27. 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.

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

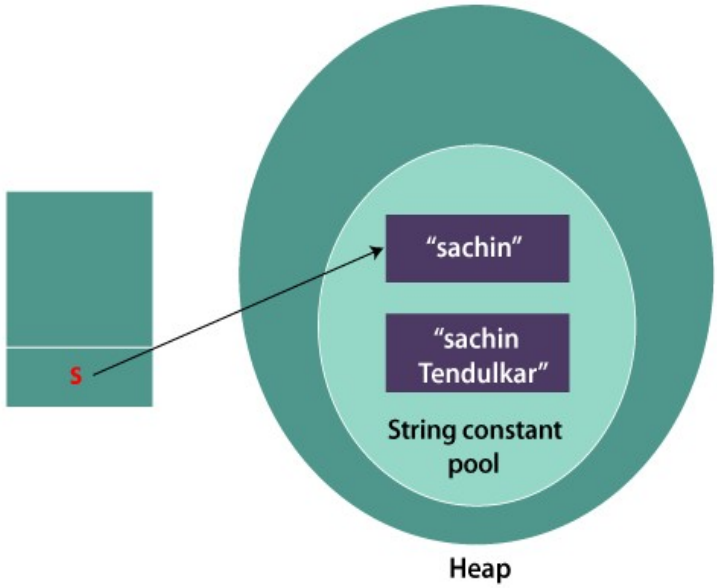
Let's try to understand the concept of immutability by the example given below:

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

Output:

```
Sachin
```

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



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

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

For example:

```
public class Testimmutablestring1
{
      public static void main(String args[])
      {
            String s="Sachin";
            //concat() method appends the string at the end
            s=s.concat(" Tendulkar");
            //will print Sachin because strings are immutable objects
            System.out.println(s);
      }
}
```

Output:

```
Sachin Tendulkar
```

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

27.1 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 "Sachin". 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:**

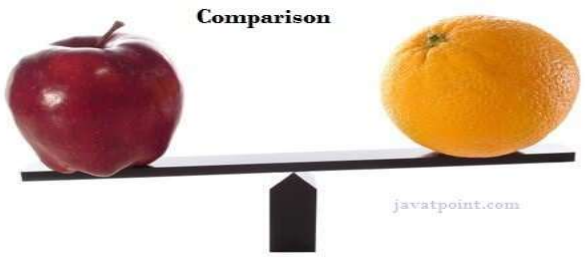
The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space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

Why String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

27.2 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

27.2.1 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.

Example1:

```
public 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:

```
public class Teststringcomparison2
{
	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.

27.2.2 By Using == operator

The == operator compares references not values.

Example:

```
public class Teststringcomparison3
{
	public static void main(String args[])
	{
		String s1="Sachin";
		String s2="Sachin";
		String s3=new String("Sachin");
		//true (because both refer to same instance)
		System.out.println(s1==s2);
		//false(because s3 refers to instance created in nonpool)
		System.out.println(s1==s3);
	}
```

```
}
```

Output:

```
true
false
```

The above code, demonstrates the use of == operator used for comparing two ***String*** objects.

27.2.3 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.

Example:

```
public class Teststringcomparison4
{
      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)
            //-1(because s3 < s1 )
            System.out.println(s3.compareTo(s1));
      }
}
```

Output:

```
0
1
-1
```

27.3 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

27.3.1 String Concatenation by + (String concatenation) operator

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

Example1:

```
public class TestStringConcatenation1
{
	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 it's 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:

Example2:

```
public class TestStringConcatenation2
{
	public static void main(String args[])
	{
		String s=50+30+"Sachin"+40+40;
		System.out.println(s);//80Sachin4040
	}
}
```

Output:

```
80Sachin4040
```

Note: After a string literal, all the + will be treated as string concatenation operator.

27.3.2 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.

Example:

```
public class TestStringConcatenation3
{
      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,

27.3.3 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 fastet way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

Example:

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

27.3.4 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.

Example:

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

27.3.5 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.

Example:

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

27.3.6 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.

Example:

```
import java.util.StringJoiner;

public class StrJoiner
{
    public static void main(String args[])
    {
        //StringeJoiner object
        StringJoiner s = new StringJoiner(", ");
        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.

27.3.7 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.

Example:

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

Output:

```
abc, pqr, xyz
```

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

27.4 String 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.

27.4.1 Java String charAt()

The **Java String class charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException,** if the given index number is greater than or equal to this string length or a negative number.

Syntax:

```
public char charAt(int index)
```

The method accepts **index** as a parameter. The starting index is 0. It returns a character at a specific index position in a string. It throws **StringIndexOutOfBoundsException** if the index is a negative value or greater than this string length.

Specified by

CharSequence interface, located inside java.lang package.

Internal implementation

```
public char charAt(int index) {
       if ((index < 0) || (index >= value.length)) {
           throw new StringIndexOutOfBoundsException(index);
       }
       return value[index];
   }
```

Example1:

Let's see Java program related to string in which we will use charAt() method that perform some operation on the give string.

```
public class CharAtExample
{
      public static void main(String args[])
      {
            String name="Bhavana";
            //returns the char value at the 4th index
            char ch=name.charAt(4);
            System.out.println(ch);
      }
}
```

Output:

```
a
```

Let's see the example of the **charAt()** method where we are passing a greater index value. In such a case, it throws **StringIndexOutOfBoundsException** at run time.

Example2:

```
public class CharAtExample
{
      public static void main(String args[])
      {
            String name="Bhavana";
            //returns the char value at the 4th index
            char ch=name.charAt(7);
            System.out.println(ch);
      }
}
```

Output:

```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out of
range: 7
      at
java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
      at java.base/java.lang.String.charAt(String.java:1515)
      at CharAtExample.main(CharAtExample.java:7)
```

Accessing First and Last Character by Using the charAt() Method

Let's see a simple example where we are accessing first and last character from the provided string.

Example3:

```
public class CharAtExample3
{
    public static void main(String[] args)
    {
    String str = "Welcome to Javatpoint portal";
    int strLength = str.length();
```

```
    // Fetching first character
    System.out.println("Character at 0 index is: "+ str.charAt(0));
    // The last Character is present at the string length-1 index
    System.out.println("Character at last index is: "+
str.charAt(strLength-1));
    }
}
```

Output:

```
Character at 0 index is: W
Character at last index is: l
```

Print Characters Presented at Odd Positions by Using the charAt() Method

Let's see an example where we are accessing all the elements present at odd index.

Example4:

```
public class CharAtExample4
{
     public static void main(String[] args)
     {
     String str = "Welcome to Javatpoint portal";
     for (int i=0; i<=str.length()-1; i++)
     {
          if(i%2!=0)
          {
          System.out.println("Char at "+i+" place "+str.charAt(i));
          }
     }
     }
}
```

Output:

```
Char at 1 place e
Char at 3 place c
Char at 5 place m
Char at 7 place
Char at 9 place o
Char at 11 place J
Char at 13 place v
Char at 15 place t
Char at 17 place o
Char at 19 place n
Char at 21 place
Char at 23 place o
Char at 25 place t
Char at 27 place l
```

The position such as 7 and 21 denotes the space.

Counting Frequency of a character in a String by Using the charAt() Method

Let's see an example in which we are counting frequency of a character in the given string.

Example5:

```
public class CharAtExample5
{
      public static void main(String[] args)
      {
      String str = "Welcome to Javatpoint portal";
      int count = 0;
      for (int i=0; i<=str.length()-1; i++)
      {
            if(str.charAt(i) == 't')
            {
            count++;
            }
      }
      System.out.println("Frequency of t is: "+count);
      }
}
```

Output:

```
Frequency of t is: 4
```

Counting the Number of Vowels in a String by Using the chatAt() Method

Let's see an example where we are counting the number of vowels present in a string with the help of the charAt() method.

Example6:

```
public class CharAtExample6
{
      public static void main(String[] args)
      {
      String str = "Lokesh Mannem";
      int count = 0;
      for (int i=0; i<str.length(); i++)
      {
            if(str.charAt(i) == 'A' || str.charAt(i) == 'a' ||
                  str.charAt(i) == 'E' || str.charAt(i) == 'e' ||
                  str.charAt(i) == 'I' || str.charAt(i) == 'i' ||
                  str.charAt(i) == 'O' || str.charAt(i) == 'o' ||
                  str.charAt(i) == 'U' || str.charAt(i) == 'u')
            {
            count++;
            }
      }
      System.out.println("Vowels in "+str+" is = "+count);
      }
}
```

Output:

```
Vowels in Lokesh Mannem is = 4
```

27.4.2 Java String length()

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

Signature

The signature of the string length() method is given below:

public int length()

Specified by

CharSequence interface

Returns

Length of characters. In other words, the total number of characters present in the string.

Internal implementation

```
public int length()
{
 return value.length;
}
```

The String class internally uses a char[] array to store the characters. The length variable of the array is used to find the total number of elements present in the array. Since the Java String class uses this char[] array internally; therefore, the length variable can not be exposed to the outside world. Hence, the Java developers created the length() method, the exposes the value of the length variable. One can also think of the length() method as the getter() method, that provides a value of the class field to the user. The internal implementation clearly depicts that the length() method returns the value of then the length variable.

Example1:

```
public class LengthExample1
{
      public static void main(String[] args)
      {
            String a="Lokesh";
            String b="Bhavana";
            String c="Bhavana Pillagurigandla";
            System.out.println(a.length());
            System.out.println(b.length());
            System.out.println(c.length());
      }
}
```

Output:

```
6
7
23
```

Example2:

Reversing a string.

