
public class CreateDir {

   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);

      // Create directory now.
      d.mkdirs();
   }
}
```

The above code will create a directory based on path given.

Note – On UNIX and Windows, Java automatically handles path separators based on conventions. If using a forward slash symbol (/) on a Java version on Windows, it will be corrected by the route.

Listing Directories

From File object, the **list()** method will list down all directories and files that are present in a directory, the following is an example:

Example. `ReadDir.java`

```
import java.io.File;
public class ReadDir {

   public static void main(String[] args) {
      File file = null;
      String[] paths;

      try {
         // create new file object
         file = new File("/tmp");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch (Exception e) {
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

The above code will result in in the following based on the available files in the directory /tmp:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

LESSON 5
GRAPHICAL USER INTERFACE IN JAVA: ABSTRACT WINDOW TOOLKIT

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. recognize what is a Graphical User Interface;
2. distinguish the difference between a component and container;
3. determine the various parts of a GUI; and
4. build a Java program that would incorporate the various types of layout managers and panel.

READY. GET SET. DISCUSS!

GRAPHICAL USER INTERFACE

Graphical User Interface (GUI) is a category of user interface that enables the end user to interact with the screen using visual components and not text commands.

The following are Java APIs for GUI programming:

1. Abstract Window Toolkit (AWT)
2. Swing
3. JavaFX

INTRODUCTION TO AWT

Developed by Sun Microsystems for its Java Foundation Classes (JFC); **Abstract Window Toolkit (AWT)** is a Java class toolkit that helps programmers build GUI components and graphics**.**

It was developed to include a collection of standard tools for the design of cross-platform GUIs. One of AWT's essential features is that it is platform dependent. A significant downside to this approach; when applied on different platforms due to platform dependence, it will look completely different on an individual platform, which hampers the application's functionality and aesthetics. AWT includes 12 packages with 370 classes; the packages widely used are java.awt and java.awt.event.

The java.awt package contains the basic graphics classes in AWT:

1. GUI Component classes: Label, Button, and TextField.

2. GUI Container classes: Frame and Panel.
3. Layout managers: GridLayout, FlowLayout, and BorderLayout.
4. Custom graphics classes: Color, Font and Graphics.

Event handling is supported by the java.awt.event package:

1. Event classes: WindowEvent, MouseEvent, KeyEvent, and ActionEvent,
2. Event Listener Interfaces: MouseListener, MouseMotionListener, KeyListener, WindowListener and ActionListener.
3. Event Listener Adapter classes: KeyAdapter, MouseAdapter, and WindowAdapter.

.

JAVA AWT HIERARCHY

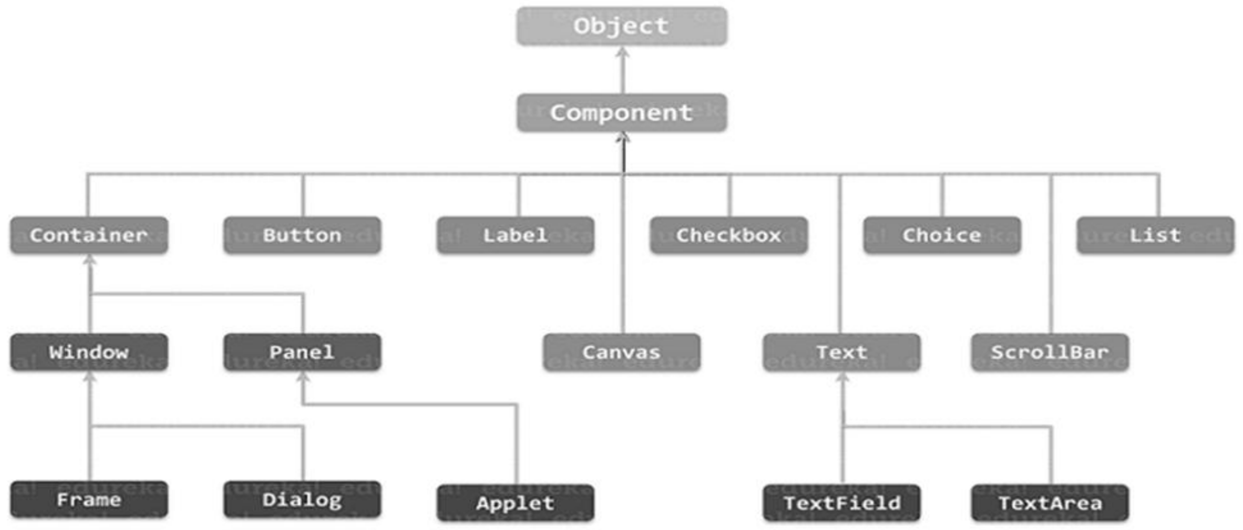


Figure 5-1. AWT Hierarchy

TWO TYPES OF GUI ELEMENTS:

1. **Component**: Components are simple GUI entities; examples are TextArea, Label, and Checkbox.

2. **Container**: Containers are used to keep components in a particular layout and hold sub-containers; examples are Frame and Panel.

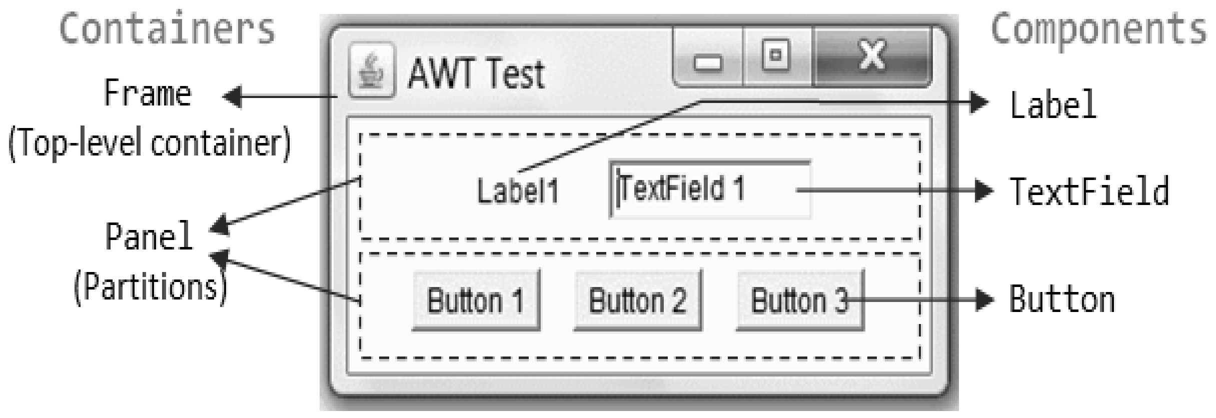


Figure 5-2. GUI Elements

The above figure shows three containers; a single frame and a couple of panels. In an AWT program, Frame is considered as a top-level container. A title bar, an optional menu bar, and a content display area can be found within a Frame. A Panel is an area that is used for grouping related GUI components in a layout. The figure shows five components: a Label, a TextField, and three buttons.

In designing a GUI, a component must be held inside a container. A container must be defined to keep the components.
For example,

```
      Panel pnl = new Panel();          // Panel is a container
      Button btn = new Button("Press"); // Button is a component
      pnl.add(btn);                    // The Panel container adds
 a Button component
```

AWT LAB ACTIVITY: AWT ACCUMULATOR

This example consists of a standard java.awt. Frame with four components: an "Enter an Integer" label, a TextField to accept user input, another "The Accumulated Sum is" label, and another TextField that is not editable to display the sum. The components re in a FlowLayout.

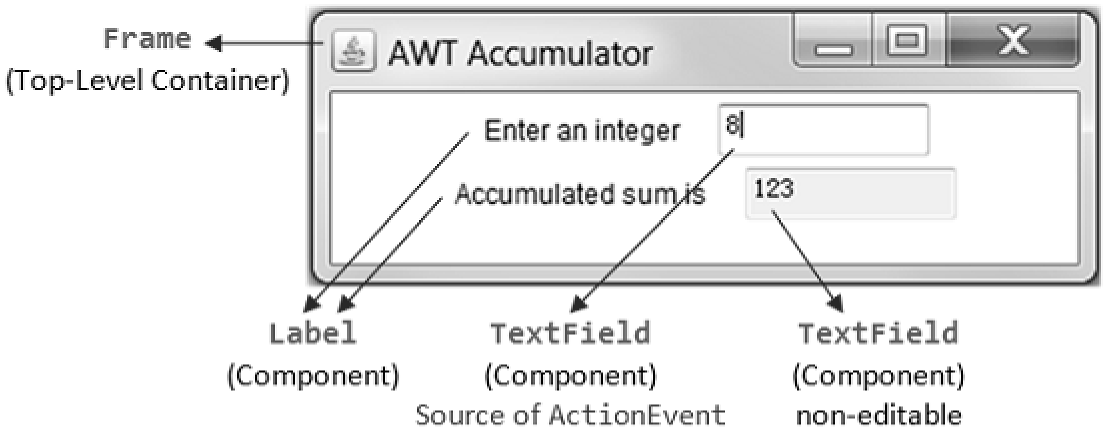


Figure 5-3. Sample AWT Accumulator Output

Example: `AWTAccumulator.java`

```
import java.awt.*;         // Using AWT container and component
classes
import java.awt.event.*;   // Using AWT event classes and
listener interfaces

// An AWT GUI program inherits from the top-level container
java.awt.Frame

public class AWTAccumulator extends Frame implements
ActionListener {
   private Label lblInput;      // Declare input Label
   private Label lblOutput;     // Declare output Label
   private TextField tfInput;   // Declare input TextField
   private TextField tfOutput;  // Declare output TextField
   private int sum = 0;         // Accumulated sum, init to 0

   // Constructor to setup the GUI components and event handlers
   public AWTAccumulator() {
      setLayout(new FlowLayout());
       // "super" Frame (container) sets layout to FlowLayout,
which arranges
       // the components from left-to-right, and flow to next
row from top-to-bottom.

      lblInput = new Label("Enter an Integer: "); // Construct
Label
      add(lblInput);               // "super" Frame container
adds Label component

      tfInput = new TextField(10); // Construct TextField
      add(tfInput);                  // "super" Frame adds
TextField

      tfInput.addActionListener(this);
        // "tfInput" is the source object that fires an
ActionEvent upon entered.
        // The source add "this" instance as an ActionEvent
listener, which provides
        //  an ActionEvent handler called actionPerformed().
        // Hitting "enter" on tfInput invokes actionPerformed().

      lblOutput = new Label("The Accumulated Sum is: ");  //
allocate Label
      add(lblOutput);                 // "super" Frame adds Label

      tfOutput = new TextField(10); // allocate TextField
      tfOutput.setEditable(false);  // read-only
      add(tfOutput);                // "super" Frame adds
TextField

      setTitle("AWT Accumulator");  // "super" Frame sets title
      setSize(350, 120);  // "super" Frame sets initial window
size
      setVisible(true);   // "super" Frame shows
   }

   // The entry main() method
   public static void main(String[] args) {
    // Invoke the constructor to setup the GUI, by allocating an
anonymous instance
      new AWTAccumulator();
   }

   // ActionEvent handler - Called back upon hitting "enter" key
on TextField
   @Override
   public void actionPerformed(ActionEvent evt) {
      // Get the String entered into the TextField tfInput,
convert to int
      int numberIn = Integer.parseInt(tfInput.getText());
      sum += numberIn;      // Accumulate numbers entered into
sum
      tfInput.setText("");  // Clear input TextField
      tfOutput.setText(sum + ""); // Display sum on the output
TextField
                                    // convert int to String
   }
}
```

AWT EVENT-HANDLING

Java uses an event-handling programming paradigm called "Event-Driven," like other visual programming languages. In an event-driven program, an event triggers a response from a specific user input, which then executes a piece of event handling code.

CALL BACK METHODS

In the Laboratory example, The process actionPerformed() is also known as a *call-back method*.. This means that; actionPerformed() is never explicitly invoked in your codes. Under certain circumstances, the graphics subsystem is called back as a reaction to specific user actions to the method actionPerformed().

SOURCE, EVENT AND LISTENER OBJECTS

The package java.awt.event contains the AWT's event-handling classes. Handling of events includes three types of objects: a source, listener(s) and an event-object.

The User interacts with the source object. When the source object is activated, it generates an event object that records the action. Triggering a source informs all listener(s) and invokes a proper event-handler for listener(s). Known as an observable-observer design pattern. Triggering a source will fire an event and invoke a suitable listener event-handler. As described in the order of steps shown in following figure:

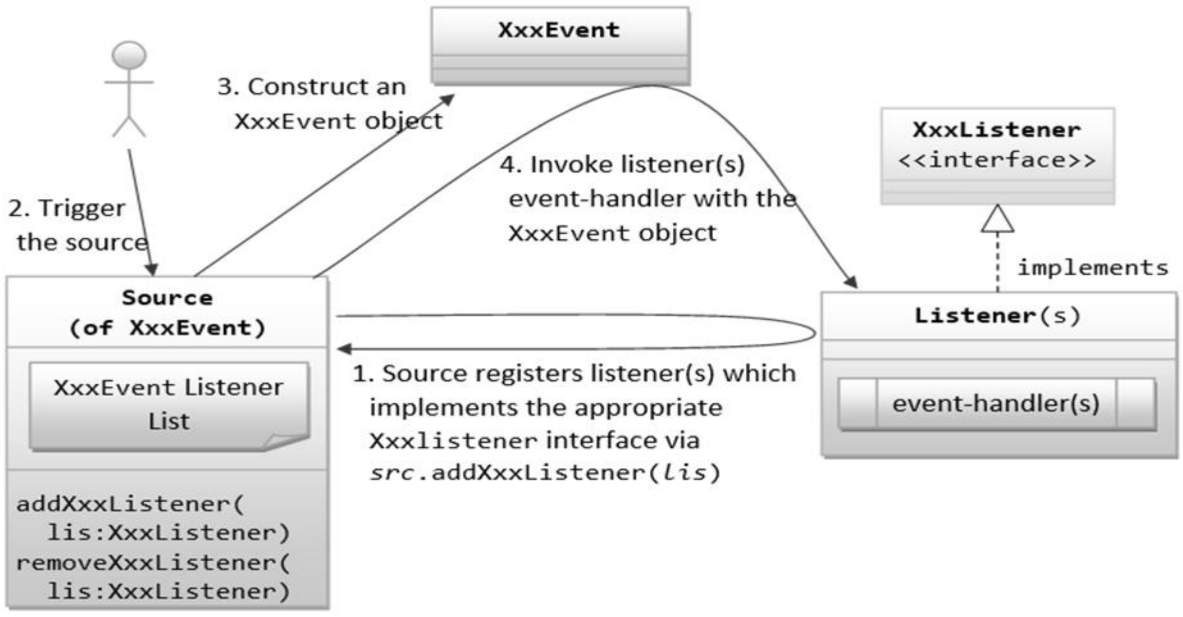

XxxEvent
3. Construct an XxxEvent object
4. Invoke listener(s) event-handler with the XxxEvent object
XxxListener <<interface>>
implements
2. Trigger the source
Source (of XxxEvent)
XxxEvent Listener List
addXxxListener(lis:XxxListener)
removeXxxListener(lis:XxxListener)
Listener(s)
event-handler(s)
1. Source registers listener(s) which implements the appropriate Xxxlistener interface via src.addXxxListener(lis)

Figure 5-4. Sequence of Steps for Source, Events, and Listener Objects

LAYOUT MANAGERS AND PANEL

A container has its components designed by a layout manager. The *layout managers* are platform-independent, in order to plot the user interface through all windowing systems. AWT offers the following managers for layout (in java.awt package):

1. FlowLayout,
2. GridLayout,
3. BorderLayout,
4. GridBagLayout,
5. BoxLayout,
6. CardLayout, and others.

FlowLayout

Considered as the default layout of the panel; the *FlowLayout* positions its components in a line. The following are fields of the FlowLayout class:

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

The following are FlowLayout constructors:

1. FlowLayout()
2. FlowLayout(int align)
3. FlowLayout(int align, int hgap, int ygap)
4. f.setLayout(new FlowLayout(FlowLayout.RIGHT))

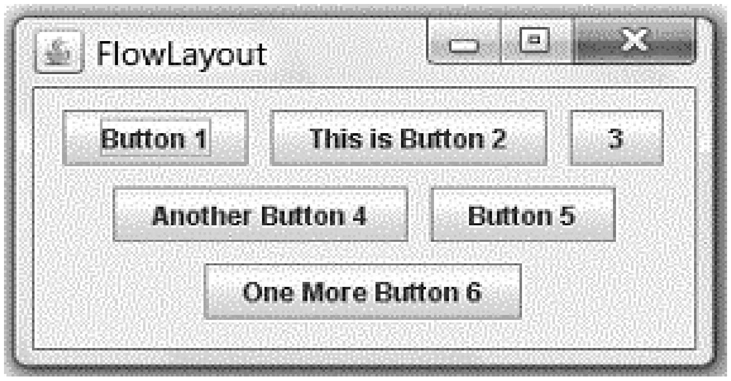


Figure 5-5. FlowLayout

GridLayout

The *GridLayout* presents the components in a grid-like rectangular form. Each rectangle displays one component. The following are GridLayout constructors:

1.GridLayout()
2.GridLayout(int rows, int columns)
3.JGridLayout(int rows, int columns, int hgap, int vgap)

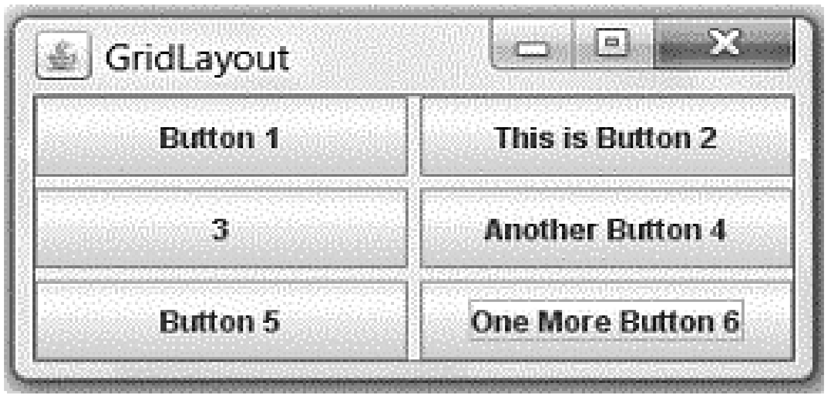


Figure 5-5. GridLayout

BorderLayout

The *BorderLayout* is the considered default layout for the frame and organizes its components in a specific area. Each area may contain only one component. Each of the regions has different constants:

1. public static final int NORTH
2. public static final int EAST
3. public static final int WEST
4. public static final int SOUTH
5. public static final int CENTER

BorderLayout constructors:

1. BorderLayout()
2. JborderLayout(int hgap, int vgap)

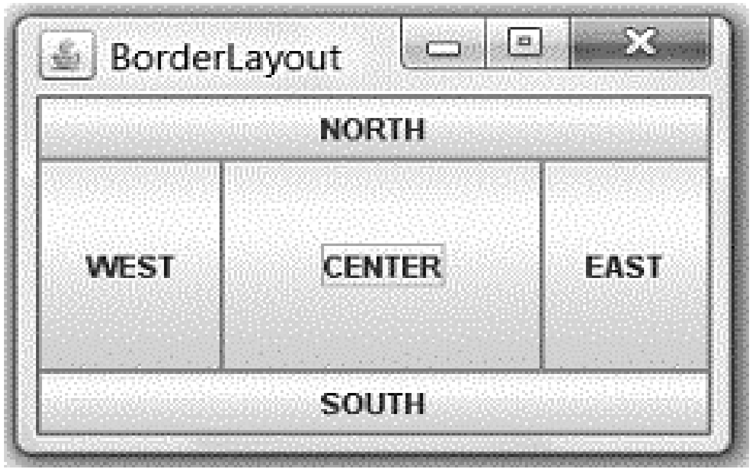


Figure 5-6. BorderLayout

Box Layout

The *BoxLayout* is used for the vertical or horizontal arrangement of the components. The following are constants for using the BoxLayout:

BoxLayout Fields:

1. public static final int X_AXIS
2. public static final int Y_AXIS
3. public static final int LINE_AXIS
4. public static final int PAGE_AXIS

BoxLayout Constructors:

1.BoxLayout(Container x, int axis)

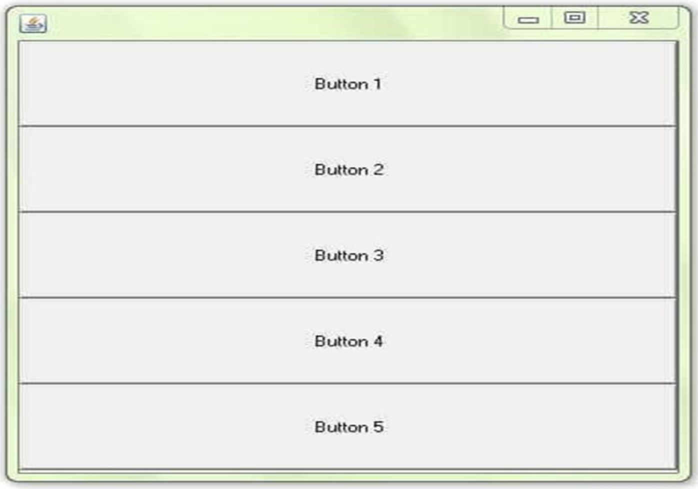


Figure 5-7. BoxLayout

Card Layout

The CardLayout class can only access one component at a time as it views each component as a card.

CardLayout constructors:

1. CardLayout()
2. CardLayout(int hgap, int vgap)

Commonly used CardLayout methods:

1. public void next(Container parent)
2. public void previous(Container parent)
3. public void first(Container parent)
4. public void last(Container parent)
5. public void show(Container parent, String name)

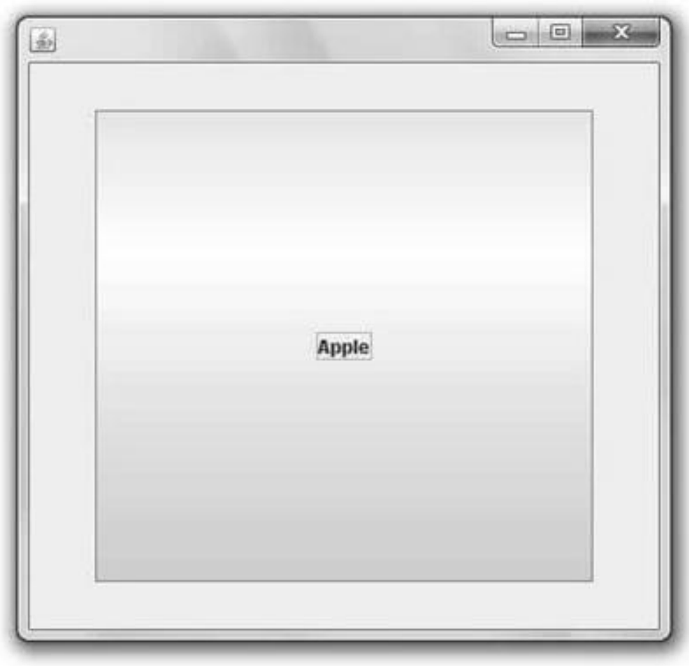


Figure 5-8. CardLayout

QUIZ IS IT!

Instruction: Select the best answer from the choices given below.

