# Important questions and Answers

| Q. NO | Questions and Answers (2Marks Questions) |
|---|---|
| 1 | **Define class and object?**<br>**Answer:** class is a template for an object, and an object is an instance of a class.<br>Syntax to define class: class class_name { -members of the class }<br>Example: class first {int x,y,z;}<br>Syntax to define object: class_name object_name =new class_name();<br>Example: object obj=new object(); |
| 2 | **Define inheritance?**<br>**Answer:** Acquiring the properties of one class into another class is known as inheritance. Or Deriving one class from another class in known as inheritance.<br>I. In Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass.<br>II. We used a keyword extends to inherit a class from another class. |
| 3 | **Define polymorphism?**<br>**Answer:** polymorphism is a Greek word meaning is many forms or Behaving different ways based on given input is known as polymorphism. |
| 4 | **Write a program to print "welcome" followed by your name and "How are you"?**<br>**Answer:**<br>`class printname`<br>`{`<br>`    public static void main(String[] args)`<br>`    {`<br>`        String name="Rakesh Kumar";`<br>`        System.out.print("Welcome "+name+" How are you?");`<br>`    }`<br>`}` |
| 5 | **How many access specifiers are available in java language?**<br>**Answer:** There are totally four access specifiers are available in java language. i.e<br>I. Private<br>II. Default<br>III. Protected<br>IV. Public |
| 6 | **What is the use of final keyword?**<br>**Answer:** final keyword can be used in three ways. i.e<br>I. To define constant: If we used final keyword while defining a variable that variable value can't be modified in the program.<br>II. To prevent overriding: If we used final keyword while declaring methods that method can't be overridden.<br>III. To prevent inheritance: If we used final keyword while declaring class that class can't be inherited. |
| 7 | **What was the name of first version of java?**<br>**Answer:** Java Development Kit (JDK) Alpha and Beta |
| 8 | **Can we overload main() method? Explain with an example.**<br>**Answer:** Yes, we can do overload main() method.<br>`class overloadmain`<br>`{`<br>`    public void main()`<br>`    {`<br>`        System.out.println("main method was overloded.");`<br>`    }`<br>`    public static void main(String[] args)`<br>`    {`<br>`        overloadmain om=new overloadmain();`<br>`        om.main();`<br>`    }`<br>`}` |
| 9 | **Explain about finalize method in java.** |

**Answer:** Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization.

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class.
- The **finalize( )** method has this general form:
- **Syntax:**

```
protected void finalize( )
{
// finalization code here
}
```

10 **What is the name of the tool that is used for compiling a java program?**

**Answer:** javac – java programming language compiler
Or JDK – java Development Kit Tool

11 **List out the methods available in String class?**

**Answer:** List of String class methods

I. equals()
II. equalsIgnoreCase()
III. compareTo()
IV. compareToIgnoreCase()
V. startsWith()
VI. endsWith()
VII. contains()
VIII. getBytes()

12 **Differentiate between this and super keywords with an example?**

**Answer:** super keyword can be used to refer parent class member from child class.
Example:

```
class A
{
        int x;
        void show(int a)
        {
                System.out.println("Square value is: "+(a*a));
        }
}

class superandthis extends A
{
        int y;
        superandthis(int x,int y)
        {
                super.x=x;
                this.y=y;
                this.display(this.y);
        }
        void display(int a)
        {
                super.show(x);
                System.out.println("Cube value is: "+(a*a*a));
        }
        public static void main(String[] args)
        {
                superandthis st=new superandthis(5,6);
        }
}
```

| | |
|---|---|
| 13 | **What is meant by constructor?**<br>**Answer:** constructor in java is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provide data for the object that's why it's known as constructor. |
| 14 | **What is the difference between single and multilevel inheritance?**<br>**Answer:**<br>**Single-Inheritance:** Acquiring the properties of one class into another class in known as single inheritance.<br>Syntax: class class_name1 { -members of class }<br>Class class_name2 extends class_name1 { -members of class along with members of class_name1 }<br>**Multilevel-Inheritance:** Acquiring the properties of one class into second class and acquiring second class properties into third class and so on in known as single inheritance.<br>Syntax: class class_name1 { -members of class }<br>Class class_name2 extends class_name1 { -members of class along with members of class_name1 }<br>Class class_name3 extends class_name2 { -members of class along with members of class_name1 and class_name2 } …. So on. |
| 15 | **List out OOP principles?**<br>**Answer:** list of OOP principles<br>I. Encapsulation<br>II. Inheritance<br>III. Polymorphism |

| **Q. No** | **Questions and Answers (8Marks Questions)** |
|---|---|
| 1 | **Explain why java known as a platform-independent and architectural neutral language?**<br>**Answer:**<br>**Architectural Neutral:**<br>➢ A central issue for the Java designers was that of code longevity and portability.<br>➢ At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine.<br>➢ Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.<br>➢ The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.<br>**Platform-independent:**<br>➢ Running an applications in any operating system is known as platform – independent<br>➢ Whenever you compile java program by using java compiler it generated .class file if your code doesn't contain any errors.<br>➢ If you execute that .class file in any operating system that can be executed but for executing that operating system should consist Java Virtual Machine in it.<br>➢ So, java .class file can run any operating system that's why we can call java is a platform – independent language. |
| 2 | **Write a program that takes number of hours worked by an employee and basic hourly pay, and outputs the total pay due?**<br>**Answer:**<br><pre>class employee_pay<br>{<br>        final double pay=250.00;<br>        double hours;<br>        employee_pay(double hours)<br>        {<br>                this.hours=hours;<br>        }</pre> |

```
	public double due_pay()
	{
		return hours*pay;
	}
	public static void main(String... args)
	{
		employee_pay ep=new employee_pay(Double.parseDouble(args[0]));
		System.out.println("Payment due is: "+ ep.due_pay());
	}
}
```

3 **Create a class Rectangle .The class has two attributes, length and width, each of which defaults to 0.It has methods that calculate the perimeter and area of the rectangle .It has set and get methods for both length and width .The set method should verify that length and width are floating point numbers larger than 0.0 and less than 20.0.**

**Answer:**

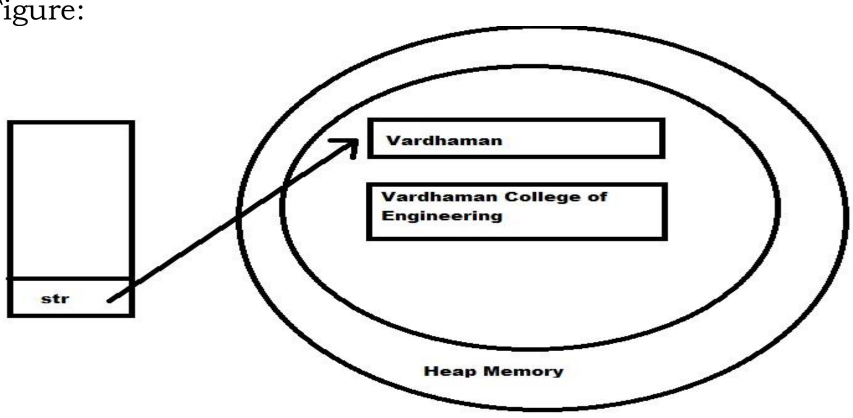
```
class rectangle
{
	static double width;
	static double height;
	public void set(double w,double h)
	{
		System.out.println(w+" "+h);
		if((w<0.0)||(w>20.0)||(h<0.0)||(h>20.0))
		{
			System.out.println("Your value should be greater than 0.0 and less
than 20.0");
			System.exit(0);
		}
		else
		{
			this.width=w;
			this.height=h;
		}
	}
	public double getwidth()
	{
		return this.width;
	}
	public double getheight()
	{
		return this.height;
	}
	public double area()
	{
		return getwidth()*getheight();
	}
	public double perimeter()
	{
		return 2*(getwidth()+getheight());
	}
	public static void main(String[] args)
	{
		rectangle r=new rectangle();
		r.set(Double.parseDouble(args[0]),Double.parseDouble(args[1]));
		System.out.println("Area of rectangle is: "+r.area());
		System.out.println("Perimeter of rectangle is: "+r.perimeter());
	}
}
```