```
public class LengthExample2
{
	public static void main(String[] args)
	{
		String a="Lokesh";
		String b="";
		for(int i=a.length()-1; i>=0; i--)
		{
			b=b+a.charAt(i);
		}
		System.out.println(b);
	}
}
```

Output:

```
hsekoL
```

27.4.3 Java String compareTo()

The Java String class compareTo() method compares the given string with the current string lexicographically. It returns a positive number, negative number, or 0.

It compares strings on the basis of the Unicode value of each character in the strings.

If the first string is lexicographically greater than the second string, it returns a positive number (difference of character value). If the first string is less than the second string lexicographically, it returns a negative number, and if the first string is lexicographically equal to the second string, it returns 0.

if s1 > s2, it returns positive number
if s1 < s2, it returns negative number
if s1 == s2, it returns 0

Syntax
public int compareTo(String anotherString)

The method accepts a parameter of type String that is to be compared with the current string.

It returns an integer value. It throws the following two exceptions:

ClassCastException: If this object cannot get compared with the specified object.
NullPointerException: If the specified object is null.

Example1:

```
public class CompareToExample
{
	public static void main(String args[])
	{
		String s1="hello";
		String s2="hello";
		String s3="meklo";
		String s4="hemlo";
		String s5="flag";
		//0 because both are equal
		System.out.println(s1.compareTo(s2));
		//-5 because "h" is 5 times lower than "m"
		System.out.println(s1.compareTo(s3));
		//-1 because "l" is 1 times lower than "m"
		System.out.println(s1.compareTo(s4));
		//2 because "h" is 2 times greater than "f"
		System.out.println(s1.compareTo(s5));
	}
}
```

Output:

```
0
-5
-1
2
```

Example2:

When we compare two strings in which either first or second string is empty, the method returns the length of the string. So, there may be two scenarios:

1. If **first** string is an empty string, the method returns a **negative**
2. If **second** string is an empty string, the method returns a **positive** number that is the length of the first string.

```
public class CompareToExample2
{
	public static void main(String args[])
	{
		String s1="hello";
		String s2="";
		String s3="me";
		System.out.println(s1.compareTo(s2));
		System.out.println(s2.compareTo(s3));
	}
```

```
}
```

Output:

```
5
-2
```

Example3:

```
public class CompareToExample3
{
	public static void main(String argvs[])
	{
	String st1 = new String("INDIA IS MY COUNTRY");
	String st2 = new String("india is my country");
	System.out.println(st1.compareTo(st2));
	}
}
```

Output:

```
-32
```

Example4:

The NullPointerException is thrown when a null object invokes the compareTo() method. Observe the following example.

```
public class CompareToExample4
{
	public static void main(String[] args)
	{
		String str = null;
		int no =  str.compareTo("India is my country.");
		System.out.println(no);
	}
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.compareTo(String)" because "str" is null
	at CompareToExample4.main(CompareToExample4.java:6)
```

27.4.4 Java String concat()

The **Java String class concat()** method *combines specified string at the end of this string*. It returns a combined string. It is like appending another string.

Signature

The signature of the string concat() method is given below:
public String concat(String anotherString)

Parameter
anotherString : another string i.e., to be combined at the end of this string.

Returns
combined string

Example1:

```
public class ConcatExample
{
      public static void main(String args[])
      {
      String s1="java string";
      // The string s1 does not get changed,
      //even though it is invoking the method
      // concat(), as it is immutable. Therefore,
      //the explicit assignment is required here.
      s1.concat("is immutable");
      System.out.println(s1);
      s1=s1.concat(" is immutable so assign it explicitly");
      System.out.println(s1);
      }
}
```

Output:

```
java string
java string is immutable so assign it explicitly
```

Example2:

```
public class ConcatExample2
{
      public static void main(String[] args)
      {
      String str1 = "Hello";
      String str2 = "Javatpoint";
      String str3 = "Reader";
      // Concatenating one string
      String str4 = str1.concat(str2);
      System.out.println(str4);
      // Concatenating multiple strings
      String str5 = str1.concat(str2).concat(str3);
      System.out.println(str5);
      }
}
```

Output:

```
HelloJavatpoint
HelloJavatpointReader
```

27.4.5 Java String contains()

The **Java String class contains()** method searches the sequence of characters in this string. It returns *true* if the sequence of char values is found in this string otherwise returns *false*.

Signature

The signature of string contains() method is given below:

public boolean contains(CharSequence sequence)

Parameter

sequence : specifies the sequence of characters to be searched.

Returns

true if the sequence of char value exists, otherwise false.

Exception

NullPointerException : if the sequence is null.

Here, the conversion of CharSequence takes place into String. After that, the indexOf() method is invoked. The method indexOf() either returns 0 or a number greater than 0 in case the searched string is found.

However, when the searched string is not found, the indexOf() method returns -1. Therefore, after execution, the contains() method returns true when the indexOf() method returns a non-negative value (when the searched string is found); otherwise, the method returns false.

Example1:

```
public class ContainsExample1
{
	public static void main(String args[])
	{
		String name="what do you know about me";
		System.out.println(name.contains("do you know"));
		System.out.println(name.contains("about"));
		System.out.println(name.contains("hello"));
	}
}
```

Output:

```
true
true
```

```
false
```

Example2:

The contains() method searches case-sensitive char sequence. If the argument is not case sensitive, it returns false. Let's see an example.

```
public class ContainsExample2
{
    public static void main(String[] args)
    {
        String str = "Hello Javatpoint readers";
        boolean isContains = str.contains("Javatpoint");
        System.out.println(isContains);
        // Case Sensitive
        System.out.println(str.contains("javatpoint")); // false
    }
}
```

Output:

```
true
false
```

Example3:

The contains() method raises the NullPointerException when one passes null in the parameter of the method. The following example shows the same.

```
public class ContainsExample3
{
     public static void main(String argvs[])
     {
          String str = "Welcome to JavaTpoint!";
          if(str.contains(null))
          {
               System.out.println("Inside the if block");
          }
          else
          {
               System.out.println("Inside the else block");
          }
     }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "java.lang.CharSequence.toString()" because "s" is null
     at java.base/java.lang.String.contains(String.java:2854)
```

```
        at ContainsExample3.main(ContainsExample3.java:6)
```

Example4:

```
public class ContainsExample4
{
        public static void main(String argvs[])
        {
                String str = "Welcome to JavaTpoint!";
                System.out.println(str.contains("v"));
        }
}
```

Output:

```
true
```

Limitations of the Contains() method

Following are some limitations of the contains() method:

1. The contains() method should not be used to search for a character in a string. Doing so results in an error.
2. The contains() method only checks for the presence or absence of a string in another string. It never reveals at which index the searched index is found. Because of these limitations, it is better to use the indexOf() method instead of the contains() method.

27.4.6 Java String equals()

The Java String class equals() method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of the Object class.

Signature
publicboolean equals(Object anotherObject)

Parameter
anotherObject : another object, i.e., compared with this string.

Returns
true if characters of both strings are equal otherwise false.

Example1:

```
public class EqualsExample1
{
        public static void main(String args[])
        {
                String s1="javatpoint";
                String s2="javatpoint";
```

```
            String s3="JAVATPOINT";
            String s4="python";
            //true because content and case is same
            System.out.println(s1.equals(s2));
            //false because case is not same
            System.out.println(s1.equals(s3));
            //false because content is not same
            System.out.println(s1.equals(s4));
      }
}
```

Output:

```
true
false
false
```

Example2:

```
public class EqualsExample2
{
      public static void main(String argvs[])
      {
            // Strings
            String str = "a";
            String str1 = "123";
            String str2 = "45.89";
            String str3 = "false";
            Character c = new Character('a');
            Integer i = new Integer(123);
            Float f = new Float(45.89);
            Boolean b = new Boolean(false);
            // reference of the Character object is passed
            System.out.println(str.equals(c));
            // reference of the Integer object is passed
            System.out.println(str1.equals(i));
            // reference of the Float object is passed
            System.out.println(str2.equals(f));
            // reference of the Boolean object is passed
            System.out.println(str3.equals(b));
            // the above print statements show a false value because
            // we are comparing a String with different data types
            // To achieve the true value, we have to convert
            // the different data types into the string using the
            // toString() method
            System.out.println(str.equals(c.toString()));
            System.out.println(str1.equals(i.toString()));
            System.out.println(str2.equals(f.toString()));
            System.out.println(str3.equals(b.toString()));
      }
```

