CBCS SCHEME

USN 1 C R 2 1 M C 0 2 4

20MCA22

# Second Semester MCA Degree Examination, July/August 2022
## Object Oriented Programming with Java

Time: 3 hrs. Max. Marks:100

**Note:** ***Answer any FIVE full questions, choosing ONE full question from each module.***



### Module-1

1 a. Explain the Buzzwords of Java. **(10 Marks)**

b. Explain the various primitive data types used in Java. Give suitable examples. **(10 Marks)**

**OR**

2 a. Define an Array. Write the array declaration syntax for multidimensional arrays. Write a simple Java program to create an array of objects. **(10 Marks)**

b. Demonstrate the concept of garbage collection through suitable examples. **(10 Marks)**

### Module-2

3 a. Explain method overloading and constructor overloading with suitable examples. **(10 Marks)**

b. Explain the following :
i) Varargs
ii) Final keyword. **(10 Marks)**

**OR**

4 a. What are super class constructors? How does super class members are used in Java. **(10 Marks)**

b. What is Method-Overriding? Explain how it allows Java to support run-time polymorphism with an example. **(10 Marks)**

### Module-3

5 a. What are interfaces? What are their benefits? Explain how it is implemented in java with suitable example. **(10 Marks)**

b. What is Exception? Explain how exception handling mechanism can be used for debugging a program. **(10 Marks)**

**OR**

6 a. Define package. Explain the creation of a package using a suitable example program. **(10 Marks)**

b. Develop a simple program and explain multiple catch blocks. **(10 Marks)**

## Module-4

7 a. Define synchronization? Explain how inter-thread communication can be achieved in multithreading using producer and consumer problem. (10 Marks)

b. Define multi-threading? Construct a java program to create multiple threads in Java by implementing runnable interface. (10 Marks)

**OR**

8 a. Develop a simple program and explain how enumerations are class types. (10 Marks)

b. Analyze the following with syntax and example :
i) values( ) and valuesof( )
ii) compareTo( ). (10 Marks)

## Module-5

9 a. What are URL classes? Explain URL connections with syntax. (10 Marks)

b. What HttpURL connection class. Explain the methods of its. (10 Marks)

**OR**

10 a. Identify different types of methods defined by collection interface. (10 Marks)

b. Demonstrate a simple program, the constructors and methods in ArrayList class. (10 Marks)

* * * * *

Answers:

1.a. The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.

Simple

Object-Oriented

Portable

Platform independent

Secured

Robust

Architecture neutral

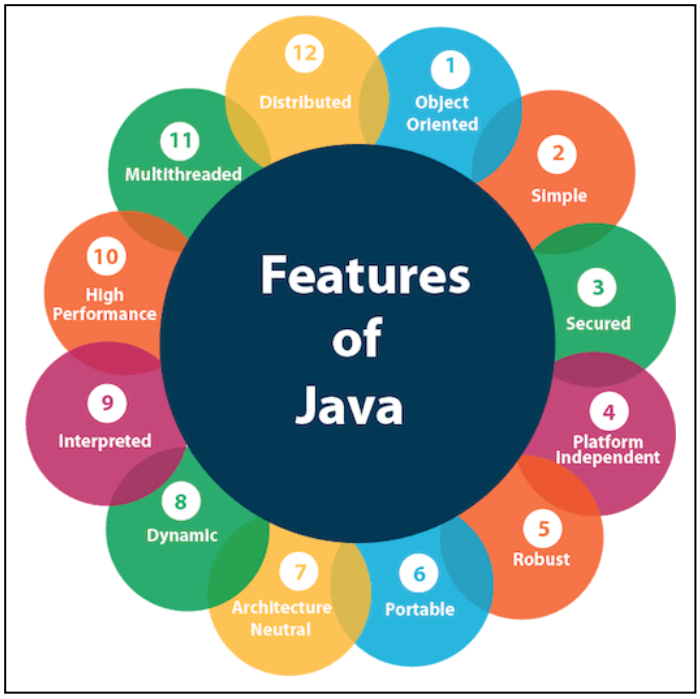
Interpreted

High Performance

Multithreaded

Distributed

Dynamic




**Platform Independent**

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

**Runtime Environment**

API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

**Secured**

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

No explicit pointer

Java Programs run inside a virtual machine sandbox

how Java is secured

Classloader: Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

Bytecode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.

Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

**Robust**

The English mining of Robust is strong. Java is robust because:

It uses strong memory management.

There is a lack of pointers that avoids security problems.

Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.

There are exception handling and the type checking mechanism in Java. All these points make Java robust.

**Architecture-neutral**

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

**Portable**

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

**High-performance**

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

**Distributed**

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

**Multi-threaded**

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

**Dynamic**

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

1.b. Data types are divided into two groups:

Primitive data types - includes byte, short, int, long, float, double, boolean and char

Non-primitive data types - such as String, Arrays and Classes

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

| Data Type | Size | Description |
|---|---|---|
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

Example:

```
int myNum = 5;              // Integer (whole number)

float myFloatNum = 5.99f;   // Floating point number

char myLetter = 'D';        // Character

boolean myBool = true;      // Boolean

String myText = "Hello";    // String
```

2.a. Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Program to create array of objects in Java

```
// Java program to demonstrate initializing

// an array of objects using a method

class GFG {

        public static void main(String args[])

        {
```

```
		// Declaring an array of student
		Student[] arr;
		// Allocating memory for 2 objects
		// of type student
		arr = new Student[2];
		// Creating actual student objects
		arr[0] = new Student();
		arr[1] = new Student();
		// Assigning data to student objects
		arr[0].setData(1701289270, "Satyabrata");
		arr[1].setData(1701289219, "Omm Prasad");
		// Displaying the student data
		System.out.println("Student data in student arr 0: ");
		arr[0].display();
		System.out.println("Student data in student arr 1: ");
		arr[1].display();
	}
}
// Creating a Student class with
// id and name as a attributes
class Student {
	public int id;
	public String name;
	// Method to set the data to
	// student objects
	public void setData(int id, String name)
	{
		this.id = id;
		this.name = name;
	}
	// display() method to display
```

```
        // the student data
        public void display()
        {
                System.out.println("Student id is: " + id + " "+ "and Student name is: "+ name);
                System.out.println();
        }
}
```

2.b.

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

What is Garbage Collection?

In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing OutOfMemoryErrors.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying unreachable objects. The garbage collector is the best example of the Daemon thread as it is always running in the background.

How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Types of Activities in Java Garbage Collection

Two types of garbage collection activity usually happen in Java. These are:

Minor or incremental Garbage Collection: It is said to have occurred when unreachable objects in the young generation heap memory are removed.

Major or Full Garbage Collection: It is said to have occurred when the objects that survived the minor garbage collection are copied into the old generation or permanent generation heap memory

are removed. When compared to the young generation, garbage collection happens less frequently in the old generation.

Important Concepts Related to Garbage Collection in Java

1. Unreachable objects: An object is said to be unreachable if it doesn't contain any reference to it. Also, note that objects which are part of the island of isolation are also unreachable.

```
Integer i = new Integer(4);

// the new Integer object is reachable  via the reference in 'i'

i = null;

// the Integer object is no longer reachable.
```

garbage collection

2. Eligibility for garbage collection: An object is said to be eligible for GC(garbage collection) if it is unreachable. After i = null, integer object 4 in the heap area is suitable for garbage collection in the above image.

Ways to make an object eligible for Garbage Collector

Even though the programmer is not responsible for destroying useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.

There are generally four ways to make an object eligible for garbage collection.

Nullifying the reference variable

Re-assigning the reference variable

An object created inside the method

Island of Isolation

Ways for requesting JVM to run Garbage Collector

Once we make an object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.

We can also request JVM to run Garbage Collector. There are two ways to do it :

Using System.gc() method: System class contain static method gc() for requesting JVM to run Garbage Collector.

Using Runtime.getRuntime().gc() method: Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

There is no guarantee that any of the above two methods will run Garbage Collector.

The call System.gc() is effectively equivalent to the call : Runtime.getRuntime().gc()

Finalization

Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.

finalize() method is present in Object class with the following prototype.

protected void finalize() throws Throwable

Based on our requirement, we can override finalize() method for performing our cleanup activities like closing connection from the database.