1. GUI stands for ___________________________
   a) Graphic User Interface
   b) Graphical User Interaction
   c) Graphical User Interfacing
   d) Graphical User Interface

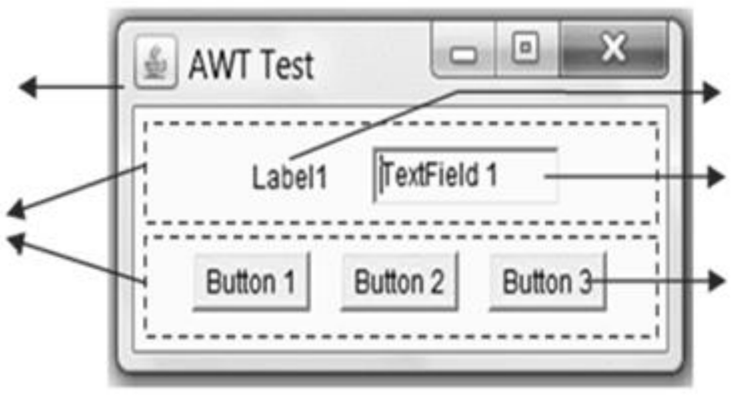
2. GUI does not allow users to interact and manipulate the screen, rather, GUI is used to develop the database
   a) First statement is true, second statement is false
   b) First statement is false, second statement is true
   c) Both statements are true
   d) Both statements are false

3. Abstract Window Toolkit consist of ____________________
   a) 12 packages and 360 classes
   b) 12 packages and 370 classes
   c) 360 packages and 12 classes
   d) 370 packages and 12 classes

4. Among the following, what not an example of a component in GUI?
   a) Frame
   b) Label
   c) Button
   d) TextField

5. Among the following, what is not an example of a container?
   a) Button
   b) Panel
   c) Frame
   d) All of the above

6 – 10. Identify the parts in the figure below



11. What component can the user use for keeping other components as a container?
    a) Tupperware
    b) Choice
    c) JPanel
    d) None of the Above

12. Select what step must be first completed inside a container?
    a) Create a layout manager
    b) Doesn't Matter
    c) Add Component
    d) All of the Above

13. What method determines how the components are organized in a container?
    a) setLayout ( )
    b) setVisible ( )
    c) setLayoutManager ( )
    d) None of the Above

14. What container is often used for dividing an interface into different layout managers?
    a) JWindow
    b) Container
    c) JPanel
    d) None of the Above

15. What is considered the default panel layout manager?
    a) FlowLayout
    b) No default
    c) GridLayout
    d) BoxLayout

16. Where does the BorderLayout class derive its name from?
    a) The components border
    b) The way components are organized at a container 's borders
    c) The borders of Java Developers
    d) None of the Above

17. What are action events?
    a) The events that takes place as a reaction to something else.
    b) The response indicates some form of action should be taken.
    c) A tribute to the movie Action Jackson.
    d) Both A and B

18. What does the addActionListener() method imply as an argument?
    a) the listener should respond when an event is triggered.
    b) this event takes superiority over others.
    c) this object controls the events.
    d) this object has no event

19. Which one stores input as integers?
    a) JButton
    b) JTextField or JTextArea
    c) Neither A nor B
    d) GridLayout

20. Who developed AWT?
    a) KrnFX
    b) JavaFX
    c) JavaBeans
    d) Sun Microsystems

Draw Your Insight

1. How do components become organized in a container if there is no designated layout manager?

2. Why do GUI classes, have the letter J at the start of their names? (e.g., JButton, JLabel, and JTextField)

Laboratory Exercise

Write a java program that would implement one among the different types of layout managers and panel. The program should ask for your Student Number, Last Name, First Name, and Course. (Note: You can include additional GUI components to enhance your output)

LESSON 6
GRAPHICAL USER INTERFACE IN JAVA: SWING AND JAVAFX

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. compare and Contrast Swing and AWT java source file and output;
2. recognize the major differences between SWING, AWT and JavaFX; and
3. construct a java program that would implement either SWING, AWT, or JavaFx.

READY, GET SET, DISCUSS!

INTRODUCTION TO SWING

Swing was launched in 1997 as part of JFC (Java Foundation Classes) after JDK 1.1 was release. Subsequently, *JFC* has been an integral part of JDK starting from JDK 1.2. JFC consists of the following**:**

- Swing API: for advanced graphics programming.
- Accessibility API: offers support services for individuals with disabilities.
- Java 2D API: for quality two-dimensional graphics.
- Pluggable L&F(look-and-feel) support.
- Support for drag and drop.

The aim of Java GUI is to help developers build a user interface that looks good on any platforms. JDK 1.0's AWT was cumbersome and non-object-oriented (using a lot of event.getSource()). JDK 1.1's AWT presented a much simpler and object-oriented event-delegation (event-driven) model. Furthermore, the introduction of JavaBeans and inner classes enables the Java programmers to have a visual programming environment similar to VB (Visual Basic).

SWING'S FEATURES

Swing is large it was made up of 737 classes unevenly distributed inside 18 packages. Compared to AWT, Swing offers a broad and comprehensive set of reusable Interface components, as shown in the figure below.

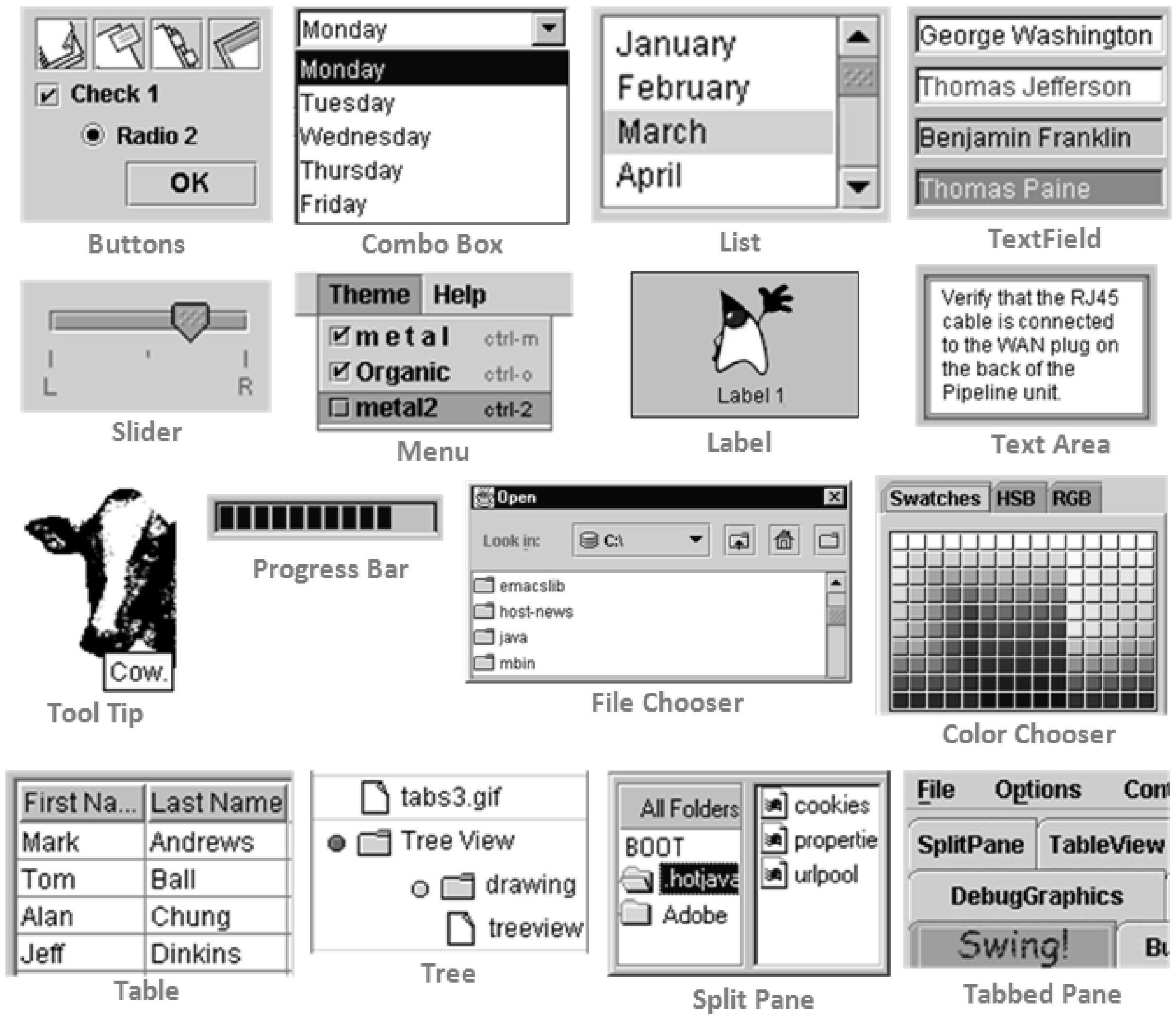
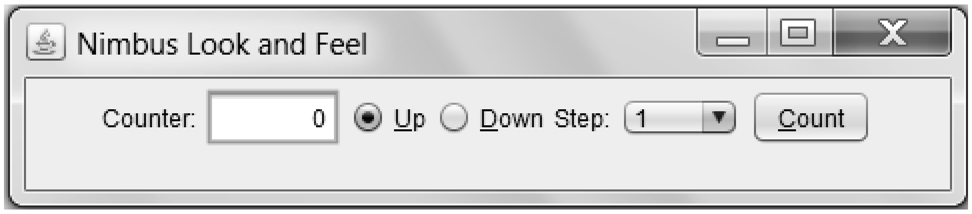


Figure 6-1. Features of SWING

THE MAIN FEATURES OF SWING ARE:

1. Swing is developed using Java and is 100% portable.
2. Swing components are very lightweight compared to AWT components which uses more system resources.
3. Swing components support a pluggable L&F(look-and-feel). You can switch between Java L&F and your underlying OS L&F. They are shown in the picture below.



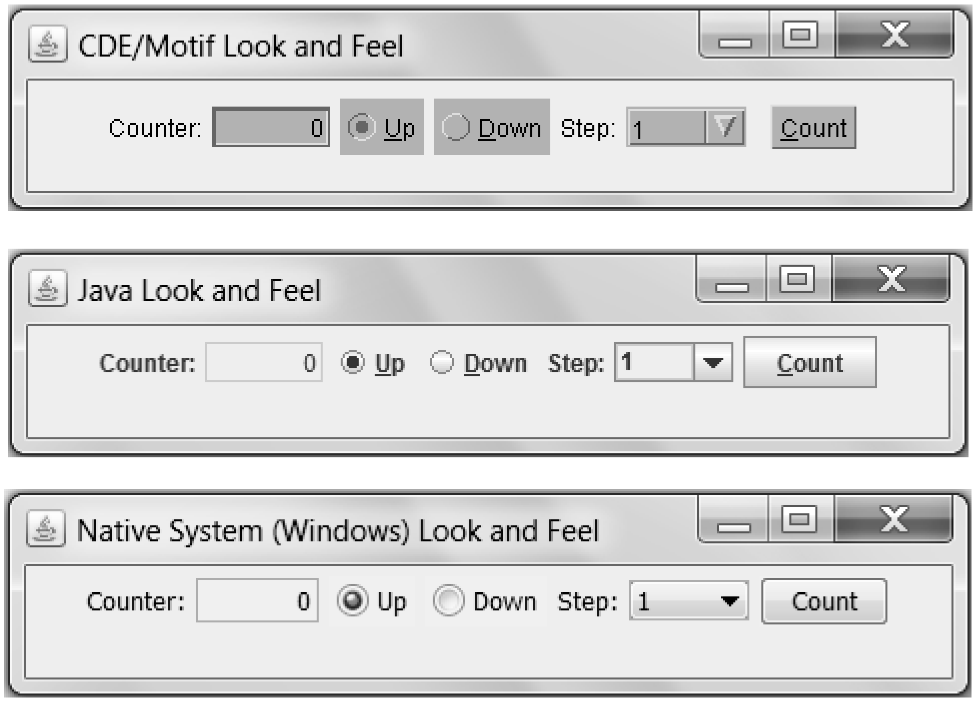


Figure 6-2. Different Representations of SWING using different OS

4. Swing can work(mouse-less) even with just a keyboard.
5. Swing components support "tool-tips".
6. Swing components can be drag-and-drop into the "design form" by using the "GUI builder" and adding event handlers to it is a piece of cake (just like in Visual Basic).
7. To handle events, Swing uses the classes from java.awt.event and javax.swing.event package, however the former was often used than the latter.
8. To handle layouts, Swing uses the layout manager classes from java.awt(e.g. BorderLayout and FlowLayout) and javax.swing(e.g BoxLayout, Springs and Struts) package.
9. Swing incorporates double-buffering and automatic batching for smoother and faster screen repainting.
10. For building multiple document interface (MDI) applications Swing uses the JLayeredPane and JInternalFrame classes.
11. Swing also supports “undo”, splitter and floating toolbars using JToolBar.

DIFFERENCE BETWEEN AWT AND SWING

FEATURE	AWT	SWING
Platform Independent	No	Yes
Follows MVC	No	Yes
Components are	Lesser and heavyweight	More powerful and lightweight
Supports pluggable look and feel	No	Yes

SWING'S COMPONENTS

Similar to AWT's component classes, *Swing components* also have components such as *Button, TextField, Label, Panel, Frame*, and *Applet* you just need to prefix it with letter "J" e.g. JButton, JTextField and so on and so forth.

The figure below shows the hierarchy of the GUI swing classes. Same as AWT, it has two class groups: *containers* and *components.* The *container* shall be used to house the components. A container can also hold another container because it extends the Component class. As a rule of thumb, do not use AWT components with Swing components as heavy-weight components (AWT components) masks lightweight components (Swing components) when rendered.

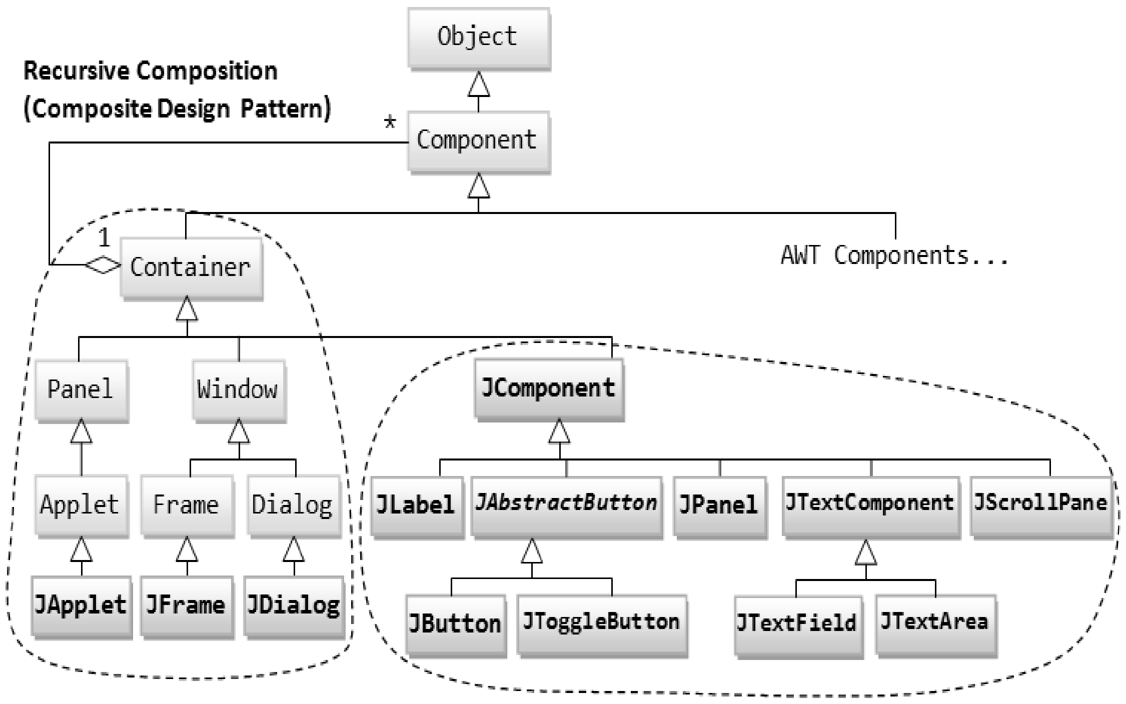
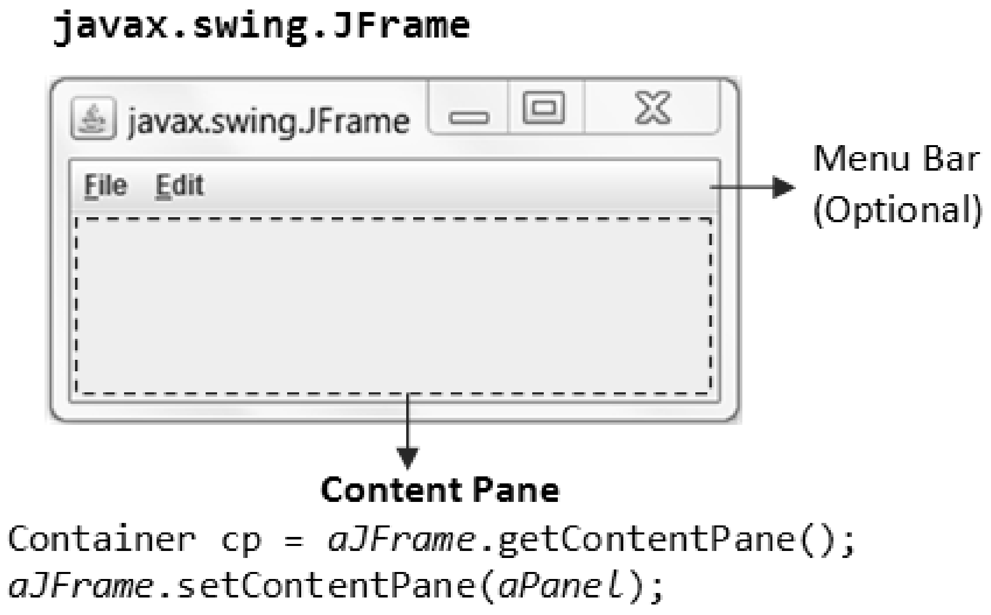


Figure 6-3. Components of SWING

SWING'S TOP-LEVEL AND SECONDARY CONTAINERS



```
Container cp = aJFrame.getContentPane();
aJFrame.setContentPane(aPanel);
```

Figure 6-4. SWING top level and secondary containers

Just like the AWT application, the Swing application requires a top-level container, and Swing has three:

1. **JFrame** - for creating the main window. It has an icon, title, content pane and an optional menu bar. It also has the maximize, minimize, restore and close button at the top-right position on Windows and at the top-left position on Mac and Linux.
2. **JDialog** - for creating the pop-up window. It has an icon, title, content pane and a close button.
3. **JApplet** - for rendering applets within the browser window.

And just like AWT, Swing has a similar Panel class called JPanel for grouping components.

EVENT-HANDLING IN SWING

To handle events Swing uses the classes from java.awt.event and javax.swing.event packages, however the former is often used than the latter.

Writing Swing Applications

In order to create an application using Swing you must:

1. Use the "J" prefix in Frame, Button, TextField, and Label, etc. for example JFrame, JButton and so on and so forth.
2. Use a top-level container like JFrame to house the components. And in order to add a component like a JButton button to the frame you must invoke the getContentPane().add() method e.g. frame.getContentPane().add(button).
3. Use AWT's event-handling classes such as ActionEvent to handle events.
4. For thread safety do not run the event constructor in the Main Thread, instead you must run it in the Event Dispatcher Thread.

Example: `SwingAccumulator.java`

```
import java.awt.*;        // Using layouts
import java.awt.event.*; // Using AWT event classes and listener
interfaces
import javax.swing.*;     // Using Swing components and
containers

// A Swing GUI application inherits the top-level container
javax.swing.JFrame
public class SwingAccumulator extends JFrame {
     private JTextField tfInput, tfOutput;
     private int sum = 0;     // accumulated sum, init to 0

     // Constructor to setup the GUI components and event
handlers
     public SwingAccumulator() {
          // Retrieve the content-pane of the top-level
container JFrame
          // All operations done on the content-pane
          Container cp = getContentPane();
          cp.setLayout(new GridLayout(2, 2, 5, 5));  // The
content-pane sets its layout

          cp.add(new JLabel("Enter an Integer: "));
          tfInput = new JTextField(10);
          cp.add(tfInput);
          cp.add(new JLabel("The Accumulated Sum is: "));
          tfOutput = new JTextField(10);
          tfOutput.setEditable(false);  // read-only
```

```
            cp.add(tfOutput);
            // Allocate an anonymous instance of an anonymous
inner class that
            //  implements ActionListener as ActionEvent listener
            tfInput.addActionListener(new ActionListener() {
                  @Override
                  public void actionPerformed(ActionEvent evt) {
                        // Get the String entered into the input
TextField, convert to int
                        int numberIn =
Integer.parseInt(tfInput.getText());
                        sum += numberIn;      // accumulate numbers
entered into sum
                        tfInput.setText("");  // clear input
TextField
                        tfOutput.setText(sum + ""); // display sum
on the output TextField
                  }
            });

            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //
Exit program if close-window button clicked
            setTitle("Swing Accumulator"); // "super" Frame sets
title
            setSize(350, 120);  // "super" Frame sets initial size
            setVisible(true);   // "super" Frame shows
      }

      // The entry main() method
      public static void main(String[] args) {
            // Run the GUI construction in the Event-Dispatching
thread for thread-safety
            SwingUtilities.invokeLater(new Runnable() {
                  @Override
                  public void run() {
                        new SwingAccumulator(); // Let the
constructor do the job
                  }
            });
      }
}
```

Activity:

1. Review your AWTAccumulator program and compare it to the SwingAccumulator source file. Draw insights about the differences between the output of the two source file. How do they differ? And what do you think are the reasons why the two source files have differences in their outputs?

INTRODUCTION TO JavaFX

Same as AWT and Swing, JavaFX was developed to build GUI for Java applications. Having a rich collection of graphic libraries it was initially designed for Rich Interface Applications(RIA) and the like such as rendering GUI for web apps with the use of a browser plugin and was just later become a part of JDK library starting from JDK 8.

JavaFX KEY FEATURES

The main features of JavaFX are the following:

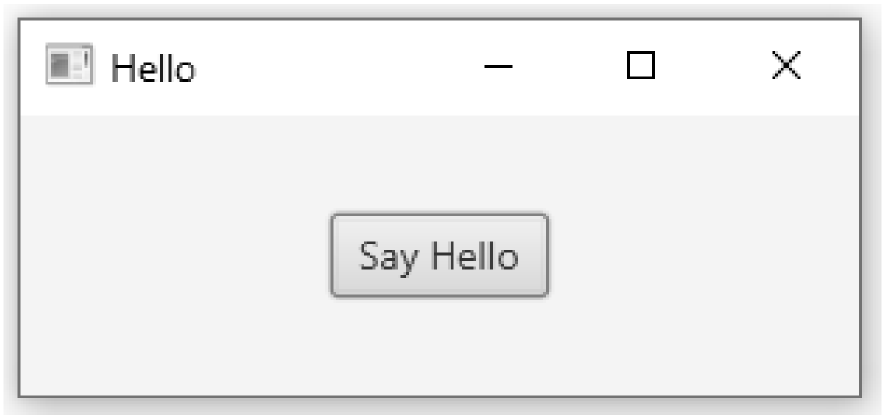
1. It was developed in Java and is a part of the JDK library since JDK 8.
2. It has CSS support for skinning.
3. It supports FXML.
4. It supports Swing interoperability where you can use Swing UI for JavaFX applications.
5. It has a WebView for the embedding of HTML content.
6. It has a 2D/3D Graphics.
7. It has a media support for: audio, video and image.
8. It provides a JavaScript engine.

