UNIT II

Inheritance and Interfaces

Inheritance-Interfaces-Packages and Java Library-String Handling

INHERITANCE

- ❖ Inheritance is the mechanism in java by which one class is allow to inherit the features (fields and methods) of another class.
- ❖ It is process of deriving a new class from an existing class.
- ❖ A class that is inherited is called a *superclass* and the class that does the inheriting is called a *subclass.*
- ❖ Inheritance represents the IS-A relationship, also known as *parent-child* relationship.
- ❖ **The keyword used for inheritance is extends**.

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

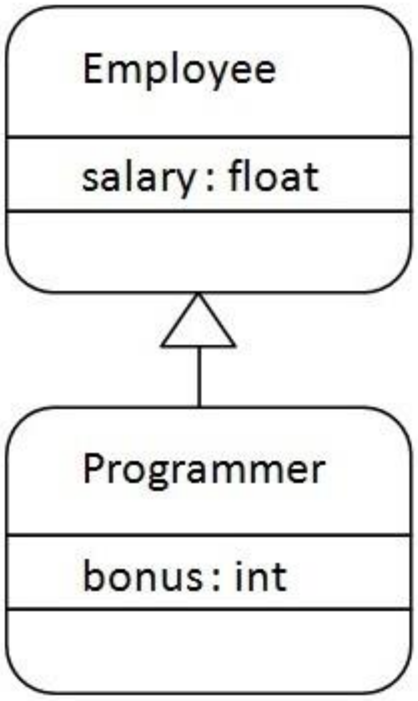
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

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

Advantages of Inheritance:

- Code reusability - public methods of base class can be reused in derived classes
- Data hiding – private data of base class cannot be altered by derived class
- Overriding- With inheritance, we will be able to override the methods of the base class in the derived class

Java Inheritance Example



As displayed in the 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.

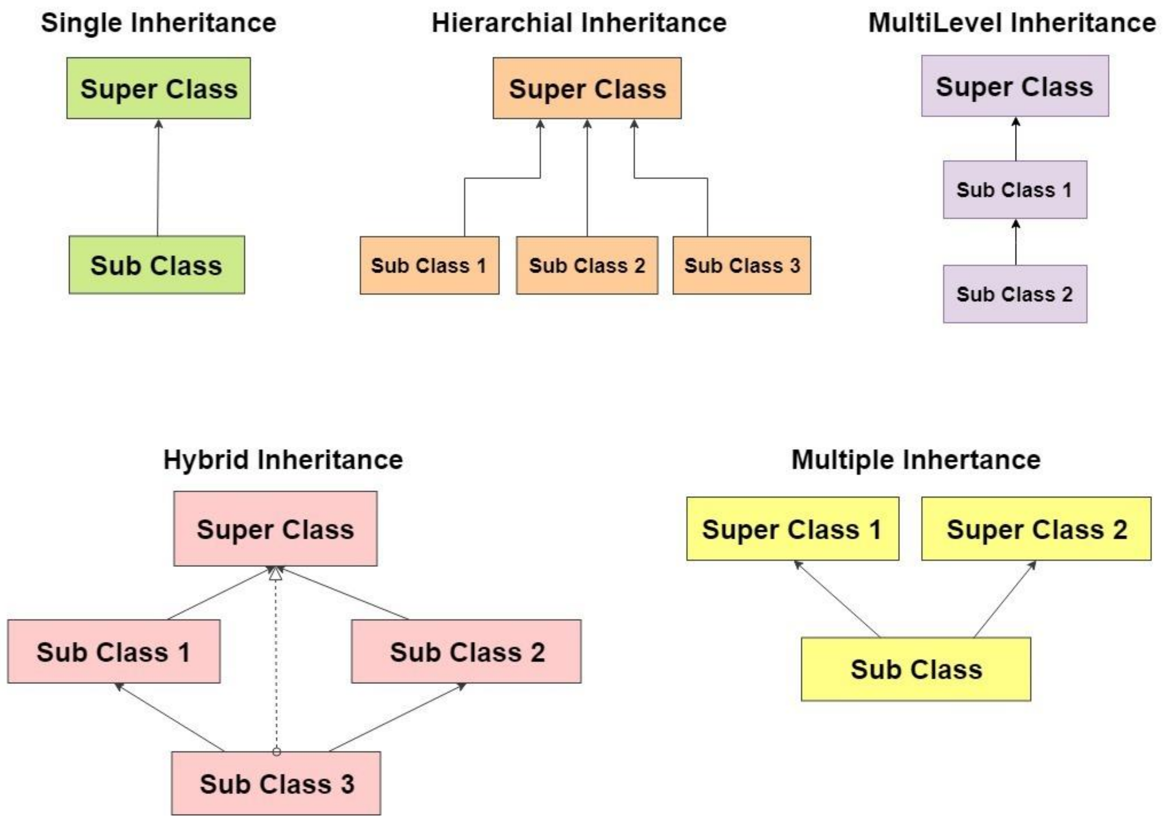
Example:

```
// Create a superclass.
class BaseClass
{
 int a=10,b=20;
 public void add()
{
System.out.println(“Sum:”+(a+b));
}
}
// Create a subclass by extending class BaseClass.

public class Main extends BaseClass
{
public void sub()
{
System.out.println(“Difference:”+(a-b));
}
}
Public class inheritanceex
{
public static void main(String[] args)
{
Main obj=new Main();
obj.add();
obj.sub();
}
}
```

Sample Output:

```
Sum:30
Difference:-10
```

Types of inheritance



Single Inheritance :

In single inheritance, a subclass inherit the features of one superclass.

