# Unit-3

## 4 marks

**1.Name the keyword used to create a new instance of a class in java.**

### Java new Keyword

The Java new keyword is used to create an instance of the class. In other words, it instantiates a class by allocating memory for a new object and returning a reference to that memory. We can also use the new keyword to create the array object.

- It is used to create the object.
- It allocates the memory at runtime.
- All objects occupy memory in the heap area.
- It invokes the object constructor.
- It requires a single, postfix argument to call the constructor

**Syntax:**

```
NewExample obj=new NewExample();
```

### Examples of Java new Keyword

Let's see a simple example to create an object using new keyword and invoking the method using the corresponding object reference.

```
public class NewExample1 {

   void display()
   {
      System.out.println("Invoking Method");
   }

   public static void main(String[] args) {
      NewExample1 obj=new NewExample1();
           obj.display();
   }

}
```

**Output:**

```
Invoking Method
```

## 2.Recall the method used to define a class in java.

**Java Class Methods**

You learned from the Java Methods chapter that methods are declared within a class, and that they are used to perform certain actions:

**Example**

Create a method named myMethod() in Main:

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
      }}
```

**Static vs. Public**

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

**Example**

An example to demonstrate the differences between static and public **methods**:

```
public class Main {

// Static method

static void myStaticMethod() {

System.out.println("Static methods can be called without creating objects");

 }

 // Public method
public void myPublicMethod() {

System.out.println("Public methods must be called by creating objects");

}

// Main method

 public static void main(String[] args) {

 myStaticMethod(); // Call the static method

 // myPublicMethod(); This would compile an error

 Main myObj = new Main(); // Create an object of Main

 myObj.myPublicMethod(); // Call the public method on the object

  }}
```

**Output:**

```
Static methods can be called without creating objects
Public methods must be called by creating objects
```

**3.Describe how objects are created from classes and how they interact with each other.**

**Java Classes**

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

**Class Declaration in Java**

*access_modifier* **class** <*class_name*>
{
    data member;
    method;

```
    constructor;
    nested class;
    interface;
}
```

**Example of Java Class**

```
// Java Program for class example

class Student {
      // data member (also instance
variable)
   int id;
      // data member (also instance
variable)
   String name;

      public static void main(String
args[])
   {
      // creating an object of
      // Student
      Student s1 = new Student();
      System.out.println(s1.id);
      System.out.println(s1.name);
   }
}
```

**Output**

```
Null
```

**Java Objects**
An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class

**Example program**

```
public class GFG {
   // sw=software
   static String sw_name;
   static float sw_price;
```

```
    static void set(String n, float p)
    {
        sw_name = n;
        sw_price = p;
    }

    static void get()
    {
        System.out.println("Software name is: " + sw_name);
        System.out.println("Software price is: "
                           + sw_price);
    }

    public static void main(String args[])
    {
        GFG.set("Visual studio", 0.0f);
        GFG.get();
    }
}
```

**Output**

```
Software name is: Visual studio
Software price is: 0.0
```

**4.Explain the role of constructors in java classes and their importance in object creation.**

**Java Classes**

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

**Class Declaration in Java**

```
access_modifier class <class_name>
{
    data member;
    method;
    constructor;
    nested class;
    interface;
}
```

**Example of Java Class**

```
// Java Program for class example

class Student {
      // data member (also instance variable)
   int id;
      // data member (also instance variable)
   String name;

      public static void main(String args[])
   {
      // creating an object of
      // Student
      Student s1 = new Student();
      System.out.println(s1.id);
      System.out.println(s1.name);
   }
}
```

**Output**

```
Null
```

**Java Objects**
An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class

**Example program**

```
public class GFG {
   // sw=software
   static String sw_name;
   static float sw_price;

   static void set(String n, float p)
   {
      sw_name = n;
      sw_price = p;
   }
   static void get()
   {
      System.out.println("Software name is: " + sw_name);
      System.out.println("Software price is: "
                     + sw_price);
   }

   public static void main(String args[])
   {
      GFG.set("Visual studio", 0.0f);
      GFG.get();
   }
}
```

**Output**

