<u>**DAY 1**</u>

Week 1 Recap

- **History of Java** – Developed in the 1990's by James Gosling. Hundreds of millions if not billions of devices run java globally. Capable of building Desktop apps, Web apps and mobile apps (Koltin).
- **Features of Java** – Object oriented, platform independence, Automatic memory management, Robust and secure, Rich standard library, multi-threading support, High performance, Developer community and Ecosystem.
- **Structure of Java Program –** package declaration, import statements, class definition, method definition and main method definition.
- Java classes, Objects, Methods and main methods.
- **Syntax of Java programs –** variabletype variableName = expressions;
- **Stages of Java Program development** – Coding (editing), compiling, execution.
- **JVM Role –** Class Loading, Verification, Execution, Garbage Collection, Security.
- **Java Naming Convention –** Classes, Variables, Constants, Methods.
- **Java Data types –** Primitive Data Types, Reference Data Types.
- **Casting –** Widening casting and Narrowing casting.

<u>Control Flow Conditional Statements, Looping and Branching Statements</u>

Control Flow – Conditional Statements/Decision making statements

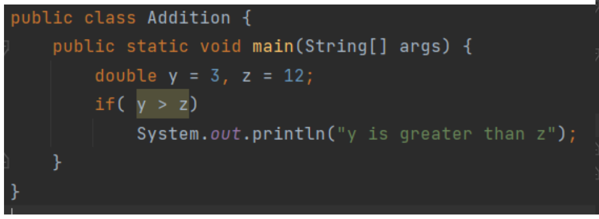
Control flow statements in Java are used to control the order in which the statements in a program are executed. They allow you to make decisions, loop through code, and jump to different parts of your program. Here are examples of control flow statements.

- The 'if' statement
- The 'if - else' statement
- The 'if - else if' statement
- The 'switch' statement

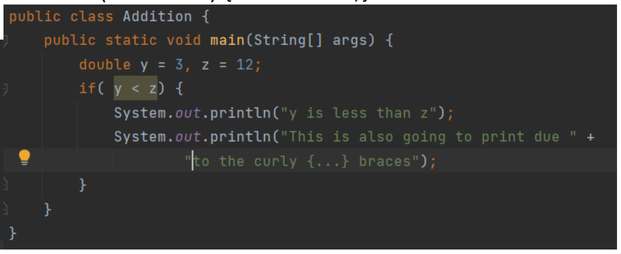
The 'if' statement

The if statement is used to check if a condition is true before executing the next single line or block of code in the curly bracket immediately after the 'if' statement. Examples are:

if(condition) statement; OR if(condition) {statements;}



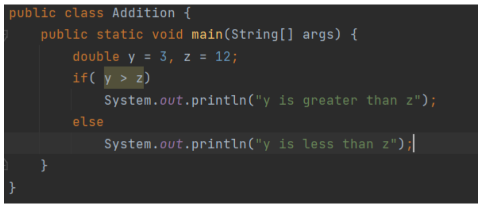
```java
public class Addition {
    public static void main(String[] args) {
        double y = 3, z = 12;
        if( y > z)
            System.out.println("y is greater than z");
    }
}
```



```java
public class Addition {
    public static void main(String[] args) {
        double y = 3, z = 12;
        if( y < z) {
            System.out.println("y is less than z");
            System.out.println("This is also going to print due " +
                    "to the curly {...} braces");
        }
    }
}
```

The 'if - else' statements

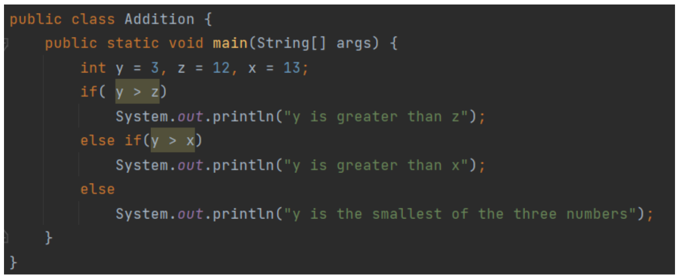
The if-else statement checks a condition, if true, the code after the condition is executed; if not, the code after the else statement is executed.



```java
public class Addition {
    public static void main(String[] args) {
        double y = 3, z = 12;
        if( y > z)
            System.out.println("y is greater than z");
        else
            System.out.println("y is less than z");
    }
}
```

