String classes and its methods: String, String Buffer, String Tokenizer, Inheritance, Types of inheritance, Super keyword, Method overriding, Abstract classes, Final keyword.

Java String Class Methods

The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

Let's use some important methods of String class.

Java String toUpperCase() and toLowerCase() method

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

Stringoperation1.java

```
1. public class Stringoperation1
2. {
3. public static void main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.toUpperCase());//SACHIN
7. System.out.println(s.toLowerCase());//sachin
8. System.out.println(s);//Sachin(no change in original)
9. }
10. }
```

Test it Now

Output:

```
SACHIN
sachin
Sachin
```

Java String trim() method

The String class trim() method eliminates white spaces before and after the String.

Stringoperation2.java

```
1. public class Stringoperation2
2. {
3. public static void main(String ar[])
4. {
```

```
5. String s="  Sachin  ";
6. System.out.println(s);//  Sachin
7. System.out.println(s.trim());//Sachin
8. }
9. }
```

Test it Now

Output:

```
Sachin
Sachin
```

Java String startsWith() and endsWith() method

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Stringoperation3.java

```
1. public class Stringoperation3
2. {
3. public static void main(String ar[])
4. {
5. String s="Sachin";
6.  System.out.println(s.startsWith("Sa"));//true
7.  System.out.println(s.endsWith("n"));//true
8. }
9. }
```

Test it Now

Output:

```
true
true
```

Java String charAt() Method

The String class charAt() method returns a character at specified index.

Stringoperation4.java

```
1. public class Stringoperation4
2. {
3. public static void main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.charAt(0));//S
```

```
7. System.out.println(s.charAt(3));//h
8. }
9. }
```

Test it Now

Output:

```
S
h
```

Java String length() Method

The String class length() method returns length of the specified String.

Stringoperation5.java

```
1. public class Stringoperation5
2. {
3. public static void main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.length());//6
7. }
8. }
```

Test it Now

Output:

```
6
```

Java String intern() Method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a String equal to this String object as determined by the equals(Object) method, then the String from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

Stringoperation6.java

```
1. public class Stringoperation6
2. {
3. public static void main(String ar[])
4. {
5. String s=new String("Sachin");
6. String s2=s.intern();
7. System.out.println(s2);//Sachin
```

```
8. }
9. }
```

Test it Now

Output:

```
Sachin
```

Java String valueOf() Method

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

Stringoperation7.java

```
1. public class Stringoperation7
2. {
3. public static void main(String ar[])
4. {
5. int a=10;
6. String s=String.valueOf(a);
7. System.out.println(s+10);
8. }
9. }
```

Output:

```
1010
```

Java String replace() Method

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

Stringoperation8.java

```
1. public class Stringoperation8
2. {
3. public static void main(String ar[])
4. {
5. String s1="Java is a programming language. Java is a platform. Java is an Island.";
6. String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
7. System.out.println(replaceString);
8. }
9. }
```

Output:

Kava is a programming language. Kava is a platform. Kava is an Island.

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the 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 Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.

public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

StringBufferExample.java

```
1. class StringBufferExample{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

Output:

```
Hello Java
```

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

StringBufferExample2.java

```
1. class StringBufferExample2{
2. public static void main(String args[]){
```

```
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

Output:

```
HJavaello
```

3) StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

StringBufferExample3.java

```
1. class StringBufferExample3{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

Output:

```
HJavalo
```

4) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

StringBufferExample4.java

```
1. class StringBufferExample4{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

Output:

```
Hlo
```

5) StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

StringBufferExample5.java

```
1. class StringBufferExample5{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

Output:

```
olleH
```

6) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample6.java

```
1. class StringBufferExample6{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. }
10. }
```

Output:

```
16
16
34
```

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample7.java

```
1. class StringBufferExample7{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }
```

Output:

