BCA SEM-3

Programming using Java
(05BC3306)

Unit – 2
Inheritance and packages

Inheritance in Java

- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviours of 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 parent class, and you can add new methods and fields also.

- Inheritance represents the **IS-A relationship,** also known as *parent-child* relationship.

Why we use inheritance in java ?

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

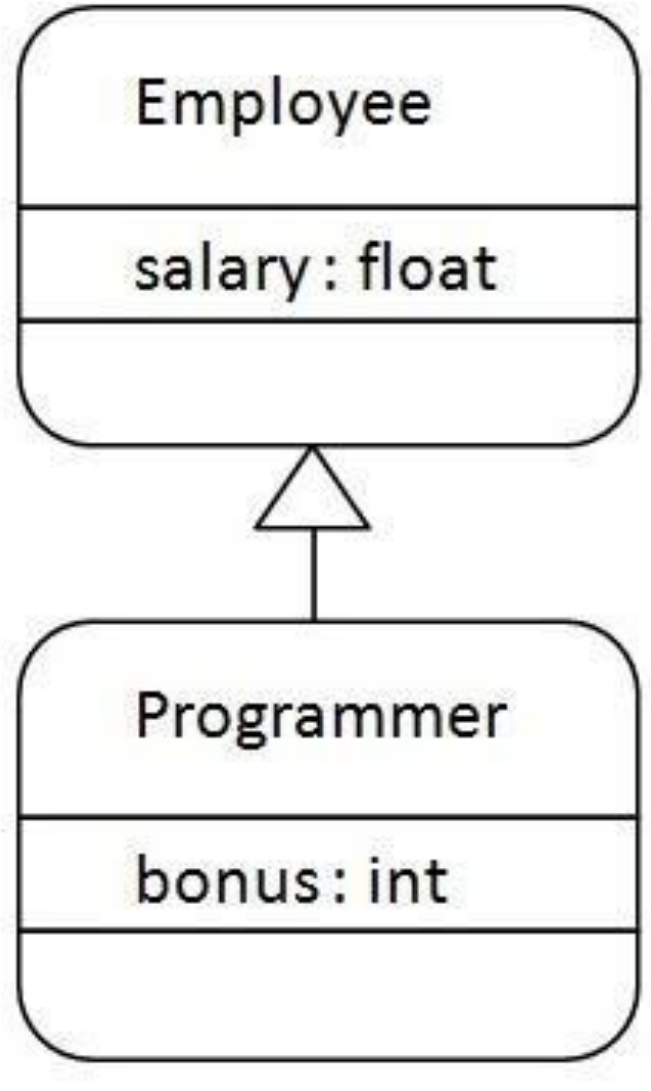
- For Code Reusability.

Syntax of Java Inheritance

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class.
- In the terminology of Java, a class that is inherited is called a super class. The new clas s is called a subclass.

Understanding the simple example of inheritance



As displayed in the figure, Programmer is the subclass and Employee is the superclass.

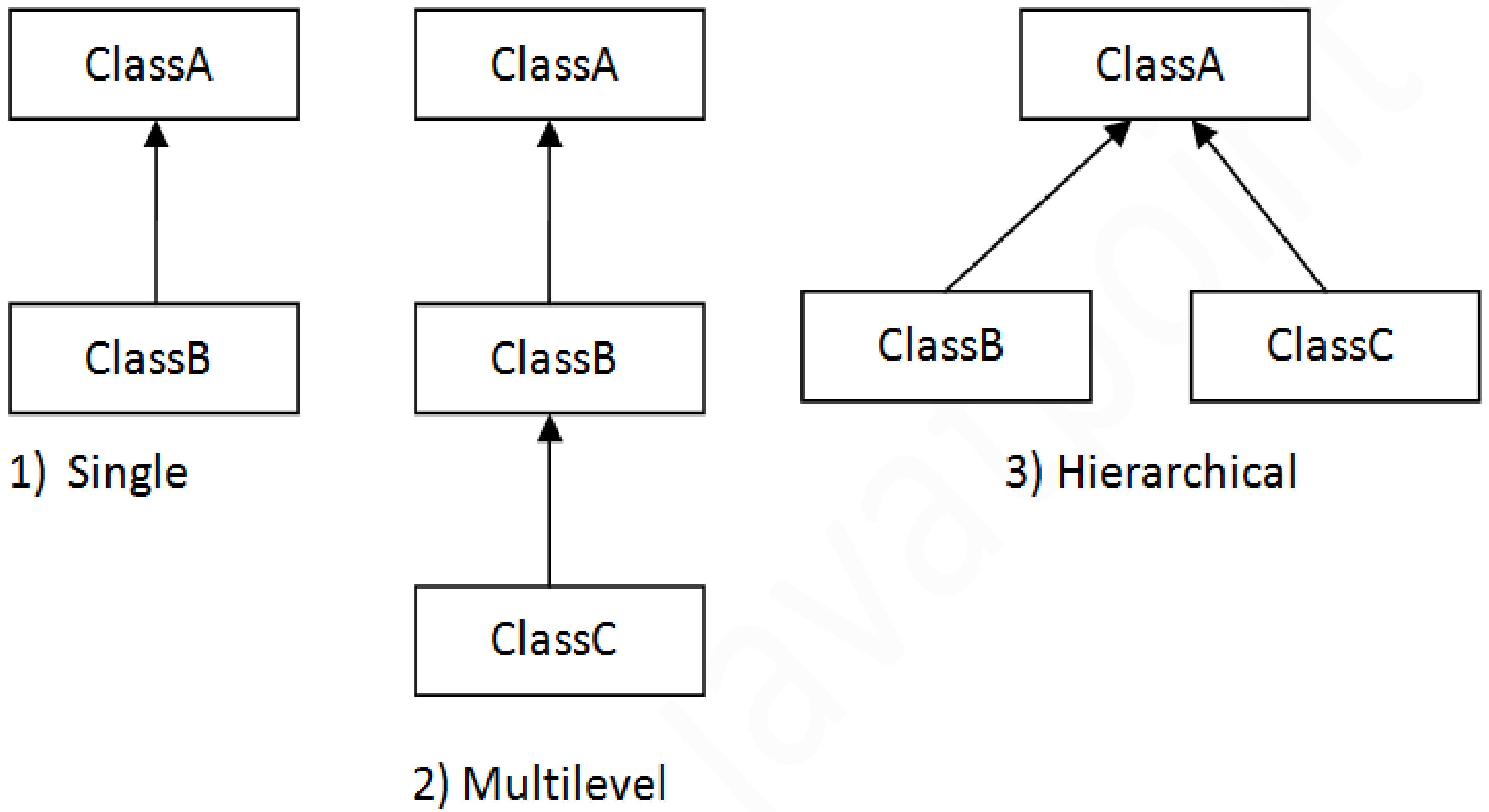
Relationship between two classes is **Programmer IS-A Employee.**

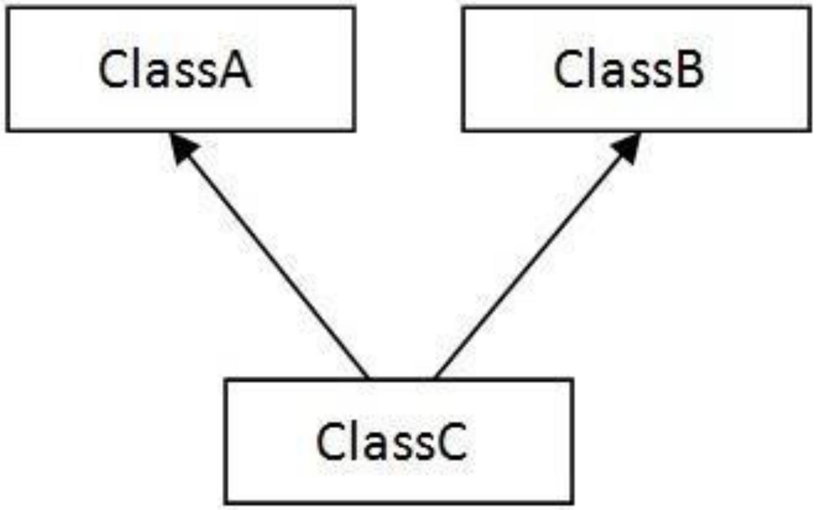
It means that Programmer is a type of Employee.