4 **Why Strings are immutable? Explain with suitable example.**
**Answer:**

- In java, String objects are immutable means unmodifiable or unchangeable.
- Once String object is created its data or state can't be changed but a new String object is created.
- Example:

```
class testStr
{
    public static void main(String[] args)
    {
            String str="Vardhaman";
            str.concat(" College of Engineering");
            System.out.println(str);
    }
}
```

- If we execute above program we'll get output as **Vardhaman** but not Vardhaman College of Engineering. Here, concat() method appends the string at the end.
- Here, Vardhaman is not changed to Vardhaman College of Engineering but a new object is created with Vardhaman College of Engineering.
- Figure:



- If you see the figure that two objects are created but **str** reference variable still refer to Vardhaman but not to Vardhaman College of Engineering.
- But, if we explicitly assign it to the reference variable. It will refer to Vardhaman College of Engineering.
- **For example:**

```
class testStr
{
    public static void main(String[] args)
    {
            String str="Vardhaman";
            str=str.concat(" College of Engineering");
            System.out.println(str);
    }
}
```

5 **What is mean by runtime polymorphism explain with an example program?**
**Answer:**

- Runtime polymorphism is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not

the type of the reference variable) that determines which version of an overridden method will be executed.

- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.
- Here is an example that illustrates runtime polymorphism:

```
class A
{
    public void display()
    {
            System.out.println("A class display method.");
    }
}
class B extends A
{
    public void display()
    {
            System.out.println("B class display method.");
    }
}
class C extends B
{
    public void display()
    {
            System.out.println("C class display method.");
    }
}
class D
{
    public static void main(String[] ar)
    {
            A a=new A();
            B b=new B();
            C c=new C();
            A x; //defining super class reference
            x=a;
            x.display();
            x=b;
            x.display();
            x=c;
            x.display();
    }
}
```

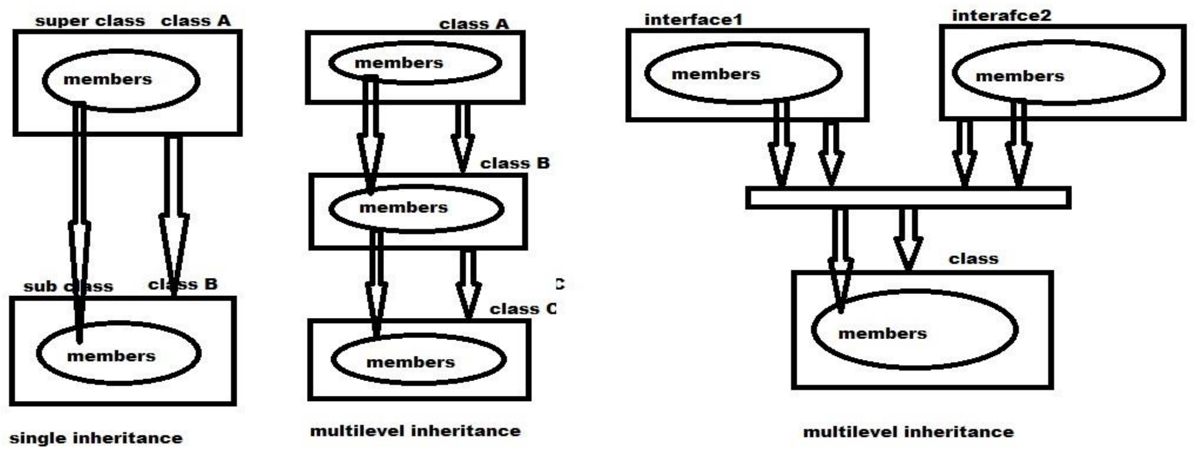
6 **Write the difference between abstract class and interface?**

**Answer:**

| **S. No** | **Abstract Class** | **Interface** |
|---|---|---|
| 1 | Abstract class can have abstract and non-abstract methods | Interface can have only abstract methods (Only method definition but not implementation) |
| 2 | Abstract class doesn't support multiple inheritance | Interface does support multiple inheritance |
| 3 | In abstract class we can define all types of variables | Interface has only static and final variables |
| 4 | In abstract class we cannot declare abstract<br>constructors, or abstract static methods | Interface can't have static methods and main method or constructor. |
| 5 | Abstract class provide the implementation of interfaces | Interface can't provide the implementation of abstract class |

| | | |
|---|---|---|
| 6 | The abstract keyword is used to define abstract class | The interface keyword is used to define interfaces |
| 7 | Syntax: abstract class class_name<br>{<br>-abstract methods and non-abstract methods<br>-variables<br>} | Syntax: interface interface_name<br>{<br>-final and static variables<br>-methods which contain only definition<br>} |

7 **Explain types of inheritance with the help of diagrams?**
**Answer:**



- Java supports mainly two types of inheritance but it supports another interface by using interfaces. List
  - I. Single inheritance
  - II. Multilevel inheritance
  - III. Multiple inheritance (Only with interfaces)

**Single-Inheritance:** Acquiring the properties of one class into another class in known as single inheritance.

- Syntax: class class_name1 { -members of class }
  class class_name2 extends class_name1 { -members of class along with members of class_name1 }
- Example:

```
class first
{
        void test1()
        {
                System.out.println("Test1 method from parent");
        }
        void test2()
        {
                System.out.println("Test2 method from parent");
        }
}

class second extends first
{
        void test3()
        {
                System.out.println("Test3 method from child");
        }
        void test4()
        {
                System.out.println("Test4 method from child");
        }
        public static void main(String... args)
        {
                second d=new second ();
                d.test1();
                d.test2();
                d.test3();
                d.test4();
        }
}
```
**Multilevel-Inheritance:** Acquiring the properties of one class into second class and acquiring second class properties into third class and so on in known as single inheritance.

Syntax: class class_name1 { -members of class }
Class class_name2 extends class_name1 { -members of class along with members of class_name1 }

- Class class_name3 extends class_name2 { -members of class along with members of class_name1 and class_name2 } …. So on.
- Example:

```
class first
{
    void test1()
    {
            System.out.println("Test1 method from parent");
    }
    void test2()
    {
            System.out.println("Test2 method from parent");
    }
}
class second extends first
{
    void test3()
    {
            System.out.println("Test3 method from child");
    }
    void test4()
    {
            System.out.println("Test4 method from child");
    }
    public static void main(String... args)
    {
            second d=new second();
            d.test1();
            d.test2();
            d.test3();
            d.test4();
    }
}

class third extends second
{
    void test5()
    {
            System.out.println("Test5 method from grand child");
    }
    void test6()
    {
            System.out.println("Test2 method from grand child");
    }
    public static void main(String... args)
```

```
        {
                third t=new third();
                t.test1();
                t.test2();
                t.test3();
                t.test4();
                t.test5();
                t.test6();
                first f=t;
                f.test1();
                f.test2();
                second s=t;
                s.test1();
                s.test2();
                s.test3();
                s.test4();
        }

    }
```

**Multiple Inheritance:** Acquiring the properties of more than one interface into another class is known as multiple inheritance.