The finalize() method is called by Garbage Collector, not JVM. However, Garbage Collector is one of the modules of JVM.

Object class finalize() method has an empty implementation. Thus, it is recommended to override the finalize() method to dispose of system resources or perform other cleanups.

The finalize() method is never invoked more than once for any object.

If an uncaught exception is thrown by the finalize() method, the exception is ignored, and the finalization of that object terminates.

Advantages of Garbage Collection in Java

The advantages of Garbage Collection in Java are:

It makes java memory-efficient because the garbage collector removes the unreferenced objects from heap memory.

It is automatically done by the garbage collector(a part of JVM), so we don't need extra effort.

Real-World Example

Let's take a real-life example, where we use the concept of the garbage collector.

Question: Suppose you go for the internship at GeeksForGeeks, and you were told to write a program to count the number of employees working in the company(excluding interns). To make this program, you have to use the concept of a garbage collector.

This is the actual task you were given at the company:

Write a program to create a class called Employee having the following data members.

```
// Correct code to count number

// of employees excluding interns.

class Employee {

        private int ID;

        private String name;

        private int age;

        private static int nextId = 1;

        // it is made static because it

        // is keep common among all and
```

```
	// shared by all objects
	public Employee(String name, int age)
	{
		this.name = name;
		this.age = age;
		this.ID = nextId++;
	}
	public void show()
	{
		System.out.println("Id=" + ID + "\nName=" + name+ "\nAge=" + age);
	}
	public void showNextId()
	{
		System.out.println("Next employee id will be="+ nextId);
	}
	protected void finalize()
	{
		--nextId;
		// In this case,
		// gc will call finalize()
		// for 2 times for 2 objects.
	}
}
public class UseEmployee {
	public static void main(String[] args)
	{
		Employee E = new Employee("GFG1", 56);
		Employee F = new Employee("GFG2", 45);
		Employee G = new Employee("GFG3", 25);
		E.show();
		F.show();
```

```
                G.show();
                E.showNextId();
                F.showNextId();
                G.showNextId();
                {
                        // It is sub block to keep
                        // all those interns.
                        Employee X = new Employee("GFG4", 23);
                        Employee Y = new Employee("GFG5", 21);
                        X.show();
                        Y.show();
                        X.showNextId();
                        Y.showNextId();
                        X = Y = null;
                        System.gc();
                        System.runFinalization();
                }
                E.showNextId();
        }
}
```

3.a.

Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.

Method overloading is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding in Java. In Method overloading compared to parent argument, child argument will get the highest priority.

```
// Java program to demonstrate working of method
// overloading in Java
public class Sum {
        // Overloaded sum(). This sum takes two int parameters
```

```
    public int sum(int x, int y) { return (x + y); }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double
    // parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

In addition to overloading methods, we can also overload constructors in java. Overloaded constructor is called based upon the parameters specified when new is executed.

When do we need Constructor Overloading?

Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. For example, Thread class has 8 types of constructors. If we do not want to specify anything about a thread then we can simply use default constructor of Thread class, however if we need to specify thread name, then we may call the parameterized constructor of Thread class with a String args like this:

Thread t= new Thread (" MyThread ");

Let us take an example to understand need of constructor overloading. Consider the following implementation of a class Box with only one constructor taking three arguments.

```
// An example class to understand need of
// constructor overloading.
class Box
{
   double width, height,depth;
   // constructor used when all dimensions
   // specified
   Box(double w, double h, double d)
   {
      width = w;
      height = h;
      depth = d;
   }
   // compute and return volume
   double volume()
   {
      return width * height * depth;
   }
}
```

As we can see that the Box() constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box() constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since Box() requires three arguments, it's an error to call it without them. Suppose we simply wanted a box object without initial dimension, or want to initialize a cube by specifying only one value that would be used for all three dimensions. From the above implementation of Box class these options are not available to us.

These types of problems of different ways of initializing an object can be solved by constructor overloading. Below is the improved version of class Box with constructor overloading.