```
Software name is: Visual studio
Software price is: 0.0
```

**5.Write a java program that demonstrates various string manipulation operations like concatenation, substring , and length retrieval.**

**Use StringBuilder or StringBuffer for String Concatenation**

Avoid using the "+" operator repeatedly when concatenating multiple strings. This can create unnecessary string objects, leading to poor performance. Instead, use StringBuilder (or StringBuffer for thread safety) to efficiently concatenate strings.

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString(); // "Hello
World"
```

**Prefer StringBuilder over StringBuffer**

If thread safety is not a concern, use StringBuilder instead of StringBuffer. StringBuilder is faster because it's not synchronized.

```
public class StringBuilderExample {
   public static void main(String[] args) {
      StringBuilder stringBuilder = new StringBuilder();
```

```
        stringBuilder.append("Hello");
        stringBuilder.append(" ");
       stringBuilder.append("World");

        String result = stringBuilder.toString();

        System.out.println("StringBuilder result: " + result); // Output: StringBuilder result: 
Hello World
    }
}
```

**Output**

```
StringBuilder result: Hello World
```

**length retrieval.**

```
String str = "Hello";
for (char c : str.toCharArray()) {
   // Do something with each character
}

// Alternatively, using StringBuilder
StringBuilder sb = new StringBuilder(str);
for (int i = 0; i < sb.length(); i++) {
   char c = sb.charAt(i);
   // Do something with each character
}
```

**6.Write a java program that demonstrates single inheritance between two classes**

```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class One {
        public void print_geek()
        {
                System.out.println("Geeks");
        }
}

class Two extends One {
        public void print_for() { System.out.println("for"); }
}

// Driver class
public class Main {
	// Main function
	public static void main(String[] args)
	{
		Two g = new Two();
		g.print_geek();
		g.print_for();
		g.print_geek();
	}
}
```

**Output:**

```
Geeks
for
Geeks
```

**7.Write a java program that demonstrates the usage of regular expressions to validate user input (e.g., email address, phone number).**

```
// Java Program to Check if Mobile Number is Valid or Not

// Importing all regex classes from java.util package to
// match character sequence against patterns
// Importing input output classes
import java.io.*;
import java.util.regex.*;
// Main class
class GFG {

	// Method 1
	// To check whether number is valid or not
	public static boolean isValid(String s)
	{

		// The given argument to compile() method
		// is regular expression. With the help of
		// regular expression we can validate mobile
		// number.
		// The number should be of 10 digits.

		// Creating a Pattern class object
		Pattern p = Pattern.compile("^\\d{10}$");

		// Pattern class contains matcher() method
		// to find matching between given number
		// and regular expression for which
```

```
        // object of Matcher class is created
        Matcher m = p.matcher(s);

        // Returning boolean value
        return (m.matches());
    }

    // Method 2
    // Main driver method
    public static void main(String[] args)
    {

        // Considering two numbers as inputs
        // Custom input numbers
        String s = "7984286257";
        String s_1 = "5426391";

        // Checking over method 1 for string 1
        if (isValid(s))

            // Print statement
            System.out.println("Valid Number");
        else

            // Print statement
            System.out.println("Invalid Number");

        // Again, checking over method 1 for string 1
        if (isValid(s_1))

            // Print statement
            System.out.println("Valid Number");
        else

            // Print statement
            System.out.println("Invalid Number");
    }
}
```

**Output:**

```
Valid Number
Invalid Number
```

**8.Analyze and illustrate how exception handling in java affects program flow and error management.**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

**Types of Exceptions**

**1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

**2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, Arithmetic Exception, NullPointerException, Array Index Out Of Bounds Exception, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

**3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, Assertion Error etc.

**ExceptionExample program**

```
public class JavaExceptionExample{
 public static void main(String args[]){
  try{
     //code that may raise exception
     int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");    }
}
```

**Output:**

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

**9.Analyze a java code snippet that demonstrates inheritance and identify the interactions between superclass and subclass methods.**