**Note:** multiple inheritance is not possible through classes but it can be possible by using interfaces. It can be possible by using one class and one interfaces or more but class count should be one.

**Example:**

```
interface impinter1
{
        void test1();
        void test2();
}
interface impinter2
{
        void test3();
        void test4();
}
class test implements impinter1,impinter2
{
        public void test1()
        {
                System.out.println("Hello from test1");
        }
        public void test2()
        {
                System.out.println("Hello from test2");
        }
        public void test3()
        {
                System.out.println("Hello from test3");
        }
        public void test4()
        {
                System.out.println("Hello from test4");
        }
        public static void main(String[] args)
        {
                impinter1 i1=new test();
                i1.test1();
                i1.test2();
                impinter1 i2=new test();
```

```
                    i2.test1();
                    i2.test2();
            }
    }
```

8

**What is method overriding? Can methods be override? Write a program in support of your answer?**

**Answer:**

- When a method in a subclass has the same name and type signature as a method in its superclass is known as method overriding.
- Yes, methods can be override. Here is the procedure.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.
- Consider the following Example:

```
class parent
{
    void test()
    {
            System.out.println("Test method from parent class.");
    }
}
class child extends parent
{
    void test()
    {
            System.out.println("Test method from child class.");
    }

    public static void main(String[] args)
    {
            child c=new child();
            c.test();
    }
}
```

- In the above program we defined test() method in both the classes. By defining object of child class and call test() method by using child class object it will be called child class test() method.
- If you doesn’t define test() method inside the child class, if you call test() method by using child class object it will be called parent class test() method.

9

**What is method overloading? Explain it with suitable example program?**

**Answer:** Defining a method with same name but different parameters within a class or between parent and child class is known as method overloading.

- So, here that method will behave in different ways based on a given input.
- Example:

```
class parent
{
    void test()
    {
            System.out.println("Test method from parent class.");
    }
}
class child extends parent
{
    void test(String str)
    {
            System.out.println("Hello "+str);
    }
    void test1(int x)
```

```
        {
                System.out.println("Square is: "+(x*x));
        }
        void test1(int x,int y)
        {
                System.out.println("Addition is: "+(x+y));
        }
        public static void main(String[] args)
        {
                child c=new child();
                c.test();
                c.test("Rajesh");
                c.test1(10);
                c.test1(100,200);
        }
    }
```

10 **Explain Data types in java?**

**Answer:** Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

I. **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
II. **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
III. **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
IV. **Boolean:** This group includes boolean, which is a special type for representing true/false values.

**Integers:**
**byte:**
- The smallest integer type is byte.
- This is a signed 8-bit type that has a range from –128 to 127.
- Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the byte keyword.
- For example, the following declares two byte variables called b and c:
  byte b, c;

**short:**
- short is a signed 16-bit type.
- It has a range from –32,768 to 32,767.
- It is probably the least used Java type.
- Here are some examples of short variable declarations
  short s;
  short t;

**int:**
- The most commonly used integer type is int.
- It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated
- int is often the best choice when an integer is needed.
- Example: int x,y;

**Long:**
- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.
- The range of a long is quite large. This makes it useful when big, whole numbers are needed.
- Example: long x,y;

**Floating-point numbers:**
**float:**
- The type float specifies a single-precision value that uses 32 bits of storage.
- Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small.
- Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.
- Float can be useful when representing dollars and cents.
- Here are some example float variable declarations:
  float hightemp, lowtemp;

**double:**
- Double precision, as denoted by the double keyword, uses 64 bits to store a value.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.
- When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.
- Example: double x,y;

**Characters:**
**Char:**
- In Java, the data type used to store characters is char.
- char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type.
- The range of a char is 0 to 65,536.
- There are no negative chars.
- Example: char ch;

**Booleans:**
**Boolean:**
- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of a < b.
- Boolean is also the type required by the conditional expressions that govern the control statements such as if and for.
- Example: boolean b;

11 **List out and explain the methods of StringBuffer class in java (at-least 4 methods).**
**Answer:** List of methods available in StringBuffer class.

I. length() and capacity()
II. ensureCapacity()
III. charAt and setCharAt()
IV. getChars()
V. append()
VI. insert()
VII. reverse()
VIII. delete() nad deleteCharAt()
IX. replace()
X. substring()

**length( ) and capacity( )**

- The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.
- They have the following general forms:
  int length( )
  int capacity( )

**ensureCapacity( )**

- If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
- This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
- ensureCapacity( ) has this general form:
  void ensureCapacity(int minCapacity)

**charAt( ) and setCharAt( )**

- The value of a single character can be obtained from a StringBuffer via the charAt( ) method.
- You can set the value of a character within a StringBuffer using setCharAt( ). Their general forms are shown here:
  char charAt(int where)
  void setCharAt(int where, char ch)

**getChars( )**

- To copy a substring of a StringBuffer into an array, use the getChars( ) method.
- It has this general form:
  void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

**append( )**

- The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.
- It has several overloaded versions. Here are a few of its forms:
  StringBuffer append(String str)
  StringBuffer append(int num)
  StringBuffer append(Object obj)

**insert( )**

- The insert( ) method inserts one string into another.
- It is overloaded to accept values of all the primitive types, plus Strings, Objects, and CharSequences. Like append( ), it obtains the string representation of the value it is called with.
- This string is then inserted into the invoking StringBuffer object. These are a few of its forms:
  StringBuffer insert(int index, String str)
  StringBuffer insert(int index, char ch)
  StringBuffer insert(int index, Object obj)

**reverse( )**

- You can reverse the characters within a StringBuffer object using reverse( ), shown here:
  StringBuffer reverse( )

**delete( ) and deleteCharAt( )**

- You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).
- These methods are shown here:
  StringBuffer delete(int startIndex, int endIndex)
  StringBuffer deleteCharAt(int loc)

**replace( )**

- You can replace one set of characters with another set inside a StringBuffer object by calling replace( ).
- Its signature is shown here:
  StringBuffer replace(int startIndex, int endIndex, String str)