```
16
16
34
34
70
```

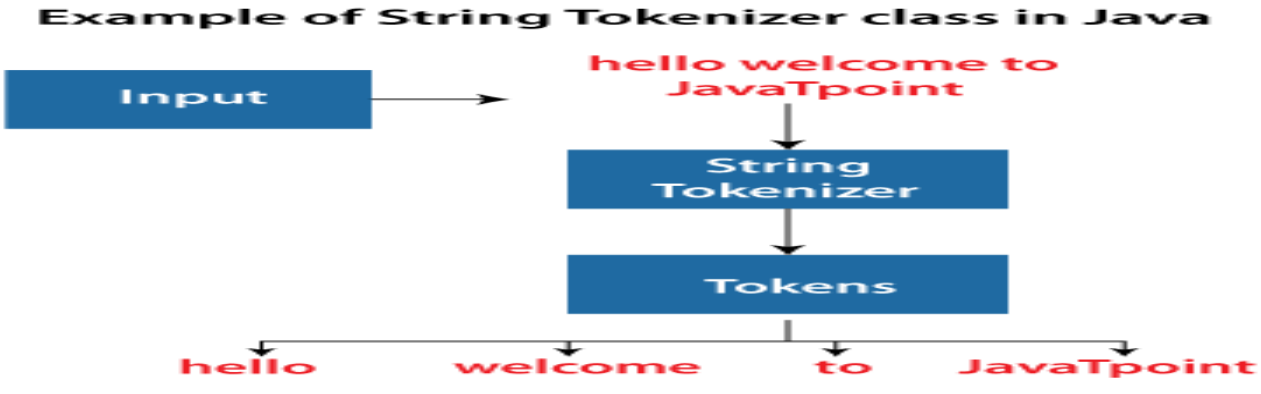
StringTokenizer in Java

1. StringTokenizer
2. Methods of StringTokenizer
3. Example of StringTokenizer

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.



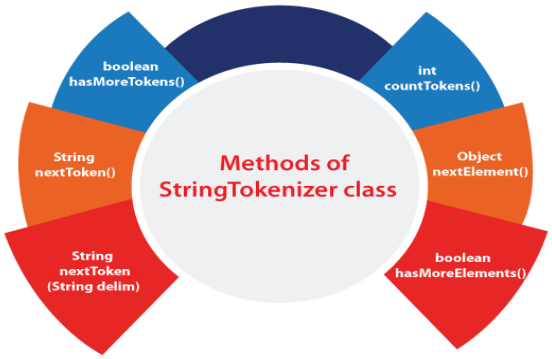
Constructors of the StringTokenizer Class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	It creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of the StringTokenizer Class

The six useful methods of the StringTokenizer class are as follows:



Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.
int countTokens()	It returns the total number of tokens.

Example of StringTokenizer Class

Let's see an example of the StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

Simple.java

```
1. import java.util.StringTokenizer;
2. public class Simple{
3.  public static void main(String args[]){
4.    StringTokenizer st = new StringTokenizer("my name is khan"," ");
5.     while (st.hasMoreTokens()) {
6.         System.out.println(st.nextToken());
7.     }
8.   }
9. }
```

Output:

```
my
name
is
khan
```

The above Java code, demonstrates the use of StringTokenizer class and its methods hasMoreTokens() and nextToken().

Example of nextToken(String delim) method of the StringTokenizer class

Test.java

```
1. import java.util.*;
2.
3. public class Test {
4.   public static void main(String[] args) {
5.       StringTokenizer st = new StringTokenizer("my,name,is,khan");
6.
7.       // printing next token
8.       System.out.println("Next token is : " + st.nextToken(","));
9.   }
10. }
```

Output:

```
Next token is : my
```

Note: The StringTokenizer class is deprecated now. It is recommended to use the split() method of the String class or the Pattern class that belongs to the java.util.regex package.

Example of hasMoreTokens() method of the StringTokenizer class

This method returns true if more tokens are available in the tokenizer String otherwise returns false.

StringTokenizer1.java

```
1. import java.util.StringTokenizer;
2. public class StringTokenizer1
3. {
4.  /* Driver Code */
5.  public static void main(String args[])
6.  {
7.    /* StringTokenizer object */
8.    StringTokenizer st = new StringTokenizer("Demonstrating methods from StringTokenizer class","
   ");
9.      /* Checks if the String has any more tokens */
10.    while (st.hasMoreTokens())
11.    {
12.        System.out.println(st.nextToken());
13.    }
14. }
15. }
```

Output:

```
Demonstrating
methods
from
StringTokenizer
class
```

The above Java program shows the use of two methods hasMoreTokens() and nextToken() of StringTokenizer class.

Example of hasMoreElements() method of the StringTokenizer class

This method returns the same value as hasMoreTokens() method of StringTokenizer class. The only difference is this class can implement the Enumeration interface.

StringTokenizer2.java

```
1. import java.util.StringTokenizer;
2. public class StringTokenizer2
3. {
4.  public static void main(String args[])
5.  {
6.    StringTokenizer st = new StringTokenizer("Hello everyone I am a Java developer"," ");
7.     while (st.hasMoreElements())
8.     {
9.        System.out.println(st.nextToken());
10.    }
```

```
    }
}
```

Output:

```
Hello
everyone
I
am
a
Java
developer
```

The above code demonstrates the use of hasMoreElements() method.

Example of nextElement() method of the StringTokenizer class

nextElement() returns the next token object in the tokenizer String. It can implement Enumeration interface.

StringTokenizer3.java

```
import java.util.StringTokenizer;
public class StringTokenizer3
{
 /* Driver Code */
 public static void main(String args[])
 {
  /* StringTokenizer object */
  StringTokenizer st = new StringTokenizer("Hello Everyone Have a nice day"," ");
   /* Checks if the String has any more tokens */
   while (st.hasMoreTokens())
   {
      /* Prints the elements from the String */
      System.out.println(st.nextElement());
   }
}
}
```

Output:

```
Hello
Everyone
Have
a
nice
day
```

The above code demonstrates the use of nextElement() method.

Example of countTokens() method of the StringTokenizer class

This method calculates the number of tokens present in the tokenizer String.

StringTokenizer4.java

```
1. import java.util.StringTokenizer;
2. public class StringTokenizer3
3. {
4.  /* Driver Code */
5.  public static void main(String args[])
6.  {
7.    /* StringTokenizer object */
8.    StringTokenizer st = new StringTokenizer("Hello Everyone Have a nice day"," ");
9.          /* Prints the number of tokens present in the String */
10.         System.out.println("Total number of Tokens: "+st.countTokens());
11. }
12. }
```

Output:

```
Total number of Tokens: 6
```

The above Java code demonstrates the countTokens() method of StringTokenizer() class.

Inheritance in Java

1. Inheritance
2. Types of Inheritance
3. Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

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.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

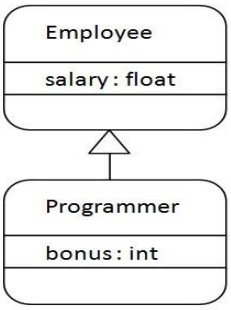
The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
4. class Programmer extends Employee{
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

Test it Now

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

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

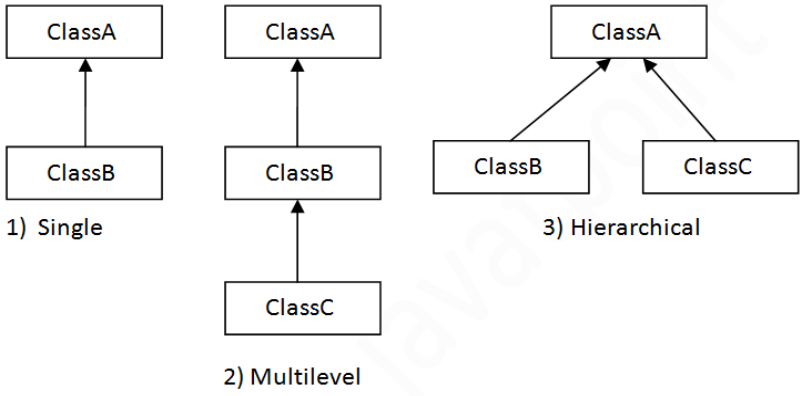
Types of inheritance in java

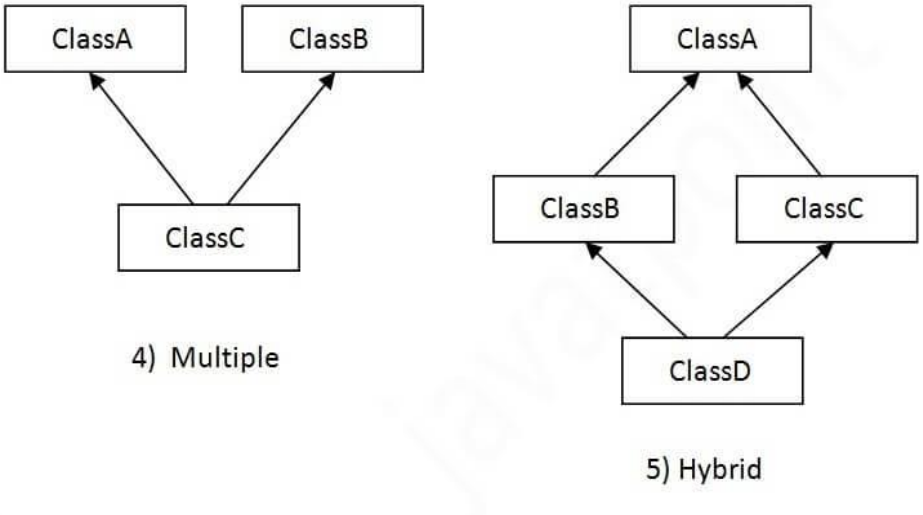
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:





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.

File: TestInheritance.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class TestInheritance{
8.  public static void main(String args[]){
9.  Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

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.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

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.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
```

```
8.  void meow(){System.out.println("meowing...");}
9.  }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

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

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 the same method and you call it from child class object, there will be ambiguity to call the 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.