Types of inheritance in java

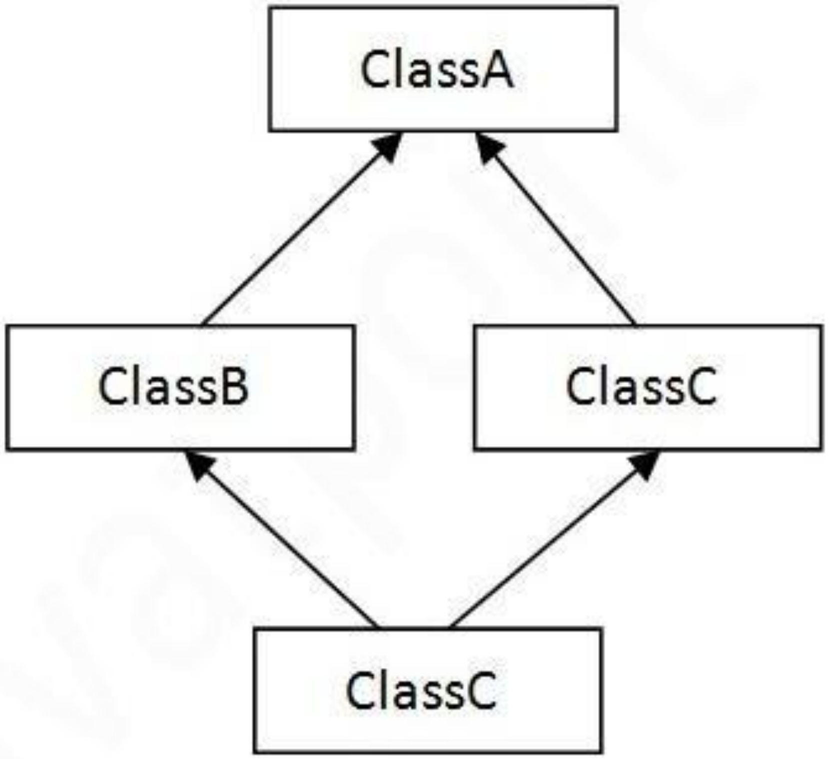
- On the basis of class, there can be three types of inheritance in java:
- single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.

Types of inheritance in java





4) Multiple



5) Hybrid

Note: Multiple inheritance is not supported in java through class.
When a class extends multiple classes i.e. known as multiple inheritance.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

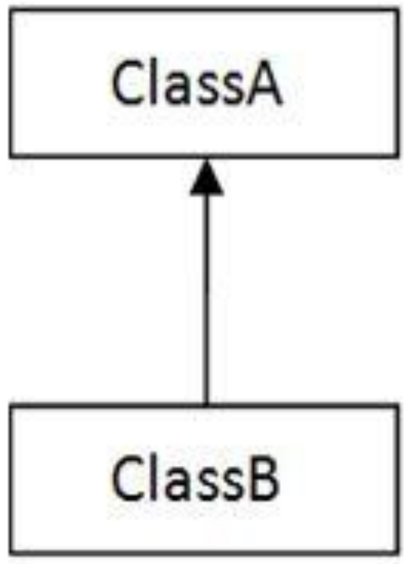
Try this to implement multiple inheritance and see the result

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
 Public Static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Result - Compile Time Error

(1) Single inheritance

In **Single Inheritance** there is only one Super Class and Only one Sub Class Means they have one to one Communication between them.



1) Single

(1) Single inheritance

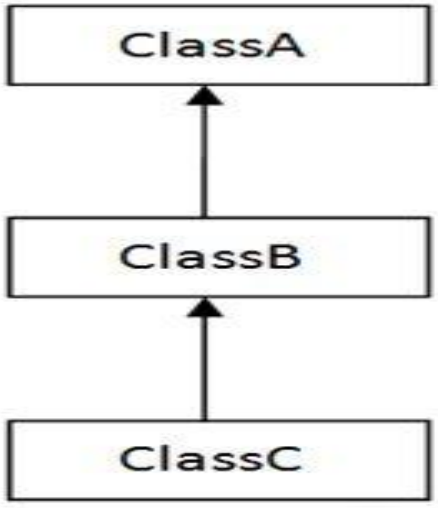
```
class Room
{
        int l,w;
        Room(int l,int w)
        {
                this.l=l;
                this.w=w;
        }
        int area()
        {
                return (l*w);
        }
}
class BedRoom extends Room
{
        int h;
        BedRoom(int x,int y,int z)
        {
                super(x,y);
                h=z;
        }
        int volume()
        {
                return(area()*h);
        }
}
```

(1) Single inheritance

```
class ex11
{
        public static void main(String args[])
        {
                BedRoom r1=new BedRoom(14,12,10);
                int a1=r1.area();
                int v1=r1.volume();
                System.out.println("Area= " +a1);
                System.out.println("Volume = "+v1);
        }
}
```

(2) Multilevel inheritance

In Multilevel inheritance there is a concept of grand parent class.



2) Multilevel

If we take the example of above diagram then class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that’s what is called multilevel inheritance.

(2) Multilevel inheritance

```
class stud
{
		int r;
		String n;

		void getData(int r,String n)
		{
				this.r=r;
				this.n=n;
		}

		void putData()
		{
				System.out.println("Roll Number : "+r);
				System.out.println("Name is : "+n);
		}
}
```

(2) Multilevel inheritance

```
class marks extends stud
{
        int m1,m2,m3;

        void getMarks(int s1,int s2,int s3)
        {
                m1=s1;
                m2=s2;
                m3=s3;
        }
        void putMarks()
        {
                System.out.println("Marks 1 : "+m1);
                System.out.println("Marks 2 : "+m2);
                System.out.println("Marks 3 : "+m3);
        }
}
```

(2) Multilevel inheritance

```
class total extends marks
{
		int tot=0;
		void cal()
		{
				tot=m1+m2+m3;
		}
		void putTot()
		{
				putData();
				putMarks();
				System.out.println("Total is  :"+tot);
		}
}
```

(2) Multilevel inheritance

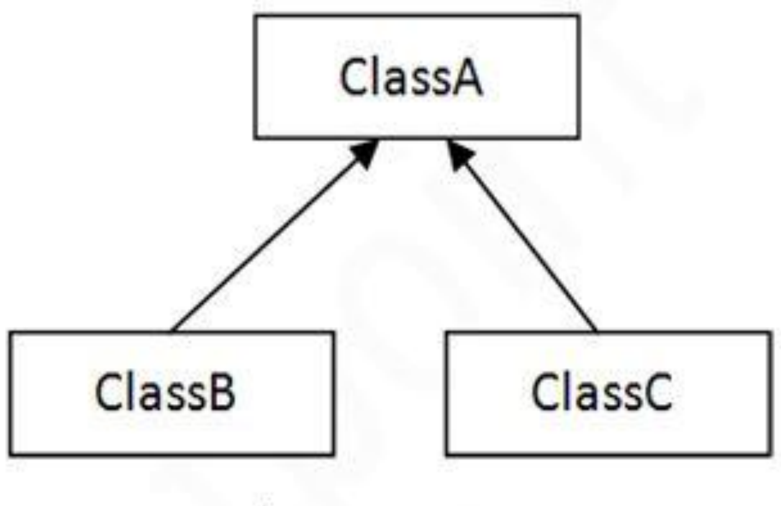
```
class MultilevelEx
{
        public static void main(String args[])
        {
                total t=new total();
                t.getData(1,"Abc");
                t.getMarks(50,50,50);
                t.cal();

                t.putTot();
        }

}
```

(3) Hierarchical inheritance

Hierarchical Inheritance is that in which a Base Class has Many Sub Classes or When a Base Class is used or inherited by many Sub Classes.