Example:

Dog class inherits the Animal class, so there is the single inheritance.

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

Output:

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

Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class i.e. a derived class in turn acts as a base class for another class.

Example:

BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

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

Output:

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

Hierarchical Inheritance:

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Example:

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

Output:

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

```
//c.bark();//C.T.Error  } }
```

Multiple inheritance

Java does not allow multiple inheritance:

- To reduce the complexity and simplify the language
- To avoid the ambiguity caused by multiple inheritance

Why multiple inheritance is not supported in java?

- ❖ 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.

```
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?
}
}
```

Output

```
 Compile Time Error
```

Method Overriding in Java Inheritance

- we see the object of the subclass can access the method of the superclass.
- **However, if the same method is present in both the superclass and subclass, what will happen?**
- In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Java.

Example:

```
class Animal
{
  // method in the superclass

  public void eat()
 {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal

class Dog extends Animal
 {
  // overriding the eat() method
  public void eat()
 {
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark()
{
    System.out.println("I can bark");
  }
}

class Main
 {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

Output

```
I eat dog food
I can bark
```

❖ In the above example, the eat() method is present in both the superclass Animal and the subclass Dog.Here, we have created an object labrador of Dog.

- ❖ Now when we call eat() using the object labrador, the method inside Dog is called. This is because the method inside the derived class overrides the method inside the base class.
- ❖ This is called method overriding.

using super

The super keyword refers to 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.

- It can be used to refer immediate parent class instance variable when both parent and child class have member with same name
- It can be used to invoke immediate parent class method when child class has overridden that method.
- super() can be used to invoke immediate parent class constructor.

Use of super with variables:

When both parent and child class have member with same name, we can use super keyword to access mamber of parent class.

Example:

```
class SuperCls
{
 int x = 20;
}
 /* subclass  SubCls extending SuperCls */
class SubCls extends SuperCls
{
  int x = 80;
void display()
{
System.out.println("Super Class x: " + super.x); //print x of super class
System.out.println("Sub Class x: " + x);  //print x of subclass
}
}
 /* Driver program to test */
class Main
 {
    public static void main(String[] args)
    {
        SubCls obj = new SubCls();
       obj.display();
    }
}
```

sample Output:

```
Super Class x: 20
Sub Class x: 80
```

Use of super with methods:

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 (Method Overriding).

- super keyword is used to call the method of the parent class from the method of the child class.

Super Keyword in Inheritance

```
class Animal
{
  // method in the superclass
  public void eat()
{
    System.out.println("I can eat");
  }
}
// Dog inherits Animal
class Dog extends Animal
{
  // overriding the eat() method
  @Override
  public void eat()
{
    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }
  // new method in subclass
  public void bark()
{
    System.out.println("I can bark");
  }
}
class Main
 {
  public static void main(String[] args)
{
    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

Output

```
I can eat
I eat dog food
I can bark
```

- In the above example, the eat() method is present in both the base class Animal and the derived class Dog.

```
super.eat();
```

- Here, the super keyword is used to call the eat() method present in the superclass.use the super keyword to call the constructor of the superclass from the constructor of the subclass.

Use of super with constructors:

The super keyword can also be used to invoke the parent class constructor.

Syntax:

```
super();
```

- super() if present, must always be the first statement executed inside a subclass constructor.
- When we invoke a super() statement from within a subclass constructor, we are invoking the immediate super class constructor

Example:

```
    class SuperCls
    {
    SuperCls()
   {
      System.out.println("In Super Constructor");
 }

class SubCls extends SuperCls

    {

      SubCls()
    {
           super();
           System.out.println("In Sub Constructor");
       }

     }

    }

    /* Driver program to test */

    class Main

     {

       public static void main(String[] args)
       {

           SubCls obj = new SubCls();
       }
    }
```

Output:

```
In Super Constructor
In Sub Constructor
```

ABSTRACT CLASSES AND METHODS

- 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.

Rules

- 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.
- The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes).
- We use the abstract keyword to declare an abstract class.

Syntax

```
// create an abstract class
abstract class classname
{
  // fields and methods
}
```

- For example: both the regular methods and abstract methods

```
abstract class Language
{
  // abstract method
  abstract void method1();
  // regular method
  void method2()
{
    System.out.println("This is regular method");
  }
}
```

Abstract Method

- A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods.

  Note:

  A normal class (non-abstract class) cannot have abstract methods.

  Syntax:

  ```
  abstract returntype functionname (); //No definition
  ```

For example,

```
abstract void display();
```

- Here, display() is an abstract method. The body of display() is replaced by ;.
- If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// class should be abstract
class Language
 {
  // abstract method
  abstract void method1();
}
```

syntax for abstract class and method: modifier

```
abstract class className
{    //declare fields
    //declare methods
    abstract dataType methodName();
}
modifier class childClass extends className
{
dataType methodName()
{
}
}
```

Rules

1. Abstract classes are not Interfaces.
2. An abstract class may have concrete (complete) methods.
3. An abstract class may or may not have an abstract method. But if any class has one or more abstract methods, it must be compulsorily labeled abstract.
4. Abstract classes can have Constructors, Member variables and Normal methods.
5. Abstract classes are never instantiated.
6. For design purpose, a class can be declared abstract even if it does not contain any abstract methods.
7. Reference of an abstract class can point to objects of its sub-classes thereby achieving run-time polymorphism Ex: Shape obj = new Rectangle();
8. A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

example 1

