

## **RMI (Remote Method Invocation)**

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

### **Understanding stub and skeleton**

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM.

#### ***stub***

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

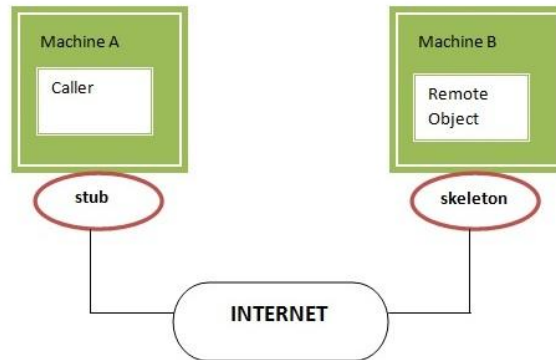
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads the return value or exception, and
5. It finally, returns the value to the caller.

#### **skeleton**

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and

3. It writes and transmits (marshals) the result to the caller.



### **Steps to write the RMI program**

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

## **Design and implement a simple Client Server Application using RMI.**

### **AddServerIntf.java**

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
    int add(int x, int y) throws RemoteException;
}
```

### **AddServerImpl.java**

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
implements AddServerIntf{
    public AddServerImpl() throws RemoteException {}
    public int add(int x, int y) throws RemoteException {
        return x+y;
    }
}
```

### **AddServer.java**

```
import java.rmi.*;
public class AddServer {
    public static void main(String[] args) {
        try{
            AddServerImpl server = new AddServerImpl();
            Naming.rebind("registerme",server);
            System.out.println("Server is running...");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

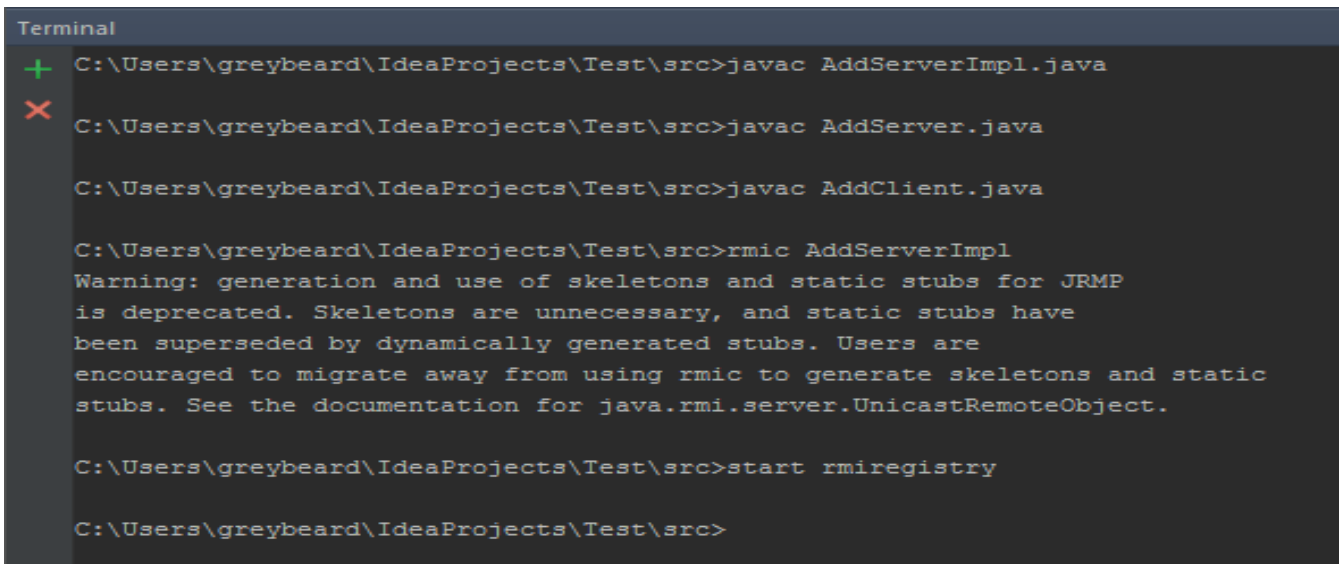
### **AddClient.java**

```
import java.rmi.*;
public class AddClient {
    public static void main(String[] args) {
        try{
            AddServerIntf client =
(AddServerIntf)Naming.lookup("registerme");
            System.out.println("First number is :" + args[0]);
            int x = Integer.parseInt(args[0]);
            System.out.println("Second number is :" + args[1]);
            int y = Integer.parseInt(args[1]);
            System.out.println("Sum =" + client.add(x,y));
        } catch (Exception e){
            System.out.println(e);
        }
    }
}
```

### **Output:**

Open a terminal

Navigate to the src folder of your project

A screenshot of a Windows command prompt terminal window. The title bar says "Terminal". The background is dark grey. The text is white. The first command is "C:\Users\greybeard\IdeaProjects\Test\src>javac AddServerImpl.java" with a green plus icon to its left. The second command is "C:\Users\greybeard\IdeaProjects\Test\src>javac AddServer.java" with a red X icon to its left. The third command is "C:\Users\greybeard\IdeaProjects\Test\src>javac AddClient.java". The fourth command is "C:\Users\greybeard\IdeaProjects\Test\src>rmic AddServerImpl", followed by a multi-line warning message: "Warning: generation and use of skeletons and static stubs for JRMP is deprecated. Skeletons are unnecessary, and static stubs have been superseded by dynamically generated stubs. Users are encouraged to migrate away from using rmic to generate skeletons and static stubs. See the documentation for java.rmi.server.UnicastRemoteObject." The fifth command is "C:\Users\greybeard\IdeaProjects\Test\src>start rmiregistry". The sixth command is "C:\Users\greybeard\IdeaProjects\Test\src>".