3) Hierarchical

(3) Hierarchical inheritance

```
class One
{
		int x=10,y=20;
		void display()
		{
				System.out.println("Value of X : "+x);
				System.out.println("Value of Y : "+y);
		}
}

class Two extends One
{
		void addNum()
		{
				int z=x+y;
				System.out.println("Addition is : "+z);
		}
}
```

(3) Hierarchical inheritance

```
class Three extends One
{
		void mulNum()
		{
				int z=x*y;
				System.out.println("Multiplication is : "+z);
		}
}

class HierarchicalEx
{
		public static void main(String args[])
		{
				Two t2=new Two();
				Three t3=new Three();
				t2.display();
				t2.addNum();
				t3.mulNum();
		}
}
```

Method Overloading in Java

If a class has multiple methods by same name but different 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.

Method Overloading in Java

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 java, Method Overloading is not possible by changing the return type of the method.

Method Overloading in Java

1) Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

Method Overloading in Java

```
class Calculation
{
void sum(int a,int b) { System.out.println(a+b); }
void sum(int a,int b,int c) { System.out.println(a+b+c); }

public static void main(String args[]){
Calculation obj=new Calculation();
obj.sum(10,10,10);
obj.sum(20,20);
}
}
```

```
Output: 30
40
```

Method Overloading in Java

2) Example of Method Overloading by changing data type of argument

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

Method Overloading in Java

```
class Calculation2
{
void sum(int a,int b){System.out.println(a+b);}
void sum(double a,double b){System.out.println(a+b);}
public static void main(String args[]){
Calculation2 obj=new Calculation2(); obj.sum(10.5,10.5);
obj.sum(20,20);
}
}
```

Test it Now Output: 21.0

```
40
```

Method Overriding in Java

- 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 subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Use of Method Overriding in Java

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.

- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

Example of Method Overriding

```
class abc
{
    int x;
    abc(int x)
    {
    this.x=x;
    }
    void display()
    {

    System.out.println("Output From Base Class x = "+x);
    }

}
```

Example of Method Overriding

```
class xyz extends abc
{
    int y;
    xyz(int x,int y)
    {
    super(x);
    this.y=y;
    }
    void display()
    {
    System.out.println("Output From Derived Class x = "+x+" y = "+y);
    super.display();
    }
}
class ex12
{
    public static void main(String args[])
    {
    xyz p1=new xyz(100,200);
    p1.display();

    }
}
```

FAQ of Method Overriding

(1) Can we override static method ?

Ans: No, static method cannot be overridden.It can be proved by runtime polymorphism.

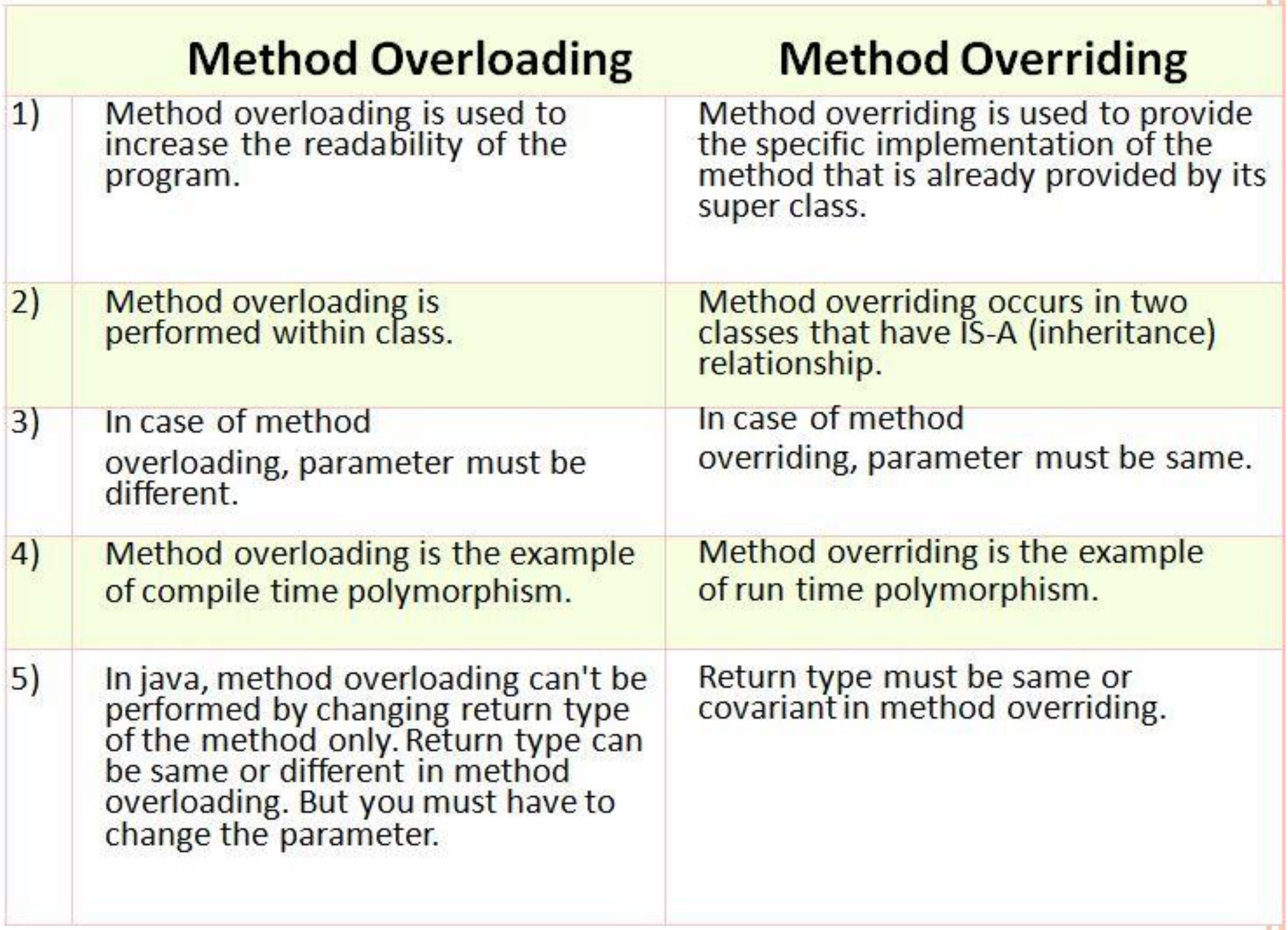
(2) Why we cannot override static method?

Ans: because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

(3) Can we override java main method?

Ans: No, because main is a static method

Method Overloading Vs. Method Overriding



	Method Overloading	Method Overriding
1)	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2)	Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3)	In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
4)	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
5)	In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding.

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5
{
        int id;
        String name;
        int age;
        Student5(int i,String n)
        {
                id = i;
                name = n;
        }
```

Constructor Overloading in Java

```
        Student5(int i,String n,int a)
        {
                id = i;
                name = n;
                age=a;
        }
        void display(){System.out.println(id+" "+name+" "+age);}
        public static void main(String args[])
        {
                Student5 s1 = new Student5(111,"Karan");
                Student5 s2 = new Student5(222,"Aryan",25);
                s1.display();
                s2.display();
        }
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Command Line Arguments

- Sometimes we need to pass some values while running the program. Here our program will work according to the values given by us at the time of executing the program.

- The java command-line argument is an argument i.e. passed at the time of running the java program.