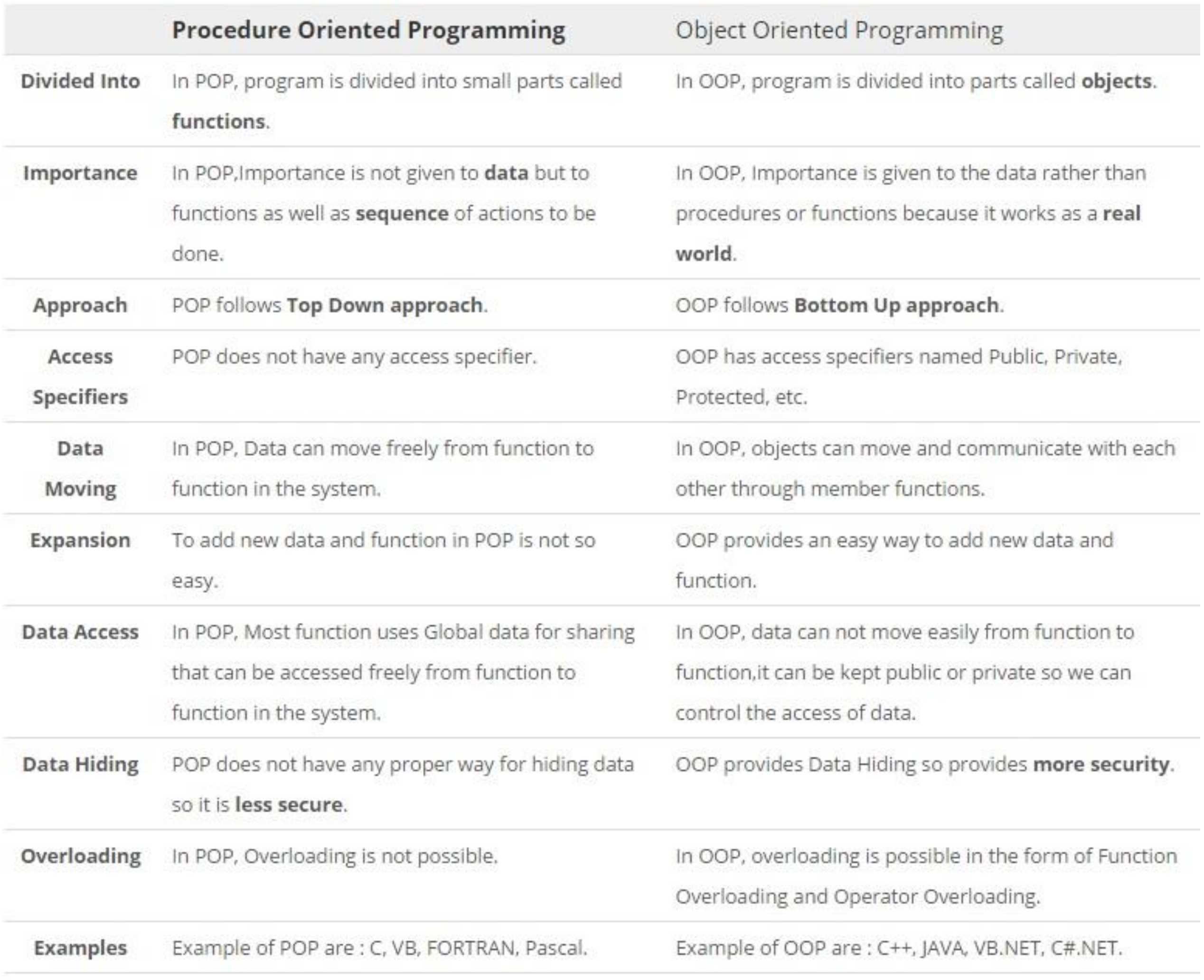
**substring( )**

- You can obtain a portion of a StringBuffer by calling substring( ).
- It has the following two forms:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

12 **Differentiate between procedure oriented programming and object oriented programming?**

**Answer:**



| | Procedure Oriented Programming | Object Oriented Programming |
|---|---|---|
| **Divided Into** | In POP, program is divided into small parts called **functions**. | In OOP, program is divided into parts called **objects**. |
| **Importance** | In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done. | In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**. |
| **Approach** | POP follows **Top Down approach**. | OOP follows **Bottom Up approach**. |
| **Access Specifiers** | POP does not have any access specifier. | OOP has access specifiers named Public, Private, Protected, etc. |
| **Data Moving** | In POP, Data can move freely from function to function in the system. | In OOP, objects can move and communicate with each other through member functions. |
| **Expansion** | To add new data and function in POP is not so easy. | OOP provides an easy way to add new data and function. |
| **Data Access** | In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system. | In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data. |
| **Data Hiding** | POP does not have any proper way for hiding data so it is **less secure**. | OOP provides Data Hiding so provides **more security**. |
| **Overloading** | In POP, Overloading is not possible. | In OOP, overloading is possible in the form of Function Overloading and Operator Overloading. |
| **Examples** | Example of POP are : C, VB, FORTRAN, Pascal. | Example of OOP are : C++, JAVA, VB.NET, C#.NET. |

13 **Explain in detailed about arrays in java language?**

**Answer:**

- **Arrays:** Collection of similar data elements with a single name is known as arrays.
- Java supports two types of arrays. i.e.
  - I. Single – dimensional Arrays
  - II. Multi- dimensional Arrays
- **Single – dimensional Arrays:**
- **Syntax:**

```
datatype[] var_name;
var_name=new datatype[size]; or
datatype[] var_name=new datatype[size];
```

**Example:**

```
import java.util.*;
class sdArray
{
        public static void main(String... args)
        {
                int[] arr;
                int i;
                arr=new int[10];
                Scanner sc=new Scanner(System.in);
                System.out.println("Enter elemenst into an array:");
                //Reading values into an array
                for(i=0;i<10;i++)
                {
                        System.out.print("Enter "+(i+1)+" element:");
                        arr[i]=sc.nextInt();
                }
                //Printing array elements by using length property
                for(i=0;i<arr.length;i++)
                {
                        System.out.print(arr[i]+" ");
                }
                //printing array elements by using for each loop
                for(int k:brr)
                        System.out.print(k+" ");
        }
}
```

- **Multi- dimensional Arrays:** Java supports two types of multi – dimensional arrays. i.e.
  I. Two – dimensional arrays
  II. Jagged / Ragged arrays
- **Two –dimensional arrays:**
- **Syntax:**

```
datatype[][] var_name;
Var_name = new datatype[rowsizw][columnsize]; OR
datatype[][] var_name = new datatype[rowsizw][columnsize];
```

- **Example:**

```
import java.util.*;
class tdArray
{
        public static void main(String[] args)
        {
                int[][] arr=new int[3][];
                arr[0]=new int[3];
                arr[1]=new int[4];
                arr[2]=new int[5];
                int i,j;
                Scanner sc=new Scanner(System.in);
                for(i=0;i<arr.length;i++)
                {
                        for(j=0;j<arr[i].length;j++)
                        {
                                arr[i][j]=sc.nextInt();
                        }
                }
                Arrays.sort(arr[0]);
                for(i=0;i<arr.length;i++)
                {
                        for(j=0;j<arr[i].length;j++)
                        {
                                System.out.print(arr[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
```

- **Jagged / Ragged arrays:**
- **Syntax:**

```
datatype[][] var_name;
```

```
var_name=new datatype[rowsize][];  OR
datatype[][] var_name=new datatype[rowsize][];

var_name[0]=new datatype[columnsize];
var_name[1]=new datatype[columnsize];
var_name[2]=new datatype[columnsize];
var_name[rowsize-2]=new datatype[columnsize];
var_name[rowsize-1]=new datatype[columnsize];
```