```
// Java program to illustrate Constructor Overloading
class Box
{
        double width, height, depth;
```

```
        // constructor used when all dimensions specified
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box()
        {
                width = height = depth = 0;
        }
        // constructor used when cube is created
        Box(double len)
        {
                width = height = depth = len;
        }
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}
// Driver code
public class Test
{
        public static void main(String args[])
        {
                // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
```

```
        Box mycube = new Box(7);
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);
        // get volume of cube
        vol = mycube.volume();
        System.out.println(" Volume of mycube is " + vol);
    }
}
```

3.b.i. Varargs - Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments. Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

Need of Java Varargs

Until JDK 4, we cannot declare a method with variable no. of arguments. If there is any change in the number of arguments, we must declare a new method. This approach increases the length of the code and reduces readability.

Before JDK 5, variable-length arguments could be handled in two ways. One uses an overloaded method (one for each), and another puts the arguments into an array and then passes this array to the method. Both are potentially error-prone and require more code.

To resolve these problems, Variable Arguments (Varargs) were introduced in JDK 5. From JDK 5 onwards, we can declare a method with a variable number of arguments. Such types of methods are called Varargs methods. The varargs feature offers a simpler, better option.

Syntax of Varargs

Internally, the Varargs method is implemented by using the single dimensions arrays concept. Hence, in the Varargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(…).

For Example,

```
public static void fun(int ... a)
{
```

```
  // method body
}
```

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here, a is implicitly declared as an array of type int.

3.b.ii. Final Keyword:

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

variable

method

class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}
class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

4.a. Whenever you inherit/extend a class, a copy of superclass's members is created in the subclass object and thus, using the subclass object you can access the members of both classes.

Example

In the following example we have a class named SuperClass with a method with name demo(). We are extending this class with another class (SubClass).

Now, you create an object of the subclass and call the method demo().

```
class SuperClass{
   public void demo() {
      System.out.println("demo method");
```

```
  }
}
public class SubClass extends SuperClass {
  public static void main(String args[]) {
    SubClass obj = new SubClass();
    obj.demo();
  }
}
```

Output

```
demo method
```

Super class's Constructor in inheritance

In inheritance constructors are not inherited. You need to call them explicitly using the super keyword.

If a Super class have parameterized constructor. You need to accept these parameters in the sub class's constructor and within it, you need to invoke the super class's constructor using "super()" as

```
public Student(String name, int age, String branch, int Student_id){
  super(name, age);
  this.branch = branch;
  this.Student_id = Student_id;
}
```

Example

Following java program demonstrates how to call a super class's constructor from the constructor of the sub class using the super keyword.

```
class Person{
  public String name;
  public int age;
  public Person(String name, int age){
    this.name = name;
    this.age = age;
  }
  public void displayPerson() {
    System.out.println("Data of the Person class: ");
```

```
    System.out.println("Name: "+this.name);
    System.out.println("Age: "+this.age);
  }
}
public class Student extends Person {
  public String branch;
  public int Student_id;
  public Student(String name, int age, String branch, int Student_id){
    super(name, age);
    this.branch = branch;
    this.Student_id = Student_id;
  }
  public void displayStudent() {
    System.out.println("Data of the Student class: ");
    System.out.println("Name: "+this.name);
    System.out.println("Age: "+this.age);
    System.out.println("Branch: "+this.branch);
    System.out.println("Student ID: "+this.Student_id);
  }
  public static void main(String[] args) throws CloneNotSupportedException {
    Person person = new Student("Krishna", 20, "IT", 1256);
    person.displayPerson();
  }
}
```

Output

