OBJECT ORIENTED PROGRAMMING WITH JAVA

BCA II SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

UNIT – I

Introduction to Java:

OOPs concepts, Basics of Java programming, Data types, Variables, Operators, Control structures including selection, Looping, Arrays in java.

Objects and Classes:

Basics of objects and classes in java, Methods and objects, Instance of operator, Visibility modifiers, Method Overloading, Constructors, Static Members, Inbuilt classes like String, Character, String Buffer, this reference.

Java: Java is a high-level, object-oriented programming language that was developed by Sun Microsystems in the mid-1990s. It is a popular language that is used to develop desktop and web applications, mobile apps, games, and other types of software.

Java Features:

Some key features of Java are:

Platform independence - Java code can run on any platform that has a JVM, which means that Java programs can be written once and run anywhere.

Object-oriented - Java is an object-oriented language, which means that it is based on the concept of objects that have attributes and behaviors.

Garbage collection - Java has automatic memory management, which means that the programmer does not have to manually allocate and deallocate memory.

Robust - Java is designed to be robust, which means that it has built-in error handling and exception handling mechanisms.

Multi-threading - Java has built-in support for multi-threading, which means that multiple threads can run concurrently within a single program.

Security - Java has built-in security features that help protect against viruses, malware, and other security threats.

Java Applications:

Java is used in a wide range of applications, including:

Enterprise applications - Java is commonly used to develop enterprise applications, such as banking systems, customer relationship management (CRM) systems, and inventory management systems.

Web applications - Java is used to develop web applications, such as e-commerce sites, social networking sites, and content management systems.

Mobile applications - Java is used to develop mobile applications for Android devices.

Console applications:

Games - Java is used to develop games for desktop and mobile devices.

JDK:

JDK stands for Java Development Kit. It is a software development kit that contains tools necessary for developing, testing, and deploying & running Java applications. It includes a Java compiler, which converts Java source code into bytecode, as well as other tools such as the Java Virtual Machine (JVM), the Java Runtime Environment (JRE), and the JavaFX SDK.

JDK = JRE + Development Kit

JRE:

JRE stands for Java Runtime Environment. It is a software environment that provides the necessary runtime support for running Java applications. It includes the Java Virtual Machine (JVM), which is responsible for executing Java bytecode, as well as core libraries and other components that are required to run Java applications.

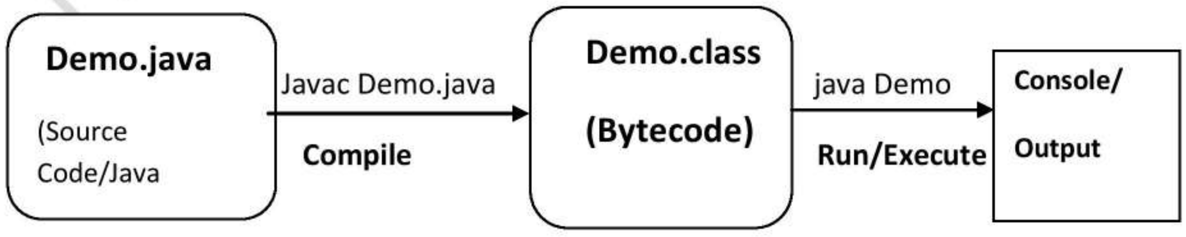
JRE = JVM + Lib Classes

JVM:

JVM stands for Java Virtual Machine. It is a software component that provides an environment for executing Java bytecode. The JVM is responsible for interpreting the bytecode and converting it into machine code that can be executed by the underlying operating system. It also provides features such as automatic memory management and security. The JVM is a key component of the Java platform and is used for running Java applications on different platforms.

Bytecode: Machine-readable code that is generated by the java compiler when a java source code is compiled.

When a Java program is compiled, the Java compiler converts the source code into bytecode, which is saved in a file with the .class extension. The bytecode is then interpreted and executed by the JVM.



OOPs concepts:

Java is an object-oriented programming language, and it incorporates four fundamental OOPs concepts, which are:

Encapsulation: Encapsulation is a mechanism to bundle the data and the methods that operate on that data within a single unit, called a class. The data is protected from direct access by external entities, and can only be accessed through methods provided by the class. This ensures that the internal representation of an object is hidden from the outside world and only the class has control over its internal state.