```
}
```

Output:

```
false
false
false
false
true
true
true
true
```

27.4.7 Java String equalsIgnoreCase()

The Java String class equalsIgnoreCase() method compares the two given strings on the basis of the content of the string irrespective of the case (lower and upper) of the string. It is just like the equals() method but doesn't check the case sensitivity. If any character is not matched, it returns false, else returns true.

Signature

publicboolean equalsIgnoreCase(String str)

Parameter

str : another string i.e., compared with this string.

Returns

It returns true if characters of both strings are equal, ignoring case otherwise false.

Example:

```
public class EqualsIgnoreCaseExample
{
      public static void main(String args[])
      {
            String s1="javatpoint";
            String s2="javatpoint";
            String s3="JAVATPOINT";
            String s4="python";
            //true because content and case both are same
            System.out.println(s1.equalsIgnoreCase(s2));
            //true because case is ignored
            System.out.println(s1.equalsIgnoreCase(s3));
            //false because content is not same
            System.out.println(s1.equalsIgnoreCase(s4));
      }
}
```

Output:

```
true
true
false
```

27.4.8 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.

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

Example1:

```
public class SubstringExample1
{
      public static void main(String args[])
      {
            String s1="javatpoint";
            System.out.println(s1.substring(2,4));//returns va
            System.out.println(s1.substring(2));//returns vatpoint
      }
```

```
}
```

Output:

```
va
vatpoint
```

Example2:

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

Output:

```
Javatpoint
point
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: begin 5, end 15, length
10
      at
java.base/java.lang.String.checkBoundsBeginEnd(String.java:4604)
      at java.base/java.lang.String.substring(String.java:2707)
      at SubstringExample2.main(SubstringExample2.java:13)
```

27.4.9 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 startWith() 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

Example1:
The startsWith() method considers the case-sensitivity of characters. Consider the following example.

```
public class StartsWithExample1
{
      // main method
      public static void main(String args[])
      {
      // input string
      String s1="java string split method by javatpoint";
      System.out.println(s1.startsWith("ja"));// true
      System.out.println(s1.startsWith("java string"));// true
      // false as 'j' and 'J' are different
      System.out.println(s1.startsWith("Java string"));
      }
}
```

Output:

```
true
true
false
```

Example2:
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.

```
public class StartsWithExample2
{
    public static void main(String[] args)
    {
        String str = "Javatpoint";
        // 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
```

```
        System.out.println(str.startsWith("a",1)); // True
        System.out.println(str.startsWith("a",3)); // True
        System.out.println(str.startsWith("a",2)); // False
    }
}
```

Output:

```
true
false
true
true
false
```

Example3:

```
public class StartsWithExample3
{
    public static void main(String[] args)
    {
        String str = "Javatpoint";
        System.out.println(str.startsWith(""));
    }
}
```

Output:

```
true
```

27.4.10 Java String endsWith()

The Java String class endsWith() method checks if this string ends with a given suffix. It returns true if this string ends with the given suffix; else returns false.

Signature

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

public boolean endsWith(String suffix)

Parameter

suffix : Sequence of character Returns true or false

Example1:

```
public class EndsWithExample1
{
     public static void main(String args[])
     {
          String s1="java by javatpoint";
```

```
            System.out.println(s1.endsWith("t"));
            System.out.println(s1.endsWith("point"));
      }
}
```

Output:

```
true
true
```

Example2:

Since the endsWith() method returns a boolean value, the method can also be used in an if statement. Observe the following program.

```
public class EndsWithExample2
{
    public static void main(String[] args)
    {
        String str = "Welcome to Javatpoint.com";
        System.out.println(str.endsWith("point"));
        if(str.endsWith(".com"))
        {
            System.out.println("String ends with .com");
        }
        else
        {
           System.out.println("It does not end with .com");
        }
    }
}
```

Output:

```
false
String ends with .com
```

Example3:

The endsWith() method takes care of the case sensitiveness of the characters present in a string. The following program shows the same.

```
public class EndsWithExample3
{
      public static void main(String argvs[])
      {
            String str = "Welcome to JavaTpoint";
            //false because J and j are different
            System.out.println(str.endsWith("javaTpoint"));
```

```
            //false because T and t are different
            System.out.println(str.endsWith("Javatpoint"));
            //true because every character is same
            System.out.println(str.endsWith("JavaTpoint"));
      }
}
```

Output:

```
false
false
true
```

Example4:

```
public class EndsWithExample4
{
      public static void main(String argvs[])
      {
            String str = "Welcome to JavaTpoint";
            System.out.println(str.endsWith(""));
      }
}
```

Output:

```
true
```

Example5:

The endsWith() method raises the NullPointerException if one passes null in the parameter of the method. The following example shows the same.

```
public class EndsWithExample5
{
      public static void main(String argvs[])
      {
            String str = "Welcome to JavaTpoint";
            System.out.println(str.endsWith(null));
      }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.length()" because "suffix" is null
      at java.base/java.lang.String.endsWith(String.java:2315)
      at EndsWithExample5.main(EndsWithExample5.java:6)
```

27.4.11 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.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

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.

Example1:

```
public class StringLowerExample1
{
	public static void main(String args[])
	{
		String s1="JAVATPOINT HELLO stRIng";
		String s1lower=s1.toLowerCase();
		System.out.println(s1lower);
	}
}
```

Output:

```
javatpoint hello string
```

Example2:

This method allows us to pass locale too for the various langauges. Let's see an example below where we are getting string in english and turkish both.

```
import java.util.Locale;
public class StringLowerExample2
{
public static void main(String[] args)
{
String s = "JAVATPOINT HELLO stRIng";
```

```
String eng = s.toLowerCase(Locale.ENGLISH);
System.out.println(eng);
//It shows i without dot
String turkish = s.toLowerCase(Locale.forLanguageTag("tr"));
System.out.println(turkish);
}
}
```

Output:

```
javatpoint hello string
javatpoınt hello strıng
```

27.4.12 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.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

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.

Example1:

```
public class StringUpperExample1
{
      public static void main(String args[])
      {
            String s1="hello string JAVa";
            String s1upper=s1.toUpperCase();
            System.out.println(s1upper);
      }
}
```

Output:

```
HELLO STRING JAVA
```

Example2:

```
import java.util.Locale;
public class StringUpperExample2
{
public static void main(String[] args)
{
String s = "hello string";
String turkish = s.toUpperCase(Locale.forLanguageTag("tr"));
String english = s.toUpperCase(Locale.forLanguageTag("en"));
//will print I with dot on upper side
System.out.println(turkish);
System.out.println(english);
}
}
```

Output:

```
HELLO STRİNG
HELLO STRING
```

27.4.13 Java String format()

The java string format() method returns the formatted string by given locale, format and arguments.

If you don't specify the locale in String.format() method, it uses default locale by calling Locale.getDefault() method.

The format() method of java language is like sprintf() function in c language and printf() method of java language.

Signature

There are two type of string format() method:

public static String format(String format, Object... args)
and,
public static String format(Locale locale, String format, Object... args)

Parameters

locale : specifies the locale to be applied on the format() method.
format : format of the string.

args : arguments for the format string. It may be zero or more.

Returns
formatted string

Throws
NullPointerException : if format is null.
IllegalFormatException : if format is illegal or incompatible.

Java String Format Specifiers
Here, we are providing a table of format specifiers supported by the Java String.

Format Specifier	Data Type	Output
%a	floating point (except *BigDecimal*)	Returns Hex output of floating point number.
%b	Any type	"true" if non-null, "false" if null
%c	character	Unicode character
%d	integer (incl. byte, short, int, long, bigint)	Decimal Integer
%e	floating point	decimal number in scientific notation
%f	floating point	decimal number
%g	floating point	decimal number, possibly in scientific notation depending on the precision and value.
%h	any type	Hex String of value from hashCode() method.
%n	none	Platform-specific line separator.
%o	integer (incl. byte, short, int, long, bigint)	Octal number
%s	any type	String value

%t	Date/Time (incl. long, Calendar, Date and TemporalAccessor)	%t is the prefix for Date/Time conversions. More formatting flags are needed after this. See Date/Time conversion below.
%x	integer (incl. byte, short, int, long, bigint)	Hex string.

Example1:

```
public class FormatExample1
{
public static void main(String args[])
{
String name="sonoo";
String sf1=String.format("name is %s",name);
String sf2=String.format("value is %f",32.33434);
//returns 12 char fractional part filling with 0
//32-15(12+3(32.))=17 spaces before printing
String sf3=String.format("value is %32.12f",32.33434);
System.out.println(sf1);
System.out.println(sf2);
System.out.println(sf3);
}
}
```

Output:

```
name is sonoo
value is 32.334340
                  32.334340000000
```

Example2:

```
public class FormatExample2
{
     public static void main(String args[])
     {
          //combining to strings with +
          String a=String.format("%s+%s","hlo","bye");
          System.out.println(a);
          //combining to strings with ,
          String b=String.format("%s,%s","hlo","bye");
          System.out.println(b);
          //combining to strings with space
          String c=String.format("%s %s","hlo","bye");
          System.out.println(c);
          //combining String and integer
```

```
            String d=String.format("%s+%d","hlo",1234);
            System.out.println(d);
      }
}
```

Output:

```
hlo+bye
hlo,bye
hlo bye
hlo+1234
```

Example3:

```
public class FormatExample3
{
      public static void main(String args[])
      {
            String a=String.format("%a", 2.23456);
            System.out.println("using %a"+a);
      }
}
```

Output:

```
using %a0x1.1e060fe47991cp1
```

Example4:

```
public class FormatExample4
{
      public static void main(String args[])
      {
            String a=String.format("%b", 2.23456);
            System.out.println("using %b "+a);
            String b=String.format("%b", 229);
            System.out.println("using %b "+b);
            String c=String.format("%b", 'z');
            System.out.println("using %b "+c);
            String d=String.format("%b", 1.234f);
            System.out.println("using %b "+d);
            String e=String.format("%b", null);
            System.out.println("using %b "+e);
      }
}
```

Output:

```
using %b true
using %b true
using %b true
```

```
using %b true
using %b false
```

Example5:

```
public class FormatExample5
{
	public static void main(String args[])
	{
		String a=String.format("%c", 's');
		System.out.println(a);
	}
}
```

Output:

```
s
```

Example6:

```
public class FormatExample6
{
	public static void main(String args[])
	{
		String a=String.format("%d", 1111);
		System.out.println(a);
		String b=String.format("%d", 1110123);
		System.out.println(b);
		String c=String.format("%d", 111123214214L);
		System.out.println(c);
		String d=String.format("%d", 35);
		System.out.println(d);
	}
}
```

Output:

```
1111
1110123
111123214214
35
```

Example7:

```
public class FormatExample7
{
	public static void main(String args[])
	{
		String a=String.format("%e", 123.23456);
```

```
            System.out.println(a);
      }
}
```

Output:

```
1.232346e+02
```

Example8:

```
public class FormatExample8
{
      public static void main(String args[])
      {
            String a=String.format("%f", 123.23456789);
            System.out.println(a);
            String b=String.format("%f", 123.234);
            System.out.println(b);
      }
}
```

Output:

```
123.234568
123.234000
```

Example9:

```
public class FormatExample9
{
      public static void main(String args[])
      {
            String a=String.format("%g", 123.23456789);
            System.out.println(a);
      }
}
```

Output:

```
123.235
```

Example10:

```
public class FormatExample10
{
      public static void main(String args[])
      {
            String a=String.format("%h", 123.23456789);
            System.out.println(a);
      }
```

```
}
```

Output:

```
6954c07b
```

Example11:

```
public class FormatExample11
{
	public static void main(String args[])
	{
		String a=String.format("%n");
		System.out.println("hlo"+a+"bye");
	}
}
```

Output:

```
hlo
bye
```

Example12:

```
public class FormatExample12
{
	public static void main(String args[])
	{
		String a=String.format("%o",123);
		System.out.println(a);
	}
}
```

Output:

```
173
```

Example13:

```
public class FormatExample13
{
	public static void main(String args[])
	{
		String a=String.format("%x",123);
		System.out.println(a);
	}
}
```

Output:

```
7b
```

Example14:

```
public class FormatExample14
{
    public static void main(String[] args)
    {
        String str1 = String.format("%d", 101);
        // Specifying length of integer
        String str2 = String.format("|%10d|", 101);
        // Left-justifying within the specified width
        String str3 = String.format("|%-10d|", 101);
        String str4 = String.format("|% d|", 101);
        // Filling with zeroes
        String str5 = String.format("|%010d|", 101);
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
        System.out.println(str4);
        System.out.println(str5);
        String str6 = String.format("+%010d+", 101);
        System.out.println(str6);
    }
}
```

Output:

```
101
|       101|
|101       |
| 101|
|0000000101|
+0000000101+
```

27.4.14 Java String join()

The Java String class join() method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element. The join() method is included in the Java string since JDK 1.8.

There are two types of join() methods in the Java String class.

Signature

The signature or syntax of the join() method is given below:

public static String join(CharSequence delimiter, CharSequence... elements)
and
public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

Parameters
delimiter : char value to be added with each element
elements : char value to be attached with delimiter

Returns
joined string with delimiter

Exception Throws
NullPointerException if element or delimiter is null.

Example1:

```
public class StringJoinExample1
{
public static void main(String args[])
{
String joinString1=String.join("-","welcome","to","javatpoint");
System.out.println(joinString1);
}
}
```

Output:

```
welcome-to-javatpoint
```

Example2:
We can use a delimiter to format the string as we did in the below example to show the date and time.

```
public class StringJoinExample2
{
    public static void main(String[] args)
    {
        String date = String.join("/","25","06","2018");
        System.out.print(date);
        String time = String.join(":", "12","10","10");
        System.out.println(" "+time);
    }
}
```

Output:

```
25/06/2018 12:10:10
```

Example3:
In the case of using null as a delimiter, we get the null pointer exception. The following example confirms the same.

```
public class StringJoinExample3
{
      public static void main(String argvs[])
      {
      String str = null;
      str = String.join(null, "abc", "bcd", "apple");
      System.out.println(str);
      }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "java.lang.CharSequence.toString()" because "delimiter" is
null
      at java.base/java.lang.String.join(String.java:3228)
      at StringJoinExample3.main(StringJoinExample3.java:6)
```

Example4:

```
public class StringJoinExample4
{
public static void main(String argvs[])
{
String str = null;
// one of the element is null however it
// will be treated as normal string
str = String.join("-", null, " wake up ", " eat ",
           " write content for JTP ", " eat ", " sleep ");
System.out.println(str);
}
}
```

Output:

```
null- wake up - eat - write content for JTP - eat - sleep
```

27.4.15 Java String join()

The Java String class indexOf() method returns the position of the first occurrence of the specified character or string in a specified string.

Signature

There are four overloaded indexOf() method in Java. The signature of indexOf() methods are given below:

No.	Method	Description
1	int indexOf(int ch)	It returns the index position for the given char value
2	int indexOf(int ch, int fromIndex)	It returns the index position for the given char value and from index
3	int indexOf(String substring)	It returns the index position for the given substring
4	int indexOf(String substring, int fromIndex)	It returns the index position for the given substring and from index

Parameters

ch: It is a character value, e.g. 'a'

fromIndex: The index position from where the index of the char value or substring is returned.

substring: A substring to be searched in this string.

Returns

Index of the searched string or character.

Example1:

```
public class IndexOfExample1
{
      public static void main(String args[])
      {
      String s1="this is index of example";
      //passing substring
      //returns the index of is substring
      int index1=s1.indexOf("is");
      //returns the index of index substring
      int index2=s1.indexOf("index");
      System.out.println(index1+"  "+index2);//2 8
      //passing substring with from index
      //returns the index of is substring from 4th index
      int index3=s1.indexOf("is",4);
      //5 i.e. the index of another is
      System.out.println(index3);
      //passing char value
      int index4=s1.indexOf('s');
      //returns the index of s char value
      System.out.println(index4);//3
      }
}
```

Output:

```
2  8
5
3
```

Example2:

The method takes substring as an argument and returns the index of the first character of the substring.

```
public class IndexOfExample2
{
    public static void main(String[] args)
    {
        String s1 = "This is indexOf method";
        // Passing Substring
        //Returns the index of this substring
        int index = s1.indexOf("method");
        System.out.println("index of substring "+index);
    }
}
```

Output:

```
index of substring 16
```

Example3:

The method takes substring and index as arguments and returns the index of the first character that occurs from the given fromIndex.

```
public class IndexOfExample3
{
    public static void main(String[] args)
    {
        String s1 = "This is indexOf method";
        // Passing substring and index
        //Returns the index of this substring
        int index = s1.indexOf("method", 10);
        System.out.println("index of substring "+index);
        // It returns -1 if substring does not found
        index = s1.indexOf("method", 20);
        System.out.println("index of substring "+index);
    }
}
```

Output:

```
index of substring 16
index of substring -1
```

Example4:
The method takes char and index as arguments and returns the index of the first character that occurs from the given fromIndex.

