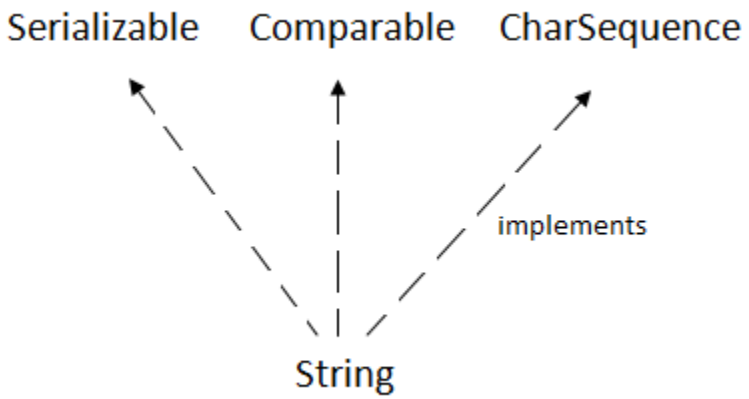
Java String

In Java, a string is an object that represents a sequence of characters or char values. The *java.lang.String* class is used to create a Java string object.

Java String class provides a lot of methods to perform operations on strings such as

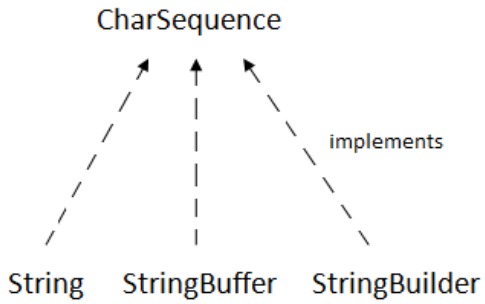
1. String charAt()
2. String compareTo()
3. String concat()
4. String contains()
5. String endsWith()
6. String equals()
7. equalsIgnoreCase()
8. String format()
9. String getBytes()
10. String getChars()
11. String indexOf()
12. String intern()
13. String isEmpty()
14. String join()
15. String lastIndexOf()
16. String length()
17. String replace()
18. String replaceAll()
19. String split()
20. String startsWith()
21. String substring()
22. String toCharArray()
23. String toLowerCase()
24. String toUpperCase()
25. String trim()
26. String valueOf()

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



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, **StringBuffer** and **StringBuilder** classes implement it. It means, we can create strings in Java by using these three classes.



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

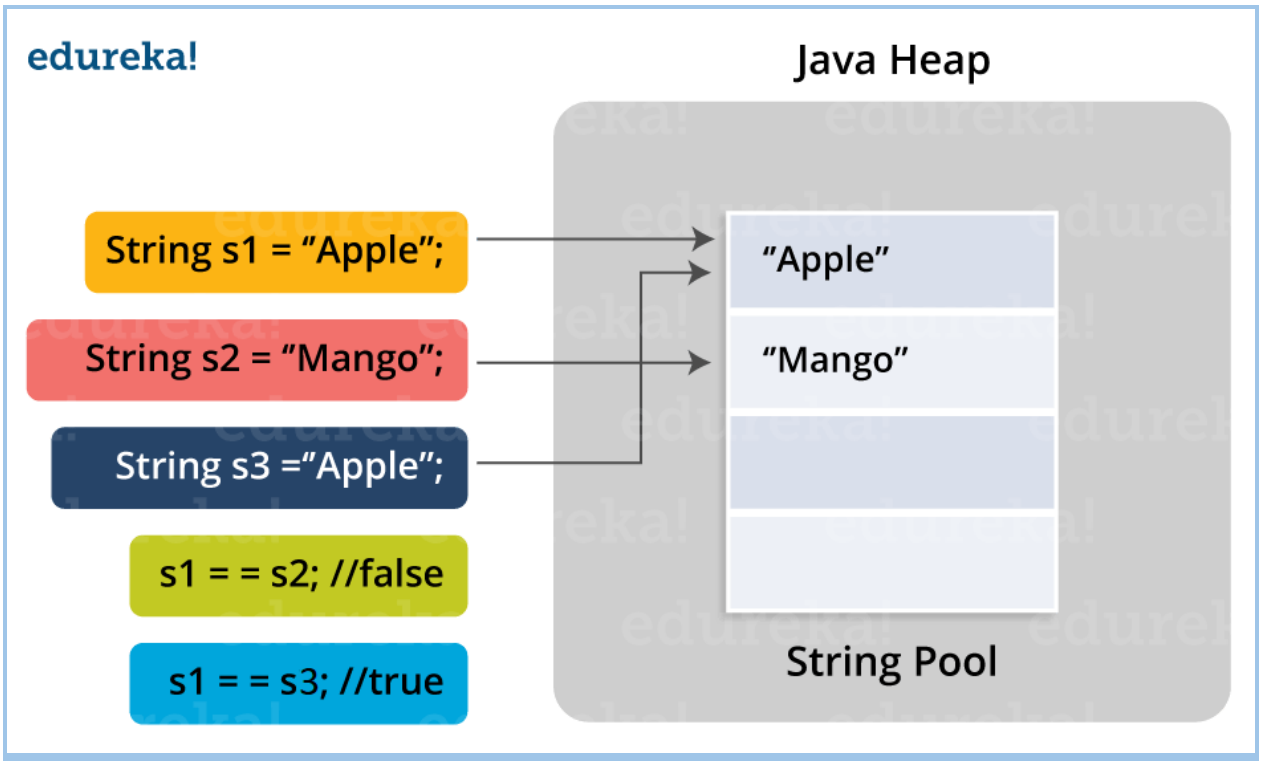
There are two ways to create a String object:

1. **By string literal** : Java String literal is created by using double quotes.
   For Example: String s="Java String";
2. **By new keyword** : Java String is created by using a keyword "new".
   For example: String s=new String("Java String");

It creates two objects (in String pool and in heap) and one reference variable where the variable 's' will refer to the object in the heap.

