

Saint Louis University

Baguio City, Philippines



MODULE IN

Computer Programming

COMPROG

Computer Applications Department

**SCHOOL OF ACCOUNTANCY, MANAGEMENT, COMPUTING
AND INFORMATION STUDIES**

Property of and for the exclusive use of SLU. Reproduction, storing in a retrieval system, distributing, uploading or posting online, or transmitting in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise of any part of this document, without the prior written permission of SLU, is strictly prohibited.



COMPUTER PROGRAMMING(LAB)

COMPROG

COURSE LEARNING OUTCOMES

At the end of the module, you should be able to:

1. Analyze and explain the behaviors of simple programs involving the fundamental programming constructs;
2. Create algorithms for solving simple problems;
3. Create, modify, or expand short programs that use standard conditional and iterative control structures;
4. Write programs that use arrays and strings, modularization and polymorphisms;
5. Write programs using graphical user interface (GUI) components;
6. Design, implement, test and debug programs that use fundamental programming constructs: basic computation, simple input and output, standard conditional and iterative structures, and the definition of functions.
7. Describe the mechanics of parameter passing.
8. Act in accordance to the professional, social, legal, security and ethical responsibilities in programming.
9. Maximize individual differences and talents to effectively accomplish programming tasks.
10. Apply the techniques of structured decomposition to break a program into smaller pieces.
11. Demonstrate ethical programming standards for his/her personal, professional, and social advancement;
12. Imbued the virtue of honesty and fair-play in programming practices.



Everyone should know how to program a computer because it teaches you how to think.

Steve Jobs

COURSE INTRODUCTION

The content of this module focuses on fundamentals of algorithm development and fundamental concepts of programming. The concepts include basic syntax and semantics; variables, data types, operators, and expressions; the role of algorithms in the problem-solving process; input and output statements; conditional and iterative control structures; arrays; methods and parameter passing; program structure decomposition/modularization; arrays; string processing; wrapper classes; object-oriented programming, JOptionPane class, inheritance, method overloading and polymorphisms. The student will use a high-level programming language in creating computer solutions to engineering problems.



Table of Contents

Module 01: Java Fundamentals	7
Programming Language	7
About Java.....	7
Java Platform.....	7
Java Programming Environment.....	7
How are Java Programs Written?.....	8
Fundamental Concepts	8
Java Comments.....	9
Java Statements, Blocks and Expressions	9
Java Identifiers.....	10
Naming Rules and Styles.....	10
Java Keywords.....	10
Primitive Data Types in Java.....	11
Character Strings	12
Separators	12
Variables	12
Constants	13
The println Method.....	14
The print Method	14
String Concatenation	14
Escape Sequences	14
Module 02: Java Operators, Flowcharting Symbols and Sample Programs.....	15
Operators	15
Arithmetic Operators	15
Increment/Decrement Operators.....	16
Relational Operators	17
Logical Operators	17
Creating a Java Program	17
Simple Calculation	17
The Program Development Life Cycle	18
Flowcharting Symbols and Their Meanings	19
Representing Algorithms Using Flowcharts	20



Module 03: Getting Program Input and Sequence Problems.....	21
Problems for Program Development.....	23
Sequence Programs.....	23
Module 04: Java Control Structures: Decision Control Structure	27
Java's Selection Statements	27
If Statement.....	27
Nested ifs	28
The if-else-if Ladder	29
Problems for Program Development	31
Switch	34
Problems for Program Development	35
Module 05: Java Control Structures: Iteration Control Structure	36
While Loop.....	36
While Loop Flowchart Diagram.....	37
More While Loop Examples.....	37
Do-while Loop	41
Do-while Loop Flowchart Diagram	42
More Do While Loop Examples.....	42
For Loop.....	45
For Loop Flowchart Diagram	45
More For Loop Examples.....	46
Java For Loop vs While Loop vs Do While Loop	48
Problems for Program Development.....	49
Repetition Control Structure.....	49
Module 06: Java Arrays	53
Arrays	53
One-Dimensional Arrays.....	54
More examples of One-Dimensional Array	56
Problems for Program Development	56
Multidimensional Arrays.....	58
Module 07: Java Methods	60
Creating Method.....	61



Method Calling:.....	61
The void Keyword:.....	62
Passing Parameters by Value:	63
Method Overloading:	64
The Constructors	65
Parameterized Constructor	66
The this keyword.....	66
Problems for Program Development.....	68
Module 08: Java Wrapper Classes	69
Definition	69
Purpose	69
Autoboxing and Unboxing	69
Module 09: JOptionPane Class	70
Definition	70
JOptionPane Dialog Boxes.....	70
Programming Examples	72
Video Materials	73
Programming Exercises.....	73
Module 10: Object Oriented Programming	74
Inheritance	75
Superclass and Subclass	75
Polymorphism	76
Method Overloading	77
Programming Exercises.....	78
References:.....	79



Module 01: Java Fundamentals

Programming Language

- a standardized communication technique for expressing instructions to a computer. Like human languages, each language has its own syntax and grammar
- enables a programmer to precisely specify what data a computer will act upon, how these data will be stored/transmitted, and precisely what actions to take under various circumstances
- set of rules, symbols, and special words used to construct programs or instructions that are translated into machine language that can be understood by computers
- include Java, Python, C, C++, C#, Objective-C, Visual Basic, Pascal, Delphi, FORTRAN, and COBOL

About Java

A little Bit History

- developed in early 1990s by James Gosling et. al. as the programming language component of the Green Project at Sun Microsystems
- originally named Oak and intended for programming networked “smart” consumer electronics
- launched in 1995 as a “programming language for the Internet”; quickly gained popularity with the success of the World Wide Web
- currently used by around 5 million software developers and powers more than 2.5 billion devices worldwide, from computers to mobile phones
- Design Goals
 - simple: derived from C/C++, but easier to learn
 - secure: built-in support for compile-time and run-time security
 - distributed: built to run over networks
 - object-oriented: built with OO features from the start
 - robust: featured memory management, exception handling, etc.
 - portable: “write once, run anywhere”
 - interpreted: “bytecodes” executed by the Java Virtual Machine
 - multithreaded, dynamic, high-performance, architecture-neutral
 - Bytecodes are the machine language understood by the Java virtual machine

Java Platform

- Java Virtual Machine or JVM: a virtual machine, usually implemented as a program, which interprets the bytecodes produced by the Java compiler; the JVM converts the bytecodes instructions to equivalent machine language code of the underlying hardware; compiled Java programs can be executed on any device that has a JVM

Java Programming Environment

- Java programming language specification
 - Syntax of Java programs
 - Defines different constructs and their semantics
- Java byte code: Intermediate representation for Java programs
- Java compiler: Transform Java programs into Java byte code



- Java interpreter: Read programs written in Java byte code and execute them
- Java virtual machine: Runtime system that provides various services to running programs
- Java programming environment: Set of libraries that provide services such as GUI, data structures, etc.
- Java enabled browsers: Browsers that include a JVM + ability to load programs from remote hosts

How are Java Programs Written?

```
/* This program displays the message "Hello, World!" on the standard output device  
(usually the screen). This code must be saved in a file named HelloWorld.java...  
*/  
// this is the declaration of the HelloWorld class...  
public class HelloWorld {  
// the main method defines the "starting point" of the execution  
// of this program...  
public static void main(String[] args) {  
// this statement displays the program's output...  
    System.out.println("Hello, World!");  
} // end of method main...  
} // end of class HelloWorld...
```

The following are video tutorials of some of the programming editors or Integrated Development Environment (IDE) that can assist you in writing java programs. Follow the steps in the installation of any of the following IDEs.

- [Module 1: How to Install Eclipse IDE w Java JDK 13 on Windows 10.mp4](#)
- [Module 1: How to Install NetBeans 11 IDE And Java JDK SE 14 on Windows 10 8 7.mp4](#)
- [Module 1: How to Install JDK and JCREATOR LE.mp4](#)

Fundamental Concepts

- Java programs are made up of one or more *classes*.
- A Java class is defined through a *class declaration*, which, aside from assigning a *name* for the class, also serves to define the *structure* and *behavior* associated with the class.
- By convention, Java class names start with an *uppercase* letter. Java programs are *case-sensitive*.
- A Java *source code file* usually contains one *class declaration*, but two or more classes can be declared in one source code file. The file is named after the class it declares, and uses a *.java* filename extension.
- For a class to be *executable*, it must be declared *public*, and must provide a *public static method* called *main*, with an *array argument of type String*.
- If a file contains more than one class declaration, only one of the classes can be declared *public*, and the file must be named after the sole public class.
- The Java compiler (*javac*) is used to *compile* a Java source code file into a *class file* in *bytecode* format. The resulting class file has the same name as the source code file, but with a *.class* filename extension.



- The Java Virtual Machine (*java*) is used to execute the class file.

Java Comments

- Comments are notes written to a code for documentation purposes. Those texts are not part of the program and do not affect the flow of the program.

Two types of comment

- Single Line Comment

Example:

//This is an example of a single line comment

- Multiline Comment

Example:

/ This is an example of a
multiline comment enclosed by two delimiters
that starts with a /* and ends with a */
/

Java Statements, Blocks and Expressions

- A statement is any complete sentence that causes some action to occur. A valid Java statement must end with a semicolon.

Examples:

*System.out.println("Hello world");
int k; int j = 10;
double d1, d2, k, squareRootTwo;
k = a + b - 10;
boolean p = (a >= b);
squareRootTwo = Math.sqrt(2);*

- A block is one or more statements bounded by an opening and closing curly braces that groups the statements as one unit. Block statements can be nested indefinitely. Any amount of white space is allowed.

Example:

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

- An expression is a value, a variable, a method, or one of their combinations that can be evaluated to a value.

Examples:

*int cadence = 0;
anArray[0] = 100;
8 >= x;
p || q ;
System.out.println("Element 1 at index 0: " + anArray[0]);
double squareRootTwo = Math.sqrt(2)*



Java Identifiers

- Identifiers are tokens that represent names of variables, methods, classes, etc. Examples of identifiers are: Hello, main, System, out.
- Java identifiers are **case-sensitive**. This means that the identifier: **Hello** is not the same as **hello**.

Naming Rules and Styles

- There are certain rules for the naming of Java identifiers. Valid Java identifier must be consistent with the following rules.
 - An identifier cannot be a Java reserve word.
 - An identifier must begin with an alphabetic letter, underscore (_), or a dollar sign (\$).
 - If there are any characters subsequent to the first one, those characters must be alphabetic letters, digits, underscores (_), or dollar signs (\$).
 - Whitespace cannot be used in a valid identifier.
 - An identifier must not be longer than 65,535 characters.
- Also, there are certain styles that programmers widely used in naming variables, classes and methods in Java. Here are some of them.
 - Use lowercase letter for the first character of variables' and methods' names.
 - Use uppercase letter for the first character of class names.
 - Use meaningful names.
 - Compound words or short phrases are fine, but use uppercase letter for the first character of the words subsequent to the first. Do not use underscore to separate words.
 - Use uppercase letter for all characters in a constant. Use underscore to separate words.
 - Apart from the mentioned cases, always use lowercase letter.
 - Use verbs for methods' names.
- Here are some examples for good Java identifiers.
 - Variables: height, speed, filename, tempInCelcius, incomingMsg, textToShow.
 - Constant: SOUND_SPEED, KM_PER_MILE, BLOCK_SIZE.
 - Class names: Account, DictionaryItem, FileUtility, Article.
 - Method names: locate, sortItem, findMinValue, checkForError.
- Not following these styles does not mean breaking the rules, but it is always good to be in style!

Java Keywords

- Keywords are predefined identifiers reserved by java for specific purposes. You cannot use keywords as names of variables, classes, methods, etc.

abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		



Primitive Data Types in Java

- four of them represent integers: *byte*, *short*, *int*, *long*
- two of them represent floating point numbers: *float*, *double*
- one of them represents characters: *char*
- and one of them represents boolean values: *boolean*

Numeric Primitive Data

- The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type		Size	Range of values that can be stored
Integer	byte	1 byte	−128 to 127
	short	2 bytes	−32768 to 32767
	int	4 bytes	−2,147,483,648 to 2,147,483,647
	long	8 bytes	9,223,372,036,854,775,808 to 9,223,372,036,854,755,807
Floating Point	float	4 bytes	3.4e−038 to 3.4e+038
	double	8 bytes	1.7e−308 to 1.7e+038

Boolean Primitive Data

- a boolean value represents a true or false condition
- the reserved words *true* and *false* are the only valid values for a boolean type

Example:

boolean done = false;

- boolean variable can represent any two states such as a light bulb being on or off

Example:

boolean isOn = true;

Characters

- a *char* variable stores a single character
- character literals are delimited by single quotes:

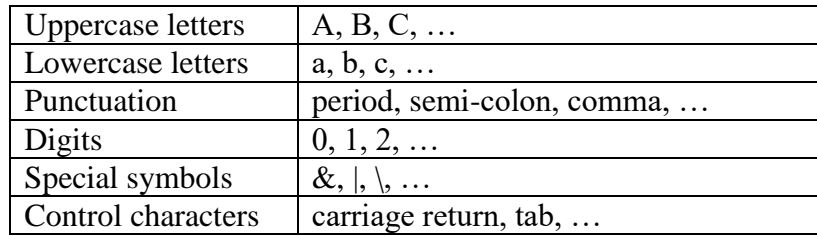
'a' 'X' '7' '\$' ',' '\n'

Example declarations:

char topGrade = 'A';

char terminator = ';', separator = ' ';

- A *character set* is an ordered list of characters, with each character corresponding to a unique number
- A *char* variable in Java can store any character from the *Unicode character set*
- The Unicode character set uses sixteen bits per character, allowing for 65,536 unique characters
- It is an international character set, containing symbols and characters from many world languages
- The *ASCII character set* is older and smaller than Unicode, but is still quite popular (in C programs)
- The ASCII characters are a subset of the Unicode character set, including:





Examples:

```
int count, temp, result;  
double pi, average;  
char cc, mm, tk;
```

Variable Initialization

- assigning a value to a variable for the first time
- a variable can be given an initial value in the declaration with an equal sign

Example:

```
int sum = 0;  
int base = 32, max = 149;
```

- when a variable is referenced in a program, its current value is used
example:

```
int keys = 88;  
System.out.println("A piano has " + keys + " keys");
```

- prints as:
A piano has 88 keys

Assignment Statement

- an *assignment statement* changes the value of a variable
- the equals sign is also the assignment operator

Example:

```
total = 55;  
↑  
└──┘
```

- the expression on the right is evaluated and the result is stored as the value of the variable on the left
- the value previously stored in total is overwritten
- you can only assign a value to a variable that is consistent with the variable's declared type

Constants

- a constant is an identifier that is similar to a variable except that it holds the same value during its entire existence
- as the name implies, it is constant, not variable
- in Java, we use the reserved word `final` in the declaration of a constant

example:

```
final int min_height = 69;
```

- any subsequent assignment statement with `min_height` on the left of the `=` operator will be flagged as an error

Constants are useful for three important reasons

- first, they give meaning to otherwise unclear literal values
 - for example, `NUM_STATES` means more than the literal 50
- second, they facilitate program maintenance
 - if a constant is used in multiple places and you need to change its value later, its value needs to be updated in only one place
- third, they formally show that a value should not change, avoiding inadvertent errors by other programmers



The println Method

- the *System.out* object represents a destination (the monitor screen) to which we can send output

System.out.println ("Whatever you are, be a good one.");

Object Method name Information provided to the method (Parameters)

The print Method

- the *System.out* object provides another method
 - the print method is similar to the *println* method, except that it does not start the next line
 - therefore, any parameter passed in a call to the print method will appear on the same line
- System.out.print* ("Three... ");
System.out.print ("Two... ");
 prints as: Three... Two...

String Concatenation

- the *string concatenation operator* (+) is used to append one string to the end of another
"Peanut butter " + "and jelly"
- it can also be used to append a number to a string
- a string literal cannot be broken across two lines in a program so we must use concatenation
System.out.println("The following facts are for your " + "extracurricular edification");
- the + operator is also used for arithmetic addition
- the function that it performs depends on the type of the information on which it operates
- if both operands are strings, or if one is a string and one is a number, it performs string concatenation
- if both operands are numeric, it adds them
- the + operator is evaluated left to right, but parentheses can be used to force the order
System.out.println("24 and 45 concatenated: " + 24 + 45);
 prints as: 24 and 45 concatenated: 2445
- the + operator is evaluated left to right, but parentheses can be used to force the order
System.out.println("24 and 45 added: " + (24 + 45));

Then concatenation is done Addition is done first

prints as: 24 and 45 added: 69

Escape Sequences

- What if we want to include the quote character itself?
- The following line would confuse the compiler because it would interpret the two pairs of quotes as two strings and the text between the strings as a syntax error:

System.out.println ("I said "Hello" to you.");

A String Syntax Error A String

- An *escape sequence* is a series of characters that represents a special character



- Escape sequences begin with a backslash character (\)

```
System.out.println ("I said \"Hello\" to you.");
```

A String

- Some Java Escape Sequences

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

```
System.out.println("Roses are red,\n\tViolets are blue");
```

Prints as:

*Roses are red,
Violets are blue*

- To put a specified Unicode character into a string using its code value, use the escape sequence: \uhhhh where hhhh are the hexadecimal digits for the Unicode value

Example: Create a string with a temperature value and the degree symbol:

```
double temp = 98.6;
```

```
System.out.println("Body temperature is " + temp + " \u00b0F.");
```

Prints as:

Body temperature is 98.6 °F.

Module 02: Java Operators, Flowcharting Symbols and Sample Programs

Operators

In Java, there are different types of operators. There are arithmetic operators, relational operators, logical operators and conditional operators. These operators follow a certain kind of precedence so that the compiler will know which of the operator to evaluate first in case multiple operators are used in one statement.

Arithmetic Operators

Operator	Operation	Example
+	Addition	$2 + 4 = 6$ $2 + 2.4 = 4.4$



		$5.2 + 7.3 = 13.5$
-	Subtraction	$45 - 90 = -45$ $2.9 - 2.5 = 0.3999999999$ $2.9 - 2 = 0.89999999999$
*	Multiplication	$2 * 7 = 14$ $2.9 * 2 = 5.8$ $2.9 * 2.1 = 6.09$
/	Division	$2 / 7 = 0$ $2.9 / 2 = 1.45$ $2.1 / 2.9 = 0.7241379310$
%	Modulus (Remainder)	$2 \% 7 = 2$ $2.9 \% 2 = 0.8999999999$ $2.9 \% 2.1 = 0.7999999998$

Note:

When evaluating the mod operator with negative integer operands, the answer always takes the sign of the dividend.

Illustration:

$$-34 \% 5 = -4$$

$$34 \% -5 = 4$$

$$-34 \% -5 = -4$$

$$34 \% 5 = 4$$

Increment/Decrement Operators

- Aside from basic arithmetic operators, Java also includes a unary increment operator and unary decrement operator.

- Increment Operator (++)

- increases the value of a variable by 1

Example:

```
int count = 3;
```

```
count = count + 1;
```

// the above statement may be written as the one

//shown below.

```
count++;
```

- Decrement Operator (--)

- decreases the value of a variable by 1

Example:

```
int count = 3;
```

```
count = count - 1;
```

// the above statement may be written as the one

//shown below.

```
count--;
```



Relational Operators

- Relational operators compare two values and determine the relationship between those values.
- The outputs of evaluation are the boolean values true or false.