```
Data of the Person class:
Name: Krishna
Age: 20
```

4.b. If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

Example of method overriding

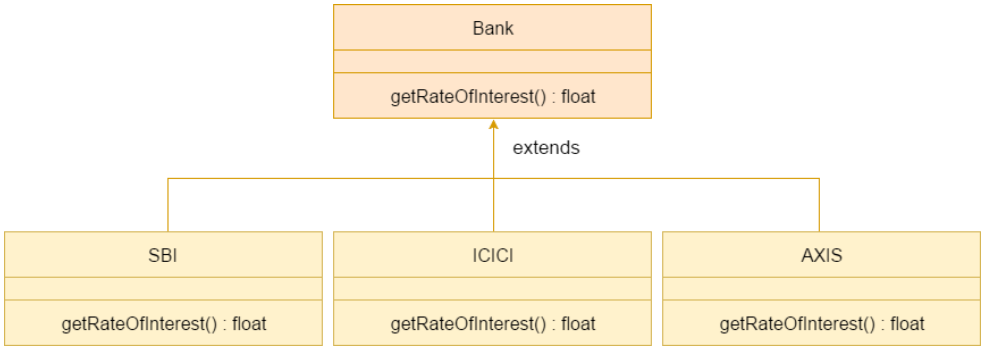
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding

//Creating a parent class.

class Vehicle{

  //defining a method

  void run(){System.out.println("Vehicle is running");}

}

//Creating a child class

class Bike2 extends Vehicle{

  //defining the same method as in the parent class

  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){

  Bike2 obj = new Bike2();//creating object

  obj.run();//calling method

  }

}
```

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example: Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.




```
class Bank{

float getRateOfInterest(){return 0;}}

class SBI extends Bank{

float getRateOfInterest(){return 8.4f;}}

class ICICI extends Bank{

float getRateOfInterest(){return 7.3f;}}

class AXIS extends Bank{

float getRateOfInterest(){return 9.7f;}}

class TestPolymorphism{

public static void main(String args[]){

Bank b;

b=new SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

b=new ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

b=new AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());

} }
```

5.a. 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. Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. Since Java 8, we can have default and static methods in an interface. Since Java 9, we can have private methods in an interface.

Benefits of Interface:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Example:

```
//Interface declaration: by first user

interface Drawable{

void draw();  }

//Implementation: by second user

class Rectangle implements Drawable{

public void draw(){System.out.println("drawing rectangle");}  }

class Circle implements Drawable{

public void draw(){System.out.println("drawing circle");}  }

//Using interface: by third user

class TestInterface1{

public static void main(String args[]){

Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()

d.draw();  }}
```

5.b.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc. The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error.

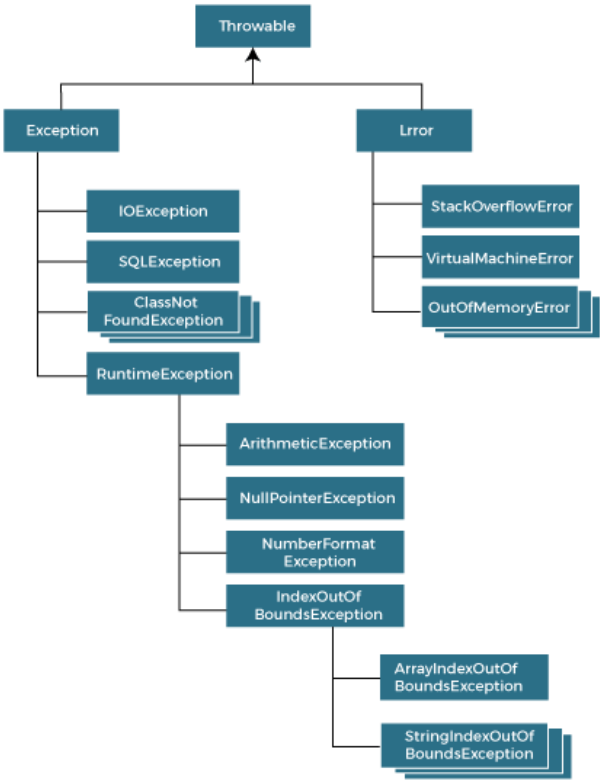
There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

Checked Exception

Unchecked Exception

Error

The hierarchy of Java Exception classes is given below:




Any good program makes use of a language's exception handling mechanisms. There is no better way to frustrate an end-user then by having them run into an issue with your software and displaying a big ugly error message on the screen, followed by a program crash. Exception handling is all about ensuring that when your program encounters an issue, it will continue to run and provide informative feedback to the end-user or program administrator. Any Java programmer becomes familiar with exception handling on day one, as some Java code will not even compile unless there is some form of exception handling put into place via the try-catch-finally syntax. Python has similar constructs to that of Java, and we will discuss them in this chapter.