JavaFX PACKAGES

JavaFX has 36 packages. The following are the commonly used ones:

- javafx.application: JavaFX application
- javafx.stage: Top-level containers
- javafx.scene: Scenes and Scene Graphs.
- javafx.scene.*: Controls, Layouts, and Shapes, etc.
- javafx.event: Event Handling
- javafx.animation: Animation

JAVAFX LAB ACTIVITY: HELLO WORLD



Example: `JavaFXHello.java`

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXHello extends Application {
     private Button btnHello;  // Declare a "Button" control

     @Override
     public void start(Stage primaryStage) {
          // Construct the "Button" and attach an "EventHandler"
          btnHello = new Button();
          btnHello.setText("Say Hello");
          // Using JDK 8 Lambda Expression to construct an
EventHandler<ActionEvent>
          btnHello.setOnAction(evt -> System.out.println("Hello
World!"));

          // Construct a scene graph of nodes
```

```
            StackPane root = new StackPane();  // The root of
scene graph is a layout node
            root.getChildren().add(btnHello);  // The root node
adds Button as a child

            Scene scene = new Scene(root, 300, 100);//Construct a
scene given the root of scene graph
            primaryStage.setScene(scene);    // The stage sets
scene
            primaryStage.setTitle("Hello");  // Set window's title
            primaryStage.show();             // Set visible (show
it)
      }

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

How It Works

1. A JavaFX program extends javafx.application.Application (same as Swing who extends javax.swing.JFrame).
2. Just like AWT JavaFX has a variety of components such as Label, Button, TextField, etc.
3. In JavaFX you can add an event handler to a component using the setOnAction() method which accepts a parameter with a type EventHandler<ActionEvent>. Just like the code below:

```
public final void setOnAction(EventHandler<ActionEvent> value)
```

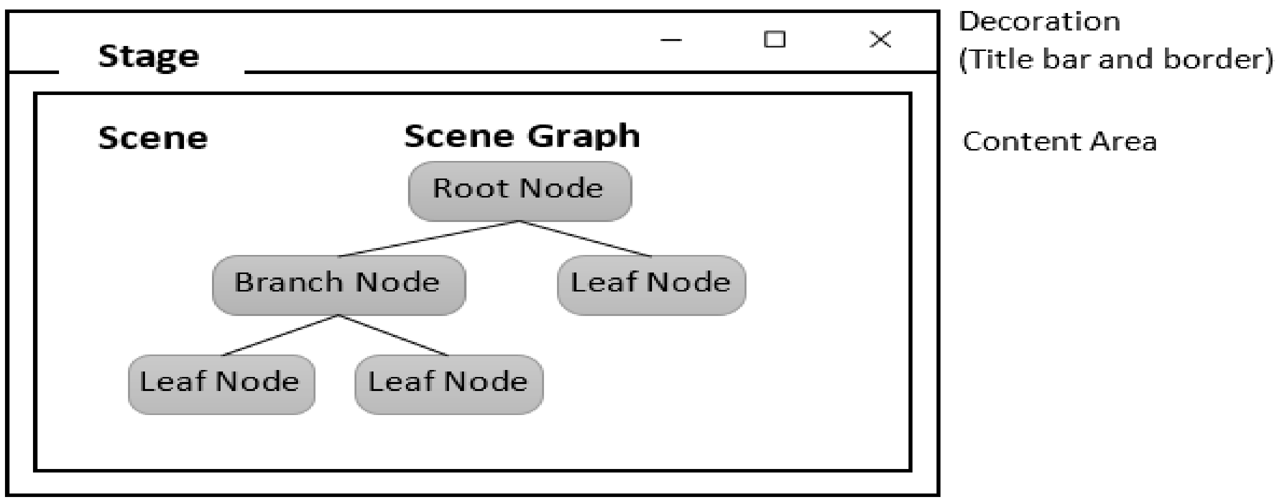
The EventHandler interface requires a handle() method as defined below:

```
package javafx.event;
@FunctionalInterface
public   interface   EventHandler<T   extends   Event>   extends
EventListener {
      void handle(T event);  // public abstract
}
```

The handle() method can be triggered programmatically using fire() method or by a click, touch or keypress event.

4. JavaFX uses a theater metaphor to model the graphics application. The **stage(javafx.stage.Stage)** is the top-level container. And the components are house inside the **scene(javafx.scene.Scene)** where the scene can be more than one, but only one scene

can be displayed on the stage at any given time. The contents of the scene are reflected in the **node(javafx.scene.Node)** graph of the hierarchical scene.



5. JavaFX creates the UI by:
   a) Preparing a graph of the scene.
   b) Building the scene in the root of the scene graph.
   c) Setting up the stage with the created scene.

In the code example, the root node is the container(in this case the javafx.scene.layout.StackPane) that layouts the child nodes. It has a child node the Button which is added to the layout using the code below:

```
aLayout.getChildren().add(Node node)          // Add one node
aLayout.getChildren().addAll(Node... nodes)   // Add all nodes
```

To instantiate a scene(javafx.scene.Scene) a root of the scene graph is required by the constructor:

```
public Scene(Parent root, double width, double height)
```

where root(javafx.scene.Parent) must be subclass of javafx.scene.Node. Then we must set the stage's scene and title before showing it.

Laboratory Exercise

1. Recall your Java Exercise on Lesson 2 (BMI). Modify the java program and implement a Graphical User Interface in it. You can choose between SWING, AWT and JavaFX to modify and construct the GUI of the program.

LESSON 7
OBJECT-ORIENTED APPLICATION DEVELOPMENT PART 1

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. give comparison between the classes in the real-world and objects to software classes and objects;
2. survey structured and object oriented application development;
3. draw/sketch a class diagram for a specified class;
4. code object defining classes;
5. code applications that creates objects; and
6. relate and use the composition as relationship between classes.

READY. GET SET. DISCUSS!

OVERVIEW OF OBJECT ORIENTED APPLICATION DEVELOPMENT

In the real world, we are surrounded by different interacting things such as people, trees, cars, buildings, and so on. This real world things are often considered parts of the real world systems in which the things within the system are related and can interact in an organize way. A real world system is often managed using an information system. For example, a business enterprise or a university are complex systems that requires sophisticated information system. The things that are components of such systems (managers, supervisor, accountant, inventories, sales, products, teachers, courses, and students) are often modeled or represented within the information system to keep the real world system organized and running smoothly.

You can divide the real-world things such as students, teachers and courses into groups or categories, called *classes.* If you are reading and studying this chapter you might and probably belong to the student group or class. Your professor/instructor belongs to teacher class and this IPT01 belongs to a course class. If you use the term object instead of *thing*, then you could say that a ***real-world object*** (*a thing in the real world object that has attributes and behaviors*) can belong to a particular ***real-world class*** (*a category to which real-world object belongs*). Figure 4.1 shows examples of specific objects that belong to particular, tree, car and people classes. The representation of the real world object which belongs to a real-life class in necessary as with use of ***software object*** (depiction of a real-world object in computer memory) that goes to a ***software class*** (program code that depicts a real-world class) that you write using a programming language such as Java.

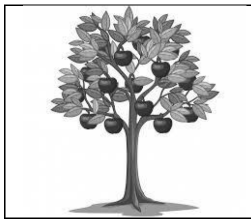


Figure 7.1 Objects in the real world

Attributes and Behaviors of Objects

What exactly is an object? It is simply an entity with two primary features. First a real-world object has attributes. **Object attributes** are the characteristics or features that define the object. For example a real-world student objects in a university information system have many attributes, including the name, address, favorite color, and favorite food. Though, if your task is to develop a UIS(University Information System), the attributes you must consider are the name and address, and not the favorite color and favorite food. A student object in a UIS could have other attributes which includes student ID number, a gender, contact number, address and birthdate.

A real-world student object would have another important attribute such as a class schedule, which by itself can also be considered as an object. A schedule object has attributes, such as semesters and courses. The course can also be assumed as an object in this example. A course has attributes such as course number, course title, and section. A course section is an object that has attributes such as number, meeting time, room number, and instructor. An instructor is an object that has attributes such as a faculty ID number, name, gender and birthdate.

You can see that analyzing and designing an information system using all these objects can get very involved. The fact that all these objects (student, schedule, course, section, and instructor) contains attributes/feature/characteristics and can be related to each other. Modelling this complex real-world system (UIS) involves real-world objects like students, course and instructors can be done using an OOP language such as Java.

The second major feature of a real-world objects is **object behavior**. The object's behavior are methods, actions, operations or processes that describe what the object can do. As with attributes, there are some action/methods relevant to a particular information system. The action like eating, sleeping and studying can be an example of an object's behavior, but those are not applicable to a university information system. The behaviors that are relevant in the university information system can include student behavior of registering for a particular course, paying tuition fee, taking exam, attending class, submitting requirements and checking grades. You are undoubtedly aware that a complex university system needs an information system to keep track of all students, course and instructors, including their relevant attributes and behavior.

Structured Application Development

As you know, information systems are needed and manage real-world systems such as a university or a business enterprise. An information system is often composed of many interacting applications, such as a student-scheduling application, a grade-assignment application, and learning management system. Applications can be developed in essentially one of two ways, either using ***structured application development*** or ***object-oriented application development***. These two approaches have much in common- regardless of the approach, the developer must perform these steps:

1. Understand the problem.
2. Designing or planning the application
3. Write the code.
4. Compile and test.
5. Deploy the application.

The difference is how these steps are performed. The **structured application development** is a method for developing and creating an application that emphasize on the data flow of the application from one system component to another, and in a way the system components are related by system events. The applications created using this methodology often lean towards procedural approach. For example: retrieve the data for the student, retrieve the data for course, and assign the student to the course, and the output the data for schedule. The procedural applications are typically made for a procedural programming language like C.

The Object-Oriented Application Development (OOAD) in Action

The object-oriented application development primary focus is on framework or context of object, which involves the combination of data (attributes) and behavior (process). The **object-oriented application development (OOAD)** is an approach in creating applications or programs that stresses and highlights on objects with in the system and how this objects interrelate or interact. The OO developer must still go through the steps of understanding the problem, planning the application, writing the code, compiling and testing and deploying the application, thinking in terms of object. For example a student scheduling application involves student object, course objects and schedule objects.

OBJECTS AND CLASSES

The objects and classes are the most important element or concept in an object oriented application development. The following section will discuss and explain these concepts.

Objects

As discussed previously, a real-world object such as a person, a car, a tree, an invoice, course is something that has attributes/states and behaviors/methods. For example, an object student can be stored in computer memory to model a certain real-world student. The term **instance variable** is used to describe an attribute of an object, and the term **instance method** is used to describe the behavior of the object (instance methods are usually called as method). A student object can have instance variable of name, gender, address, course and grade with the corresponding methods for course and grade the setCourse(), getGrade().

An object can be recognized by giving reference or name. For example a student object can be created as shown in the following Java statement:

*Student **myStudent** = new Student (sid, lastName, firstName, gender);* ☜

The reference to the student object is *myStudent* (you can use any reference you choose, such as *theStudent, aStudent, studentA, or student1*), which refers to the object in the memory. The variable *myStudent* is actually a reference variable that holds a memory address of the object. The data type for this object is *Student*. This is an example of an **abstract data type,** which is an entire set of instance variable (*sid, lastName, firstName, gender*) and methods for manipulating those variables.

The java keyword *new* is use to create an object. The code *Student (sid, lastName, firstName, gender)* is a call to the constructor named *Student*, a class method that brings out the specifics or the details in creating a *Student* object.

Classes

Representing real world objects with software objects is a great idea, but how are the objects actually implemented? This is accomplished with the aid of a java class. You have already created a dozen of class using java. The first one is created in the first chapter with the class name *Welcome* and prints “Welcome to java!”. However, this class that you have created contains a *main* method that contains procedural code designed to accomplish a specific computing objective. This kind of software class, which is called **application class,** has nothing to do with the concept of software objects. Its only purpose, as its name suggests, is to manage specific computer application, not OO applications. The class can be used and considered as the blue print or template for creating objects in OO applications. Such class is called an **object-defined class.** An object-defining class is said to have members –data members (instance variable or field) and method members (usually designed to manipulate the member fields). The object-defining class in an OO application is the specification or blueprint for objects, rigorously defining the object’s fields and methods. The class diagram is a tool that can be used to create, illustrate or define object model. **Instantiate** is the technical term used to signify the method or procedure of making and creating an object from a specific class.

The Visibility of Instance Variables and Methods

Variables that have been declared within a method have method scope. This means that a variable can only be referred or access by their simple names inside that method (the local scope). In object-defining class, the instance variables is declared within a class, not within a method, and basically define what object is. Such variables have a characteristic known as visibility which is similar to scope. The **visibility** means that the instance variable can also be referenced or access by its simple name. The access modifier describe and define the visibility of the instance variable. The access modifiers can be private, public or protected. In the private access modifier variables has only visibility within its defining class, public access modifier means it has visibility in any class, protected implies that the instance variable has visibility only in subclasses of that class, or classes in the same package.

Why visibility is important? It is a matter of security. The instance variable helps define exactly what an object is. Through visibility, accessing and manipulating other classes without using carefully planned procedures can be avoided. The best way to access a private instance variable of an object is to use public method written specifically for that purpose.

The Class Diagram

The class diagram is component of a Unified Modeling Language (UML) which defines the static or fixed view of a class or application. Class diagram often use in describing, documenting, and visualizing diverse components of an application. It defines both the attributes and behaviors of a specific class. It also captures the constraints and relationship between classes on the system. The use of class diagram is highly suggested in the modeling of object oriented systems. Because this is the only UML diagrams that can be mapped directly with object-oriented languages like java.

There are three sections in a standard class diagram which is shown in figure 4.2:

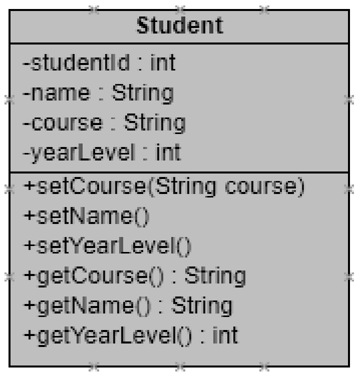


Figure 4.2 Class Diagram for Student Class

Upper portion: The ***class name*** (Student). This portion contain the name of the class and is required for reference.

Middle portion: The ***attributes*** of the class, (studentId, name, course, yearLevel and their corresponding data types), is the portion where the attribute/qualities/characteristics of the class is defined.

Bottom portion: The class **operations or methods**, is the portion where each operations and methods are listed. The methods defines the interactions with data in the class.

> The bottom portion in this class diagram can contain methods which can be a **mutator method** (setter method) and **accessor method** (getter method). The term mutator means to change or alter and the accessor means to access or retrieve. These sets of methods are used to assign and get values from the instance variables.

Member access modifiers or visibility.

There are different symbols use to denote the access modifier or visibility of attribute or method in a class. The access modifiers can use *plus sign (+) for public, minus (-) for private, hashtag (#) for protected,* tilde *(~) for package, slash (/) for derived, and undeline (_) for static* attributes and methods. In visibility, the public and private are the most commonly used access modifiers. See figure 4.2.

Application:

✓ Understanding the Problem

To begin a simple OO application understanding the domain problem should come first.

The domain of the project would be a selling of clothes. In this application the sale is accomplished by providing an invoice that contains invoice number, date of sale, and the total amount for the sale. The application allows the salesperson to enter an invoice number and total. The invoice date comes from the system clock of the computer. Because the manager offers 5 percent discount to the store’s best customers. There are two types of invoices the one with no discount and the other with 5 percent discount.

✓ Planning the Application

In creating a plan for this application the UML class diagram will be used. This diagram will serve as the blueprint of the application.

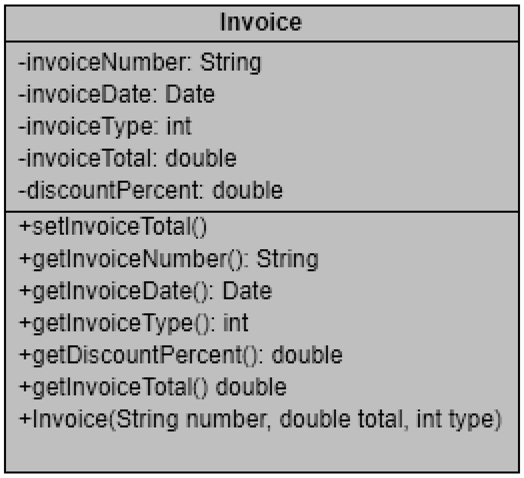


Figure 7-3 Class Diagram for Invoice

The name of the class is *Invoice* that is displayed on the top section. The data member are listed in the middle section. The invoice has five instance variables: *invoiceNumber, invoiceDate, invoiceType, invoiceTotal* and *discountPercent*. All instance variables in the invoice are set to private by putting (-) sign in the begging of each variables. These variables are accessed and manipulated using the public mutator methods and the accessor methods. The (+) sign is placed in the beginning of each method to denote that the methods are public.

✓ Writing the Code

Figure 7.4 provides the program code for Invoice.java. This file contains the class used to create Invoice objects in an application

```
/*
 * File Name : Invoice.java Date Created : 06/22/2020
 * Purpose : to define Invoice Objects
 */

import java.util.Date
public class Invoice{
      //declare instance variables
      private String invoiceNumber;
      private Date invoiceDate;
      private int invoiceType;
      private double invoiceTotal;
      private double discountPercent;

      //define constructor
      public Invoice(String number, double total, int type){
            invoiceNumber = number;
            invoiceDate = new Date();
            invoiceType = type;
            setInvoiceTotal(total, type);
      }
      //mutator method
      public void setInvoiceTotal(double invTotal, int invType){

            if (invType ==1){
                  discountPercent= 0.05;
                  invoiceTotal = invTotal *(1-discountPercent);
            }
            else{
                  discountPercent = 0;
                  invoiceTotal= invTotal;
            }
      }
      //accessor method
      public String getInvoiceNumber(){
            return invoiceNumber;
      }
      public Date getInvoiceDate(){
            return invoiceDate;
      }
      public int getInvoiceType(){
            return invoiceType;
      }
      public double getDiscountPercent(){
            return discountPercent;
      }
      public double getInvoiceTotal(){
            return invoiceTotal;
      }
} // end class
```

✓ Compiling and Testing

The file Invoice.java can be compiled, but cannot be tested unless an application class (with a main method) is added in the class.

Encapsulation

The *encapsulation* can be defined as the process of wrapping, bundling or binding the data and methods that manipulate data together. This OOP concept preserves the implementation away from its user thus keeping it safe from outside, intrusion, and misuse. This concept headed towards another important concept of OOP which is the **data hiding** and **data abstraction**.

Data abstraction is a process in which only the interface are expose to the users thus hiding its implementations and details away from the user. Figure 4.5 shows how the attributes and functions of the Invoice class are bundled into a single class.

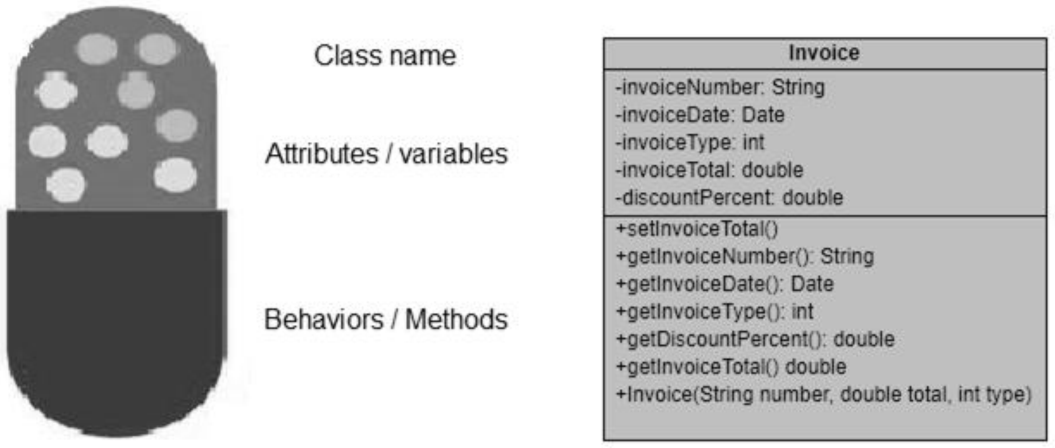


Figure 7-5 Example of Encapsulation (the Invoice class)

Information Hiding

Encapsulation of an object's attributes and behaviors within a class enables an important characteristic of OOP called information hiding. The **information hiding** is an OOP concept that describes the mechanism of hiding and concealing the information about how an object is implemented within the specified class. For example, for an OO application to create an invoice, it just needs to know how to call the Invoice constructor. The details of how an Invoice object is constructed within the Invoice class are of little concern to other programmers who may want to use the Invoice class in other applications. In this concept, the programmer will only be concerned with the required data in Invoice constructor and what the Invoice class can provide.

A COMPLETE OBJECT ORIENTED APPLICATION

The succeeding application is an example of a complete OOA which follow the IPO(Input-Process-Output) pattern. The application allows the users to enter data, generate objects in computer memory, and provide necessary output or information about those objects.

Application Classes

The class with main method is required in an application class because this method is the starting point of execution. The object-defining class simply contains attributes and behavior, and another class is required in order to check whether the object-defining class is working properly. This class is called an ***application class*** or a ***driver class***. The *Invoice class* is used to define invoice objects, but a program called *InvoiceApp.java* that contains the main method is needed to drive the entire application, making it useful for processing invoices. This application class follows the input-process-output pattern of most procedural applications. That is because application class is indeed very procedural.

Object Interface

In an OO application the **object interface** can be described as a set of methods in a class, use to operate the object (instance variables). Each method in a class has a **method signature**, which refers to everything a programmer needs to know about how to use the method (the name, visibility, return type, and parameters).

For example, part of the interface of an Invoice object could be its constructor. The construct's signature looks like this:

```
public Invoice ( String number, double total, int type)
```

The Invoice object requires three variables in order to create/instantiate an invoice object: the invoice number, invoice total, and invoice type). Another method in the Invoice class is setInvoiceTotal() its signature is:

```
public void setInvoiceTotal(double invTotal, int invType)
```

The method's signature signify that the method is public, does not have return data, and requires an invoice total and invoice type to work properly.

Application:

✓ Understanding the Problem

Continue building the application described in the previous section by creating invoice application that produces either discounted or non-discounted sales invoice. The application should know how to interface with the Invoice objects in order to create and manipulate them. The application class should accept information such as invoice number, invoice total and discount type. The application creates Invoice objects with the appropriate discount applied and display the necessary information about the invoice.

✓ Planning the Application

Develop an application class, that creates an invoice object and output invoice information using the Invoice class. The flowchart for the InvoiceApp.java is shown in figure 4.6. This application class is going to use the variables for input and output. It make use of loops to allow the user to generate several invoices without having to restart the application each time. Inside the loop, the application requests the data from user, creates an Invoice object and produces the output.