```
1.  class A{
2.  void msg(){System.out.println("Hello");}
3.  }
4.  class B{
5.  void msg(){System.out.println("Welcome");}
6.  }
7.  class C extends A,B{//suppose if it were
8.
9.   public static void main(String args[]){
10.   C obj=new C();
11.   obj.msg();//Now which msg() method would be invoked?
12. }
13. }
```

Test it Now

```
Compile Time Error
```

Super Keyword in Java

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

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

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
11. class TestSuper1{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Test it Now

Output:

```
black
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.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog d=new Dog();
15. d.work();
16. }}
```

Test it Now

Output:

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

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Test it Now

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. class TestSuper4{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Test it Now

Output:

```
animal is created
dog is created
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. }
8. }
```

```
9. class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

Test it Now

Output:

```
1 ankit 45000
```

Method Overriding in Java

1. Understanding the problem without method overriding
2. Can we override the static method
3. Method overloading vs. method overriding

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

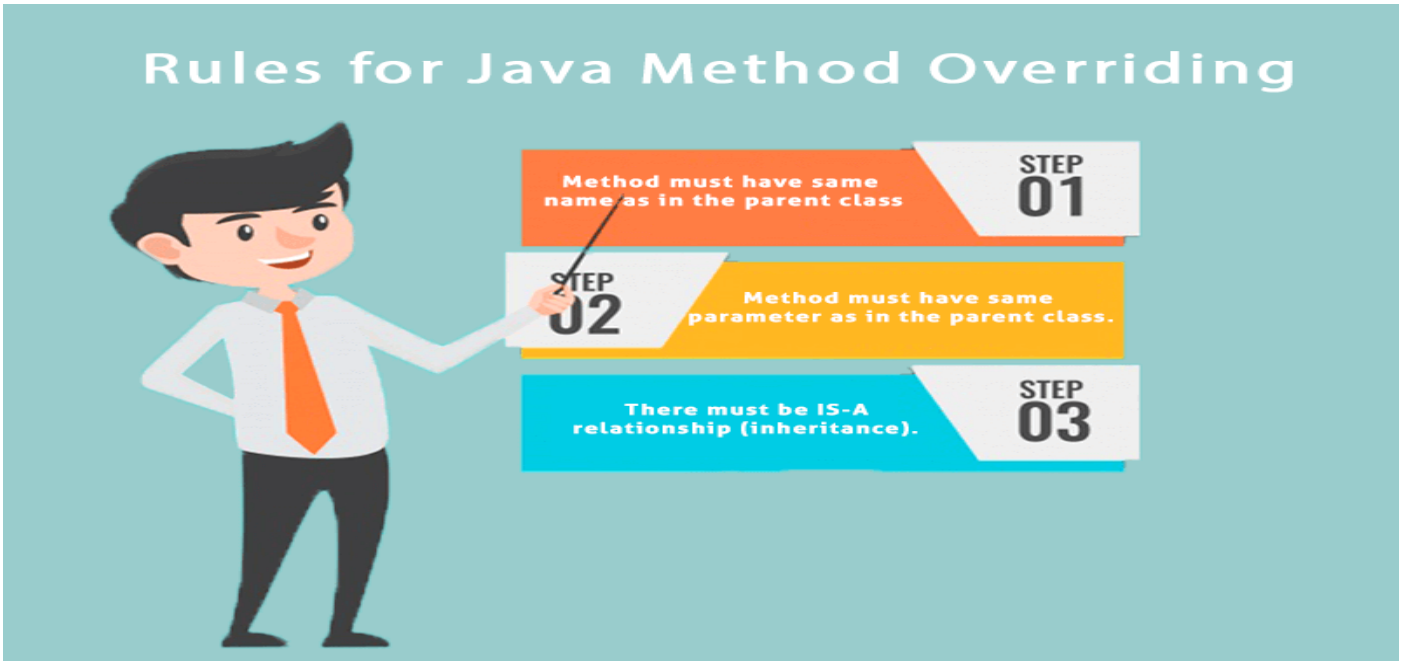
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