After you have found an exception, or preferably before your software is distributed, you should go through the code and debug it in order to find and repair the erroneous code. There are many ways to debug and repair code; we will go through some debugging methodologies in this chapter. In Python as well as Java, the assert keyword can help tremendously in this area. We will cover assert in depth here and learn the different ways that it can be used to help you out and save time debugging those hard-to-find errors.

6.a. 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.

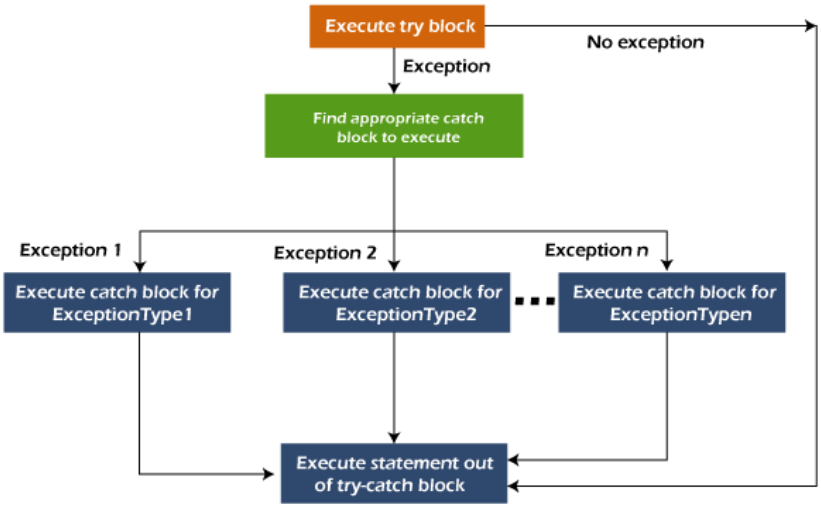
2) Java package provides access protection.

3) Java package removes naming collision.

Example:

```
//save by A.java

package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.*;


class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }

}
```

6.b. A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.




Example:

```
public class MultipleCatchBlock1 {
```

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


        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

7.a. In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the

same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Example:

```
// Java program to implement solution of producer consumer problem.

import java.util.LinkedList;

public class Threadexample {

	public static void main(String[] args)

		throws InterruptedException

	{

		// Object of a class that has both produce() and consume() methods

		final PC pc = new PC();

		// Create producer thread

		Thread t1 = new Thread(new Runnable() {

			@Override

			public void run()

			{

				try {

					pc.produce();

				}

				catch (InterruptedException e) {

					e.printStackTrace();

				}

			}

		});

		// Create consumer thread

		Thread t2 = new Thread(new Runnable() {

			@Override

			public void run()

			{

				try {
```

```
                        pc.consume();
                    }
                    catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            // Start both threads
            t1.start();
            t2.start();
            // t1 finishes before t2
            t1.join();
            t2.join();
}
// This class has a list, producer (adds items to list
// and consumer (removes items).
public static class PC {
            // Create a list shared by producer and consumer
            // Size of list is 2.
            LinkedList<Integer> list = new LinkedList<>();
            int capacity = 2;
            // Function called by producer thread
            public void produce() throws InterruptedException
            {
                int value = 0;
                while (true) {
                    synchronized (this)
                    {
                        // producer thread waits while list
                        // is full
                        while (list.size() == capacity)
```

```
                    wait();
                System.out.println("Producer produced-"+ value);
                // to insert the jobs in the list
                list.add(value++);
                // notifies the consumer thread that
                // now it can start consuming
                notify();
                // makes the working of program easier
                // to understand
                Thread.sleep(1000);
            }
        }
}
// Function called by consumer thread
public void consume() throws InterruptedException
{
        while (true) {
            synchronized (this)
            {
                // consumer thread waits while list
                // is empty
                while (list.size() == 0)
                    wait();
                // to retrieve the first job in the list
                int val = list.removeFirst();
                System.out.println("Consumer consumed-"+ val);
                // Wake up producer thread
                notify();
                // and sleep
                Thread.sleep(1000);
            }
```

```
                }
            }
        }
}
```

7.b. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process. Threads can be created by using two mechanisms :

Extending the Thread class and Implementing the Runnable Interface.