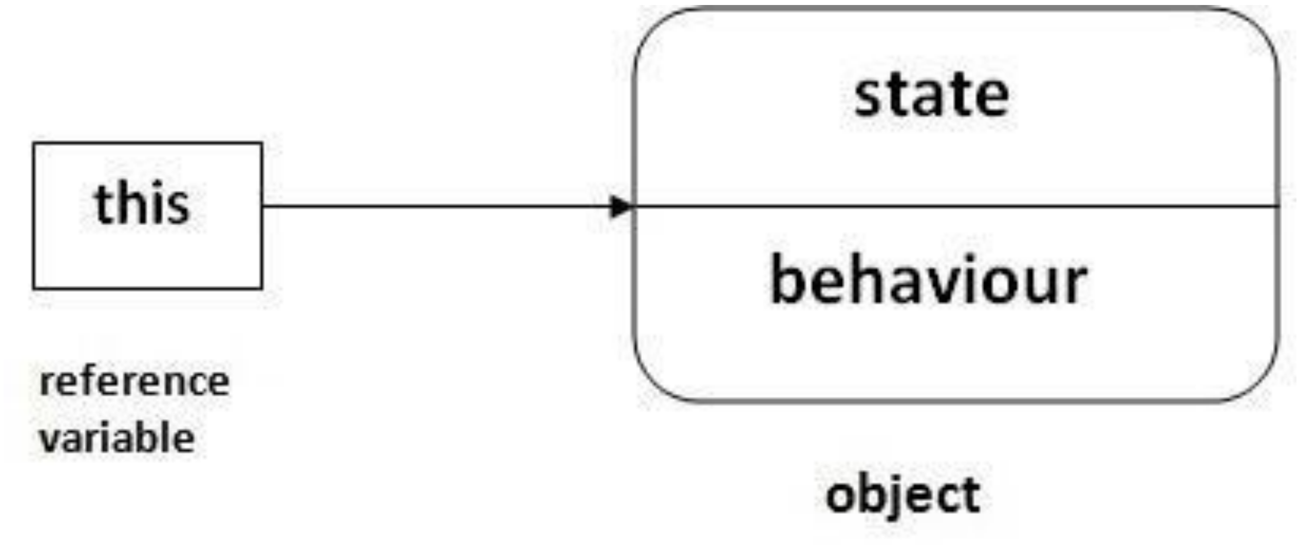
- The arguments passed from the console can be received in the java program and it can be used as an input.

Command Line Arguments

```
class CommandLineEx
{
    public static void main(String args[])
    {
    int count,i=0;
    String s;
    count=args.length;
    System.out.println("Number Of Arguments = "+count);
    while(i<count)
    {
            s=args[i];
            i=i+1;
            System.out.println(i+" Java Is "+s);
    }
    }
}
```

this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



Usage of java this keyword

Here is given the 6 usage of java this keyword.

- this keyword can be used to refer current class instance variable.
- this() can be used to invoke current class constructor.
- this keyword can be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this keyword can also be used to return the current class instance.
- If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student10
{
		int id;
		String name;
		Student10(int id,String name)
		{
				id = id;
				name = name;
		}
		void display()
		{System.out.println(id+" "+name);}

		public static void main(String args[])
		{
				Student10 s1 = new Student10(111,"Karan");
				Student10 s2 = new Student10(321,"Aryan");
				s1.display();
				s2.display();
		}
}
```

Test it Now

Output:

```
0 null
0 null
```

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
//example of this keyword
class Student11
{
		int id;
		String name;

		Student11(int id,String name)
		{
			this.id = id;
			this.name = name;
		}
		void display(){System.out.println(id+" "+name);}

		public static void main(String args[])
		{
			Student11 s1 = new Student11(121,“Rahul");
			Student11 s2 = new Student11(122,“Hardik");
			s1.display();
			s2.display();
		}
}
```

Final Keyword In Java

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.

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
```

Result
Output: Compile Time Error

2) final method

If we wish to prevent the subclass from overriding the members of the super class then we can use final keyword.

```
class abc
{
	int x;
	abc(int x)
	{
		this.x=x;
	}
	final void display()
	{

		System.out.println("Output From Base Class x = "+x);
	}

}
```

2) final method

```
class xyz extends abc
{
        int y;
        xyz(int x,int y)
        {
                super(x);
                this.y=y;
        }
        void display()
        {
                System.out.println("Output From Derived Class x = "+x+" y = "+y);
                super.display();
        }
}
class ex12
{
        public static void main(String args[])
        {
                xyz p1=new xyz(100,200);
                p1.display();

        }
}
```

Now display method cannot be override compiler will throw error.

Questions

Q) Is final method inherited?
Ans) Yes, final method is inherited but you cannot override it.3

Q) What is blank or uninitialized final variable?
A final variable that is not initialized at the time of declaration is known as blank final variable.
If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee. It can be initialized only in constructor.

Example of blank final variable

```
class Student {
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Questions

Que) Can we initialize blank final variable? Yes, but only in constructor. For example:

```
class Bike10
{
	final int speedlimit;//blank final variable

	Bike10()
	{
		speedlimit=70;
		System.out.println(speedlimit);
	}

	public static void main(String args[])
	{
		new Bike10();
	}
}
```

3) final class

Sometimes we need to prevent our class to be extended. For this purpose we have to declare the class with final keyword.
e.g. final class abc

Here in above example when we declare class abc as final it will never be extended to any subclass.(Consider example of method overriding)

abstract class

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

When we define any class with final keyword we know that it will be never extended. Here the word abstract is totally opposite to this. When we define any class with keyword abstract at that time we need to extend this class.

An abstract class is a class that is like all normal classes .

abstract class

Only has two differences.

One, an application can not create an instance of this class. Only instance of its non-abstract sub class can be created.

Two , an abstract class can have abstract methods, which are not allowed in a non-abstract class.

Non abstract class can not have any abstract methods.

A class may be declared to be abstract even if the class has no abstract methods declared and there are no abstract methods that have been inherited. This may be done to prevent creating instances of the class and it enforces the creation of sub-classes.

abstract class

```
abstract class Bike
{
		abstract void runBike();
}

class Honda extends Bike
{
		void runBike()
		{
				System.out.println("running safely..");
		}
}
class ex144
{
		public static void main(String args[])
		{
				Honda obj = new Honda();
				obj.runBike();
		}
}
```

abstract methods

When we define any method with final keyword we know that it will never be overridden. Here the word abstract will work totally opposite to this. When we define any method with the word abstract then it must be implement it’s body in subclass.

A method that is declared as abstract and does not have implementation is known as abstract method.

Abstract method does not have body part.

Interface

Since beginning we know that java does not support multiple Inheritance. Because we can’t write like

class A extends B extends C

On the other hand we also know that multiple inheritance is also 100% useful tool. So java provided us a new approach known as interfaces.

Remember that : java class cannot be a subclass of more than one Super class but it can implement more than one interface.

Interface

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

The interface in java is **a mechanism to achieve fully abstraction**.

There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Defining Interface

In general words we can say interface is basically a kind of class. Like classes in interfaces it also contains methods, variables etc. but with some difference. The main difference is in interface we can define only abstract methods and final fields.

Syntax :

```
interface <interface name>
{
        variable declaration;
        methods declaration;
}
```

Defining Interface

Example

```
interface Item
{
        static final int code=1001;
        static final String name="Fan";
        void display();
}
```

Extending Interface

Just like simple class we can extend one interface to another also. The new interface will inherit all the members of the super interface in the same manner like subclass syntax :

```
interface i2 extends i1
{
        body of i2;
}
```

Example :

```
interface ItemConstants
{
        static final int code=1001;
        static final String name="Fan";
}
interface Item extends ItemConstants
{
        void display();
}
```

Implementing Interface

Interfaces are used as a super class which we can inherit in our class which is must.

syntax :

```
class <clasname> implements <interfacename>
{
            body of class
}
```

Remember that : We can implement more then one interface into one class by implementing more then one interface separated by comma.

```
class <classname> extends <superclass>
        implements <interface1>,<interface2>….
{
            body of the class
}
```

Implementing Interface

```
interface area
{
	final static float pi=3.14f;
	float compute(float x,float y);
}
class rectangle implements area
{
	public float compute(float x,float y)
	{
		return(x*y);
	}
}
class circle implements area
{
	public float compute(float x,float y)
	{
		return(pi*x*x);
	}
}
```

Implementing Interface

```
class InterfaceEx
{
	public static void main(String args[])
	{
		rectangle r1=new rectangle();
		circle c1=new circle();
		System.out.println("Area Of Rectangle = "+r1.compute(10.0f,20.0f));
		System.out.println("Area Of Circle    = "+c1.compute(10.0f,0.0f));
	}
}
```

Implementing Hybrid inheritance

Interface can be used to declare a set of constants that can be used in different classes.

we can use one class to be extended and more than one interfaces to be implemented.