```
Terminal
+ C:\Users\greybeard\IdeaProjects\Test\src>javac AddServerImpl.java
X C:\Users\greybeard\IdeaProjects\Test\src>javac AddServer.java

C:\Users\greybeard\IdeaProjects\Test\src>javac AddClient.java

C:\Users\greybeard\IdeaProjects\Test\src>rmic AddServerImpl
Warning: generation and use of skeletons and static stubs for JRMP
is deprecated. Skeletons are unnecessary, and static stubs have
been superseded by dynamically generated stubs. Users are
encouraged to migrate away from using rmic to generate skeletons and static
stubs. See the documentation for java.rmi.server.UnicastRemoteObject.

C:\Users\greybeard\IdeaProjects\Test\src>start rmiregistry

C:\Users\greybeard\IdeaProjects\Test\src>
```

In another terminal (while previous one is still running)

Navigate to the src folder of your project

```
C:\Users\greybeard\IdeaProjects\Test\src>java AddServer  
Server is running...
```

In third terminal (while previous both are still open)

Navigate to the src folder of your project

```
C:\Users\greybeard\IdeaProjects\Test\src>java AddClient 2 3  
First number is :2  
Second number is :3  
Sum =5  
  
C:\Users\greybeard\IdeaProjects\Test\src>
```

## **Java Networking**

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

### **Advantage of Java Networking**

1. Sharing resources
2. Centralize software management

The java.net package supports two protocols,

1. **TCP:** Transmission Control Protocol provides reliable communication between the sender and receiver. TCP is used along with the Internet Protocol referred as TCP/IP.
2. **UDP:** User Datagram Protocol provides a connection-less protocol service by allowing packet of data to be transferred along two or more nodes

**Java Socket programming** is used for communication between the applications running on different JRE.

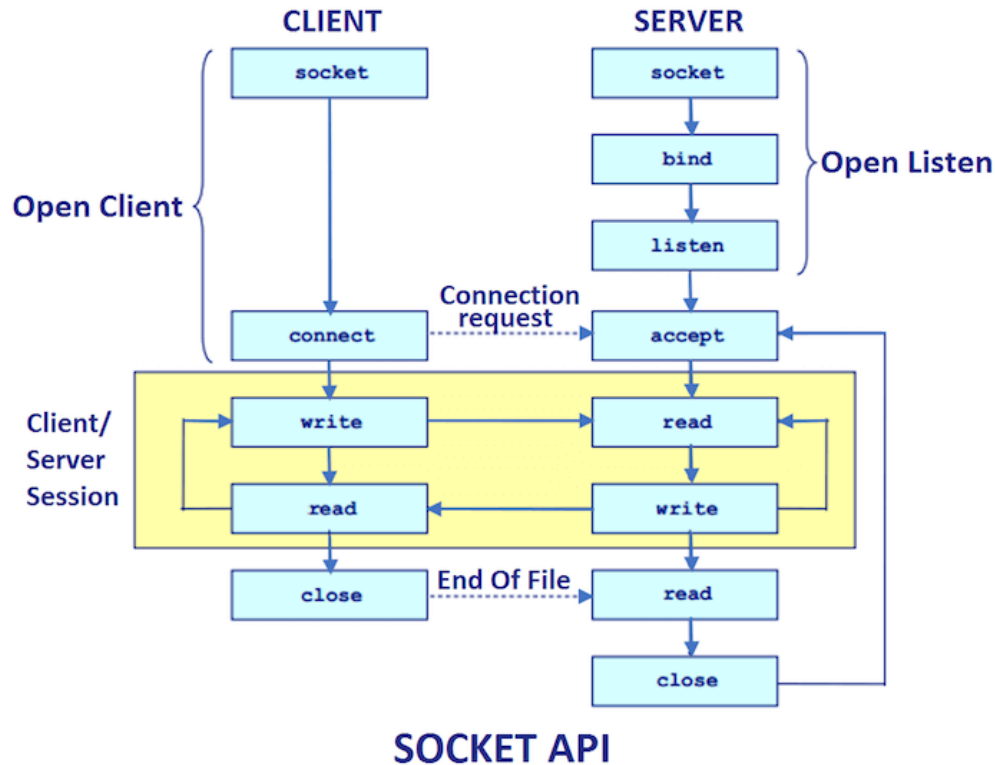
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server. When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class represents a socket, and the **java.net.ServerSocket** class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a **ServerSocket** object, denoting which port number communication is to occur on.
- The server invokes the **accept()** method of the **ServerSocket** class. This method waits until a client connects to the server on the given port.

- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the **accept()** method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an **OutputStream** and an **InputStream**. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, so data can be sent across both streams at the same time. There are following useful classes providing complete set of methods to implement sockets.

### **ServerSocket Class Methods:**

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

One of the four ServerSocket constructors are shown below:

#### **public ServerSocket(int port) throws IOException**

Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

Sl.No	Methods with Description
1	<b>public int getLocalPort()</b> Returns the port that the server socket is listening on. This method is useful if you passed



	in 0 as the port number in a constructor and let the server find a port for you.
<b>2</b>	<b>public Socket accept() throws IOException</b>  Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely
<b>3</b>	<b>public void setSoTimeout(int timeout)</b>  Sets the time-out value for how long the server socket waits for a client during the accept().
<b>4</b>	<b>public void bind(SocketAddress host, int backlog)</b>  Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

### **Socket Class Methods:**

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the **accept()** method.

The Socket class has five constructors that a client uses to connect to a server. One of them is shown below:

**public Socket(String host, int port) throws UnknownHostException, IOException.**

This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

Sl.No.	Methods with Description
1	<b>public int getPort()</b> Returns the port the socket is bound to on the remote machine.
2	<b>public SocketAddress getRemoteSocketAddress()</b> Returns the address of the remote socket.
3	<b>public InputStream getInputStream() throws IOException</b> Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
4	<b>public OutputStream getOutputStream() throws IOException</b> Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket
5	<b>public void close() throws IOException</b> Closes the socket, which makes this Socket object no longer capable of connecting again to any server

**Design and implement Client Server communication using socket programming (Client requests a file, Server responds to client with contents of that file which is then display on the screen by Client).**

#### **Client.java**