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

**Example program**

```
class Vehicle {

 protected String brand = "Ford";        // Vehicle attribute

 public void honk() {                    // Vehicle method

   System.out.println("Tuut, tuut!");

     }}

   class Car extends Vehicle {

   private String modelName = "Mustang";    // Car attribute

   public static void main(String[] args) {

   // Create a myCar object

   Car myCar = new Car();

   // Call the honk() method (from the Vehicle class) on the myCar object

   myCar.honk();

   // Display the value of the brand attribute (from the Vehicle class) and the value of the
modelName from the Car class

      System.out.println(myCar.brand + " " + myCar.modelName);

   }}
```

**Output:**

```
Tuut, tuut!
   Ford Mustang
```

**10.Analyze a java code snippet that contains multiple string manipulation operations and identify potential issues related to string immutability.**

**Use StringBuilder or StringBuffer for String Concatenation**

Avoid using the "+" operator repeatedly when concatenating multiple strings. This can create unnecessary string objects, leading to poor performance. Instead, use StringBuilder (or StringBuffer for thread safety) to efficiently concatenate strings.

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString(); // "Hello
World"
```

**Prefer StringBuilder over StringBuffer**

If thread safety is not a concern, use StringBuilder instead of StringBuffer. StringBuilder is faster because it's not synchronized.

```
public class StringBuilderExample {
   public static void main(String[] args) {
      StringBuilder stringBuilder = new StringBuilder();

      stringBuilder.append("Hello");
      stringBuilder.append(" ");
     stringBuilder.append("World");

      String result = stringBuilder.toString();

      System.out.println("StringBuilder result: " + result); // Output: StringBuilder result:
Hello World
   }
}
```

**Output**

```
StringBuilder result: Hello World
```

**length retrieval.**

```
String str = "Hello";
for (char c : str.toCharArray()) {
    // Do something with each character
}

// Alternatively, using StringBuilder
StringBuilder sb = new StringBuilder(str);
for (int i = 0; i < sb.length(); i++) {
    char c = sb.charAt(i);
    // Do something with each character
}
```

**Output:**

## 11.Analyze and illustrate the role of constructors in initializing object states and behaviours.

**Java Objects**

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

**Example program**

```
public class GFG {
    // sw=software
    static String sw_name;
    static float sw_price;

    static void set(String n, float p)
    {
        sw_name = n;
        sw_price = p;
    }

    static void get()
    {
        System.out.println("Software name is: " + sw_name);
        System.out.println("Software price is: "
                         + sw_price);
    }
    public static void main(String args[])
    {
        GFG.set("Visual studio", 0.0f);
        GFG.get();
    }
}
```

**Output**

```
Software name is: Visual studio
Software price is: 0.0
```

**12.Design a java program that efficiently uses classes and objects to model a real-world scenario or problem**

```
// Java Program to Check if Mobile Number is Valid or Not

// Importing all regex classes from java.util package to
// match character sequence against patterns
// Importing input output classes
import java.io.*;
import java.util.regex.*;
// Main class
class GFG {

        // Method 1
        // To check whether number is valid or not
        public static boolean isValid(String s)
        {

                // The given argument to compile() method
                // is regular expression. With the help of
                // regular expression we can validate mobile
                // number.
                // The number should be of 10 digits.

                // Creating a Pattern class object
                Pattern p = Pattern.compile("^\\d{10}$");

                // Pattern class contains matcher() method
                // to find matching between given number
                // and regular expression for which
                // object of Matcher class is created
                Matcher m = p.matcher(s);

                // Returning boolean value
                return (m.matches());
        }
```

```
    // Method 2
    // Main driver method
    public static void main(String[] args)
    {

        // Considering two numbers as inputs
        // Custom input numbers
        String s = "7984286257";
        String s_1 = "5426391";

        // Checking over method 1 for string 1
        if (isValid(s))

            // Print statement
            System.out.println("Valid Number");
        else

            // Print statement
            System.out.println("Invalid Number");

        // Again, checking over method 1 for string 1
        if (isValid(s_1))

            // Print statement
            System.out.println("Valid Number");
        else

            // Print statement
            System.out.println("Invalid Number");
    }
}
```

**Output:**

```
Valid Number
Invalid Number
```

**13.Identify the java operator used for string concatenation.**

**Concatenation** is the process of combining two or more strings to form a new string by subsequently appending the next string to the end of the previous strings.

In Java, two strings can be concatenated by using the + or += **operator**, or through the **concat() method**, defined in the java.lang.String class.

This shot will discuss how to perform string concatenation using both of these methods.

**Example 1**

The program below demonstrates how to concatenate two strings using the + operator in Java.