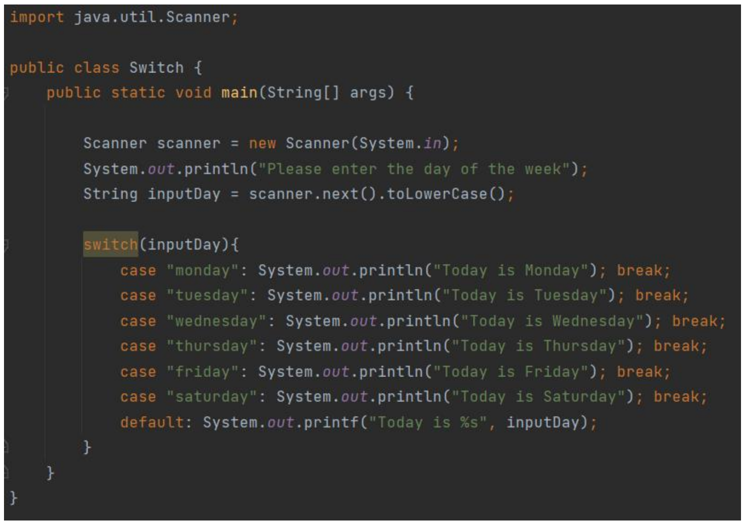
In the above example, the else part will be executed since the condition will evaluate to false.

The 'if - else if' statements
This is used to check for several conditions where the else is immediately followed by another 'if' condition.



```java
public class Addition {
    public static void main(String[] args) {
        int y = 3, z = 12, x = 13;
        if( y > z)
            System.out.println("y is greater than z");
        else if(y > x)
            System.out.println("y is greater than x");
        else
            System.out.println("y is the smallest of the three numbers");
    }
}
```

The 'switch' statement
Selects one of many code blocks to execute based on the value of an expression. Let's take a closer look at the ***switch*** statement structure.



```java
import java.util.Scanner;

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

        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter the day of the week");
        String inputDay = scanner.next().toLowerCase();

        switch(inputDay){
            case "monday": System.out.println("Today is Monday"); break;
            case "tuesday": System.out.println("Today is Tuesday"); break;
            case "wednesday": System.out.println("Today is Wednesday"); break;
            case "thursday": System.out.println("Today is Thursday"); break;
            case "friday": System.out.println("Today is Friday"); break;
            case "saturday": System.out.println("Today is Saturday"); break;
            default: System.out.printf("Today is %s", inputDay);
        }
    }
}
```

Control Flow – Looping Statement
These statements allow you to execute a line or block of codes for a specific number of times until the condition to exit is met. The major looping types in Java include ***the 'for' loop and enhanced for loop, forEach loop, while loop and do-while loop.*** Let's examine each with examples.

The 'for' Loop
The for-loop is a pre-condition loop or 0-trip loop. This implies that the loop may execute the statements zero or more times. The syntax is:
for(initialization; condition; increment) {statement(s)}.
Here below is an example:

```java
public                 class                 ForLoop                 {
        public       static       void       main(String[]       args)       {
            for(int       y       =       0;       y       <       10;       y++){
          System.out.println("The     value     of     y     is:     "     +     y);
      }}}
```

The loop above will start printing from zero (0) and terminate at nine (9); the condition evaluates to 'true' once 'y' gets to 9.

Enhance 'for' Loop

The enhanced for loop is used basically to iterate through a data structure such as array, list, map, queue etc. We'll get to see it work in a future module.