```
import java.net.*;
import java.io.*;
public class Client {
    public static void main(String[] args) {
        Socket client = null;
```

```

        BufferedReader br = null;
        try {
            System.out.println(args[0] + " " + args[1]);
            client = new
Socket(args[0],Integer.parseInt(args[1]));
        } catch (Exception e){}
        DataInputStream input = null;
        PrintStream output = null;
        try {
            input = new
DataInputStream(client.getInputStream());
            output = new PrintStream(client.getOutputStream());
            br = new BufferedReader(new
InputStreamReader(System.in));
            String str = input.readLine(); //get the prompt
from the server
            System.out.println(str);
            String filename = br.readLine();
            if (filename!=null){
                output.println(filename);
            }
            String data;
            while ((data=input.readLine())!=null) {
                System.out.println(data);
            }
            client.close();
        } catch (Exception e){
            System.out.println(e);
        }
    }
}

```

### **Server.java**

```

import java.net.*;
import java.io.*;

public class Server {
    public static void main(String[] args) {
        ServerSocket server = null;
        try {
            server = new ServerSocket(Integer.parseInt(args[0]));
        } catch (Exception e) {
        }
        while (true) {
            Socket client = null;

```

```

        PrintStream output = null;
        DataInputStream input = null;
        try {
            client = server.accept();
        } catch (Exception e) {
            System.out.println(e);
        }
        try {
            output = new PrintStream(client.getOutputStream());
            input = new DataInputStream(client.getInputStream());
        } catch (Exception e) {
            System.out.println(e);
        }
        //Send the command prompt to client
        output.println("Enter the filename :>");
        try {
            //get the filename from client
            String filename = input.readLine();
            System.out.println("Client requested file : " +
filename);
            try {
                File f = new File(filename);
                BufferedReader br = new BufferedReader(new
FileReader(f));
                String data;
                while ((data = br.readLine()) != null) {
                    output.println(data);
                }
            } catch (FileNotFoundException e) {
                output.println("File not found");
            }
            client.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

### **Output**

Create a file called testfile.txt in the folder where Client.java and Server.java is located. Add some content.

Open two terminals

Navigate to the src folder of your project

```
C:\Users\greybeard\IdeaProjects\Test\src>javac Server.java
Note: Server.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

```
C:\Users\greybeard\IdeaProjects\Test\src>java Server 4000
Client requested file :testfile.txt
```

```
C:\Users\greybeard\IdeaProjects\Test\src>javac Client.java
Note: Client.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\greybeard\IdeaProjects\Test\src>java Client localhost 4000
localhost 4000
Enter the filename :>
testfile.txt
Hello
How are you?