```
class HelloWorld {
public static void main( String args[] ) {
 String first = "Hello";
 String second = "World";
String third = first + second;
 System.out.println(third);
// yet another way to concatenate strings first += second;
System.out.println(first);
 }
 }
```

**Output:**

```
    Hello world
```

**14.Recall the method used to obtain the length of a string in java.**

**Java String length()**

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

The String class internally uses a char[] array to store the characters. The length variable of the array is used to find the total number of elements present in the array. Since the Java String class uses this char[] array internally; therefore, the length variable can not be exposed to the outside world.

Hence, the Java developers created the length() method, the exposes the value of the length variable. One can also think of the length() method as the getter() method, that provides a value of the class field to the user. The internal implementation clearly depicts that the length() method returns the value of then the length variable.

**Java String length() method example**

```
public class LengthExample{

public static void main(String args[]){
String s1="javatpoint";
String s2="python";
System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
System.out.println("string length is: "+s2.length());//6 is the length of python string
}}
```

**Output:**

```
string length is: 10
string length is: 6
```

## 6 marks

## 1.Polymorphism and interface in java

## Polymorphism

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

**Example program**

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

**Output**

```
running safely with 60km
```

## Interfaces

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Syntax:**

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

**Example program**

```
interface printable{
void print();
}
    class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

**Output:**

```
Hello
```

## 2.Differentiate classes and objects in c++ and java

**Difference Between Class And Object:**

There are many differences between object and class. Some differences between object and class are given below:

| Class | Object |
|---|---|
| Class is used as a template for declaring and | An object is an instance of a class. |

| **Class** | **Object** |
|---|---|
| creating the objects. | |
| When a class is created, no memory is allocated. | Objects are allocated memory space whenever they are created. |
| The class has to be declared first and only once. | An object is created many times as per requirement. |
| A class can not be manipulated as they are not available in the memory. | Objects can be manipulated. |
| A class is a logical entity. | An object is a physical entity. |
| It is declared with the class keyword | It is created with a class name in C++ and with the **new** keywords in Java. |
| Class does not contain any values which can be associated with the field. | Each object has its own values, which are associated with it. |
| A class is used to bind data as well as methods together as a single unit. | Objects are like a variable of the class. |
| **Syntax:** Declaring Class in C++ is as follows:<br>class &lt;classname&gt; {}; | **Syntax:** Instantiating an object for a Class in C++ is as follows:<br>class Student {<br>  public:<br>    void put(){<br>      cout<<“Function Called”<<endl;<br>    }<br>};   // The class is declared here<br>int main(){<br>      Student s1;   // Object created<br>      s1.put();<br>} |
| **Example:** Bike | **Example:** Ducati, Suzuki, Kawasaki |

**3.inheritance and it’s types**

Inheritance is the mechanism in which one class acquire all the feautures of another class. Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.we can achieve inheritance by using the extends keyword .it facilitates the reusability of the code.

**Syntax :**

```
class derived-class extends base-class
{
  //methods and fields
}
```

**Java Inheritance Types**

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

**1. Single Inheritance**

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.

**2. Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

**3. Hierarchical Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D

**4. Multiple Inheritance (Through Interfaces)**

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.

**4. Hybrid Inheritance**

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

**Example program**

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

**Output:**

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

## 4.Packages in JAVA

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

### Advantage of Java Package

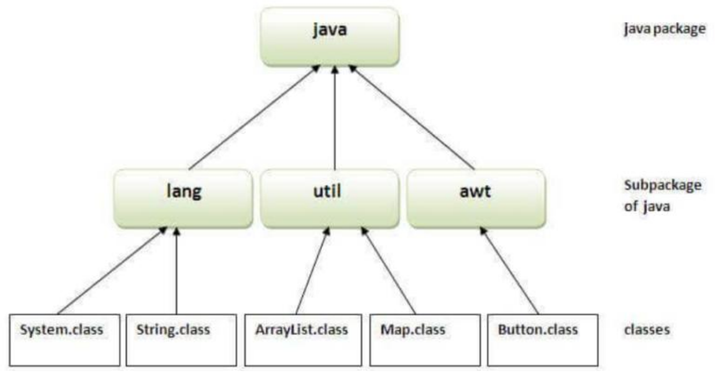
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