Inheritance: Inheritance is a mechanism by which one class inherits the properties and methods of another class. The class that inherits the properties and methods is called a subclass, while the class that provides the properties and methods is called a superclass. Inheritance allows the code reuse and helps in achieving polymorphism.

Polymorphism: Polymorphism refers to the ability of an object to take on multiple forms. Java supports two types of polymorphism: compile-time polymorphism and runtime polymorphism. Compile-time polymorphism is achieved through method overloading, while runtime polymorphism is achieved through method overriding.

Abstraction: Abstraction is the process of hiding implementation details while showing only the necessary information to the user. In Java, abstraction is achieved through abstract classes and interfaces. Abstract classes are classes that cannot be instantiated, and can only be inherited by subclasses. Interfaces define a set of methods that a class must implement.

Class: is a blueprint or template that defines the properties and behaviors of objects of a particular type. A class is essentially a collection of related data and methods that describe the characteristics and actions of objects of that class. The data defined in a class is referred to as its attributes, while the methods defined in a class are referred to as its behavior.

Object: is an instance of a class. It is a tangible entity that can be created and manipulated by the code. When an object is created, it is created from a class and has all the attributes and behaviors defined in the class. Objects can interact with each other through their methods, and can also modify their own state through their attributes.

Basics of Java programming:

Data types: Java has several built-in data types, which are divided into two categories: primitive types and reference types.

1. Primitive/Basic/Built-in Data Types: Primitive types are basic data types that are built into the Java language. They are used to represent simple values and are stored directly in memory. There are eight primitive types in Java: Primitive types are basic data types that are built into the Java language. They are used to represent simple values and are stored directly in memory. There are eight primitive types in Java:

Integer Types:

- **byte:** 8-bit signed integer (-128 to 127).
- **short**: 16-bit signed integer (-32,768 to 32,767).
- **int:** 32-bit signed integer (-2,147,483,648 to 2,147,483,647).
- **long:** 64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807).

Floating-Point Types:

- **float**: 32-bit floating-point number (IEEE 754 standard).
- **double:** 64-bit floating-point number (IEEE 754 standard).

Character Types:

- **char:** 16-bit Unicode character.

Boolean Type:

- **boolean:** true or false value.

2. Reference Types:
Reference types are more complex data types that are used to refer to objects in memory. They are also known as object types or class types. Reference types are stored as references to objects, rather than the objects themselves. There are four reference types in Java:

Class types: represents classes.
Interface types: represents interfaces.
Array types: represents arrays.
Object types: represents objects.

Variables:

A variable is a named storage location in memory that holds a value of a particular data type. Variables are used to store values that can be manipulated and used throughout a program.
Here are the key features of variables in Java:

Declaration: Variables must be declared before they can be used in a program. A variable declaration specifies the data type of the variable and assigns a name to it.

Initialization: Variables can be initialized with a value when they are declared or at a later time in the program.

Scope: The scope of a variable is the part of the program where the variable can be accessed. Variables can have block scope, method scope, class scope, or global scope.

Type: Variables have a data type, which determines the kind of data that can be stored in the variable. Java has several built-in data types.

Naming conventions: Variable names in Java must follow certain naming conventions. They must start with a letter, can include letters, numbers, and underscores, and should be descriptive of the value they hold.

Types of variables:

In Java, variables can be classified into several categories based on their scope and lifetime. Here are the main types of variables in Java:

Local variables:
Local variables are declared inside a method or a block and have a local scope. They are accessible only within the block or method where they are declared. Local variables are not initialized automatically and must be assigned a value before they can be used.

Instance variables:
Instance variables are declared inside a class but outside of any method, constructor or block. They are also known as non-static fields because they are associated with instances of the class. Instance variables have a default value and can be accessed by any method or constructor within the class.