```
//abstract parent class
abstract class Animal
{
  //abstract method
  public abstract void sound();
}
//Lion class extends Animal class
public class Lion extends Animal
{
  public void sound()
    {
    System.out.println("Roars");
  }
  public static void main(String args[])
{
Animal obj = new Lion();
obj.sound();
}
}
```

Output:

```
Roars
```

In the above code, Animal is an abstract class and Lion is a concrete class.

example 2

```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
```

```
{
int getRateOfInterest()
{
   return 7;
}
}
class PNB extends Bank
{
int getRateOfInterest()
{
   return 8;
}
}
public class TestBank
{
public static void main(String args[])
{
Bank b=new SBI();   //if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println("Rate   of   Interest   is:   "+interest+"   %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

Output:

```
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.

```
//Example of an abstract class that has abstract and non-abstract methods
 abstract class Bike
{
   Bike()
{
 System.out.println("bike is created");}
   abstract void run();
   void changeGear()
{
System.out.println("gear changed");
}
}
//Creating a Child class which inherits Abstract class
 class Honda extends Bike
{
 void run()
{
System.out.println("running safely..");
}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2
{
 public static void main(String args[])
{
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Output

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

FINAL METHODS AND CLASSES

The final keyword in java is used to restrict the user. The java final keyword can be applied to

- variable
- method
- class

Java final variable	-	To prevent constant variables
Java final method	-	To prevent method overriding
Java final class	-	To prevent inheritance

JAVA FINAL VARIABLE:

- 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.

Example of final variable

A final variable speedlimit is defined within a class Vehicle, when you try to change the value of this variable,we get an error. It can't be changed because final variable once assigned a value can never be changed.

```
public class Vehicle
{
 final int speedlimit=60;   //final variable
void run()
 {
speedlimit=400;
  }
 public static void main(String args[])
 {
   Vehicle obj=new  Vehicle();
   obj.run();
 }
 }
```

Output: /Vehicle.java:6: error: cannot assign a value to final variable speedlimit

```
 speedlimit=400;

^1 error
```

java final Method:

A Java method with the final keyword is called a final method and it cannot be overridden In general, final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded at compile time in the subclass.

Sample Code:

```
class XYZ
    {
      final void demo()
      {
    System.out.println(“XYZ Class Method”);
      }
    }
    public class ABC extends XYZ
    {
    void demo()
      {
    System.out.println(“ABC Class Method”);
     }
    public static void main(String args[])
      {
          ABC obj= new ABC();
          obj.demo();
        }
     }
```

Output:

```
    /ABC.java:11: error: demo() in ABC cannot override demo() in XYZ
    void demo()
          ^
     overridden method is final  1 error
```

JAVA FINAL CLASS

- Final class is a class that cannot be extended i.e. it cannot be inherited.
- A final class can be a subclass but not a superclass.
- Declaring a class as final implicitly declares all of its methods as final.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Several classes in Java are final e.g. String, Integer, and other wrapper classes.
- The final keyword can be placed either before or after the access specifier.

Syntax:

```
final public class A

{

    //code

}
```

OR

```
public final class A

{

    //code

}
```

Example of final class

```
final class Bike
{
}
 class Honda1 extends Bike
{
  void run()
{
System.out.println("running safely with 100kmph");
}
```

```
  public static void main(String args[])
{
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output: Compile Time Error

INTERFACES

- ❖ An **interface in Java** is a blueprint of a class. It is similar to class. It is a collection of abstract methods.
- ❖ The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- ❖ Java Interface also **represents the IS-A relationship**.

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, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Uses of interface:

- Since java does not support multiple inheritance in case of class, it can be achieved by using interface.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

Defining an Interface

An interface is defined much like a class.

Syntax:

```
accessspecifier interface interfacename
{
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
 type final-varnameN = value;
}
```

Sample Code:

The following code declares a simple interface Animal that contains two methods called eat() and travel() that take no parameter. /* File name : Animal.java */

```
 interface Animal
{
public void eat();
public void travel();
}
```

Implementing an Interface

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the ‘implements’ clause is included in a class definition and then the methods defined by the interface are created.

Syntax:

```
class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

properties of java interface

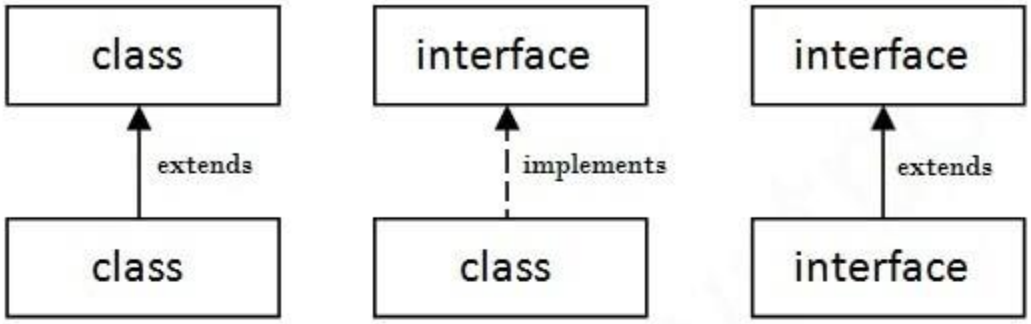
- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public.
- The type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

rules

- A class can implement more than one interface at a time.
- A class can extend only one class, but can implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

The relationship between classes and interfaces

A class extends another class, an interface extends another interface, but a **class implements an interface**.