Now, let us understand the concept of the Java String pool.

Java String Pool: Java String pool refers to a collection of Strings which are stored in heap memory. In this, whenever a new object is created, String pool first checks whether the object is already present in the pool or not. If it is present, then the same reference is returned to the variable, else a new object will be created in the String pool and the respective reference will be returned. Refer to the diagrammatic representation for better understanding :



In the above image, two Strings are created using literal i.e "Apple" and "Mango". Now, when a third String is created with the value "Apple", instead of creating a new object, the already present object reference is returned. That's the reason Java String pool came into the picture.

Before we go ahead, One key point I would like to add is that unlike other data types in Java, Strings are immutable. By immutable, we mean that Strings are constant, their values cannot be changed after they are created. Because String objects are immutable, they can be shared. For example:

```
String str ="abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};

String str = new String(data);
```

Let us now look at some of the inbuilt methods in String class.

Let’s Explain Java String class methods :

1. 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 a **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 a **StringIndexOutOfBoundsException** if the index is a negative value or greater than this string length.

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

        String name = "javaString";
        char ch = name.charAt(2);
        System.out.println(ch); // return : v

    }
}

```

2. 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.

1. **if s1 > s2, it returns positive number**
2. **if s1 < s2, it returns negative number**
3. **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.

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

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

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

```java
public class solution {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "";
        String s3 = "me";
        System.out.println(s1.compareTo(s2)); // 5
        System.out.println(s2.compareTo(s3)); // -2
    }
}

```

3. String concat() :

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

Syntax

```
public String concat(String anotherString)
```

Parameter

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

Returns

combined string

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

    }
}

```

4. 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*.

Syntax

```
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.

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

```

Limitations of the Contains() method

Following are some limitations of the contains() method:

- The **contains() method** should not be used to search for a character in a string. Doing so results in an error.
- 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.

5. 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.

Syntax

```
public boolean endsWith(String suffix)
```

Parameter

suffix : Sequence of character

Returns

true or false

```java
public class solution {
    public static void main(String[] args) {
        String s1 = "javaString";
        System.out.println(s1.endsWith("t")); // true
        System.out.println(s1.endsWith("strong")); // false
    }
}

```

6. 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.

Syntax

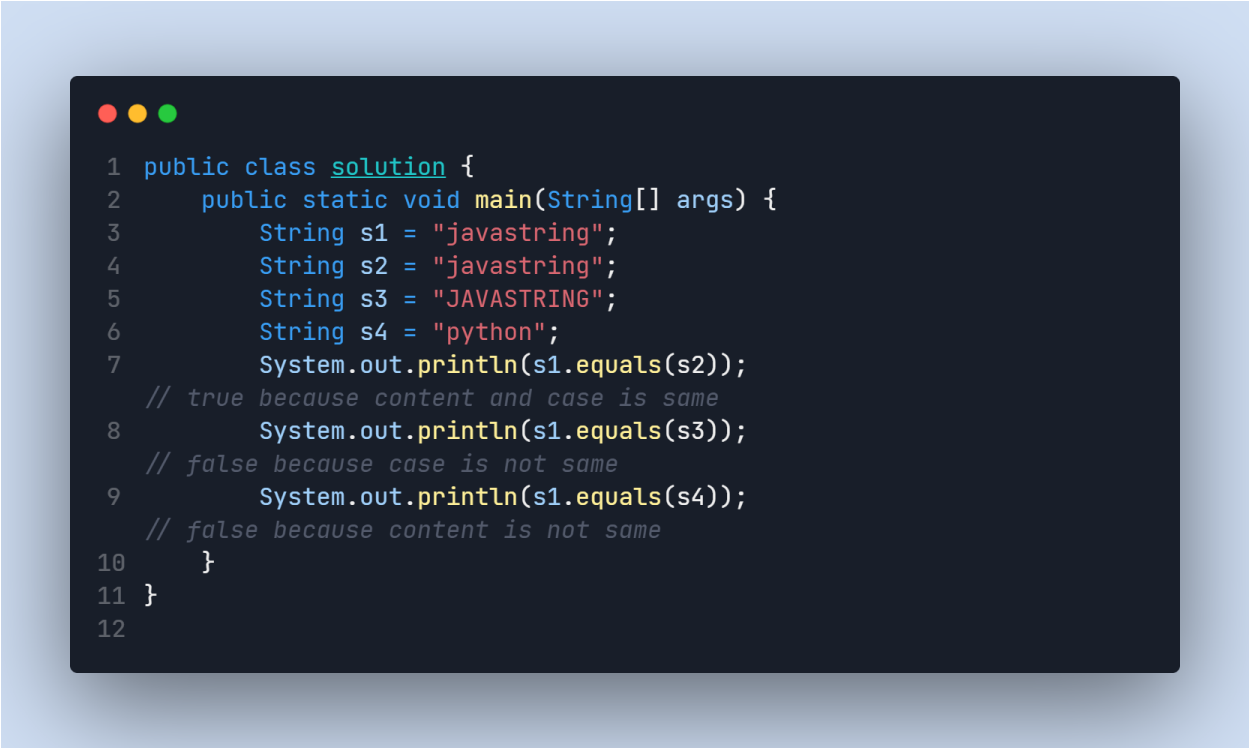
```
public boolean equals(Object anotherObject)
```