Class/static variables:
Class or static variables are declared inside a class and outside of any method or constructor, but they are preceded by the keyword "static". Class variables have a single value for all instances of the class and can be accessed using the class name. They are initialized when the class is loaded into memory and are not dependent on any instance of the class.

```
class Student
{
                int rollno;                 //instance variables
                String name;
                static int collCode = 1234;      //Class variables

                Student(int rollno, String name)
                {
                        this.rollno = rollno;
                        this.name = name;
                }

                public void printPerc(int marks)
                {
                        double perc = marks/6;        //perc, marks are local variables
                        System.out.println("Percentage : "+perc);
                }

                public static void main(String[] args)
                {
                        Student s1 = new Student(1, "Ravi");   //s1, s2 are reference variables
                        Student s2 = new Student(2, "Raju");

                        s1.printPerc(560);
                        s2.printPerc(540);
                }
}
```

Naming conventions in Java:

Variables: Use camelCase to name variables. The first letter of the first word is lowercase and the first letter of each subsequent word is uppercase. For example, firstName, numberOfStudents, isAvailable.

Methods: Use camelCase to name methods as well. The first letter of the first word is lowercase and the first letter of each subsequent word is uppercase. Use verbs to name methods that perform some action. For example, calculateSum(), printName(), getAddress().

Constants: Use uppercase letters to name constants. If the name contains multiple words, separate them with an underscore (_). For example, MAX_VALUE, MIN_VALUE, PI.

Packages: Use lowercase letters to name packages. Use a domain name in reverse order as a prefix to avoid naming conflicts. For example, com.example.projectname, org.apache.commons.lang3.

Interfaces: Use a capitalized letter at the beginning of each word to name interfaces. Use nouns to name interfaces. For example, Serializable, Comparable, List.

Classes: Use a capitalized letter at the beginning of each word to name classes. Use nouns to name classes. For example, Student, Address, Calculator.

Operators: **Operators** are symbols or keywords used to perform specific operations on operands. Operands can be variables, literals, method calls, or expressions.
Here are the main types of operators in Java:

1. Arithmetic Operators:
Arithmetic operators are used to perform basic mathematical operations on operands. The arithmetic operators in Java are:

+ (addition) - (subtraction) * (multiplication) / (division)
% (modulo, remainder of division)

2.Assignment Operators:
Assignment operators are used to assign a value to a variable. The assignment operator in Java is =. Other assignment operators combine arithmetic operations with assignment, such as += (add and assign), -= (subtract and assign), *= (multiply and assign), /= (divide and assign), and %= (modulo and assign).

3. Comparison/Relational Operators:
Comparison operators are used to compare two operands and return a boolean value (true or false). The comparison operators in Java are:

== (equal to)
!= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

4. Logical Operators:
Logical operators are used to combine two or more boolean expressions and return a boolean value. The logical operators in Java are:

&& (logical AND)
|| (logical OR)
! (logical NOT)

5. Bitwise Operators:
Bitwise operators are used to perform operations on individual bits of an operand. The bitwise operators in Java are:

& (bitwise AND)
| (bitwise OR)
^ (bitwise XOR)
~ (bitwise complement, one's complement)

6. Conditional Operator:
The conditional operator (also known as the ternary operator) is a shorthand way of writing an if-else statement. The conditional operator in Java is condition ? expression1 :

expression2. If the condition is true, expression1 is evaluated and returned; otherwise, expression2 is evaluated and returned.

Example program:

Control structures including selection: if we want to control the flow of execution of the program. These control structures allow the program to make decisions and execute specific code blocks based on the conditions or values in the program.

1. Decision making/Conditional/if-else/branching statements: statements are used to execute a block of code based on a condition. The most common selection statement in Java is the if-else statement.

1. simple if : The if statement checks a single condition and executes a block of code if the condition is true. The syntax for the if statement is as follows: The if statement checks a single condition and executes a block of code if the condition is true. The syntax for the if statement is as follows:

```
if (condition)
{
        // code to be executed if condition is true
}
```

Ex:

```
int num = 10;
if (num > 0)
{
        System.out.println("Number is positive");
}
```

2. if-else: The if-else statement checks a condition and executes if body of code if the condition is true, otherwise, it executes else block code.
The syntax for the if-else statement is as follows:

```
if (condition)
{
  // code to be executed if condition is true
}
else
{
  // code to be executed if condition is false
}
```

Ex:

```
int num = -5;
if (num > 0)
{
  System.out.println("Number is positive");
}
else
{
  System.out.println("Number is negative");
}
```

3. nested if else: Defining if-else statements into another if-else body. Inner if else will be executed if outer if condition is true, otherwise outer else block will execute.