C:\Users\greybeard\IdeaProjects\Test\src>
```

## Example of Java Socket Programming

### MyServer.java

```
import java.io.*;
import java.net.*;

public class MyServer {
    public static void main(String[] args){
        try{
            ServerSocket ss=new ServerSocket(6666);
            Socket s=ss.accept();//establishes connection
            DataInputStream dis=new DataInputStream(s.getInputStream());
            String str=(String)dis.readUTF();
            System.out.println("message= "+str);
            ss.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

```
}
```

### **MyClient .java**

```
import java.io.*;
import java.net.*;
public class MyClient {
    public static void main(String[] args) {
        try{
            Socket s=new Socket("localhost",6666);
            DataOutputStream dout=new DataOutputStream(s.getOutputStream());
            dout.writeUTF("Hello Server");
            dout.flush();
            dout.close();
            s.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

# Java String

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','t','p','o','i','n','t'};
```

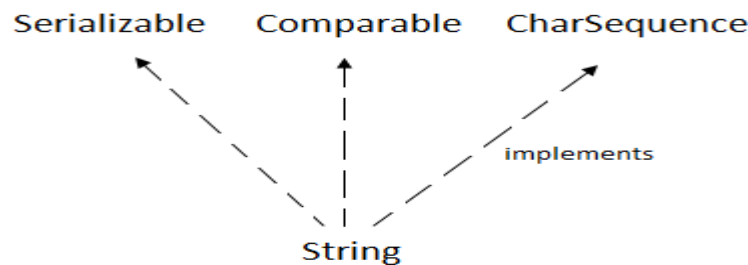
```
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

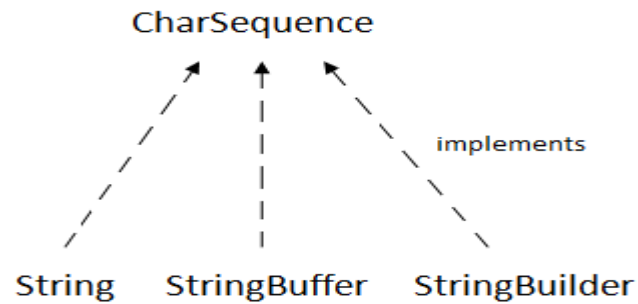
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

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



## CharSequence Interface

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.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

### **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. The `java.lang.String` class is used to create a string object.

### **How to create a string object?**

There are two ways to create String object:

By string literal

By new keyword

### **String Literal**

Java String literal is created by using double quotes.

For Example:

```
String s="welcome";
```

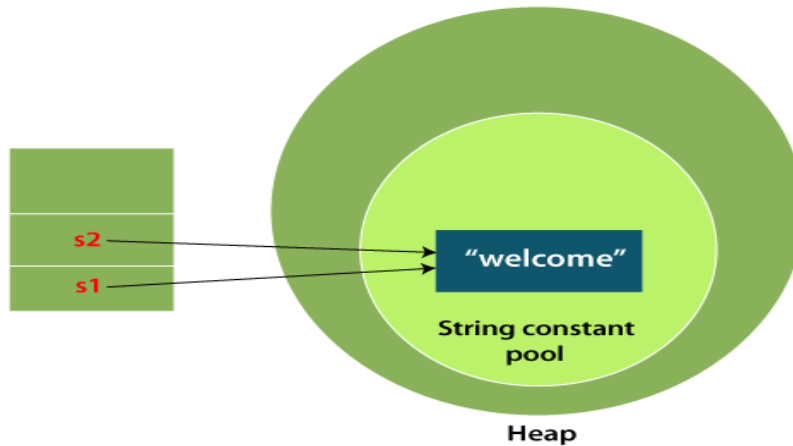
Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.



For example:

```
String s1="Welcome";
```

```
String s2="Welcome";//It doesn't create a new instance
```



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 that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

*Note: String objects are stored in a special memory area known as the "string constant pool".*

### Why Java uses the concept of String literal?

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

### 2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

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 a heap (non-pool).

### Java String Example 1:

```
public class StringExample{  
    public static void main(String args[]){  
        String s1="java";//creating string by Java string literal  
        char ch[]={'s','t','r','i','n','g','s'};  
        String s2=new String(ch);//converting char array to string  
        String s3=new String("example");//creating Java string by new keyword  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println(s3);  
    }  
}
```

### Output:

```
java  
strings  
example
```

The above code, converts a *char* array into a **String** object. And displays the String objects *s1*, *s2*, and *s3* on console using *println()* method.

### Java String class methods

The **java.lang.String** class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length

3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.

19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.

### **Immutable String in Java**

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

### **Example 2**

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

### **Output:**

**Java**

Now it can be understood by the diagram given below. Here Java is not changed but a new object is created with Java Programm. That is why String is known as immutable.

Here Java is not changed but a new object is created with Java Programm. That is why String is known as immutable. Two objects are created but *s* reference variable still refers to "Java" not to " Java Programm ".

But if we explicitly assign it to the reference variable, it will refer to " Java Programm " object.

For example:

#### **Testimmutablestring1.java**

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

#### **Output:**

##### **Java Programm**

In such a case, *s* points to the " **Java Programm**". Please notice that still Java object is not modified.

#### **Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Java". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

#### **1. ClassLoader:**

A `ClassLoader` in Java uses a `String` object as an argument. Consider, if the `String` object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, `String` is immutable.

## **2. Thread Safe:**

As the `String` object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

## **3. Security:**

As we have seen in class loading, immutable `String` objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because `String` objects are immutable. This can make the application program more secure.

## **4. Heap Space:**

The immutability of `String` helps to minimize the usage in the heap memory. When we try to declare a new `String` object, the JVM checks whether the value already exists in the `String` pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

## **Java String compare**

We can compare `String` in Java on the basis of content and reference.

It is used in **authentication** (by `equals()` method), **sorting** (by `compareTo()` method), **reference matching** (by `==` operator) etc.

There are three ways to compare `String` in Java:

### **1. By Using `equals()` Method**

### **2. By Using `==` Operator**

### 3. By compareTo() Method

#### 1) By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