Sample Code 1:

The following code implements an interface Animal shown earlier.

```
/* File name : MammalInt.java */

public class Mammal implements Animal
{
  public void eat()
{
System.out.println("Mammal eats");
 }
  public void travel()
{
    System.out.println("Mammal travels");
  }
  public int noOfLegs()
{
    return 0;
 }
  public static void main(String args[])
{
    Mammal m = new Mammal();
    m.eat();
    m.travel();
  }
}
```

Output:

```
Mammal eats
Mammal travels
```

Rectangle and Circle classes.

In this example,Drawable interface has only one method.its implementation is provided by Rectangle and circle classes.

```
interface Drawable
{
void draw();
}
class Rectangle implements Drawable
{
public void draw()
{
   System.out.println("Drawing rectangle");
}
}

class Circle implements Drawable

{

public void draw()

{    System.out.println("Drawing circle");

}

}

public class TestInterface

{

public static void main(String args[])

{

Drawable d=new Circle();

d.draw();

}

}
```

Output:

```
Drawing circle
```

nested Interface

Example

Here we make interface as Shape with two methods as input() and area() which are implemented by further two classes as circle and rectangle who implements the interface Shape.

```
interface Shape
{
   void input();
   void area();
}
```

```
class Circle implements Shape
{
    int r = 0;
    double pi = 3.14, ar = 0;
    @Override
    public void input()
    {
        r = 5;
    }
    @Override
    public void area()
    {
        ar = pi * r * r;
        System.out.println("Area of circle:"+ar);
    }
}
class Rectangle extends Circle
{
    int l = 0, b = 0;
    double ar;
    public void input()
    {
        super.input();
        l = 6;
        b = 4;
    }
    public void area()
    {
        super.area();
        ar = l * b;
        System.out.println("Area of rectangle:"+ar);
    }
}
public class Demo
{
    public static void main(String[] args)
    {
        Rectangle obj = new Rectangle();
        obj.input();
        obj.area();
    }
}
```

Output:

```
$ javac Demo.java
$ java  Demo

Area of circle:78.5
Area of rectangle:24.0
```

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A
 {
  // members of A
}

interface B
{
  // members of B
}

class C implements A, B
{
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces.

For example,

```
interface Line
{
  // members of Line interface
}

// extending interface
interface Polygon extends Line
{
  // members of Polygon interface
  // members of Line interface
}
```

- Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

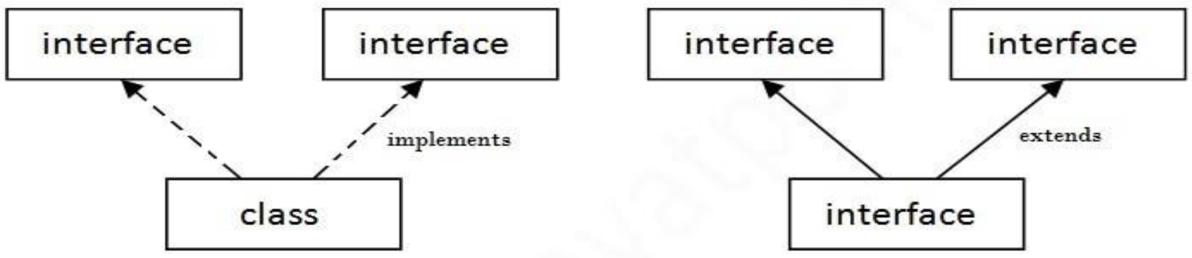
Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A
{
   ...
}
interface B
{
   ...
}
interface C extends A, B
{
   ...
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable
{
void print();
}
interface Showable
{
void show();
}
class A7 implements Printable,Showable
{
public void print()
{
System.out.println("Hello");
}
public void show()
{
System.out.println("Welcome");
}
public static void main(String args[])
{
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

Nested Interface in Java

An interface can have another interface which is known as a nested interface. For example:

```
interface printable
{
 void print();
 interface MessagePrintable
{
   void msg();
 }
}
```

PACKAGES

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.

Defining a Package

To create a package include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If **package** statement is omitted, the class names are put into the default package, which has no name.

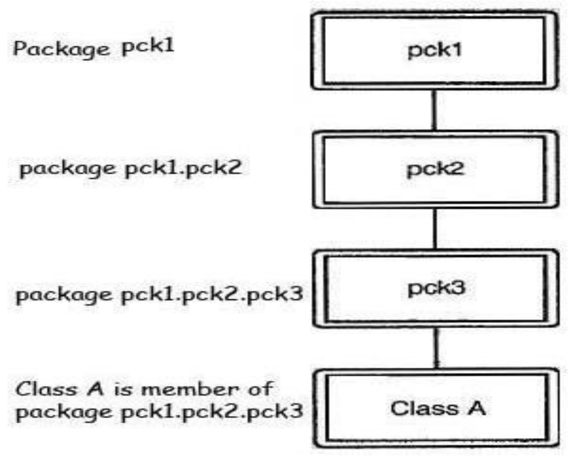
Syntax:

```
package <fully qualified package name>;
```

package *pkg*;

- Here, *pkg* is the name of the package. For example, the following statement creates a package called MyPackage. **package MyPackage;**
- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

It is possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here: **package pkg1[.pkg2[.pkg3]];**



Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

Third, you can use the **classpath** option with **java** and **javac** to specify the path to your classes.