Syntax:

```
if (condition1)
{
        if(condition2)
        {
                //Code to be executed cond1 & cond2 are true
        }
        else
        {
                //Code to be executed cond1 is true & cond2 is false
        }
}
else
{
        if(condition3)
        {
                //Code to be executed cond1 & cond3 are true
        }
        else
        {
                //Code to be executed cond1 is true & cond3 is false
        }
}
```

Ex:

```
        int a,b,c;
        a = 10;
        b = 20;
        c = 30;
        if (a>b)
        {
                if(a>c)
                {
                        System.out.println(a+"is Greater");
                }
                else
                {
                        System.out.println(c+"is Greater");
                }
        }
        else
```

```
{
        if(b>c)
        {
                System.out.println(b+"is Greater");
        }
        else
        {
                System.out.println(c+"is Greater");
        }
}
```

4. else if ladder: The else-if ladder is a conditional statement in Java that allows for the execution of different blocks of code based on multiple conditions. It is similar to the if-else statement, but it can check multiple conditions using a series of if-else blocks.

Syntax:

```
if (condition1)
{
  // code to be executed if condition1 is true
}
else if (condition2)
{
  // code to be executed if condition2 is true
}
else if (condition3)
{
  // code to be executed if condition3 is true
}
.
.
else
{
  // code to be executed if none of the conditions are true
}
```

Ex:

```
int num = 5;

if (num > 0)
{
  System.out.println("Number is positive");
}
else if (num < 0)
{
  System.out.println("Number is negative");
}
else
{
  System.out.println("Number is zero");
}
```

2. Selection/switch statements:
A switch statement is used to execute different statements based on the value of a variable or expression. The switch statement evaluates the value of an expression and compares it to various case labels. When it finds a match, the code associated with that case is executed.

The syntax of the switch statement in Java is as follows:

```
switch (expression/variable)
{
  case value1:
    // code to be executed if expression matches value1
    break;
  case value2:
    // code to be executed if expression matches value2
    break;
  ...
  default:
    // code to be executed if expression doesn't match any of the cases
}
```

Ex:

```
int x = 3;
switch (x)
{
  case 1:
    System.out.println("One");
    break;
  case 2:
    System.out.println("Two");
    break;
  case 3:
    System.out.println("Three");
    break;
  default:
    System.out.println("Invalid");
    break;
}
```

3. Looping: Looping statements are used to execute a block of code repeatedly. In Java, the looping statements are: every loop has 3 parts, initialization, TestExpression, Inc/Dec.

1. **for loop:** repeats a block of code a fixed number of times.

Syntax:

```
for (initialization; condition; increment/decrement)
{
 // code to be executed repeatedly
}
```

Ex:

```
for (int i = 0; i < 5; i++)
{
  System.out.println("The value of i is " + i);
}
```

Output:

2. **while loop:** repeats a block of code if a condition is true.

The while loop is used when you don't know the number of times you want to execute a block of code, but you know the condition that will make the loop stop.

Syntax:

```
Initialization
while (condition)
{
 // code to be executed repeatedly
 Inc/Dec
}
```

Ex:

```
int i = 0;
while (i < 5)
{
  System.out.println("The value of i is " + i);
  i++;
}
```

3. **do-while loop:** repeats a block of code at least once and then as long as a condition is true. The do-while loop is similar to the while loop, but it will execute the block of code at least once, even if the condition is initially false.

Syntax:

```
Initialization
do
{
 // code to be executed repeatedly
 Inc/Dec
} while (condition);
```

Ex:

```
int i = 0;

do
{
  System.out.println("The value of i is " + i);
  i++;
} while (i < 5);
```

4. Jump/Branching statements: Jump statements are used to transfer the control of a program to a different part of the program. In Java, the jump statements are:

break statement: The break statement is used to terminate a loop or a switch statement. When the break statement is encountered inside a loop, the loop immediately terminates and the control is transferred to the next statement after the loop.