The 'while' Loop
The 'while' loop is a pre-condition loop or 0-trip loop. This is similar to the for-loop. However, its initialization happens outside the loop while the increment happens within the loop. The syntax is:
initialization; while(condition){statements; increment;}
Example:

```
public class WhileLoop {
      public static void main(String[] args) {
       int count = 1; //initialization
       while(count <= 20){ //condition
           System.out.println("The count is: " + count); //statement
           count++; //increment
       }
   }
}
```

do-while Loop
This loop is a post condition loop or a 1-trip loop. This implies that the statement(s) will execute at least one before checking the condition. The syntax is:
do { statement(s); } while(condition);
Example:

```
public class DoWhileLoop {
      public static void main(String[] args) {
       int count = 1; //initialization

       do{
           System.out.println("The count is: " + count); //statement
           count++; //statement
       }
       while (count <= 20); //condition
   }
}
```

Control Flow – Branching Statements
Branching statements allow you to jump out of a loop or to a different section of your program. The common ones are *break and continue.* Let's examine each.

break
As seen in our switch statement. The break keyword tells the loop to stop execution of the statements below the keyword. Here is an example.

```
            public class Break {
             public static void main (String [] args) {
              for (int count = 0; count < 10; count++){
                  System.out.println("The value of y is: " + count);

                  if (count == 5)
                      break;
              }
          }
      }
```

The code above will stop printing once the count is equal to five (5).

continue

The continue keyword can be used to skip execution of statement(s) in a loop once the condition evaluates to true. Here is an example.

```
public class Break {
           public static void main(String[] args) {
            for(int count = 0; count < 10; count++){
               if(count == 5)
               continue;

          System.out.println("The value of y is: " + count);
      }
   }
}
```

In the example above, the print statement will not be executed once the count is equal to five (5). It will continue printing from six (6) till the loop condition evaluates to false.

Exercises:

- Write a for-loop to count downwards from 15 to 1.
- Write a while loop to print even numbers from 2 to 30.
- Write a do-while loop to print “Hello World” 5 times.
- Using the continue keyword, print numbers 1 to 25 without multiples of 4 in the series.

DAY 2

Methods and Classes

Methods

A method in Java is a block of code that performs a specific task. It is a reusable piece of code that can be called from anywhere in your program. Methods are defined in classes, and they are used to encapsulate code and make it easier to maintain and understand your program.
There are two main types of methods in Java:

- Static methods do not belong to any object, and they can be called without creating an object of the class.
- Instance methods belong to an object, and they must be called from an object of the class.

Here is an example of a static method in Java:

```
public          static          void          printMessage()          {
        System.out.println("This      is      a      static      method");
    }
```

Above method can be called from anywhere within the program. It doesn't need an object created for it to be called.

Here is an example of an instance method in Java:

```
    public               void               printMessage()               {
        System.out.println("This      is      an      instance      method");
    }
```

The method above can only be called using an object of the class that contains it or extends it. We'll see more practical examples later.

Methods can have parameters, which are variables that are passed to the method when it is called. The parameters are used to provide information to the method, and they can be used to modify the behavior of the method.

Methods can also return a value, which is a value that is returned by the method when it is finished executing. The return value can be used by the code that called the method.

Methods are an essential part of Java programming. They allow you to encapsulate code, make your code more reusable, and make your code easier to understand.

Here are some of the benefits of using methods in Java:

- **Reusability**: Methods can be reused in different parts of your program, which makes your code more concise and easier to maintain.
- **Modularity**: Methods can be grouped together into logical units, which makes your code easier to understand and debug.
- **Abstraction**: Methods can hide the implementation details of a task, which makes your code more robust and easier to change.

Declaring and Calling methods

Declaring Methods: Methods are declared using a modifier, return type, method name and parameters (if any). Any method with a return type declared must end with the statement '*return return-type data'.*

Example: Method declaration

```
public class Addition {
    public static void main(String[] args) {
        int y = 3, z = 12, x = 6;
        getMaximum(x,y,z); //Method call
    }

    public static int getMaximum(int x, int y, int z){ //Method declaration
        int max = x; //assuming x is the highest
        if( y > max){
            max = y;
        }
        if( z > max){
            max = z;
        }
        System.out.println("max is : " + max);
        return max;
    }
}
```

Methods that do not have to return any value are declared with the keyword 'void' as seen in previous examples above.