Example

```
class Student
{
	int rno;
	void getNumber(int n)
	{
		rno=n;
	}
	void putNumber()
	{
		System.out.println("Roll No. : ="+rno);
	}
}
```

Implementing Hybrid inheritance

```
class Test extends Student
{
	float p1,p2;
	void getMarks(float m1,float m2)
	{
		p1=m1;
		p2=m2;
	}
	void putMarks()
	{
		System.out.println("Marks obtained");
		System.out.println("Part1=" + p1);
		System.out.println("Part2=" + p2);
	}
}
interface sports
{
		final static float sw=6.0f;
		void putsport();
}
```

Implementing Hybrid inheritance

```
class result extends Test implements sports
{
	float total;
	public void putsport()
	{
		System.out.println("Sports Score " +sw);
	}
	void display()
	{
		total=p1+p2+sw;
		putNumber();
		putMarks();
		putsport();
		System.out.println("Total Score = "+total);
	}
}
```

Implementing Hybrid inheritance

```
class HybridEx
{
        public static void main(String args[])
        {
                result s1 = new result();
                s1.getNumber(1234);
                s1.getMarks(27.5f,33.0f);
                s1.display();
        }
}
```

Interface and Class

An interface is similar to a class in the following ways:

An interface can contain any number of methods.

An interface is written in a file with a **.java** extension.

The bytecode of an interface appears in a **.class** file.

However, an interface is different from a class in several ways, including:
You cannot instantiate an interface.

An interface does not contain any constructors.

All of the methods in an interface are abstract.

An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
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 have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

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

super keyword

The **super** is a reference variable that is used to refer immediate parent class object.

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

Usage of super Keyword

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

super keyword with variable

- super is used to refer immediate parent class instance variable. By using super.instance variable name of parent class.

```
class Vehicle
{
 int speed=50;
}
class ex148 extends Vehicle
{
 int speed=100;
 void display()
 {
  System.out.println(super.speed);//will print speed of Vehicle
 }
 public static void main(String args[])
 {
  ex148 b=new ex148();
  b.display();

 }
}
```

super keyword with constructor

super() is used to invoke immediate parent class constructor.

You can invoke parent class default constructor using super() and parameteized constructor using super(Parameter List)

When we create Object of Sub class Default constructor of parent class is called automatically.

super keyword with constructor

```
class Vehicle
{
  Vehicle(int speed)
  {
		System.out.println("Vehicle is created "+speed);
  }
}

class ex150 extends Vehicle
{
  ex150()
  {
		super(5);//will invoke parent class constructor
		System.out.println("Bike is created");
  }
  public static void main(String args[])
  {
   ex150 b=new ex150();

  }
}
```

super keyword with method

super is used to invoke immediate parent class method.

You can call method of parent class by using super.methodName.

```
class Person
{
        void message()
        {
                System.out.println("welcome");
        }

}
```

super keyword with method

```
class ex151 extends Person
{
        void message()
        {
                System.out.println("welcome to java");
        }

        void display()
        {
                message();//will invoke current class message() method
                super.message();//will invoke parent class message() method
        }

        public static void main(String args[])
        {
                ex151 s=new ex151();
                s.display();
        }
}
```

Java Package

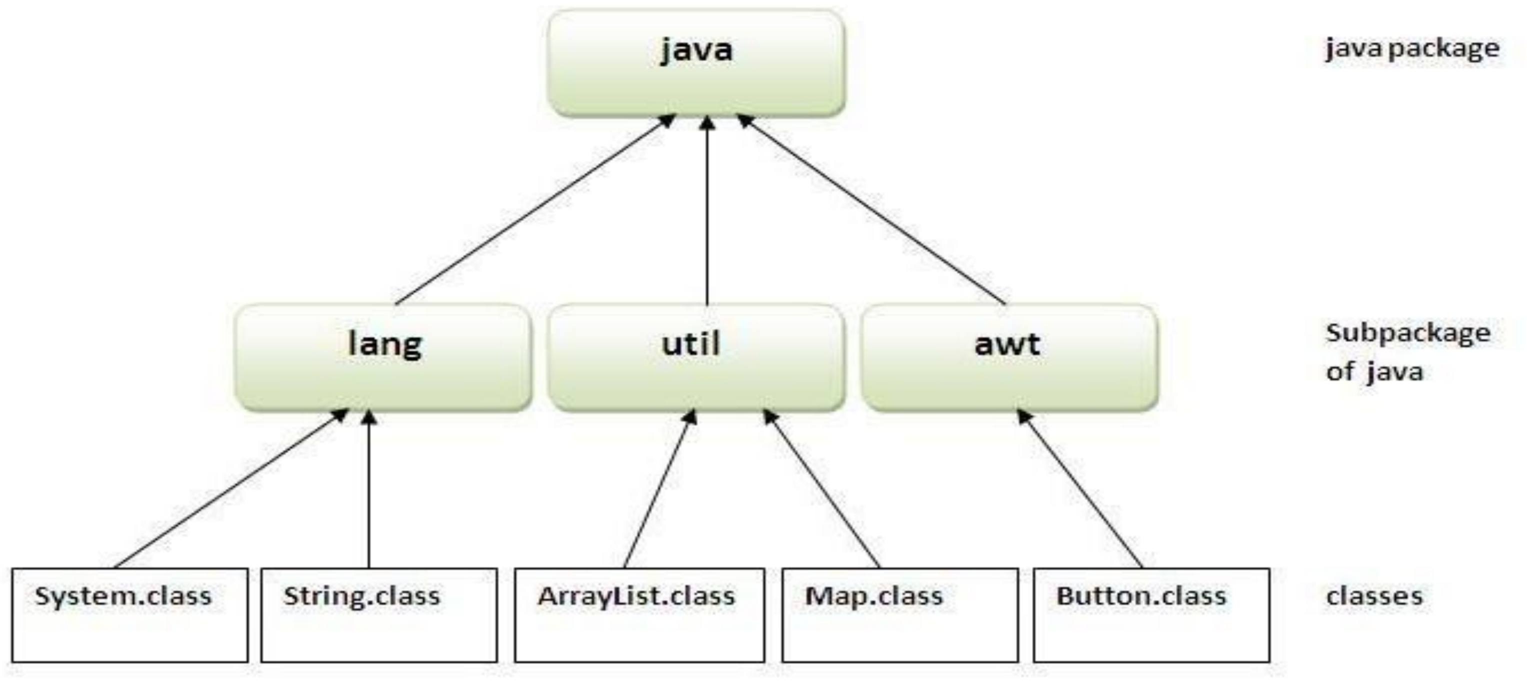
- A **java package** is a group of similar types of classes, interfaces and sub-packages.

- 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

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

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

How to compile java package

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

```
javac -d directory javafilename
```

For example

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

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

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

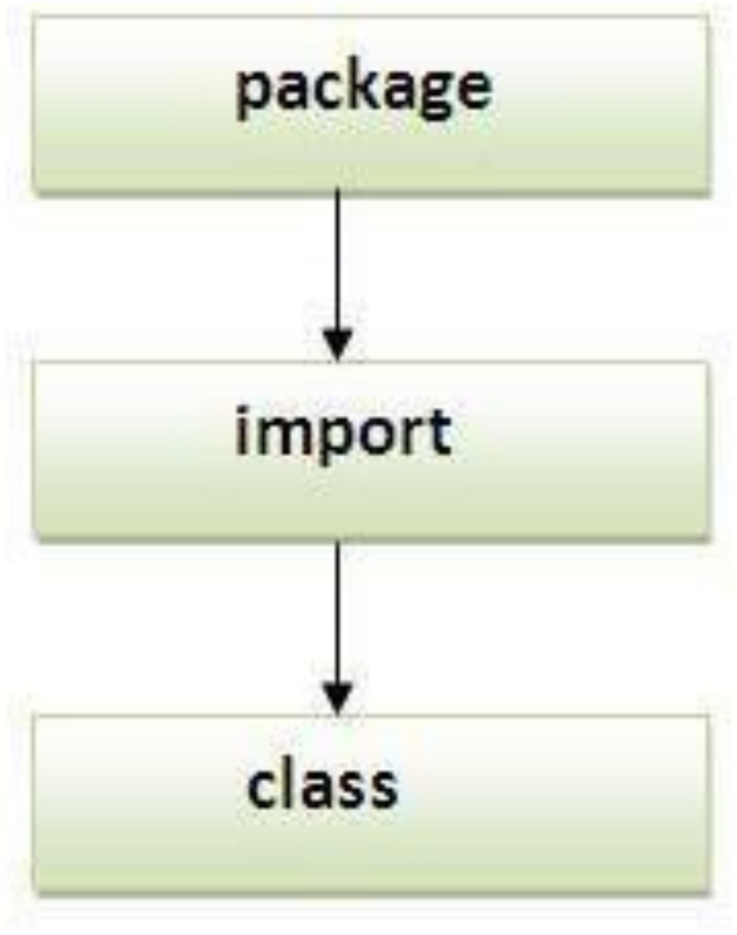
How to access package from another package?