Parameter

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

Returns

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



```java
public class solution {
    public static void main(String[] args) {
        String s1 = "javastring";
        String s2 = "javastring";
        String s3 = "JAVASTRING";
        String s4 = "python";
        System.out.println(s1.equals(s2));
// true because content and case is same
        System.out.println(s1.equals(s3));
// false because case is not same
        System.out.println(s1.equals(s4));
// false because content is not same
    }
}

```

The internal implementation of the **equals() method** shows that one can pass the reference of any object in the parameter of the method. The following example shows the same.

```java
public class solution {
    public static void main(String[] args) {
        // 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)); // false
        // reference of the Integer object is passed
        System.out.println(str1.equals(i));// false
        // reference of the Float object is passed
        System.out.println(str2.equals(f));// false
        // reference of the Boolean object is passed
        System.out.println(str3.equals(b));// false
        // 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()));// true
        System.out.println(str1.equals(i.toString()));// true
        System.out.println(str2.equals(f.toString()));// true
        System.out.println(str3.equals(b.toString()));// true
    }
}

```

7. 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.

Syntax

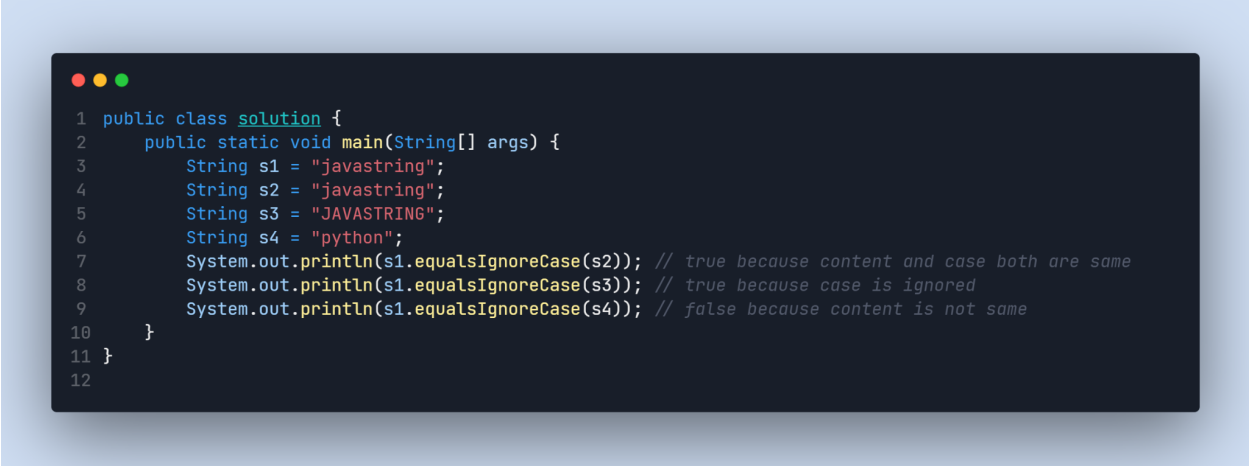
public boolean equalsIgnoreCase(String str)

Parameter:

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

Returns

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



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

8. String format() :

In java, ***String format() method*** returns a formatted string using the given **locale**, specified **format string**, and **arguments**. We can concatenate the strings using this method and at the same time, we can format the output concatenated string.

Syntax

public boolean equalsIgnoreCase(String str)

Parameter:

- *The locale value to be applied on the format() method*
- *The format of the output string.*
- *args specifying the number of arguments for the format string. It may be zero or more.*

Returns

formatted string

Exception Thrown:

- ***NullPointerException****: If the format is null.*
- ***IllegalFormatException****: If the format specified is illegal or there are insufficient arguments.*

```java
public class solution {
    public static void main(String[] args) {
        String name = "java";
        String sf1 = String.format("name is %s", name);
        String sf2 = String.format("value is %f", 32.33434);
        String sf3 = String.format("value is 2.12f"  , 32.33434);
  // returns 12 char fractional part filling with 0
        System.out.println(sf1); // name is java
        System.out.println(sf2); // value is 32.334340
        System.out.println(sf3); // value is 32.334340000000
    }
}

```

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.*

9. String getBytes() :

getbytes() function in java is used to convert a string into a sequence of bytes and returns an array of bytes.

The **java.lang.String.getBytes(String charsetName)** method encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

Syntax

```
public byte[] getBytes()

public byte[] getBytes(Charset charset)
```

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.

This function can be implemented in two ways. Both ways are discussed in this article.

Syntax 1 – public byte[] getBytes() : This function takes no arguments and uses the default charset to encode the string into bytes.

```java
public class solution {
    public static void main(String[] args) {
        String gfg = "ASTHA GFG";
        // converting the string into byte
        // using getBytes ( converts into ASCII values )
        byte[] b = gfg.getBytes();
        // Displaying converted string after conversion
        System.out.println("The String after conversion is : ");
        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i]);
// The String after conversion is : 658384726532717071
        }
    }
}

```

Syntax 2 : public byte[] getBytes(Charset charset): This implementation accepts the charset according to which string has to be encoded while conversion into bytes. There are many charset defined and are discussed below.