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

Logical Operators

- Logical operators have Boolean operands that yield a Boolean result.

Operator	Description	Illustration
&&	Logical AND <ul style="list-style-type: none">• Returns True if all of its boolean operands are True, False if otherwise.	True && True = True True && False = False False && False = False
	Logical OR <ul style="list-style-type: none">• Returns True if at least one of its Boolean operands are True, otherwise False	True True = True True False = True False False = False
!	Logical NOT <ul style="list-style-type: none">• Reverses the value of its operand.	!True = False !False = True

Creating a Java Program

Simple Calculation

- Numeric values can be used in calculation using arithmetic operators, such as add (+), subtract (-), multiply (*), divide (/), and modulo (%). An assignment operator (=) is used to assign the result obtained from the calculation to a variable. Parentheses are used to define the order of the calculation.
- The following program computes and prints out the average of the integers from 1 to 10.

```
public class AverageDemo      1
{                               2
    public static void main(String[] args)  3
    {                               4
        double avg, sum;          5
        sum = 1.0+2.0+3.0+4.0+5.0+6.0+7.0+8.0+9.0+10.0;  6
        avg = sum/10;             7
        System.out.println(avg);  8
    }                               9
}                                  10
```

- In the above example, the statement on line 5 declares two variables that are used to store floating point numbers. On line 6, the values from 1.0 to 10.0 are summed together using



the + operator and the resulting value is assigned to sum. On line 7, the value of sum is divided by 10 which to obtain their average. The statement on line 8 just prints the result on screen.

The Program Development Life Cycle

- Programmers do not sit down and start writing code right away when trying to make a computer program. Instead, they follow an organized plan or methodology that breaks the process into a series of tasks.

Here are the basic steps in trying to solve a problem on the computer:

In order to understand the basic steps in solving a problem on a computer, let us define a single problem that we will solve step-by-step as we discuss the problem solving methodologies in detail.

1. Problem Definition

- A programmer is usually given a task in the form of a problem. Before a program can be designed to solve a particular problem, the problem must be well and clearly defined first in terms of its input and output requirements.
- A clearly defined problem is already half the solution. Computer programming requires us to define the problem first before we even try to create a solution.

Let us now define our example problem:

“Create a program that will determine the number of times a name occurs in a list.”

2. Problem Analysis

- After the problem has been adequately defined, the simplest and yet the most efficient and effective approach to solve the problem must be formulated.
- Usually, this step involves breaking up the problem into smaller and simpler sub-problems.

Example Problem:

Determine the number of times a name occurs in a list.

Input Program:

list of names, name to look for

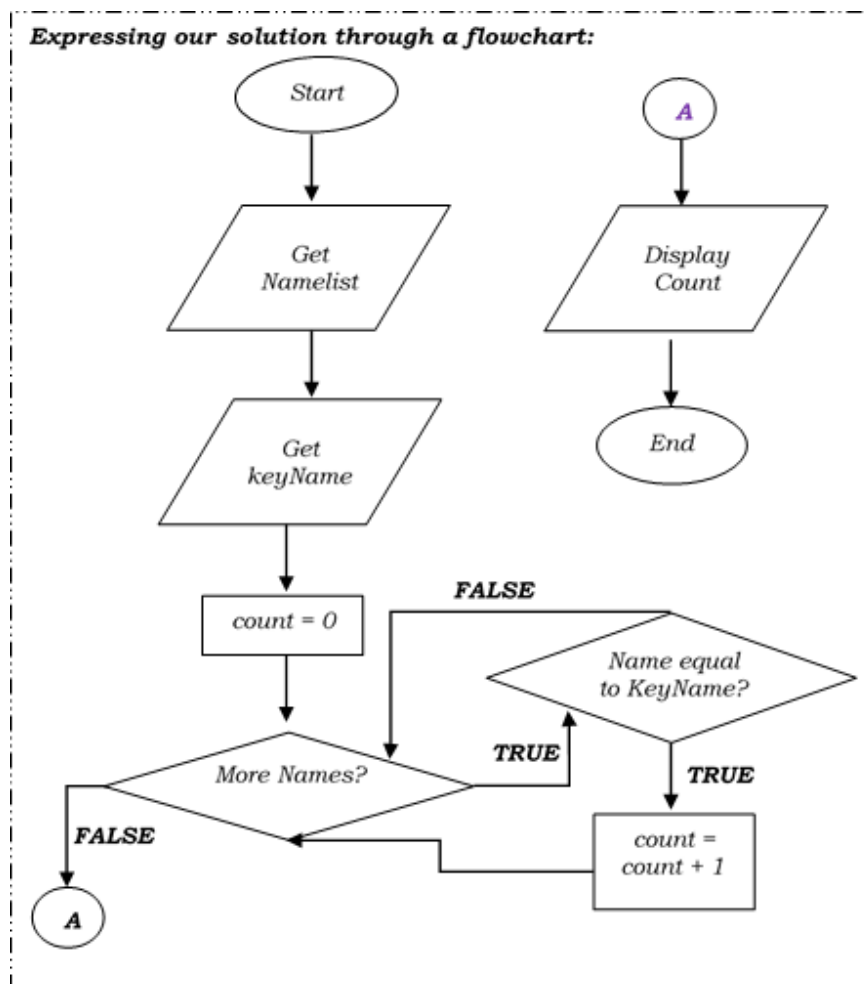
Output Program:

the number of times the name occurs in a list

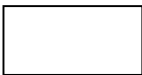

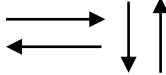
3. Algorithm Design and Representation (flowchart)




- Once our problem is clearly defined, we can set to finding a solution. In computer programming, it is normally required to express our solution in a step-by-step manner.

- An **Algorithm** is a clear and unambiguous specification of the steps needed to solve a problem. It may be expressed in either Human Language (English, Tagalog), through a graphical representation like **flowchart**.
- Now given that the problem is defined, how do we express our general solution in such a way that it is simple yet understandable?



Flowcharting Symbols and Their Meanings

Symbol	Name	Meaning
	Process Symbol	<i>Represents the process of executing a defined operation or groups of operations that results in a change in value, form, or location of information. Also functions as the default symbol when no other symbol is available.</i>
	Input/Output Symbol	<i>Represents as an input/output function, which makes data available for processing (input) or displaying (output) of processed information.</i>
	Flowline Symbol	<i>Represents the sequence of available information and executable operations. The lines connect other symbols, and</i>

		<i>the arrowheads are mandatory only for right-to-left and bottom-to-top flow.</i>
	Decision Symbol	<i>Represents a decision that determines which of a number of alternative paths is to be followed.</i>
	Terminal Symbol	<i>Represents the beginning and the end, or a point of interruption or delay in program.</i>
	Connector Symbol	<i>Represents any entry from, or exit to, another part of the flowchart. Also serves as an off-page connector.</i>

4. Coding and Debugging

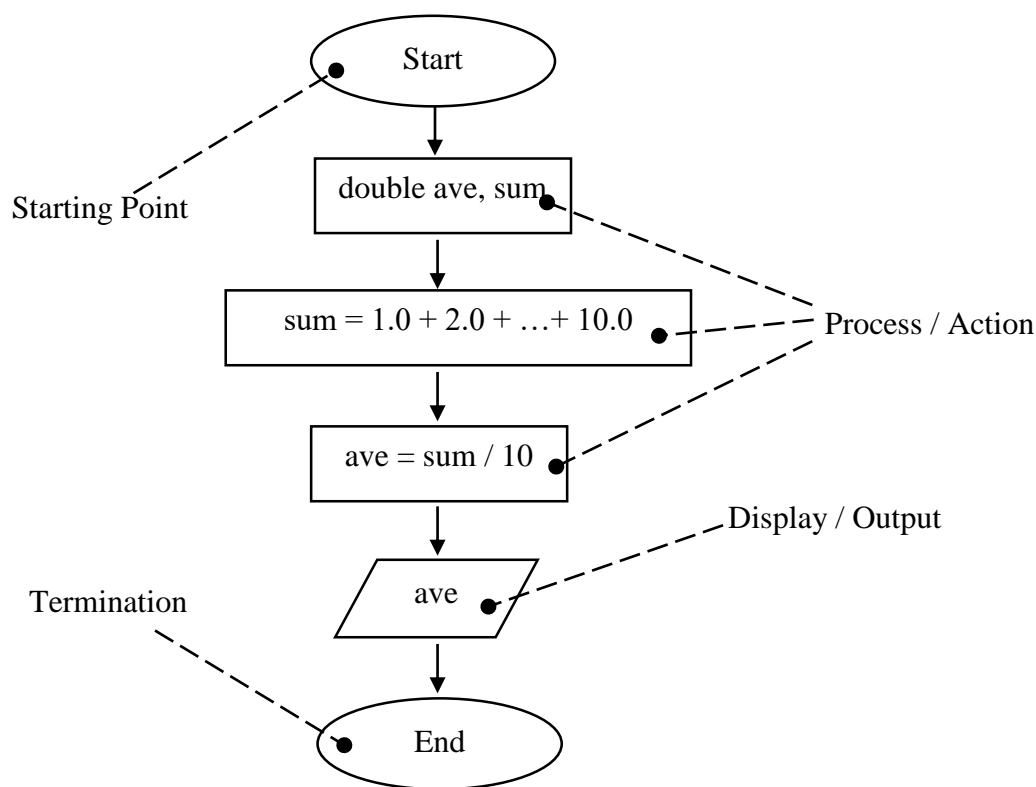
- After constructing the program, it is now possible to create the source code. Using the algorithm as basis, the source code can now be written using the chosen programming language.
- Most of the time, after a programmer has written the program, the program isn't 100% working right away. The programmer has to add some fixes to the program in case of errors (also called bugs) that occurs in the program. This process is called **debugging**.
- Two Types of Program Errors
 - Compile-Time Error
 - Occur if there is a syntax error in the code. The compiler will detect the error and the program won't even compile. At this point, the programmer is unable to form an executable that a user can run until errors are fixed. Forgetting a semi-colon at the end of a statement or misspelling a certain command, for example, is a compile-time error.
 - Runtime Error
 - Compilers aren't perfect and so can't catch all errors at compile time. This is especially true for logic errors such as infinite loops. This type of error is called runtime error. For example, the actual syntax of the code looks okay. But when you follow the code's logic, the same piece of code keeps executing over and over again infinitely so that it loops.

Representing Algorithms Using Flowcharts

- To create a computer program that works, one need not only the knowledge about the rules and syntaxes of a programming language but also a procedure or a process that is used to accomplish the objectives of that program. Such a procedure is called an algorithm. Usually, before creating a computer program, an algorithm is developed based on the objective of the program before the source code is written. An algorithm could be as simple as a single command. More often than not, they are more complex.
- Representing an algorithm using diagrams is useful both in designing an algorithm for a complicate task and for expressing the algorithm to other people. More than one way of creating such diagrams have been created and standardized. One of the simplest ways is to use a flowchart. Although representing an algorithm using a flowchart might not be an

ideal way for some situations, it is the most intuitive and should be sufficient for beginners of computer programming.

- A flowchart needs a starting point of the program it represents and one or more terminations. Steps or commands involved in the program are represented using rectangles containing textual descriptions of the corresponding steps or commands. These steps as well as the starting and terminating points are connected together with line segments in which arrows are used for determining the order of these steps. Shapes are typically placed from the top of the chart to the bottom according to their orders. The starting and terminating points are represented using oval shapes. Different shapes apart from the two shapes already mentioned are defined so that they imply some specific meanings. The following flowchart shows an algorithm of a computer program that prints out the average of the integers from 1 to 10.



Module 03: Getting Program Input and Sequence Problems

- Java provides different ways to get input from the user, the *BufferedReader Class*, the *Console Class* and the *Scanner Class*. The *Scanner Class* is presumably the most favoured technique to take input. The primary reason for the *Scanner class* is to parse primitive composes and strings utilizing general expressions, in any case, it can utilize to peruse



contribution from the client in the order line. In order to use the object of Scanner, the programmer need to import *java.util.Scanner* package.

- *//this line will import the Scanner Class*
import java.util.Scanner;
- *//create an object of the Scanner*
Scanner kbd = new Scanner(System.in);
- *//take input from the user*
int num = kbd.nextInt();

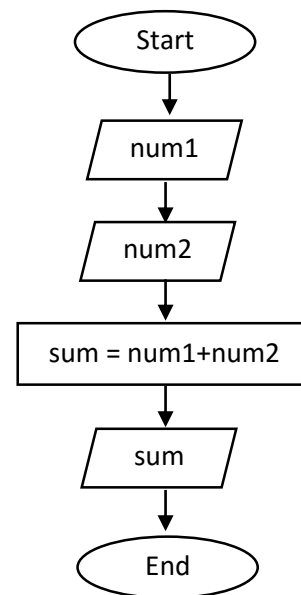
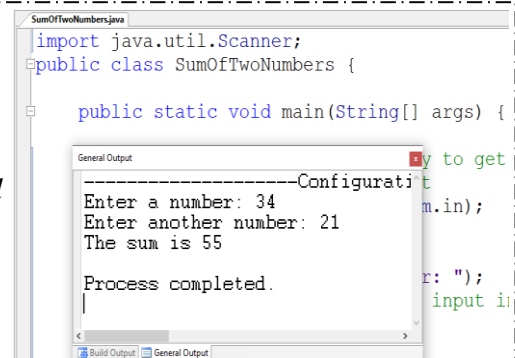
Example: A java program that allows the user to input two numbers. The program then calculates the sum and displays it.

```
//import the Scanner Class
import java.util.Scanner;
public class sumOfTwoNumbers {
    public static void main(String[] args) {
        //set up the program to be ready to get keyboard
        // input by creating the scanner object
        Scanner kbd = new Scanner(System.in);

        //output a prompt message
        System.out.print("Enter a number: ");
        //read user input and store the input in variable num1
        int num1 = kbd.nextInt();
        //output a prompt message
        System.out.print("Enter another number: ");
        //read user input and store the input in variable num2
        int num2 = kbd.nextInt();

        //formula to add the two numbers
        int sum = num1+num2;

        //display the sum
        System.out.println("The sum is "+sum);
    }
}
```



For additional information about the Scanner class: watch
[Module 3: Read Integers and Doubles from Keyboard with Scanner](#)

Problems for Program Development

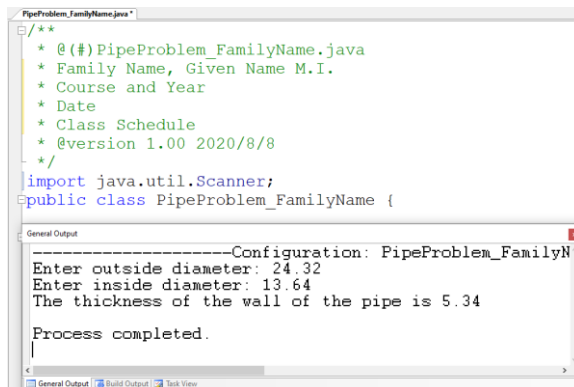
Sequence Programs

1. **Filename: PipeProblem_FamilyName**

Create a java program that will input the outside and the inside diameter (outDia, inDia) of a pipe. Calculate and print the thickness (T) of the wall of the pipe.

$$\text{Formula: } T = \frac{(\text{outDia} - \text{inDia})}{2.0}$$

Sample Output:



```
/**
 * @(#)PipeProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class PipeProblem_FamilyName {

    -----Configuration: PipeProblem_FamilyName
    Enter outside diameter: 24.32
    Enter inside diameter: 13.64
    The thickness of the wall of the pipe is 5.34

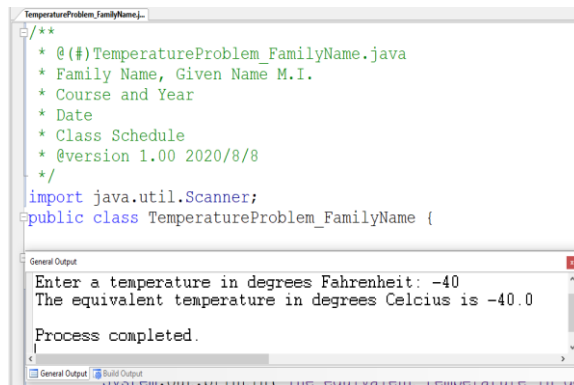
    Process completed.
```

2. **Filename: TemperatureProblem_FamilyName**

Create a java program that will read a temperature in degrees Fahrenheit (F), output in degrees Centigrade (C).

$$\text{Formula to convert degrees Centigrade to degrees Celcius: } F = \frac{9}{5} * (C + 32)$$

Sample Output:



```
/**
 * @(#)TemperatureProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class TemperatureProblem_FamilyName {

    Enter a temperature in degrees Fahrenheit: -40
    The equivalent temperature in degrees Celcius is -40.0

    Process completed.
```

3. **Filename: AngleToRadianProblem_FamilyName**

Create a java program that will read an angle expressed in degrees (deg), minutes (min) and seconds (sec), output in radians (rad).

$$\text{Formula: } rad = \pi * \left(\frac{degdeg + \frac{min}{36} + \frac{sec}{3600}}{180} \right)$$

```
AngleToRadianProblem_FamilyName.java
/**
 * @(#)AngleToRadianProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class AngleToRadianProblem_FamilyName {

    General Output
    -----Configuration: AngleToRadianProblem_FamilyName
    Enter an angle expressed in degrees: 36
    Enter an angle expressed in minutes: 24
    Enter an angle expressed in seconds: 35
    The equivalent angle in radians is 0.6355
    Process completed.
```

4. **Filename: *PowerLossProblem_FamilyName***

Create a java program that will read a current (I_AMP) flowing through a cable and the resistance (R_OHM) of the cable, compute and output the power loss (P_WATT) through the cable.

Formula: $P_WATT = R_OHM * I_AMP^2$

```
PowerLossProblem_FamilyName.java
/**
 * @(#)PowerLossProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class PowerLossProblem_FamilyName {

    General Output
    -----Configuration: PowerLossProblem_FamilyName
    Enter the current flowing through the cable: 6.12
    Enter the resistance flowing through the cable: 2.34
    Power loss is 87.643296
    Process completed.
```

5. **Filename: *ElectricalWireProblem_FamilyName***

An electrical wire supplier sells wire in 500-foot rolls, 300-foot rolls, and 100-foot rolls and the number of feet additional wire. Create a java program that will input the total length of wire needed by a customer, determine and output the number of foot rolls to be given to the customer.

Formula Hint: Apply the concept of modulus operator

Sample output:

```

ElectricalWireProblem_FamilyName.java
/**
 * @(#)ElectricalWireProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class ElectricalWireProblem_FamilyName {

    General Output
    -----Configuration: ElectricalWireProbl
    Enter length of wire needed by a customer: 950
    950 feet requires:
    500 foot roll(s) - 1
    300 foot roll(s) - 1
    100 foot roll(s) - 1
    Add'l feet wire - 1

    Process completed.
  
```

6. **Filename: ChangeProblem_FamilyName**

Input the amount of purchase which is less than P100.00. Create a java program that will calculate the change of P100.00 given by the customer with the following breakdown:

P 50.00 - _____;
 P 20.00 - _____;
 P 10.00 - _____;
 P 5.00 - _____;
 P 1.00 - _____;

Note: Purchases are all in pesos. No centavos.