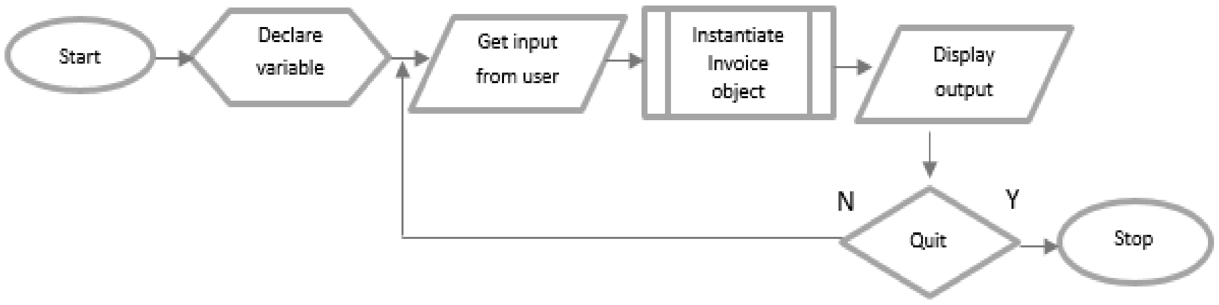


Figure 7-6 Flowchart of the Application Class

✓ Writing the Code

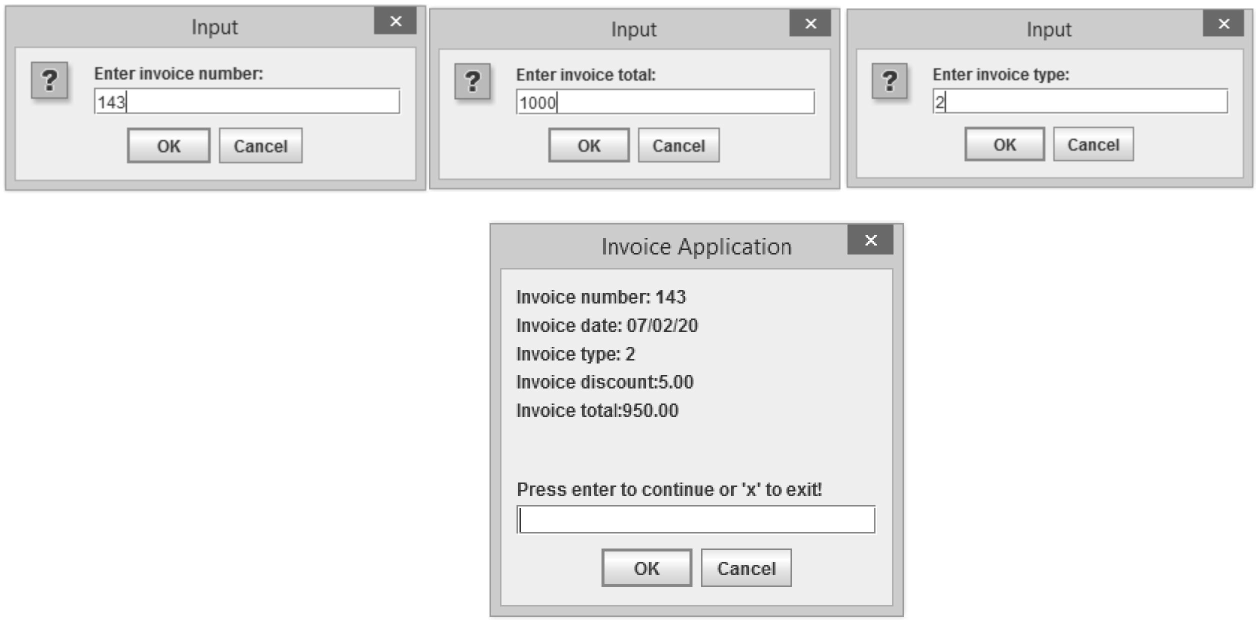
Figure 7-7 provides the program code for InvoiceApp.java.

```
/*
 *Figure 7-7
 * Filename : InvoiceApp.java
 * Date Created : 06/22/2020
 * Purpose : Application class for creating Invoice objects
 */
```

```
import javax.swing.JOptionPane;
 import java.util. Formatter;

 public class InvoiceApp{
       public static void main(String[] args){
            //declare and initialize variables
            String choice="", strNumber ="", strTotal= "", strType="",
strOutput = "";
            int type = 0;
            double total = 0;

            //declare invoice object
            Invoice invInvoice= null;

            while( !(choice.equalsIgnoreCase("x"))){
                  //get user input
                  strNumber = JOptionPane.showInputDialog(null,"Enter
invoice number:");
                  strTotal = JOptionPane.showInputDialog(null,"Enter
invoice total:");
                  //convert String to Double
                  total = Double.parseDouble(strTotal);

                  strType = JOptionPane.showInputDialog(null,"Enter
invoice type:");
                  //convert String to Integer
                  type = Integer.parseInt(strType);

                  //create Invoice object
                  invInvoice = new Invoice(strNumber, total, type);

                  //prepare for output
                  strOutput = String.format("Invoice number: %s\nInvoice
                  date: %tD\nInvoice type: %d\nInvoice
                  discount:%.2f\nInvoice total:%.2f \n\n\nPress enter to
                  continue or 'x' to exit!",
                  invInvoice.getInvoiceNumber(),
                  invInvoice.getInvoiceDate(),
                  invInvoice.getInvoiceType(),
                  invInvoice.getDiscountPercent()*100,invInvoice.getInvoi
                  ceTotal() );

                  //display output
                  choice = JOptionPane.showInputDialog(null,
strOutput,"Invoice Application",
                  JOptionPane.PLAIN_MESSAGE);

            }//end while
       }//end main
```

✓ Compiling and Testing

After InvoiceApp.java is coded, it should be saved, compiled and tested. If error occurs try to locate the errors and debug it. Figure 4.8 shows the sample output of InvoiceApp.java



```
Enter invoice number:
143

Enter invoice total:
1000

Enter invoice type:
2

Invoice number: 143
Invoice date: 07/02/20
Invoice type: 2
Invoice discount:5.00
Invoice total:950.00


Press enter to continue or 'x' to exit!
```

Figure 7-8 Sample output of invoiceApp.java

✓ Deploy the Application

After the compile and test phases are completed satisfactory, the application is ready to be deployed and used by the user.

The Composition Relationship between Classes

Composition is a typical component of an object-defining class which specifies the class is made of other objects. The **composition** is used when one instance variable is reference to an object. Figure 4.9 illustrates example of composition relationship between the InvoiceWithProduct class and the Product class using UML class diagram.

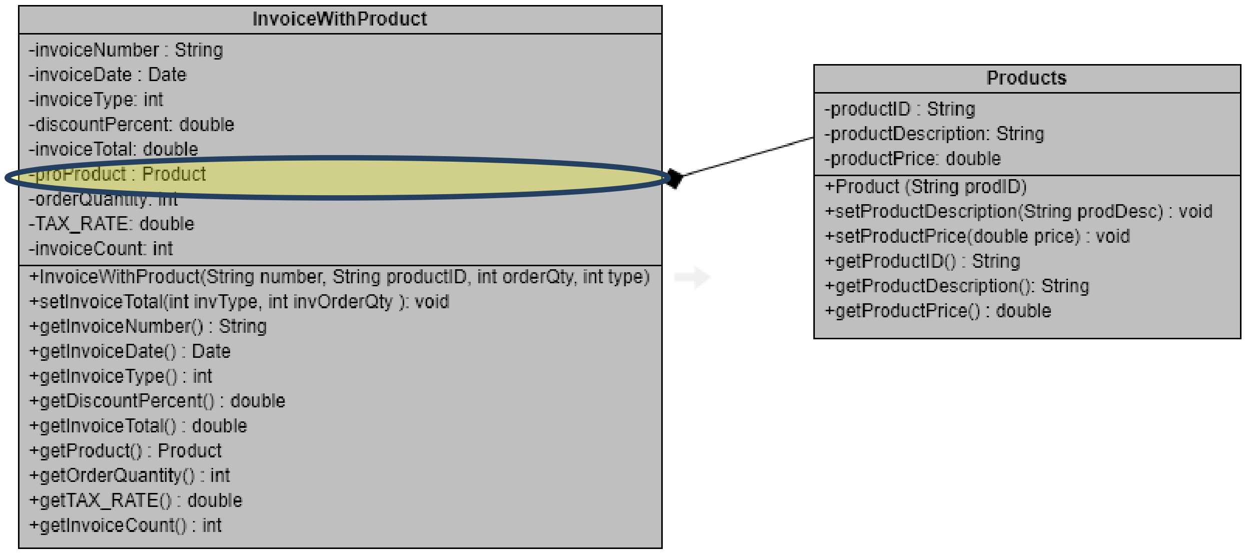


Figure 7-9 Class diagram of InvoiceWithProduct and Products class showing composition

This kind of relationship is also called **has-a relationship** which implies that one class “has a/an” object from another class. In figure 4.10 illustrates a composition. Example Person **has-a** Leg, Person **has-a** Hand

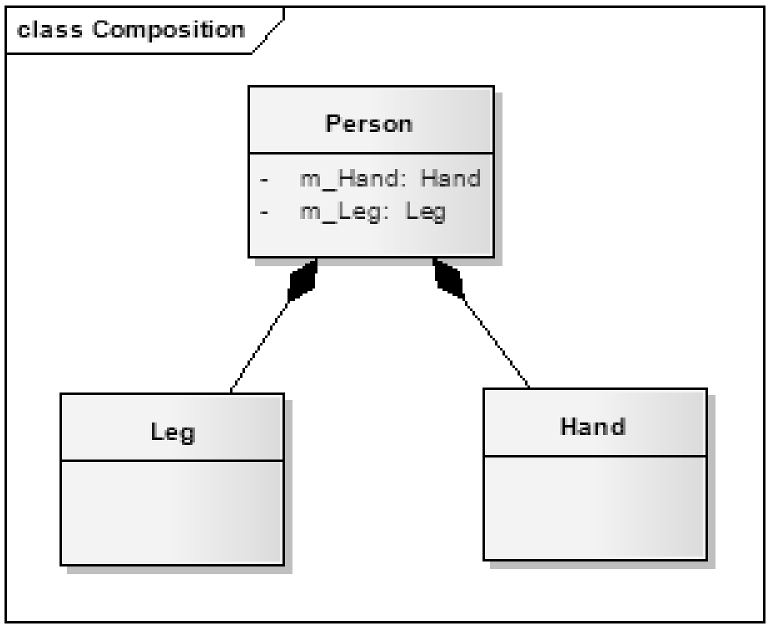


Figure 7-10 Composition

Activities and Exercises:

1. Describe/Define in your own word, the meaning of real-world classes, real-world objects, software classes and objects.
2. Explain the difference between structured and object oriented application development methodologies.
3. Using UML class diagram, create/draw a diagram for Student/Instructor/Employee capturing the significant attributes and behaviors.
4. Code/Create an object defining classes for your selected class.
5. Code/Create an applications that make use of your objects defining class following the input-process-output pattern.
6. Draw a UML class diagram that illustrates the composition relationship between classes.

QUIZ IS IT!

True or False. Write ***True*** if the Statement is Correct; otherwise, write ***False***.

_________1. Objects have two major characteristics: *attributes* and *behaviors.*

_________2. *Information hiding* refers to the process of concealing information about how an object is implemented within the class.

_________3. *String* is an example of abstract data type.

_________4. Your class name and its constructor name should be different.

_________5. An object-defining class usually contains a main method.

_________6. A good analogy for an application class is a blueprint.

_________7. A class diagram uses "-" to indicate private and "+" to indicate public access.

_________8. *Mutator methods* are also called getter methods.

_________*9.* An application class is also called a *driver class.*

_________10. An object's behavior are the characteristics that define an object.

_________11. An object attributes are the actions or operations that define what an object does.?

_________12. The structured application development is a methodology for developing application that focus on how data flow from one application to another and on how the system entities are related by system events.

_________13. The object-oriented application development is a methodology for developing applications that focuses on objects in the system and how they interact.

Multiple Choice – Encircle the letter of the correct answer.

14. The code declared as public Invoice in an Invoice class is called a(n) ________.
    a. Mutator   c. Accessor
    b. Constructor   d. Interface
15. A class in an OO application that has a main method is called __________ class.
    a. Constructor   c. Driver
    b. Mutator   d. Static

16. In UML class diagram, which of the following is contained within the upper section?
    a. Data member / Attributes
    b. Method member / Behavior
    c. Class name
    d. None of the above
17. In visibility, which of the following symbol is used for protected?
    a. +
    b. –
    c. #
    d. $
18. On the given choices below, which is an example of abstract data type (ADT)?
    a. int
    b. float
    c. byte
    d. String
19. What keyword is always used with accessor methods?
    a. private
    b. main
    c. new
    d. return
20. On the given choices below, which is not an abstract data type (ADT)?
    a. JFrame
    b. Date
    c. boolean
    d. Scanner

LESSON 8
OBJECT-ORIENTED APPLICATION DEVELOPMENT PART 2

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. learn and use the principles of polymorphism and inheritance in application development;
2. discover more about java's predefined classes and explore how inheritance is used in programmer-defined classes;
3. learn the importance of overloading, overriding, static and dynamic binding in object oriented applications; and
4. develop skills in using inheritance and polymorphism in object oriented applications.

READY. GET SET. DISCUSS!

Inheritance

Inheritance is a special relationship between classes where one class has a direct access to the public members of another class. In java there are numerous predefined classes that inherits attributes and behaviors from other predefined classes. In the following section both predefined classes and programmer-defined classes will be used to demonstrate Inheritance and how it is implemented.

Inheritance and Java's Predefined Classes

Predefined classes in java are arranged in a hierarchy. The **inheritance hierarchy** consist of parent and child classes where the child class is in lower hierarchy and the parent is in the upper hierarchy. The child class can receive both the attributes (variables) and behavior (methods) of its parent class. The classes in an inferior hierarchy can be called as the **child class,** subclass**, extended class, or derived class** while the classes in the superior hierarchy can be called as **parent class, super class, or the base class**. The open arrows (the outlined arrowhead) in Figure 8-1 indicate that the subclass inherits from its super class.

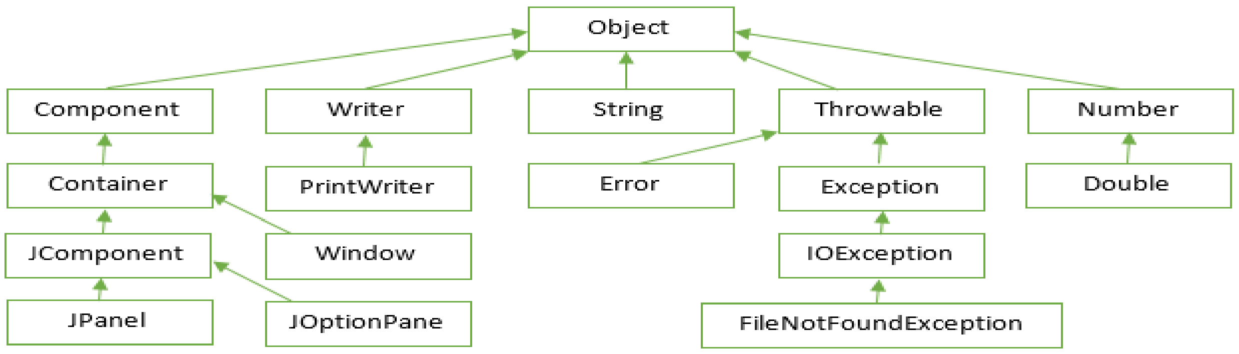


Figure 8-1 Partial inheritance hierarchy of predefined java classes in java

Inheritance can also be defined as a process that create or generates new classes by extending it in an existing class. The inheritance implements the **is-a** or **is-an** relationship between classes. Referring to figure 8-1 the object with an abstract data type of **Double** “is-a” **Number** and a **Number** object “is-an” **Object** object as well.

In UML notation, inheritance uses a solid line followed by a hollow arrowhead from the child class to its parent class. To show hierarchy, the parent class is typically drawn on uppermost of its child classes. See figure 8-2.

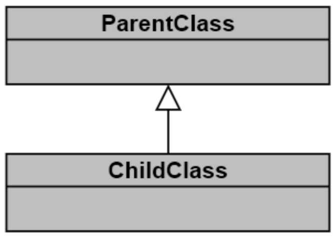
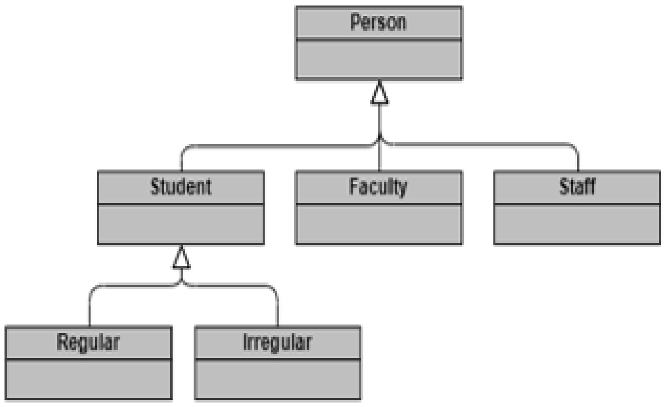


Figure 8-2 UML notation of inheritance

Inheritance and Java’s Programmer-defined Classes

The heart of object oriented application development is the creation of object-defining classes by the programmer, classes that usually model real-world objects. Inheritance is needed when one general type of object can be broken down into more specific types of object, where each specific type have some of their own unique attributes and behaviors. To define the derived class in java, the **“extends”** keyword is placed along with the name of its base class. The figure 8-3 shows how to create/define a subclass.



```
//defining the parent class
public class Person {.......}

//defining the subclass
public class Student extends Person {...}
public class Faculty extends Person {...}
public class Staff extends Person {...}
public class Regular extends Student {...}
public class Irregular extends Student {...}
```

Figure 8-3 Defining a subclass using the keyword extends

Generalization and Specialization

In creating a programmer-defined classes we can use two approach to model a real-word objects. We can use generalization or specialization. The process of extracting common characteristics from two or more classes and combining it into a generalized superclass is called **Generalization**. On the other hand **Specialization** is the reverse process of Generalization which means creating a new subclass from an existing class. Figure 8-4 illustrates the two approach.

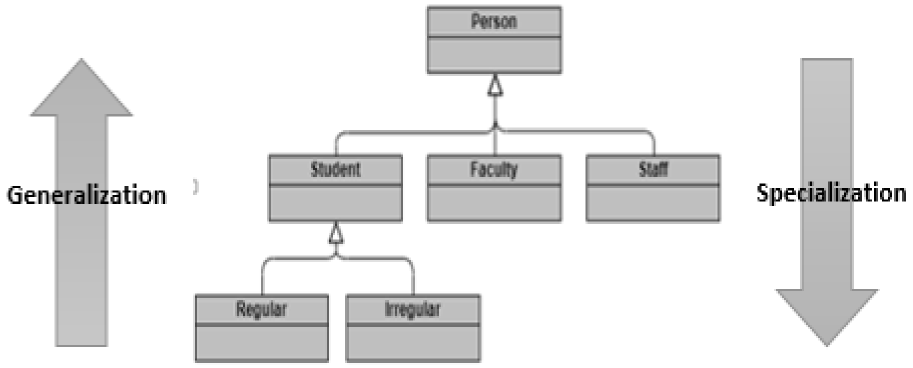


Figure 8-4 Generalization and Specialization

To give a working example let us create two object-defining classes. The **Person** class as the parent class and **Student** class as the child class. As shown in figure 8-5 the **Person** class has 5 attributes (idNo, lastName, middleName,givenName and address), **Student** has a single attribute (collegeClass) followed by the constructor, setter method and the getter methods respectively. The arrow between the classes indicates that the **Student** class is inheriting both the attributes and behavior of the **Person** class.

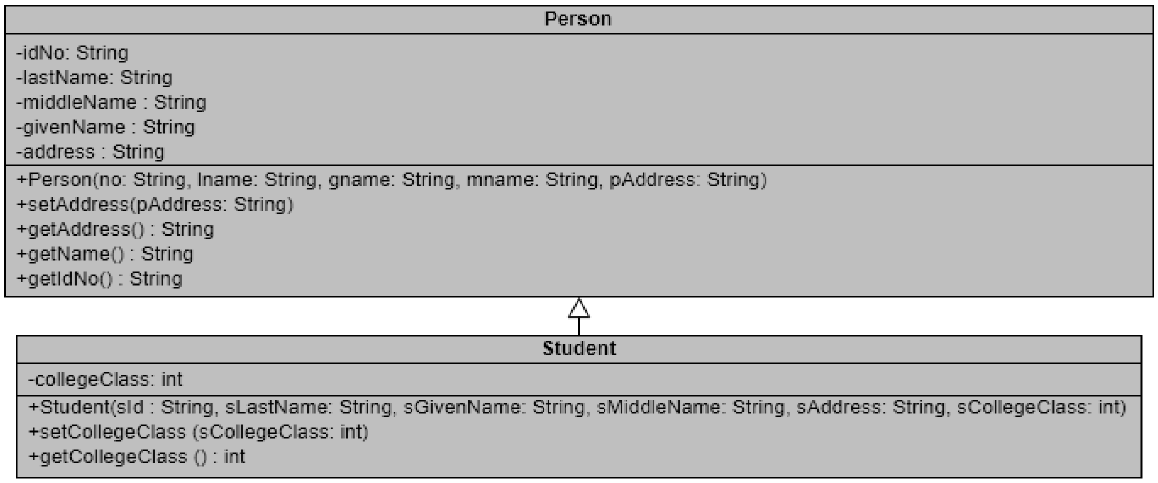

Person
-idNo: String
-lastName: String
-middleName : String
-givenName : String
-address : String
+Person(no: String, lname: String, gname: String, mname: String, pAddress: String)
+setAddress(pAddress: String)
+getAddress() : String
+getName() : String
+getIdNo() : String
Student
-collegeClass: int
+Student(sId : String, sLastName: String, sGivenName: String, sMiddleName: String, sAddress: String, sCollegeClass: int)
+setCollegeClass (sCollegeClass: int)
+getCollegeClass () : int

Figure8-5 UML Class diagram of Person and Student class

The following figures contain the program code for *Person.java (*figure 8-6), *Student.java* (figure 8-7) and the driver class *StudentApp*.java (figure 8-8).