```
public class IndexOfExample4
{
    public static void main(String[] args)
    {
        String s1 = "This is indexOf method";
        // Passing char and index from
        //Returns the index of this char
        int index = s1.indexOf('e', 12);
        System.out.println("index of char "+index);
    }
}
```

Output:

```
index of char 17
```

27.4.16 Java String isEmpty()

The Java String class isEmpty() method checks if the input string is empty or not. Note that here empty means the number of characters contained in a string is zero.

Signature
The signature or syntax of string isEmpty() method is given below:
public boolean isEmpty()

Returns
true if length is 0 otherwise false.

Example1:

```
public class IsEmptyExample1
{
     public static void main(String args[])
     {
          String s1="";
          String s2="javatpoint";
          System.out.println(s1.isEmpty());
          System.out.println(s2.isEmpty());
     }
}
```

Output:

```
true
false
```

Example2:

```
public class IsEmptyExample2
{
    public static void main(String[] args)
    {
        String s1="";
        String s2="Javatpoint";
        // Either length is zero or isEmpty is true
        if(s1.length()==0 || s1.isEmpty())
        {
           System.out.println("String s1 is empty");
        }
        else
        {
           System.out.println("s1");
        }

        if(s2.length()==0 || s2.isEmpty())
        {
           System.out.println("String s2 is empty");
        }
        else
        {
           System.out.println(s2);
        }
    }
}
```

Output:

```
String s1 is empty
Javatpoint
```

Empty Vs. Null Strings

Earlier in this tutorial, we have discussed that the empty strings contain zero characters. However, the same is true for a null string too. A null string is a string that has no value.

String str = ""; // empty string
String str1 = null; // null string. It is also not containing any characters.

The isEmpty() method is not fit for checking the null strings. The following example shows the same.

Example3:

```
public class StringIsEmptyExample3
{
      public static void main(String argvs[])
      {
```

```
            String str = null;
            if(str.isEmpty())
            {
            System.out.println("The string is null.");
            }
            else
            {
            System.out.println("The string is not null.");
            }
      }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.isEmpty()" because "str" is null
      at StringIsEmptyExample3.main(StringIsEmptyExample3.java:6)
```

Example4:

Here, we can use the == operator to check for the null strings.

```
public class StringIsEmptyExample4
{
      public static void main(String argvs[])
      {
      String str = null;
      if(str == null)
      {
      System.out.println("The string is null.");
      }
      else
      {
      System.out.println("The string is not null.");
      }
      }
}
```

Output:

```
The string is null.
```

27.4.17 Java String intern()

The Java String class intern() method returns the interned string. It returns the canonical representation of string.

It can be used to return string from memory if it is created by a new keyword. It creates an

exact copy of the heap string object in the String Constant Pool.

Signature
The signature of the intern() method is given below:

public String intern()

Returns
interned string

The need and working of the String.intern() Method

When a string is created in Java, it occupies memory in the heap. Also, we know that the String class is immutable. Therefore, whenever we create a string using the new keyword, new memory is allocated in the heap for corresponding string, which is irrespective of the content of the array. Consider the following code snippet.

```
String str = new String("Welcome to JavaTpoint.");
String str1 = new String("Welcome to JavaTpoint");
System.out.println(str1 == str); // prints false
```

The println statement prints false because separate memory is allocated for each string literal. Thus, two new string objects are created in the memory i.e. str and str1. that holds different references.

We know that creating an object is a costly operation in Java. Therefore, to save time, Java developers came up with the concept of String Constant Pool (SCP). The SCP is an area inside the heap memory. It contains the unique strings. In order to put the strings in the string pool, one needs to call the intern() method. Before creating an object in the string pool, the JVM checks whether the string is already present in the pool or not. If the string is present, its reference is returned.

```
String str = new String("Welcome to JavaTpoint").intern(); // statement - 1
String str1 = new String("Welcome to JavaTpoint").intern(); // statement - 2
System.out.println(str1 == str); // prints true
```

In the above code snippet, the intern() method is invoked on the String objects. Therefore, the memory is allocated in the SCP. For the second statement, no new string object is created as the content of str and str1 are the same. Therefore, the reference of the object created in the first statement is returned for str1. Thus, str and str1 both point to the same memory. Hence, the print statement prints true.

Example1:

```
public class InternExample1
{
```

```
    public static void main(String args[])
    {
        String s1=new String("hello");
        String s2="hello";
        //returns string from pool, now it will be same as s2
        String s3=s1.intern();
        //false because reference variables are
        //pointing to different instance
        System.out.println(s1==s2);
        //true because reference variables are
        //pointing to same instance
        System.out.println(s2==s3);
    }
}
```

Output:

```
false
true
```

Example2:

Let's see one more example to understand the string intern concept.

```
public class InternExample2
{
    public static void main(String[] args)
    {
        String s1 = "Javatpoint";
        String s2 = s1.intern();
        String s3 = new String("Javatpoint");
        String s4 = s3.intern();
        System.out.println(s1==s2); // True
        System.out.println(s1==s3); // False
        System.out.println(s1==s4); // True
        System.out.println(s2==s3); // False
        System.out.println(s2==s4); // True
        System.out.println(s3==s4); // False
    }
}
```

Output:

```
true
false
true
false
true
false
```

Points to Remember

1) A string literal always invokes the intern() method, whether one mention the intern() method along with the string literal or not. For example,

```
String s = "d".intern();
String p = "d"; // compiler treats it as String p = "d".intern();
System.out.println(s == p); // prints true
```

2) Whenever we create a String object using the new keyword, two objects are created. For example,

```
String str = new ("Hello World");
```

Here, one object is created in the heap memory outside of the SCP because of the usage of the new keyword. As we have got the string literal too ("Hello World"); therefore, one object is created inside the SCP, provided the literal "Hello World" is already not present in the SCP.

27.4.18 Java String getBytes()

The Java String class getBytes() method does the encoding of string into the sequence of bytes and keeps it in an array of bytes.

Signature

There are three variants of getBytes() method. The signature or syntax of string getBytes() method is given below:

```
public byte[] getBytes()
public byte[] getBytes(Charset charset)
public byte[] getBytes(String charsetName)throws UnsupportedEncodingException
```

Parameters

charset / charsetName - The name of a charset the method supports.

Returns

Sequence of bytes.

Exception Throws

UnsupportedEncodingException: It is thrown when the mentioned charset is not supported by the method.

Example1:

The parameterless getBytes() method encodes the string using the default charset of the

platform, which is UTF - 8. The following two examples show the same.

```
public class StringGetBytesExample1
{
      public static void main(String args[])
      {
            String s1="ABCDEFG";
            byte[] barr=s1.getBytes();
            for(int i=0;i<barr.length;i++)
            {
                  System.out.println(barr[i]);
            }
      }
}
```

Output:

```
65
66
67
68
69
70
71
```

Example2:

The method returns a byte array that again can be passed to the String constructor to get String.

```
public class StringGetBytesExample2
{
    public static void main(String[] args)
    {
        String s1 = "ABCDEFG";
        byte[] barr = s1.getBytes();
        for(int i=0;i<barr.length;i++)
        {
            System.out.println(barr[i]);
        }
        // Getting string back
        String s2 = new String(barr);
        System.out.println(s2);
    }
}
```

Output:

```
65
66
```

```
67
68
69
70
71
ABCDEFG
```

Example3:
The following example shows the encoding into a different charset.

```
// Import statement
import java.io.*;

public class StringGetBytesExample3
{
public static void main(String argvs[])
{
// input string
String str = "Welcome to JavaTpoint.";
System.out.println("The input String is : ");
System.out.println(str + "\n");

// inside try block encoding is
// being done using different charsets
try
{
//16 - bit UCS Transformation format
byte[] byteArr = str.getBytes("UTF-16");
System.out.println("After converted into UTF-16 the String is : ");

for(int j = 0; j < byteArr.length; j++)
{
     System.out.print(byteArr[j]);
}
}
catch (UnsupportedEncodingException g)
{
System.out.println("Unsupported character set" + g);
}
}
}
```

Output:

```
The input String is :
Welcome to JavaTpoint.

After converted into UTF-16 the String is :
```

```
-2-
1087010101080990111010901010320116011103207409701180970840112011101050
1100116046
```

Example4:

```
// Import statement
import java.io.*;

public class StringGetBytesExample4
{
public static void main(String argvs[])
{
// input string
String str = "Welcome to JavaTpoint.";
System.out.println("The input String is : ");
System.out.println(str + "\n");

// inside try block encoding is
// being done using different charsets
try
{
// Big Endian byte order, 16 - bit UCS Transformation format
byte[] byteArr1 = str.getBytes("UTF-16BE");
System.out.println("After converted into UTF-16BE the String is :
");

for(int j = 0; j < byteArr1.length; j++)
{
     System.out.print(byteArr1[j]);
}
}
catch (UnsupportedEncodingException g)
{
System.out.println("Unsupported character set" + g);
}
}
}
```

Output:

```
The input String is :
Welcome to JavaTpoint.