**syntax**

```
javac -d directory javafilename
```



**Simple example of java package**

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

**OUTPUT:**

```
Welcome to package
```

**5. is there any alternative solution for inheritance? If so explain the advantages and disadvantages of it.**

Inheritance  in Java programming is the process by which one class takes the property of another other class. i.e. the new classes, known as derived or child class, take over the attributes and behavior of the pre-existing classes, which are referred to as base classes or super or parent class.
**Delegation** is simply passing a duty off to someone/something else.

- Delegation can be an alternative to inheritance.
- Delegation means that you use an object of another class as an instance variable, and forward messages to the instance.
- Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate.

**Example program**

```
// Java program to illustrate
// delegation
class RealPrinter {
   // the "delegate"
   void print()
   {
      System.out.println("The Delegate");
   }
}

class Printer {
   // the "delegator"
   RealPrinter p = new RealPrinter();

   // create the delegate
   void print()
   {
      p.print(); // delegation
   }
}

public class Tester {

    // To the outside world it looks like Printer actually
prints.
public static void main(String[] args)
   {
      Printer printer = new Printer();
      printer.print();
   }
}
```

**Output:**

```
The Delegate
```

## 6. Summarize the JAVA Structure

Let's see which elements are included in the structure of a Java program. A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables

- Main Method Class
- Methods and Behaviors

**Documentation Section**

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program.

**Package Declaration**

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration.

**Import Statements**

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement.

**Interface Section**

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword.

**Class Definition**

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class.

**Class Variables and Constants**

In this section, we define variables and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters.

**Main Method Class**

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class.

**Methods and behavior**

In this section, we define the functionality of the program by using the methods. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task.

**7.Give a note on Classes and Objects in JAVA.**

**Java Classes**

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

**Class Declaration in Java**

```
access_modifier class <class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

**Example of Java Class**

```
// Java Program for class example

class Student {
      // data member (also instance variable)
   int id;
      // data member (also instance variable)
   String name;

      public static void main(String args[])
   {
      // creating an object of
      // Student
      Student s1 = new Student();
      System.out.println(s1.id);
      System.out.println(s1.name);
   }
}
```

**Output**

```
Null
```

**Java Objects**
An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class

**Example program**

```
public class GFG {
   // sw=software
   static String sw_name;
   static float sw_price;

   static void set(String n, float p)
   {
      sw_name = n;
      sw_price = p;
   }

   static void get()
   {
      System.out.println("Software name is: " + sw_name);
      System.out.println("Software price is: "
                  + sw_price);
   }

   public static void main(String args[])
   {
      GFG.set("Visual studio", 0.0f);
      GFG.get();
   }
}
```

**Output**

```
Software name is: Visual studio
Software price is: 0.0
```

**8.Develop a performance of Strings in JAVA.**

Strings are the type of objects that can store the character of values and in Java, every character is stored in 16 bits i,e using UTF 16-bit encoding. A string acts the same as an array of characters in Java.

**Example Program:**

```
// Java Program to demonstrate
// String
public class StringExample {

   // Main Function
   public static void main(String args[])
   {
      String str = new String("example");
      // creating Java string by new keyword
      // this statement create two object i.e
      // first the object is created in heap
      // memory area and second the object is
      // created in String constant pool.