Classes and Objects

Classes and objects are the two most fundamental concepts in object-oriented programming (OOP). A class is a blueprint for creating objects, while an object is an instance of a class.

A class defines the properties and behaviors of an object. The properties of an object are called its attributes, and the behaviors of an object are called its methods.

Here is an example of a class with attributes (class fields) and behavior (methods) in Java:

```
public class Dog {
    private String name;         //achieve encapsulation by declaring variables as private
    private int age;

    public Dog(String name, int age) { //Class constructor
        this.name = name;
        this.age = age;
    }

    public String getName() {    //getter method to access the private variable 'name'. It's return type
        return name;             //is String
    }

        public void bark() {     //void methods do not return any value and do not have a return type
        System.out.println("Woof!");

    }

}
```

To access the instance methods in the Dog class, we need to create and object first. See below.

Creating a dog object and calling the bark() method of the Dog class.

```
Dog dog = new Dog("Alsatian", 2); //A dog object was created with name and age

dog.bark(); //We use the dog object created to call the method 'bark()'
```

Encapsulation, Inheritance and Polymorphism

- **Encapsulation**: Encapsulation refers to the bundling of data and methods within a class, and the restriction of access to the internal state of an object from outside the class. It helps in achieving data hiding and abstraction. In the Dog class above, the methods are public which is used to access the encapsulated field in the Dog class. This provides some sort of security for your class fields and makes it unreachable for unwanted parties.
- **Inheritance**: Inheritance enables you to create a new class (derived or child class) from an existing class (base or parent class). The derived class inherits the properties and behaviors of the parent class, allowing code reuse and the creation of class hierarchies. Java supports single inheritance (one class extends another class) and multiple inheritance through interfaces (a class implements multiple interfaces). Let's look at a class Animal that will be extended by another class Cat.

```
public class Animal {
    private String name;
    private int age;

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

    public void move(){
        System.out.println("Animal is moving");
    }
    public void run(){
        System.out.println("Animal is running");
    }
    public void speak(){
        System.out.println("Animal makes a unique sound");
    }
}

public class Cat extends Animal{
    private String name;
    private int age;
    private int numberOfLegs;

    public Cat(String name, int age, int numberOfLegs){
        super(name, age);       //Calling super comes before any other calls in the constructor
        this.numberOfLegs = numberOfLegs;
    }

    @Override
    public void move() {        //Override the move method of Animal class
        System.out.println("Cat is walking");    }
   @Override
```

```
    public void run(){          //Override the run method of Animal class
         System.out.println("Cat is running");
     }

    @Override
     public void speak(){         //Override the speak method of Animal class
         System.out.println("Cat says meow");
     }
}
```

- **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables you to perform a single action in different ways, depending on the objects involved. Polymorphism in Java is typically achieved through method overriding and method overloading.

DAY 3

Arrays

An array in Java is a data structure that can store a collection of data of the same type. Arrays are objects in Java, and they are created using the new keyword. The syntax for declaring an array in Java is as follows:

Array Declaration: You declare an array by specifying the type of elements it will hold, followed by square brackets [] and the array name. For example, to declare an array of integers:

int[] myArray; // Declaration of an integer array

e.g. int[] numbers = new int[10]; //*this will create an array of type integer with a capacity to store 10 int values.*

String[] names = new String[5]; //*this will create an array of type String with a capacity to store 5 string values.*

Array Initialization: After declaring an array, you can initialize it with values. There are several ways to initialize an array:

- **Static Initialization:** Provide the values at the time of declaration.

int[] numbers = {1, 2, 3, 4, 5}; // Initialization with values

- **Dynamic Initialization:** Initialize the array later using the new keyword.