Syntax & Example:

```
for (int i = 0; i < 10; i++)
{
 if (i == 5)
 {
  break;
 }
 System.out.println(i);
}
```

Output:

continue statement: The continue statement is used to skip the current iteration of a loop and proceed to the next iteration. When the continue statement is encountered, the control is transferred to the condition of the loop.

Exa:

```
for (int i = 0; i < 10; i++)
{
 if (i == 5)
 {
  continue;
 }
 System.out.println(i);
}
```

return statement: The return statement is used to exit a method and return a value to the caller. When the return statement is encountered, the method immediately terminates and control is returned to the caller.

Exa:

```
public int sum(int a, int b)
{
 int result = a + b;
 return result;
}
```

Objects and Classes:

A class is a template or blueprint for creating objects. It defines the properties and behaviors that an object of that class should have. An object is an instance of a class, and it has its own unique values for the properties defined in the class.
Class contains related members like data members(Instance variables), Constructors, Methods(Behaviors).

Class can contain static as well as non-static members.

Syntax:

```
[access modifier] class ClassName
{
  // Fields - Data members
  //Constructors
  //Methods
}
```

[access modifier]: This is an optional keyword that specifies the visibility of the class, fields, and methods. There are four access modifiers in Java: public, protected, private, and no modifier (also known as package-private).

class: This is the keyword used to define a class.

ClassName: This is the name of the class, which should be written in PascalCase (the first letter of each word is capitalized).

Fields: These are the variables that store data for the class.

Constructors: These are special methods that are used to create new instances of the class. The constructor has the same name as the class and can take parameters.

Methods: These are the functions that define the behavior of the class. Methods can also take parameters and can have a return type.

Here's an example of a class that defines a Person object:

```
class Person
{
  // Fields
  String name;
  int age;

  // Constructor
  Person(String name1, int age1)
  {
```

```
            name = name1;
            age = age1;
          }

          // Method
          void sayHello()
           {
            System.out.println("Hello, my name is " + name + " and I am " + age + " years
        old.");
          }
        }
```

Objects: object is an instance of a class that contains data (fields) and code (methods) that operate on that data. In java, you create an object using the new keyword followed by a call to the class constructor.

The syntax for creating an object in Java is as follows:

ClassName objectName = new ClassName();

ClassName: This is the name of the class that you want to create an object from.

objectName: This is the name that you give to the object. It is used to reference the object later in your code.

new: This keyword is used to create a new instance of the class.

ClassName(): This is the constructor for the class. It is a special method that is used to initialize the object. The parentheses are empty if the class has a default constructor (a constructor with no parameters).

Ex:

```
// Define a class called Person
class Person
{
  String name;
  int age;
}

// Create an object of the Person class
Person john = new Person();
```

- In this example, we've defined a class called Person with two fields (name and age). We then create an object of the Person class called john using the new keyword and the class constructor.
- When you create an object in Java, the JVM allocates memory to hold the object's fields and methods.
- Once the memory has been allocated, the JVM calls the object's constructor to initialize the object's fields. If the class has no constructor, then the JVM initializes the fields to their default values (0, false, or null, depending on the data type).

After the object has been created and initialized, you can use it to perform operations by calling its methods and accessing its fields.

Constructors: a constructor is a special method that is called when an object of a class is created using the new keyword. Constructors are used to initialize the object's fields.

```
class MyClass
{
  // Fields
  int field1;
  String field2;

  // Constructor
  MyClass(int arg1, String arg2)
        {
    field1 = arg1;
    field2 = arg2;
  }

  // Methods
  void doSomething(){
    // Method code here
  }
}
```

In this example, we've defined a class called MyClass with two fields (field1 and field2) and a constructor that takes two arguments (arg1 and arg2). The constructor sets the values of the fields using the values of the arguments.