- **US-ASCII:** Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
- **ISO-8859-1:** ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
- **UTF-8:** Eight-bit UCS Transformation Format
- **UTF-16BE:** Sixteen-bit UCS Transformation Format, big-endian byte order
- **UTF-16LE:** Sixteen-bit UCS Transformation Format, little-endian byte order
- **UTF-16:** Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark.

```java

import java.io.*;
public class solution {
    public static void main(String[] args) {
        // Initializing String
        String gfg = new String("ASTHA GFG");
        try {
            // converting the string into byte
            // using getBytes ( converts into UTF-16 values )
            byte[] b = gfg.getBytes("UTF-16");
            // Displaying converted string after conversion
            // into UTF-16
            System.out.println(
"The String after conversion into UTF-16 is : ");
            for (int i = 0; i < b.length; i++) {
                System.out.print(b[i]);
            }
            System.out.print("\n");
            // converting the string into byte
            // using getBytes ( converts into UTF-16BE values )
            byte[] c = gfg.getBytes("UTF-16BE");
            // Displaying converted string after conversion
            // into UTF-16BE
            System.out.println(
"The String after conversion into UTF-16BE is : ");
            for (int i = 0; i < c.length; i++) {
                System.out.print(c[i]);
            }
        } catch (UnsupportedEncodingException g) {
            System.out.println("Unsupported character set" + g);
        }
    }
}

output:
The String after conversion into UTF-16 is :
-2-106508308407206503207107007 1
The String after conversion into UTF-16BE is :
0650830840720650320710700 71
```

10. 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.

Syntax

```
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.

- If **srcBeginIndex** is less than zero.
- If **srcBeginIndex** is greater than srcEndIndex.
- If **srcEndIndex** is greater than the size of the string that invokes the method
- If **dstEndIndex** is less than zero.
- If **dstEndIndex + (srcEndIndex - srcBeginIndex)** is greater than the size of the destination array.

```java
public class solution {
    public static void main(String[] args) {
        String str = new String("hello javaString 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); // javaString
        }
    }
}
```

The method throws an exception if index value exceeds array range. Let's see an example.

```java
public class solution {
    public static void main(String[] args) {
        String str = new String("Welcome to JavaString");
        char[] ch  = new char[20];
        try {
            str.getChars(1, 26, ch, 0);
            System.out.println(ch);
        } catch (Exception e) {
            System.out.println(e);  // java.lang.StringIndexOutOfBoundsException: offset 10, count 14, length 20
        }
    }
}
```

11. String indexOf() :

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

There are four overloaded **indexOf()** methods 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*

1. int indexOf() : This method **returns** the **index** within this string of the **first** occurrence of the specified character or -1, if the character does not occur.

Syntax

int indexOf (char ch)

Parameters

ch a character.

```java
public class solution {
    public static void main(String[] args) {
        String gfg = new String("Welcome to javaString");
        System.out.print("Found j first at position : ");
        System.out.println(gfg.indexOf('j'));//Found g first at position : 11
    }
}
```

2. int indexOf(char ch, int strt) : This method **returns** the index within this string of the **first** occurrence of the specified character, starting the search at the specified index or -1, if the character does not occur.

Syntax

int indexOf (char ch, int start)

Parameters

ch a character.

Start the index to start the search from.

```
1
2 public class solution {
3     public static void main(String[] args) {
4         String gfg = new String("Welcome to javaString");
5         System.out.print("Found n after 13th index at position : ");
6         System.out.println(gfg.indexOf('n', 13)); //Found n after 13th index at position : 19
7     }
8 }
```

3. int indexOf(String str) : This method **returns** the index within this string of the **first** occurrence of the specified **substring**. If it does not occur as a substring, -1 is returned.

Syntax

int indexOf (String str)

Parameters

str a string.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to javaString");
        String subst = new String("java");
        System.out.print("Found java starting at position : ");
        System.out.print(Str.indexOf(subst)); //Found java starting at position : 11
    }
}
```

4. int indexOf(String str, int strt) : This method **returns** the index within this string of the **first** occurrence of the specified **substring**, **starting** at the specified **index**. If it does not occur, -1 is returned.

Syntax

int indexOf (String str, int start)

Parameters

start the index to start the search from.

str a string.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to javaStringjava");
        String subst = new String("java");
        System.out.print("Found java(after 14th index) starting at position : ");
        System.out.print(Str.indexOf(subst, 14)); //Found java(after 14th index) starting at position : 21
    }
}
```

12. String intern() :

More info: What is Java String interning? ref. stackoverflow

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

It can be used to return a 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.

Syntax

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 JavaString.");
String str1 = new String("Welcome to JavaString");
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 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 JavaString.").intern();
String str1 = new String("Welcome to JavaString.").intern();
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.

```java
public class solution {
    public static void main(String[] args) {
        String s1 = "javaString";
        String s2 = s1.intern();
        String s3 = new String("javaString");
        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
    }
}
```

Points to Remember

Following are some important points to remember regarding the intern() method:

1) A string literal always invokes the intern() method, whether one mentions 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.

13. 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.

Syntax

Public boolean isEmpty()

Returns

True if length is 0 otherwise false.

```java

public class solution {
    public static void main(String[] args) {
        String s1 = "";
        String s2 = "javaString";

        System.out.println(s1.isEmpty()); // true
        System.out.println(s2.isEmpty()); // false
    }
}
```

14. String join() :

The **java.lang.string.join()** method concatenates the given elements with the delimiter and returns the concatenated string.Note that if an element is null, then null is added.The join() method is included in java string since JDK 1.8.

There are two types of join() methods in java string.

Syntax

public static String join(CharSequence delimiter, 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.

Since

1.8

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

15. 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.

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 returned

substring: substring to be searched in this string

Returns

last index of the string

There are **four** variants of lastIndexOf() method.This article depicts about all of them, as follows:

1. lastIndexOf() : This method **returns** the index of the **last** occurrence of the character in the character sequence.

Syntax

int lastIndexOf(int ch)

Parameters:

ch : a character.

Return Value:

This method returns the index.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to javaString");
        System.out.print("Found Last Index of a at : ");
        System.out.println(Str.lastIndexOf('a', 13)); //Found Last Index of a at : 12
    }
}
```

2. lastIndexOf(int ch, int beg) : This method **returns** the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to beg, or -1 if the character does not occur before that point.