      System.out.println(str);
   }
}
```

**Output:**

```
example
```

**Ways of Creating a String**

There are two ways to create a string in Java:

- String Literal
- Using new Keyword

**Syntax:**

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

**1. String literal**

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

**Example:**

String demoString = “GeeksforGeeks”;

**2. Using new keyword**

- String s = new String(“Welcome”);
- In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal “Welcome” will be placed in the string constant pool. The variable s will refer to the object in the heap (non-pool)

**Example:**

String demoString = new String (“GeeksforGeeks”);

**9.Develop a performance of inheritance and its type**

Inheritance is the mechanism in which one class acquire all the feautures of another class. Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.we can achieve inheritance by using the extends keyword .it facilitates the reusability of the code.

**Syntax :**

```
class derived-class extends base-class
{
  //methods and fields
}
```

**Java Inheritance Types**

Below are the different types of inheritance which are supported by Java.

1.Single Inheritance
2.Multilevel Inheritance
3.Hierarchical Inheritance
4.Multiple Inheritance
5.Hybrid Inheritance

**1.Single Inheritance**

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.

**2.Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

**3.Hierarchical Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D

**4. Multiple Inheritance (Through Interfaces)**

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.

**5.Hybrid Inheritance**

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

## Example program

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

**Output:**

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

**10.Analyze the impact of interfaces in JAVA.**

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class.

An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

The interface in Java is *a* mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritances in Java using Interface.

**Syntax for Java Interfaces**

```
interface {
   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

**Example program**

```
interface printable{
void print();
}
    class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

**Output:**

```
Hello
```

**11.Discuss in detail about exception Handling:**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

**Types of Exceptions**

**1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

**2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, Arithmetic Exception, NullPointerException, Array Index Out Of Bounds Exception, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

**3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, Assertion Error etc.

**ExceptionExample program**

```
public class JavaExceptionExample{
 public static void main(String args[]){
  try{
     //code that may raise exception
     int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");    }
}
```

**Output:**

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

**12.Assess the impact of Regular Expressions**

In Java, Regular Expressions or Regex (in short) in Java is an API for defining String patterns that can be used for searching, manipulating, and editing a string in Java. Email validation and passwords are a few areas of strings where Regex is widely used to define the constraints. Regular Expressions in Java are provided under **java.util.regex** package. This consists of **3 classes and 1 interface**.

**Regex Classes and Interfaces**

**Regex in Java provides 3 classes and 1 interface which are as follows:**

1. Pattern Class
2. Matcher Class
3. PatternSyntaxException Class
4. MatchResult Interface

**Pattern Class**
This class is a compilation of regular expressions that can be used to define various types of patterns, providing no public constructors. This can be created by invoking the compile() method which accepts a regular expression as the first argument, thus returning a pattern after execution.

**Matcher class**
This object is used to perform match operations for an input string in Java, thus interpreting the previously explained patterns. This too defines no public constructors. This can be implemented by invoking a matcher() on any pattern object.

**Example Program**

```
import java.util.regex.*;
public class RegexExample1{
public static void main(String args[]){
//1st way
Pattern p = Pattern.compile(".s");//. represents single character
Matcher m = p.matcher("as");
boolean b = m.matches();

//2nd way
boolean b2=Pattern.compile(".s").matcher("as").matches();

//3rd way
boolean b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);
}}
```

**Output**

```
true true true
```

**13.Constract a training program for Polymorphism**

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

**Example program**

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

**Output**

```
running safely with 60km
```

**14.create a java class hierarchy that demonstrates multiple levels of inheritance to showcase the depth and flexibility of inheritance in java.**

Inheritance is the mechanism in which one class acquire all the feautures of another class. Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.we can achieve inheritance by using the extends keyword .it facilitates the reusability of the code.

**Syntax :**

```
class derived-class extends base-class
{
   //methods and fields
}
```

**Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

**Example program**

```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class One {
   public void print_geek()
   {
      System.out.println("Geeks");
   }
}

class Two extends One {
   public void print_for() { System.out.println("for"); }
}

class Three extends Two {
   public void print_geek()
   {
      System.out.println("Geeks");
   }
}

// Drived class
public class Main {
   public static void main(String[] args)
   {
      Three g = new Three();
      g.print_geek();
      g.print_for();
      g.print_geek();
   }
}
```

**Output**

```
Geeks
for
Geeks
```

## 10 marks

## 1.How to add interfaces to your JAVA Program

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Advantages of Interfaces in Java**
The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

**Uses of Interfaces in Java**

*Uses of Interfaces in Java are mentioned below:*

- *It is used to achieve total abstraction.*
- *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
- *Any class can extend only 1 class, but can any class implement an infinite number of interfaces.*
- *It is also used to achieve loose coupling.*
- *Interfaces are used to implement abstraction.*

**Syntax:**