Formula Hint: Apply the concept of modulus operator

Sample Output:

```

ChangeProblem_FamilyName.java
/**
 * @(#)ChangeProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class ChangeProblem_FamilyName {

    General Output
    -----Configuration: ChangeProblem_Famil
    Enter amount of purchase which is less than 100: 23
    Breakdown of Change:
    P50 - 1
    P20 - 1
    P10 - 0
    P05 - 1
    P01 - 2

    Process completed.
  
```

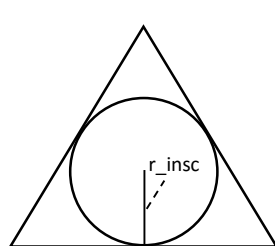
7. **Filename: TriangleProblem_FamilyName**

If a, b and c represent the three sides of a triangle, then the area of the triangle is:

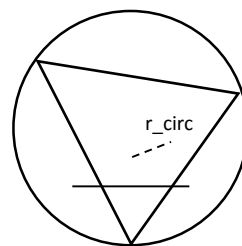
$$A = \sqrt{s * (s - a) * (s - b) * (s - c)} \text{ where: } s = \frac{a+b+c}{2.0}$$

Also the radius of the largest inscribed circle is given by: $r_{insec} = \frac{A}{s}$

and the radius of the smallest circumscribed circle is: $r_{circ} = \frac{a*b*c}{4*A}$



Inscribed Circle



Circumscribed Circle

Create a java program that will input the three sides of a triangle. Calculate and output the area of the triangle, the area of the largest inscribed circle, and the area of the smallest circumscribed circle given a value for a, b and c.

Sample Output:

```
TriangleProblem_FamilyName.java
/**
 * @(#)TriangleProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class TriangleProblem_FamilyName {

    General Output
    Configuration: TriangleProblem_FamilyName
    Enter the three sides of a triangle:
    Enter side a: 3
    Enter side b: 4
    Enter side c: 5
    The area of the triangle is 6.0
    The area of the largest inscribed circle is 3.1416
    The area of the smallest circumscribed circle is 19.6350
    Process completed.
```

8. **Filename: *CellphoneLoadProblem_FamilyName***

A Java program that solves a real-world problem

The remaining amount (balance) for a prepaid cellular phone, is computed by subtracting the cost of the phone usage from the original phone load. The phone usage is based only on the total cost of text messages and phone calls.

The cost of text messages is determined from the number of text messages sent and the cost of phone calls is determined from the number of minutes of calls made.

- Based from the data, write a java program that would compute for the remaining balance on the cellphone load of a subscriber. Let him enter values for the cellphone load, cost per text, cost per minute, number per messages and calls in minutes.
- From the subscriber's inputs, your java program should be able to display the cost of messages, cost of calls, total usage and remaining balance of the cellphone load.

Formula Hint: Use the concept of addition, subtraction and multiplication



Depicted below are sample outputs when the program is executed.

```
/**
 * @(#)CellphoneLoadProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/8
 */
import java.util.Scanner;
public class CellphoneLoadProblem_FamilyName {

    Configuration: CellphoneLoadProblem
    Load: 300
    Cost per text: 1.00
    Cost per minute: 5.50
    Number of messages: 24
    Calls in minutes: 12

    Calls of messages: 24.0
    Calls of calls: 66.0
    total Usage: 90.0
    Balance: 210.0

    Process completed.

    Configuration: CellphoneLoadProblem
    Load: 500
    Cost per text: 0.50
    Cost per minute: 3.50
    Number of messages: 100
    Calls in minutes: 10

    Calls of messages: 50.0
    Calls of calls: 35.0
    total Usage: 85.0
    Balance: 415.0

    Process completed.
```

Module 04: Java Control Structures: Decision Control Structure

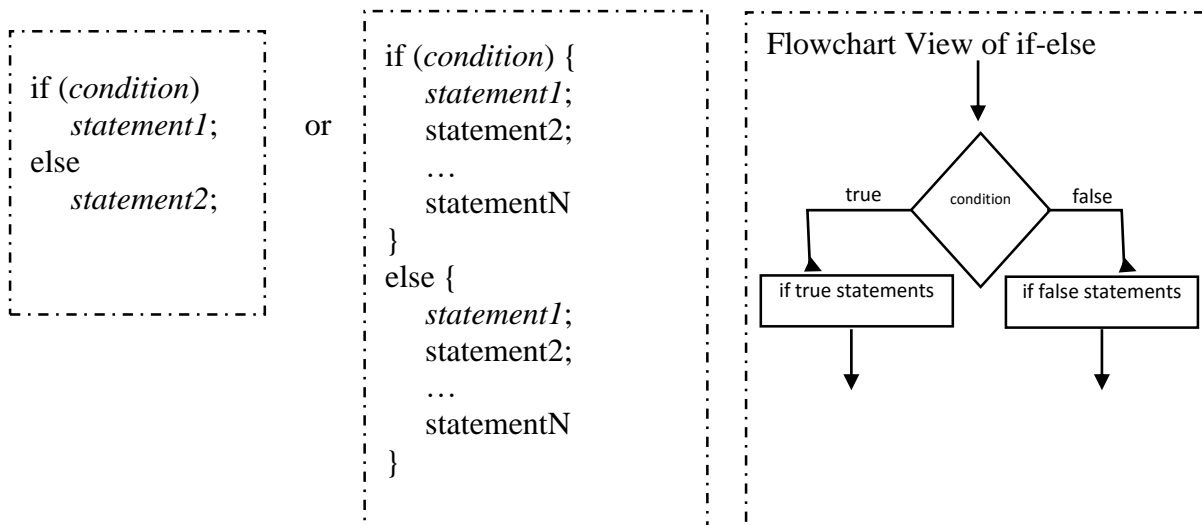
- A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

- Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

If Statement

- The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:



- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a block). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.
- The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;  
// ...  
if(a < b)  
    a = 0;  
else  
    b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero. Most often, the expression used to control the **if** will involve the relational operators.

Nested ifs

- A nested **if** is an **if** statement that is the target of another **if** or **else**. Nested **ifs** are very common in programming. When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i == 10) {  
    if(j < 20)  
        a = b;  
    if(k > 100)  
        c = d; // this if is  
    else
```



```
        a = c; // associated with this else
    }
    else
        a = d; // this else refers to if(i == 10)
```

- As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if ladder*. It looks like this:

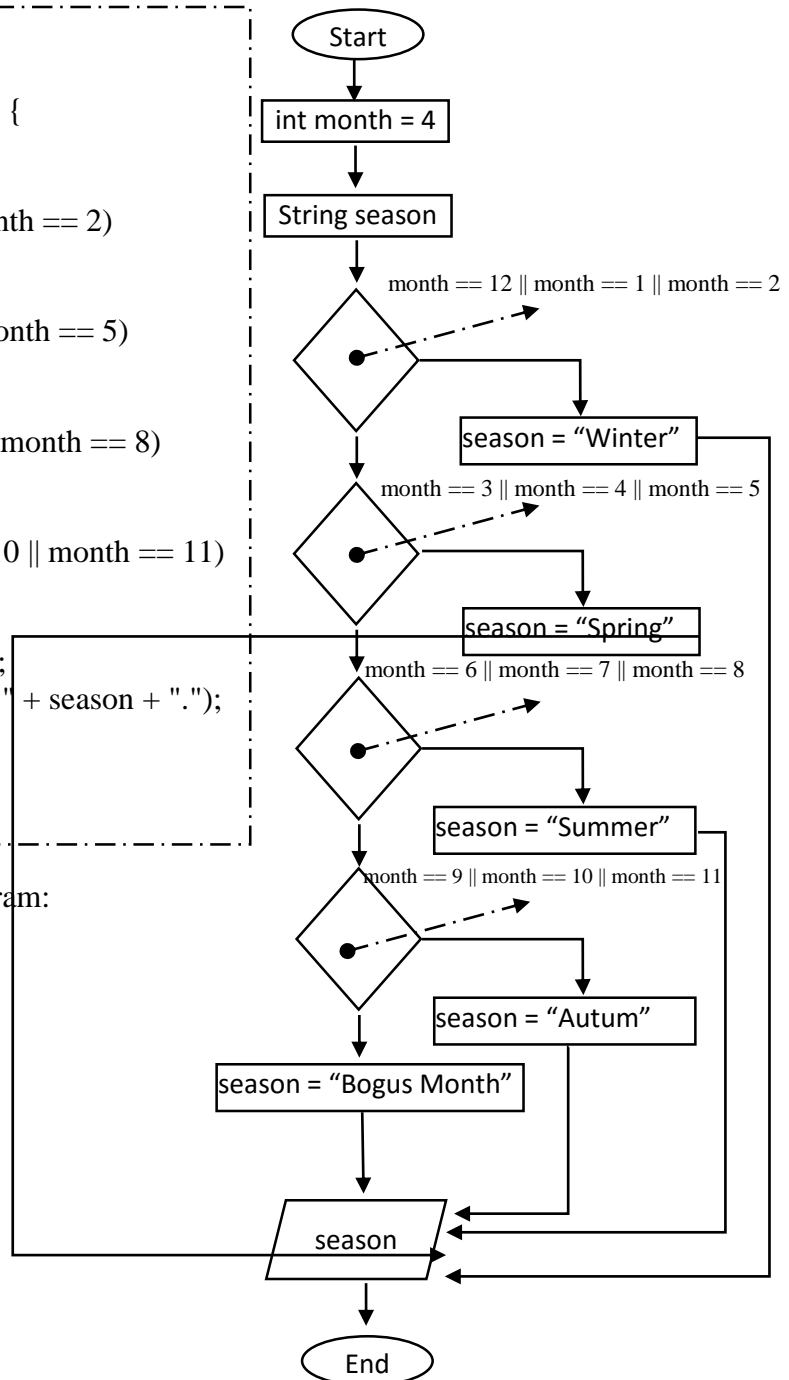
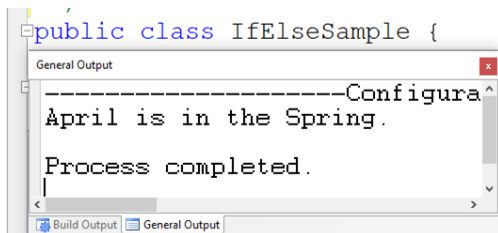
```
if(condition)
    statement;
else
    if(condition)
        statement;
    else
        if(condition)
            statement;
    ...
    else
        statement;
```

- The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is true, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are false, then no action will take place. Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

// Demonstrate if-else-if statements.

```
public class SeasonProblem {
    public static void main(String args[]) {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else
            if(month == 3 || month == 4 || month == 5)
                season = "Spring";
            else
                if(month == 6 || month == 7 || month == 8)
                    season = "Summer";
                else
                    if(month == 9 || month == 10 || month == 11)
                        season = "Autumn";
                    else
                        season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Here is the output produced by the program:



For additional information about java's selection statements: watch the following videos

- [Module 4: If Statement.mp4](#)
- [Module 4: If-else Statement.mp4](#)
- [Module 4: Nested IF Statements.mp4](#)
- [Module 4: The If-Else-If Ladder.mp4](#)



Problems for Program Development

Decision Control Structure

1. TotalResistanceProblem_FamilyName

Create a java program that reads three resistors and the connection desired. Compute and output the total resistance based on desired connection.

Type of Connection	
Series	Parallel
Formula to compute total resistance: $\text{totalResistance} = r_1 + r_2 + \dots + r_N$	Formula to compute total resistance: $\text{totalResistance} = \frac{1}{\left(\frac{1}{r_1} + \frac{1}{r_2} + \dots + \frac{1}{r_3}\right)}$

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter resistor1: **2.34**
Enter resistor2: **4.23**
Enter resistor3: **6.7**
Enter type of connection: **series**
The total resistance is **13.27**

Enter resistor1: **4.21**
Enter resistor2: **2.67**
Enter resistor3: **3.98**
Enter type of connection: **parallel**
The total resistance is **1.158321988654458**

2. ThreeNumbersProblem_FamilyName

Create a java program that reads three distinct numbers (X,Y,Z). Determine and output the following:

- highest number (HN)
- middle number (MN)
- smallest number (SN)
- the numbers in ascending order

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter x: **4**
Enter y: **2**
Enter z: **9**
The Highest Number is **9**
The Median Number is **4**
The Lowest Number is **2**
Ascending Order: **2 4 9**

Enter x: **1**
Enter y: **8**
Enter z: **5**
The Highest Number is **8**
The Median Number is **5**
The Lowest Number is **1**
Ascending Order: **1 5 8**

Enter x: **6**
Enter y: **2**
Enter z: **4**
The Highest Number is **6**
The Median Number is **2**
The Lowest Number is **4**
Ascending Order: **2 4 6**



3. YearProblem_FamilyName

A leap year is a year divisible by 4 unless it is a century year, in which case it must be divisible by 100. Create a java program that reads a year and output a message whether the year is a leap year, century year or ordinary year.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter a year: **2012**
It is a leap year.

Enter a year: **2018**
It is an ordinary year.

Enter a year: **2100**
It is a century year.

4. GradeProblem_FamilyName

A professor converts numeric grades to letter grades in the following way:

Grade	Description
93 – 99	Excellent
87 – 92	Very Good
80 – 86	Good
70 – 79	Fair
65 - 69	Poor

Create a java program that reads a numeric grade and output the equivalent description.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter a grade: **97**
Excellent.

Enter a grade: **79**
Fair.

Enter a grade: **102**
Invalid Input.

5. ParkingFeeProblem_FamilyName

Parking charge per hour at SMBC underground parking is as follows:

P 35.00	- minimum charge for 4 hours parking or less,
P 15.00/hr.	- additional charge in excess of 4 hours parking,
P 250.00	- maximum charge.

Create a java program that reads the number of hours a vehicle was parked. Calculate and output the parking charge.

(Note: Inputs should be integers only)

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter number of hours: **3**
Parking Fee: **P 35**

Enter number of hours: **6**
Parking Fee: **P 65**

Enter number of hours: **23**
Parking Fee: **P 250**



6. TelephoneAreaCodeProblem_FamilyName

A telephone area code is a three-digit number. The first of which is either 1 or 9, the second is any number among 5, 6, 7, 8, and 9. The third digit is any non-zero number. Create a java program that reads a three-digit number, determine and output a message if it is a valid or invalid code.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter telephone area code: **357**
Invalid Code

Enter telephone area code: **163**
Valid Code

7. ElectricBillProblem_FamilyName

The ABC Electric Company bases its electricity charges on two rates. Customers are charged P30.12 per kilowatt-hour (KWH) for the first 400 KWH used in a month, and P25.23 for all KWH used thereafter. Create a java program that reads an electric consumption and output the amount to be charged to the customer.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter electric consumption: **331**
Electric bill is **P9969.72**

Enter electric consumption: **403**
Electric bill is **12123.69**

8. CommodityCodeProblem_FamilyName

A certain store has the following scheme:

Commodity Code:

- A - commodities are discounted by 15%
- B - commodities are taxed by 12%
- C - commodities are charged as priced

Create a program that reads a commodity code, quantity of the commodities bought and the unit price and output the amount to be paid by the customer.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter commodity code: **E**
Invalid Code

Enter commodity code: **A**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P90.53**

Enter commodity code: **B**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P119.28**

Enter commodity code: **A**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P106.50**



Switch

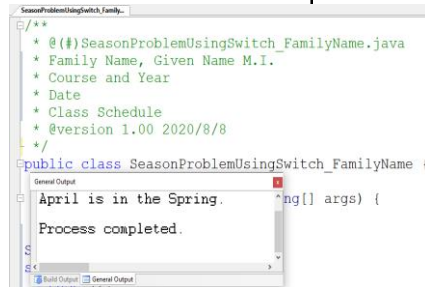
- The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression) {  
    case value1:  
        // statement sequence  
        break;  
    case value2:  
        // statement sequence  
        break;  
    ...  
    case valueN:  
        // statement sequence  
        break;  
    default:  
        // default statement sequence  
}
```

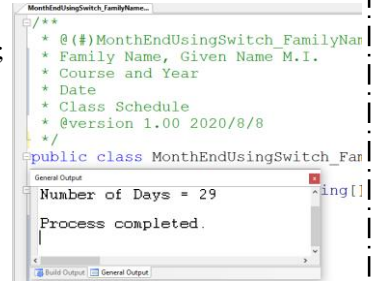
- The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. (An enumeration value can also be used to control a **switch** statement. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.
- The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.
- The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of “jumping out” of the **switch**.
- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **cases** without **break** statements between them.
- The following are simple examples that uses a **switch** statement:



```
//Switch Example1
// An improved version of the season program.
public class SeasonProblemUsingSwitch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
            case 12:
            case 1:
            case 2:
                season = "Winter";
                break;
            case 3:
            case 4:
            case 5:
                season = "Spring";
                break;
            case 6:
            case 7:
            case 8:
                season = "Summer";
                break;
            case 9:
            case 10:
            case 11:
                season = "Autumn";
                break;
            default:
                season = "Bogus Month";
        }
        System.out.println("April is in the " + season + ".");
    }
}
```



```
//Switch Example2
public class MonthEnd {
    public static void main(String[] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;
        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12: numDays = 31;
                    break;
            case 4:
            case 6:
            case 9:
            case 11: numDays = 30;
                    break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```



For additional information about the switch statement, watch:

- [Module 4: The Switch Statement.mp4](#)

Problems for Program Development

Switch Statement

1. GradeProblemUsingSwitch_FamilyName

A professor converts numeric grades to letter grades in the following way:

Grade	Description
93 – 99	Excellent
87 – 92	Very Good
80 – 86	Good
70 – 79	Fair
65 - 69	Poor



Create a java program that reads a numeric grade and output the equivalent description.
Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter a grade: **97**
Excellent.

Enter a year: **79**
Fair.

Enter a year: **102**
Invalid Input.