When you create an object of the MyClass class using the new keyword, the constructor is called automatically and the object is initialized with the values of the arguments passed to the constructor. Here's an example:

```
MyClass obj = new MyClass(10, "hello");
```

Features/Rules of constructors:

1. Constructors have the same name as the class in which they are defined.
2. Constructors do not have a return type, not even void.
3. Constructors are used to initialize the object's fields and perform any other setup that is required before the object can be used.
4. Any number of constructors can be defined in a class.
5. Constructor contains this and this() statements.

Types of constructors:

1. **Default Constructor:**
2. **Parameterized Constructor:**

1. Default Constructor: A default constructor is a constructor that takes no arguments. It is provided by the compiler if you do not define any constructor in your class.

The default constructor initializes all the instance variables of a class to their default values, which is 0 for numeric types, false for boolean, and null for object references.

Ex:

```
class MyClass
{
   int x; // default value is 0

   // Default Constructor
   public MyClass()
   {
   }
}
```

2. Parameterized Constructor: A parameterized constructor is a constructor that takes one or more parameters. It is used to initialize the instance variables of a class with the values passed as arguments to the constructor.

```
public class MyClass
{
   int x;
   String name;
```

```
    // Parameterized Constructor
    public MyClass(int x, String name)
     {
       this.x = x;
       this.name = name;
    }
}
```

Note: this is a current class current object reference, it is present in every non static contexts of the class. It is used to access data members of the object.

Constructor Overloading: Constructor overloading is a feature in Java that allows a class to have multiple constructors with the same name but with different parameters. When an object is created, the constructor is called to initialize the object. By overloading constructors, you can create objects using different sets of arguments.

To overload a constructor, you need to define multiple constructors with the same name but with different parameter lists. The parameter lists can differ in the number, type, or order of parameters.

For example:

```
class MyClass
{
   int x;
   int y;

   MyClass()
   {
      x = 0;
      y = 0;
   }

   MyClass(int x)
   {
      this.x = x;
      this.y = 0;
   }

   MyClass(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
}
```

When you create an object of MyClass, the appropriate constructor is called based on the arguments passed:

```
MyClass obj1 = new MyClass(); // calls the first constructor
MyClass obj2 = new MyClass(10); // calls the second constructor
MyClass obj3 = new MyClass(20, 30); // calls the third constructor
```

Methods: a method is a block of code that performs a specific task. A method can be defined inside a class and can be called from other parts of the program. It is also a members of the class.

Here's the syntax for creating a method in Java:

```
access_modifier return_type method_name(parameter_list)
{
  // method body
}
```

Access modifier: It specifies the access level of the method. It can be public, private, protected, or default (package-private).
Return type: It specifies the type of value that the method returns. If the method doesn't return anything, you can use the void keyword.
Method name: It's the name of the method that you want to define.
Parameter list: It's a list of parameters (comma-separated) that the method takes as input. If the method doesn't take any input, you can leave it empty.
Method body: It's the code that performs the specific task of the method.

Example:

```
class MyMath
{
  int sum(int a, int b)
  {
    int result = a + b;
    return result;
  }
}
```

In this example, we have defined a class called MyMath that has a public method called sum. The sum method takes two integers as input (a and b) and returns their sum as an integer. Inside the method body, we have defined a local variable called result and assigned the sum of a and b to it. Finally, we have returned the result from the method.

To call this method from another part of the program, you can create an object of the MyMath class and call the sum method on it:

```
class Main
{
   public static void main(String[] args)
    {
      MyMath m1 = new MyMath();
      int sum = m1.sum(10, 20);
      System.out.println("The sum of 10 and 20 is " + sum);
   }
}
```

Method Overloading: Method overloading in Java is a technique that allows you to define multiple methods with the same name in the same class. The methods must have different parameter lists (number, order, or type of parameters) but can have the same or different return types.