After converted into UTF-16BE the String is :
0870101010809901110109010103201160111032074097011809708401120111010501
100116046
```

Example5:

```
// Import statement
import java.io.*;

public class StringGetBytesExample5
{
public static void main(String argvs[])
{
// input string
String str = "Welcome to JavaTpoint.";
System.out.println("The input String is : ");
System.out.println(str + "\n");

// inside try block encoding is
// being done using different charsets
try
{
// ISO Latin Alphabet
byte[] byteArr2 = str.getBytes("ISO-8859-1");
System.out.println("After converted into ISO-8859-1 the String is :
");

for (int j = 0; j < byteArr2.length; j++)
{
     System.out.print(byteArr2[j]);
}
}
catch (UnsupportedEncodingException g)
{
System.out.println("Unsupported character set" + g);
}
}
}
```

Output:

```
The input String is :
Welcome to JavaTpoint.

After converted into ISO-8859-1 the String is :
87101108991111091013211611132749711897841121111051101646
```

Example6:

```
// Import statement
import java.io.*;

public class StringGetBytesExample6
{
```

```
public static void main(String argvs[])
{
// input string
String str = "Welcome to JavaTpoint.";
System.out.println("The input String is : ");
System.out.println(str + "\n");

// inside try block encoding is
// being done using different charsets
try
{
// Little Endian byte order, 16 - bit UCS Transformation format
byte[] byteArr3 = str.getBytes("UTF-16LE");
System.out.println("After converted into UTF-16LE the String is :
");

for (int j = 0; j < byteArr3.length; j++)
{
     System.out.print(byteArr3[j]);
}
}
catch (UnsupportedEncodingException g)
{
System.out.println("Unsupported character set" + g);
}
}
}
```

Output:

```
The input String is :
Welcome to JavaTpoint.

After converted into UTF-16LE the String is :
87010101080990111010901010320116011103207409701180970840112011101050
11001160460
```

27.4.19 Java String getChars()

The **Java String** class **getChars()** method copies the content of this string into a specified char array. There are four arguments passed in the getChars() method. The signature of the getChars() method is given below:

Signature

public void getChars(int srcBeginIndex, int srcEndIndex, char[] destination, int dstBeginIndex)

Parameters

int srcBeginIndex: The index from where copying of characters is started.
int srcEndIndex: The index which is next to the last character that is getting copied.
Char[] destination: The char array where characters from the string that invokes the getChars() method is getting copied.
int dstEndIndex: It shows the position in the destination array from where the characters from the string will be pushed.

Returns
It doesn't return any value.

Exception Throws
The method throws StringIndexOutOfBoundsException when any one or more than one of the following conditions holds true.
1. If srcBeginIndex is less than zero.
2. If srcBeginIndex is greater than srcEndIndex.
3. If srcEndIndex is greater than the size of the string that invokes the method
4. If dstEndIndex is is less than zero.
5. If dstEndIndex + (srcEndIndex - srcBeginIndex) is greater than the size of the destination array.

Example:

```
public class StringGetCharsExample
{
public static void main(String args[])
{
 String str = new String("hello javatpoint how r u");
      char[] ch = new char[10];
      try
      {
         str.getChars(6, 16, ch, 0);
         System.out.println(ch);
      }
      catch(Exception ex)
      {
        System.out.println(ex);
      }
}
}
```

Output:

```
javatpoint
```

27.4.20 Java String lastIndexOf()

The **Java String class lastIndexOf()** method returns the last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Signature

There are four types of lastIndexOf() method in Java. The signature of the methods are given below:

No.	Method	Description
1	int lastIndexOf(int ch)	It returns last index position for the given char value
2	int lastIndexOf(int ch, int fromIndex)	It returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	It returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	It returns last index position for the given substring and from index

Parameters

ch: char value i.e. a single character e.g. 'a'
fromIndex: index position from where index of the char value or substring is retured
substring: substring to be searched in this string

Returns

last index of the string

Java String lastIndexOf() method example

```
public class LastIndexOfExample1
{
      public static void main(String args[])
      {
            //there are 2 's' characters in this sentence
            String s1="this is index of example";
            //returns last index of 's' char value
            int index1=s1.lastIndexOf('s');
            System.out.println(index1);//6
      }
}
```

Output:

```
6
```

Java String lastIndexOf(int ch, int fromIndex) Method Example

Here, we are finding the last index from the string by specifying fromIndex.

```
public class LastIndexOfExample2
{
    public static void main(String[] args)
    {
        String str = "This is index of example";
        int index = str.lastIndexOf('s',5);
        System.out.println(index);
    }
}
```

Output:

```
3
```

Java String lastIndexOf(String substring) Method Example

It returns the last index of the substring.

```
public class LastIndexOfExample3
{
    public static void main(String[] args)
    {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of");
        System.out.println(index);
    }
}
```

Output:

```
19
```

Java String lastIndexOf(String substring, int fromIndex) Method Example

It returns the last index of the substring from the fromIndex.

```
public class LastIndexOfExample4
{
    public static void main(String[] args)
    {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of", 25);
        System.out.println(index);
        index = str.lastIndexOf("of", 10);
        System.out.println(index); // -1, if not found
    }
```

```
}
```

Output:

```
19
-1
```

27.4.21 Java String toCharArray()

The **java string toCharArray()** method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

Signature

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

public char[] toCharArray()

Returns

character array

Example1:

```
public class StringToCharArrayExample1
{
      public static void main(String args[])
      {
            String s1="hello";
            char[] ch=s1.toCharArray();
            for(int i=0;i<ch.length;i++)
            {
                  System.out.print(ch[i]);
            }
      }
}
```

Output:

```
hello
```

Example2:

Let's see one more example of char array. It is useful method which returns char array from the string without writing any custom code.

```
public class StringToCharArrayExample2
{
```

```
    public static void main(String[] args)
    {
        String s1 = "Welcome to Javatpoint";
        char[] ch = s1.toCharArray();
        int len = ch.length;
        System.out.println("Char Array length: " + len);
        System.out.println("Char Array elements: ");
        for (int i = 0; i < len; i++)
        {
            System.out.println(ch[i]);
        }
    }
}
```

Output:

```
Char Array length: 21
Char Array elements:
W
e
l
c
o
m
e

t
o

J
a
v
a
t
p
o
i
n
t
```

27.4.22 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

Example1:

```
public class StringTrimExample1
{
      public static void main(String args[])
      {
            String s1="  hello string   ";
            //without trim()
            System.out.println(s1+"javatpoint");
            //with trim()
            System.out.println(s1.trim()+"javatpoint");
      }
}
```

Output:

```
  hello string   javatpoint
hello stringjavatpoint
```

Example2:
The example demonstrates the use of the trim() method. This method removes all the trailing spaces so the length of the string also reduces. Let's see an example.

```
public class StringTrimExample2
{
    public static void main(String[] args)
    {
        String s1 ="  hello java string   ";
        System.out.println(s1.length());
        System.out.println(s1); //Without trim()
        String tr = s1.trim();
        System.out.println(tr.length());
        System.out.println(tr); //With trim()
    }
}
```

Output:

```
22
  hello java string
```

```
17
hello java string
```

Example3:

The trim() can be used to check whether the string only contains white spaces or not. The following example shows the same.

```
public class TrimExample3
{
// main method
public static void main(String argvs[])
{
String str = " abc ";
if((str.trim()).length() > 0)
{
System.out.println("The string contains characters other than white
spaces \n");
}
else
{
System.out.println("The string contains only white spaces \n");
}

str = "    ";
if((str.trim()).length() > 0)
{
System.out.println("The string contains characters other than white
spaces \n");
}
else
{
System.out.println("The string contains only white spaces \n");
}

}
}
```

Output:

```
The string contains characters other than white spaces

The string contains only white spaces
```

Example4:

Since strings in Java are immutable; therefore, when the trim() method manipulates the string

by trimming the whitespaces, it returns a new string. If the manipulation is not done by the trim() method, then the reference of the same string is returned. Observe the following example.

```
public class TrimExample4
{
// main method
public static void main(String argvs[])
{

// the string contains white spaces
// therefore, trimming the spaces leads to the
// generation of new string
String str = " abc ";

// str1 stores a new string
String str1 = str.trim();

// the hashcode of str and str1 is different
System.out.println(str.hashCode());
System.out.println(str1.hashCode() + "\n");

// no white space present in the string s
// therefore, the reference of the s is returned
// when the trim() method is invoked
String s = "xyz";
String s1 = s.trim();

// the hashcode of s and s1 is the same
System.out.println(s.hashCode());
System.out.println(s1.hashCode());

}
}
```

Output:

```
32539678
96354

119193
119193
```

27.4.23 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)
```

Returns

string representation of given value

Example1:

```
public class StringValueOfExample1
{
      public static void main(String args[])
      {
            int value=30;
            String s1=String.valueOf(value);
            //concatenating string with 10
            System.out.println(s1+10);
      }
}
```

Output:

```
3010
```

Example2:

This is a boolean version of overloaded valueOf() method. It takes boolean value and returns a string. Let's see an example.

```
public class StringValueOfExample2
{
    public static void main(String[] args)
    {
        // Boolean to String
        boolean bol = true;
        boolean bol2 = false;
        String s1 = String.valueOf(bol);
        String s2 = String.valueOf(bol2);
        System.out.println(s1);
        System.out.println(s2);
```

```
    }
}
```

Output:

```
true
false
```

Example3:

This is a char version of overloaded valueOf() method. It takes char value and returns a string. Let's see an example.

```
public class StringValueOfExample3
{
    public static void main(String[] args)
    {
        // char to String
        char ch1 = 'A';
        char ch2 = 'B';
        String s1 = String.valueOf(ch1);
        String s2 = String.valueOf(ch2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output:

```
A
B
```

Example4:

This is a float version of overloaded valueOf() method. It takes float value and returns a string. Let's see an example.