2. CommodityCodeProblemUsingSwitch_FamilyName

A certain store has the following scheme:

Commodity Code:

- A - commodities are discounted by 15%
- B - commodities are taxed by 12%
- C - commodities are charged as priced

Create a program that reads a commodity code, quantity of the commodities bought and the unit price and output the amount to be paid by the customer.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Enter commodity code: **E**
Invalid Code

Enter commodity code: **B**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P119.28**

Enter commodity code: **A**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P90.53**

Enter commodity code: **A**
Enter quantity of commodity: **2**
Enter unit price: **53.25**
Amount to be paid is **P106.50**

Module 05: Java Control Structures: Iteration Control Structure

- Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

While Loop

- The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {  
    // body of loop/statements  
}
```



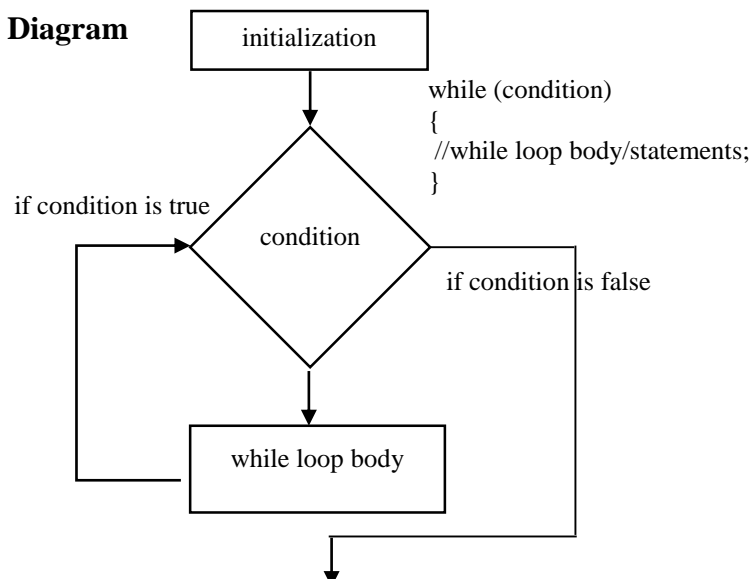
- The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Sample Program:

```
public class WhileLoopExample
{
    public static void main(String args[])
    {
        int num = 0;
        System.out.println("Let's count to 10!");
        while(num < 10)
        {
            num = num + 1;
            System.out.println("Number: " + num);
        }
        System.out.println("We have counted to 10! Hurray! ");
    }
}
```

```
-----Configuratio
Let's count to 10!
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10
We have counted to 10! Hurray!
Process completed.
```

While Loop Flowchart Diagram



For additional information about the while loop: watch
[Module 5: The While Loop.mp4](#)

More While Loop Examples

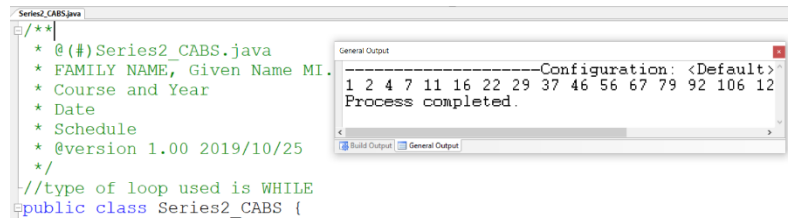
Filename: Series2Problem_FamilyName

Output the set of numbers in the series 1 2 4 7 11 16 ... until 211 is reached.

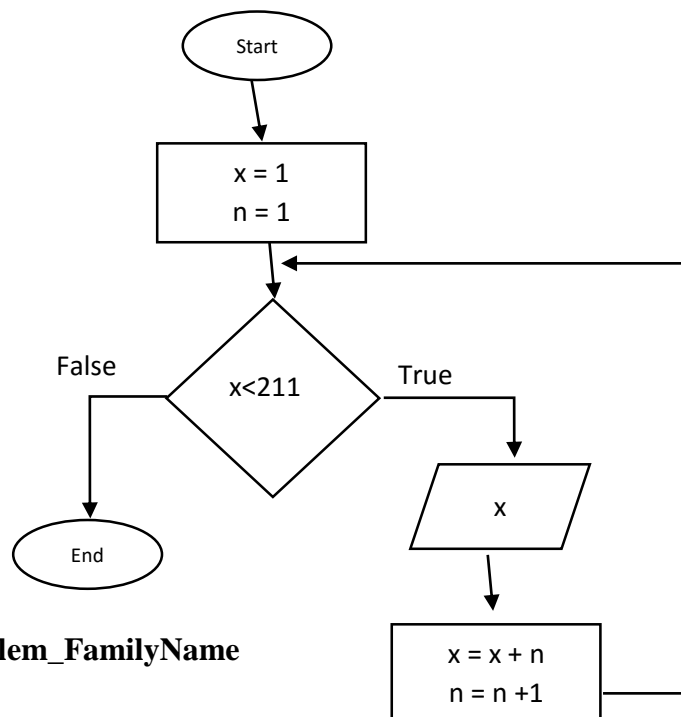
Depicted below is a sample output when the program is executed:

```
General Output
1 2 4 7 11 16 22 29 37 46 56 67 79 92 106 121 137 154 172 191 211
Process completed.
```

```
//type of loop used is WHILE
public class Series2_CABS {
    public static void main(String[] args) {
        //initialize the value of x
        int x = 1;
        //initialize the value of the counter n
        int n = 1;
        //this is the termination test
        //the loop is terminated when the value of x is greater than 211
        while(x<=211){
            //the following block of statements are executed while the termination test is true
            //display the value of x
            System.out.print(x+" ");
            //formula to display the series
            //the previous value of x is added to the increasing value of n
            x = x+n;
            //increment the counter
            n++;
        }
    }
}
```



Flowchart Diagram of Series2 using While Loop



Filename: SphereProblem_FamilyName

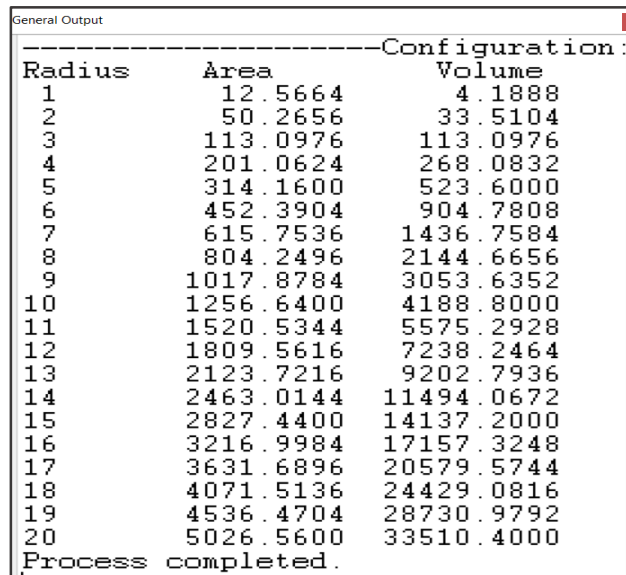


Create a java program that will calculate and output the volume and area of spheres using the formula:

$$V = (4\pi R^3)/3 \quad A = 4\pi R^2$$

where R is the radius of the sphere is from 1 to 20.

Depicted below is a sample output when the program is executed:



Radius	Area	Volume
1	12.5664	4.1888
2	50.2656	33.5104
3	113.0976	113.0976
4	201.0624	268.0832
5	314.1600	523.6000
6	452.3904	904.7808
7	615.7536	1436.7584
8	804.2496	2144.6656
9	1017.8784	3053.6352
10	1256.6400	4188.8000
11	1520.5344	5575.2928
12	1809.5616	7238.2464
13	2123.7216	9202.7936
14	2463.0144	11494.0672
15	2827.4400	14137.2000
16	3216.9984	17157.3248
17	3631.6896	20579.5744
18	4071.5136	24429.0816
19	4536.4704	28730.9792
20	5026.5600	33510.4000

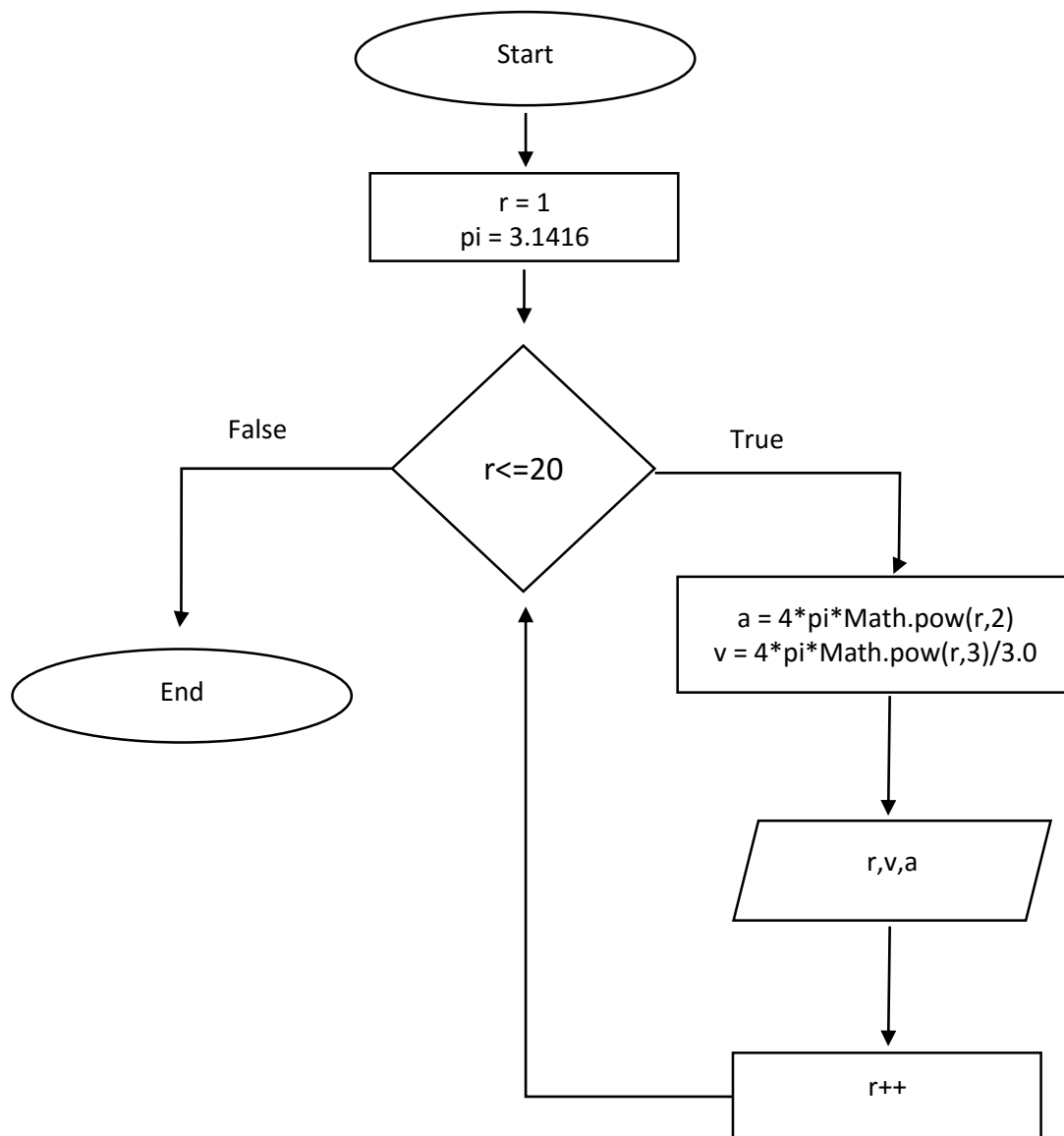
Process completed.

//no need to import the Scanner Class since no input is required

```
public class Sphere {
    public static void main(String[] args) {
        System.out.print("Radius   Area       Volume");
        //declare the value of pi as constant
        final double pi = 3.1416;
        //initialize the value of radius (r)
        int r = 1;
        //this is the termination test/condition. the body of the loop is executed while r<=20
        //at r = 21, the loop is terminated
        while(r<=20){
            //formula to compute the area
            double a = 4*pi*Math.pow(r,2);
            //formula to compute the volume
            double v = 4*pi*Math.pow(r,3)/3.0;
            //display the radius (r), area (a) and volume (v)
            System.out.printf("\n%2d   \t%10.4f\t%10.4f",r,a,v);
            //increment r
            r++;
        }
    }
}
```


Radius	Area	Volume
1	12.5664	4.1888
2	50.2656	33.5104
3	113.0976	113.0976
4	201.0624	268.0832
5	314.1600	523.6000
6	452.3904	904.7808
7	615.7536	1436.7584
8	804.2496	2144.6656
9	1017.8784	3053.6352
10	1256.6400	4188.8000
11	1520.5344	5575.2928
12	1809.5616	7238.2464
13	2123.7216	9202.7936
14	2463.0144	11494.0672
15	2827.4400	14137.2000
16	3216.9984	17157.3248
17	3631.6896	20579.5744
18	4071.5136	24429.0816
19	4536.4704	28730.9792
20	5026.5600	33510.4000

Flowchart Diagram for Sphere Using While Loop





Do-while Loop

- If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {  
    // body of loop  
} while (condition);
```

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Sample Program

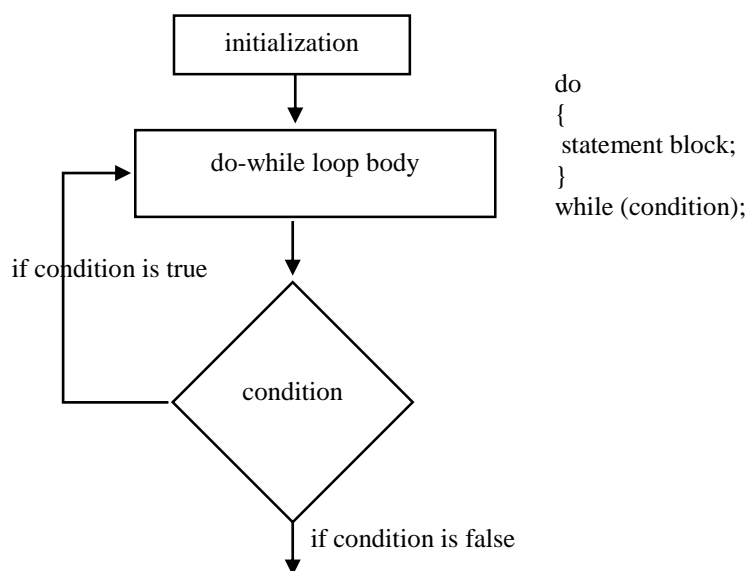
```
import java.util.Scanner;  
public class DoWhileMenuSelection {  
    public static void main(String[] args) {  
        // Using a do-while to process a menu selection  
        Scanner ram = new Scanner(System.in);  
        int choice;  
        do {  
            System.out.println("Help on. Please choose a number:");  
            System.out.println(" 1. if");  
            System.out.println(" 2. switch");  
            System.out.println(" 3. while");  
            System.out.println(" 4. do-while");  
            System.out.println(" 5. for\n");  
            System.out.print("Choose a number: ");  
            choice = ram.nextInt();  
        } while( choice < 1 || choice > 5);  
        switch(choice) {  
            case 1:  
                System.out.println("The if:\n");  
                System.out.println("if(condition) statement;");  
                System.out.println("else statement;"); break;  
            case 2:  
                System.out.println("The switch:\n");  
                System.out.println("switch(expression) {");  
                System.out.println(" case constant;");  
                System.out.println(" statement sequence");  
                System.out.println(" break;");  
                System.out.println(" // ...");  
                System.out.println("}"); break;  
            case 3:  
                System.out.println("The while:\n");  
                System.out.println("while(condition) statement;"); break;  
            case 4:  
                System.out.println("The do-while:\n");  
                System.out.println("do {");  
                System.out.println(" statement;");  
                System.out.println("} while (condition);"); break;  
            case 5:  
                System.out.println("The for:\n");  
                System.out.println("for(init; condition; iteration);");  
                System.out.println(" statement;"); break;  
        }  
    }  
}
```

```
DoWhileMenuSelection.java  
/**  
 * @(#)DoWhileMenuSelection.java  
 * Family Name, Given Name M.I.  
 * Course and Year  
 * Date  
 * Class Schedule  
 * @version 1.00 2020/8/9  
 */  
import java.util.Scanner;  
public class DoWhileMenuSelection {  
    public static void main(String[] args) {  
        // Using a do-while to process a menu selection  
        Scanner ram = new Scanner(System.in);  
        int choice;  
        do {  
            System.out.println("Help on. Please choose a number:");  
            System.out.println(" 1. if");  
            System.out.println(" 2. switch");  
            System.out.println(" 3. while");  
            System.out.println(" 4. do-while");  
            System.out.println(" 5. for\n");  
            System.out.print("Choose a number: ");  
            choice = ram.nextInt();  
        } while( choice < 1 || choice > 5);  
        switch(choice) {  
            case 1:  
                System.out.println("The if:\n");  
                System.out.println("if(condition) statement;");  
                System.out.println("else statement;"); break;  
            case 2:  
                System.out.println("The switch:\n");  
                System.out.println("switch(expression) {");  
                System.out.println(" case constant;");  
                System.out.println(" statement sequence");  
                System.out.println(" break;");  
                System.out.println(" // ...");  
                System.out.println("}"); break;  
            case 3:  
                System.out.println("The while:\n");  
                System.out.println("while(condition) statement;"); break;  
            case 4:  
                System.out.println("The do-while:\n");  
                System.out.println("do {");  
                System.out.println(" statement;");  
                System.out.println("} while (condition);"); break;  
            case 5:  
                System.out.println("The for:\n");  
                System.out.println("for(init; condition; iteration);");  
                System.out.println(" statement;"); break;  
        }  
    }  
}
```

```
General Output  
Help on.  
1. if  
2. switch  
3. while  
4. do-while  
5. for  
  
Choose a number: 4  
The do-while:  
  
do {  
    statement;  
} while (condition);  
  
Process completed.
```

In the program, the **do-while** loop is used to verify that the user has entered a valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the **dowhile** is the perfect loop to accomplish this.

Do-while Loop Flowchart Diagram



For additional information about do-while loop: watch

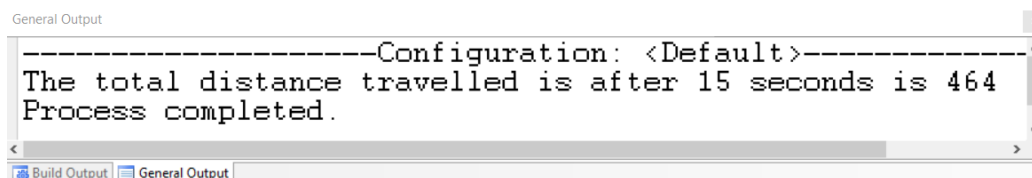
- [Module 5: The Do-While Loop.mp4](#)
- [Module 5: Java printf Method - Displaying data using System.out.printf.mp4](#)
- [Module 5: Print Formatting printf\(\) Conversion Type Characters \(Java\).mp4](#)

More Do While Loop Examples

Filename: ObjectProblem_FamilyName

An object falling from rest in a vacuum falls 16 feet on the first second, 48 feet on the 2nd second, 80 feet on the third second, 112 feet the 4th second and so on. Create a java program that will output the distance traveled by the object after 15 seconds.

Depicted below is a sample output when the program is executed:



```

-----Configuration: <Default>-----
The total distance travelled is after 15 seconds is 464
Process completed.
  