Thread Class vs Runnable Interface

If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Using runnable will give you an object that can be shared amongst multiple threads.

Example:

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}
```

```
// Main Class
class Multithread {
	public static void main(String[] args)
	{
		int n = 8; // Number of threads
		for (int i = 0; i < n; i++) {
			Thread object
				= new Thread(new MultithreadingDemo());
			object.start();
		}
	}
}
```

8.a. The Enum in Java is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class.

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

```
class EnumExample{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
public static void main(String args[]){
Day day=Day.MONDAY;
switch(day){
case SUNDAY:
 System.out.println("sunday");
 break;
case MONDAY:
 System.out.println("monday");
 break;
default:
System.out.println("other day");
}
}}
```

8.b.i. All enumerations automatically contain two predefined methods: values() and valueOf(). Their general forms are:

```
public static enum-type[ ] values()
public static enum-type valueOf(String str)
```

The values() method returns an array that contains a list of the enumeration constants. The valueOf() method returns the enumeration constant whose value corresponds to the string passed in str. In both cases, enum-type is the type of the enumeration.

The following program demonstrates the values() and valueOf() methods:

```
enum Direction {// j a v a 2 s . c o m
  East, South, West, North
}
public class Main {
  public static void main(String args[]) {
    Direction dir;
    // use values()
    Direction all[] = Direction.values();
    for (Direction a : all)
```

```
    System.out.println(a);
   System.out.println();
   // use valueOf()
   dir = Direction.valueOf("South");
   System.out.println(dir);
  }
}
```

8.b.ii. The Java String class compareTo() method compares the given string with the current string lexicographically. It returns a positive number, negative number, or 0.

It compares strings on the basis of the Unicode value of each character in the strings.

If the first string is lexicographically greater than the second string, it returns a positive number (difference of character value). If the first string is less than the second string lexicographically, it returns a negative number, and if the first string is lexicographically equal to the second string, it returns 0.

if s1 > s2, it returns positive number

if s1 < s2, it returns negative number

if s1 == s2, it returns 0

Example:

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
}}
```

9.a. A Class URL represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more

complicated object, such as a query to a database or to a search engine. Constructors of the URL class

URL(String address) throws MalformedURLException: It creates a URL object from the specified String.

URL(String protocol, String host, String file): Creates a URL object from the specified protocol, host, and file name.

URL(String protocol, String host, int port, String file): Creates a URL object from protocol, host, port, and file name.

URL(URL context, String spec): Creates a URL object by parsing the given spec in the given context.

URL(String protocol, String host, int port, String file, URLStreamHandler handler):

Creates a URL object from the specified protocol, host, port number, file, and handler.

URL(URL context, String spec, URLStreamHandler handler):

Creates a URL by parsing the given spec with the specified handler within a specified context.

Example:

```
// Java program to demonstrate working of URL


// Importing required classes
import java.net.MalformedURLException;
import java.net.URL;


// Main class
// URL class
public class GFG {


    // Main driver method
    public static void main(String[] args)
            throws MalformedURLException
    {


        // Creating a URL with string representation
        URL url1 = new URL(
                "https://www.google.co.in/?gfe_rd=cr&ei=ptYq"
```

```
            + "WK26I4fT8gfth6CACg#q=geeks+for+geeks+java");
    // Creating a URL with a protocol,hostname,and path
    URL url2 = new URL("http", "www.geeksforgeeks.org",
                                        "/jvm-works-jvm-architecture/");
    URL url3 = new URL(
            "https://www.google.co.in/search?"
            + "q=gnu&rlz=1C1CHZL_enIN71"
            + "4IN715&oq=gnu&aqs=chrome..69i57j6"
            + "9i60l5.653j0j7&sourceid=chrome&ie=UTF"
            + "-8#q=geeks+for+geeks+java");
    // Printing the string representation of the URL
    System.out.println(url1.toString());
    System.out.println(url2.toString());
    System.out.println();
    System.out.println(
            "Different components of the URL3-");
    // Retrieving the protocol for the URL
    System.out.println("Protocol:- "
                                        + url3.getProtocol());
    // Retrieving the hostname of the url
    System.out.println("Hostname:- " + url3.getHost());
    // Retrieving the default port
    System.out.println("Default port:- "
                                        + url3.getDefaultPort());

    // Retrieving the query part of URL
    System.out.println("Query:- " + url3.getQuery());
    // Retrieving the path of URL
    System.out.println("Path:- " + url3.getPath());
    // Retrieving the file name
    System.out.println("File:- " + url3.getFile());
```

```
            // Retrieving the reference
            System.out.println("Reference:- " + url3.getRef());
        }
}
```