Syntax

public int lastIndexOf(int ch, int beg)

Parameters:

ch : a character.
beg : the index to start the search from.

Returned Value:

This method returns the index.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to javaString");
        System.out.print("Found Last Index of a at : ");
        System.out.println(Str.lastIndexOf('a')); //Found Last Index of a at : 14
    }
}
```

3. lastIndexOf(String str) : This method accepts a String as an argument, if the string argument occurs one or more times as a substring within this object, then it returns the index of the **first** character of the **last** such substring is returned. If it does not occur as a substring, -1 is returned.

Syntax

public int lastIndexOf(String str)

Parameters:

str : a string.

Returned Value:

This method returns the index.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to javaStringJava");
        System.out.print("Found substring java at : ");
        System.out.println(Str.lastIndexOf("java")); //Found substring java at : 11
    }
}
```

4. lastIndexOf(String str, int beg) : This method **returns** the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

Syntax

public int lastIndexOf(String str, int beg)

Parameters

beg : the index to start the search from.
str : a string.

Return Value

This method returns the index.

```java
public class solution {
    public static void main(String[] args) {
        String Str = new String("Welcome to JavaStringJava");
        System.out.print("Found substring Java at : ");
        System.out.println(Str.lastIndexOf("Java", 15)); //Found substring Java at : 11
    }
}
```

16. 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.

Syntax

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, which 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 the length variable.

17. 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.

Syntax

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 has been 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.

```java
public class solution {
    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);  //oooooo-ssss-oooooo
        rs = rs.replace("s","h"); // Replace 's' with 'h'
        System.out.println(rs);  //oooooo-hhhh-oooooo
    }
}
```

18. String replaceAll() :

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

Syntax

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.

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

19. String Split() :

The **string split()** method breaks a given string around matches of the given regular expression. After splitting against the given regular expression, this method returns a char array.

Following are the two variants of the split() method in Java:

1. Public String [ ] split (String regex, int limit)

Parameters:

- **regex –** a delimiting regular expression
- **Limit -** the resulting threshold

Returns:

An array of strings is computed by splitting the given string.

Throws:

PatternSyntaxException – if the provided regular expression's syntax is invalid.

The limit parameter can have 3 values:

- **limit > 0 –** If this is the case, then the pattern will be applied at most limit-1 times, the resulting array's length will not be more than n, and the resulting array's last entry will contain all input beyond the last matched pattern.
- **limit < 0 –** In this case, the pattern will be applied as many times as possible, and the resulting array can be of any size.
- **limit = 0 –** In this case, the pattern will be applied as many times as possible, the resulting array can be of any size, and trailing empty strings will be discarded.

```java
public class solution {
    public static void main(String[] args) {
        String str1 = "java@string@java";
        String[] arrOfStr1 = str1.split("@", 2);
        for (String a : arrOfStr1) {
            System.out.println(a);
            // java
            // string@java
        }

        String str2 = "java@string@java";
        String[] arrOfStr2 = str2.split("@", 5);
        for (String a : arrOfStr2) {
            System.out.println(a);
            // java
            // string
            // java
        }
    }
}
```

2. public String[ ] split(String regex)

This variant of the split method takes a regular expression as a parameter and breaks the given string around matches of this regular expression regex. Here, by default limit is 0.

Parameters:

regex – a delimiting regular expression

Returns:

An array of strings is computed by splitting the given string.

Throws:

PatternSyntaxException – if the provided regular expression's syntax is invalid.

20. 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.

Syntax

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

The startsWith() method considers the case-sensitivity of characters

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

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

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

21. 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.

Syntax

public String substring(int startIndex)
public String substring(int startIndex, int endIndex)

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

Parameters

startIndex : starting index is inclusive
endIndex : ending index is exclusive

Returns

specified string

Exception Throws

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

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

```java
public class solution {
    public static void main(String[] args) {
        String s1 = "JavaString";
        System.out.println(s1.substring(2, 4));// returns va
        System.out.println(s1.substring(2));// returns vaString
        String substr = s1.substring(0); // Starts with 0 and goes to end
        System.out.println(substr);// JavaString
        String substr2 = s1.substring(5, 10); // Starts from 5 and goes to 10
        System.out.println(substr2); // tring
        String substr3 = s1.substring(5, 15); // Returns Exception
    }
}
```

More information:

22. String toCharArray() :

The **java string toCharArray()** method converts this string into a 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.

Internal implementation

```
1. public char[] toCharArray() {
2.     // Cannot use Arrays.copyOf because of class initialization order
   issues
3.     char result[] = new char[value.length];
4.     System.arraycopy(value, 0, result, 0, value.length);
5.     return result;
6.  }
```

Syntax

public char[] toCharArray()

Returns

character array

```java
public class solution {
    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]);  //hello
        }
    }
}
```

23. String toLowerCase() :

The **java string toLowerCase()** method converts all characters of the string into lowercase letters. There are two types of **toLowerCase()** method.

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

Syntax

There are two variants of the toLowerCase() method.

public String toLowerCase()

public String toLowerCase(Locale locale)

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

Returns

string in lowercase letter.

```java
import java.util.Locale;
public class solution {
    public static void main(String[] args) {
        String s1 = "JAVASTRING HELLO stRIng";
        String s1lower = s1.toLowerCase();
        System.out.println(s1lower); // javastring hello string
        // This method allows us to pass locale too for the various langauges where we
        // are getting string in english and turkish both.
        String s = "JAVASTRING HELLO stRIng";
        String eng = s.toLowerCase(Locale.ENGLISH);
        System.out.println(eng); // javastring hello string
        String turkish = s.toLowerCase(Locale.forLanguageTag("tr")); // It shows i without dot
        System.out.println(turkish); // javastr?ng hello str?ng
    }
}
```

24. 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 letters.

