Module 3

String Handling

In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.
In Java, string is basically an object that represents sequence of char values. An arrayof characters works same as Java string. For example:

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

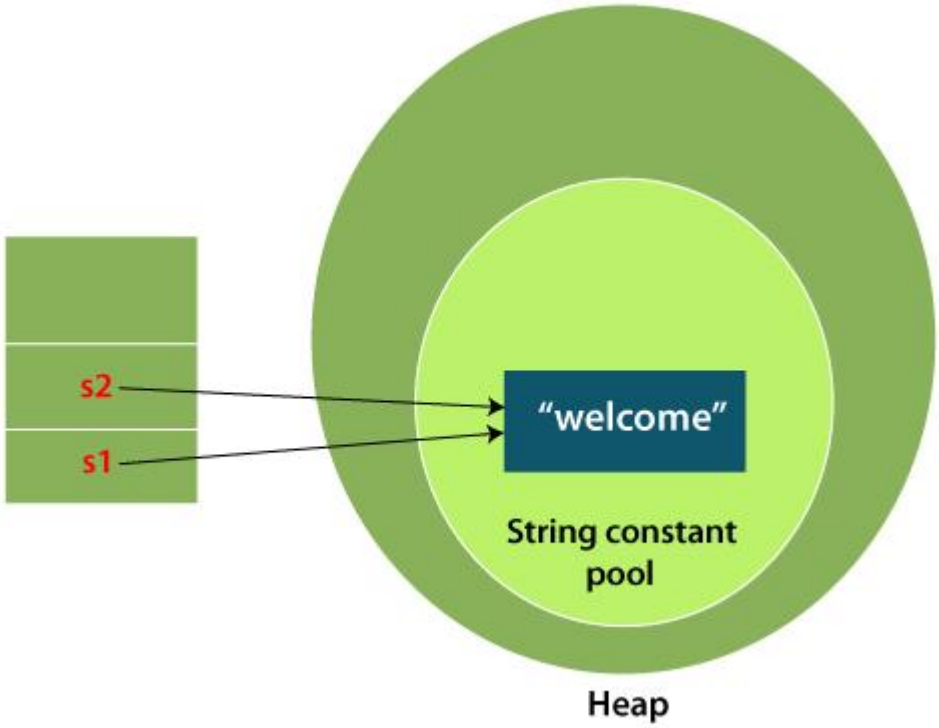
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1.	String s="welcome";
```

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

```
1.	String s1="Welcome";
2.	String s2="Welcome";//It doesn't create a new instance
```



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

2) By new keyword

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

In such case, JVM

will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
```

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

Output:

```
java
strings
example
```

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

String Constructors

String s = new String();

will create an instance of String with no characters in it

```
char chars[] = { 'a', 'b',
'c' }; String s = new
String(chars);
// Construct one String from
another.class MakeString {
public static void main(String args[])
{char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
```

```
System.out.println(s2);

} }

// Construct string from subset of char array.

class SubStringCons {

 public static void main(String args[])

 {byte ascii[] = {65, 66, 67, 68, 69, 70 };

String s1 = new String(ascii);

System.out.println(s1);

String s2 = new String(ascii, 2, 3);

System.out.println(s2); } }
```

This program generates the following output:

ABCDEF

CDE

Java String class methods

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

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

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.

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

		specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

String Operations

String Length

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Concatenating String

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using + operator

```java
public class Demo{

    public static void main(String[] args) {
        String s = "Hello";
        String str = "Java";
        String str1 = s.concat(str);
        System.out.println(str1);
    }
}
```

```java
public class Demo{
    public static void main(String[] args) {
        String s = "Hello";
        String str = "Java";
        String str1 = s+str;
        String str2 = "Java"+11;
        System.out.println(str1);
        System.out.println(str2);
    }
}
```