```
interface <interface_name>{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

**Example program**

```
interface printable{
void print();
}
    class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

**Output:**

```
Hello
```

**2.Describe about designing tools in multimedia.**

Java provides a wide range of tools for different parts of the development process, from coding to launching applications.
Some tools are included in the **Java Development Kit** (**JDK**), while others are third-party tools developed by the community or other companies.
Java has a variety of tools to enhance application efficiency, code quality, and overall productivity of the application. Following are the **Java tools** that can be extremely useful for **Java developers**.

**Tools and Technology in Java**

**1. Integrated Development Environments (IDEs)**

**Integrated Development Environments** (IDEs) are software applications that provide a comprehensive set of tools for developers to write, edit, debug, test, and deploy code more efficiently.

**2. Build Tools and Dependency Management**

The build tools that automates the process of compiling, testing, documentation, automation that basically provides the consistent build process for the application, and packaging Java applications.

**3. Version Control System**

The **Version Control System** (VCS) is the tool which is used in software development for the tracking and managing of the changes that have been done in the codebases. It allows multiple developers to work on a project simultaneously without overwriting each other's changes and helps to maintain a proper record of code modifications.

**4. Continuous Integration/Continuous Deployment (CI/CD)**

The **Continuous Integration/Continuous Deployment** is a fundamental concept in modern software development, including projects written in Java, where CI refers to the practice of frequently integrating code changes into a shared repository. When developers push their changes to the shared repository multiple times a day to the master branch, then after each push, their automated tests run to verify these changes to ensure that they don't break the existing codebase in the actual repository.

## 5. Testing Tools

**Testing tools** are the tools which support various types of testing, ranging from unit testing to integration testing and beyond that build the test cases according to the boundary conditions of your application.

## 6. Static Code Analysis Tools

This tool examines the Java code without actually executing. Basically, they spot mistakes or parts of the code that might cause problems later, the main purpuse is they catch issues so you can fix them before they cause bigger problems.

## 7. Application Profilers

The Application Profilers is the diagnostic tool which is designed to analyse the pattern of Java applications, and monitors the runtime behaviour of your Java applications.

## 8. Database Management/ORM (Object Relational Mapping)

**It is the tools and techniques which are used to interact with the databases more easily and efficiently**. It's all about handling and organising data in the databases. In Java, it uses specific tools or such libraries to connect with the databases, it reduces the redundancy of the code or queries, and manage data in databases.

## 9. Web Development and Application Servers

**Building web apps with Java means making the online programs using Java**. These can be simple sites or big online systems developed by the Java developers. In Java, 'Servlets' are used to manage the website requests and give back responses.

## 10. Containerization & Orchestration

**Containerization is like putting a Java program and all it needs into a special box**. This ensures the program acts the same everywhere. This ensures the application runs the same, regardless of where the container is deployed meaning that you can package or integrate. Basically **it is like the program works the same no matter where you use the box**.

**3.Discuss about Exception Handling in JAVA.**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

**Types of Exceptions**

**1. Built-in Exceptions**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time

**2. User-Defined Exceptions:**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

**ExceptionExample program**

```
public class JavaExceptionExample{
 public static void main(String args[]){
  try{
     //code that may raise exception
     int data=100/0;
  }catch(ArithmeticException e){System.out.println(e);}
  //rest code of the program
  System.out.println("rest of the code...");    }
}
```

**Output:**

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

**4.Create a java interface and implement it in multiple classes demonstrate its usage.**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Multiple Inheritance in Java Using Interface**

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface.

**Advantages of Interfaces in Java**

The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

**Uses of Interfaces in Java**

*Uses of Interfaces in Java are mentioned below:*

- *It is used to achieve total abstraction.*
- *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
- *Any class can extend only 1 class, but can any class implement an infinite number of interfaces.*
- *It is also used to achieve loose coupling.*
- *Interfaces are used to implement abstraction.*

**Syntax:**

```
interface <interface_name>{

    // declare constant fields
```

```
        // declare methods that abstract
        // by default.
    }
```

**Example program**

```
interface printable{
void print();
}
    class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

**Output:**

```
    Hello
```

**5.Given a complex program, analyze and illustrate how java interfaces are implementated in different classes and their impact on the programs design.**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Multiple Inheritance in Java Using Interface**

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface.