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

Syntax

There are two variants of the toUpperCase() method.

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

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

Returns

string in uppercase letter.

```java
import java.util.Locale;
public class solution {
    public static void main(String[] args) {
        String s1 = "hello string";
        String s1upper = s1.toUpperCase();
        System.out.println(s1upper); // HELLO STRING
        String s = "hello string";
        String turkish = s.toUpperCase(Locale.forLanguageTag("tr"));
        String english = s.toUpperCase(Locale.forLanguageTag("en"));
        System.out.println(turkish);// HELLO STR?NG
        System.out.println(english); // HELLO STRING
    }
}
```

25. 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.***

Syntax

public String trim()

Returns

string with omitted leading and trailing spaces

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

26. String trim() :

The **java string valueOf()** method converts different types of values into strings. 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.

Internal implementation

```
1. public static String valueOf(Object obj) {
2.     return (obj == null) ? "null" : obj.toString();
3.  }
```

Syntax

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

```java
public class solution {
    public static void main(String[] args) {
        int value=30;
        String s1=String.valueOf(value);
        System.out.println(s1+10);//3010
        boolean bol = true;
        boolean bol2 = false;
        String s2 = String.valueOf(bol);
        String s3 = String.valueOf(bol2);
        System.out.println(s2); //true
        System.out.println(s3);  //false
        float f  = 10.05f;
        double d = 10.02;
        String s4 = String.valueOf(f);
        String s5 = String.valueOf(d);
        System.out.println(s4); //10.05
        System.out.println(s5);  //10.02
    }
}
```

Let's start Java Character Class....

Java Character Class – Implement Methods with Coding Examples

We know that there are 8 primitive *data types in Java*. Java's primitive data types are the data values and not objects. Java Character Class wraps the value of Primitive data type in an object.

There are some cases when we may encounter a situation where numerical values are needed but in the form of objects. Java solves this problem by providing the concept of wrapper classes.

Wrapper classes are part of Java's standard library java.lang and these convert primitive data type into an object.

Java provides 8 wrapper classes which are:

Datatype	***Wrapper Class***
boolean	***Boolean***
byte	***Byte***
char	***Character***
short	***Short***
int	***Integer***
long	***Long***
float	***Float***
double	***Double***

Java Character Class

Character Class in Java which wraps the value of primitive data type char into its object. All the attributes, methods, and constructors of the Character class are specified by the Unicode Data file which is maintained by the ***Unicode Consortium.***

The instances or objects of Character class can hold single character data. In addition, this wrapper class also provides several methods useful when manipulating, inspecting or dealing with the single-character data.

Syntax of creating an object from the Character class is as follows:

```
Character letter = new Character( 'g' );
Character num = new Character( '7' );
```

We created the object of the Character wrapper class using the *Character constructor*. In the above syntax, the Java compiler will automatically convert the 'char' value into an object of Character type.

This process of converting primitive data type into an object is called autoboxing and, the reverse process, that is, converting the object into primitive data types is called unboxing.

The Character Class is immutable, which means, once we create its object, we cannot change it.

The above syntax can also be written like this:

```
Character char1 = new Character( 'g' );
// primitive char 'g' is wrapped in a Character object char1.
Character char2 = new Character( char1 );
//primitive value of char char1 is wrapped in a Character object char2.
```

Escape Sequences

An escape sequence is a character preceded by a backslash(\), which gives a different meaning to the compiler. The following table shows the escape sequences in Java:

Escape Sequence	*Description*
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a new line in the text at this point
\r	Inserts a carrier return in the text at this point
\f	Inserts a form feed in the text at this point
\’	Inserts a single quote character in the text at this point
\”	Inserts a double quote character in the text at this point
****	Inserts a backslash in the text at this point

Code to illustrate some escape sequences:

```java
public class solution {
    public static void main(String[] args) {
        System.out.print("Hello \nWelcome to"); // using \n
        System.out.println(" The \"TechVidvan\" tutorial."); // using \"
        System.out.println("This is a \'Java\' tutorial."); // using \'
        System.out.println("My java file is in: projects\\src\\java"); // using \\
        // Output:
        // Hello
        // Welcome to The “TechVidvan” tutorial.
        // This is a ‘Java’ tutorial.
        // This is a “TechVidvan” tutorial.
        // My java file is in: projects\src\java
    }}
```

Methods of Character Class in Java

The Character Class comes with several methods that are useful for performing operations on characters. These methods are static in nature, that is, they can be directly called with the help of class names without creating any object.

1. static boolean isDigit(char ch) Method

The method isDigit() is used to determine whether the specific character value (ch) is a digit or not. It checks whether the value is a digit that is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

As it is a boolean method, it returns true if the character is a digit and false if character value is not a digit.