There are three ways to access the package from outside the package

1. import package.*;
2. import package.classname;
3. fully qualified name.

Note: If you import a package, sub-packages will not be imported.

Note: Sequence of the program must be package then import then class.



Package Example

```
package mypack;
public class MyPackage
{
		public void getCityName()
		{
				System.out.println(“City is : Rajkot”);
		}
}
```

Package Example

1)Using packagename.*

```
import mypack.*;
class MainEx
{
        public static void main(String args[])
        {
                MyPackage  a=new MyPackage();
                a.getCityName();
        }
}
```

Package Example

2)Using packagename.classname

```
import mypack.MyPackage;
class MainEx
{
        public static void main(String args[])
        {
                MyPackage  a=new MyPackage();
                a.getCityName();
        }
}
```

Package Example

3) Using fully qualified name

```
class MainEx
{
        public static void main(String args[])
        {
                mypack.MyPackage  a=new mypack.MyPackage();
                a.getCityName();
        }
}
```

Subpackage in java

- Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

- Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server an d ServerSocket classes in net packages and so on.

Subpackage in java

Example of Subpackage

```
package com.javatpoint.core;

class Simple
{
    public static void main(String args[])
    {
            System.out.println("Hello subpackage");
    }
}
```

To Compile: `javac -d . Simple.java`

To Run: `java com.javatpoint.core.Simple`

Output:Hello subpackage

Access Modifiers

Class Modifiers

Keyword	Detail
abstract	Must be extended
final	Cannot be extended
public	can be accessed by any other class but if the keyword is missing then access is limited to the current package

Access Modifiers

Method Modifiers

Keyword	Detail
abstract	Must be overridden
final	Must not be overridden
native	Implemented in machine code used by the host cpu not by the Java ByteCode
private	Can be invoked only by the same class
protected	Can be invoked only by code of subclass of same package
public	Can be invoked by any other class
static	Not an instance variable
synchronized	acquires a lock when begins execution

Access Modifiers

Variable Modifiers

Keyword	Detail
final	It is a constant
private	Can be accessed only by the same class only
protected	Can be accessed only by the class in which it is declared and sub classes of the same package
public	Can be accessed by any other class
static	is not an instance variable

Static Import

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:
Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:
If you overuse the static import feature, it makes the program unreadable and unmaintainable.

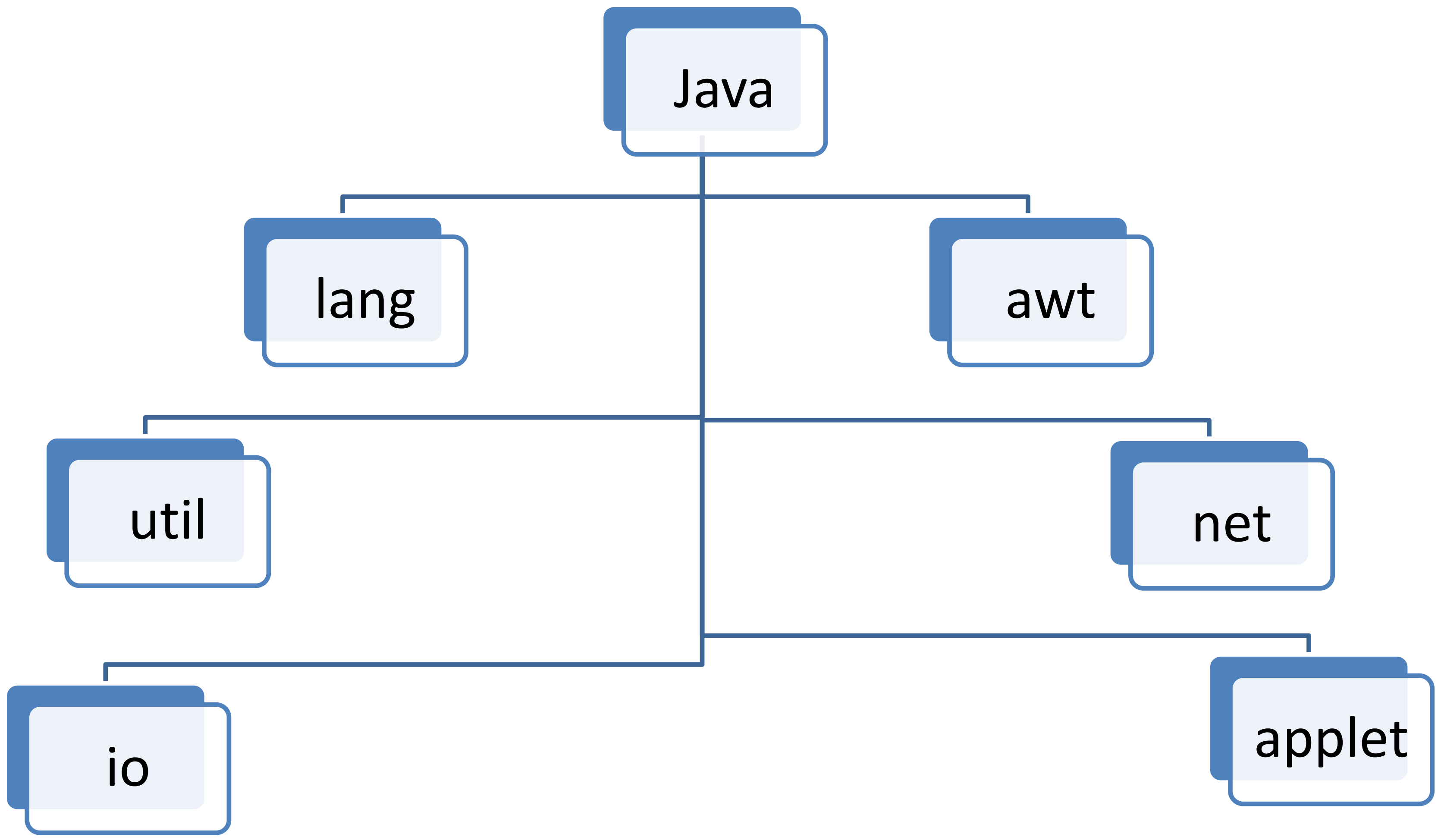
Simple Example of static import

```
import static java.lang.System.*;
class StaticImportExample
{
        public static void main(String args[])
        {
                out.println("Hello"); //Now no need of  System.out
                out.println("Java");
        }
}
```

What is the difference between import and static import?

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Java API Packages



Java API Packages

Package Name	Description
java.lang	Language Support Classes. This package imported itself and includes classes for String, Maths, Threads and Exceptions
java.util	Language utility classes such as vectors, hash tables random numbers, date etc.
java.awt	Set of classes for implementing GUI
java.io	Input output classes – Provides facility for data input and output
java.net	Classes for networking.
java.applet	Classes for creating and implementing applets.

Java String

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, substring etc.

In java, string is basically an object that represents sequence of char values.