```
class Teststringcomparison1 {  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        String s4="Saurav";  
        System.out.println(s1.equals(s2));//true  
        System.out.println(s1.equals(s3));//true  
        System.out.println(s1.equals(s4));//false  
    }  
}
```

**Output:**

```
true  
true  
false
```

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

### Example2.java

```
class Example2{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="SACHIN";  
  
        System.out.println(s1.equals(s2));//false  
        System.out.println(s1.equalsIgnoreCase(s2));//true  
    }  
}
```

#### Output:

```
false  
true
```

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

### 2) By Using == operator

The == operator compares references not values.

### Example3.java

```
class Example3{  
    public static void main(String args[]){  
        String s1="Java";  
        String s2="Java";  
        String s3=new String("Java");  
        System.out.println(s1==s2);//true (because both refer to same instance)  
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```



```
}  
}
```

### **3) By Using compareTo() method**

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

- **s1 == s2** : The method returns 0.
- **s1 > s2** : The method returns a positive value.
- **s1 < s2** : The method returns a negative value.

```
Class stringcomparison{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3="Ratan";  
        System.out.println(s1.compareTo(s2));//0  
        System.out.println(s1.compareTo(s3));//1(because s1>s3)  
        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )  
    }  
}
```

**Output:**

```
0  
1  
-1
```

### **String Concatenation in Java**

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method

### 1) String Concatenation by + (String concatenation) operator

Java String concatenation operator (+) is used to add strings. For Example:

#### Concatenation1.java

```
class Concatenation1 {  
    public static void main(String args[]){  
        String s="Sachin"+" Tendulkar";  
        System.out.println(s);//Sachin Tendulkar  
    }  
}
```

#### Output:

```
Sachin Tendulkar
```

The **Java compiler transforms** above code to this:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar").toString();
```

In Java, String concatenation is implemented through the `StringBuilder` (or `StringBuffer`) class and its `append` method. String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also. For Example:

#### Concatenation2.java

```
class Concatenation2{  
    public static void main(String args[]){  
        String s=50+30+"Sachin"+40+40;  
        System.out.println(s);//80Sachin4040  
    }  
}
```

```
}
```

## 2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

```
public String concat(String another)
```

Let's see the example of String concat() method.

### Concatenation3.java

```
class Concatenation3{  
    public static void main(String args[]){  
        String s1="Sachin ";  
        String s2="Tendulkar";  
        String s3=s1.concat(s2);  
        System.out.println(s3);//Sachin Tendulkar  
    }  
}
```

### Output:

**Sachin Tendulkar**

The above Java program, concatenates two String objects *s1* and *s2* using *concat()* method and stores the result into *s3* object.

## There are some other possible ways to concatenate Strings in Java.

### 1. String concatenation using StringBuilder class

StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastest way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

## StrBuilder.java

```
public class StrBuilder
{
    public static void main(String args[])
    {
        StringBuilder s1 = new StringBuilder("Hello"); //String 1
        StringBuilder s2 = new StringBuilder(" World"); //String 2
        StringBuilder s = s1.append(s2); //String 3 to store the result
        System.out.println(s.toString()); //Displays result
    }
}
```

### Output:

**Hello World**

In the above code snippet, **s1**, **s2** and **s** are declared as objects of **StringBuilder** class. **s** stores the result of concatenation of **s1** and **s2** using **append()** method.

## 2. String concatenation using format() method

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

## StrFormat.java

```
public class StrFormat
{
    public static void main(String args[])
    {
        String s1 = new String("Hello"); //String 1
        String s2 = new String(" World"); //String 2
        String s = String.format("%s%s",s1,s2); //String 3 to store the result
        System.out.println(s.toString()); //Displays result
    }
}
```

```
}  
}
```

#### Output:

```
Hello World
```

Here, the String objects **s** is assigned the concatenated result of Strings **s1** and **s2** using **String.format()** method. **format()** accepts parameters as format specifier followed by String objects or values.

### 3. String concatenation using **String.join()** method (Java Version 8+)

The **String.join()** method is available in Java version 8 and all the above versions. **String.join()** method accepts arguments first a separator and an array of String objects.

#### StrJoin.java:

```
public class StrJoin  
{  
    public static void main(String args[])  
    {  
        String s1 = new String("Hello"); //String 1  
        String s2 = new String(" World"); //String 2  
        String s = String.join("",s1,s2); //String 3 to store the result  
        System.out.println(s.toString()); //Displays result  
    }  
}
```

#### Output:

```
Hello World
```

In the above code snippet, the String object **s** stores the result of **String.join("",s1,s2)** method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

#### 4. String concatenation using **StringJoiner** class (Java Version 8+)

**StringJoiner** class has all the functionalities of **String.join()** method. In advance its constructor can also accept optional arguments, prefix and suffix.