```
public class StringValueOfExample4
{
    public static void main(String[] args)
    {
        // Float and Double to String
        float f  = 10.05f;
        double d = 10.02;
        String s1 = String.valueOf(f);
        String s2 = String.valueOf(d);
        System.out.println(s1);
        System.out.println(s2);
    }
```

```
}
```

Output:

```
10.05
10.02
```

Example5:

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@5acf9800
```

27.4.24 Java String replace()

The **Java String class replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced that allows us to replace a sequence of char values.

Signature
There are two types of replace() methods in Java String class.

public String replace(char oldChar, char newChar)
public String replace(CharSequence target, CharSequence replacement)

The second replace() method is added since JDK 1.5.

Parameters
oldChar : old character
newChar : new character
target : target sequence of characters
replacement : replacement sequence of characters

Returns
replaced string

Exception Throws
NullPointerException: if the replacement or target is equal to null.

Example1:

```
public class ReplaceExample1
{
      public static void main(String args[])
      {
            String s1="javatpoint is a very good website";
            //replaces all occurrences of 'a' to 'e'
            String replaceString=s1.replace('a','e');
            System.out.println(replaceString);
```

```
    }
}
```

Output:

```
jevetpoint is e very good website
```

Example2:

```
public class ReplaceExample2
{
    public static void main(String args[])
    {
        String s1="my name is khan my name is java";
        //replaces all occurrences of "is" to "was"
        String replaceString=s1.replace("is","was");
        System.out.println(replaceString);
    }
}
```

Output:

```
my name was khan my name was java
```

Example3:

```
public class ReplaceExample3
{
    public static void main(String[] args)
    {
        String str = "oooooo-hhhh-oooooo";
        String rs = str.replace("h","s"); // Replace 'h' with 's'
        System.out.println(rs);
        rs = rs.replace("s","h"); // Replace 's' with 'h'
        System.out.println(rs);
    }
}
```

Output:

```
oooooo-ssss-oooooo
oooooo-hhhh-oooooo
```

Example4:

The replace() method throws the NullPointerException when the replacement or target is null. The following example confirms the same.

```
public class ReplaceExample4
```

```
{
// main method
public static void main(String argvs[])
{
String str = "For learning Java, JavaTpoint is a very good site.";
int size = str.length();

System.out.println(str);
String target = null;

// replacing null with JavaTpoint. Hence, the NullPointerException
// is raised.
str = str.replace(target, "JavaTpoint ");
System.out.println(str);

}
}
```

Output:

```
For learning Java, JavaTpoint is a very good site.
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "java.lang.CharSequence.toString()" because "target" is null
      at java.base/java.lang.String.replace(String.java:2958)
      at ReplaceExample4.main(ReplaceExample4.java:13)
```

27.4.25 Java String replaceAll()

The Java String class replaceAll() method returns a string replacing all the sequence of characters matching regex and replacement string.

Signature

public String replaceAll(String regex, String replacement)

Parameters

regex : regular expression

replacement : replacement sequence of characters

Returns

replaced string

Exception Throws

PatternSyntaxException: if the syntax of the regular expression is not valid.

Example1:
Let's see an example to replace all the occurrences of a single character.

```
public class ReplaceAllExample1
{
      public static void main(String args[])
      {
            String s1="javatpoint is a very good website";
            //replaces all occurrences of "a" to "e"
            String replaceString=s1.replaceAll("a","e");
            System.out.println(replaceString);
      }
}
```

Output:

```
jevetpoint is e very good website
```

Example2:
Let's see an example to replace all the occurrences of a single word or set of words.

```
public class ReplaceAllExample2
{
public static void main(String args[])
{
String s1="My name is Khan. My name is Bob. My name is Sonoo.";
//replaces all occurrences of "is" to "was"
String replaceString=s1.replaceAll("is","was");
System.out.println(replaceString);
}
}
```

Output:

```
My name was Khan. My name was Bob. My name was Sonoo.
```

Example3:
Let's see an example to remove all the occurrences of white spaces.

```
public class ReplaceAllExample3
{
public static void main(String args[])
{
String s1="My name is Khan. My name is Bob. My name is Sonoo.";
String replaceString=s1.replaceAll("\\s","");
System.out.println(replaceString);
}
```

```
}
```

Output:

```
MynameisKhan.MynameisBob.MynameisSonoo.
```

Example4:

The replaceAll() method throws the PatternSyntaxException when there is an improper regular expression. Look at the following example.

```
public class ReplaceAllExample4
{
// main method
public static void main(String argvs[])
{

// input string
String str = "For learning Java, JavaTpoint is a very good site.";

System.out.println(str);

String regex = "\\"; // the regular expression is not valid.

// invoking the replaceAll() method raises the
PatternSyntaxException
str = str.replaceAll(regex, "JavaTpoint ");

System.out.println(str);

}
}
```

Output:

```
For learning Java, JavaTpoint is a very good site.
Exception in thread "main" java.util.regex.PatternSyntaxException:
Unexpected internal error near index 1
\
     at java.base/java.util.regex.Pattern.error(Pattern.java:2028)
     at
java.base/java.util.regex.Pattern.compile(Pattern.java:1789)
     at java.base/java.util.regex.Pattern.<init>(Pattern.java:1430)
     at
java.base/java.util.regex.Pattern.compile(Pattern.java:1069)
     at java.base/java.lang.String.replaceAll(String.java:2942)
     at ReplaceAllExample4.main(ReplaceAllExample4.java:15)
```

Example5:
The replaceAll() method can also be used to insert spaces between characters.

```
public class ReplaceAllExample5
{
      // main method
      public static void main(String argvs[])
      {
      // input string
      String str = "JavaTpoint";
      System.out.println(str);
      String regex = "";
      // adding a white space before and after
      // every character of the input string.
      str = str.replaceAll(regex, " ");
      System.out.println(str);
      }
}
```

Output:

```
JavaTpoint
 J a v a T p o i n t
```

Example6:
Even the null regular expression is also not accepted by the replaceAll() method as the NullPointerException is raised.

```
public class ReplaceAllExample6
{
      // main method
      public static void main(String argvs[])
      {

      // input string
      String str = "JavaTpoint";
      System.out.println(str);

      String regex = null; // regular expression is null

      str = str.replaceAll(regex, " ");

      System.out.println(str);

      }
}
```

Output:

```
JavaTpoint
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.isEmpty()" because "this.pattern" is null
      at java.base/java.util.regex.Pattern.<init>(Pattern.java:1428)
      at
java.base/java.util.regex.Pattern.compile(Pattern.java:1069)
      at java.base/java.lang.String.replaceAll(String.java:2942)
      at ReplaceAllExample6.main(ReplaceAllExample6.java:13)
```

27.4.26 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.

Returns

array of strings

Throws

PatternSyntaxException if pattern for regular expression is invalid

Example1:

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

```
public class SplitExample1
{
      public static void main(String args[])
      {
            String s1="java string split method by javatpoint";
            //splits the string based on whitespace
            String[] words=s1.split("\\s");
```

```
            //using java foreach loop to print elements of string
            array
            for(String w:words)
            {
                  System.out.println(w);
            }
      }
}
```

Output:

```
java
string
split
method
by
javatpoint
```

Example2:

```
public class SplitExample2
{
      public static void main(String args[])
      {
            String s1="welcome to split world";
            System.out.println("returning words:");
            for(String w:s1.split("\\s",0))
            {
                  System.out.println(w);
            }
            System.out.println("returning words:");
            for(String w:s1.split("\\s",1))
            {
                  System.out.println(w);
            }
            System.out.println("returning words:");
            for(String w:s1.split("\\s",2))
            {
                  System.out.println(w);
            }
      }
}
```

Output:

```
returning words:
welcome
to
split
world
```

```
returning words:
welcome to split world
returning words:
welcome
to split world
```

Example3:

Here, we are passing split limit as a second argument to this function. This limits the number of splitted strings.

```
public class SplitExample3
{
    public static void main(String[] args)
    {
        String str = "Javatpointttt";
        System.out.println("Returning words:");
        String[] arr = str.split("t", 0);
        for (String w : arr)
        {
            System.out.println(w);
        }
        System.out.println("Split array length: "+arr.length);
    }
}
```

Output:

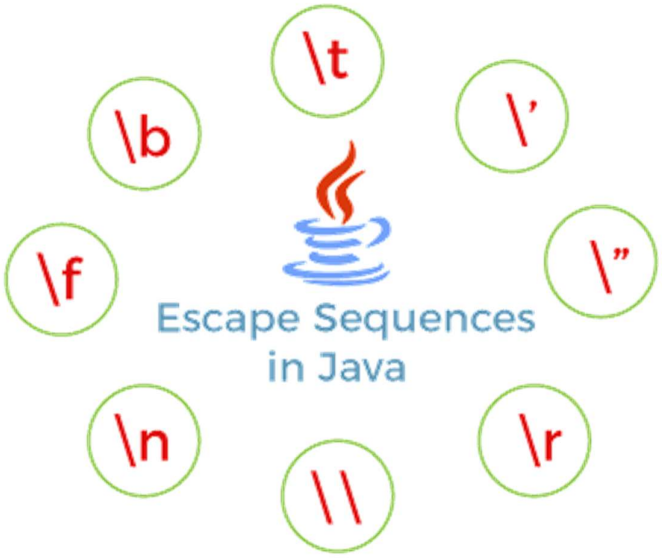
```
Returning words:
Java
poin
Split array length: 2
```

27.5 Escape Characters in Java

In this section, we will discuss **Java escape characters** or **escape sequences**. Also, we will use these **escape sequences or characters in a Java program**.