```

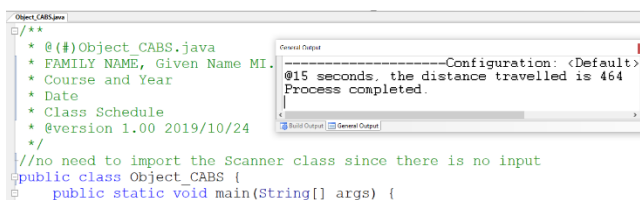
//no need to import the Scanner class since there is no input

```

public class Object_CABS {
    public static void main(String[] args) {
        //s - second, d - distance
        //@s = 1 initial distance is 16
        int s = 1, d = 16;
        do{
  
```

```

        //remove the (//)comment if you want to check the distance travelled every after a second
        //System.out.printf("\n@%2d, the distance travelled is %3d",s,d);
        //formula to solve for the distance since it increases 32ft every second
  
```



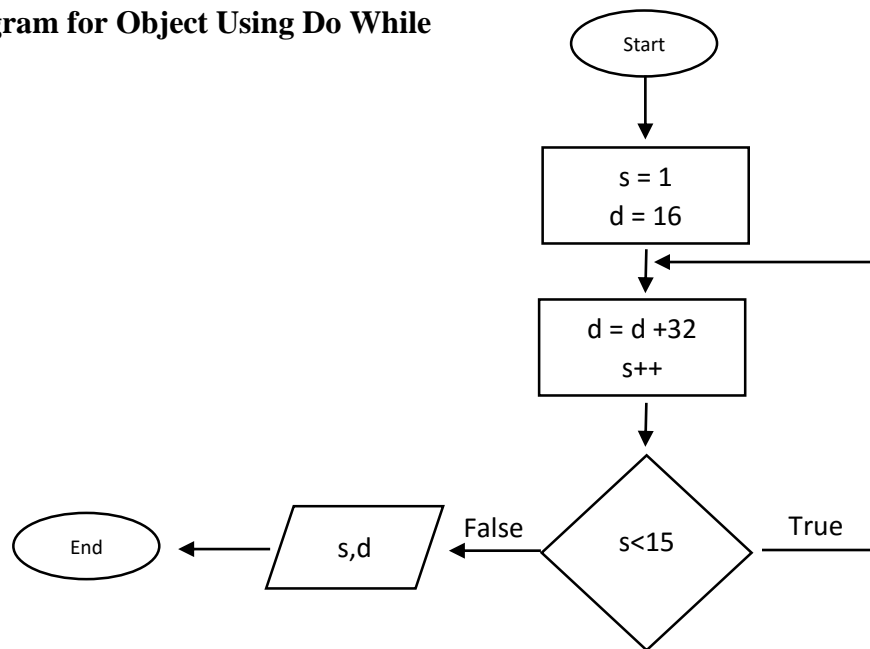
```

Object_CABS.java
* * *
* @(#)Object_CABS.java
* FAMILY NAME, Given Name MI.
* Course and Year
* Date
* Class Schedule
* @version 1.00 2019/10/24
* * *
//no need to import the Scanner class since there is no input
public class Object_CABS {
    public static void main(String[] args) {
  
```



```
        d = d+32;
        //increment s by 1 (last value of s before the loop terminates is 15)
        s++;
    }
    //the loop terminates when s = 15 or more
    while(s<15);
    //display the total distance travelled which is the latest value of d at s second
    System.out.printf("@%2d seconds, the distance travelled is %3d",s,d);
}
}
```

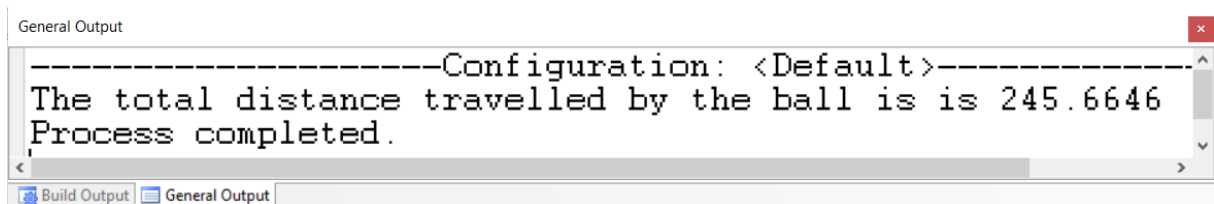
Flowchart Diagram for Object Using Do While



Filename: BounceProblem_Familyname

A ball is dropped from an initial height of 50 feet. If the ball bounces $\frac{2}{3}$ of the previous height, make a program that will calculate and print the total distance traveled by the ball after the 10th bounce.

Depicted below is a sample output when the program is executed:



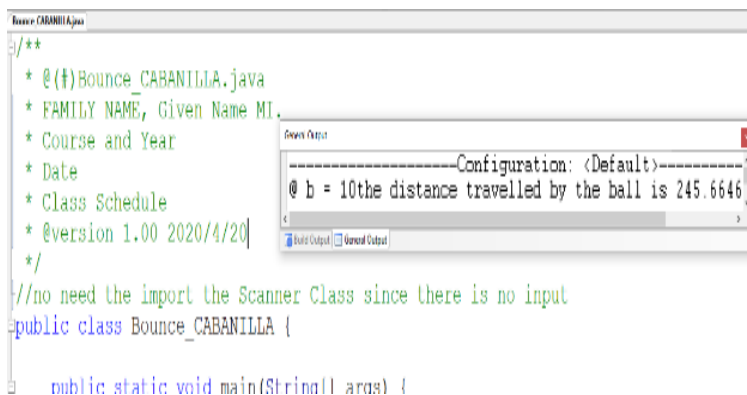
```
//no need the import the Scanner Class since there is no input
public class Bounce_CABANILLA {
    public static void main(String[] args) {
        //in order to better understand this program, please try to solve is manually
        double d = 50, dist = 50.0;
```

```

int b = 1;
do{
    //the ball bounces 2/3 of the previous height
    d = (2.0/3.0)*d;
    //if the ball will bounce it goes up and down so distance travelled is doubled
    dist = dist + 2*d;
    //counter/number of bounce
    b++;
}
//the loop is terminated after the 10th bounce
while(b<=10);
//the 1/2 of the distance on the 10th bounce is subtracted
dist = dist-d;
//output the total distance travelled
System.out.println("@ b = "+(b-1)+ "the distance travelled by the ball is "+dist);
}
}

```

Flowchart Diagram for Bounce using DoWhile

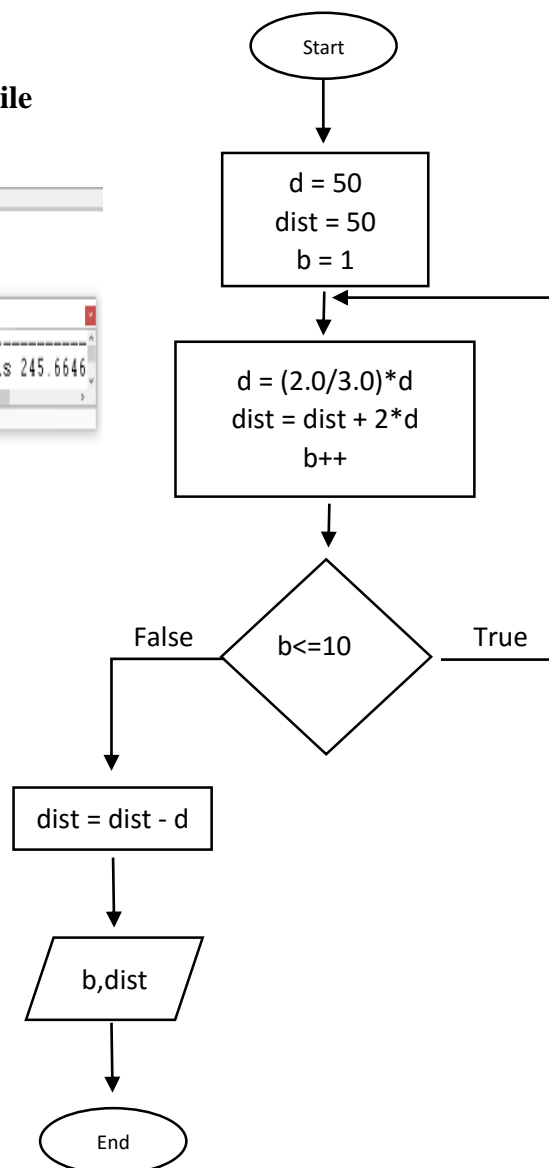


```

/**
 * @(#)Bounce_CABANILLA.java
 * FAMILY NAME, Given Name MI.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/4/20
 */
//no need the import the Scanner Class since there is no input
public class Bounce_CABANILLA {
    public static void main(String[] args) {

```

Output: @ b = 10 the distance travelled by the ball is 245.6646

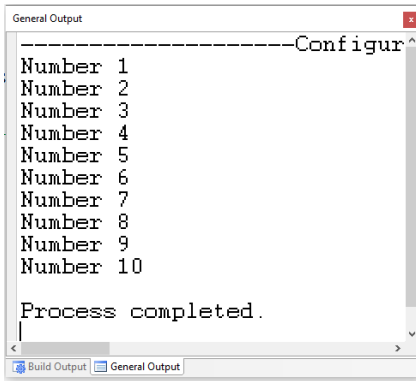


For Loop

- The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.
- Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

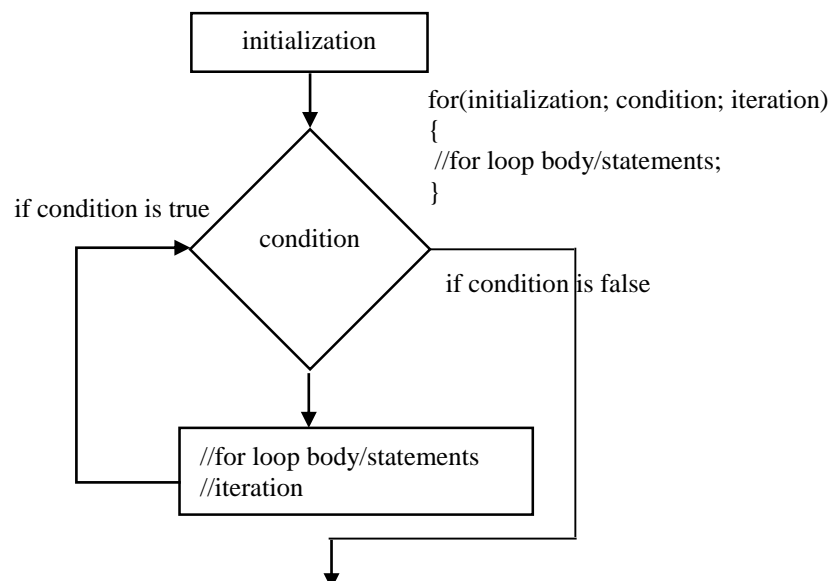
Sample Program

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



A java program using for loop that displays the numbers from 1 to 10;

For Loop Flowchart Diagram





For additional information about the for loop: watch

- [Module 5: The FOR Loop.mp4](#)

More For Loop Examples

FileName: Series3Problem_FamilyName

Create a java program the will output the numbers 1 4 7 ... between 1 and 150.

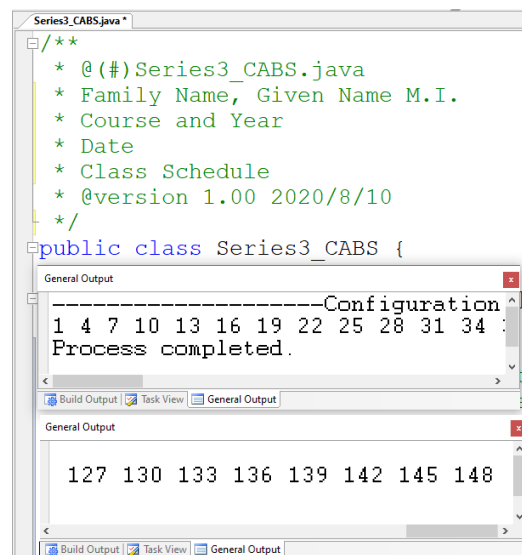
Depicted below is a sample output when the program is executed:

```
General Output
-----Configuration
1 4 7 10 13 16 19 22 25 28
Process completed.
```

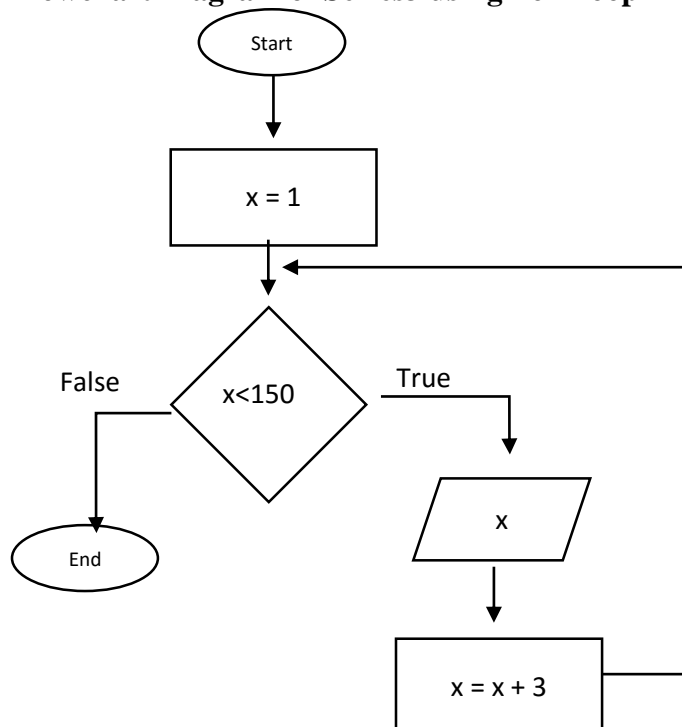
```
General Output
130 133 136 139 142 145 148
```

//no need to import the Scanner Class since there in no input in this program

```
public class Series3_CABS {
    public static void main(String[] args) {
        //initial value of x = 1
        //this loop is terminated when the value of x is greater than 150
        //value of the counter increases by 3
        for(int x = 1; x<=150; x=x+3){
            //output the value of x while x<=150
            System.out.print(x+" ");
        }
    }
}
```



Flowchart Diagram of Series3 using For Loop





FileName: FactorialProblem_FamilyName

Create a java program that reads a number and calculate and output its factorial.

Depicted below are sample outputs when the program is executed (the items in red bold characters are inputted by the user, while the items in blue bold characters are calculated and outputted by the program):

Input a number: **8**
The factorial of 8 is **40320**

Input a number: **4**
The factorial of 8 is **24**

Input a number: **10**
The factorial of 8 is **3628800**

Input a number: **5**
The factorial of 8 is **120**

//the type of loop used here is for

//research on the def'n of Factorial

import java.util.Scanner;

public class FactorialProblem_FamilyName {

public static void main(String[] args) {

Scanner ram = new Scanner(System.in);

//input a number(num) an determine its factorial

System.out.print("Input a number: ");

int num = ram.nextInt();

//initial value of F is 1

int F = 1;

//initial value of the counter x is 1

//when x is greater than the value of num

//then the loop is terminated

for(int x = 1; x<=num; x++){

//formula to determine the factorial as the value of x increases

F = F*x;

}

//the following statement will be executed once the loop is terminated

System.out.println("The factorial of "+num+" is "+F);

}

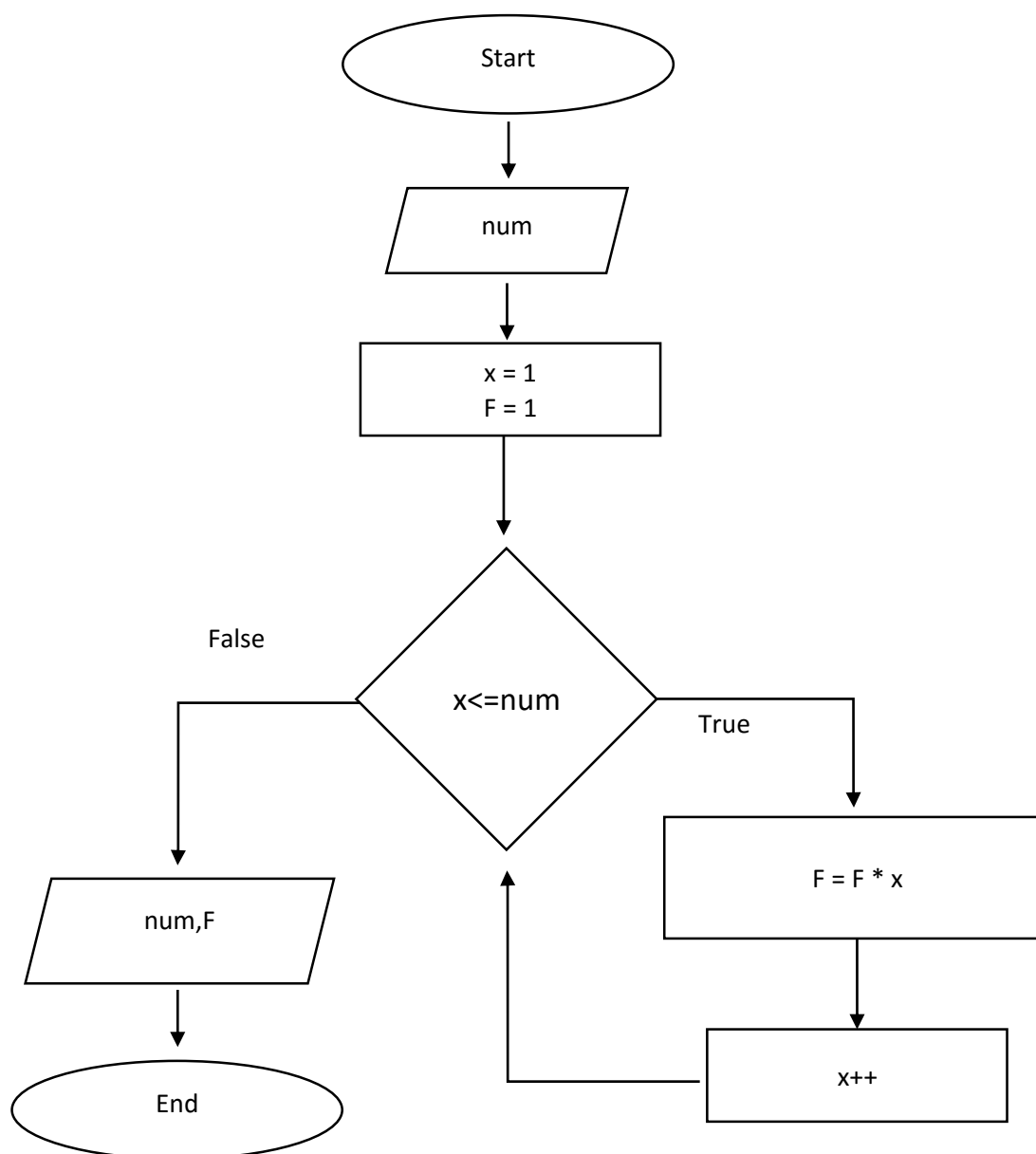
}

```
FactorialProblem_FamilyName.java
/**
 * @(#)FactorialProblem_FamilyName.java
 * Family Name, Given Name M.I.
 * Course and Year
 * Date
 * Class Schedule
 * @version 1.00 2020/8/10
 */
//the type of loop used here is for
//research on the def'n of Factorial
import java.util.Scanner;
public class FactorialProblem_FamilyName {
    public static void main(String[] args) {
        Scanner ram = new Scanner(System.in);
        //input a number(num) an determine its factorial
        System.out.print("Input a number: ");
        int num = ram.nextInt();
        //initial value of F is 1
        int F = 1;
        //initial value of the counter x is 1
        //when x is greater than the value of num
        //then the loop is terminated
        for(int x = 1; x<=num; x++){
            //formula to determine the factorial as the value of x increases
            F = F*x;
        }
        //the following statement will be executed once the loop is terminated
        System.out.println("The factorial of "+num+" is "+F);
    }
}
```

General Output

Input a number: 8
The factorial of 8 is 40320
Process completed.