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).



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10.  public static void main(String args[]){
11.  //creating an instance of child class
12.  Bike obj = new Bike();
13.  //calling the method with child class instance
14.  obj.run();
15.  }
16. }
```

Test it Now

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

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

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. class Bike2 extends Vehicle{
9.   //defining the same method as in the parent class
10.  void run(){System.out.println("Bike is running safely");}
11.
12.  public static void main(String args[]){
13.  Bike2 obj = new Bike2();//creating object
14.  obj.run();//calling method
15.  }
16. }
```

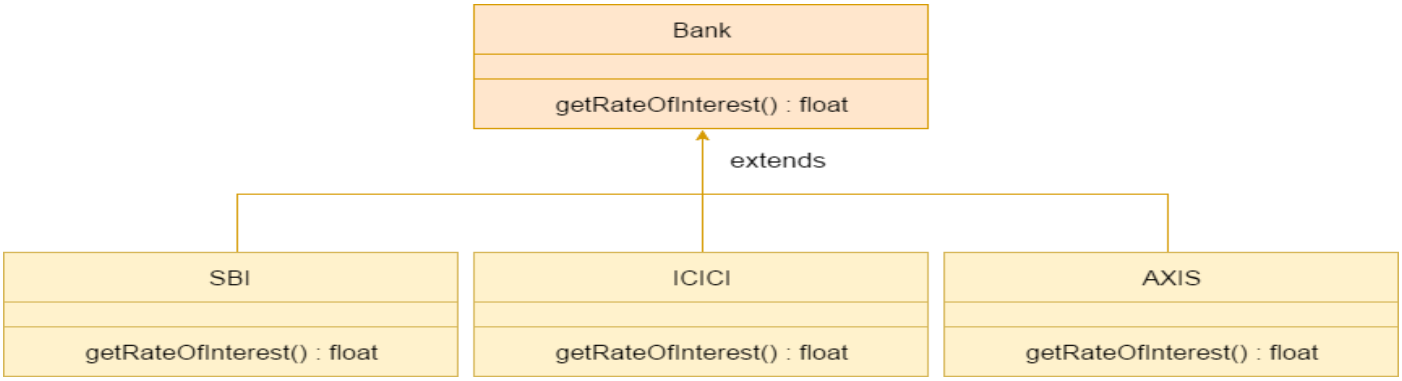
Test it Now

Output:

```
Bike is running safely
```

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
1. //Java Program to demonstrate the real scenario of Java Method Overriding
2. //where three classes are overriding the method of a parent class.
3. //Creating a parent class.
4. class Bank{
5. int getRateOfInterest(){return 0;}
6. }
7. //Creating child classes.
8. class SBI extends Bank{
9. int getRateOfInterest(){return 8;}
10. }
11.
12. class ICICI extends Bank{
13. int getRateOfInterest(){return 7;}
14. }
15. class AXIS extends Bank{
16. int getRateOfInterest(){return 9;}
17. }
18. //Test class to create objects and call the methods
19. class Test2{
20. public static void main(String args[]){
21. SBI s=new SBI();
22. ICICI i=new ICICI();
23. AXIS a=new AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }
```

Test it Now

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Abstract class in Java

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).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

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.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

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