What are escape characters?

In Java, if a character is preceded by a **backslash** (\) is known as **Java escape sequence** or **escape characters**. It may include **letters, numerals, punctuations,** etc. Remember that escape characters must be enclosed in **quotation marks** (""). These are the valid character literals. The Java compiler interprets these characters as a single character that adds a specific meaning to the compiler.



List of Java Escape Characters

In Java, there is a total of eight escape sequences that are described in the following table.

Escape Characters	Description
\t	It is used to insert a **tab** in the text at this point.
\'	It is used to insert a **single quote** character in the text at this point.
\"	It is used to insert a **double quote** character in the text at this point.
\r	It is used to insert a **carriage return** in the text at this point.
\\	It is used to insert a **backslash character** in the text at this point.
\n	It is used to insert a **new line** in the text at this point.
\f	It is used to insert a **form feed** in the text at this point.
\b	It is used to insert a **backspace** in the text at this point.

Why do we use escape characters?

Let's understand the uses of escape characters through the following example. Suppose, we have to print the following statement with double quotes:

"Java" is an object-oriented programming language.

The following statements do not print Java enclosed in quotation marks.

```
System.out.println("Java is an object-oriented programming language.");
System.out.println(""Java" is an object-oriented programming language.");
```

While we compile the program with the above two statements, the compiler gives errors, as shown below.

```
/EscapeSequenceDemo.java:6: error: ')' expected
System.out.println(""Java" is an object-oriented programming language.");
                    ^
/EscapeSequenceDemo.java:6: error: ';' expected
System.out.println(""Java" is an object-oriented programming language.");
                          ^
2 errors
```

In such a case, the compiler needs to be told that quotation marks do not signal the start or end of a string, but instead are to be printed. The following statement prints statements with **quotation marks.**

```
System.out.println("\"Java\" is an object-oriented programming language.");
```

Example:

```
public class EscapeCharaterExample
{
public static void main(String args[])
{
//it inserts a Tab Space
String str = "Andrew\tGarfield";
System.out.println(str);
//it inserts a New Line
String str1 = "the best way\nto communicate \nan idea \nis to act it
out";
System.out.println(str1);
//it insert a backslash
String str2 = "And\\Or";
System.out.println(str2);
//it insert a Carriage
String str3 = "Carriage\rReturn";
System.out.println(str3);
//it prints a single quote
String str4 = "Wall Street\'s";
System.out.println(str4);
//it prints double quote
//String str5 = "New\'Twilight'Line";
String str5 = "'JavaTpoint'";
System.out.println(str5);
}
}
```

Output:

```
Andrew     Garfield
the best way
to communicate
an idea
is to act it out
And\Or
Carriage
Return
Wall Street's
'JavaTpoint'
```

27.6 Unicode Escape Characters in Java

Java also supports **Unicode escape characters.** A Unicode escape character consists of a backslash (/) followed by one or more **u** characters and **four** hexadecimal digits (**\uxxxx**). Here, \uxxxx represents **\u0000** to **\uFFFF**.

While interpreting the string if the compiler finds something in the Unicode representation, the compiler replaces it with a respective symbol according to the Java specification.

List of Unicode Character or Escape Sequence

The following table describes the widely used Unicode Character Sequence.

Char	Unicode	Escape Sequence	Description
Special Codes			
	U+0009	\u0009	Horizontal Tab
	U+000A	\u000A	Line Feed
	U+000D	\u000D	Carriage Return / Enter
	U+00A0	\u00A0	Non-Breaking Space
Symbols Codes			
&	U+0026	\u0026	Ampersand
•	U+2022	\u2022	Bullet

?	U+25E6	\u25E6	White Bullet
·	U+2219	\u2219	Bullet Operator
‣	U+2023	\u2023	Triangular Bullet
⁃	U+2043	\u2043	Hyphen Bullet
°	U+00B0	\u00B0	Degree
∞	U+221E	\u221E	Infinity
Currency Codes			
$	U+0024	\u0024	Dollar
€	U+20AC	\u20AC	Euro
£	U+00A3	\u00A3	Pound
¥	U+00A5	\u00A5	Yen / Yuan
¢	U+00A2	\u00A2	Cent
₹	U+20B9	\u20B9	Indian Rupee
Rs	U+20A8	\u20A8	Rupee
₱	U+20B1	\u20B1	Peso
₩	U+20A9	\u20A9	Korean Won
฿	U+0E3F	\u0E3F	Thai Baht
₫	U+20AB	\u20AB	Dong
₪	U+20AA	\u20AA	Shekel
Intellectual Property Codes			
©	U+00A9	\u00A9	Copyright
®	U+00AE	\u00AE	Registered Trademark
℗	U+2117	\u2117	Sound Recording Copyright
™	U+2122	\u2122	Trademark

℠	U+2120	\u2120	Service Mark
Greek Alphabet Codes			
α	U+03B1	\u03B1	Small Alpha
β	U+03B2	\u03B2	Small Beta
γ	U+03B3	\u03B3	Small Gamma
δ	U+03B4	\u03B4	Small Delta
ε	U+03B5	\u03B5	Small Epsilon
ζ	U+03B6	\u03B6	Small Zeta
η	U+03B7	\u03B7	Small Eta
θ	U+03B8	\u03B8	Small Theta
ι	U+03B9	\u03B9	Small Iota
κ	U+03BA	\u03BA	Small Kappa
λ	U+03BB	\u03BB	Small Lambda
μ	U+03BC	\u03BC	Small Mu
ν	U+03BD	\u03BD	Small Nu
ξ	U+03BE	\u03BE	Small Xi
ο	U+03BF	\u03BF	Small Omicron
π	U+03C0	\u03C0	Small Pi
ρ	U+03C1	\u03C1	Small Rho
σ	U+03C3	\u03C3	Small Sigma
τ	U+03C4	\u03C4	Small Tau
υ	U+03C5	\u03C5	Small Upsilon
φ	U+03C6	\u03C6	Small Phi
χ	U+03C7	\u03C7	Small Chi
ψ	U+03C8	\u03C8	Small Psi
ω	U+03C9	\u03C9	Small Omega

Α	U+0391	\u0391	Capital Alpha
Β	U+0392	\u0392	Capital Beta
Γ	U+0393	\u0393	Capital Gamma
Δ	U+0394	\u0394	Capital Delta
Ε	U+0395	\u0395	Capital Epsilon
Ζ	U+0396	\u0396	Capital Zeta
Η	U+0397	\u0397	Capital Eta
Θ	U+0398	\u0398	Capital Theta
Ι	U+0399	\u0399	Capital Iota
Κ	U+039A	\u039A	Capital Kappa
Λ	U+039B	\u039B	Capital Lambda
Μ	U+039C	\u039C	Capital Mu
Ν	U+039D	\u039D	Capital Nu
Ξ	U+039E	\u039E	Capital Xi
Ο	U+039F	\u039F	Capital Omicron
Π	U+03A0	\u03A0	Capital Pi
Ρ	U+03A1	\u03A1	Capital Rho
Σ	U+03A3	\u03A3	Capital Sigma
Τ	U+03A4	\u03A4	Capital Tau
Υ	U+03A5	\u03A5	Capital Upsilon
Φ	U+03A6	\u03A6	Capital Phi
Χ	U+03A7	\u03A7	Capital Chi
Ψ	U+03A8	\u03A8	Capital Psi
Ω	U+03A9	\u03A9	Capital Omega

Example:

```
public class UnicodeCharacterExample
{
public static void main(String args[])
{
System.out.println("\"Example of Unicode Character Sequence\",
\u00A9 2021 JavaTpoint");
}
}
```

Output:

```
"Example of Unicode Character Sequence", © 2021 JavaTpoint
```