Code to illustrate the use of the isDigit() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 'A';
    char ch1 = '1';
    //checks whether the values of ch and cha1 are digits or not.
    System.out.println(Character.isDigit(ch)); //false
    System.out.println(Character.isDigit(ch1)); //true
    //checks whether the values 't' , '8', ' H' are digits or not.
    System.out.println(Character.isDigit('t')); //false
    System.out.println(Character.isDigit('8')); //true
    System.out.println(Character.isDigit('H')); //false
    }}
```

2. static boolean isLetter(char ch) Method

The method isLetter() is used to determine whether the specific character value (ch) is a letter or not. It checks whether the value is a letter that is, [A – Z] or [a – z]. As it is a boolean method, it returns true if the character is a letter and false if character value is not a letter.

We can also write the ASCII value of the letter because Java can implicitly typecast the value from char to int.

Code to illustrate the use of the isLetter() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 65; //Passing an ASCII value 65 which is equal to 'A'
        char ch1 = '3';
        //checks whether the values of ch and ch1 are letters or not
        System.out.println(Character.isLetter(ch)); //true
        System.out.println(Character.isLetter(ch1)); //false
        //checks whether the 'b', '8' and 'H' are letters or not
        System.out.println(Character.isLetter('b')); //true
        System.out.println(Character.isLetter('8')); //false
        System.out.println(Character.isLetter('H')); //true
    }}
```

3. static boolean isWhiteSpace(char ch) Method

Whitespace in Java can be considered as space, tab, or a new line, and the method isWhiteSpace() determines whether the given char(ch) is whitespace or not. As it is also a boolean method, it returns true if the character is whitespace and false if character value is not whitespace.

Code to illustrate the use of the isWhiteSpace() method:

```java
public class solution {
    public static void main(String[] args) {
        System.out.println(Character.isWhitespace('W')); //false
        System.out.println(Character.isWhitespace(' ')); //true
        System.out.println(Character.isWhitespace(0)); //false
        System.out.println(Character.isWhitespace('t')); //false
        System.out.println(Character.isWhitespace('\n')); //true
        System.out.println(Character.isWhitespace('\t')); //true
        System.out.println(Character.isWhitespace('\b')); //false
    }}
```

4. static boolean isUpperCase(char ch) Method

The method isUpperCase() is used to determine whether the specific character value (ch) is an uppercase letter or not. It checks whether the value is a letter that is, [A – Z].

As it is a boolean method, it returns true if the character is in uppercase or capital letter and false if character value is not an uppercase letter.

Code to illustrate the use of the isUpperCase() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 78;
        //here the value in the numeric is the ASCII value of N
        System.out.println(Character.isUpperCase(ch)); //true
        //checks whether 'B' and 'b' are in uppercase or not
        System.out.println(Character.isUpperCase('B')); //true
        System.out.println(Character.isUpperCase('b')); //false
    }}
```

5. static boolean isLowerCase(char ch) Method

The method isLowerCase() is used to determine whether the specific character value (ch) is a lowercase letter or not. It checks whether the value is a letter that is, [a – z].

As it is a boolean method, it returns true if the character is in lowercase or a small letter and false if character value is not a lowercase letter.

Code to illustrate the use of the isLowerCase() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 78;
        //here the value in the numeric is the ASCII value of N
        System.out.println(Character.isLowerCase(ch)); //false
        //checks whether 'f', 'B' and 'b' are in Lowercase or not
        System.out.println(Character.isLowerCase('f')); //true
        System.out.println(Character.isLowerCase('B')); //false
        System.out.println(Character.isLowerCase('b')); //true
    }}
```

6. static char toUpperCase(char ch) Method

The method toUpperCase() is used to convert the specific character value (ch) into an uppercase letter. It returns the uppercase form of the input char value.

Code to illustrate the use of the toUpperCase() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 122;    //ASCII value of z is 122
        char ch1 = 108;    //ASCII value of l is 108
        System.out.println(Character.toUpperCase(ch)); //Z
        System.out.println(Character.toUpperCase(ch1)); //L
        System.out.println(Character.toUpperCase('a')); //A
        System.out.println(Character.toUpperCase('t'));//T
        System.out.println(Character.toUpperCase('S')); //S
    }}
```

7. static char toLowerCase(char ch) Method

The method toLowerCase() is used to convert the specific character value (ch) into a lowercase letter. It returns the lowercase form of the input char value.

Code to illustrate the use of the toLowerCase() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 66; // ASCII value of B is 66
        char ch1 = 90; // ASCII value of Z is 90
        System.out.println(Character.toLowerCase(ch)); //b
        System.out.println(Character.toLowerCase(ch1)); //z
        System.out.println(Character.toLowerCase('A')); //a
        System.out.println(Character.toLowerCase('R')); //r
        System.out.println(Character.toLowerCase('e')); //e
    }}
```

8. static String toString(char ch) Method

The toString(char ch) method in Java returns an object of String class for the specified char value. In simple words, it converts a char value into String.

We cannot use ASCII value in this method because this method is of String type and the ASCII value cannot be converted to the character value directly.

Code to illustrate the use of the toString() method:

```java
public class solution {
    public static void main(String[] args) {
        char ch = 'R';
        // the character will be printed as it is
        System.out.println(Character.toString(ch)); //R
        System.out.println(Character.toString('A')); //A
        System.out.println(Character.toString('b')); //b
        System.out.println(Character.toString('C')); //c
    }
}
```

Java String compare

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

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

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1) By Using equals() Method

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

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

Teststringcomparison1.java

```
1. class Teststringcomparison1{
2.  public static void main(String args[]){
3.    String s1="Java";
4.    String s2="Java";
5.    String s3=new String("Java");
6.    String s4="String";
7.    System.out.println(s1.equals(s2));//true
8.    System.out.println(s1.equals(s3));//true
9.    System.out.println(s1.equals(s4));//false
10.    } }
```

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

Teststringcomparison2.java

```
1. class Teststringcomparison2{
2.  public static void main(String args[]){
3.    String s1="Java";
4.    String s2="JAVA";
5.
6.    System.out.println(s1.equals(s2));//false
7.    System.out.println(s1.equalsIgnoreCase(s2));//true
8.  }  }
```

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 the same case.* ***equalsIgnoreCase()*** *returns true regardless of cases of strings.*

2) By Using == operator

The == operator compares references not values.

```
1. class Teststringcomparison3{
2.  public static void main(String args[]){
3.    String s1="Java";
4.    String s2="Java";
5.    String s3=new String("Java");
6.    System.out.println(s1==s2);//true (because both refer to
   same instance)
7.    System.out.println(s1==s3);//false(because s3 refers to
   instance created in nonpool)
8.  }
9. }
```

3) By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if the 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.