Flowchart Diagram for Factorial Using For Loop



Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends



			upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	<pre>for(init;condition;iteration) { // code to be executed }</pre>	<pre>while(condition){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(condition);</pre>
Example	<pre>//for loop for(int i=1;i<=10;i++){ System.out.println(i); }</pre>	<pre>//while loop int i=1; while(i<=10){ System.out.println(i); i++; }</pre>	<pre>//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);</pre>
Syntax for infinitive loop	<pre>for(;;){ //code to be executed }</pre>	<pre>while(true){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(true);</pre>

Problems for Program Development

Repetition Control Structure

1. FileName: ReverseProblem_FamilyName (Use While Loop)

Create a java program that reads a number (NUM) and determine its reverse by using operators div and mod. If the last digit is zero, replace it with a one(1) before reversing the number. Output also the sum of all the digits.

Depicted below are sample outputs when the program is executed:

Input a number: **1034**

The reversed order is **4301**

Input a number: **241620**

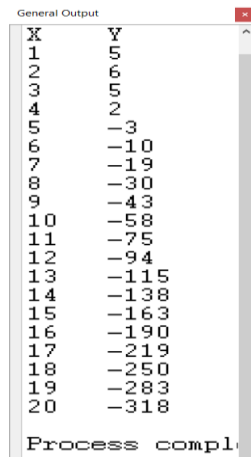
The reversed order is **126142**



2. **FileName: EquationProblem_FamilyName (Use While Loop)**

Using the equation $Y=2+4X-X^2$, create a java program that will compute and output values of Y for values of X from 1 to 20, and in increment of 1.

Depicted below is a sample output when the program is executed:



```
General Output
X      Y
1      5
2      6
3      5
4      2
5     -3
6    -10
7    -19
8    -30
9    -43
10   -58
11   -75
12  -94
13 -115
14 -138
15 -163
16 -190
17 -219
18 -250
19 -283
20 -318
Process completed
```

3. **FileName: Series1Problem_FamilyName (Use Do While Loop)**

The value of S is computed from the formula:

$$S = 1/1 + 1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/N$$

Create a java program that will output the number of terms required and the value of S before S exceeds 3.1.

Example:

1st term: $S = 1$;

2nd term: $S = 1 + 1/2 = 1.5$;

3rd term: $S = 1 + 1/2 + 1/3 = 1.8333$;

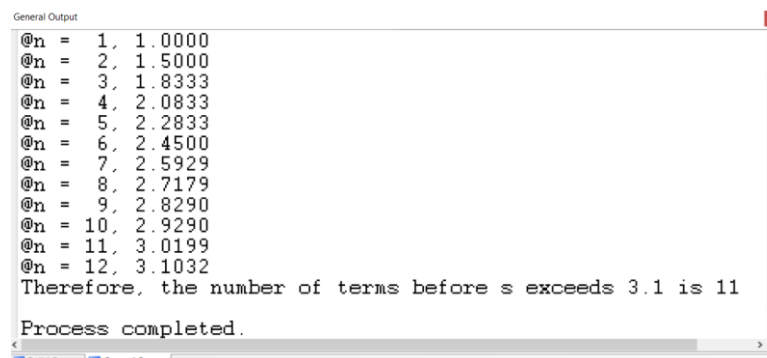
4th term: $S = 1 + 1/2 + 1/3 + 1/4 = 2.08333$;

5th term: $S = 1 + 1/2 + 1/3 + 1/4 + 1/5 = 2.2833$;

...

nth term: $S = ?$

Depicted below is a sample output when the program is executed:



```
General Output
@n = 1, 1.0000
@n = 2, 1.5000
@n = 3, 1.8333
@n = 4, 2.0833
@n = 5, 2.2833
@n = 6, 2.4500
@n = 7, 2.5929
@n = 8, 2.7179
@n = 9, 2.8290
@n = 10, 2.9290
@n = 11, 3.0199
@n = 12, 3.1032
Therefore, the number of terms before s exceeds 3.1 is 11
Process completed.
```



4. **FileName: FibonacciSeries_FamilyName (Use Do While Loop)**

The Fibonacci Sequence is a peculiar series of numbers named after Italian mathematician, known as Fibonacci. Starting with 0 and 1, each new number in the Fibonacci Series is simply the sum of the two before it. Create a java program that will display the first 13 fibonacci numbers.

Depicted below is a sample output when the program is executed:

```
-----Configuration:
0 1 1 2 3 5 8 13 21 34 55 89 144
Process completed.
```

5. **FileName: PrimeOrComposite_FamilyName (Use for Loop)**

Create a java program that reads an integer, determine and output a message if the integer is prime or composite.

Definition:

- Composite number is a positive integer that can have more than 2 factors. Ex : 4,6,8,9 are the example of composite numbers.
- A prime number is a whole number greater than 1 whose only factors are 1 and itself. A factor is a whole number that can be divided evenly into another number. The first few prime numbers are 2, 3, 5, 7, 11, 13, and 17. The number 1 is neither prime nor composite.

Depicted below are sample outputs when the program is executed:

```
-----Configurat
Input an integer: 23
23 is a prime number.
Process completed.
```

```
Input an integer: 14
14 is a composite number.
Process completed.
```

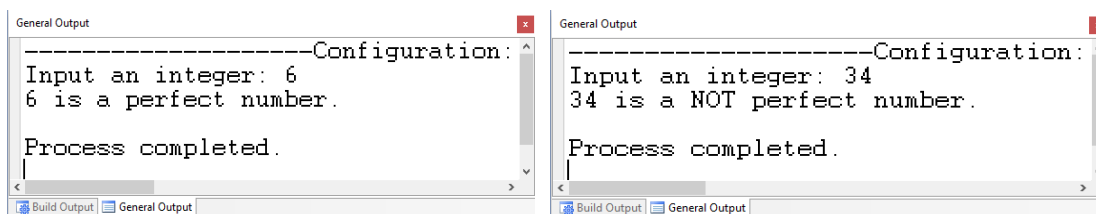
6. **FileName: PerfectNumber_FamilyName (Use for Loop)**

Create a java program that reads an integer, determine and output a message if the integer is a perfect number or not.

Definition: A Perfect Number N is defined as any positive integer where the sum of its factors/divisors minus the number itself equals the number. The first few perfect numbers are 6, 28, 496, and 8128.

Illustrations:		
Number	Factors of the number less than itself	Sum of Factors
6	3, 2, 1	6
28	14, 7, 4, 2, 1	28

Depicted below are sample outputs when the program is executed:



7. FileName: HappyNumber_FamilyName (Use any type of Loop)

Create a java program that reads an integer, determine and output a message if the integer is an Happy Number or not.

Definition:

- The happy number can be defined as a number which will yield 1 when it is replaced by the sum of the square of its digits repeatedly. If this process results in an endless cycle of numbers containing 4, then the number is called an unhappy number.
- Some of the other examples of happy numbers are 7, 28, 32, 49, 100, 320 and so on. The unhappy number will result in a cycle of 4, 16, 37, 58, 89, 145, 42, 20, 4, ...

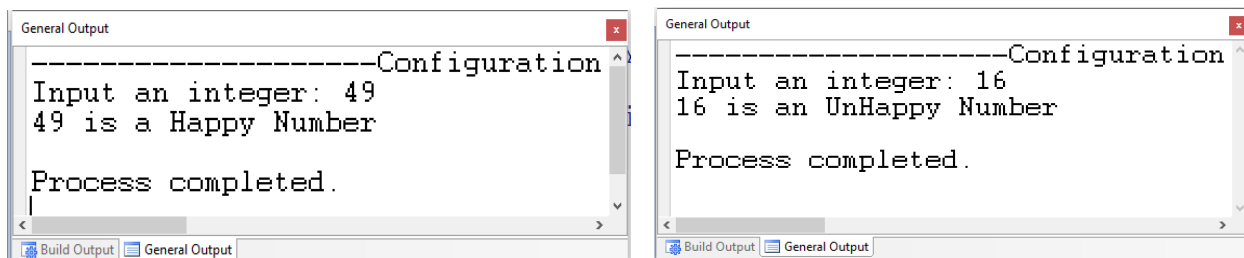
Illustrations:

32	49	4
$3^2 + 2^2 = 13$	$4^2 + 9^2 = 97$	$4^2 = 16$
$1^2 + 3^2 = 10$	$9^2 + 7^2 = 130$	$1^2 + 6^2 = 37$
$1^2 + 0^2 = 1$	$1^2 + 3^2 + 0^2 = 10$	$3^2 + 7^2 = 58$
\therefore Happy Number	$1^2 + 0^2 = 1$	$5^2 + 8^2 = 89$
	\therefore Happy Number	$8^2 + 9^2 = 145$
		$1^2 + 4^2 + 5^2 = 42$
		$4^2 + 2^2 = 20$
		$2^2 + 0^2 = 4$
		\therefore Unhappy Number

Algorithm:

- Determine whether a given number is happy or not.
 - If the number is greater than 0, then calculate remainder rem by dividing the number with 10.
 - Calculate square of rem and add it to a variable sum.
 - Divide number by 10.
 - Repeat the steps from a to c till the sum of the square of all digits present in number has been calculated.
 - Finally, return the sum.
- Define and initialize variable num.
- Define a variable result and initialize it with a value of num.
- If the result is neither equal to 1 nor 4 then, repeat step 1 (a to e).
- Otherwise, if the result is equal to 1 then, given number is a happy number.
- If the result is equal to 4 then, given number is not a happy number.

Depicted below are sample outputs when the program is executed:



8. FileName: ArmstrongNumber_FamilyName (Use any type of Loop)

Create a java program that reads an integer, determine and output a message if the integer is an Armstrong Number or not.

Definition:

- Armstrong number is a number that is equal to the sum of cubes of its digits. For example 0, 1, 153, 370, 371 and 407 are the Armstrong numbers.

Illustrations:

$$153 = (1*1*1) + (5*5*5) + (3*3*3)$$

where:

$$(1*1*1) = 1$$

$$(5*5*5) = 125$$

$$(3*3*3) = 27$$

So:

$$1 + 125 + 27 = 153$$

$$371 = (3*3*3) + (7*7*7) + (1*1*1)$$

where:

$$(3*3*3) = 27$$

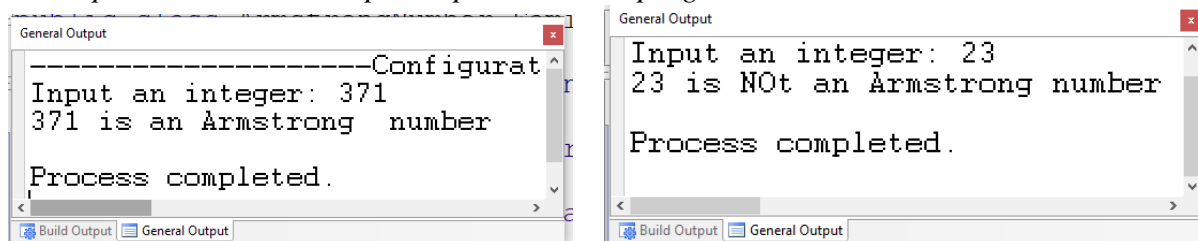
$$(7*7*7) = 343$$

$$(1*1*1) = 1$$

So:

$$27 + 343 + 1 = 371$$

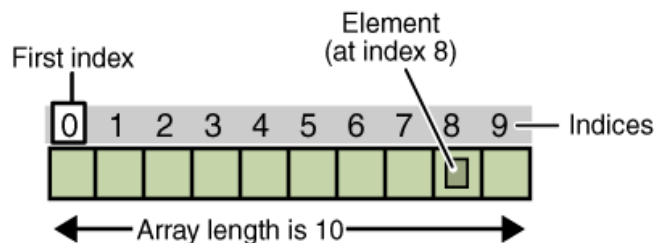
Depicted below are sample outputs when the program is executed:



Module 06: Java Arrays

Arrays

- An array is a collection of objects (consists of a collection of data) that holds a fixed number of values of the same data type.
- Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its numerical index.



One-Dimensional Arrays

- A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

type var-name[]; or *type[] var-name;*

- Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type “array of int”:

int month_days[]; or *int [] month_days;*

- Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory. You will look more closely at **new** later, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

array-var = new type[size];

- Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero. The following example allocates a 12-element array of integers and links them to **month_days**.

month_days = new int[12];

- After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.
- Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**.

month_days[1] = 28;



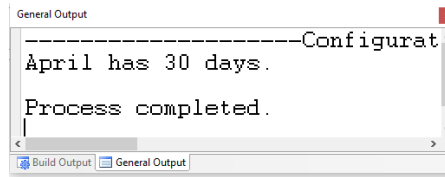
- The next line displays the value stored at index 3.

```
System.out.println(month_days[3]);
```

- Putting together all the pieces, here is a program that creates an array of the number of days in each month.

// Demonstrate a one-dimensional array.

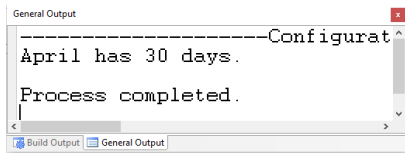
```
public class Array {  
    public static void main(String args[]) {  
        int month_days[];  
        month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        month_days[3] = 30;  
        month_days[4] = 31;  
        month_days[5] = 30;  
        month_days[6] = 31;  
        month_days[7] = 31;  
        month_days[8] = 30;  
        month_days[9] = 31;  
        month_days[10] = 30;  
        month_days[11] = 31;  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```



- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

// An improved version of the previous program.

```
public class AutoArray {  
    public static void main(String args[]) {  
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```



For additional information about one-dimensional arrays: watch
[Module 6: Declaring Arrays & Accessing Elements.mp4](#)

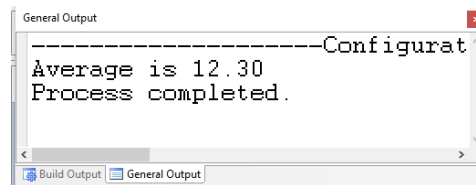
For additional information about one-dimensional arrays and the for loop: watch
[Module 6: Using a Loop to Access an Array.mp4](#)



More examples of One-Dimensional Array

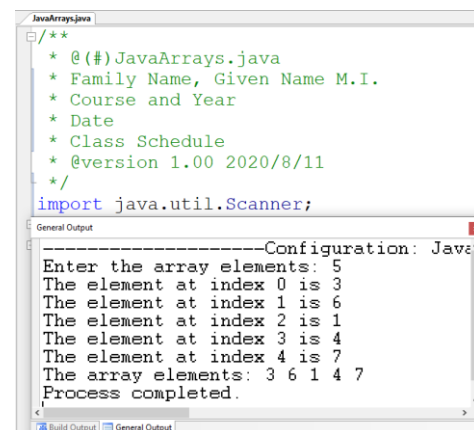
1. A java program that determines and displays the average an array of values.

```
public class Average {  
    public static void main(String args[]) {  
        double arrayKo[] = {10.1, 11.2, 12.3, 13.4, 14.5};  
        double sum = 0, ave;  
        int i;  
        for(i=0; i<5; i++)  
        {  
            sum = sum + arrayKo[i];  
        }  
        ave = sum/5.0;  
        System.out.println("Average is " + ave);  
    }  
}
```



2. A java program that allows the user to manually input the array size and elements in the memory then displays its elements.

```
import java.util.Scanner;  
public class JavaArrays {  
    public static void main(String[] args) {  
        Scanner ram = new Scanner(System.in);  
        System.out.print("Enter the array elements: ");  
        int size = ram.nextInt();  
        //declaring the 1-Dimensional Array  
        int arrayKo[] = new int[size];  
        int i;  
        //accessing the indices in the array  
        //then allocating the elements manually  
        for(i=0; i<size; i++)  
        {  
            System.out.print("The element at index "+i+" is ");  
            arrayKo[i] = ram.nextInt();  
        }  
        System.out.print("The array elements: ");  
        //access each of the indices in the array then display the array elements  
        for(i=0; i<5; i++){  
            System.out.print(arrayKo[i] + " ");  
        }  
    }  
}
```



Problems for Program Development

One-Dimensional Array

1. Create a java program that stores all elements of a 1-dimensional array of numbers at random in the memory. Output the following:



- a) Sum of the list
- b) Average of the list
- c) The biggest element
- d) The smallest element
- e) The median element

Depicted below is a sample output when the program is executed:

The image shows two identical screenshots of a Java IDE's General Output window. The output text is as follows:

```
-----Configuration:
Enter array size: 5
The array elements: 0 4 5 8 3
a. The sum is 20
b. The average is 4.0
c. The biggest element is 3
d. The smallest element is 0
e. The median element is 5

Process completed.
```

2. Create a java program which develops an algorithm that would perform the following routines:
 - 1) Input N digits as the elements of a 1-dimensional array.
 - 2) Print the original array.And the sorted elements in:
 - 3) Ascending order
 - 4) Descending order.Output the original array and the sorted lists.

Depicted below is a sample output when the program is executed:

The image shows a screenshot of a Java IDE's General Output window. The output text is as follows:

```
-----Configuration:
Enter array size: 6
The array elements: 9 6 3 5 6 7
Descending order: 9 7 6 6 5 3
Ascending order: 3 5 6 6 7 9
Process completed.
```

3. Create a java program which develops an algorithm that would perform the following routines:
 - a. Input array size.
 - b. Read in N integers as elements 1-dimensional array.
 - c. Display the original array.
 - d. Accept a digit and remove it from the list and output the new array.
 - e. Accept a digit and its subscript. Include that input in the list by inserting that digit in the specified place. It is necessary to adjust the other elements.

Depicted below is a sample output when the program is executed:

```

General Output
-----Configuration: <Default>
a.
Enter the size of the array: 5
b.
The element at index 0 is 3
The element at index 1 is 7
The element at index 2 is 1
The element at index 3 is 5
The element at index 4 is 2
c.
The array elements: 3 7 1 5 2
d.
Input a digit found in the array: 5
The new array is 3 7 1 2
e.
Enter a number to be inserted: 24
Enter its subscript: 4
The new array(V2.0): 3 7 1 5 24
Process completed.

```

4. Create a java program that inputs N digits as the elements of a 1-dimensional array. Accept a digit and search it in the array. Output the subscript of the element.

Depicted below is a sample output when the program is executed:

```

General Output
-----Configuration: <Default>
The array size: 8
Array elements: 4 0 4 9 0 4 1 0
Enter an element found in the array: 4
The index/ces of efound is/are: 0 2 5
Process completed.

```

Multidimensional Arrays

- In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[5][5];
```

- The above code allocates a 5 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array of arrays* of **int**.
- The following program randomly assign numbers as elements in the array from left to right, top to bottom, and then displays these values:

// Demonstrate a two-dimensional array using the Random Class.
//import the Random Class

```

import java.util.Random;
public class TwoDArray {
    public static void main(String[] args) {
        //set up the program to be ready to accept
        //a random input
    }
}

```

```

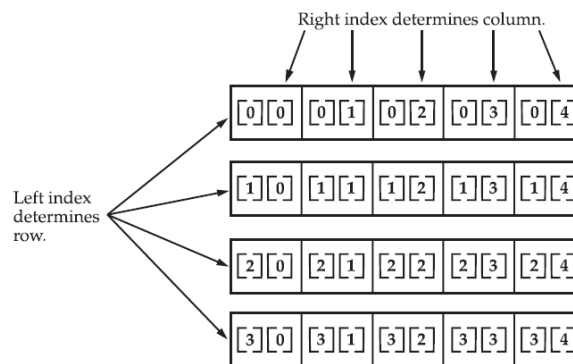
General Output
-----Configuration: <Default>
The array elements:
1 6 0 7 8
9 0 7 2 7
1 9 9 5 8
8 9 1 0 6
9 2 0 6 6
Process completed.

```



```
Random rand = new Random();
//declaring the 2-Dimensional Array (5x5)
int twoD[][]= new int[5][5];
int i, j, k = 0;
System.out.println("The array elements: ");
//allocating the elements of a 5 by 5 array using the random class
for(i=0; i<5; i++){
    for(j=0; j<5; j++){
        //random number from 0 to 9 will be store at index twoD[i][j]
        twoD[i][j] = rand.nextInt(10);
    }
}
//access each of the indices in the array then display the array elements
for(i=0; i<5; i++) {
    for(j=0; j<5; j++){
        System.out.print(twoD[i][j] + " ");
    }
    System.out.println();
}
}
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. The following figure is a conceptual view of a 4 by 5, two-dimensional array.