int[] numbers = new int[5]; // Initialization with a specific size

numbers[0] = 1; // Assigning a value to the first element

numbers[1] = 2; // Assigning a value to the second element

Array Access: You can access individual elements of an array using their index. Array indices start from 0 and go up to array.length - 1. For example:

int[] numbers = {1, 2, 3, 4, 5};

int firstElement = numbers[0]; // Accessing the first element

int thirdElement = numbers[2]; // Accessing the third element

Array Length: The length of an array is determined by the number of elements it can hold. You can get the length of an array using the length property:

int[] numbers = {1, 2, 3, 4, 5};

int length = numbers.length; // Length of the array (returns 5)

Iterating over an Array: You can use loops, such as for or foreach, to iterate over the elements of an array:

```
int[] numbers = {1, 2, 3, 4, 5};
        for (int i = 0; i < numbers.length; i++) {
        System.out.println(numbers[i]); // Accessing and printing each element
        }


        int[] numbers = {1, 2, 3, 4, 5};
        for (int number : numbers) {
          System.out.println(number); // Accessing and printing each element using foreach loop
        }
```

You can also declare multidimensional arrays in Java. A multidimensional array is an array of arrays. For example, the following code declares a 2D array of integers called matrix:

```
int[][] matrix = new int[3][2];
```

The matrix array has 3 rows and 2 columns. You can access the elements of a multidimensional array using their index. For example, the following code prints the value of the element at the intersection of row 0 and column 1 of the matrix array:

e.g. `System.out.println(matrix[0][1]);`

Arrays are a powerful and versatile data structure in Java. They can be used to store a wide variety of data, and they can be accessed and manipulated in a variety of ways.

Here are some of the important points about arrays in Java:

- Arrays are objects in Java.
- Arrays can be of any data type, including primitive data types and object types.
- Arrays can be declared with one or more dimensions.
- The elements of an array are stored in contiguous memory locations.
- The length of an array is fixed once it is created.
- Arrays can be accessed using their index.
- Arrays can be manipulated using a variety of methods.

Strings

In Java, the String class is used to represent a sequence of characters. It is widely used for manipulating and working with textual data. The String class is part of the Java standard library and provides various methods for performing operations on strings. Here are some commonly used string operations in Java:

1. String Declaration and Initialization: You can declare and initialize a string in Java using the following syntax:

```
String str1 = "Hello, World!"; // Declaration and initialization
          String str2 = new String("Hello"); // Using the new keyword to create a string object
```

Note that strings in Java are immutable, which means their values cannot be changed once they are created. However, you can assign a new string to the same variable.

2. Concatenation: The + operator can be used for string concatenation in Java. It combines two strings into a single string:

```
String str1 = "Hello";
          String str2 = "World";
          String result = str1 + ", " + str2; // Concatenation
          System.out.println(result); // Output: Hello, World
```

Alternatively, you can use the concat() method to concatenate strings:

```
String str1 = "Hello";
          String str2 = "World";
          String result = str1.concat(", ").concat(str2); // Concatenation using concat() method
          System.out.println(result); // Output: Hello, World
```

3. String Length: You can find the length (number of characters) of a string using the length() method:

```
String str = "Hello, World";
          int length = str.length(); // Length of the string (returns 12)
```

4. Substring: The substring() method extracts a portion of a string based on the specified indices:

```
String str = "Hello, World";
          String substr1 = str.substring(7); // Extracts from index 7 to the end (returns "World")
          String substr2 = str.substring(0, 5); // Extracts from index 0 to 4 (returns "Hello")
```

5. String Comparison: You can compare strings using the equals() method, which compares the contents of two strings:

```
String str1 = "Hello";
          String str2 = "Hello";
          boolean isEqual = str1.equals(str2); // String comparison (returns true)
```

To perform a case-insensitive comparison, you can use the equalsIgnoreCase() method.