```
1. abstract class A{}
```

Abstract Method in Java

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

Example of abstract method

```
1. abstract void printStatus();//no method body and abstract
```

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.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7.  Bike obj = new Honda4();
8.  obj.run();
9. }
10. }
```

Test it Now

```
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. class TestAbstraction1{
13. public static void main(String args[]){
14. Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() meth
    od
```

```
15. s.draw();
16. }
17. }
```

Test it Now

```
drawing circle
```

Another example of Abstract class in java

File: TestBank.java

```
1.  abstract class Bank{
2.  abstract int getRateOfInterest();
3.  }
4.  class SBI extends Bank{
5.  int getRateOfInterest(){return 7;}
6.  }
7.  class PNB extends Bank{
8.  int getRateOfInterest(){return 8;}
9.  }
10.
11. class TestBank{
12. public static void main(String args[]){
13. Bank b;
14. b=new SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=new PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}
```

Test it Now

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
1.  //Example of an abstract class that has abstract and non-abstract methods
2.   abstract class Bike{
3.     Bike(){System.out.println("bike is created");}
4.     abstract void run();
```

```
5.    void changeGear(){System.out.println("gear changed");}
6.   }
7.  //Creating a Child class which inherits Abstract class
8.   class Honda extends Bike{
9.   void run(){System.out.println("running safely..");}
10.  }
11. //Creating a Test class which calls abstract and non-abstract methods
12.  class TestAbstraction2{
13.  public static void main(String args[]){
14.   Bike obj = new Honda();
15.   obj.run();
16.   obj.changeGear();
17.  }
18. }
```

Test it Now

```
bike is created
running safely..
gear changed
```

Rule: If there is an abstract method in a class, that class must be abstract.

```
1.  class Bike12{
2.  abstract void run();
3.  }
```

Test it Now

```
compile time error
```

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1.  interface A{
2.  void a();
3.  void b();
4.  void c();
5.  void d();
6.  }
7.
8.  abstract class B implements A{
```

```
9.  public void c(){System.out.println("I am c");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

Test it Now

```
Output:I am a
       I am b
       I am c
       I am d
```

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:

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. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

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

Example of final variable

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

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  }
10. }//end of class
```

Test it Now

Output:Compile Time Error

---

2) Java final method

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

Example of final method

```
1. class Bike{
2.   final void run(){System.out.println("running");}
3. }
4.
5. class Honda extends Bike{
6.    void run(){System.out.println("running safely with 100kmph");}
7.
8.    public static void main(String args[]){
9.    Honda honda= new Honda();
10.   honda.run();
11.   }
12. }
```

Test it Now

Output:Compile Time Error

---

3) Java final class

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

Example of final class

```
1. final class Bike{}
2.
3. class Honda1 extends Bike{
4.   void run(){System.out.println("running safely with 100kmph");}
5.
6.   public static void main(String args[]){
7.   Honda1 honda= new Honda1();
8.   honda.run();
9.   }
10. }
```

Test it Now

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2.   final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5.    public static void main(String args[]){
6.     new Honda2().run();
7.    }
8. }
```

Test it Now

Output:running...

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

```
1. class Student{
2. int id;
3. String name;
4. final String PAN_CARD_NUMBER;
```

```
5. ...
6. }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1. class Bike10{
2.   final int speedlimit;//blank final variable
3.
4.   Bike10(){
5.   speedlimit=70;
6.   System.out.println(speedlimit);
7.   }
8.
9.   public static void main(String args[]){
10.    new Bike10();
11.  }
12. }
```

Test it Now

Output: 70

---

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1. class A{
2.   static final int data;//static blank final variable
3.   static{ data=50;}
4.   public static void main(String args[]){
5.     System.out.println(A.data);
6.   }
7. }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1. class Bike11{
2.   int cube(final int n){
```

```
3.      n=n+2;//can't be changed as n is final
4.      n*n*n;
5.     }
6.     public static void main(String args[]){
7.       Bike11 b=new Bike11();
8.       b.cube(5);
9.    }
10. }
```

Test it Now

Output: Compile Time Error