Given: `int twoD [] [] = new int [4] [5];`

For additional information about multidimensional arrays: watch
[Module 6: Two Dimensional Arrays.mp4](#)

Problems for Program Development Two-Dimensional Array

- Create a java program that reads elements (randomly from 0 – 9) of a 2-dimensional array (5x5) using the Random Class then perform the following:



- 1) Output the array elements
- 2) Output the sum of each row.
- 3) Output the sum of each column.
- 4) Output the sum of all the elements.
- 5) Output the sum of prime numbers in the array
- 6) Output the sum of the elements in the main diagonal.
- 7) Output the elements below the diagonal.
- 8) Output the elements above diagonal.
- 9) Output the sum of odd numbers below the diagonal.
- 10) Output the sum of even numbers above the diagonal.

Depicted below is a sample output when the program is executed:

```
General Output
-----Configuration: TwoDimArray_FamilyName - J
1. Output the array elements
The array elements:
6 4 2 6 7
7 2 7 9 9
8 9 4 6 0
6 2 0 3 4
5 4 4 8 6
2. Output the sum of each row
The sum of row 1 is 25
The sum of row 2 is 34
The sum of row 3 is 27
The sum of row 4 is 15
The sum of row 5 is 27
3. Output the sum of each column
The sum of column 1 is 32
The sum of column 2 is 21
The sum of column 3 is 17
The sum of column 4 is 32
The sum of column 5 is 26
4. Output the sum of all the elements
The sum of all the elements is 128
5. Output the sum of prime numbers in the array
The sum of prime numbers in the array is 35
6. Output the sum of the elements in the main diagonal
The sum of the elements in the main diagonal is 21
7. Output the sum of the elements below the main diagonal
The sum of the elements below the main diagonal is 53
8. Output the sum of the elements above the main diagonal
The sum of the elements above the main diagonal is 54
9. Output the sum of odd elements below the main diagonal
The sum of odd elements below the main diagonal is 21
10. Output the sum of even elements above the main diagonal
The sum of even elements above the main diagonal is 22
Process completed.
```

Module 07: Java Methods

- A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println` method, for example, the system actually executes several statements in order to display a message on the console.



Creating Method

- Considering the following example to explain the syntax of a method:

```
public static int methodName(int a, int b) {  
    // body  
}
```
- Here,
 public static : modifier.
 int: return type
 methodName: name of the method
 a, b: formal parameters
 int a, int b: list of parameters
- Method definition consists of a method header and a method body. The same is shown below:

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```
- The syntax shown above includes:
 modifier: It defines the access type of the method and it is optional to use.
 returnType: Method may return a value.
 nameOfMethod: This is the method name. The method signature consists of the method name and the parameter list.
 Parameter List: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
 method body: The method body defines what the method does with statements.
- **Example**: Here is the source code of the above defined method called max. This method takes two parameters num1 and num2 and returns the minimum between the two:

```
/*  
 * the snippet returns the minimum between two numbers  
 */  
public static int minFunction(int n1, int n2) {  
    int min;  
    if (n1 > n2){  
        min = n2;  
    }  
    else{  
        min = n1;  
    }  
    return min;  
}
```

Method Calling:

- For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing *noreturnvalue*. The process of method calling is simple. When a program invokes a method, the program control gets transferred



to the called method. This called method then returns control to the caller in two conditions, when:

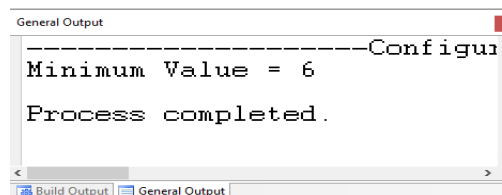
- return statement is executed.
- reaches the method ending closing brace.
- The methods returning void is considered as call to a statement. Let is consider an example:
System.out.println("Calling a method!");
- The method returning value can be understood by the following example:
int result = sum (6, 9);

Example:

- The following is an example which demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber_Method{
    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }
    //returns the minimum of two numbers
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;
        return min;
    }
}
```

This would produce the following result:



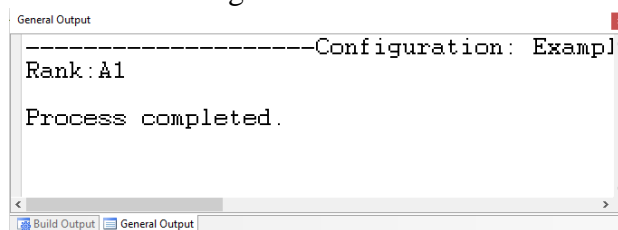
The void Keyword:

- The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints255.7;*. It is a Java statement which ends with a semicolon as shown in the following example.
- **Example:**



```
public class ExampleVoid_Method {  
    public static void main(String[] args) {  
        methodRankPoints(255.7);  
    }  
    public static void methodRankPoints(double points) {  
        if (points >= 202.5) {  
            System.out.println("Rank:A1");  
        }  
        else  
            if (points >= 122.4) {  
                System.out.println("Rank:A2");  
            }  
            else {  
                System.out.println("Rank:A3");  
            }  
        }  
    }  
}
```

This would produce the following result:



Passing Parameters by Value:

- While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference. Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter.

- **Example:**

The following program shows an example of passing parameter by value. The values of the arguments remain the same even after the method invocation.

```
public class SwappingExample_Method {  
    public static void main(String[] args) {  
        int a = 30;  
        int b = 45;  
        System.out.println("Before swapping, a = " + a + " and b = " + b);  
        // Invoke the swap method  
        swapFunction(a, b);  
        System.out.println("\n*Now, Before and After swapping values will be same*");  
        System.out.println("After swapping, a = " + a + " and b is " + b);  
    }  
    public static void swapFunction(int a, int b) {  
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);  
        // Swap n1 with n2  
    }  
}
```




```
int c = a;
a = b;
b = c;
System.out.println("After swapping(Inside), a = " + a + " b = " + b);
}
}
```

This would produce the following result:

```
-----Configuration: SwappingExample_Method - JI
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

*Now, Before and After swapping values will be same here*
After swapping, a = 30 and b is 45

Process completed.
```

Method Overloading:

- When a class has two or more methods by same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has same method name, type, number of parameters etc. Let us consider the example shown before for finding minimum numbers of integer type. If we want to find minimum number of double type, then the concept of Overloading will be introduced to create two or more methods with the same name but different parameters. The below example explains the same:

```
public class ExampleOverloading_Method{
    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        double c = 7.3;
        double d = 9.4;
        int result1 = minFunction(a, b);
        //same function name with different parameters
        double result2 = minFunction(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
    }
    //for integer
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;
        return min;
    }
    //for double
    public static double minFunction(double n1, double n2) {
```



```
double min;  
if (n1 > n2)  
    min = n2;  
else  
    min = n1;  
return min;  
}  
}
```

This would produce the following result:

- Overloading methods makes program readable. Here, two methods are given same name but with different parameters. The minimum number from integer and double types is the result.

The Constructors

- A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object. All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

- **Example:**

Here is a simple example that uses a constructor without parameters:

//A simple constructor.

```
class MyClass {  
    int x;  
    //Following is the constructor  
    MyClass() {  
        x = 10;  
    }  
}
```

- You would call constructor to initialize objects as follows:

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
    }  
}
```



```
        MyClass t2 = new MyClass();
        System.out.println(t1.x + " " + t2.x);
    }
}
```

Parametarized Constructor

- Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

- **Example:**

Here is a simple example that uses a constructor with parameter:

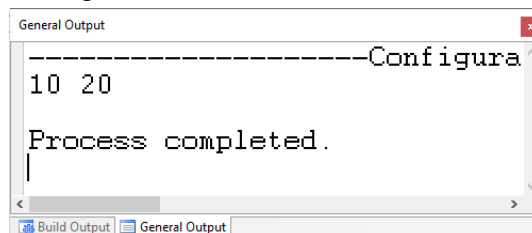
// A simple constructor.

```
class MyClass {
    int x;
    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

This would produce the following result:



The this keyword

- **this** is a keyword in Java which is used as a reference to the object of the current class, within an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods. **Note:** The keyword *this* is used only within instance methods or constructors.
- In general the keyword *this* is used to :
 - Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student{
    int age;
```



```
        Student(int age){
            this.age=age;
        }
    }
    ○ Call one type of constructor parametrized constructor or default from other in a class.
    It is known as explicit constructor invocation .
    class Student{
        int age
        Student(){
            this(20);
        }
        Student(int age){
            this.age=age;
        }
    }
```

- **Example**

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the below given program in a file with name Using_this_Example.java

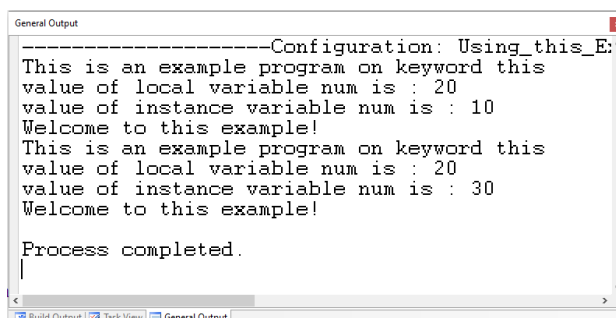
```
public class This_Example {
    //Instance variable num
    int num =10;
    This_Example(){
        System.out.println("This is an example program on keyword this ");
    }
    this_Example(int num ){
        //Invoking the default constructor
        this();
        //Assigning the local variable num to the instance variable num
        this.num =num ;
    }
    public void greet(){
        System.out.println("Welcome to this example!");
    }
    public void print(){
        //Local variable num
        int num =20;
        //Printing the instance variable
        System.out.println("value of local variable num is : "+num );
        //Printing the local variable
        System.out.println("value of instance variable num is : "+this.num );
        //Invoking the greet method of a class
        this.greet();
    }
    public static void main(String[] args){
        //Instantiating the class
```

```

Using_this_Example obj1=new Using_this_Example();
//Invoking the print method
obj1.print();
//Passing a new value to the num variable through parameterized constructor
Using_this_Example obj2=new Using_this_Example(30);
//Invoking the print method again
obj2.print();
}
}

```

This would produce the following result:



```

-----Configuration: Using_this_E
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Welcome to this example!
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Welcome to this example!

Process completed.

```

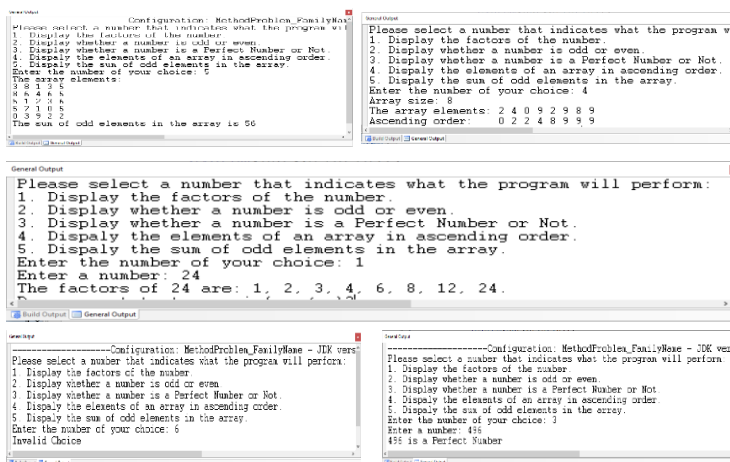
For additional information about multidimensional arrays: watch

- [Module 7: Introduction To Methods.mp4](#)
- [Module 7: Adding Parameters to a Method & Returning Values.mp4](#)

Problems for Program Development

1. Create a java program that will display a menu for choices to perform the following routines:
 1. Input a number then determine and display the factors of the number.
 2. Input a number then determine and display if the number is odd or even.
 3. Input a number then determine and display if the number is a Perfect Number or Not.
 4. Input the elements of a 1-dimensional array using the Random Class (random elements from 0 to 9 and random size from 0 to 10) then display the sorted elements in ascending order.
 5. Input the elements (from 0 to 9) of a (5x5) 2-dimensional array using the Random Class then display the sum of the of all odd elements in the array.

Depicted below is a sample output when the program is executed:



```

-----Configuration: MethodProblems_FamilyWise
Please select a number that indicates what the program will perform:
1. Display the factors of the number.
2. Display whether a number is odd or even.
3. Display whether a number is a Perfect Number or Not.
4. Display the elements of an array in ascending order.
5. Display the sum of odd elements in the array.
Enter the number of your choice: 2
The array elements:
2 1 5 5
6 4 6 5
1 7 6 5
3 9 2
The sum of odd elements in the array is 56

-----Configuration: MethodProblems_FamilyWise - JDK vers
Please select a number that indicates what the program will perform:
1. Display the factors of the number.
2. Display whether a number is odd or even.
3. Display whether a number is a Perfect Number or Not.
4. Display the elements of an array in ascending order.
5. Display the sum of odd elements in the array.
Enter the number of your choice: 1
Enter a number: 24
The factors of 24 are: 1, 2, 3, 4, 6, 8, 12, 24.

-----Configuration: MethodProblems_FamilyWise - JDK vers
Please select a number that indicates what the program will perform:
1. Display the factors of the number.
2. Display whether a number is odd or even.
3. Display whether a number is a Perfect Number or Not.
4. Display the elements of an array in ascending order.
5. Display the sum of odd elements in the array.
Enter the number of your choice: 0
Invalid Choice

-----Configuration: MethodProblems_FamilyWise - JDK vers
Please select a number that indicates what the program will perform:
1. Display the factors of the number.
2. Display whether a number is odd or even.
3. Display whether a number is a Perfect Number or Not.
4. Display the elements of an array in ascending order.
5. Display the sum of odd elements in the array.
Enter the number of your choice: 3
Enter a number: 456
456 is a Perfect Number

```



Module 08: Java Wrapper Classes

We've already established that Java, being an object-oriented programming language, is all about objects or "Classes". Certain data structures, such as ArrayLists, LinkedLists, HashMap, etc. provide functionalities exclusive only to objects. However, the eight primitive data types - byte, short, int, long, float, double, char and boolean are not objects, and therefore cannot use said functionalities. To resolve this issue, Java implemented what's known as Wrapper Classes.

Definition

- Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.
- Wrapper classes are used for converting primitive data types into objects, like int to Integer etc. For example, when working with Collections in Java, we use generics for type safety like this: `ArrayList<Integer>` instead of this `ArrayList<int>`. The Integer is the wrapper class of int primitive type. The table below shows the different wrapper classes designated for each of the primitive data types.

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

Table: Java Wrapper Classes

Purpose

- As previously mentioned, wrapper classes are used when dealing with collections and other data structures that requires objects in order to properly work. The primitive data types are not objects, so they do not belong to any class. It is required to convert the primitive type to object first which we can do by using wrapper classes. Another purpose for using a wrapper class over its primitive data type counterpart is that it allows support for null values.
- However, instances of wrapper classes require more memory compared to primitive types. So, use primitive types when you need efficiency and use wrapper class when you need objects instead of primitive types.

Autoboxing and Unboxing

- Unlike other objects in Java, wrapper classes don't require the 'new' keyword when instantiating an instance. For example, instead of writing `Integer num = new Integer(100);`, as you normally would when instantiating an object, you could write



Integer num = 100;” instead – similar to how you would instantiate a variable of primitive data type. This process is called Autoboxing.

- Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Module 09: JOptionPane Class

From the previous modules, we’ve been using the class Scanner to input data into a program from the keyboard, and we’ve been using the object System.out to output the results to the screen. However, as you may have already experienced, most if not all applications make use of a graphical user interface (GUI) in order to get input and print output. In this module you’ll learn how to create basic GUI components through the JOptionPane class.

Definition

- Java provides a class named JOptionPane that allows you to create GUI components for I/O (input/output). JOptionPane is contained in the javax.swing package, which is not imported to any Java programs by default.
- In order to use the JOptionPane class, you should use the import statement *import javax.swing.JOptionPane;* at the start of your program.

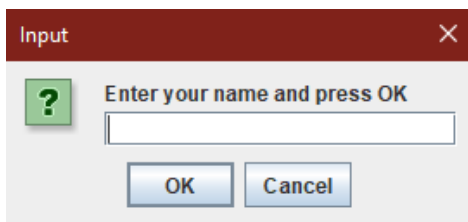
JOptionPane Dialog Boxes

- Since we’re using this class in order to get and print input and output respectively, we’ll use the methods showInputDialog and showMessageDialog. The former will allow the user to input a string from the keyboard, and the latter will allow the program to show a message on the screen.
- The syntax for using the method showInputDialog is:
str = JOptionPane.showInputDialog(stringExpression);

where str is a String variable, and stringExpression is an expression evaluating to a string. When this statement executes, a dialog box containing stringExpression appears on the screen prompting the user to enter the data. Essentially, we use the stringExpression parameter to inform the user what to enter. The data entered is then returned as a string and assigned to the variable str.

- Consider the following statement:
String name = JOptionPane.showInputDialog(“Enter your name and press OK”);

When this statement executes, the dialog box shown below appears on the screen.





The user enters the name in the white area, called the text field. If the user enters a name and clicks the OK button (or if the Enter key is pressed), the dialog box disappears and the entered name is assigned to the variable name.

- That's the input part of the I/O. In order to show messages through a dialog box, we use the method `showMessageDialog` for output. The syntax to use the method `showMessageDialog` is:

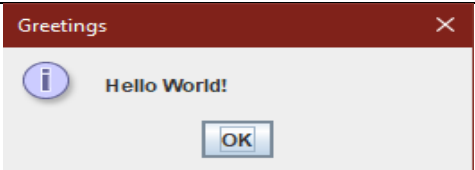
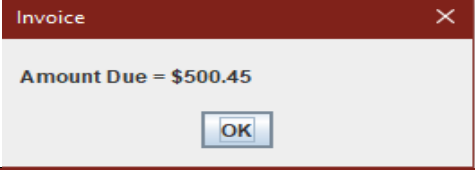
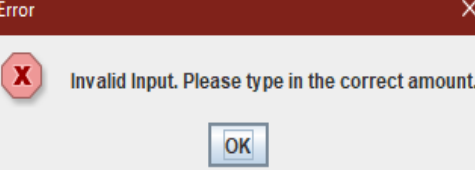
`JOptionPane.showMessageDialog(parentComponent, messageStringExpression, boxTitleString, messageType);`

As you can see, the method has four parameters, which are described in the table below.