Here's an example:

```
class Calculator
{
   int add(int a, int b)
  {
      return a + b;
   }

double add(double a, double b)
  {
      return a + b;
   }
}
class Main
{
   public static void main(String[] args)
         {
      Calculator calc = new Calculator();

      int sum1 = calc.add(10, 20);
      double sum2 = calc.add(10.5, 20.7);

      System.out.println("Sum of 10 and 20 is " + sum1);
      System.out.println("Sum of 10.5 and 20.7 is " + sum2);

   }
}
```

In this example, we have defined three methods with the name "add" but with different parameter lists. The first method takes two integers as input and returns their sum as an integer. The second method takes two doubles as input and returns their sum as a double.

When you call the add method with different types or numbers of arguments, the Java compiler automatically chooses the appropriate version of the method based on the arguments you pass in. For example:

instanceof operator: The instanceof operator in Java is used to test whether an object is an instance of a particular class, subclass, or interface. It returns a boolean value that indicates whether the object is an instance of the specified type or not.

Syntax:

```
object instanceof type
```

Ex:

```
Boolean res = p1 instancef Person;
```

Visibility modifiers/Access Modifiers/Access Specifiers: **access modifiers, also known as visibility modifiers or access specifiers, are keywords used to specify the visibility or accessibility of classes, variables, methods, and constructors**. They determine whether a particular class or member can be accessed by other classes and code outside of its own class.

Java has four access modifiers:

public: A public class, method, or variable can be accessed from anywhere in the Java program. It has the widest scope and is used to make a class or member visible to all.

protected: A protected class, method, or variable can be accessed within its own package and by any subclass of the class in any package. It is used to make a class or member visible within its package and its subclasses.

default or package: A class, method, or variable without any access modifier is known as default or package-private access. It can only be accessed within the same package. If no access modifier is specified, this is the default visibility.

private: A private class, method, or variable can only be accessed within its own class. It is used to restrict access to a class or member to the class in which it is declared.

Here is an example to illustrate the use of access modifiers:

Example:

Static Members Vs Non-static(instance) members:

There are two types of class members: static (class) members and non-static (instance) members.

Static members are prefixed with static keyword and single copy will be created and shared across all instances of a class and are associated with the class itself,.
Accessed through Class names.

while non-static members are unique to each instance of the class and are associated with each individual object. A separate copy of the variables will be created for each object.
Accessed through object references.

Ex:

```
class MyClass
{
   static int x=10;
   int y;

   MyClass(int y)
   {
         this.y = y;
   }

   public static void main(String[] args)
    {
         MyClass m1 = new MyClass(12);
         System.out.println("instance variable = "+m1.y);
         System.out.println("static variable = "+MyClass.x);
         System.out.println("static variable through object = "+m1.x);
    }

}
```

Output:

this reference: this is a keyword which holds current object reference in constructor and non-static methods. It is used to access the instance variables.

When we have a naming conflict between local and instance variables that time, we can use this to access instance variables.

Ex:

```
class MyClass
{
   int x;                          //instance variable

   public void setX(int x)    //local variable
   {
      this.x = x; // Use 'this' to refer to the instance variable 'x'
   }
}
```

Arrays in java: array is a collection of elements of the same type that is stored in a contiguous block of memory. Each element in the array can be accessed using an index, which starts from 0 and goes up to the length of the array minus 1.

Here's an example of how to declare and initialize an array of integers in Java:

```
int[] myArray = new int[5]; // Declare an array of 5 integers
myArray[0] = 1; // Set the first element to 1
myArray[1] = 2; // Set the second element to 2
myArray[2] = 3; // Set the third element to 3
myArray[3] = 4; // Set the fourth element to 4
myArray[4] = 5; // Set the fifth element to 5
```

Arrays can be created for primitive types, object types also.

Ex:

```
Person p[] = new Person[5];
p[0] = new Person();
p[1] = new Person();
p[2] = new Person();
p[3] = new Person();
p[4] = new Person();
```

Inbuilt classes like : java.lang pckage provides many inbuilt and wrapper classes for many java application developments.

String: The String class is used to represent a sequence of characters. Strings in Java are immutable, which means that once a string is created, it cannot be changed.