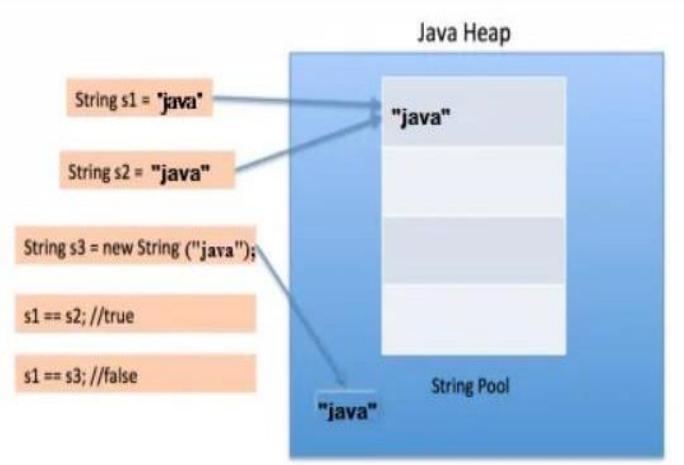
String Comparison

We can compare Strings in three Ways

1. Using equals() and **equalsIgnoreCase()** methods
2. Using == operator
3. By CompareTo() method

```java
public class Demo{
    public static void main(String[] args) {
        String s = "Hell";
        String s1 = "Hello";
        String s2 = "Hello";
        boolean b = s1.equals(s2);    //true
        System.out.println(b);
        b = s.equals(s1) ;   //false
        System.out.println(b);
    }
}
```

```java
public class Demo{
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java";
        String s3 = new String ("Java");
        boolean b = (s1 == s2);     //true
        System.out.println(b);
        b = (s1 == s3);      //false
        System.out.println(b);
    }
}
```




```java
public class HelloWorld{
    public static void main(String[] args) {
        String s1 = "Abhi";
        String s2 = "Viraaj";
        String s3 = "Abhi";
        int a = s1.compareTo(s2);     //return -21 because s1 < s2
        System.out.println(a);
        a = s1.compareTo(s3);     //return 0 because s1 == s3
        System.out.println(a);
        a = s2.compareTo(s1);     //return 21 because s2 > s1
        System.out.println(a);
    }
}
```

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Character Extraction

charAt()

char ch;

ch = "abc".charAt(1);

getChars()

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

Here is the output of this program:

```
demo
```

startsWith( ) and endsWith()

"Foobar".endsWith("bar") and

"Foobar".startsWith("Foo") are both true.

"Foobar".startsWith("bar", 3)

Searching Strings

indexOf( ) Searches for the first occurrence of a character or substring.
lastIndexOf( ) Searches for the last occurrence of a character orsubstring.

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

replace()

String s = "Hello".replace('l', 'w');
puts the string "Hewwo" into s.

trim( )

The trim( ) method returns a copy of the invoking string from which any leadingand trailing whitespace has been removed.
It has this general form:
String trim( )
Here is an example: String s = " Hello World ".trim();

Changing the Case of Characters Within a String

String

toLowerCase( )

String

toUpperCase( )

String s = "This is a test.";

String upper =

s.toUpperCase();

String lower = s.toLowerCase();

Substring in Java

public String substring(int startIndex)

public String substring(int startIndex, int endIndex)

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

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:

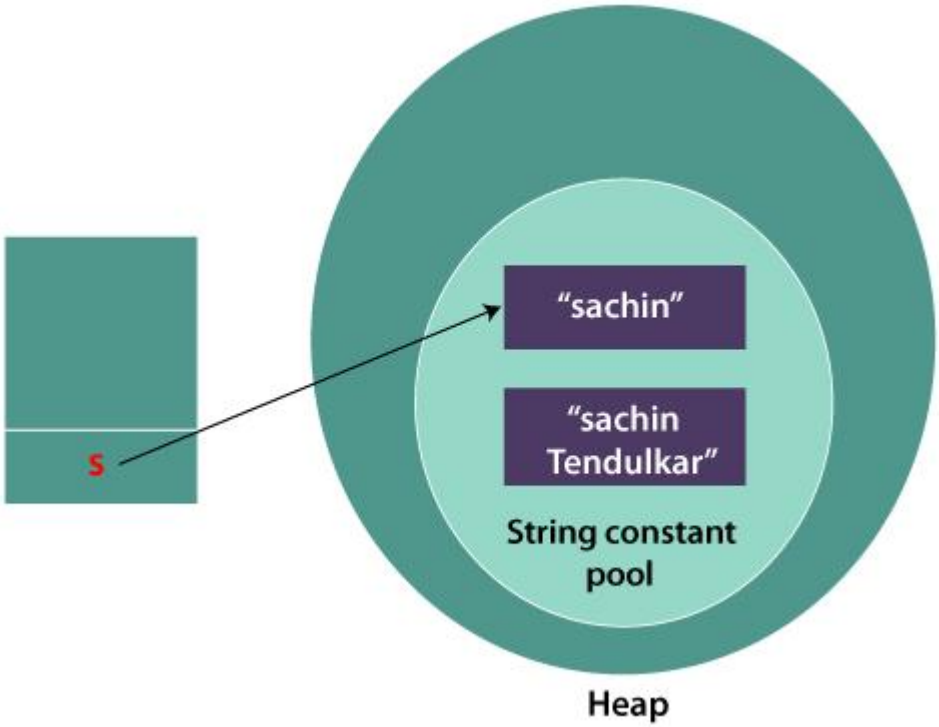
Testimmutablestring.java

```
1.  class Testimmutablestring{
2.   public static void main(String args[]){
3.     String s="Sachin";
4.     s.concat(" Tendulkar");//concat() method appends the string at the end
5.     System.out.println(s);//will print Sachin because strings are immutable objects
6.   }
7.  }
```

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:

Testimmutablestring1.java

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   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.

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.

StringBuffer

StringBuffer supports a modifiable string. As you know, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writable character sequences.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int size)

StringBuffer(String str)

StringBuffer(CharSequence chars)

length( ) and capacity( )

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

charAt( ) and setCharAt( )

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " + sb.charAt(1));

    sb.setCharAt(1, 'i');
    sb.setLength(2);
    System.out.println("buffer after = " + sb);
    System.out.println("charAt(1) after = " + sb.charAt(1));
  }
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

append( )

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

insert( )

```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");


    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

The output of this example is shown here:

```
I like Java!
```

reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

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

delete( ) and deleteCharAt( )

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");


    sb.delete(4, 7);
    System.out.println("After delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

Here is the output:

```
After replace: This was a test.
```

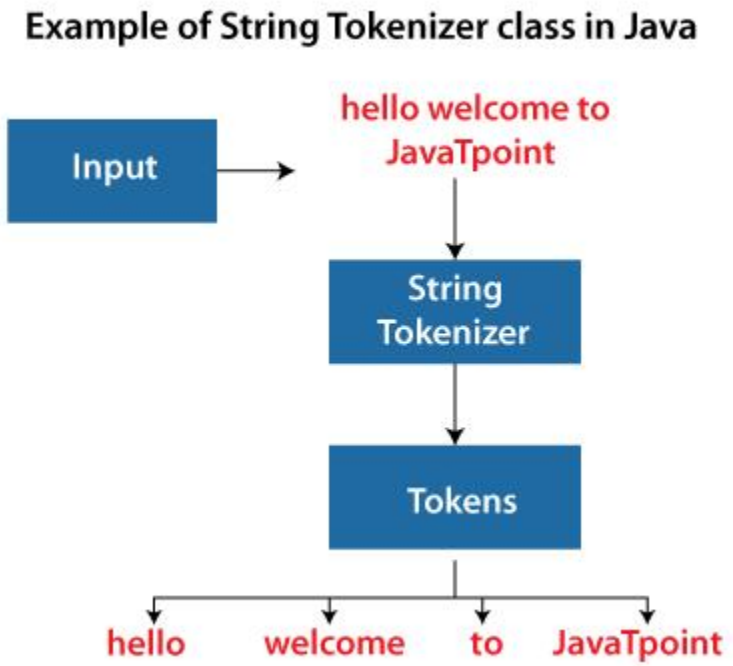
StringTokenizer in Java

- StringTokenizer
- Methods of StringTokenizer
- Example of StringTokenizer

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

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

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



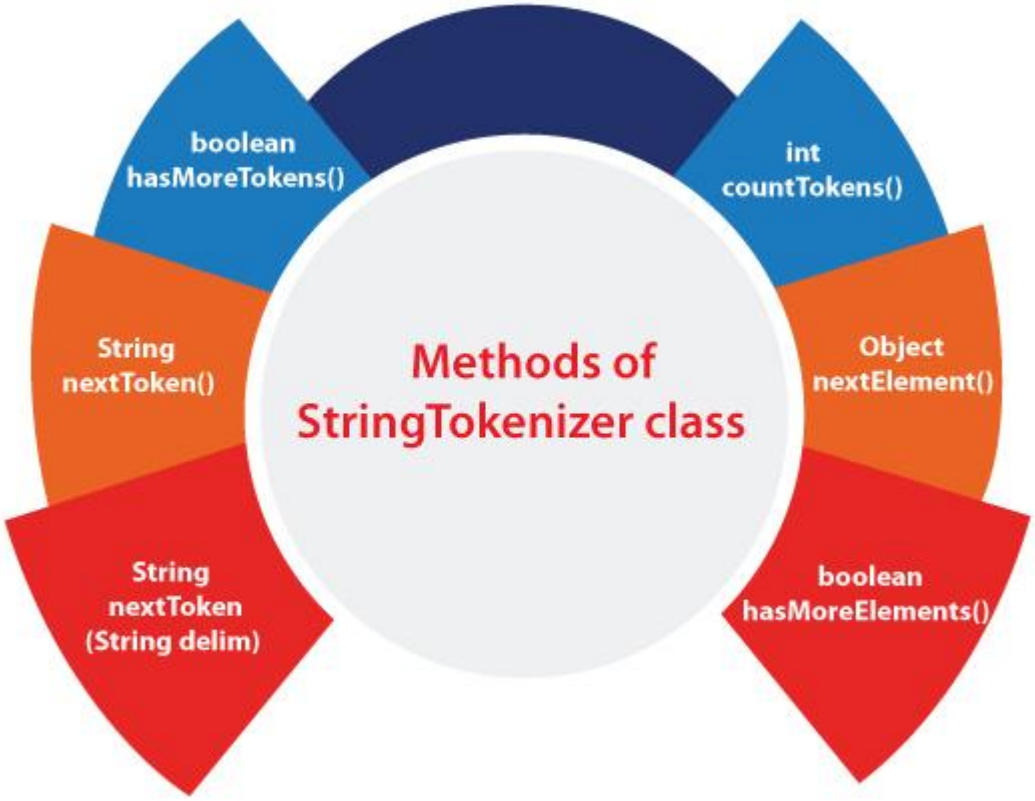
Constructors of the StringTokenizer Class

There are 3 constructors defined in the StringTokenizer class

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

Methods of the StringTokenizer Class

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



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

Example of StringTokenizer Class

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

Simple.java

```
import java.util.StringTokenizer;
```

```
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
    while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
    }
   }
}
```

Output:

```
my
name
is
khan
```

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

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

Test.java

```
. import java.util.*;
.
. public class Test {
.   public static void main(String[] args) {
.       StringTokenizer st = new StringTokenizer("my,name,is,khan");
.
.       // printing next token
.       System.out.println("Next token is : " + st.nextToken(","));
.   }
. }
. Output:
```

Next token is : my

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

Example of hasMoreTokens() method of the StringTokenizer class

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

StringTokenizer1.java

```
import java.util.StringTokenizer;
public class StringTokenizer1
{
 /* Driver Code */
 public static void main(String args[])
 {
  /* StringTokenizer object */
  StringTokenizer st = new StringTokenizer("Demonstrating methods from StringTokenizer class"," ");

    /* Checks if the String has any more tokens */
    while (st.hasMoreTokens())
    {
       System.out.println(st.nextToken());
    }
 }
}
```

Output:

```
Demonstrating
methods
from
StringTokenizer
class
```

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

Example of hasMoreElements() method of the StringTokenizer class

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

StringTokenizer2.java