- **Example:**

```
import java.util.*;
class jdarray
{
    public static void main(String... args)
    {
            Scanner sc=new Scanner(System.in);
            int[][] arr=new int[5][];
            arr[0]=new int[6];
            arr[1]=new int[4];
            arr[2]=new int[3];
            arr[3]=new int[5];
            arr[4]=new int[2];
            //System.out.println("Enter elements:");
            for(int i=0;i<arr.length;i++)
            {
                    System.out.print("Enter elements into "+(i+1)+" array:");
                    for(int j=0;j<arr[i].length;j++)
                    {
                            arr[i][j]=sc.nextInt();
                    }
            }
            for(int i=0;i<arr.length;i++)
            {
                    for(int j=0;j<arr[i].length;j++)
                    {
                            System.out.print(arr[i][j]+" ");
                    }
                    System.out.println();
            }
            for(int k[]:arr)
                    for(int l:k)
                            System.out.print(l+" ");
    }
}
```

14 **Explain in detailed about the different operators available in java language with an examples?**

**Answer:** Most of its operators can be divided into the following four groups:

I. arithmetic
II. bitwise
III. relational
IV. logical

- **Arithmetic Operators:**

| Operator | Result |
|---|---|
| + | Addition (also unary plus) |
| – | Subtraction (also unary minus) |
| * | Multiplication |
| / | Division |
| % | Modulus |
| ++ | Increment |
| += | Addition assignment |
| – = | Subtraction assignment |
| *= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulus assignment |
| – – | Decrement |

➢ **Bitwise Operators:**

| Operator | Result |
|---|---|
| ~ | Bitwise unary NOT |
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |
| >> | Shift right |
| >>> | Shift right zero fill |
| << | Shift left |
| &= | Bitwise AND assignment |
| \|= | Bitwise OR assignment |
| ^= | Bitwise exclusive OR assignment |
| >>= | Shift right assignment |
| >>>= | Shift right zero fill assignment |
| <<= | Shift left assignment |

➢ **Relational Operators:**

| Operator | Result |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

➢ **Logical Operators:**

| Operator | Result |
|---|---|
| & | Logical AND |
| \| | Logical OR |
| ^ | Logical XOR (exclusive OR) |
| \|\| | Short-circuit OR |
| && | Short-circuit AND |
| ! | Logical unary NOT |
| &= | AND assignment |
| \|= | OR assignment |
| ^= | XOR assignment |
| == | Equal to |
| != | Not equal to |
| ?: | Ternary if-then-else |

**Note: Write an example that reflects all operators.**

15 **Is multilevel inheritance is possible in java language? If Yes, explain in detailed about with an example. If No, justify your answer.**

**Answer:** Yes, java supports multilevel inheritance.

➢ For explanation refer **Question No:7** answer, **Topic: Multilevel Inheritance**

16 **How can you pass parameters from child class to parent class in multilevel inheritance? Explain with an example.**

**Answer:** By using super keyword I can pass parameters from child class to parent class. Here is the example.

```
class A
{
	int x,y;
	A()
	{
		System.out.println("Class A constructor");
	}
	A(int a,int b)
	{
		x=a;
		y=b;
	}
	void sum()
	{
		System.out.println((x+y));
	}
}
class B extends A
{
	int x,y;
	B()
	{
		System.out.println("Class B constructor");
	}
	B(int a,int b,int c,int d)
	{
		super(a,b);
		x=c;
		y=d;
	}
	void sub()
	{
		System.out.println((x-y));
	}
}
class C extends B
{
	int x,y;
	C()
	{
		System.out.println("Class C constructor");
	}
	C(int a,int b,int c,int d,int e,int f)
	{
		super(a,b,c,d);
		x=e;
		y=f;
	}
	void mul()
	{
		System.out.println((x*y));
	}
	public static void main(String[] args)
	{
		C c=new C();
		C c1=new C(6,5,4,3,2,1);
		c1.sum();
		c1.sub();
		c1.mul();
```

|  |  |
|---|---|
|  | }<br>} |
| 17 | **Can you extend the interface? If Yes, explain in detailed about with an example. If No, justify your answer.**<br>**Answer:** Yes, I can one interface into another interface.<br>**Explanation:**<br>➢ One interface can inherit another by use of the keyword **extends**.<br>➢ The syntax is the same as for inheriting classes.<br>➢ When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.<br>➢ Following is an example:<br>`interface impinter1`<br>`{`<br>`    void test1();`<br>`    void test2();`<br>`}`<br>`interface impinter2 extends impinter1`<br>`{`<br>`    void test3();`<br>`    void test4();`<br>`}`<br>`class test implements impinter2`<br>`{`<br>`    public void test1()`<br>`    {`<br>`            System.out.println("Hello from test1");`<br>`    }`<br>`    public void test2()`<br>`    {`<br>`            System.out.println("Hello from test2");`<br>`    }`<br>`    public void test3()`<br>`    {`<br>`            System.out.println("Hello from test3");`<br>`    }`<br>`    public void test4()`<br>`    {`<br>`            System.out.println("Hello from test4");`<br>`    }`<br>`    public static void main(String[] args)`<br>`    {`<br>`            impinter1 i1=new test();`<br>`            i1.test1();`<br>`            i1.test2();`<br>`            impinter1 i2=new test();`<br>`            i2.test1();`<br>`            i2.test2();`<br>`    }`<br>`}` |
| 18 | **How can you implement interfaces into the classes? Explain with an example.**<br>**Answer:**<br>➢ Once an interface has been defined, one or more classes can implement that interface.<br>➢ To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.<br>➢ The general form of a class that includes the implements clause looks like this:<br>`class classname [extends superclass] [implements interface [,interface...]]`<br>`{`<br>`  // class-body` |

```
    }
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

**Example:**

```
interface inter1
{
        void display();
}
class A implements inter1
{
        public void display()
        {
                System.out.println("inter1 interface display method was implemented from
class A.");
        }
        void show()
        {
                System.out.println("Incomplete implementation by abstract class.");
        }
        public static void main(String[] args)
        {
                A a=new a();
                a.show();
                a.display();
                inter1 i1=a;
                i1.display();
        }
}
```

19 **Differentiate between Abstract class and Interface with an example.**
**Answer: Refer Question No:6 answer**

20 **Explain in detailed about super keyword in java language.**
**Answer:**

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.
- super has two general forms.
    - The first calls the superclass' constructor.
    - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

**Example:**

```
class classfirst
{
        int x;
        classfirst(int a)
        {
                x=a;
        }
        void display()
        {
                System.out.println("parent class display method.");
        }
}

class classsecond extends classfirst
{
        int x;
        classsecond(int a,int b)
        {
                super(a); //value passing to super class constructor
                //super.x=a; // Assigning value to super class variable
                x=b;
        }
        void sum()
        {
                System.out.println(super.x+x);
        }
        void display()
        {
                super.display(); //Calling super class method from child class
                System.out.println("child class display method.");

        }
        public static void main(String... args)
        {
                classsecond cs=new classsecond(100,200);
                cs.sum();
                cs.display();
        }
}
```

21 **Explain in detailed about JVM architecture?**
**Answer: Refer your class notes / Assignment notes**

22 **What is meant by constructor? Explain in detailed about constructor overloading?**
**Answer:** A constructor is like a method defines what occurs when an object of a class is created.
**Constructor overloading:** Defining a constructor for multiple times with varying of input parameters within a class is known as constructor overloading.

- The respective constructor will be called based on given parameters.
- Here is example:

```
    class single
    {
        int x,y,z;
        single()
        {
                System.out.println("Printing from constructor");
                x=100;y=200;z=x+y;
                System.out.println("Addition value is:"+z);
        }
        single(int x,int y)
        {
                x=x;
                y=y;
                z=x+y;
                System.out.println("Addition value is:"+z);
        }
        public void display()
        {
                this.x=200;
                this.y=500;
                this.z=x+y;
                System.out.println("Addition value is:"+z);
        }
        public static void main(String[] arf)
        {
                single s=new single();
                single s1=new single(1020,1020);
                s.display();
        }
    }
```

23 **What are the primitive data types available in java? Explain with an example?**
**Answer: Refer Question no: 10 answer**

24 **Explain in detailed about java buzzwords?**
**Answer:** Here the list of Java Buzzwords

I. Simple
II. Secure
III. Portable
IV. Object-oriented
V. Robust
VI. Multithreaded
VII. Architecture-neutral
VIII. Interpreted
IX. High performance
X. Distributed
XI. Dynamic

**Simple:**

- ➢ Java was designed to be easy for the professional programmer to learn and use effectively.
- ➢ Assuming that you have some programming experience, you will not find Java hard to master.
- ➢ If you already understand the basic concepts of object-oriented programming, learning Java will be even easier.
- ➢ If you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

**Security**

- ➢ As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.
- ➢ For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system.
- ➢ In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
- ➢ Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.
- ➢ The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

**Portability**

- ➢ Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- ➢ If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
- ➢ The same mechanism that helps ensure security also helps create portability.

**Object-Oriented**

- ➢ Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language.
- ➢ This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects.
- ➢ Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purists’s “everything is an object” paradigm and the pragmatist’s “stay out of my way” model.

- The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

**Robust**

- The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.
- To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development.
- At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.
- Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.
- To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors).
- Memory management can be a difficult, tedious task in traditional programming environments.
- For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.
- Java virtually eliminates these problems by managing memory allocation and deallocation for you.
- Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs.
- Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

**Multithreaded**

- Java was designed to meet the real-world requirement of creating interactive, networked programs.
- To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
- The Java run-time system comes with an elegant yet sophisticated solution for multi process synchronization that enables you to construct smoothly running interactive systems.
- Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

**Architecture-Neutral**

- A central issue for the Java designers was that of code longevity and portability.
- At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine.
- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

**Interpreted and High Performance**

- As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
- This code can be executed on any system that implements the Java Virtual Machine.
- Most previous attempts at cross-platform solutions have done so at the expense of performance.

- The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

**Distributed**

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file.
- Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

**Dynamic**

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- This makes it possible to dynamically link code in a safe and expedient manner.
- This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

25 **How many access specifiers are available in java language? List and explain all with an example.**

**Answer:** There are totally 4 access specifiers are available in java language. Here the list

I. private : can be accessed within the class only
II. default : can be accessed with in the package
III. protected : can be accessed within the package and child of another package
IV. public: can be accessed within the package and another package also.

- Here is the example:

```
package MySecond;
public class A
{
    void test1()
    {
            System.out.println("Access is:Default, from MySecond package");
    }
    private void test2()
    {
            System.out.println("Access is:privtae, from MySecond package");
    }
    protected void test3()
    {
            System.out.println("Access is:protected, from MySecond package");
    }
    public void test4()
    {
            System.out.println("Access is:public, from MySecond package");
    }
    public static void main(String... args)
    {
            A a=new A();
            a.test1();
            a.test2();
            a.test3();
            a.test4();
    }
}

class B extends A //child class of A and class within MySecond package
{
    public static void main(String... args)
    {
            A a=new A();
```

```
                a.test1();
                //a.test2(); compile error
                a.test3();
                a.test4();
        }
    }

    class C //Non-child class of A but a class within MySecond package
    {
        public static void main(String... args)
        {
                A a=new A();
                a.test1();
                //a.test2(); compile error
                a.test3();
                a.test4();
        }
    }
    //Other package
    package MyThird;
    class A extends MySecond.A  //child class of class A in MySecond package
    {
        public static void main(String... args)
        {
                A a=new A();
                //a.test1();  compile error
                //a.test2();  compile error
                a.test3();
                a.test4();
        }
    }
    class B
    {
        public static void main(String... args)
        {
                A a=new A();
                //a.test1();  compile error
                //a.test2();  compile error
                //a.test3();  compile error
                a.test4();
        }
    }
```

26 **Differentiate between Abstract class and Interface with an example.**
**Answer: Refer Question No:6 answer**

27 **Why multiple inheritance with classes not supported by java language? Explain with an example.**
**Answer:**

- ➢ In Java, it doesn't support multiple inheritance through classes because of ambiguity problem.
- ➢ For example, If we defined a method named test() in two different classes and we inherited both classes into another class. If we tried to call test() method by using child class object it shows an error.
- ➢ Here, is the example

```
    class A
    {
        public void test()
        {
                System.out.println("test method from class A");
        }
```

```
}
class B
{
    public void test()
    {
            System.out.println("test method from class B");
    }
}
class C extends A,B
{
    public static void main(String[] arg)
    {
            C c=new C();
            c.test();
    }
}
```

28 Differentiate between **this** and **super** keywords with an example.

Answer:

- **super:** super keyword can be used to refer parent class member from child class.
- **this:** this keyword can be used to refer present class members.
- Both keywords can be used before members of the class by using period operator (. Dot operator) between keyword and members.
- Syntax:
  super.member of the parent class
  this.present class member
- Example:

```
class classfirst
{
    int x;
    classfirst(int a)
    {
            x=a;
    }
    void display()
    {
            System.out.println("parent class display method.");
    }
}

class classsecond extends classfirst
{
    int x;
    classsecond(int a,int b)
    {
            super(a);
            //super.x=a;
            this.x=b;
    }
    void sum()
    {
            System.out.println(super.x+x);
            this.display();
    }
    void display()
    {
            super.display();
            System.out.println("child class display method.");

    }
```

```
        public static void main(String... args)
        {
                classsecond cs=new classsecond(100,200);
                cs.sum();
                cs.display();
        }
    }
```

29 **What is meant by method? Explain in detailed about method overloading.**

**Answer:** A block of code that can be executed whenever we called it is known as method.

- **Refer Question No: 9 answer**

30 **Can Abstract class contain both abstract and non-abstract methods? If Yes, explain in detailed about with an example. If No, justify your answer?**

**Answer:** Yes, Abstract class can contain both abstract and non-abstract methods within it.

- By using a keyword abstract we can define abstract class.
- The method which defines by using a keyword abstract and contains only definition but not implementation of it is known as abstract method.
- Abstract methods should be implemented by its child class.
- We can't create object / instance for abstract class because abstract classes are not completely defined but we can create the reference of abstract class by adding child class object to it.
- By using abstract class reference we can call only non-abstract methods of it.
- Here is an example:

```
    abstract class absclass
    {
        void display()
        {
                System.out.println("non-abstract method of abstract class.");
        }
        abstract void show();
    }
    class child extends absclass
    {
        void show()
        {
                System.out.println("abstract class method show was implemented.");
        }
        public static void main(String[] args)
        {
                child c=new child();
                c.display();
                c.show();

                absclass ac=c; //abstract class reference
                c.display();
        }
    }
```

31 **Explain in detailed about Evolution of Java language?**

**Answer:**

- The green project began in 1991 by Green team of Sun Microsystems.
    - Green Team Members: James Gosling, Mike Sheridan and Patrick Naughton
    - Designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.
- Original Language name was: Oak
- The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc.
- Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

- In 1995, Time magazine called Java one of the Ten Best Products of 1995.
- Java Slogan: Write Once, Run Anywhere

**Versions of Java language:**

I. JDK Alpha and Beta (1995)
II. JDK 1.0 (23rd Jan, 1996)
III. JDK 1.1 (19th Feb, 1997)
IV. J2SE 1.2 (8th Dec, 1998)
V. J2SE 1.3 (8th May, 2000)
VI. J2SE 1.4 (6th Feb, 2002)
VII. J2SE 5.0 (30th Sep, 2004)
VIII. Java SE 6 (11th Dec, 2006)
IX. Java SE 7 (28th July, 2011)
X. Java SE 8 (18th March, 2014)

**JDK 1.1 (19th Feb, 1997) features:**
- Inner Classes
- Java Beans
- JDBC
- RMI

**J2SE 1.2 (8th Dec, 1998) features:**
- Swing Graphical API
- JIT Compiler
- Java Plug-in
- Collection Network

**J2SE 1.3 (8th May, 2000) features:**
- HotSpot JVM
- JNDI Interface
- Java Sound API
- Debugging Architecture

**J2SE 1.4 (6th Feb, 2002) features:**
- Regular Expression implementation
- IPv6 network communication
- Logging API
- XML and XSLT
- Security and Cryptography
- Java Web Start
- JDBC 3.0 API
- Chained Exceptions
- Image I/O API

**J2SE 5.0 (30th Sep, 2004) (1.5) features:**
- Generics
- Metadata
- Enumerations
- Variable Arguments
- For-each enhancement
- AutoBoxing and UnBoxing

**Java SE 6 (11th Dec, 2006) features:**
- Performance JDBC 4.0
- GUI Improvements
- Integrated Web Services
- In 2010, Oracle bought Sun Microsystems

**Java SE 7 (28th July, 2011) features:**
- Strings in Switch

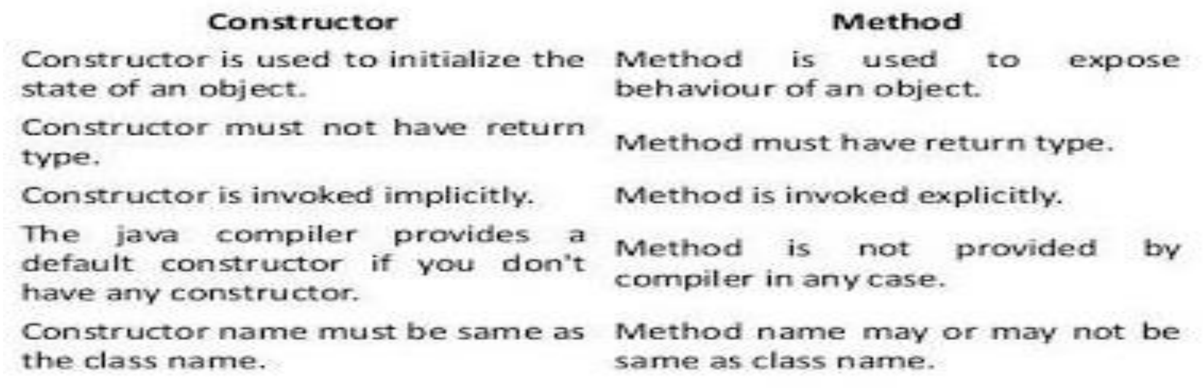
- Try-Catch improvements
- Simplified variable arguments
- Underscores in numerical literals
- Automatic Null handling
- Support for Dynamic languages
- Java nio package

**Java SE 8 (18th March, 2014)**

- Lambda expressions
- Pipelines and Streams
- Date and Time API
- Default Methods
- Type Annotations   Eg: @NoNull String str;
- Parallel operations

32 **List out and explain the methods of StringBuffer class in java (at-least 4 methods).**
**Answer: for answer refer Question No: 11 answer**

33 **Differentiate between constructor overloading and method overloading with an example?**
**Answer:**



| Constructor | Method |
|---|---|
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

Example:
```
class test
{
        int x,y;
        String str;
        test()
        {
        }
        test(String str)
        {
                this.str=str;
        }
        test(int x)
        {
                this.x=x;
        }
        test(int x,int y)
        {
                this.x=x;
                this.y=y;
        }
        public void display(String str)
        {
                System.out.println("Hello "+str);
        }
        public void display(int x)
        {
                System.out.println("Square value is "+(x*x));
```

```
	}
	public void display(int x,int y)
	{
		System.out.println("Addition value is "+(x+y));
	}
	public static void main(String[] args)
	{
		test t1=new test();
		test t2=new test("Vardhaman");
		test t3=new test(10);
		test t4=new test(100,200);
		t1.display("Vardhaman");
		t1.display(100);
		t1.display(1000,2000);
	}
}
```

34 **Explain in detailed about arrays in java language?**
**Answer: For answer refer Question No:13 answer**

35 **Explain in detailed about Abstract classes?**
**Answer:**

- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:
  abstract type name(parameter-list);
- As you can see, no method body is present.
- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.
- You cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

**Example:**

```
abstract class absclass
{
	void display()
	{
		System.out.println("non-abstract method of abstract class.");
	}
	abstract void test();
}
abstract class child extends absclass
{
	void show()
	{
		System.out.println("non-abstract method of child class.");
	}
	abstract void test1();
}
class C extends child
{
	void test()
	{
```

```
                System.out.println("abstract method of absclass was implemented.");
        }
        void test1()
        {
                System.out.println("abstract method of child class was implemented.");
        }
        public static void main(String[] args)
        {
                C c=new C();
                c.display();
                c.show();
                c.test();
                c.test1();

                absclass ac=c;
                c.display();

                child c1=c;
                c1.show();
        }
}
```

36 **Can you extend the interface? If Yes, explain in detailed about with an example. If No, justify your answer**
**Answer: For answer refer Question No: 17**

37 **How can you pass parameters from child class to parent class in multilevel inheritance? Explain with an example?**
**Answer: For answer refer Question no: 16 answer**

38 **How can you import user defined package into the program? Explain with neat example (create user defined package and import it)?**
**Answer:** By using import keyword we can import user / pre-defined package into the class.

- For importing we need to follow naming hierarchy i.e. package_name.class_name;
  Syntax: import package_name.class_name;
- If package contains sub-packages we need to follow the following syntax.
  Syntax: import main_package.sub_package.class_name
  Example: import java.util.Scanner;
- Here, is the example:

```
package MySecond;
public class A
{
    void test1()
    {
            System.out.println("Access is:Default, from MySecond package");
    }
    private void test2()
    {
            System.out.println("Access is:privtae, from MySecond package");
    }
    protected void test3()
    {
            System.out.println("Access is:protected, from MySecond package");
    }
    public void test4()
    {
            System.out.println("Access is:public, from MySecond package");
    }
    public static void main(String... args)
    {
            A a=new A();
```

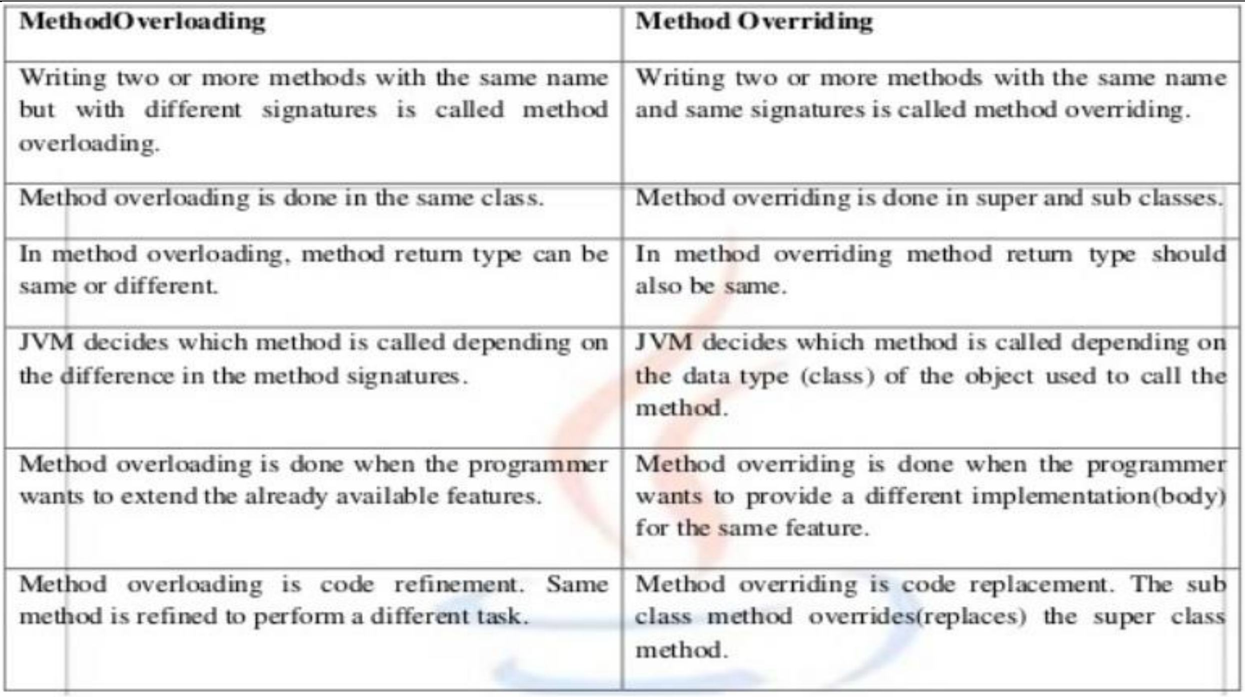
| | |
|---|---|
| | `            a.test1();`<br>`            a.test2();`<br>`            a.test3();`<br>`            a.test4();`<br>`        }`<br>`    }`<br><br>`    package MyThird;`<br>`    class A extends MySecond.A  //child class of class A in MySecond package`<br>`    {`<br>`        public static void main(String... args)`<br>`        {`<br>`            A a=new A();`<br>`            //a.test1();  compile error`<br>`            //a.test2();  compile error`<br>`            a.test3();`<br>`            a.test4();`<br>`        }`<br>`    }` |
| 39 | **What is meant by control statements? Explain control statements of java language in detailed with examples?**<br>**Answer: For answer refer your assignment notes** |
| 40 | **Differentiate between Abstract class and Interface with an example?**<br>**Answer: For answer refer Question No: 06 answer.** |
| 41 | **Is multilevel inheritance is possible in java language? If Yes, explain in detailed about with an example. If No, justify your answer?**<br>**Answer: For answer refer Question No:7 answer, Topic: Multilevel Inheritance** |
| 42 | **Explain in detailed about java buzzwords?**<br>**Answer: For answer refer Question No:24 answer** |
| 43 | **Explain in detailed about JVM architecture?**<br>**Answer: Refer your class notes / Assignment notes** |
| 44 | **What are the primitive data types available in java? Explain with an example.**<br>**Answer: Answer: Refer Question no: 10 answer** |
| 45 | **Explain in detailed about the different operators available in java language with examples?**<br>**Answer: Answer: Refer Question no: 14 answer** |
| 46 | **Differentiate between Static binding and Dynamic binding with an examples.**<br>**Answer:**<br>**Static binding:** It is the mechanism by which a call to an overridden method is resolved at compile time.<br>**Dynamic binding:** It is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.<br>**Example for static binding:**<br>`    class A`<br>`    {`<br>`        public void display()`<br>`        {`<br>`            System.out.println("A class display method.");`<br>`        }`<br>`    }`<br>`    class B extends A`<br>`    {`<br>`        public void display()`<br>`        {`<br>`            System.out.println("B class display method.");`<br>`        }`<br>`    }`<br>`    class C extends B`<br>`    {` |

```
        public void display()
        {
                System.out.println("C class display method.");
        }
    }
    class D
    {
        public static void main(String[] ar)
        {
                A a=new A();
                B b=new B();
                C c=new C();
                a.display();
                b.display();
                c.display();
        }
}
```

**Example for dynamic binding:**

```
    class A
    {
        public void display()
        {
                System.out.println("A class display method.");
        }
    }
    class B extends A
    {
        public void display()
        {
                System.out.println("B class display method.");
        }
    }
    class C extends B
    {
        public void display()
        {
                System.out.println("C class display method.");
        }
    }
    class D
    {
        public static void main(String[] ar)
        {
                A a=new A();
                B b=new B();
                C c=new C();
                A x; //defining super class reference
                x=a;
                x.display();
                x=b;
                x.display();
                x=c;
                x.display();
        }
}
```

| | |
|---|---|
| 47 | **Explain the differences between method overloading and method overriding with an example?**<br>**Answer:** |



| MethodOverloading | Method Overriding |
|---|---|
| Writing two or more methods with the same name but with different signatures is called method overloading. | Writing two or more methods with the same name and same signatures is called method overriding. |
| Method overloading is done in the same class. | Method overriding is done in super and sub classes. |
| In method overloading, method return type can be same or different. | In method overriding method return type should also be same. |
| JVM decides which method is called depending on the difference in the method signatures. | JVM decides which method is called depending on the data type (class) of the object used to call the method. |
| Method overloading is done when the programmer wants to extend the already available features. | Method overriding is done when the programmer wants to provide a different implementation(body) for the same feature. |
| Method overloading is code refinement. Same method is refined to perform a different task. | Method overriding is code replacement. The sub class method overrides(replaces) the super class method. |

Example:

```
class A
{
        public void display(String str)
        {
                System.out.println("Hello "+str);
        }
        public void show(int x)
        {
                System.out.println("Square value is "+(x*x));
        }
}
class test extends A
{
        public void display(String str,int age) //method overloading
        {
                System.out.println("Hello "+str+"age is: "+age);
        }
        public void show(int x) //method overriding
        {
                System.out.println("cube value is "+(x*x*x));
        }
        public static void main(String[] args)
        {
                test t1=new test();
                t1.display("Rajesh Kumar");
                t1.display("Rakesh",23);
                t1.show(10);
        }
}
```

| | |
|---|---|
| 48 | **How can you implement interfaces into the classes? Explain with an example?**<br>**Answer: For answer refer Question No: 18 answer** |
| 49 | **Differentiate between procedure oriented programming and object oriented programming.**<br>**Answer: For answer refer Question NO: 12 answer** |
| 50 | **Explain in detailed about super keyword with an example?**<br>**Answer: For answer refer Question No: 20 answer** |