Example:

// A simple package

```
 package MyPack;
class Balance
 {
 String name;
 double bal;
Balance(String n, double b)
 {
 name = n;
 bal = b;
 }
 void show()
{
 if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance
{
public static void main(String args[])
 {
 Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
 for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file.

Make sure that the resulting **.class** file is also in the **MyPack** directory.

Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

java AccountBalance

AccountBalance must be qualified with its package name.

Example:

```
package pck1;
class Student
{
  private int rollno;
 private String name;
private String address;
public Student(int rno, String sname, String sadd)
{
rollno = rno;
name = sname;
address = sadd;
}
public void showDetails()
{
System.out.println("Roll No :: " + rollno);
System.out.println("Name :: " + name);
System.out.println("Address :: " + address);
}
}
public class DemoPackage
{
public static void main(String ar[])
{
     Student st[]=new Student[2];
    st[0] = new Student (1001,"Alice", "New York");
    st[1] = new Student(1002,"BOb","Washington");
    st[0].showDetails();
    st[1].showDetails();
 }
}
```

Output:

```
 Roll No :: 1001
Name :: Alice
Address :: New York
Roll No :: 1002
Name :: Bob
Address :: Washington
```

JAVA STRING

- ❖ string is a sequence of characters. String is an object that represents a sequence of characters.
- ❖ For example,
  "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.
- ❖ We use **double quotes** to represent a string in Java.

- For example,

```
// create a string
String type = "Java programming";
```

- Here, we have created a string variable named type. The variable is initialized with the string Java Programming.
- Strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String.
- And, all string variables are instances of the String class.

- The java.lang.String class is used to create string object

- The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.
- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.
- **java string** class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Example: Create a String in Java

```
class Main
{
  public static void main(String[] args)
 {
    // create strings
    String first = "Java";
    String second = "Python";
    String third = "JavaScript";

    // print strings
    System.out.println(first);   // print Java
    System.out.println(second);  // print Python
    System.out.println(third);   // print JavaScript
  }
}
```

In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types.However, there is another way of creating Java strings (using the new keyword).

Creating strings using the new keyword

Example: Create Java Strings using the new keyword

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

    // create a string using new
    String name = new String("Java String");

    System.out.println(name);  // print Java String
  }
}
```

Create String using literals vs new keyword

- ❖ In Java, the JVM maintains a **string pool** to store all of its strings inside the memory. The string pool helps in reusing the strings.

1. While creating strings using string literals,

String example = "Java";

- ❖ Here, we are directly providing the value of the string (Java). Hence, the compiler first checks the string pool to see if the string already exists.

- ✓ **If the string already exists**, the new string is not created. Instead, the new reference, example points to the already existed string (Java).

- ✓ **If the string doesn't exist**, the new string (Java is created.

2. While creating strings using the new keyword,

String example = new String("Java");

Here, the value of the string is not directly provided. Hence, the new string is created all the time.

Java String Operations

Java String provides various methods to perform different operations on strings. some of the commonly used string operations.

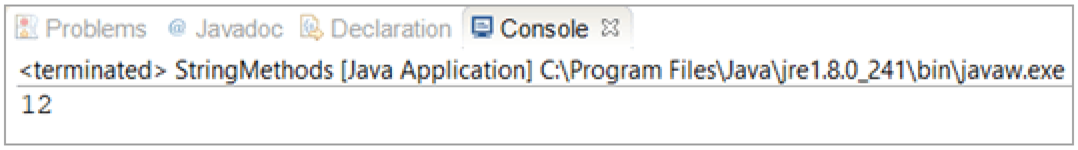
#1) Length

The length is the number of characters that a given string contains. Java has a length() method that gives the number of characters in a String.

Given below is the programming Example.

```
package codes;
import java.lang.String;
public class StringMethods
{
    public static void main(String[] args)
 {
        String str = "Saket Saurav";
        System.out.println(str.length());
    }
}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
12
```

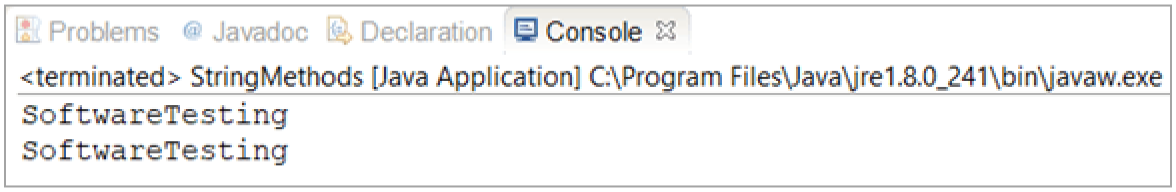
#2) Concatenation

Java uses a '+' operator for concatenating two or more strings. A concat() is an inbuilt method for String concatenation in Java.