```
public class StrJoiner

{
public static void main(String args[])
{
    StringJoiner s = new StringJoiner(", "); //StringeJoiner object
    s.add("Hello"); //String 1
    s.add("World"); //String 2
    System.out.println(s.toString()); //Displays result
}
}
```

**Output:**

```
Hello, World
```

In the above code snippet, the **StringJoiner** object **s** is declared and the constructor **StringJoiner()** accepts a separator value. A separator is specified inside quotation marks. The **add()** method appends Strings passed as arguments.

#### 5. String concatenation using **Collectors.joining()** method (Java (Java Version 8+))

The **Collectors** class in Java 8 offers **joining()** method that concatenates the input elements in a similar order as they occur.

**ColJoining.java**

```
import java.util.*;
import java.util.stream.Collectors;
public class ColJoining
```

```

{
    public static void main(String args[])
    {
        List<String> liststr = Arrays.asList("abc", "pqr", "xyz"); //List of String array
        String str = liststr.stream().collect(Collectors.joining(", ")); //performs joining operation
        System.out.println(str.toString()); //Displays result
    }
}

```

### Output:

**abc, pqr, xyz**

Here, a list of String array is declared. And a String object **str** stores the result of **Collectors.joining()** method.

### Substring in Java

A part of String is called **substring**. In other words, substring is a subset of another String. Java String class provides the built-in **substring()** method that extract a substring from the given string by using the index values passed as an argument. In case of **substring()** method **startIndex** is inclusive and **endIndex** is exclusive.

*Note: Index starts from 0.*

You can get substring from the given String object by one of the two methods:

1. **Public String substring(int startIndex):**  
This method returns new String object containing the substring of the given string from specified **startIndex** (inclusive). The method throws an **IndexOutOfBoundsException** when the **startIndex** is larger than the length of String or less than zero.
2. **public String substring(int startIndex, int endIndex):**  
This method returns new String object containing the substring of the given string from specified **startIndex** to **endIndex**. The method throws an **IndexOutOfBoundsException**

when the startIndex is less than zero or startIndex is greater than endIndex or endIndex is greater than length of String.

In case of String:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

1. String s="hello";
2. System.out.println(s.substring(0,2)); //returns he as a substring

In the above substring, 0 points the first letter and 2 points the second letter i.e., e (because end index is exclusive).

### Example of Java substring() method

#### TestSubstring.java

```
public class Substring{  
    public static void main(String args[]){  
        String s="SachinTendulkar";  
        System.out.println("Original String: " + s);  
        System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar  
        System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin  
    }  
}
```

#### Output:

```
Original String: SachinTendulkar  
Substring starting from index 6: Tendulkar  
Substring starting from index 0 to 6: Sachin
```



The above Java programs, demonstrates variants of the *substring()* method of *String* class. The startindex is inclusive and endindex is exclusive.

### **Using String.split() method:**

The split() method of String class can be used to extract a substring from a sentence. It accepts arguments in the form of a regular expression.

```
import java.util.*;

public class Substring2
{
    public static void main(String args[])
    {
        String text= new String("Hello, My name is JAVA");
        /* Splits the sentence by the delimiter passed as an argument */
        String[] sentences = text.split("\\.");
        System.out.println(Arrays.toString(sentences));
    }
}
```

### **Output:**

```
[Hello, My name is Java]
```

In the above program, we have used the split() method. It accepts an argument \\. that checks a in the sentence and splits the string into another string. It is stored in an array of String objects sentences.

### **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.

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Java";
System.out.println(s.toUpperCase());//JAVA
System.out.println(s.toLowerCase());//java
System.out.println(s);//Java(no change in original)
}
}
```

**Output:**

```
JAVA
java
Java
```

### Java String trim() method

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

### **Stringoperation2.java**

```
public class Stringoperation2
{
```

```
public static void main(String ar[])
{
String s=" JAVA  ";
System.out.println(s);// JAVA
System.out.println(s.trim());//JAVA
}
}
```

**Output:**

```
JAVA
```

```
JAVA
```

### **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**

```
public class Stringoperation3
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
}
}
```

**Output:**

```
true
```

true

### **Java String charAt() Method**

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

#### **Stringoperation4.java**

```
public class Stringoperation4
{
    public static void main(String ar[])
    {
        String s="JAVA";
        System.out.println(s.charAt(0));//J
        System.out.println(s.charAt(3));//V
    }
}
```

#### **Output:**

```
J
V
```

### **Java String length() Method**

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

#### **Stringoperation5.java**

```
public class Stringoperation5
{
    public static void main(String ar[])
    {
        String s="Java";
        System.out.println(s.length());//4
    }
}
```

```
}  
}
```

**Output:**

**4**

### **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**

```
public class Stringoperation6  
{  
public static void main(String ar[])  
{  
String s=new String("Sachin");  
String s2=s.intern();  
System.out.println(s2);//Sachin  
}  
}
```

**Output:**

**Sachin**

### **Java String valueOf() Method**

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

### **Stringoperation7.java**

```
public class Stringoperation7
{
public static void main(String ar[])
{
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
}
}
```

**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**

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

**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
<b>StringBuffer()</b>	It creates an empty String buffer with the initial capacity of 16.
<b>StringBuffer(String str)</b>	It creates a String buffer with the specified string..
<b>StringBuffer(int capacity)</b>	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.