String objects can be created using,

String str = "Hello, World!";
String s1 = new String("Hello,");

String Class methods:

Here is a list of commonly used methods in the String class in Java:

charAt(int index): Returns the character at the specified index.

length(): Returns the length of the string.

substring(int beginIndex): Returns a new string that is a substring of this string, starting from the specified index and extending to the end of the string.

substring(int beginIndex, int endIndex): Returns a new string that is a substring of this string, starting from the specified beginIndex and extending to the endIndex-1.

toLowerCase(): Returns a new string with all characters in lowercase.

toUpperCase(): Returns a new string with all characters in uppercase.

trim(): Returns a new string with leading and trailing white spaces removed.

equals(Object obj): Compares this string to the specified object. Returns true if the object is a string and has the same value as this string.

equalsIgnoreCase(String str): Compares this string to the specified string, ignoring case considerations.
startsWith(String prefix): Tests if this string starts with the specified prefix.

endsWith(String suffix): Tests if this string ends with the specified suffix.

indexOf(int ch): Returns the index within this string of the first occurrence of the specified character.

lastIndexOf(int ch): Returns the index within this string of the last occurrence of the specified character.

replace(char oldChar, char newChar): Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

replaceAll(String regex, String replacement): Replaces each substring of this string that matches the given regular expression with the given replacement string.

Example:

```
public class Demo
{
        public static void main(String[] args)
        {
                String s1 = "Hello";
                String s2 = " world";
                System.out.println("Length : "+s1.length());
                System.out.println("character position 2 : "+s1.charAt(2));
                System.out.println("Substring : "+s1.substring(2));
                System.out.println("LowerCase : "+s1.toLowerCase());
                System.out.println("Uppercase : "+s1.toUpperCase());
                System.out.println("Equality  : "+s1.equals(s2));

        }

}
```

Output:

Character: The Character class is a wrapper class for the primitive data type char in Java. It provides a number of useful methods for working with characters. Here are some commonly used methods in the Character class:

isLetter(char ch): Returns true if the specified character is a letter.

isDigit(char ch): Returns true if the specified character is a digit.

isWhitespace(char ch): Returns true if the specified character is whitespace.

isUpperCase(char ch): Returns true if the specified character is uppercase.

isLowerCase(char ch): Returns true if the specified character is lowercase.

toUpperCase(char ch): Converts the specified character to uppercase.

toLowerCase(char ch): Converts the specified character to lowercase.

Example:

```
Character c1 = 'A';
System.out.println("Is letter : "+c1.isLetter(c1));
System.out.println("Is digit : "+c1.isDigit(c1));
System.out.println("Is lowercase : "+c1.isLowerCase(c1));
System.out.println("is uppercase : "+c1.isUpperCase(c1));
System.out.println("Uppercase : "+c1.toUpperCase('d'));
```

Output:

String Buffer: the StringBuffer class provides an efficient way to manipulate strings. It is similar to the String class, but unlike String, StringBuffer objects are mutable, meaning that their contents can be changed. Here are some important features of the StringBuffer class:

Here are some commonly used methods in the StringBuffer class:

append(String str): Appends the specified string to the end of the current string.

insert(int offset, String str): Inserts the specified string at the specified offset.

delete(int start, int end): Deletes the characters from the specified start offset to the specified end offset.

replace(int start, int end, String str): Replaces the characters from the specified start offset to the specified end offset with the specified string.

reverse(): Reverses the characters in the string.

Example:

```
StringBuffer sb = new StringBuffer("Hello World");
System.out.println("Append : "+sb.append("Mr. Java"));
System.out.println("Insert substring : "+sb.insert(6, "OOPs "));
System.out.println("Delete string : "+sb.delete(6,10));
System.out.println("Replace string : "+sb.replace(6, 15, "Fresher "));
```

Difference between String and StringBuffer:

String	StringBuffer
String objects are immutable, meaning once a String object is created, its value cannot be changed.	StringBuffer, on the other hand, is mutable, meaning its value can be changed.
String objects are thread-safe, meaning they can be safely shared across multiple threads without the need for external synchronization.	StringBuffer, however, is not thread-safe by default. If multiple threads are modifying the same StringBuffer object simultaneously, then synchronization needs to be added to avoid data inconsistency issues.
Performance is higher	Lower performance

Note: Refer all Java Lab journal programs on this unit.