Example of how we can use the concat() method in our programs is given below.

```
package codes;
import java.lang.String;
public class StringMethods
{
    public static void main(String[] args)
 {

        String str1 = "Software";
        String str2 = "Testing";
        System.out.println(str1 + str2);
        System.out.println(str1.concat(str2));
    }
}
```

Output:



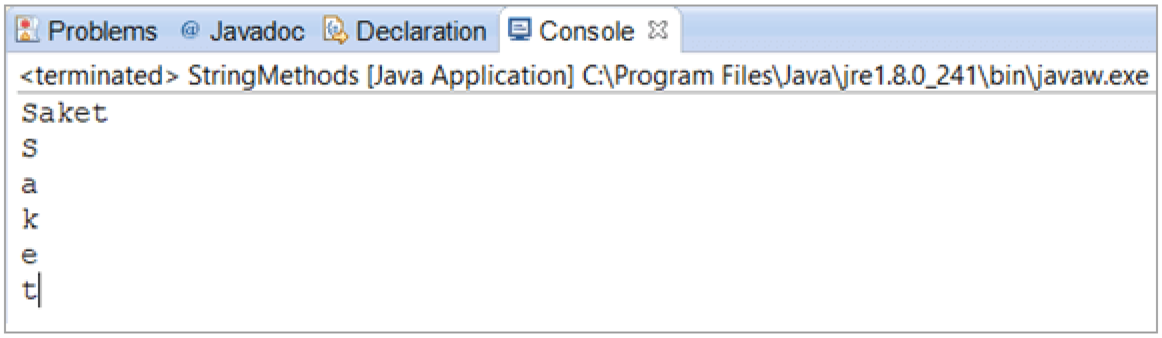
```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
SoftwareTesting
SoftwareTesting
```

#3) String to CharArray()

This method is used to convert all the characters of a string into a Character Array. This is widely used in the String manipulation programs.

```
package codes;
import java.lang.String;
public class StringMethods
{
   public static void main(String[] args)
{
      String str = "Saket";
      char[] chars = str.toCharArray();
      System.out.println(chars);
      for (int i= 0; i< chars.length; i++)
     {
         System.out.println(chars[i]);
      }
   }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
Saket
S
a
k
e
t
```

#4) String charAt()

This method is used to retrieve a single character from a given String.

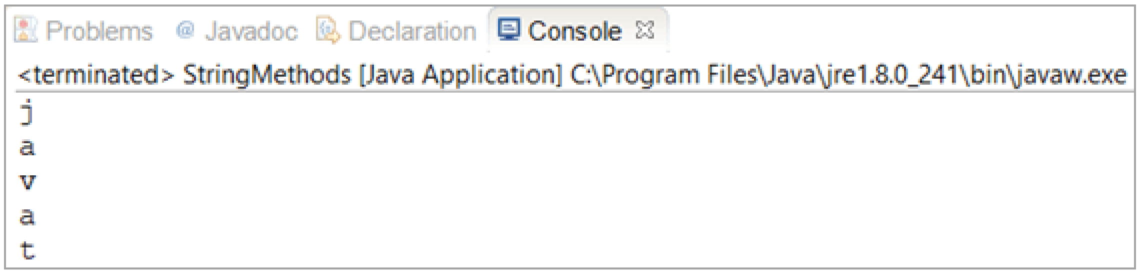
The syntax is given as:

char charAt(int i);

- The value of 'i' should not be negative and it should specify the location of a given String i.e. if a String length is 5, then the value of 'i' should be less than 5.

```
package codes;
import java.lang.String;
 public class StringMethods
{
    public static void main(String[] args)
{
        String str = "java string API";
        System.out.println(str.charAt(0));
        System.out.println(str.charAt(1));
        System.out.println(str.charAt(2));
        System.out.println(str.charAt(3));
        System.out.println(str.charAt(6));
    }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
j
a
v
a
t
```

System.out.println(str.charAt(50));

Or

System.out.println(str.charAt(-1));

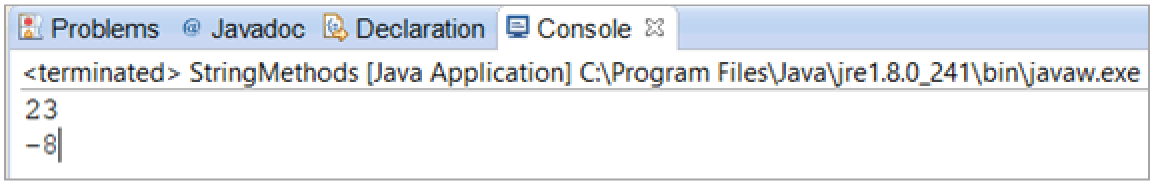
Then it will throw **“java.lang.StringIndexOutOfBoundsException:”**.

#5) Java String compareTo()

This method is used to compare two Strings. The comparison is based on alphabetical order. In general terms, a String is less than the other if it comes before the other in the dictionary.

```
package codes;
import java.lang.String;
public class StringMethods
{
   public static void main(String[] args)
{
      String str1 = "Zeus";
      String str2 = "Chinese";
      String str3 = "American";
      String str4 = "Indian";

      System.out.println(str1.compareTo(str2));
        //C comes 23 positions before Z, so it will give you 23
      System.out.println(str3.compareTo(str4));
        // I comes 8 positions after A, so it will give you -8
   }
}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
23
-8
```

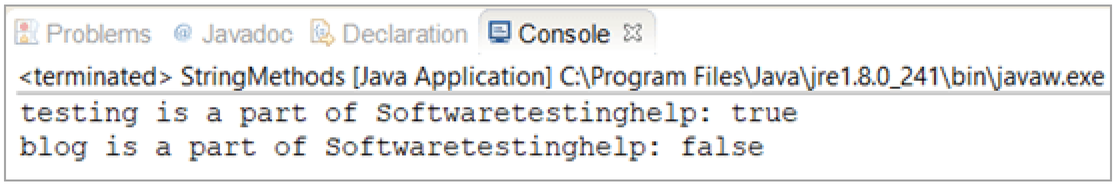
#6) String contains()

This method is used to determine whether a substring is a part of the main String or not. The return type is Boolean.