**Advantages of Interfaces in Java**

The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

**Uses of Interfaces in Java**

*Uses of Interfaces in Java are mentioned below:*

- *It is used to achieve total abstraction.*
- *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
- *Any class can extend only 1 class, but can any class implement an infinite number of interfaces.*
- *It is also used to achieve loose coupling.*
- *Interfaces are used to implement abstraction.*

**Syntax:**

```
interface <interface_name>{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

**Example program**

```
interface printable{
void print();
}
    class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

**Output:**

```
Hello
```

**6.Evaluate the advantages and disadvantages of using classes and objects in java for code organization and reuasbility.**

**Classes:**

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

- It provides template for creating objects, which can bind code into data.
- It has definitions of methods and data.
- It supports inheritance property of Object Oriented Programming and hence can maintain class hierarchy.
- It helps in maintaining the access specifications of member variables.

**Properties of Java Classes**

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:

- Data member
- Method
- Constructor
- Nested Class
- Interface

**Objects:**

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

| **Class** | **Object** |
|---|---|
| Class is the blueprint of an object. It is used | An object is an instance of the class. |

| Class | Object |
|---|---|
| to create objects. | |
| No memory is allocated when a class is declared. | Memory is allocated as soon as an object is created. |
| A class is a group of similar objects. | An object is a real-world entity such as a book, car, etc. |
| Class is a logical entity. | An object is a physical entity. |
| A class can only be declared once. | Objects can be created many times as per requirement. |
| An example of class can be a car. | Objects of the class car can be BMW, Mercedes, Ferrari, etc. |

**7.Design a java program that efficiently uses strings to handle and process user input in a specific application.**

**Methods to Take Input in Java**

There are **two ways** by which we can take Java input from the user or from a file

- BufferedReader Class
- Scanner Class

**1. Using BufferedReader Class for String Input In Java**

It is a simple class that is used to read a sequence of characters. It has a simple function that reads a character another read which reads, an array of characters, and a readLine() function which reads a line.

InputStreamReader() is a function that converts the input stream of bytes into a stream of characters so that it can be read as BufferedReader expects a stream of characters. BufferedReader can throw checked Exceptions.

```
/*package whatever //do not write package name here */

import java.io.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;
class Easy {
        public static void main(String[] args)
        {
                // creating the instance of class BufferedReader
                BufferedReader reader = new BufferedReader(
                        new InputStreamReader(System.in));
                String name;
                try {
                        System.out.println("Enter your name");
                        name = reader.readLine(); // taking string input
                        System.out.println("Name=" + name);
                }
                catch (Exception e) {
                }
        }
}
```

**Output:**

```
Enter your name:
Geeks
Name=Geeks
```

**2. Using Scanner Class for Taking Input in Java**

It is an advanced version of BufferedReader which was added in later versions of Java. The scanner can read formatted input. It has different functions for different types of data types.

- The scanner is much easier to read as we don't have to write throws as there is no exception thrown by it.
- It was added in later versions of Java
- It contains predefined functions to read an Integer, Character, and other data types as well.

```
// Java Program to implement
// Scanner Class to take input
import java.io.*;
import java.util.Scanner;

// Driver Class
class Easy {
        // main function
        public static void main(String[] args)
        {
                // creating the instance of class Scanner
                Scanner obj = new Scanner(System.in);
                String name;
                int rollno;
                float marks;

                System.out.println("Enter your name");

                // taking string input
                name = obj.nextLine();
                System.out.println("Enter your rollno");

                // taking integer input
                rollno = obj.nextInt();
                System.out.println("Enter your marks");

                // taking float input
                marks = obj.nextFloat();

                // printing the output
                System.out.println("Name=" + name);
                System.out.println("Rollno=" + rollno);
                System.out.println("Marks=" + marks);
        }
}
```

**Output**

```
Enter your name
Geeks
Enter your rollno
5
Enter your marks
84.60
Name=Geeks
Rollno=5
Marks=84.60
```