TRY IT! `Person.java`

```
/*
 * Figure 8-6
 * Filename : Person.java
 * Date Created : 06/22/2020
 * Purpose  : to define Person objects
 */

public class Person{
     //declare instance variables
     private String idNo, lastName, middleName, 
givenName,address;

     //define constructor
     public Person(String no, String lname, String gname, 
String mname, String pAddress){
          idNo = no;
          lastName = lname;
          givenName = gname;
          middleName =mname;
          address = pAddress;
     }
```

```
      //public methods
      public void setAddress(String pAddress){
           address = pAddress;
      }
      public String getAddress(){
           return address;
      }
      public String getName(){
           String fullName = lastName +", "+givenName+ "
"+middleName;
           return fullName;
      }
      public String getIdNo(){
           return idNo;
      }
} // end class
```

Figure 8-6 Code for Person.java

TRY IT! `Student.java`

```
//Purpose  : to define Student objects
 public class Student extends Person {
     //declare instance variables
     private int collegeClass;
     //define constructor
     public Student(String sId, String sLastName, String
sGivenName, String sMiddleName, String sAddress, int
sCollegeClass){
          super(sId,sLastName,sGivenName,sMiddleName,sAddress);
          collegeClass = sCollegeClass;
     }
     //public methods
     public void setCollegeClass(int sCollegeClass){
          collegeClass = sCollegeClass;
     }
     public int getCollegeClass(){
          return collegeClass;
     }
 } // end class
```

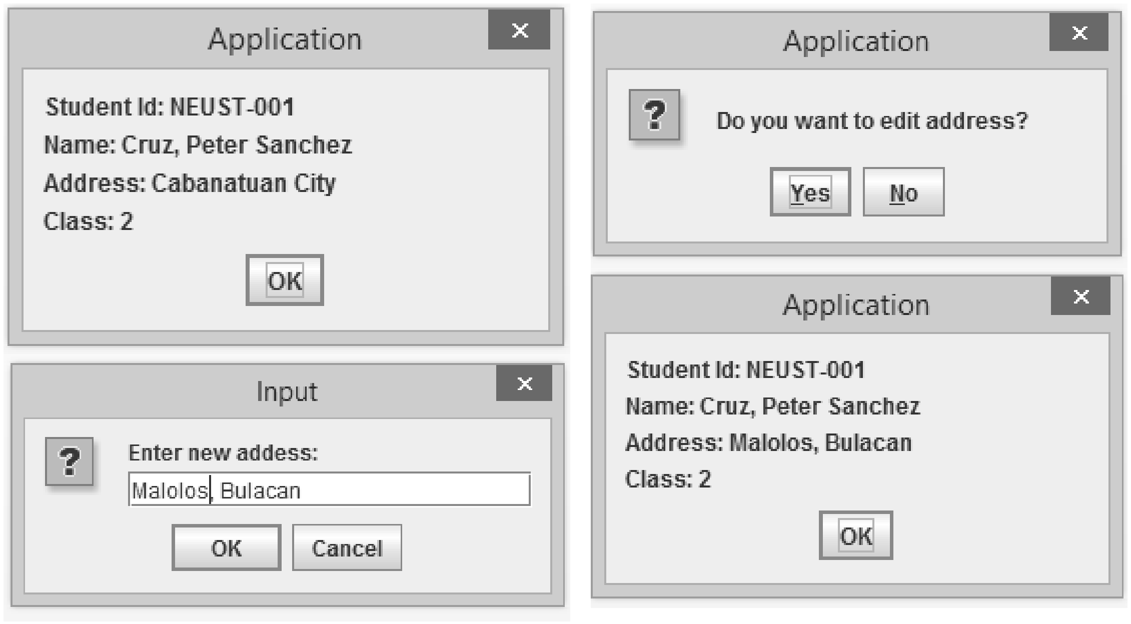
Figure 8-7 Code for Student.java

In this figure the keyword **super** is used. The “**super**” keyword denotes that it access the constructor method of its base class (**Person Class**).

The Application Class

After creating the two classes (**Person** and **Student**) the application class or the driver class is needed to perform the tasks required by the application using the objects created from **Person** and **Student**. Figure 8-8

```
//Purpose  : Application class for creating Student objects

import javax.swing.JOptionPane;
import java.util.Formatter;

public class StudentApp {
      public static void main(String[] args){
            String newAddress, output, response;

            //create Student Object
            Student student = new Student("NEUST-001", "Cruz", "Peter",
"Sanchez", "Cabanatuan City", 2);
            output = String.format("Student Id: %s\nName: %s\nAddress: %s\nClass:
%d\n",

student.getIdNo(),student.getName(),student.getAddress(),student.getCollegeClass(
));
      JOptionPane.showMessageDialog(null,output,"Application",JOptionPane.PLAIN_M
ESSAGE);
            //modifying Student Object (address)
            if (JOptionPane.showConfirmDialog(null, "Do you want to edit
address?", "Application",

JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
                  // yes option
                  newAddress = JOptionPane.showInputDialog("Enter new addess: ");
                  student.setAddress(newAddress);
                  output = String.format("Student Id: %s\nName: %s\nAddress:
%s\nClass: %d\n",

student.getIdNo(),student.getName(),student.getAddress(),student.getCollegeClass(
));
      JOptionPane.showMessageDialog(null,output,"Application",JOptionPane.PLAIN_M
ESSAGE);
            }
            else {
                  // no option
                  System.exit(0);
            }
      }//end main
 } // end class
```



```
Student Id: NEUST-001
Name: Cruz, Peter Sanchez
Address: Cabanatuan City
Class: 2
```

Do you want to edit address?

Enter new addess:

Malolos, Bulacan

```
Student Id: NEUST-001
Name: Cruz, Peter Sanchez
Address: Malolos, Bulacan
Class: 2
```

Figure 8-9 Sample output of the StudentApp.java

Application:

To reinforce the concepts of object-oriented application development and inheritance let us develop a simple application using the following problem statement.

✓ Understanding the Problem

Create an application for a bank that offers both checking and savings account to its customer. The customer can open checking account, savings account, or both. When customer open a checking account, a monthly service fee is immediately assessed depending on the amount of the original deposit. When customer open a savings account, a month's interest is paid immediately depending on the amount of original deposit.

✓ Planning the Application

In planning this application three tasks are involved. First, identify the object-defining classes required by the application. Second, draw/sketch/create a class diagram for these classes. Third, flowchart for the application class.

- ✓ *Identify possible class*
  The first step in planning an object oriented application is to identify classes that are need. We can select the several nouns from this problem such as: bank, bank account, checking account, savings account, customer, deposit, interest, and service fee that can be used to model objects in the system. The verbs in

the problem statement such as assess fee and pay interest could be used as the method.

- ✓ *Create class diagram*
  The scope of this sample application is somewhat limited so we can focus on four kinds of objects: bank account, savings account, checking account and customer. Let us assume that savings account and checking account are specialized type of bank account. Figure 8-10 display the class diagram for four identified classes and its relationship.

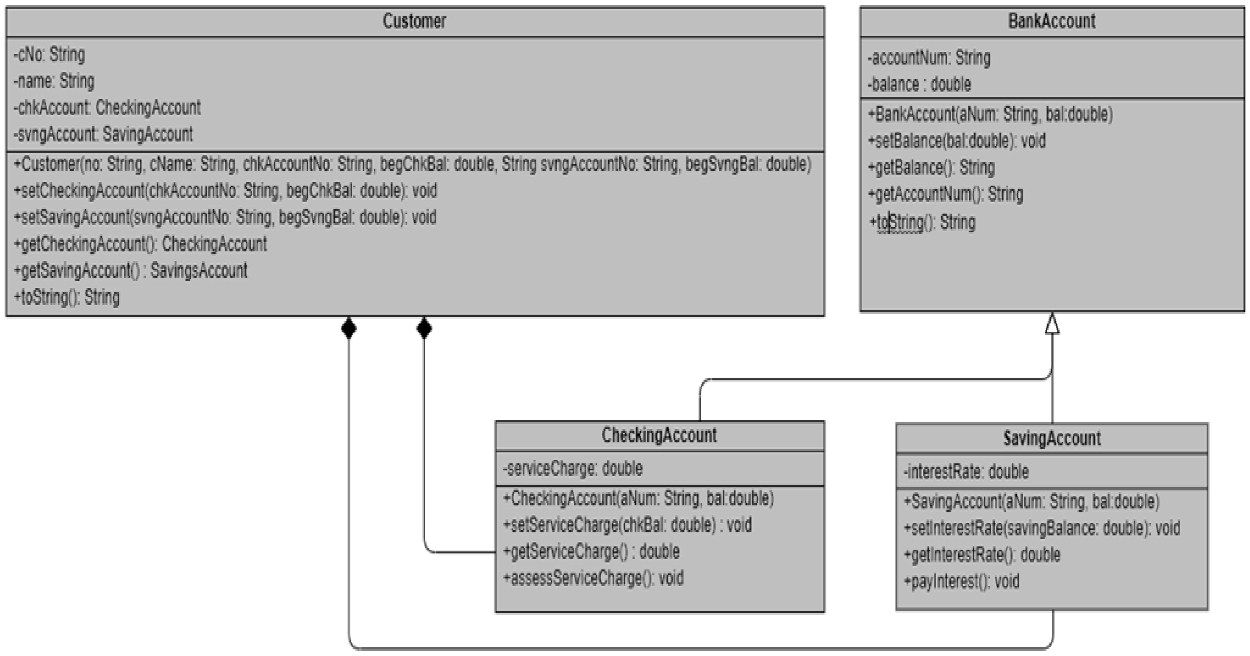

Customer
-cNo: String
-name: String
-chkAccount: CheckingAccount
-svngAccount: SavingAccount
+Customer(no: String, cName: String, chkAccountNo: String, begChkBal: double, String svngAccountNo: String, begSvngBal: double)
+setCheckingAccount(chkAccountNo: String, begChkBal: double): void
+setSavingAccount(svngAccountNo: String, begSvngBal: double): void
+getCheckingAccount(): CheckingAccount
+getSavingAccount() : SavingsAccount
+toString(): String
BankAccount
-accountNum: String
-balance : double
+BankAccount(aNum: String, bal:double)
+setBalance(bal:double): void
+getBalance(): String
+getAccountNum(): String
+toString(): String
CheckingAccount
-serviceCharge: double
+CheckingAccount(aNum: String, bal:double)
+setServiceCharge(chkBal: double) : void
+getServiceCharge() : double
+assessServiceCharge(): void
SavingAccount
-interestRate: double
+SavingAccount(aNum: String, bal:double)
+setInterestRate(savingBalance: double): void
+getInterestRate(): double
+payInterest(): void

Figure 8-10 Class Diagram of the four classes (Customer, BankAccount, CheckingAccount and SavingAccount)

In the above figure, composition and inheritance are used to show relationship among these classes. To describe the relationship, **has-a** (composition) and **is-a** (inheritance) is used. The Customer **has-a** CheckingAccount. The Customer **has-a** SavingAccount. The CheckingAccount **is-a** BankAccount. The SavingAccount **is-a** BankAccount.

✓ *Flowchart for the application class*

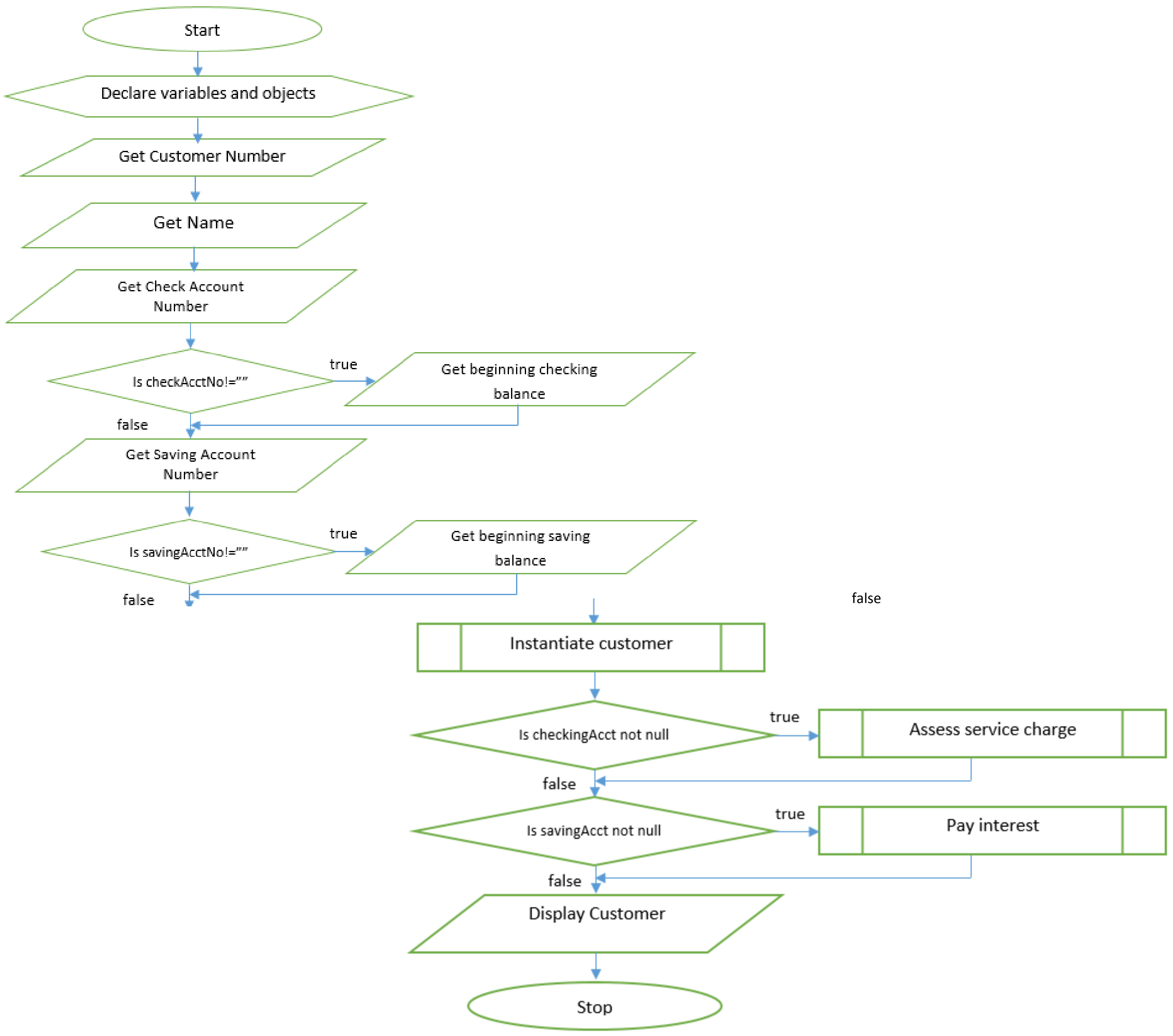


Figure 8-11 Flowchart for BankApp.java

✓ Writing the Code

Figures 8-12 through 8-15 contains code for each classes and 8-16 contains the driver class.

```
//Filename : Customer.java

  import java.util.Formatter;

 public class Customer {
      //declare instance variables
      private String cNo, name;
      private CheckingAccount chkAccount;
      private SavingAccount svngAccount;

      //define constructor
      public Customer(String no, String cName, String chkAccountNo, double 
begChkBal,
                    String svngAccountNo, double begSvngBal){
            cNo = no;
            name = cName;
            setCheckingAccount(chkAccountNo, begChkBal);
            setSavingAccount(svngAccountNo,begSvngBal);
      }
      //public methods
      public void setCheckingAccount(String chkAccountNo, double begChkBal){
            if(! chkAccountNo.equals("")){
                  chkAccount = new CheckingAccount(chkAccountNo,begChkBal);
            }
      }
      public void setSavingAccount(String svngAccountNo, double begSvngBal){
            if(! svngAccountNo.equals("")){
                  svngAccount = new SavingAccount(svngAccountNo,begSvngBal);
            }
      }
      public CheckingAccount getCheckingAccount(){
            return chkAccount;
      }
      public SavingAccount getSavingAccount(){
            return svngAccount;
      }
      public String toString(){
            String strFormat, strCustomer;
            strFormat = "Customer Number: %s\nName: %s\nAccount Type: 
%s\nAccount Type: %s\n";
            strCustomer = String.format(strFormat, cNo, name, chkAccount, 
svngAccount);
            return strCustomer;
      }
 } // end class
```

Figure 8-12 Code for Customer class

```
//Filename : BankAccount.java

  import java.util.Formatter;

 public class BankAccount{
     //declare instance variables
     private String accountNum;
     private double balance;

     //define constructor
     public BankAccount(String aNum, double bal){
          accountNum = aNum;
          balance = bal;
     }

     //public methods
     public void setBalance(double bal){
          balance = bal;
     }

     public double getBalance(){
          return balance;
     }
     public String getAccountNum(){
          return accountNum;
     }

     public String toString(){
          String strFormat,strAccountInfo, typeOfAccount="";
          if(this instanceof CheckingAccount)
               typeOfAccount="Checking";
          else
               typeOfAccount="Saving";

          strFormat = "%s Account Number: %s Balance: $%.2f";
          strAccountInfo = String.format(strFormat,
typeOfAccount, accountNum, balance);
          return strAccountInfo;
     }
 } // end class
```

Figure 8-13 Code for BankAccount class

```
//Figure 8-14
//Application of inheritance (CheckingAccount extends Bank
Account)
//Filename : CheckingAccount.java

 public class CheckingAccount extends BankAccount{

     //declare instance variables
     private double serviceCharge;

     //define constructor
     public CheckingAccount(String aNum, double bal){
          super(aNum,bal);
          setServiceCharge(bal);
     }

     //public methods
     public void setServiceCharge(double chkBal){
          if(chkBal<100)
               serviceCharge =10;
          else
               serviceCharge =0;
     }

     public double getServiceCharge(){
          return serviceCharge;
     }

     public void assessServiceCharge(){
          double currentBalance = getBalance();
          double newBalance = getBalance()-serviceCharge;
          setBalance(newBalance);
     }
```

Figure 8-14 Code for CheckingAccount class

```
//Figure 8-15
//Another application of inheritance (SavingAccount extends
BankAccount)
//Filename : SavingAccount.java

public class SavingAccount extends BankAccount{
     //declare instance variables
     private double interestRate;

     //define constructor
     public SavingAccount(String aNum, double bal){
          super(aNum,bal);
          setInterestRate(bal);
     }

     //public methods
     public void setInterestRate(double savingBalance){
          if(savingBalance>=100)
               interestRate =0.05;
          else
               interestRate =0;
     }

     public double getInterestRate(){
          return interestRate;
     }

     public void payInterest(){
          double newBalance, currentBalance = getBalance();

          if(currentBalance>=100){
               newBalance = currentBalance *(1+interestRate);
               setBalance(newBalance);
          }
     }
 } // end class
```

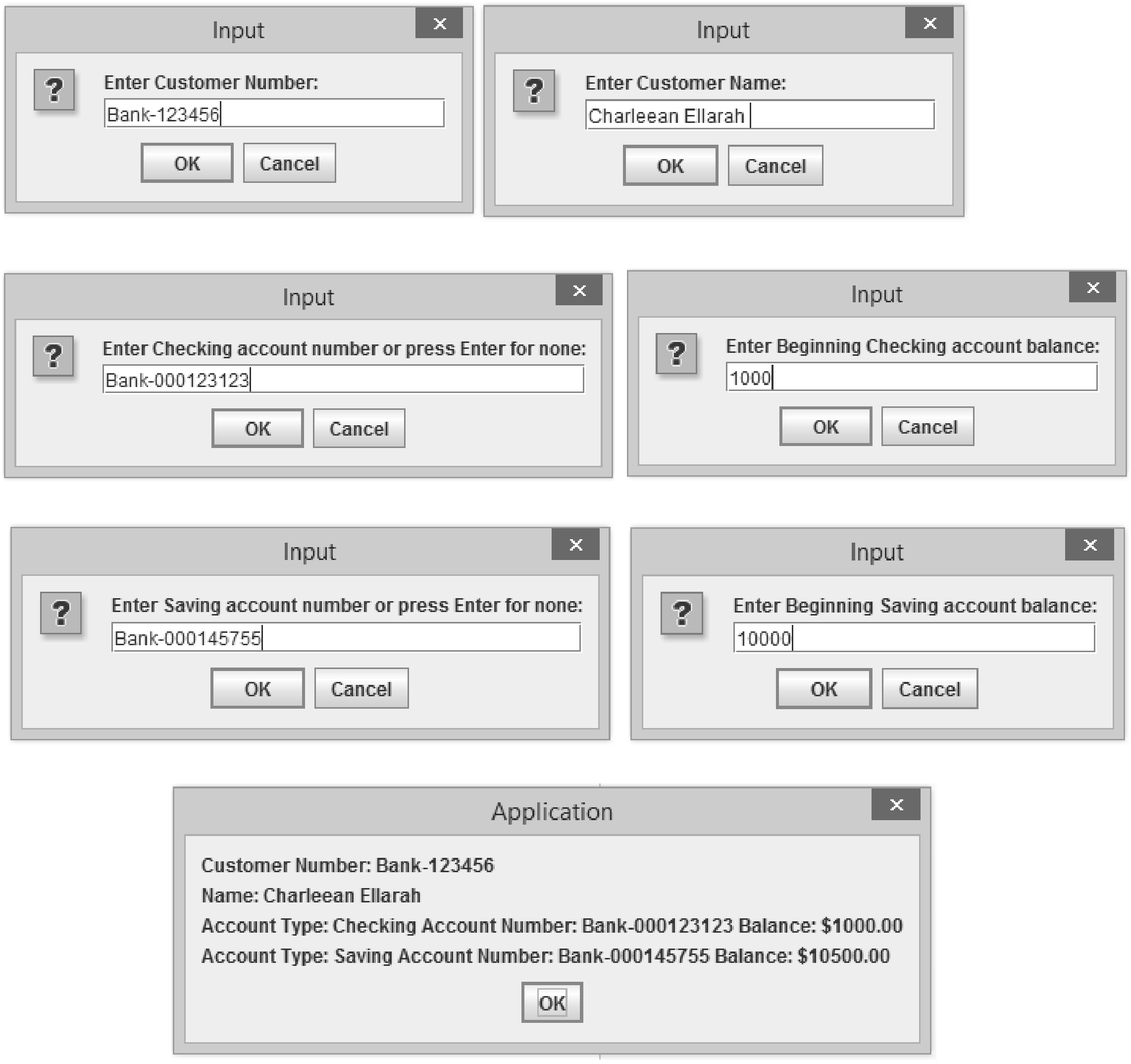
Figure 8.15 Code for SavingAccount class

```
//Figure 8-16
//This is the driver class which contain the main method.
//Filename : BankApp.java

import javax.swing.JOptionPane;

 public class BankApp {
     public static void main(String[] args){
            //declaration of objects and variables
            String customerNo, strName, strChkAccNum, strSvngAccNum, strBegChkBal, strBegSvngBal;
            double begChkBal=0.0, begSvngBal=0.0;
            Customer aCustomer;

            //input
            customerNo = JOptionPane.showInputDialog("Enter Customer Number:");
            strName = JOptionPane.showInputDialog("Enter Customer Name:");
            strChkAccNum = JOptionPane.showInputDialog("Enter Checking account number or press
Enter for none:");
            if(! strChkAccNum.equals("")){
                  strBegChkBal = JOptionPane.showInputDialog("Enter Beginning Checking account
balance:");
                  begChkBal = Double.parseDouble(strBegChkBal);
            }
            strSvngAccNum = JOptionPane.showInputDialog("Enter Saving account number or press
Enter for none:");
            if(! strSvngAccNum.equals("")){
                  strBegSvngBal = JOptionPane.showInputDialog("Enter Beginning Saving account
balance:");
                  begSvngBal = Double.parseDouble(strBegSvngBal);
            }

            //processing
            aCustomer = new Customer(customerNo, strName, strChkAccNum, begChkBal,
strSvngAccNum,begSvngBal);

            if(aCustomer.getCheckingAccount() != null){
                  aCustomer.getCheckingAccount().assessServiceCharge();
            }

            if(aCustomer.getSavingAccount() != null){
                  aCustomer.getSavingAccount().payInterest();
            }

            //output

      JOptionPane.showMessageDialog(null,aCustomer.toString(),"Application",JOptionPane.PLAIN_MES
SAGE);

      }//end main
 } // end class
```

Figure 8.16 Code for the driver class (BankApp)

✓Compiling and Testing

One of the major differences in this object oriented application when compared to traditional procedural application is that the program code is split among classes, most of which are designed to model system objects. For this application to run properly, each class must be located and stored in the same directory and each must be compiled individually. All compile-time error must be corrected for the application to run. Figure 8-17 display the sample output of the application.