6. String Manipulation: The String class provides several methods for manipulating strings, such as:

- toUpperCase(): Converts the string to uppercase.
- toLowerCase(): Converts the string to lowercase.
- trim(): Removes leading and trailing whitespace.
- replace(): Replaces occurrences of a character or substring.
- split(): Splits the string into an array of substrings based on a delimiter.

Example

```
String str = "   Hello, World!   ";
        String trimmedStr = str.trim(); // Trims leading and trailing whitespace
        System.out.println(trimmedStr); // Output: "Hello, World!"

        String replacedStr = str.replace("Hello", "Hi"); // Replaces "Hello" with "Hi"
        System.out.println(replacedStr); // Output: "   Hi, World!   "

        String[] parts = str.split(","); // Splits the string at the comma
        System.out.println(parts[0]); //

        String replacedStr = str.replace("World", "Universe");
        System.out.println(replacedStr); // Hello Universe
```

Coding Exercises

- Write an array to store the first ten even numbers. Print out all the numbers using for loop.
- Write an array to store integers from 1 to 15; sum up all the integers using forEach loop.
- Write an array to store 10 strings and print out the 1st, 5th and last strings.
- Write a program that initializes two strings; concatenate the strings with a comma; split the string using the split() method.
- Create two un-equal strings; compare both strings and print out the boolean result.
- Write a String “Hello, World!”. Write a code to replace the ‘World’ with ‘Java Globe’.

DAY 4

Java Generics

Java Generics is a feature in the Java programming language that allows you to define classes, interfaces, and methods that can operate on objects of various types, while providing type safety at compile time. Generics enables you to create reusable code that is not limited to a specific type, but can be used with multiple types, providing flexibility and type checking at compile time.

The main purpose of generics is to provide compile-time type checking and to avoid the need for explicit type casting. It helps in writing code that is more generic and reusable, as it allows you to define classes, interfaces, and methods that can work with different types of data.

To understand generics, let's start with a basic example. Consider a scenario where you want to create a collection to store a list of strings. In pre-generics Java, you would typically use the Object class to store objects of any type. However, this approach could lead to runtime type errors and require explicit type casting. With generics, you can specify the type of objects the collection will hold, like this:

List<String> strings = new ArrayList<String>();

In this example, List<String> is a generic type declaration. Here, List is a generic interface or class, and <String> is the type parameter that specifies that the list will contain objects of type String. This way, the compiler ensures that only strings can be added to this list, and you don't need to explicitly cast the elements.
Generics can be used with classes, interfaces, and methods. Here's a brief overview of how they can be used in different contexts:

Generic Classes: You can define a generic class by using angle brackets (<>) and a type parameter. For example:

```
public class Box<T> {
        private T contents;

        public void setContents(T contents) {
          this.contents = contents;
        }

        public T getContents() {
          return contents;
        }
      }
```

In this example, Box is a generic class that can hold objects of any type specified by the type parameter T. The setContents and getContents methods can work with any type, as determined by the instantiation of the class.

Generic Interfaces: Similar to generic classes, you can define generic interfaces using type parameters. For example:

```
public interface List<T> {
        void add(T element);
        T get(int index);
      }
```

Here, the List interface is generic, and the methods can work with any type specified by the type parameter T. Implementing classes will provide the actual type when implementing the methods.

Generic Methods: You can also create generic methods within non-generic classes or interfaces. These methods have their own type parameters, independent of the enclosing class or interface. For example:

```
public class Utils {
    public static <T> T doSomething(T obj) {
        // ...
    }
}
```

In this case, the doSomething method is a generic method. The type parameter <T> is declared before the return type T, and it specifies that the method can work with any type.

Generics also support bounded type parameters, wildcards, and other advanced features to provide additional flexibility and constraints. The use of generics allows for increased type safety, improved code reuse, and better documentation by specifying the expected types explicitly.

It's worth mentioning that generics are a compile-time feature and are subject to type erasure, which means that the type information is removed during compilation. This enables backward compatibility with pre-generics Java code but also imposes some limitations on working with generic types at runtime.