An array of characters works same as java string. For example:

char[] ch={'j','a','v','a',‘p',‘r','o',‘g',‘r',‘a‘,’m’};

String s=**new** String(ch);

is same as:

String s=“javaprogram";

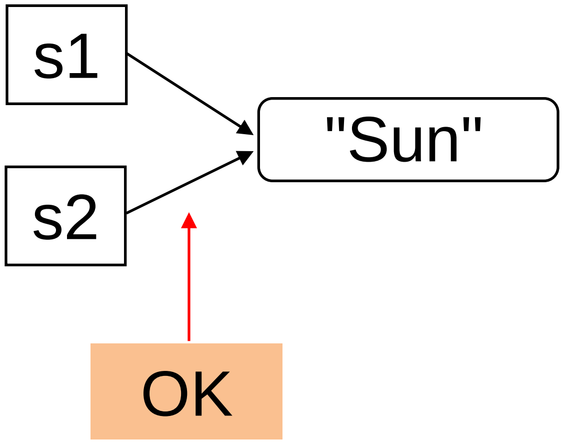
Java String

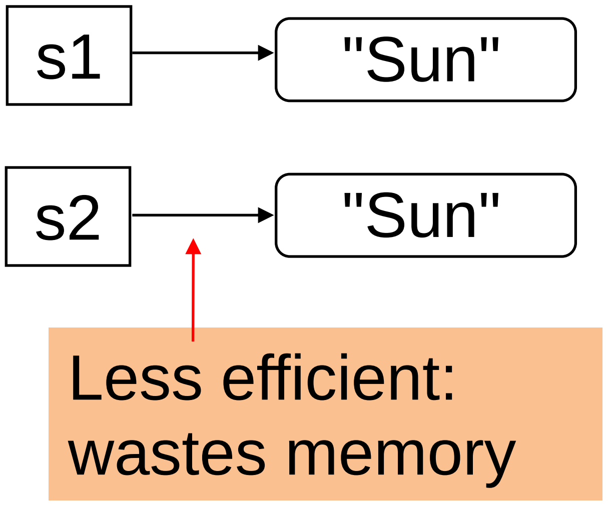
Remember that Strings in java is immutable object means that once created, a string cannot be changed: none of its methods changes the string. Such objects are called *immutable*.

Immutable objects are convenient because several references can point to the same object safely: there is no danger of changing an object through one reference without the others being aware of the change.

Advantage of using Strings

more efficient, no need to copy.

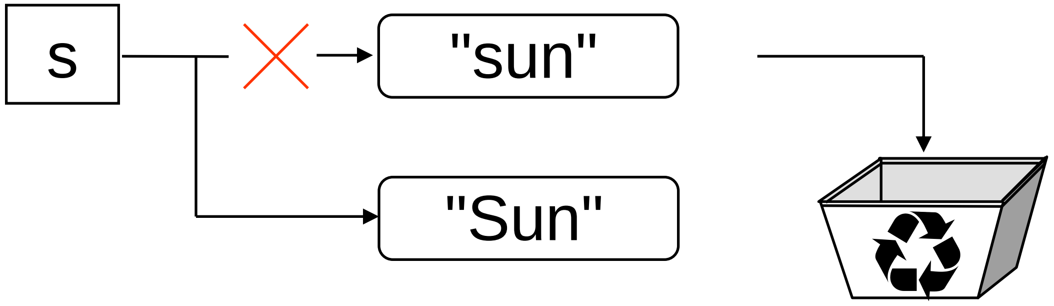




Disadvantage of using Strings

Less efficient — you need to create a new string and throw away the old one for every small change.

```
String s = "sun";
char ch = Character.toUpperCase(s.charAt (0));
s =  ch + s.substring (1);
```



Java String

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

The java String is immutable i.e. it cannot be changed but a new instance is created. For mutable class, you can use StringBuffer and StringBuilder class.

What is String in java ?

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. String class is used to create string object.

How to create String object?

There are two ways to create String object:By

1. string literal
2. By new keyword

(1) Java String literal is created by using double quotes.

For Example: String s="welcome";

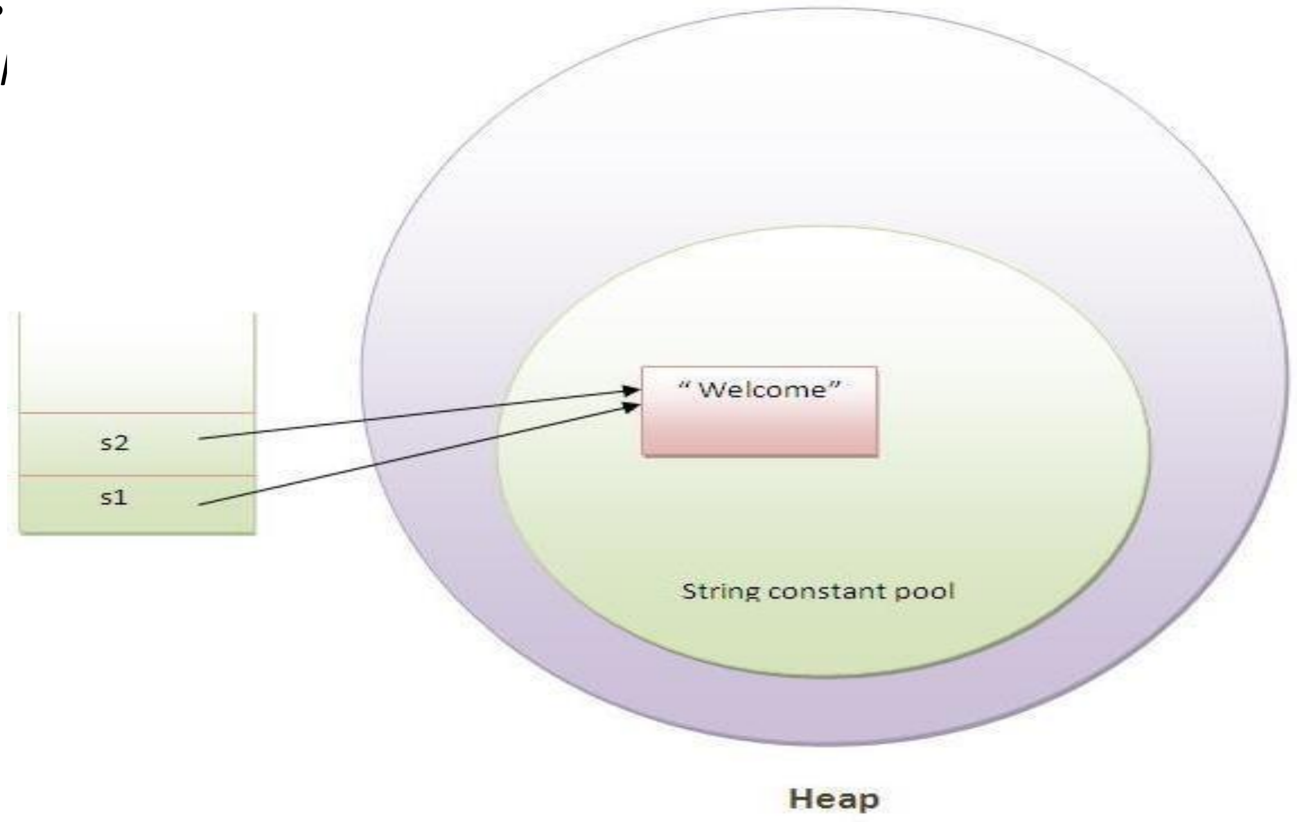
Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
String s1="Welcome";
String s2="Welcome";//will not create new instance
```

(1) Java String literal is created by using double quotes.

Note: String objects are
a special memory area
string constant pool.



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance

(1) Java String literal is created by using double quotes.

Q. Why java uses concept of string literal?

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

(2) By new keyword

- String s=**new** String("Welcome");

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

Various methods of String Class

Method	Usage	Example
length()	Returns the no. of characters in the String	String str="Computer" str.length(); Will return 8.
charAt(n)	Returns the nth character from given string. (Remember that counts from zero)	String str="Computer" str.charAt(3); Will return p.

substring()	We can use this method in two ways. (1) by passing two values starting and ending (2) by passing only starting value so that it goes upto end.	String str="Computer"
		str.substring(3,5);
		Will return pu.
		String str="Computer"
		str.substring(3);
		Will return puter

concat()	Concates two strings and stores the result in new string	String s1 = "abc" String s2 = "xyz" String s3=s1.concat(s2) Result of s3 is "abcxyz"

Various methods of String Class

Method	Usage	Example
indexOf()	Returns the position of the character from beginning or from the given position	

0 8 11 15

```
String date ="July 5, 2012 1:28:19 PM";
```

Returns:

```
date.indexOf ('J');            0
date.indexOf ('2');            8
date.indexOf ("2012");         8
date.indexOf ('2', 9);        11
```

(starts searching at position 9)

```
date.indexOf ("2020");        −1
date.lastIndexOf ('2');       15
```

(not found)

Various methods of String Class

Method	Usage	Example
equals()	Returns true if both the strings are equal.	boolean b = s1.equals(s2);
equalsIgnoreCase()	Returns true if both strings are equal (without case sensitivity)	
compareTo	Returns the difference between two strings.	
compareToIgnoreCase()	Returns the difference between two strings (ignoring case)	
trim()	Removes white spaces from both the sides of the string and return the new string.	
replace()	Replace the given character from the given string with another character.	String str="Replace Region"; System.out.println(str.replace('R','A')); Output : Aeplace Aegion.

Various methods of String Class

Method	Usage	Example
toUpperCase()	Converts given string into uppercase letters	
toLowerCase()	Converts given string into lowercase letters	

Remember that we have to write like s1 = s1.toUpperCase() then and then the string s1 will be converted to uppercase but if we just write s1.toUpperCase() then it has no effect.

Numbers to Strings

Three ways to convert a number into a string:

1.