Enter Customer Number:
Bank-123456

Enter Customer Name:
Charleean Ellarah

Enter Checking account number or press Enter for none:
Bank-000123123

Enter Beginning Checking account balance:
1000

Enter Saving account number or press Enter for none:
Bank-000145755

Enter Beginning Saving account balance:
10000

```
Customer Number: Bank-123456
Name: Charleean Ellarah
Account Type: Checking Account Number: Bank-000123123 Balance: $1000.00
Account Type: Saving Account Number: Bank-000145755 Balance: $10500.00
```

Figure 8-17 Sample output of the BankApp

✓ Deploy the Application

After compiling and testing phase are complete. The application will be now ready for deployment.

Polymorphism

The term polymorphism comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*). In an object-oriented context, **polymorphism** refers to the ability of a single method name to be used in many different ways with in single class or within multiple classes. The implementation of polymorphism can be done thru overriding and overloading of methods. This mechanism includes the early binding (static) and the late binding (dynamic binding).

We can define the term **Polymorphism** as the capability of sole object to have multiple forms or take different actions or forms. In OOP, the polymorphism occurs when a single method name is use to perform different task. The succeeding topics show how polymorphism is applied by overloading and overriding the constructors and methods.

Overloading

Overloading is kind of polymorphism that occurs when a single method name is used more than once. We can overload the constructors and methods in same class provided our method contains different argument lists or parameter lists. The following figure illustrates overloading using class a diagram.

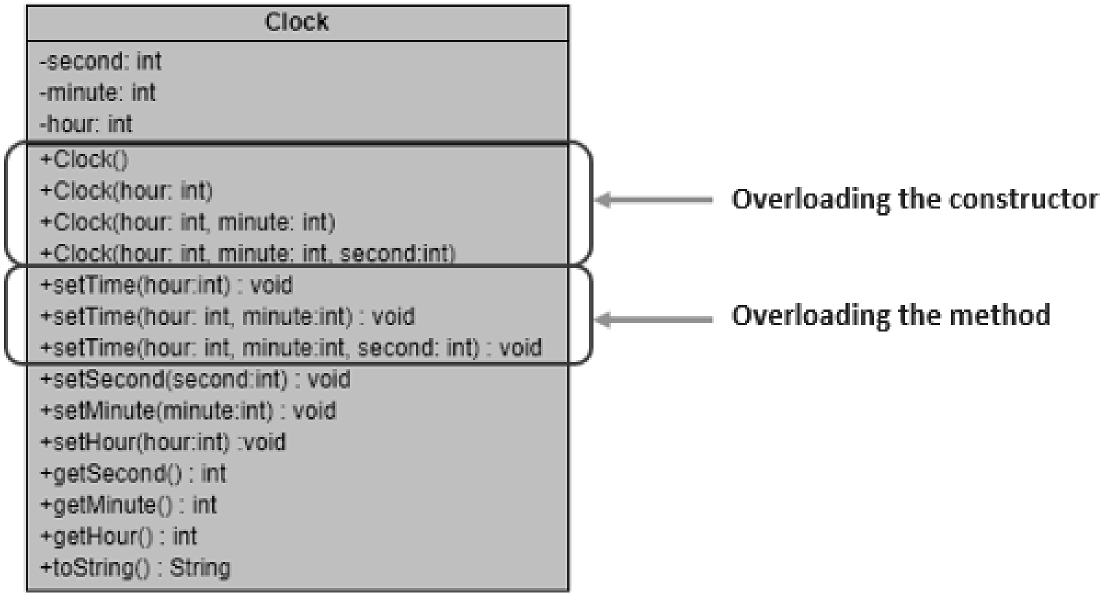

Clock
-second: int
-minute: int
-hour: int
+Clock()
+Clock(hour: int)
+Clock(hour: int, minute: int)
+Clock(hour: int, minute: int, second:int)
+setTime(hour:int) : void
+setTime(hour: int, minute:int) : void
+setTime(hour: int, minute:int, second: int) : void
+setSecond(second:int) : void
+setMinute(minute:int) : void
+setHour(hour:int) :void
+getSecond() : int
+getMinute() : int
+getHour() : int
+toString() : String
Overloading the constructor
Overloading the method

Figure 8-18 Class diagram for Clock illustrating polymorphism (overloading)

Overriding

Overriding is another form of polymorphism which one method name with a given argument list or parameter list can be used in different classes within the inheritance hierarchy. The method that contains similar name and with similar parameter or argument list can perform entirely different thing in each of the different classes. The key concept is that a method defined in a subclass will override(redefine) a method in the superclass. The following figure shows overriding.

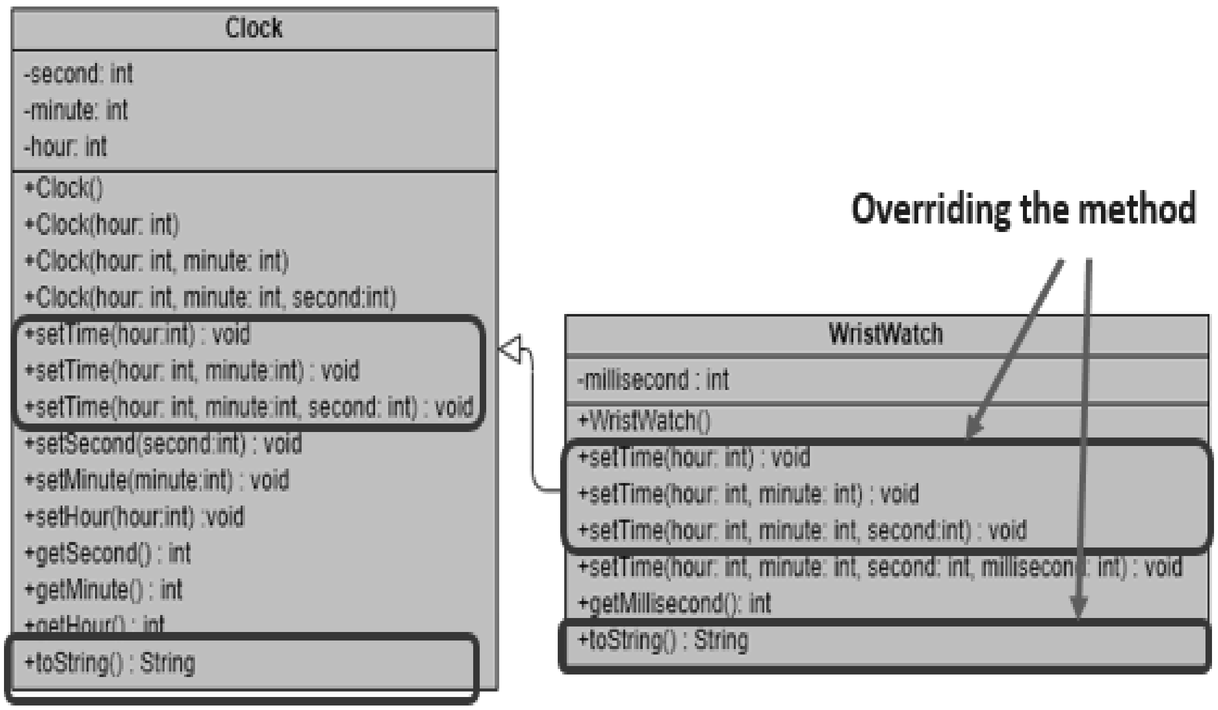

Clock
-second: int
-minute: int
-hour: int
+Clock()
+Clock(hour: int)
+Clock(hour: int, minute: int)
+Clock(hour: int, minute: int, second:int)
+setTime(hour:int) : void
+setTime(hour: int, minute:int) : void
+setTime(hour: int, minute:int, second: int) : void
+setSecond(second:int) : void
+setMinute(minute:int) : void
+setHour(hour:int) :void
+getSecond() : int
+getMinute() : int
+getHour() : int
+toString() : String
Overriding the method
WristWatch
-millisecond : int
+WristWatch()
+setTime(hour: int) : void
+setTime(hour: int, minute: int) : void
+setTime(hour: int, minute: int, second:int) : void
+setTime(hour: int, minute: int, second: int, millisecond: int) : void
+getMillisecond(): int
+toString() : String

Figure 8-18 Class diagram for Clock and WristWatch class illustrating Overriding

The Static/Early and Dynamic /Late Binding

Early Binding

This binding occurs and done at compile-time where all the final, private and static methods are linked together. This includes the binding of method calls and method definitions during compile-time. In this binding the program execution will be faster and method overloading is a concrete example. Illustration is shown in figure 8-19 the early binding in java.

```
//Figure 8-19
//Polymorphism
//Filename : EarlyBinding.java

public class EarlyBinding {
    public static class Superclass {
        static void print()
        {
            System.out.println("print in Superclass.");
        }
    } //end subclass

    public static class Subclass extends Superclass {
        static void print()
        {
            System.out.println("print in Subclass.");
        }
     } //end subclass

     public static void main(String[] args)
     {
        Superclass objectA = new Superclass();
        Superclass objectB = new Subclass();
        A.print();
        B.print();
    } //end main

}//end class
```

```
Output:
            print in Superclass.
            print in Superclass.
```

Figure 8-19 Early binding sample

Late Binding

The *late binding* or *dynamic binding* occurs and done during run-time where actual objects are used to bind or link the methods. The appropriate example of dynamic binding would be the overriding. Once an object is created, methods calls and method definition can be linked. Figure 8-20 shows schematic of dynamic binding and figure 8-21 sample code of dynamic binding (overriding).d

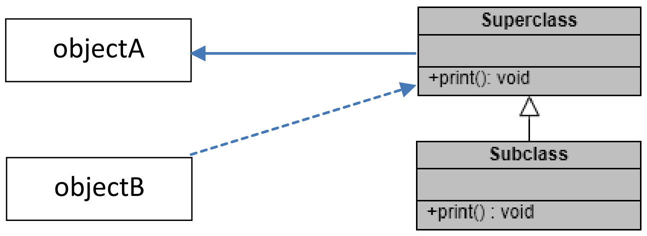


Figure 8-20 Schematic of dynamic binding

```
//Figure 8-20
//Polymorphism
//Filename : LateBinding.java

public class EarlyBinding {
    public static class Superclass {
        void print()
        {
            System.out.println("print in Superclass.");
        }
    } //end subclass

    public static class Subclass extends Superclass {
        @Override
        void print()
        {
            System.out.println("print in Subclass.");
        }
    } //end subclass

    public static void main(String[] args)
    {
        Superclass objectA = new Superclass();
        Superclass objectB = new Subclass();
        A.print();
        B.print();
    } //end main

}//end class
```

```
Output:
           print in Superclass.
           print in Subclass.
```

Figure 8-20 Sample of late binding

The **"@Override"** *annotation* is used to ask compiler to override specific method or element that is declared with in the parent class. Although placing this annotation is not required in overriding a base class method, this aid in preventing errors.

Activities and Exercises:

1. Describe/Define in your own word what inheritance and polymorphism is and how they are achieved?
2. Using UML class diagram create/draw a diagram that illustrates inheritance. Provide at least three classes including their attributes, behaviors and relationships.
3. Using UML class diagram create/draw a diagram that illustrates polymorphism in terms of overloading and overriding.
4. Describe/Define in your own word, the meaning of static and dynamic binding in polymorphism.

Programming Exercises

1. Write program code for **Faculty.java.** This class inherits from **Person.java** (using figure 8-6) with an instance variable called **title** that holds values such as “Professor”, “Associate Professor”, or “Assistant Professor”. Include a constructor, setter method and getter method for the **title**. Make sure that both classes are saved and compiled.
2. Using the program in exercise 1, write a program called **FacultyApp.java.** This application asks the users to enter information for a faculty object, instantiate the Faculty object and display information about the faculty.
3. Create a class called **Vehicle** in **Vehicle.java** with instance variables *brand* (such as Toyota, Honda, Ford, Isuzu etc), *year*, *horsepower* (200, 300 or 400) and milesPerGallon. Add necessary constructor, set and get methods. Then create classes **Bus** and **Truck** in a program files **Bus.java** and **Truck.java,** respectively. The **Bus** has *numberOfPassengers* (1 to 50) and the **Truck** has *towingCapacity* (0.5, 1.0 , 2.0 or 3.0 tons) instance variables. Both class inherit from **Vehicle** and contains their respective constructors, setter and getter methods. Both **Bus** and **Truck** have a method called computeMPG(miles per gallon) and *toString* method for displaying all attributes including the inherited one. The formula for **Bus**’ *milesPerGallon* is 1000/ *numberOfPassengers/ horsepower* and the formula for **Truck’s** *milesPerGallon* is 1000/ *towingCapacity/ horsepower.*
4. Create an application called **VehicleApp.java**. The application ask the user to enter information about Vehicle, for **Bus** the user enters brand, year, horsepower, number of passengers and for **Truck** the user enters brand, year, horsepower, number of passengers. By instantiating **Bus** and **Truck** objects, the application computes the miles per gallon and display all the information about the specific object (**Bus** and **Truck).**

QUIZ IS IT!

True or False. Read the following statements and identify whether the following statements are correct or wrong. Write **True** if the statement is correct; otherwise, write **False** before each number.

_________1. *Inheritance* is a special relationship between classes where one class has a direct access to the public members of another class.
_________2. A base class inherits all the private member of its superclass
_________3. *String* is a subclass of Object.
_________4. *Window* is the superclass of JOptionPane
_________5. The java keyword for implementing inheritance is *inherits*
_________6. In UML notation, inheritance uses a solid line followed by a hollow arrowhead from the child class to its parent class.
_________7. Inheritance can also be defined as ability of a single method name to be used in many different ways with in single class or within multiple classes.
_________8. The **"**@Override**"** *annotation* is used to ask compiler to override specific method or element that is declared with in the parent class.
_________9. The overloading and overriding are examples of polymorphism
_________10. Dynamic binding occurs on compile-time.
_________11. *Overriding* is a form of polymorphism which one method name with a given argument list or parameter list can be used in different classes within the inheritance hierarchy.
_________12. In overloading, one method name with a given signature can be used in different classes within the inheritance hierarchy.
_________13. Static binding uses actual object for binding.
_________14. The “super” keyword in inheritance, denotes that it access the constructor method of its base class.
_________15. In inheritance the term base class, superclass, and parent class is use describe the classes in upper inheritance hierarchy.

Multiple Choice – Encircle the letter of the correct answer.

16. Which of the following term implies two methods have same name, same parameters but exist in different class?
    a. Instance
    b. Overload
    c. Override
    d. clone
17. Which of the following java keyword is used to implement inheritance?
    a. inherits
    b. expands
    c. extends
    d. interface

18. This is kind of polymorphism that occurs when a single method name is used more than once with in the same class.

a. Instance	c. Overriding
b. Overloading	d. Cloning

19. Which java class is at the root of the inheritance hierarchy?

a. String	c. Object
b. Class	d. Base

20. Which of the following term does **NOT** relate to inheritance?

a. Superclass	c. Composition
b. Is-a	d.Subclass

LESSON 9
UNIFIED MODELLING LANGUAGE

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:
1. describe what UML is;
2. explain the origin and the acceptance of UML in industry;
3. differentiate Structure Diagram to Behavior Diagram;
4. identify the different types of Structure Diagram;
5. identify the different types of Behavioral Diagram; and
6. create a simple UML diagram based on made-up or real-world situations

READY. GET SET. DISCUSS!

The Unified Modelling Language

The *Unified Modeling Language* is an industry-standard modeling language with a vibrant graphic notation and a comprehensive collection of diagrams and components. It is designed to help developers of systems and applications specify, visualize, define, build, and record software systems' artifacts.

Brief History of UML

In the late 1970s and 1980s, object-oriented modeling language appeared. The object concepts were popular until 1995, but they were applied in several ways. Every developer had its notation and methodology. The industry as a whole was looking for a modeling language that would act as industry de-facto standard. In 1995, the three brilliant software engineers Grady Booch, Ivar Jacobson, and James Rumbaugh of Rational software, created a standard set of diagramming techniques known as the Unified Modeling Language. UML's goal to deliver standard object-oriented vocabulary and diagramming techniques sufficiently rich for modeling any system development project from analysis to implementation. In November 1997, UML was formally accepted as industry standards for all object developers by Object Management Group (OMG). Throughout the years, the UML has undergone numerous minor revisions. The most recent version of UML is 2.5.1, released in December 2017.

UML Diagram

In UML, a model is graphically depicted in the form of diagrams. A diagram gives a view of that part of the reality that the model describes. Currently, UML 2.5.1 offers fourteen officially recognized diagrams that define either the structure or the behavior of a system. UML specification sets out two main UML diagram types: structure diagrams and behavior diagrams. Structure diagrams display the system's static structure and components on various abstractions, implementation levels, and interrelated components. The structure diagram elements represent a system's specific principles, including theoretical, real-world, and implementation concepts. On the other hand, Behavior diagrams provide the analyst with a way to explain the complex relationships between the instances or objects comprising the business information system.

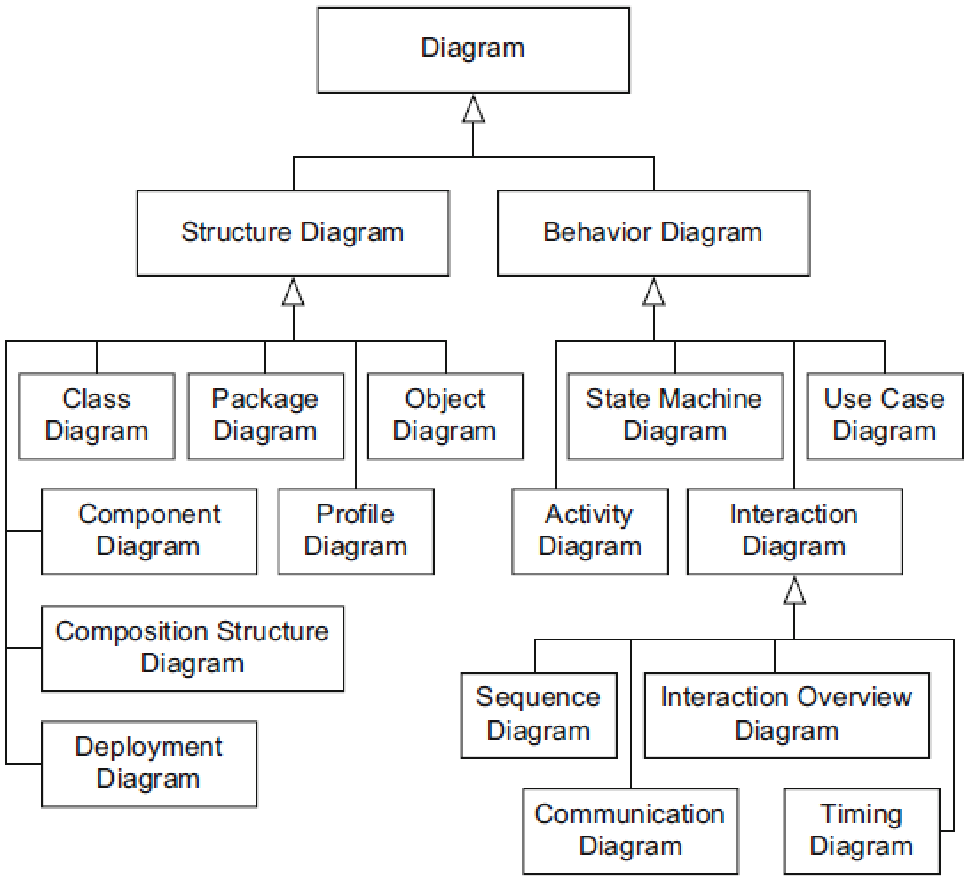


Figure 9-1 Taxonomy of UML diagrams.

Structure Diagram Types

UML offers seven diagrams to model the system's structures.

1. **Class diagrams** are used to model a system's static aspects by showing the classes and relationships. It shows each class's classes, attributes, and operations in a structure and the relationship between each class.

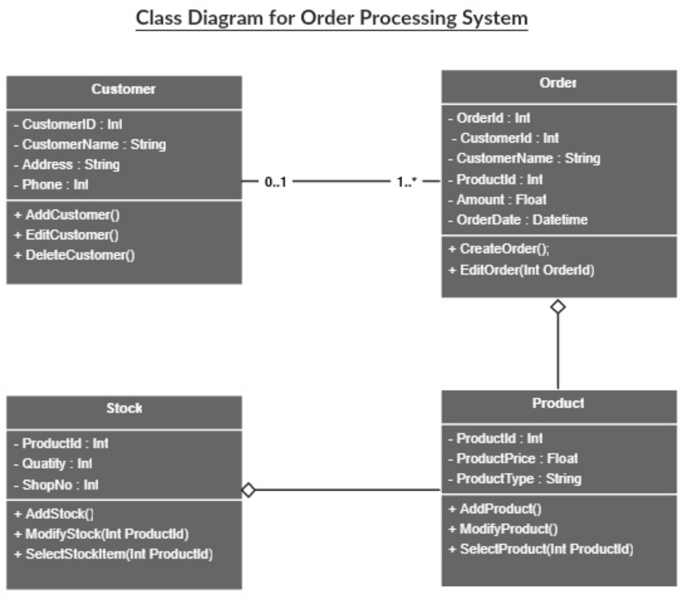


Figure 9-2 Class Diagram

2. **Object diagrams** are somewhat similar to class diagrams, often referred to as *Instance diagrams.* It shows a concrete system status snapshot at a specified time of execution. Like class diagrams, they often display the relationship between objects but use examples from the real world.

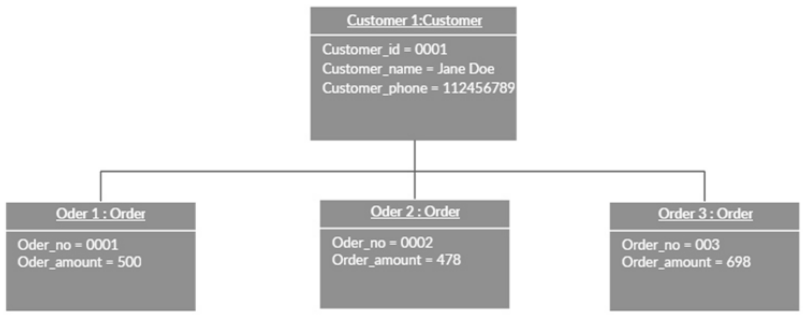


Figure 9-3 Object Diagrams

3. **Package diagrams** show the system dependencies between the various packages.