Parameter	Description
<code>parentComponent</code>	This is an object that represents the parent of the dialog box that influences where in the screen will the dialog box appear. For now, we will specify the <code>parentComponent</code> to be null that causes the dialog box to appear in the middle of the screen.
<code>messageStringExpression</code>	This is an expression that is evaluated and its value appears in the dialog box. Essentially, what's contained here is the message we want to show the user.
<code>boxTitleString</code>	This represents the title of the dialog box.
<code>messageType</code>	An int value representing the type of icon that will appear in the dialog box. Alternatively, you can use certain <code>JOptionPane</code> options described in the table below.

The table below describes the options that can be used with the parameter `messageType`. Check the following examples on how some of `messageType` option looks like.

<code>messageType</code>	Description
<code>JOptionPane.ERROR_MESSAGE</code>	The error icon is displayed in the dialog box.
<code>JOptionPane.INFORMATION_MESSAGE</code>	The information icon is displayed in the dialog box.
<code>JOptionPane.PLAIN_MESSAGE</code>	No Icon appears in the dialog box.
<code>JOptionPane.QUESTION_MESSAGE</code>	The question icon is displayed in the dialog box.
<code>JOptionPane.WARNING_MESSAGE</code>	The warning icon is displayed in the dialog box.

Statement	Output
<i>JOptionPane.showMessageDialog(null, "Hello World!", "Greetings", JOptionPane.INFORMATION_MESSAGE);</i>	
<i>JOptionPane.showMessageDialog(null, "Amount Due = \$" + 500.45, "Invoice", JOptionPane.PLAIN_MESSAGE);</i>	
<i>JOptionPane.showMessageDialog(null, "Invalid Input. Please type in the correct amount.", "Error", JOptionPane.ERROR_MESSAGE);</i>	

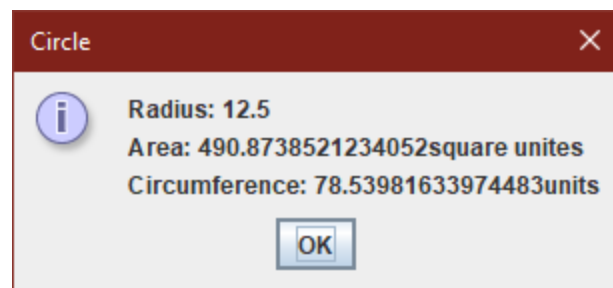
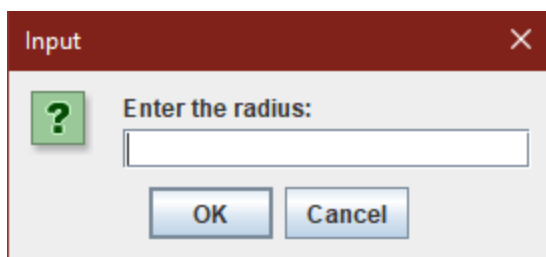
Programming Examples

The following program prompts the user to enter the radius of a circle. The program then outputs the circle's radius, area, and circumference. The class Math defines the named constant PI (π) = 3.141592653589793. We'll use this value to find the circle's area and circumference. To use this value, we'll use the expression Math.PI.

```
import javax.swing.JOptionPane;
public class AreaCircumferenceCalculator {
    public static void main(String[] args) {
        double radius;           //Line 1
        double area;             //Line 2
        double circumference;    //Line 3

        String radiusString;     //Line 4
        String outputString;     //Line 5
    }
}
```

The figures below show a sample run of the program. Then input screen on the left, and on the right, the output screen.





The program works as follows. The statements in Lines 1 through 5 declare the appropriate variables to manipulate the data. The statement in Line 6 displays the input dialog box with the message “Enter the radius:”. Assuming that the entered data is 12.5, the program assigns the value to the String variable *radiusString*. Please note that the method *showInputDialog* always returns a String. In this case, since we’re dealing with decimal numbers, we’ll need to convert the string value to double in order for our program to work. This is handled by line 7.

Lines 8 and 9 calculate the area and circumference of the circle and stores them in the variables area and circumference, respectively. The statement in Line 10 constructs the string containing the radius, area, and circumference of the circle, which will be used as the message in our output dialog box as indicated in Line 11.

The statement in Line 12 terminates the program after the user clicks on the OK button in the dialog box. Please note that if you forgot to add this statement, the program will continue to run (despite nothing happening) when the user clicks on the OK button. So, it’s advisable to add this line whenever possible.

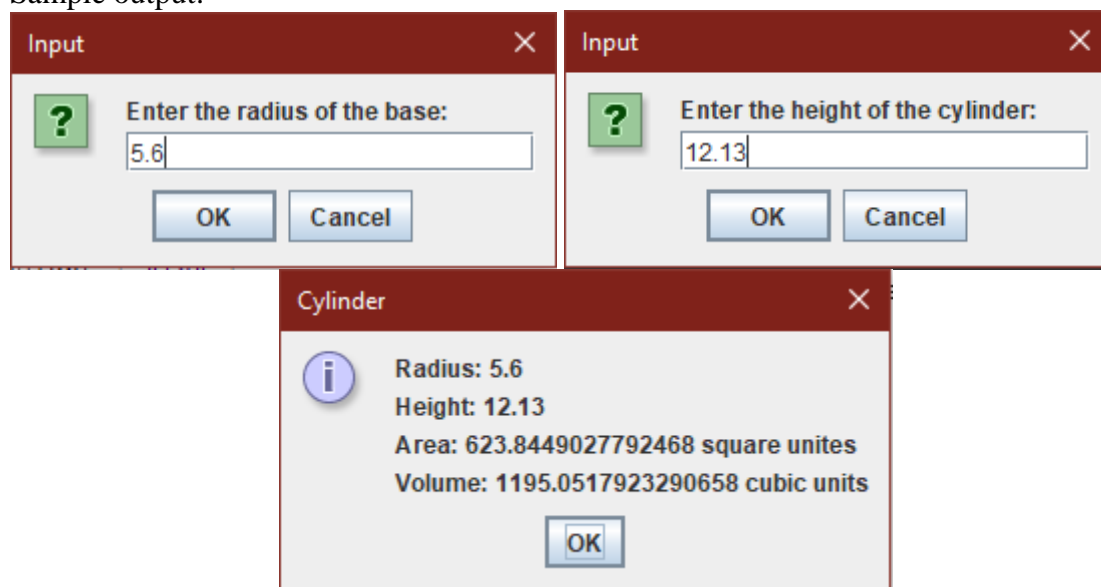
Video Materials

The following video is a tutorial from TheTigerTutorials showing how to implement JOptionPane basics in your program, including a simple error management. As an added exercise, try to modify his source code such that the texts “Thank you” and “Enter a name” are printed in a message dialog box, instead of in the console. [Module 9: JOptionPane - Java Basics.mp4](#)

Programming Exercises

1. Write a program that prompts the user to input the height and the radius of the base of a cylinder, and outputs the volume and surface area of the cylinder. Use the JOptionPane class to display a dialog box for both getting input and displaying the result.

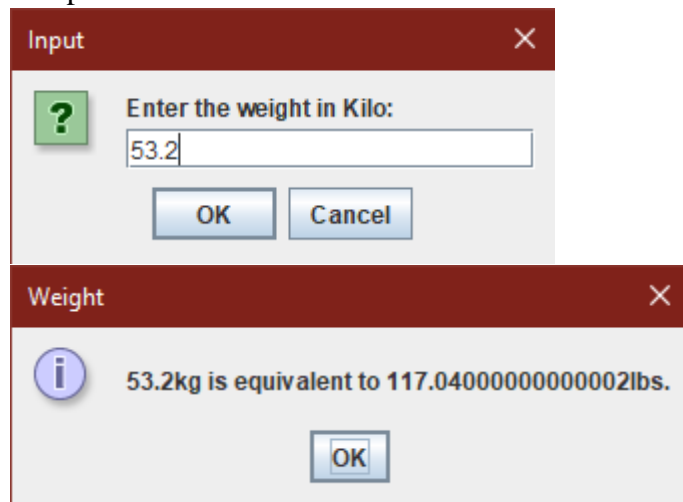
Sample output:



2. Write a program that prompts the user to enter the weight of a person in kilograms and outputs the equivalent weight in pounds. (Note that 1 kilogram = 2.2 pounds.) Use the



JOptionPane class to display a dialog box for both getting input and displaying the result.
Sample Run:



Module 10: Object Oriented Programming

- Object oriented programming (OOP) is based on the concept of '**object**', which corresponds to an actual entity in the problem one tries to program for. Objects are often used to model the real-world objects that you find in everyday life. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
- Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.
- The object in OOP encapsulates all data of a certain type of object and functions applied to that data in programming units called '**classes**'. A class is a blueprint or prototype from which objects are created.
- Such a way of conceptually collecting data and functions related to that data is found to reduce programming errors in large software systems, in addition to being easier to design as things correspond to real entities.
- The general concept of classes and objects is fairly simple – classes are essentially templates and objects are instances made from said templates. But how do we create our templates? Alex Lee and his video tutorial series detail how you can create your own classes in Java.
 - [Module 10: Java Classes - How To Use Classes in Java.mp4](#),
 - [Module 10: Java Constructor Tutorial - Learn Constructors in Java.mp4](#),
 - [Module 10: Getters and Setters - Learn Getters and Setters in Java.mp4](#)



- Aside from the concept of objects, there are other principles you must be aware of if you want to take advantage of Java's, or any other object-oriented programming languages, full capabilities. OOP has four principles: encapsulation, abstraction, inheritance and polymorphism. While we'll give a definition to each of these, this course will only focus on inheritance and polymorphism. Should you be interested in reading more about OOP in general, we've provided some interesting reading materials for you to check out.
- **Encapsulation** is the idea of hiding certain aspects of an object to the user, may it be a characteristic or a behavior. For instance, say we have an object cat that has a hunger characteristic and feed behavior. As an owner of a cat, we cannot directly change how hungry a cat is (i.e. `catHunger = 15`). However, we can feed it which then affects the cat's hunger.
- **Abstraction** can be thought of as a natural extension of encapsulation. Applying abstraction means that each object should only expose a high-level mechanism for using it. For instance, a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.

Inheritance

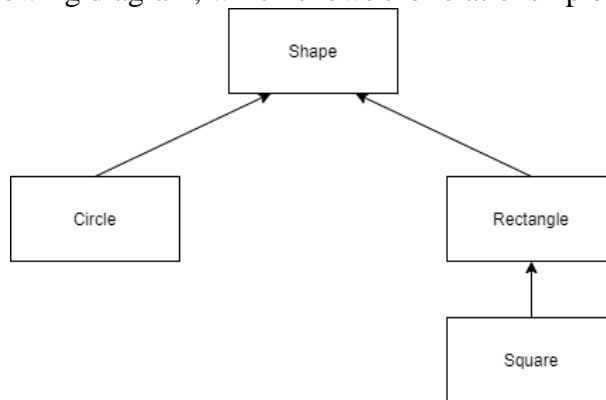
- Suppose you want to design a class, `PartTimeEmployee`, to implement and process the characteristics of a part-time employee. The main features associated with a part-time employee are the name, pay rate, and number of hours worked. Let's assume that we've already implemented another class called `Person` that implements a person's name. We know that every part-time employee is a person. Therefore, rather design the class `PartTimeEmployee` from scratch, we want to be able to extend the definition of the class `Person` by adding additional members – may it be data and/or methods.
- Of course, we do not want to make the necessary changes directly to the class `Person` – that is, edit the class `Person` and add or delete members. What we can do is create a new class called `PartTimeEmployee` without making any physical changes to the class `Person`, by adding only the members necessary to class `PartTimeEmployee`. For instance, since the class `Person` already has data members that can store name, we won't include such members in the class `PartTimeEmployee`. These data members in fact can be inherited from the class `Person`.
- In Java, the mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance. Inheritance can be viewed as an “is-a” relationship. From the example above, every part-time employee is a person.

Superclass and Subclass

- Inheritance can be viewed as a treelike, or hierarchical structure wherein a superclass is shown with its subclass. Any new class that you create from an existing class is called the subclass or derived class or child class; and existing classes are called superclasses, or base classes, or parent classes. The inheritance relationship enables a subclass to inherit features from its superclass – and add new features of its own. Again, from our previous example you can refer to the class `Person` as the superclass; and on the other hand, you can refer to the `PartTimeEmployee` as the subclass.



- Consider the following diagram, which shows the relationship between various shapes.



- In this diagram, Shape is the superclass. The classes Circle and Rectangle are derived from Shape, and the class Square is derived from Rectangle. Every Circle and every Rectangle is a Shape. Every Square is a Rectangle.
- Now, the general syntax for deriving a class from an existing class is:

```
modifier(s) class ClassName extends ExistingClassName modifier(s){  
    memberList  
}
```

Please note that **class** and **extends** is a reserved word in Java.

- Assuming we've already defined a class called Shape, the following statements specify that the class Circle is derived from Shape:

```
public class Circle extends Shape{  
    ...  
    ...  
    ...  
}
```

- Inheritance can either be single or multiple. In single inheritance, the subclass is derived from a single superclass; in multiple inheritance, the subclass is derived from more than one superclass. In Java, supports online single inheritance that is, in Java a class can extend the definition of only one class.

Polymorphism

- Java allows us to treat an object of a subclass as an object of its superclass. In other words, a reference variable of a superclass type can point to an object of its subclass. Say we have a superclass and a few subclasses which inherit from it. Sometimes we want to use a collection, for example a list, which contains a mix of all these classes. Or we have a method implemented for the superclass, but we'd like to use it for the subclasses, too. Polymorphism can be used for this scenario.
- Simply put, polymorphism gives a way to use a class exactly like its superclass so there's no confusion with mixing types. But each subclass class keeps its own methods as they are.



- This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each subclass implements its own version of these methods.
- Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method — regardless of which child is passed.
- Let's assume that classes `Person` and `PartTimeEmployee` are already defined (Check `Person.java` and `PartTimeEmployee.java`). Consider the following statements.

```
Person name, nameRef; //Line 1
PartTimeEmployee employee, employeeRef; //Line 2
name = new Person("Juan", "Dela Cruz"); //Line 3
employee = new PartTimeEmployee("Maria", "Lopez", 12.5, 45); //Line 4
```

Line 1 declares `name` and `nameRef` to be variables of type `Person`. Similarly, Line 2 declares `employee` and `employeeRef` as variables of type `PartTimeEmployee`. The statement in Line 3 instantiates the object `name`, and Line 4 instantiates the object `employee`.

- Now, consider the following statements:

```
nameRef = employee; //Line 5
System.out.println("nameRef: " + nameRef.toString()); //Line 6
```

The statement in Line 5 makes `nameRef` point to the object `employee`. After the statement in Line 5 executes, the object `nameRef` is treated as an object of the class `PartTimeEmployee`. Line 6 outputs the value of the object `nameRef`, and the output of that statement is:

```
nameRef: "Maria Lopez's wages are: 562.5 pesos."
```

Notice that even though `nameRef` is declared as a variable of type `Person`, when the program executes, the statement in Line 6 outputs the first name, the last name, and the wages of a `PartTimeEmployee`. This is because when the statement in Line 6 executes to output `nameRef`, the method `toString` of the class `PartTimeEmployee` executes, not the method `toString` of the class `Person`. This is called late binding, or dynamic binding.

- Let's take a step back. The term polymorphism means associating multiple meanings with the same method name. And binding is determining which method definition gets executed. In early binding, a method's definition is associated with the method's invocation when the code is compiled. In dynamic binding, a method's definition is associated with the method's invocation at execution time. In Java polymorphism is implemented using dynamic binding.

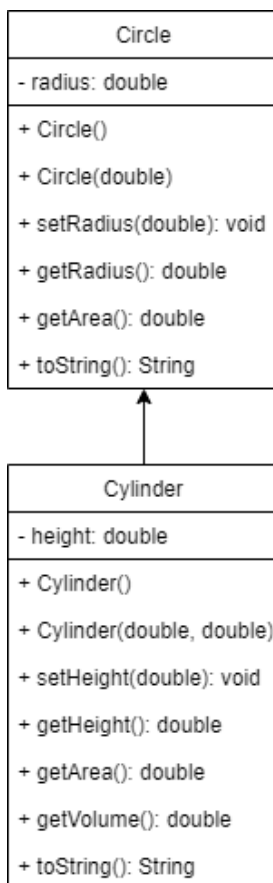
Method Overloading

- Method Overloading is a mechanism that allows a class to have more than one method having the same name, if their argument lists are different. For example the argument list of a method `add(int a, int b)` having two parameters is different from the argument list of the method `add(int a, int b, int c)` having three parameters.

- In order to overload a method, the argument lists of the methods must differ in either of these:
 - Number of parameters. The previous example is a valid case of overloading.
 - Data type of parameters. For example:
add(int x, int y)
add(int x, float y)
 - Sequence of the data type of the parameters. For example:
add(float x, int y)
add(int x, float y)
- Please note that two methods having the same method name but of different return type is not a valid form of method overloading. As a matter of fact, your program won't run as it will throw a compilation error.

Programming Exercises

Consider the following diagram that details the classes Circle and Cylinder. The second row for each box (class) represents the data member (characteristics) of the classes and the third row lists the methods, including constructors, (behavior) of the classes. (Please note that +/- sign represents private and public modifiers respectively.) Create these two classes, with class Cylinder inheriting the class Circle. Also, using your newly created classes, write a program that asks the user to input the radius and height of a cylinder and prints its surface area and volume.





References:

- Suchato, A. (2011). Learning Computer Programming Using Java with 101 Examples. Knowledge Collection and Contribution Initiatives by the Department of Computer Engineering, Chulalongkorn University. ISBN 978-616-551-368-5.
- <https://www.sitepoint.com/beginning-java-data-types-variables-and-arrays/>
- DataFlair Team. 2018. How to Read Java Console Input | 3 Ways To Read Java Input. <https://data-flair.training/blogs/read-java-console-input/>
- JavatPoint. The Best Portal to Learn Technologies. <https://www.javatpoint.com/java-for-loop>
- Schildt, H. 2007. Java: The Complete Reference, Seventh Edition. ISBN: 978-0-07-163177-8
- Java Methods. http://www.tutorialspoint.com/java/java_methods.htm
- GeeksforGeeks. (2020, August 6). *Wrapper Classes in Java*. Retrieved from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/wrapper-classes-java/](https://www.geeksforgeeks.org/wrapper-classes-java/)
- Malik, D. S. (2012). *Java Programming: Problem Analysis to Program Design*. Singapore: Cengage Learning Asia Pte Ltd.
- Oracle Corporation. (2020, June 24). *JOptionPane (Java Platform SE 7)*. Retrieved from [docs.oracle.com: https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html](https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html)
- Singh, C. (2017, September 12). *Method Overloading in Java*. Retrieved from [www.beginnersbook.com: https://beginnersbook.com/2013/05/method-overloading/](https://beginnersbook.com/2013/05/method-overloading/)
- Singh, C. (2017, September 11). *Wrapper Classes in Java*. Retrieved from [www.beginnersbook.com: https://beginnersbook.com/2017/09/wrapper-class-in-java/](https://beginnersbook.com/2017/09/wrapper-class-in-java/)
- w3resource. (n.d.). *Java Wrapper Classes*. Retrieved from [www.w3resource.com: https://www.w3resource.com/java-tutorial/java-wrapper-classes.php](https://www.w3resource.com/java-tutorial/java-wrapper-classes.php)