```
String s = "" + num;
```

2.

```
String s = Integer.toString (i);
String s = Double.toString (d);
```

3.

```
String s = String.valueOf (num);
```

Integer and **Double** are "wrapper" classes from **java.lang** that represent numbers as objects. They also provide useful static methods.

Immutable String in Java

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

- Once string object is created its data or state can't be changed but a new string object is created.

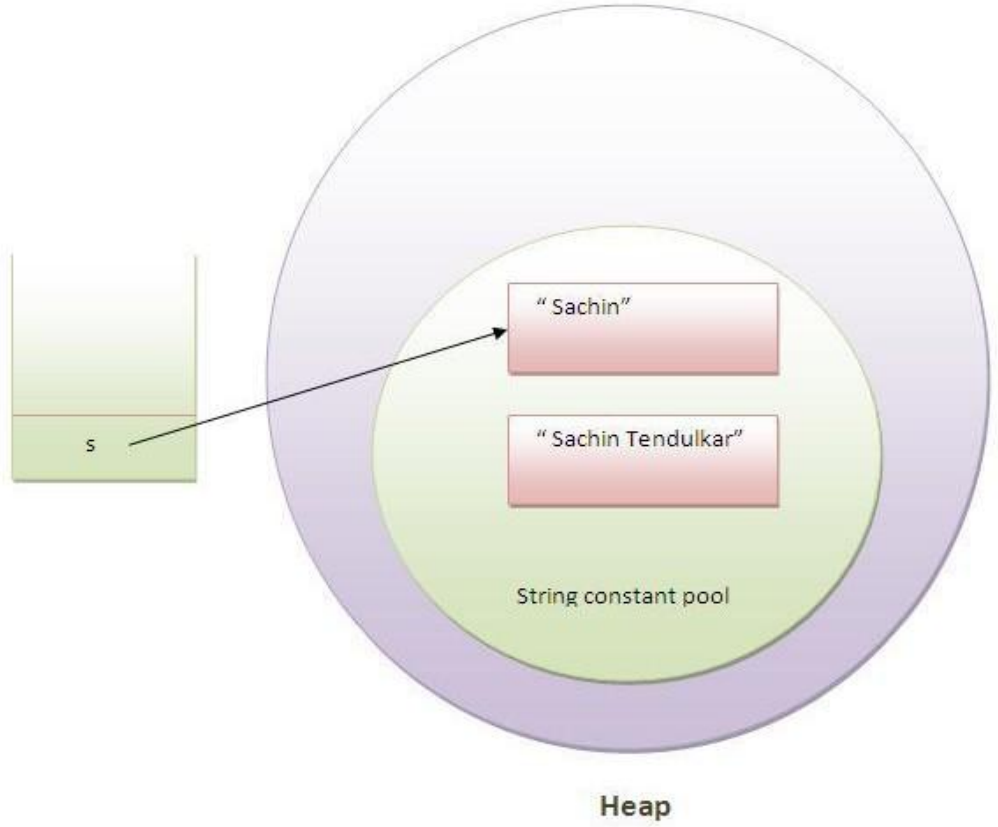
- Let's try to understand the immutability concept by the example given in next slide:

Immutable String in Java

```
class Testimmutablestring
{
    public static void main(String args[])
    {
    String s="Sachin";
    s.concat(" Tendulkar");
    //concat() method appends the  string at the end
    System.out.println(s);
    //will print Sachin because strings  are immutable objects
    }
}
```

Immutable String in Java

- Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

Immutable String in Java

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

```
class Testimmutablestring1
{
   public static void main(String args[])
   {
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
   }
}
```

Output: Sachin Tendulkar

Java StringBuffer class

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important methods of StringBuffer class

public synchronized StringBuffer append(String s): is used to append the specified string with this string.

public synchronized StringBuffer insert(int offset, String s): is used to insert the specified string with this string at the specified position.

public synchronized StringBuffer replace(int startIndex, int endIndex, String str): is used to replace the string from specified startIndex and endIndex.

Important methods of StringBuffer class

public synchronized StringBuffer delete(int startIndex, int endIndex): is used to delete the string from specified startIndex and endIndex.

public synchronized StringBuffer reverse(): is used to reverse the string.

public char charAt(int index): is used to return the character at specified index.

Important methods of StringBuffer class

public int length(): is used to return the length of the string i.e. total number of characters.

public String substring(int beginIndex): is used to return the substring from the specified beginIndex.

public String substring(int beginIndex, int endIndex): is used to return the substring from the specified beginIndex and endIndex.

Important methods of StringBuffer class

```
class ex15
{
        public static void main(String args[])
        {
                StringBuffer str=new StringBuffer("Object Language");
                System.out.println("Original String = "+str);
                System.out.println("Length of the string = "+str.length());
                for(int i=0;i<str.length();i++)
                {
                        int p=i+1;
                        System.out.println("Character At Position "+p+" is "+str.charAt(i));
                }
                String str1=new String(str.toString());
                int pos=str1.indexOf("Language");
                str.insert(pos,"Oriented ");
                System.out.println("Modified String = "+str);
                str.setCharAt(6,'-');
                System.out.println("New String = "+str);
                str.append(" Improve Security");
                System.out.println("Append String = "+str);
        }
}
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non- synchronized.

StringBuilder is basically identical to the older StringBuffer, but is slightly faster because it isn't synchronized.

Constructors

sb = new StringBuilder()

Creates empty StringBuilder

sb = new StringBuffer(n)

Creates empty StringBuilder with initial capacity n.

sb = new StringBuffer(s)

Creates StringBuilder with value initialized to String s.

It is available since JDK 1.5.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	is used to append the specified string with this string.
public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from spe

Important methods of StringBuilder class

public StringBuilder reverse()	is used to reverse the string.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified be

Java StringBuilder class

```
public class TestString
{
                public static void main(String[] args)
                {
       long startTime = System.currentTimeMillis();
       StringBuffer sb = new StringBuffer("Java");
       for (int i=0; i<100000; i++)
                                {
                sb.append("Object Oriented");
       }
       System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) +
"ms");
       startTime = System.currentTimeMillis();
       StringBuilder sb2 = new StringBuilder("Java");
       for (int i=0; i<100000; i++)
                                {
                sb2.append("Object Oriented");
       }
       System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) +
"ms");
   }
}
```

java.lang.Math Class

The **java.lang.Math** class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. In short this class is used to perform various mathematical operations.

Some functions from Math Package

Function	Action
sin(x)	returns sin of angle x
cos(x)	returns cos of angle x
tan(x)	returns tangent of angle x
asin(x)	returns the angle whose sine is x
acos(x)	returns the angle whose cosine is x
atan(x)	returns the angle whose tangent is x
pow (x,y)	returns the value of x raised to y
exp(x)	returns the value of e raised to x
log(x)	returns natural log of x
sqrt(x)	returns square root of x

Some functions from Math Package

Function	Action
ceil(x)	returns the nearest whole number less then or equal to x
floor(x)	returns the nearest whole number greater then or equal to x
round(x)	returns the integer closest to the x
abs(a)	returns absolute value of a
max(a,b)	returns maximum out of a and b
min(a,b)	returns minimum out of a and b

Thank You