For E.g. In the below program, we will check whether "testing" is a part of "Softwaretestinghelp" or not and we will also check whether "blog" is a part of "Softwaretestinghelp".

```
package codes;
import java.lang.String;

public class StringMethods
{
   public static void main(String[] args)
 {

       String str = "Softwaretestinghelp";
       String str1 = "testing";
       String str2 = "blog";
       System.out.println("testing is a part of Softwaretestinghelp: " + str.contains(str1));
       System.out.println("blog is a part of Softwaretestinghelp: " + str.contains(str2));
   }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
testing is a part of Softwaretestinghelp: true
blog is a part of Softwaretestinghelp: false
```

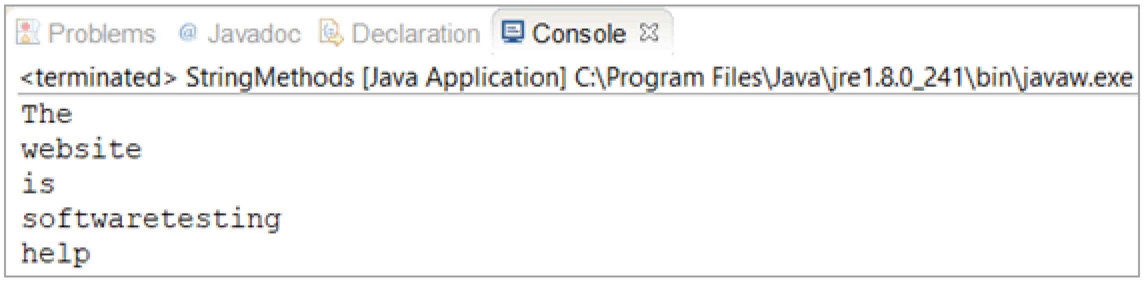
#7) Java String split()

A split() method is used to split or separate the given String into multiple substrings separated by the delimiters ("", " ", \\, etc). In the below example, we will split the String (Thexyzwebsitexyzisxyzsoftwaretestingxyzhelp) using a chunk of String(xyz) already present in the main String.

```
package codes;
```

```
import java.lang.String;

public class StringMethods
 {
    public static void main(String[] args)
{

        String str = "Thexyzwebsitexyzisxyzsoftwaretestingxyzhelp";
        String[] split = str.split("xyz");

        for (String obj: split)
      {
            System.out.println(obj);
        }
   }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
The
website
is
softwaretesting
help
```

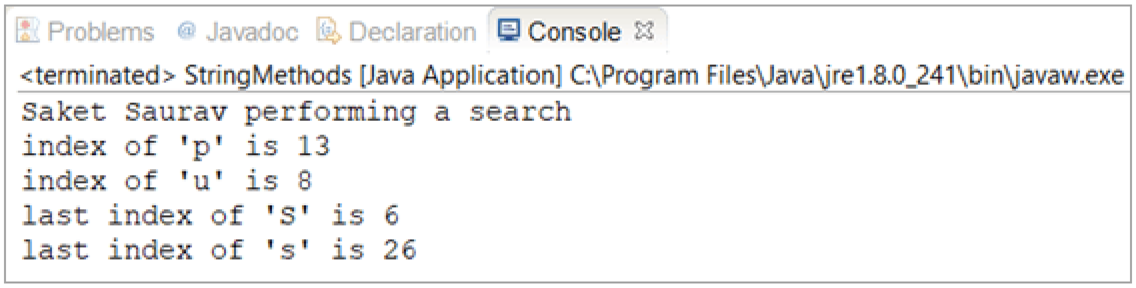
#8) Java String indexOf()

This method is used to perform a search operation for a specific character or a substring on the main String. There is one more method known as lastIndexOf() which is also commonly used.

indexOf() is used to search for the first occurrence of the character.
lastIndexOf() is used to search for the last occurrence of the character.

```
package codes;
import java.lang.String;

public class StringMethods
 {

    public static void main(String[] args)
{

        String str = "Saket Saurav " + "performing a search";
        System.out.println(str);
        System.out.println("index of 'p' is " + str.indexOf('p'));
        System.out.println("index of 'u' is " + str.indexOf('u'));
        System.out.println("last index of 'S' is " + str.lastIndexOf('S'));
        System.out.println("last index of 's' is " + str.lastIndexOf('s'));
    }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
Saket Saurav performing a search
index of 'p' is 13
index of 'u' is 8
last index of 'S' is 6
last index of 's' is 26
```

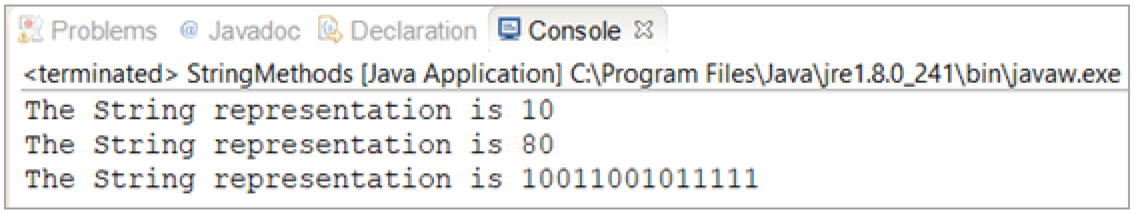
#9) Java String toString()