9.b. The Java HttpURLConnection class is http specific URLConnection. It works for HTTP protocol only. By the help of HttpURLConnection class, you can retrieve information of any HTTP URL such as header information, status code, response code etc. The java.net.HttpURLConnection is subclass of URLConnection class. HttpURLConnection Class Constructor

| Constructor | Description |
|---|---|
| protected HttpURLConnection(URL u) | It constructs the instance of HttpURLConnection class. |

Java HttpURLConnection Methods

| | |
|---|---|
| void disconnect() | It shows that other requests from the server are unlikely in the near future. |
| InputStream getErrorStream() | It returns the error stream if the connection failed but the server sent useful data. |
| Static boolean getFollowRedirects() | It returns a boolean value to check whether or not HTTP redirects should be automatically followed. |
| String getHeaderField(int n) | It returns the value of nth header file. |
| long getHeaderFieldDate(String name, long Default) | It returns the value of the named field parsed as a date. |
| String getHeaderFieldKey(int n) | It returns the key for the nth header file. |
| boolean getInstanceFollowRedirects() | It returns the value of HttpURLConnection's instance FollowRedirects field. |
| Permission getPermission() | It returns the SocketPermission object representing the permission to connect to the destination host and port. |
| String getRequestMethod() | It gets the request method. |
| int getResponseCode() | It gets the response code from an HTTP response message. |
| String getResponseMessage() | It gets the response message sent along with the response code from a server. |
| void setChunkedStreamingMode(int chunklen) | The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is not known in advance. |
| void setFixedLengthStreamingMode(int contentlength) | The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is known in advance. |
| void setFixedLengthStreamingMode(long contentlength) | The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is not known in advance. |

static void setFollowRedirects(boolean set) It sets whether HTTP redirects (requests with response code) should be automatically followed by HttpURLConnection class.

void setInstanceFollowRedirects(boolean followRedirects) It sets whether HTTP redirects (requests with response code) should be automatically followed by instance of HttpURLConnection class.

void setRequestMethod(String method) Sets the method for the URL request, one of: GET POST HEAD OPTIONS PUT DELETE TRACE are legal, subject to protocol restrictions.

abstract boolean usingProxy() It shows if the connection is going through a proxy.

10.a.

add(E e) Ensures that this collection contains the specified element (optional operation).

addAll(Collection<? extends E> c) Adds all the elements in the specified collection to this collection (optional operation).

clear() Removes all the elements from this collection (optional operation).

contains(Object o) Returns true if this collection contains the specified element.

containsAll(Collection<?> c) Returns true if this collection contains all the elements in the specified collection.

equals(Object o) Compares the specified object with this collection for equality.

hashCode() Returns the hash code value for this collection.

isEmpty() Returns true if this collection contains no elements.

iterator() Returns an iterator over the elements in this collection.

parallelStream()Returns a possibly parallel Stream with this collection as its source.

remove(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).

removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).

removeIf(Predicate<? super E> filter) Removes all the elements of this collection that satisfy the given predicate.

retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).

size() Returns the number of elements in this collection.

spliterator() Creates a Spliterator over the elements in this collection.

stream() Returns a sequential Stream with this collection as its source.

toArray() Returns an array containing all the elements in this collection.

toArray(IntFunction<T[]> generator) Returns an array containing all the elements in this collection, using the provided generator function to allocate the returned array.

toArray(T[] a)  Returns an array containing all the elements in this collection; the runtime type of the returned array is that of the specified array.

10.b.

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
  for(String fruit:list1)
    System.out.println(fruit);
 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
  list2.add(21);
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
  Collections.sort(list2);
   //Traversing list through the for-each loop
  for(Integer number:list2)
    System.out.println(number);
 }
}
```