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

**Output:**

Hello Java

### 2) StringBuffer insert() Method

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

```
class StringBufferExample2{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello ");
        sb.insert(1,"Java");//now original string is changed
        System.out.println(sb);//prints HJavaello
    }
}
```

```
}  
}
```

**Output:**

```
HJavaello
```

### **3) StringBuffer replace() Method**

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

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

**Output:**

```
HJavaalo
```

### **4) StringBuffer delete() Method**

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

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

**Output:**

```
Hlo
```

### **5) StringBuffer reverse() Method**

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

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

**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  $(\text{oldcapacity} * 2) + 2$ . For example if your current capacity is 16, it will be  $(16 * 2) + 2 = 34$ .

#### **StringBufferExample6.java**

```
class StringBufferExample6{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer();  
        System.out.println(sb.capacity());//default 16  
        sb.append("Hello");  
        System.out.println(sb.capacity());//now 16
```

```
sb.append("java is my favourite language");
System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

### 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  $(16 * 2) + 2 = 34$ .

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

## **Java toString() Method**

If you want to represent any object as a string, **toString() method** comes into existence.

The **toString()** method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

## **Advantage of Java toString() method**

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

## **Understanding problem without toString() method**

Let's see the simple code that prints reference.

### **Student.java**

```
class Student{  
    int rollno;  
    String name;  
    String city;  
  
    Student(int rollno, String name, String city){  
        this.rollno=rollno;  
        this.name=name;  
        this.city=city;  
    }  
}
```

```
public static void main(String args[]){  
    Student s1=new Student(101,"Raj","lucknow");  
    Student s2=new Student(102,"Vijay","ghaziabad");  
  
    System.out.println(s1);//compiler writes here s1.toString()  
    System.out.println(s2);//compiler writes here s2.toString()  
}  
}
```

### Output:

```
Student@1fee6fc  
Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since Java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

### Example of Java toString() method

Let's see an example of toString() method.

#### Student.java

```
class Student{  
    int rollno;  
    String name;  
    String city;  
  
    Student(int rollno, String name, String city){  
        this.rollno=rollno;  
        this.name=name;
```

```

this.city=city;
}

public String toString(){//overriding the toString() method
    return rollno+" "+name+" "+city;
}

public static void main(String args[]){
    Student s1=new Student(101,"Raj","lucknow");
    Student s2=new Student(102,"Vijay","ghaziabad");

    System.out.println(s1);//compiler writes here s1.toString()
    System.out.println(s2);//compiler writes here s2.toString()
}
}

```

### Output:

```

101 Raj lucknow
102 Vijay ghaziabad

```

In the above program, Java compiler internally calls *toString()* method, overriding this method will return the specified values of *s1* and *s2* objects of Student class.

### Java String valueOf()

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

### Signature

The signature or syntax of string valueOf() method is given below:

```

public static String valueOf(boolean b)

```

```
public static String valueOf(char c)
public static String valueOf(char[] c)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
public static String valueOf(double d)
public static String valueOf(Object o)
```

### **Java String valueOf() Complete Examples**

Let's see an example where we are converting all primitives and objects into strings.

```
public class StringValueOfExample5 {
    public static void main(String[] args) {
        boolean b1=true;
        byte b2=11;
        short sh = 12;
        int i = 13;
        long l = 14L;
        float f = 15.5f;
        double d = 16.5d;
        char chr[]={ 'j','a','v','a' };
        StringValueOfExample5 obj=new StringValueOfExample5();
        String s1 = String.valueOf(b1);
        String s2 = String.valueOf(b2);
        String s3 = String.valueOf(sh);
        String s4 = String.valueOf(i);
        String s5 = String.valueOf(l);
        String s6 = String.valueOf(f);
        String s7 = String.valueOf(d);
        String s8 = String.valueOf(chr);
```



```
String s9 = String.valueOf(obj);
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
System.out.println(s4);
System.out.println(s5);
System.out.println(s6);
System.out.println(s7);
System.out.println(s8);
System.out.println(s9);
}
}
```

#### **Output:**

```
true
11
12
13
14
15.5
16.5
java
StringValueOfExample5@2a139a55
```

#### **Java String trim()**

The **Java String class trim()** method eliminates leading and trailing spaces. The Unicode value of space character is '\u0020'. The trim() method in Java string checks this Unicode value before and after the string, if it exists then the method removes the spaces and returns the omitted string.