This method returns the String equivalent of the object that invokes it. This method does not have any parameters. Given below is the program where we will try to get the String representation of the object.

```
package codes;
import java.lang.String;
import java.lang.*;
```

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

       Integer obj = new Integer(10);
       String str = obj.toString();
       String str2 = obj.toString(80);
       String str3 = obj.toString(9823, 2);
//The above line will represent the String in base 2
       System.out.println("The String representation is " + str);
       System.out.println("The String representation is " + str2);
       System.out.println("The String representation is " + str3);
   }
}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
The String representation is 10
The String representation is 80
The String representation is 10011001011111
```

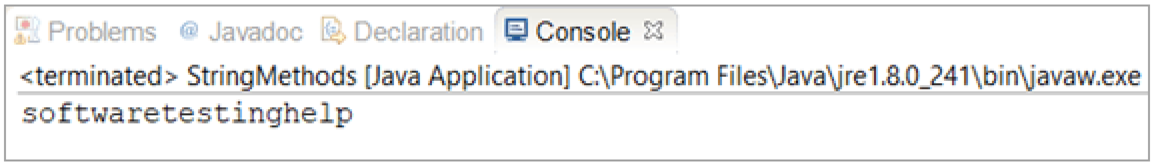
#10) String reverse()

The StringBuffer reverse() method is used to reverse the input characters of the String.

```
package codes;
import java.lang.*;

public class StringMethods
{
   public static void main(String[] args)
{

       String str = "plehgnitseterawtfos";
       StringBuffer sb = new StringBuffer(str);
```

```
        sb.reverse();
        System.out.println(sb);
    }
}
```

Output:



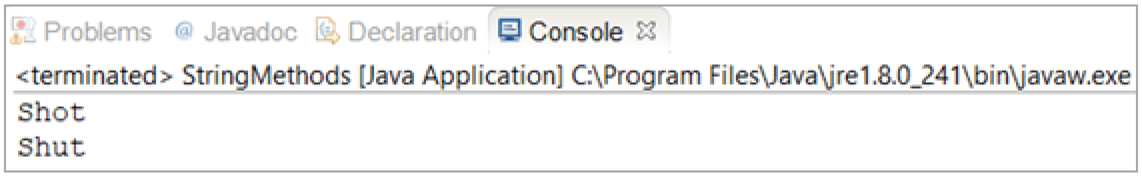
```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
softwaretestinghelp
```

#11) String replace()

The replace() method is used to replace the character with the new characters in a String.

```
package codes;
import java.lang.*;

public class StringMethods
{

    public static void main(String[] args)
{

        String str = "Shot";
        String replace = str.replace('o', 'u');
        System.out.println(str);
        System.out.println(replace);
    }

}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
Shot
Shut
```

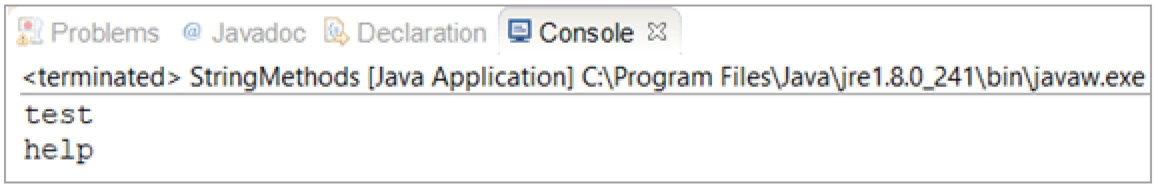
#12) Substring Method()

The Substring() method is used to return the substring of the main String by specifying the starting index and the last index of the substring.

For Example, in the given String "Softwaretestinghelp", we will try to fetch the substring by specifying the starting index and the last index.

```
package codes;
import java.lang.*;

public class StringMethods
{

   public static void main(String[] args)
{

      String str = "Softwaretestinghelp";
      System.out.println(str.substring(8,12));
//It will start from 8th character and extract the substring till 12th character
      System.out.println(str.substring(15,19));
   }
}
```

Output:



```
<terminated> StringMethods [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe
test
help
```

There are various string methods present in Java. Here are some of those methods:

Methods	Description
substring()	returns the substring of the string
replace()	replaces the specified old character with the specified new character
charAt()	returns the character present in the specified location
getBytes()	converts the string to an array of bytes
indexOf()	returns the position of the specified character in the string
compareTo()	compares two strings in the dictionary order
trim()	removes any leading and trailing whitespaces
format()	returns a formatted string
split()	breaks the string into an array of strings
toLowerCase()	converts the string to lowercase
toUpperCase()	converts the string to uppercase
valueOf()	returns the string representation of the specified argument
toCharArray()	converts the string to a char array

Escape character in Java Strings

The escape character is used to escape some of the characters present inside a string.

Suppose we need to include double quotes inside a string.

```
// include double quote
String example = "This is the "String" class";
```

Since strings are represented by **double quotes**, the compiler will treat "This is the " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Java. For example,

```
// use the escape character
String example = "This is the \"String\" class.";
```

Now escape characters tell the compiler to escape **double quotes** and read the whole text.

Java Strings are Immutable

- In Java, strings are **immutable**. This means, once we create a string, we cannot change that string.

consider an example:

```
// create a string
String example = "Hello! ";
```

Here, we have created a string variable named example. The variable holds the string "Hello! ".

Now suppose we want to change the string.

```
// add another string "World"
// to the previous string example
example = example.concat(" World");
```