Teststringcomparison4.java

```
1.  class Teststringcomparison4{
2.   public static void main(String args[]){
3.     String s1="Java";
4.     String s2="Java";
5.     String s3="String";
6.     System.out.println(s1.compareTo(s2));//0
7.     System.out.println(s1.compareTo(s3));//1(because s1>s3)
8.     System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9.   }
10.    }
```

String Concatenation in Java

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

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

1) String Concatenation by + (String concatenation) operator

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

TestStringConcatenation1.java

```
1. class TestStringConcatenation1{
2.  public static void main(String args[]){
3.    String s="Java"+"String";
4.    System.out.println(s);//Java String
5.  }
6. }
```

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

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

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

TestStringConcatenation2.java

```
1. class TestStringConcatenation2{
2.  public static void main(String args[]){
3.    String s=50+30+"string"+40+40;
4.    System.out.println(s);//80string4040
```

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

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

2) String Concatenation by concat() method

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

Syntax

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

TestStringConcatenation3.java

```
1. class TestStringConcatenation3{
2.  public static void main(String args[]){
3.    String s1="Java ";
4.    String s2="String";
5.    String s3=s1.concat(s2);
6.    System.out.println(s3);//Java String
7.   }
8. }
```

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

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class

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

StrBuilder.java

```
1.  public class StrBuilder
2.  {
3.      /* Driver Code */
4.      public static void main(String args[])
5.      {
6.          StringBuilder s1 = new StringBuilder("Hello");    //String 1
7.          StringBuilder s2 = new StringBuilder(" World");    //String 2
8.          StringBuilder s = s1.append(s2);   //String 3 to store the result
9.              System.out.println(s.toString());  //Displays result
10.         }
11.     }
```

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

2. String concatenation using format() method

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

StrFormat.java

```
1.  public class StrFormat
2.  {
```

```
3.     /* Driver Code */
4.     public static void main(String args[])
5.     {
6.         String s1 = new String("Hello");    //String 1
7.         String s2 = new String(" World");    //String 2
8.         String s = String.format("%s%s",s1,s2);   //String 3 to store the
   result
9.             System.out.println(s.toString());  //Displays result
10.        }
11.    }
```

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

StringBuffer class

StringBuffer is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

- *java.lang.StringBuffer extends (or inherits from) Object class.*
- *All Implemented Interfaces of **StringBuffer** class: **Serializable, Appendable, CharSequence.***
- *public final class StringBuffer extends Object implements **Serializable, CharSequence, Appendable.***
- *String buffers are safe for use by **multiple threads**. The methods can be **synchronized** wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.*
- *Whenever an operation occurs involving a source sequence (such as **appending or inserting** from a source sequence) this class **synchronizes** only on the **string buffer** performing the operation, not on the source.*
- *It inherits some of the methods from the Object class such as **clone(), equals(), finalize(), getClass(), hashCode(), notifies(), notifyAll().***

Constructors of StringBuffer class

1. StringBuffer(): It reserves room for 16 characters without reallocation

```
StringBuffer s = new StringBuffer();
```

2. StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.

```
StringBuffer s = new StringBuffer(20);
```

3. StringBuffer(String str): It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s = new StringBuffer("javaString");
```

Methods of StringBuffer class

Methods	Action Performed
append()	*Used to add text at the end of the existing text.*
length()	*The length of a StringBuffer can be found by the length() method.*
capacity()	*The total allocated capacity can be found by the capacity() method.*
charAt()	*This method returns the char value in this sequence at the specified index.*
delete()	*Deletes a sequence of characters from the invoking object.*
deleteCharAt()	*Deletes the character at the index specified by loc.*
ensureCapacity()	*Ensures capacity is at least equal to the given minimum.*
insert()	*Inserts text at the specified index position.*
length()	*Returns the length of the string.*
reverse()	*Reverse the Characters within a StringBuffer object.*
replace()	*Replace one set of characters with another set inside a StringBuffer object.*
indexOf(String str)	*This method returns the index within this string of the first occurrence of the specified substring.*
toString()	*This method returns a string representing the data in this sequence.*

```java
public class solution {
    public static void main(String[] args) {
        StringBuffer s = new StringBuffer();
        s.append("java");
        s.append("string");
        System.out.println("After append : " + s); // javastring
        s.insert(2, "for");
        System.out.println("After insert 2 : " + s); // jaforvastring
        char string_arr[] = { 'j', 'a', 'v', 'a' };
        s.insert(1, string_arr);
        System.out.println("After insert string_arr : " + s); // jjavaaforvastring
        s.reverse();
        System.out.println("Reserse : " + s); // gnirtsavrofaavajj
        s.delete(0, 5);
        System.out.println("After delete 0 to 5 index : " + s); // savrofaavajj
        s.replace(1, 3, "Hello");
        System.out.println("After replace 1 to 3 index : " + s); // sHellorofaavajj
        System.out.println("Capacity : " + s.capacity()); //34
    }
}
```

More Tutorial:

- **Java StringBuilder Class**
- **Difference between String and StringBuffer**
- **Difference between StringBuffer and StringBuilder**
- **Java toString() Method**