*The string trim() method doesn't omit middle spaces.*

### Signature

The signature or syntax of the String class **trim()** method is given below:

```
public String trim()
```

### Returns

string with omitted leading and trailing spaces

### Java String trim() Method Example

```
public class StringTrimExample{  
public static void main(String args[]){  
String s1=" hello string  ";  
System.out.println(s1+"javatpoint");//without trim()  
System.out.println(s1.trim()+"javatpoint");//with trim()  
}}
```

### Java String toUpperCase()

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

### Signature

There are two variant of toUpperCase() method. The signature or syntax of string toUpperCase() method is given below:

```
public String toUpperCase()  
public String toUpperCase(Locale locale)
```

The second method variant of toUpperCase(), converts all the characters into uppercase using the rules of given Locale.

### **Returns**

string in uppercase letter.

```
public class StringUpperExample{  
public static void main(String args[]){  
String s1="hello string";  
String s1upper=s1.toUpperCase();  
System.out.println(s1upper);  
}}
```

### **Output:**

```
HELLO STRING
```

## **Java String toLowerCase()**

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

### **Signature**

There are two variant of toLowerCase() method. The signature or syntax of string toLowerCase() method is given below:

```
public String toLowerCase()  
public String toLowerCase(Locale locale)
```

The second method variant of toLowerCase(), converts all the characters into lowercase using the rules of given Locale.

## Returns

string in lowercase letter.

```
public class StringLowerExample{  
    public static void main(String args[]){  
        String s1="JAVA HELLO stRIng";  
        String s1lower=s1.toLowerCase();  
        System.out.println(s1lower);  
    }  
}
```

## Output:

```
javapoint hello string
```

## Java String substring()

The **Java String class substring()** method returns a part of the string.

We pass `beginIndex` and `endIndex` number position in the Java substring method where `beginIndex` is inclusive, and `endIndex` is exclusive. In other words, the `beginIndex` starts from 0, whereas the `endIndex` starts from 1.

There are two types of substring methods in Java string.

### Signature

```
public String substring(int startIndex) // type - 1
```

and

```
public String substring(int startIndex, int endIndex) // type - 2
```

If we don't specify `endIndex`, the method will return all the characters from `startIndex`.

## Parameters

**startIndex** : starting index is inclusive

**endIndex** : ending index is exclusive

## Returns

specified string

## Exception Throws

**StringIndexOutOfBoundsException** is thrown when any one of the following conditions is met.

- if the start index is negative value
- end index is lower than starting index.
- Either starting or ending index is greater than the total number of characters present in the string.

```
public class SubstringExample2 {  
    public static void main(String[] args) {  
        String s1="Java";  
        String substr = s1.substring(0); // Starts with 0 and goes to end  
        System.out.println(substr);  
        String substr2 = s1.substring(5,10); // Starts from 5 and goes to 10  
        System.out.println(substr2);  
        String substr3 = s1.substring(5,15); // Returns Exception  
    }  
}
```

## Java String startsWith()

The **Java String class startsWith()** method checks if this string starts with the given prefix. It returns true if this string starts with the given prefix; else returns false.

## Signature

The syntax or signature of `startsWith()` method is given below.

**public boolean** `startsWith(String prefix)`

**public boolean** `startsWith(String prefix, int offset)`

## Parameter

**prefix** : Sequence of character

**offset**: the index from where the matching of the string prefix starts.

## Returns

true or false

## Java String `startsWith()` method example

The `startsWith()` method considers the case-sensitivity of characters. Consider the following example.

```
public class StartsWithExample
{
// main method
public static void main(String args[])
{
// input string
String s1="java string split method in javaProgram";
System.out.println(s1.startsWith("ja")); // true
System.out.println(s1.startsWith("java string")); // true
System.out.println(s1.startsWith("Java string")); // false as 'j' and 'J' are different
}
}
```

Output:

```
true
true
false
```

### Java String startsWith(String prefix, int offset) Method Example

It is an overloaded method of the startWith() method that is used to pass an extra argument (offset) to the function. The method works from the passed offset. Let's see an example.

StartsWithExample2.java

```
public class StartsWithExample2 {
    public static void main(String[] args) {
        String str = "javaProgram";
        // no offset mentioned; hence, offset is 0 in this case.
        System.out.println(str.startsWith("J")); // True

        // no offset mentioned; hence, offset is 0 in this case.
        System.out.println(str.startsWith("a")); // False
        // offset is 1
        System.out.println(str.startsWith("a",1)); // True
    }
}
```

**Output:**

```
true
false
true
```

## **Java String split()**

The **java string split()** method splits this string against given regular expression and returns a char array.

### **Signature**

There are two signature for split() method in java string.

```
public String split(String regex)
```

and,

```
public String split(String regex, int limit)
```

### **Parameter**

**regex** : regular expression to be applied on string.

**limit** : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

The given example returns total number of words in a string excluding space only. It also includes special characters.

```
public class SplitExample{  
public static void main(String args[]){  
String s1="java string split method by javaprogram";  
String[] words=s1.split("\\s");//splits the string based on whitespace  
//using java foreach loop to print elements of string array  
for(String w:words){  
System.out.println(w);  
}  
}}
```



java

string

split

method

by

javaprogram