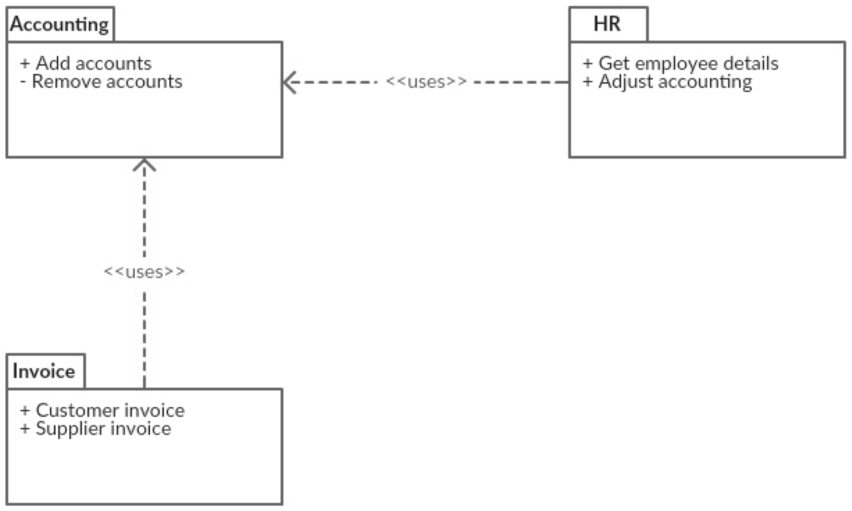


Figure 9-4 Package Diagrams

4. **Component Diagram** describes how the system's components link up to form that system. It is an independent, executable unit that provides services to other components, or uses other component services.

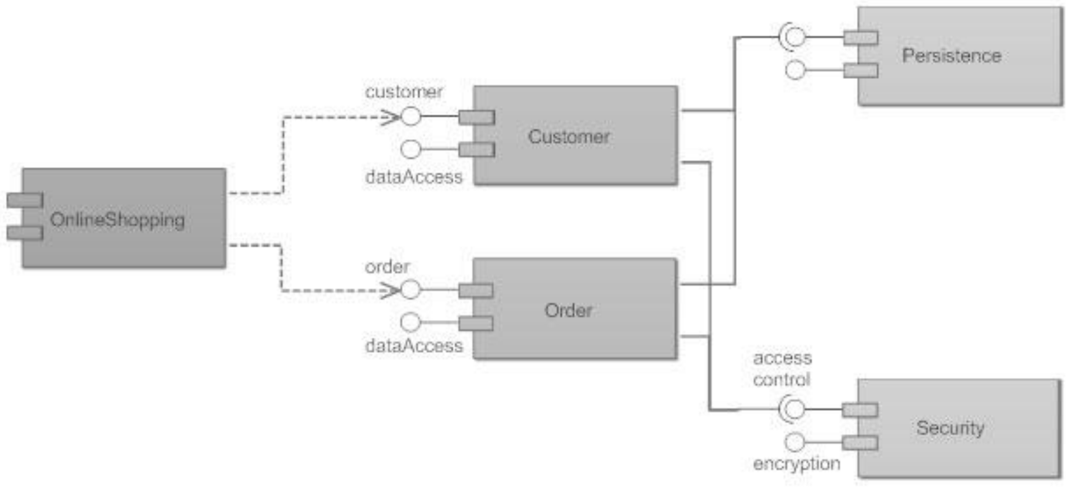


Figure 9-5 Component Diagram

5. **Composite Structure Diagram** enables the hierarchical decomposition of system parts. So you can use a composition structure diagram to describe in detail the internal structure of classes or components.

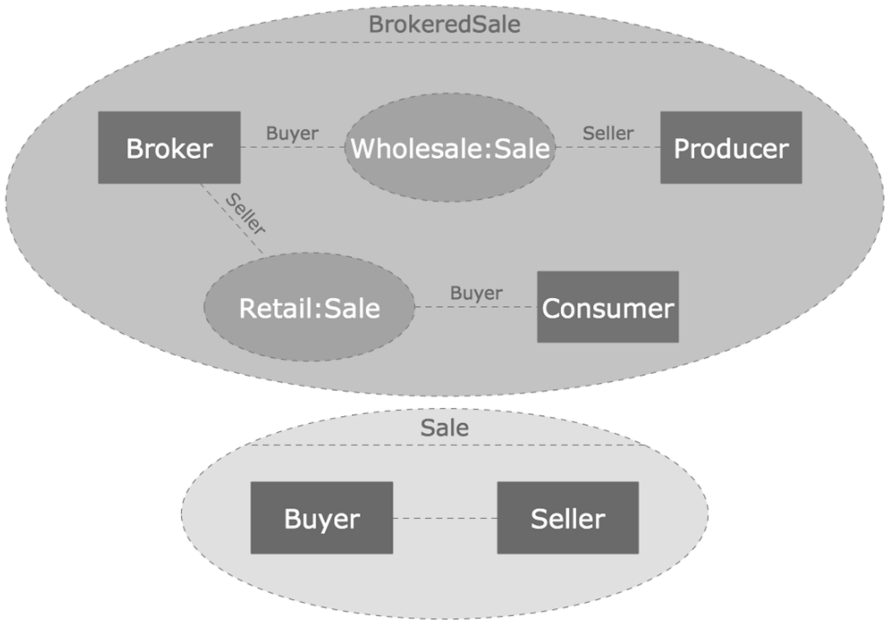


Figure 9-6 Composite Structure Diagram

6. **Profile Diagram** is a new type of diagram in UML 2.This is a type of diagram that is rarely used in any specification.

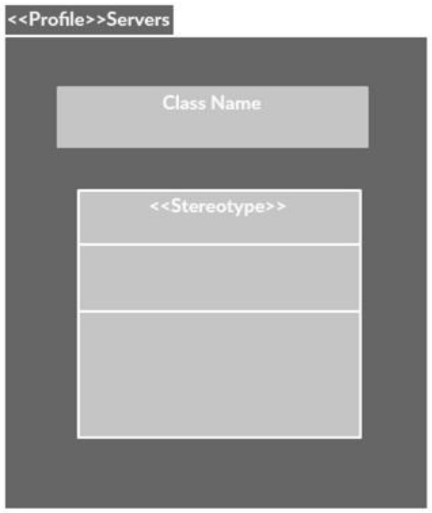


Figure 9-7 Profile Diagram

7. **Deployment Diagram** represents the hardware topology used, and the runtime system assigned.

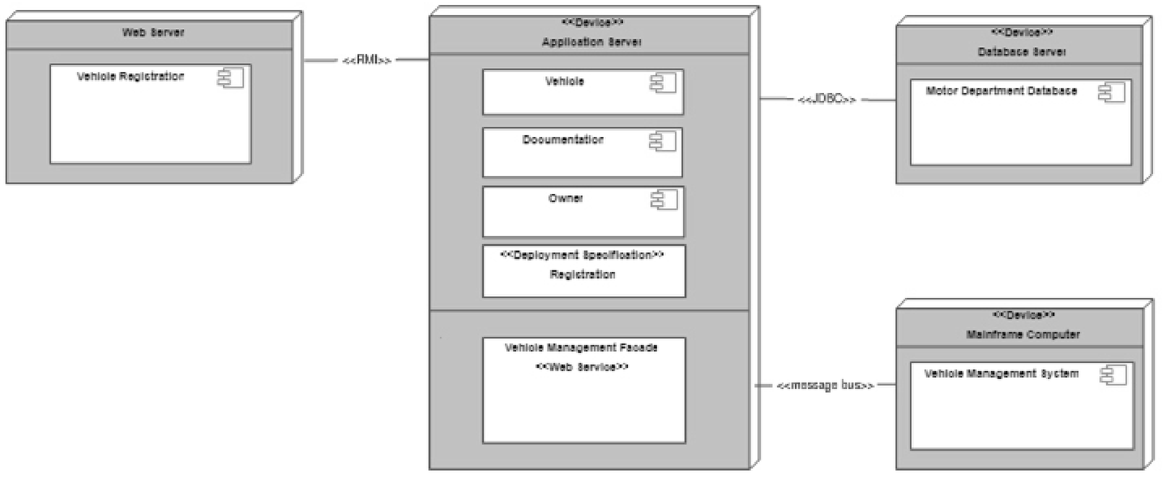


Figure 9-8 Deployment Diagram for a Vehicle Registration System

Behavioral Diagram Types

Behavior Diagram suggests the infrastructure that allows you to define behavior.

1. **Use Case Diagram** defines how users interact with a system through use cases that yield measurable results. This diagram explains which users are using certain system functionalities, but does not discuss precise implementation.

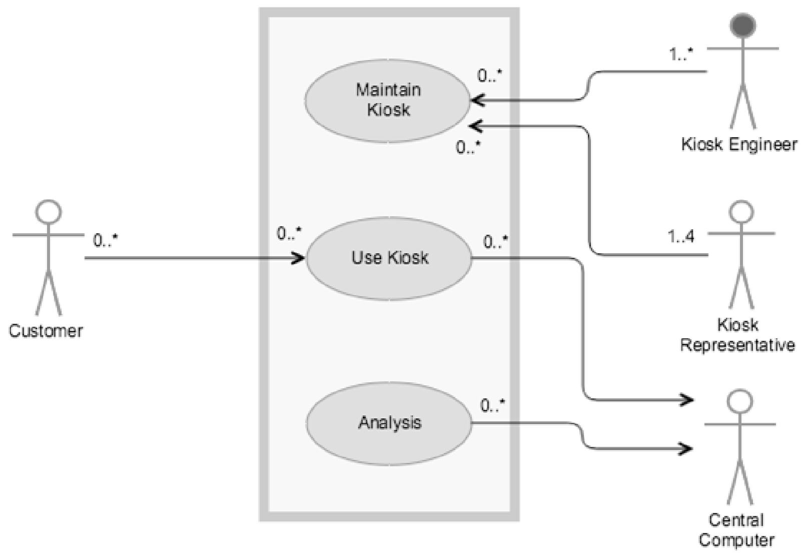


Figure 9-9 Deployment Diagram for a Vehicle Registration System

2. **State Machine Diagram** model the behavior of the system over several transitions, such that the designer can see at any point the singular 'state' of the system given the external stimuli indicated.

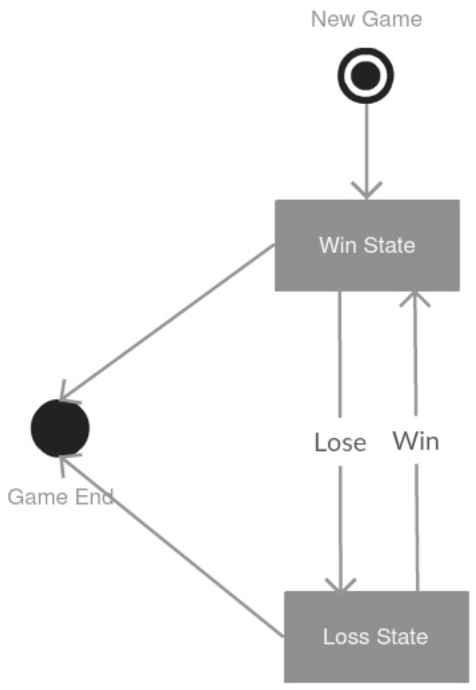


Figure 9-10 State Machine Diagram

3. **Activity Diagram** is a flowchart version of UML. It can define the business workflow of any part in a system and often used as an alternative to state machine diagrams.

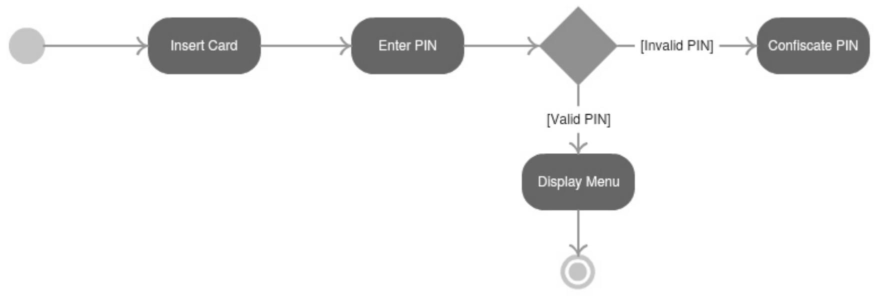


Figure 9-11 Activity Diagram

Interaction Diagram Types

4. **The sequence diagram** illustrates how events interact and the order of those interactions.

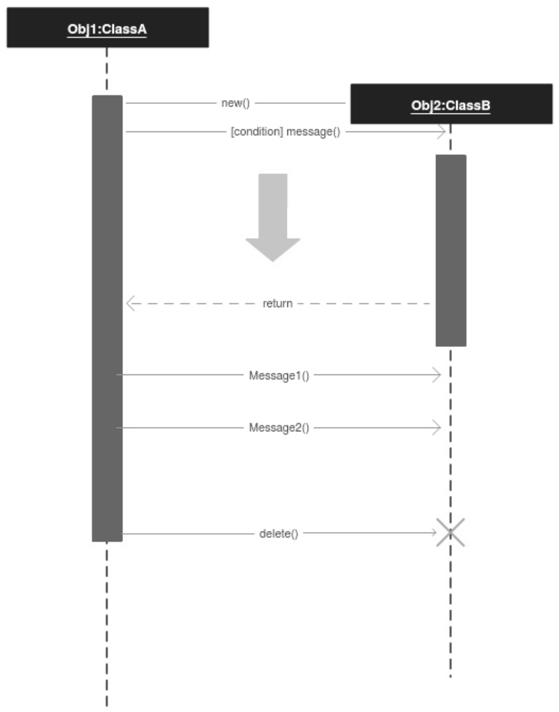


Figure 9-12 Sequence Diagram

5. **Communication diagrams** describe the communication between different objects similar to the sequence diagram.

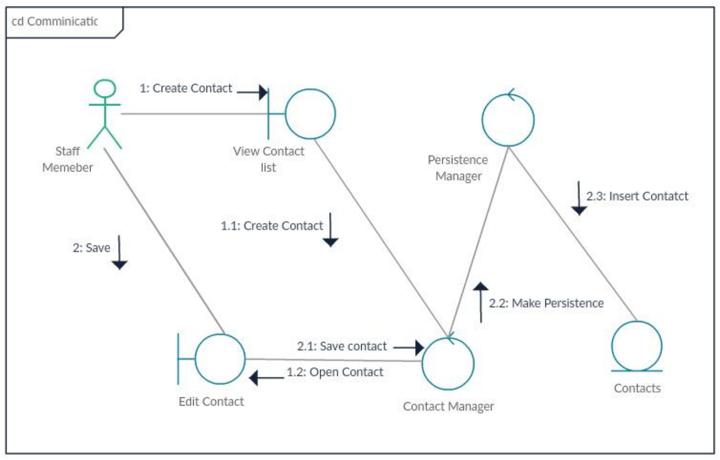


Figure 9-13 Communication Diagram

6. **Interaction overview diagrams** are somewhat related to activity diagrams. The *activity diagrams* show the process in sequence while *Interaction overview* diagrams display a series of interaction diagrams.

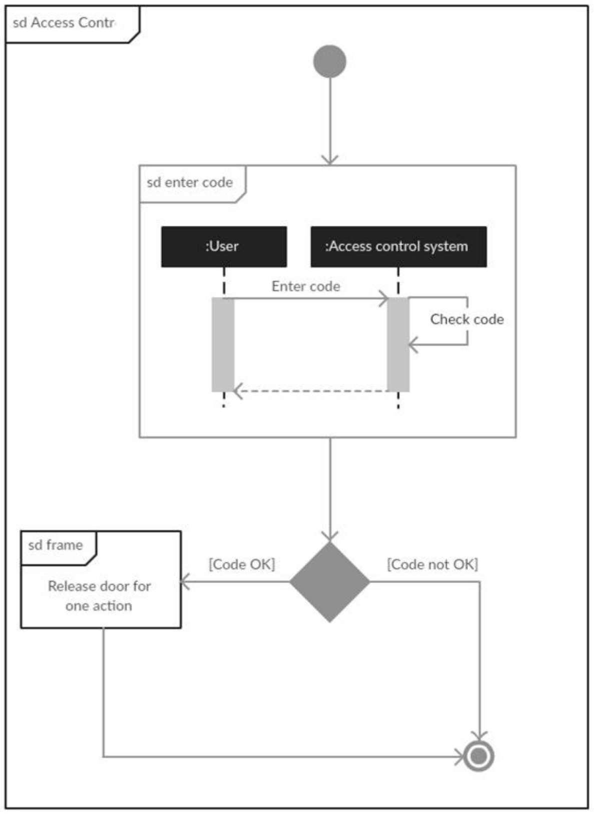


Figure 9-14 Interaction Overview Diagram

7. **Timing diagrams** closely mimic sequence diagrams. They represent object behavior within a given time frame.

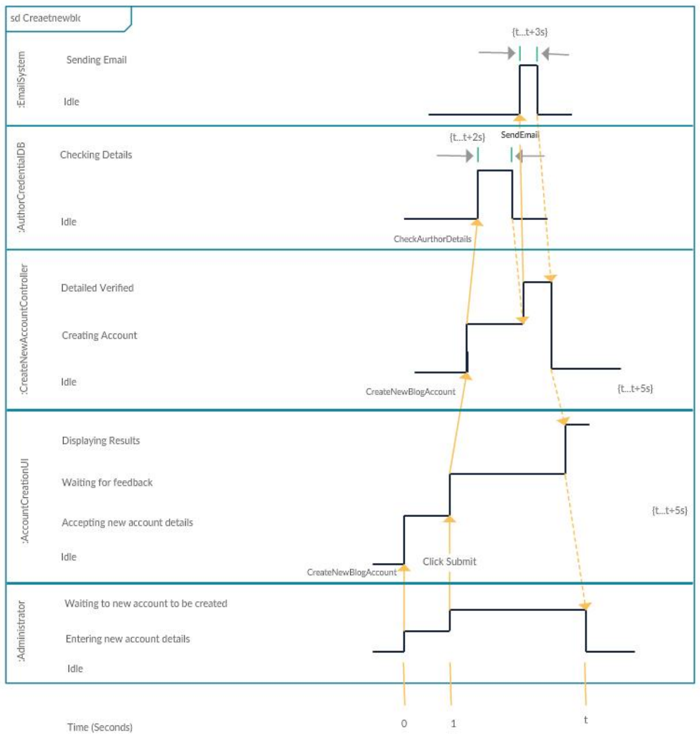


Figure 9-15 Timing Diagram

The Use Case Modeling

The *use case* is a fundamental principle for other object-oriented technologies. This is applied in the entire process of analysis and design. A *use case diagram* allows the system designer to discover the user's perspective's target system requirements.

Use Case Diagram Component

A use case diagram consists of three principal components:

1. **Actors** are the users that interact with a system. An actor may be an individual, an organization, or an external system that interacts with your application or system. There are two symbols in UML specification to represents an actor. First is the stick figure (shown in Figure 9-15a), and another way to describe an actor is a class icon with the <<actor>> stereotype (shown in Figure 9-15b).

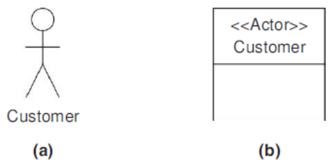


Figure 9-15a Stick Figure 9-15b. An actor icon

Also, actors can be classified into two types: *primary actors* and *secondary actors*. The principal actor takes the main benefit from case execution while the secondary actor also called supporting actors, manages, supervises, and operates the system. In Figure 16a, Professor is the primary actor, while the Email server is the secondary actor. In Figure 9-16b, the student is the primary beneficiary. They are both involved in the use case's execution, but the Professor has a lower benefit.

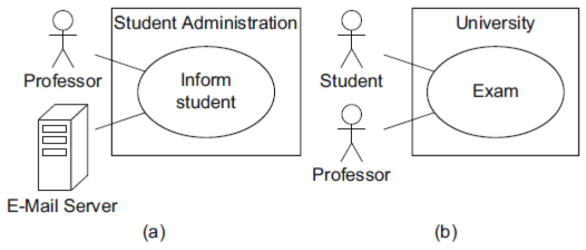


Figure 9-16 a and b Example of Primary and Secondary Actor

2. **Use Cases** define a series of actions a system performs to produce an observable outcome of value to a given actor. In the UML, a *use case* is represented by a horizontally shaped oval with a label that describes the actor's goal. It is linked to one or more actors by using straight-lines called communication links. Figure 9-17 shows an example of Interaction in the ATM System, the goal of the Actor (Customer) is to withdraw money from his/her account.

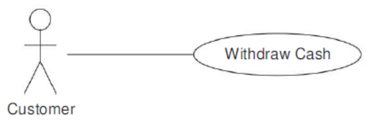


Figure 9-17 Actor, use case, and communication link.

3. **System Boundaries** are represented by rectangles that describe the scope of the system. The use cases should place within the system boundaries, while actors are residing outside the system boundary.

Use Case Models: Examples

Example 1: An Automatic Teller Machine System

An Automated Teller Machine (ATM) is a banking subsystem that provides access to bank customers' financial transactions in a public space without the need for a cashier or bank teller. The customer (actor) operates the ATM on his / her bank accounts for checking balance, deposit funds, and withdrawal and transfer funds (use cases). ATM Technician (actor) is responsible for maintaining and repairing ATM Machine. ATM Machines are owned by a bank (actor) that processes all the transactions made by the customer.

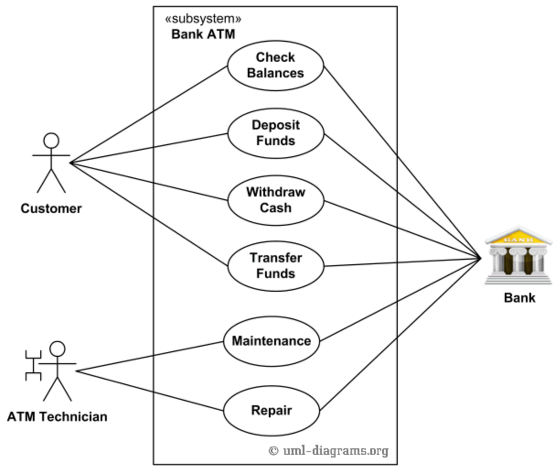


Figure 9-18 Use Case Diagram for Bank ATM subsystem

QUIZ IS IT!

I. IDENTIFICATION. Identify the following. (2 pts. Each)

_______________ 1. It is designed to help developers of systems and applications specify, visualize, define, build, and record software systems' artifacts.

_______________ 2. It is defined as a series of actions a system performs to produce an observable outcome of value to a given actor.

________________3. It is used to model a system's static aspects by showing the classes and relationships

________________4. It describes how the system's components link up to form the system .

________________5. These are the users that interact with a system.

________________6. It shows a concrete system status snapshot at a specified time of execution.

________________7. It is a new type of diagram in UML 2.This is a type of diagram that is rarely used in any specification.

_______________8. It is represented by rectangles that describes the scope of the system.

_______________9. UML was formally accepted as industry standards for all object developers by Object Management Group (OMG) on ______________.

______________10. It represents the hardware topology used, and the runtime system assigned.

______________11. This closely mimic sequence diagrams. They represent object behavior within a given time frame.

______________12. This show the system dependencies between the various packages.

______________13. It enables the hierarchical decomposition of system parts.

______________14. Model the behavior of the system over several transitions, such that the designer can see at any point the singular 'state' of the system given the external stimuli indicated.

______________15. It is define as a series of actions a system performs to produce an observable outcome of value to a given actor.

II. ENUMERATION. Enumerate the following. (2 pts. Each)

_______________1-3 Enumerate the three software engineers who created a standard set of diagramming techniques known as the Unified Modeling Language.

