Q. 1 What is inheritance? Explain different types of inheritance with example

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

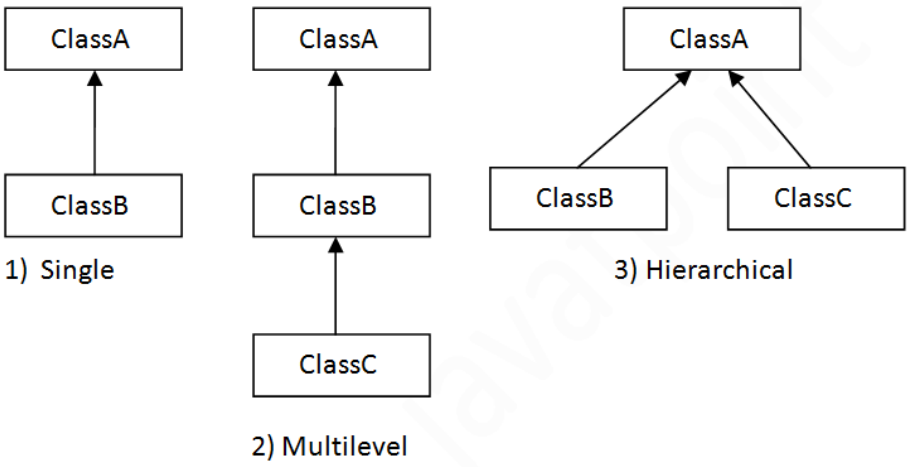
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Example –

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output –

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
 class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
```

```
d.bark();
d.eat();
}}
```

Output –

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output -

```
meowing...
eating...
```

2. What is Method Overloading? Explain with the help of program.

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b)
{
return a+b;
}
static double add(double a, double b)
{
return a+b;
}
}
class TestOverloading2
{
public static void main(String[] args)
```

```
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}
}
```

Output -

```
22
24.9
```

Q. 3 What is method overriding? Explain with example. OR What is run time polymorphism? Explain with example.

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

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example

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
  }
}
```

Output:

Bike is running safely

Q. 4 Difference between compile time & run time polymorphism

Compile Time Polymorphism	Runtime Polymorphism
It is also known as static polymorphism, early binding, or overloading	It is also known as dynamic polymorphism, late binding, or overriding
It is executed at the compile time	It is executed at the run time
The method is called with the help of a compiler	The method is not called with the help of a compiler
The program's execution is fast as it involves the use of a compiler	The program's execution is slow as it does not involve a compiler
It is achieved by function overloading and operator overloading	It is achieved by virtual overloading and pointers
Compile-time polymorphism tends to be less flexible as all commands are operated at the compile time Example: Method Overloading	Run time polymorphism tends to be more flexible as all commands are executed at the run time Example: Method Overriding

Q. 5 Explain abstract classes with example.

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body)

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike
{
  abstract void run();
}
class Honda extends Bike
{
```

```
void run()
{
System.out.println("running safely");
}
public static void main(String args[])
{
 Bike obj = new Honda();
 obj.run();
}
}
```

Output –

running safely

Q. 6 Explain interface with example.

- 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.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

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

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Java Interface Example

In this example, the Bike interface has only one method, and its implementation is provided in the Honda class.

```
interface Bike
```

```
{
  void run();  //by default - public static
}
class Honda implements Bike
{
void run()
{
System.out.println("running safely");
}
public static void main(String args[])
{
 Bike obj = new Honda();
 obj.run();
}
}
```

Output:

running safely

Q. 7 Differentiate between abstract classes & interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Q. 8 Explain final keyword. Write an example to illustrate the use of 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:

1. variable
2. method
3. 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.

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();
 }
}
```

Output:

Compile Time Error

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();
  }
}
```

Output:
Compile Time Error

3) Java final class

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

Example of final class

```
final class Bike
{
int a = 10;
}

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();
 }
}
```

Output:
Compile Time Error

Q. 9 Explain various Access modifiers in java.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
```

```
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  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();//Compile Time Error
    obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output –

Hello

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//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();
  }
}
```

Output –

Hello

Q. 10 Explain the purpose of super keyword with an example.

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword -

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Example -

```
class Animal
{
String color="white";
void printColor()
{
System.out.println(color);
}

}
class Dog extends Animal
{
String color="black";
void printColor()
{
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
void demo()
{
super.printColor();//call printColor method in Animal class
}
}
class TestSuper1
{
public static void main(String args[])
{
Dog d=new Dog();
d.printColor();
d.demo();
}
}
```

Output –

black

white

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

Q.11 What do you mean by package? Explain the procedure for creating user defined package.

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.

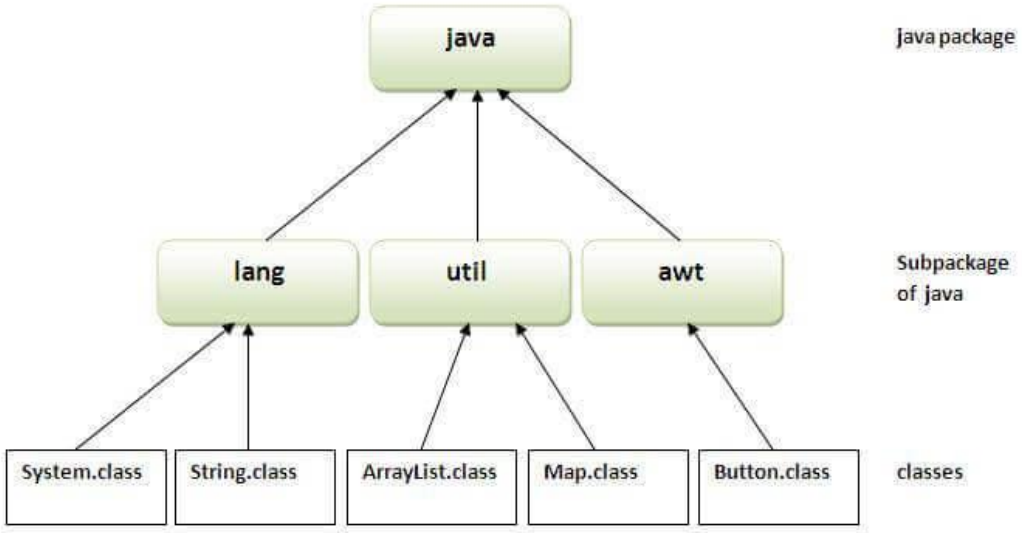
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

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.



Simple example of java package

//save as A.java

```
package mypack;

public class A

{

private int a = 10;

public void show()

{

System.out.println("Value = " +a);

}
```

```
}
```

//save as B.java

```
import mypack.A;
class B
{
public static void main(String args[])
{
A obj = new A();
obj.show();
}
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file.

How to run java package program

You need to use fully qualified name e.g. mypack.A etc to run the class.

Q. 12 Design an enumeration for week days and print their corresponding description according to traditional rules.

```
public class Demo {
  enum Days {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
  }
  public static void main(String[] args) {
    Days today = Days.Wednesday;
    Days holiday = Days.Sunday;
    System.out.println("Today = " + today);
```

```
    System.out.println(holiday+ " is holiday");
  }
}
```

Output

Today = Wednesday

Sunday is holiday