_______________2.

_______________3.

_______________4-10 Enumerate the seven diagrams to model the system's structures.

_______________5. _______________6. _______________7.

_______________8. _______________9. _______________10.

_______________11 -17 Enumerate the diagrams to model the system's behavior.

________________ 12. ________________ 13. ________________ 14.

________________ 15. ________________ 16. ________________ 17.

________________ 18-20 Enumerate three principal components of use case diagram.

_________________19.

_________________20.

Create a Use Case Diagram for Library Management System

INTEGRATIVE PROGRAMMING AND TECHNOLOGIES

LESSON 10
RATIONAL UNIFIED PROCESS

LET'S START! LEARNING OBJECTIVES

At the end of the lesson, the students must be able to:

1. define Rational Unified Process;
2. enumerate the six best practices for software development supported by RUP;
3. enumerate the four building blocks of RUP;
4. enumerate the six core engineering workflows of RUP;
5. enumerate the three core supporting workflows of RUP;
6. enumerate the four life cycle phases of RUP;
7. define software component;
8. explain the importance of use cases in RUP; and
9. state the advantages and disadvantages of using RUP.

READY. GET SET. DISCUSS!

RATIONAL UNIFIED PROCESS

The *Unified Rational Method* is a method of information engineering. It provides a structured mechanism by which to delegate tasks and responsibilities through development within an organization.

Rational Software designs and manages it, which is combined with its applications engineering solutions suite.

The Rational Unified Process (RUP) consolidates the entirety of the prescribed procedures in the creation of computerized applications in a way custom-fitted to a wide scope of projects and associations. Specifically, it contains the following six practices:

1. Develop software iteratively.
2. Manage requirements.
3. Use component-based architectures.
4. Visually model software.
5. Continuously check the efficiency of the program.
6. Control the software changes.

THE RATIONAL UNIFIED PROCESS AS A PRODUCT

Numerous associations have step by step become mindful of the significance of an all-around characterized and all around archived programming advancement process for their product tasks to be fruitful. They've assembled their mastery throughout the years and imparted it to their engineers.

Sometimes this collaborative know-how evolves out of processes, written manuals, educational programs, and tiny notes collected over several tasks. Unfortunately, such activities frequently end up accumulating dust on a developer's seldom revised shelf in pleasant binders, easily becoming outdated and almost never pursued.

The RUP imparts numerous qualities:

- Regular upgrades
- Pure Online
- Customization
- Many Integration

WHO IS USING THE RATIONAL UNIFIED PROCESS?

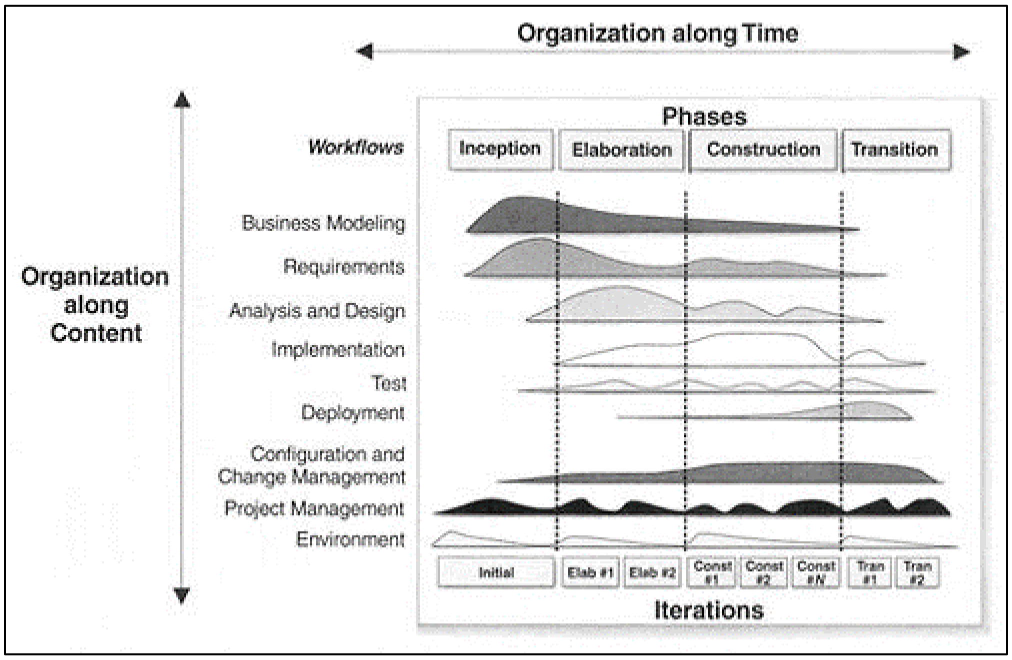
In 1999, more than 1,000 organizations were utilizing the Rational Unified Process. They used it in a variety of applications, as well as in large and small projects.

It illustrates the simplicity and broad use of the Rational Unified Method.
Here are the examples.

- System integrators (Oracle, Deloitte & Touche)
- Transport (aerospace,Lockheed Martin, British Aerospace)
- Industrial (Xerox, Volvo, Intel)
- Telecommunications (Ericsson, Alcatel, MCI)
- Finances (Visa, Schwab)

PROCESS STRUCTURE: TWO DIMENSIONS

Figure beneath shows the general designing for the Rational Unified Process.



The horizontal axis shows time and reveals the life-cycle aspects of the phase as it progresses. The vertical line reflects the main workflows of tasks that fundamentally coordinate tasks.

The first dimension is the *complex component* of the system as it is implemented and represented in terms of phases, steps, iterations, and milestones. The second aspect is the *static component* of the method: the description of process components, processes, workflows, artifacts, and personnel.

THE BUILDING BLOCKS OF RATIONAL UNIFIED PROCESS

All parts of the Rational Unified Process depend on the arrangement of building obstructs that are utilized to depict what should be made, who is responsible for creating it, how creation can happen, and when creation is finished. The four squares of the structure are as follows:

- **Workers, the 'Who'**: Acts and functions of an individual or a community of individuals operating together as a team on a project to create objects.
- **Activities, the 'How'**: Activities would have a specific aim, usually by making or modifying objects.
- **Artifacts, the 'What'**: The artifact is every measurable product that results from the process.
- **Workflows, the 'When'**: Represents a diagramed series of events intended to generate tangible meaning and objects.

Here are six (6) engineering workflows of the RUP:

- **Business Modeling Workflow**: The business sense (scope) of the project will be illustrated throughout this workflow.
- **Requirements Workflow**: Used to identify all possible project specifications during the software development life cycle.
- **Analysis & Design Workflow**: If the process specifications have been fulfilled, the review and design phase must take these criteria into consideration and turn them into a specification that can be effectively executed.
- **Implementation Workflow**: That's where a lot of actual scripting takes place, combining and sorting all the programming into layers that make up the entire system.
- **Test Workflow**: Testing of all sorts is taking place inside this system.
- **Deployment Workflow**: Finally, the development procedure is the whole delivery and production period. It ensures that the software deliver to the customer as expected.

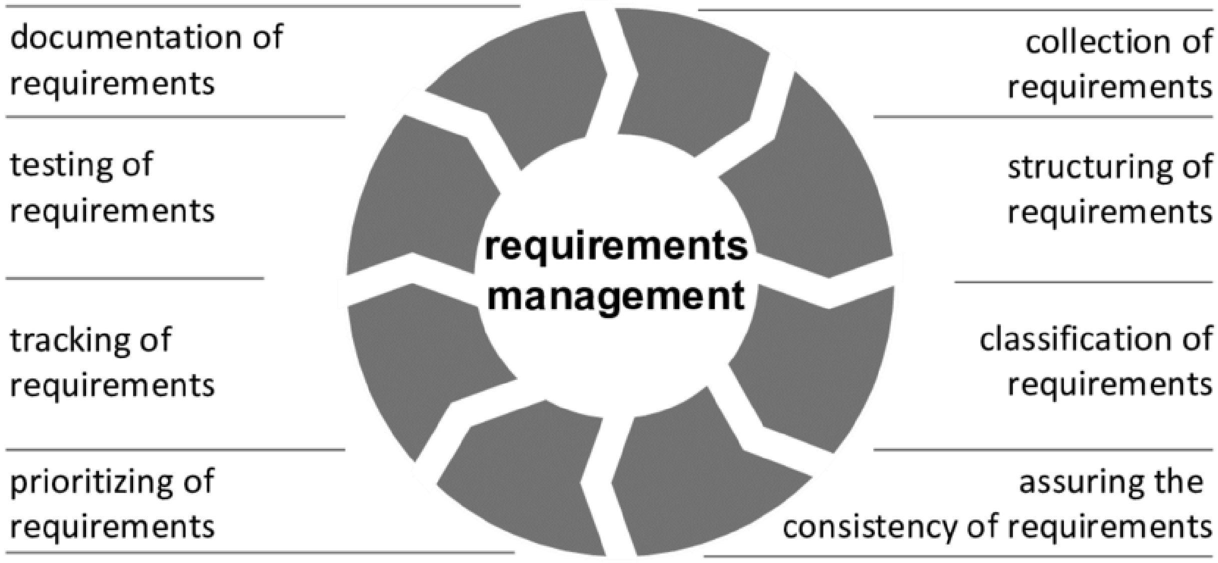
RUP BEST PRACTICES

This segment addresses the five core traditional behaviors and contrasts them to the central components of the Rational Unified Process.

Requirements Management

Requirements Management is a structured approach to creating, coordinating, interacting, and handling the evolving requirements of a software-intensive program or application.

The results of effective implementation of specifications include the following:


documentation of requirements
collection of requirements
testing of requirements
structuring of requirements
requirements management
tracking of requirements
classification of requirements
prioritizing of requirements
assuring the consistency of requirements

UML and Modelling

The preparation and evaluation of the system in implementation is a significant aspect of the Rational Unified System. Models require both the question and the answer to be grasped and shaped. A model is advancement in design that lets us consider a massive, dynamic system that cannot be entirely controlled.

UML, short for Unified Modeling Language, is a structured modeling language composed of an interconnected series of diagrams, intended to support program and software engineers define, imagine, build and record the functionality of software systems, as well as market modeling and other non-functional structures.

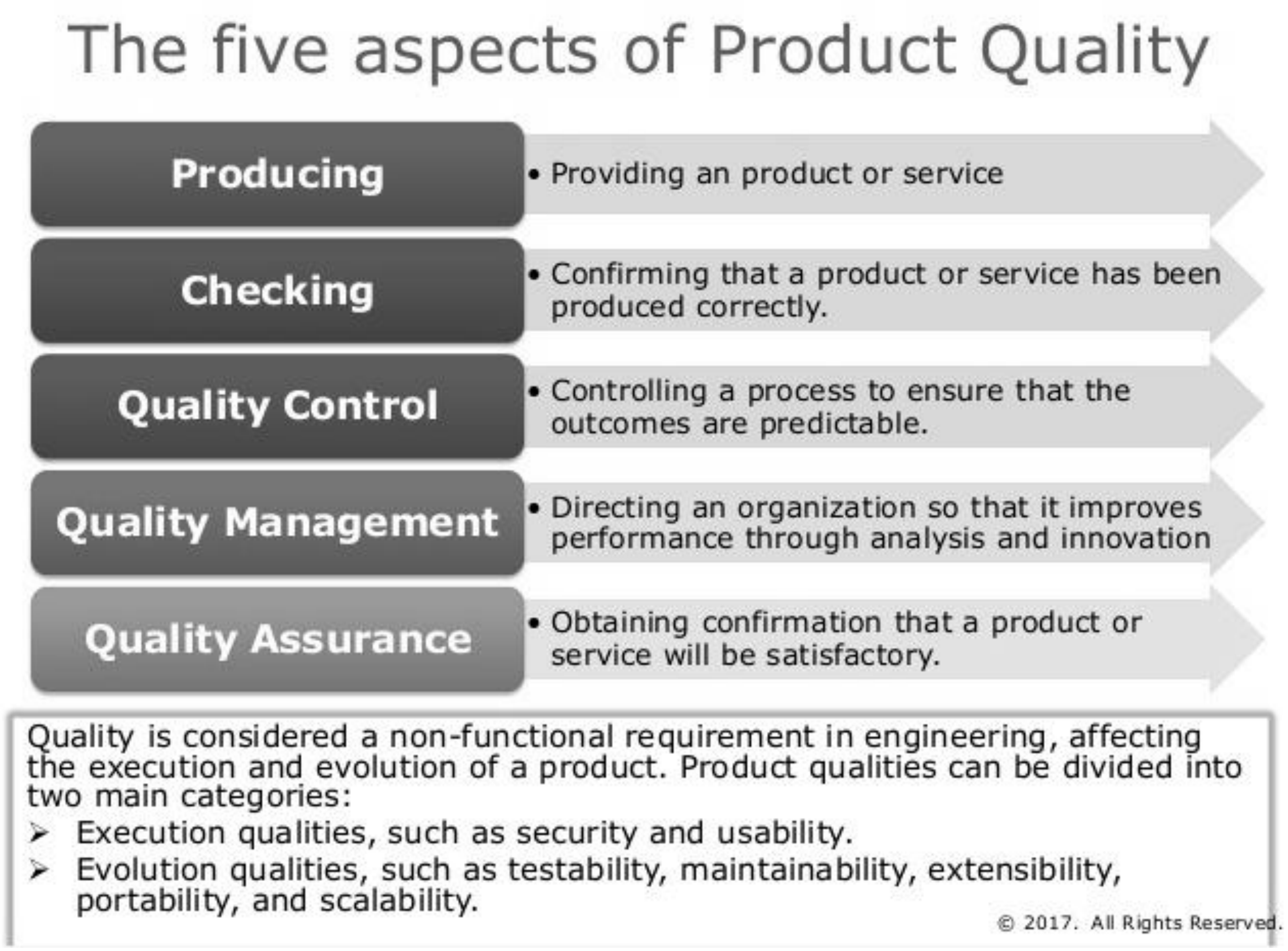
Architecture & Use of Components

The design activities are focused on the concept of architecture-either system architecture or software-intensive software architecture.

The essential target of the early emphases of the procedure is the creation and testing of a product model that, in the underlying advancement stage, appears as an executable building usage that in the end changes into the last system in ensuing cycles.

Process Quality & Product

Individuals often worry if there is no clear authority in the rational unified process. The response is that a few people are not adding consistency to the commodity. Consistency, however, is the responsibility of each employee of the development company. In software engineering, our concern for quality focuses on two areas: quality of product and process performance.


The five aspects of Product Quality

Producing	• Providing an product or service
Checking	• Confirming that a product or service has been produced correctly.
Quality Control	• Controlling a process to ensure that the outcomes are predictable.
Quality Management	• Directing an organization so that it improves performance through analysis and innovation
Quality Assurance	• Obtaining confirmation that a product or service will be satisfactory.

Quality is considered a non-functional requirement in engineering, affecting the execution and evolution of a product. Product qualities can be divided into two main categories:

- Execution qualities, such as security and usability.
- Evolution qualities, such as testability, maintainability, extensibility, portability, and scalability.

© 2017. All Rights Reserved.

Iterative

The iterative process suggested by the Rational Unified Process is obviously built for a straight-line technique or a cascade strategy for a number of reasons:

- Requires that the specification of the can be taken into account. The truth is, the requirements usually change. Changing requirements and "creep" requirements have also been the main cause of project difficulty, leading to late completion, missing deadlines, unsatisfied buyers, and disappointed developers.
- In the RUP, integration is not a single "big bang" at the end; rather, the components are slowly incorporated. This iterative approach is like a continuous integration process.

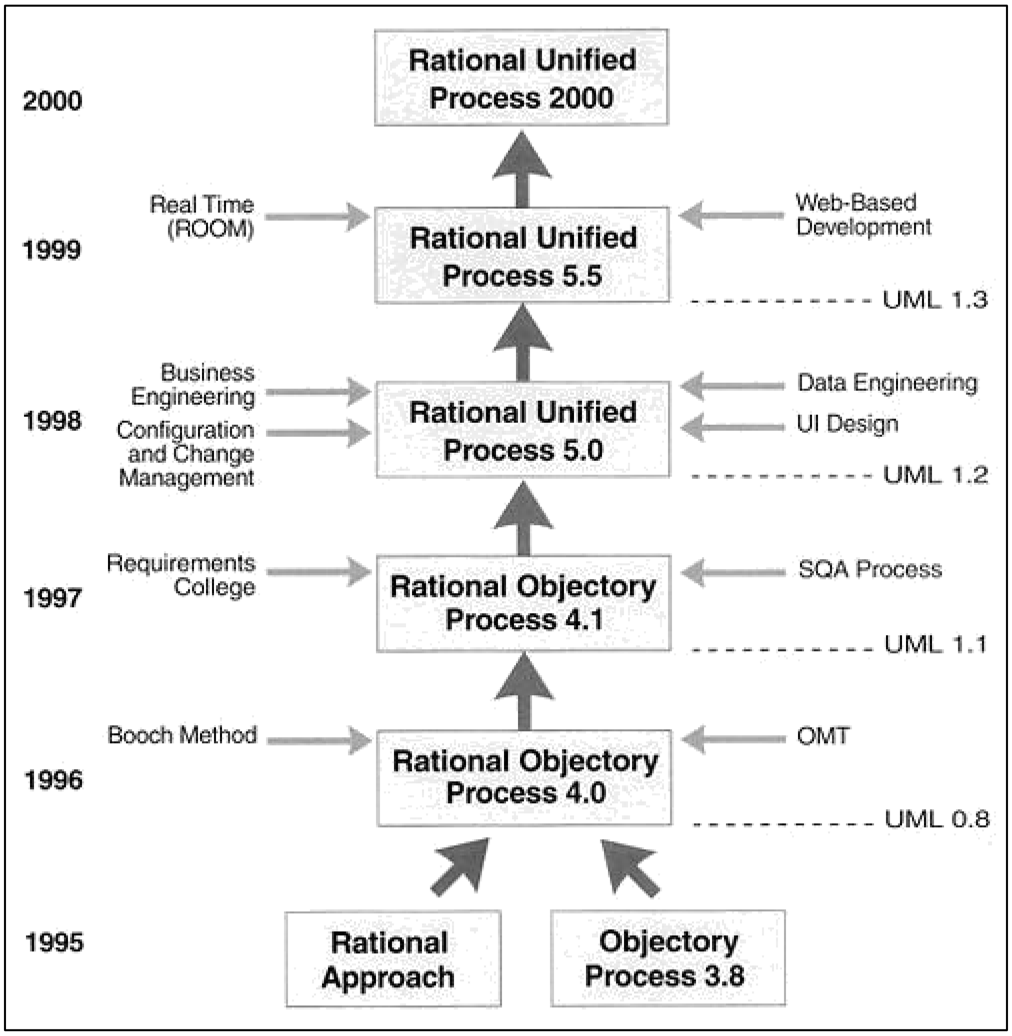
- The iterative approach helps you to minimize threats faster as integration is usually the only way that threats are detected or discussed.
- Provides executives with the ability to make incremental improvements to the company for some cause, for example, to compete with competing products.

- Facilitates re-use because it is easier to recognize similar components as partly planned or applied rather than to recognise all commonalities from the outset before something has been developed or introduced.

- This helps in a very stable architecture, since you're fixing mistakes over many iterations. Faults are observed only in early iterations as the project advances into development in the past and not at the conclusion of a significant testing cycle.

SHORT BACKGROUND OF THE RUP

The RUP has developed throughout the years and speaks to the basic comprehension of numerous people and organizations who make up the differing inheritance of Rational Software today. Taking a brief look at the rich ancestors of RUP 2000.



Rational Unified Process is a rational representation of an extremely large technique defined by Grady Booch, James Rumbaugh and Ivar Jacobson reading the Unified Application Development System course.

ADVANTAGES OF RATIONAL UNIFIED PROCESS

- Requires flexible ability to cope with evolving demands during the life cycle of construction, whether from the client or from the project itself.
- Emphasizes the need for (and thorough application of) correct documentation.
- Disseminates potential integration headaches by allowing integration to occur during production, especially in the construction phase, when all other coding and creation is taking place.

DISADVANTAGES OF RATIONAL UNIFIED PROCESS

- This depends heavily on skilled and experienced team leaders since the transfer of tasks to specific staff will yield concrete, pre-planned outcomes in the form of objects.
- Despite the focus on integration in the production process, this can often be counterproductive during experimentation or other processes, when integrations are overlapping and in the way of other, more basic operations.

Probably, RUP is a very complicated pattern. Given the number of components involved, including best practices, stages, building blocks, goal requirements, iterations, and workflows, the correct execution and use of the Rational Unified Process can also be difficult for many organizations, especially smaller teams or programs.

QUIZ IS IT!

Part I. Identification

Instruction: Put an **O** at the underline if it is one of the best practices for software development supported by RUP and an **X** if it is not

1. ___ Develop software iteratively.
2. ___ Integrate software components only at the end.
3. ___ Manage requirements.
4. ___ Invest more time in planning to develop the system smoothly.
5. ___ Use component-based architectures.
6. ___ Verify software quality only once to save cost and time.
7. ___ Use Model-View-Controller architecture.
8. ___ Visually model software.
9. ___ Continuously verify software quality.
10. ___ Control changes to software.

REFERENCES

(2003) An Introduction to the UML and the Unified Process. In: Guide to the Unified Process featuring UML, Java and Design Patterns. Springer Professional Computing. Springer, London

Ambler, S.W. (2001). Enterprise Unified Process White Paper: http://www.ronin-intl.com/publications/unifiedProcess.htm

Cadenhead, R. (2012) Sams teach yourself Java. Sams Publishing. ISBN-13: 978-0-672-33575-4

Cosmina, I. (2018) Java for Absolute Beginners, Apress Publishing. ISBN-13: 978-1-4842-3778-6

Curtis, T. Object-Oriented Technology (2nd Edition) From Diagram to Code with Visual Paradigm for UML

Harold, E.R (2006). Java I/O 2nd Edition. O'Reilly. ISBN-10. 0-596-52750-0

Johnson, R. (2012). *An introduction to java programming and object-oriented application development*. Cengage Learning.

Kasparian, R.M (2006). Java for Artists: The Art, Philosophy, and Science of Object-Oriented Programming. Pulp Free Press. ISBN:1932504052

Kjell, B. (2003). Introduction to Computer Science using Java. Central Connecticut State University

Kruchten, P. (2000). The Rational Unified Process: An Introduction, 2nd edn. Addison Wesley Longman, Reading, MA

Malhotra, S., & Chaudhary, S. (2014). *Programming in JAVA*. Oxford University Press India.

OMG Unified Modelling Language: http://www.omg.org/technology/documents/formal/uml.htm

Rational Unified Process: http://www.rational.com/products/rup/

Rational Unified Process: What Is It And How Do You Use It? https://airbrake.io/blog/sdlc/rational-unified-process

Suggested Readings

"Creating a GUI in Java" @ http://docs.oracle.com/javase/tutorial/ .
"Learn Java Programming" @ https://www.tutorialspoint.com/java/.
https://www.w3schools.com/java/java_oop.asp
http://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html
https://www.tutorialspoint.com/java/java_polymorphism.htm
https://docs.oracle.com/javase/tutorial/java/IandI/index.html
https://www.geeksforgeeks.org/difference-between-early-and-late-binding-in-java/
https://www.visual-paradigm.com/tutorials/uml-class-diagram-in-diff-programming-languages.jsp

Digital Materials

- Video: Unified Process Model: Definition & Application: https://study.com/academy/lesson/unified-process-model-definition-application.html
- Learning Java 8 – Full Tutorial for Beginners. YouTube Tutorial Video

Applications which can be downloaded in Google Play

1. Learn Java
2. Learning Java Programming
3. Learn Java