```
import java.util.StringTokenizer;
public class StringTokenizer2
{
 public static void main(String args[])
 {
  StringTokenizer st = new StringTokenizer("Hello everyone I am a Java developer"," ");
    while (st.hasMoreElements())
    {
       System.out.println(st.nextToken());
    }
 }
}
```

Output:

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

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

Example of nextElement() method of the StringTokenizer class

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

StringTokenizer3.java

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

Output:

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

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

Example of countTokens() method of the StringTokenizer class

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

StringTokenizer4.java

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

Output:

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

Exception Handling

Errors are broadly classified into two categories

Compile-time Errors: All syntax errors will be detected and displayed by the java compiler and therefore these errors are called compile time errors. When there is compile-time error, .class file will not be generated.

Examples: Missing semicolons, Missing braces, Use of undeclared variable etc.,

Run-time Errors: A program may compile successfully creating .class file, but may not run properly. Such programs may produce wrong results due to wrong logic or even may terminate due to errors such as stack overflow. Exception in java is an indication of some unusual event.

Examples: Dividing by zero
ArrayIndexOutOfBounds
NullPointerException
Class not found Exception etc.,

Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. An exception can occur for many different reasons. Following are some scenarios where an exceptionoccurs.

A user has entered an invalid data.

A file that needs to be opened cannot be found.

A network connection has been lost in the middle of communications or the JVM has run out of memory.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. If the exception object is not caught and handled properly, the java interpreter will display an error message and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as **Exception Handling**.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:

Types of Exceptions

There are three categories of Exceptions. They are

1. Checked Exception
2. Unchecked Exception
3. Error

1.Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions.
E.g. IOException, SQLException etc. Checked exceptions are checked at compile-time

2.Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation. the classes which inherit RuntimeException are known as unchecked exceptions.
E.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime

3.Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation. Error is irrecoverable.
E.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Using try and catch

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following

```
try{
        // protected code
  }catch(Exception e){
        // catch block
  }
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block.

// Using try and catch for Exception Handling

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
     //code that may raise exception
     int data=100/0;
   }catch(ArithmeticException e)
          {
          System.out.println(e);
          }
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

// Catching invalid Command line arguments

```
// Integer values to be given as command line arguments, if not, exception is raised and handled
  public class Example1{
  public static void main(String
        args[]){int invalid=0;
        int number, count=0;
```

```
   for(int i=0;i<args.length;i++)
          {
            try{
                  number=Integer.parseInt(args[i]);
                }catch(NumberFormatException e)
                  {
          invalid=invalid+1;
          System.out.println("Invalid Number"+ args[i]);
          continue;
          }
          count=count+1;
     }
       System.out.println("Valid Nos"+count);
      System.out.println("Invalid Nos"+invalid);
    }
  }
```

Nested try statements

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

Example: //To demonstrate nested try statements

```
class Example2
{       public static void main(String args[])
          {
            try
              {    int a=10,b=5, c=5, d;
                   int p[ ]={2,4};
                   p[4]=10;
                   try{
                      d=a/(b-c);
                    }catch(ArithmeticException e)
                     {
                           System.out.println("Division by Zero');
                     }
          }catch(ArrayIndexOutofBoundsException
                   e){ System.out.println("ArrayIndexOutofBoundsException ');
            }
          }
 }
```

Multiple catch Statements

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following

```
try{
```

```
        //protected code
 } catch(ExceptionType1
              e1){
        // catch block
} catch(ExceptionType2
             e2){
        // catch block
}
```

Example: // To demonstrate multiple catch statements

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
      try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
          { System.out.println("task1 is
          completed");
          }
           catch(ArrayIndexOutOfBoundsException e)
          {System.out.println("task 2 completed");
          }
           catch(Exception e)
          { System.out.println("common task
          completed");
          }
   System.out.println("rest of the code...");
  }
```

The finally block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of the catch blocks and has the following syntax.

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Example: // To demonstrate the finally block

```
public class ExcepTest {
```

```
public static void main(String args[]) { int a[] = new int[2];
try {
System.out.println("Access element three :" + a[3]);
} catch (ArrayIndexOutOfBoundsException e) { System.out.println("Exception thrown :" + e);
}finally { a[0] = 6;
System.out.println("First element value: " + a[0]); System.out.println("The finally statement is executed");
}
}
}
```

Output

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Example: // Program to demonstrate use of throws clause

```
class Example{
    static void divide( ) throws ArithmeticException
    {
        int x=22;
        int y=0, z ;
        z=x/y;
    }
    public static void main(String args[])
    {
      try{   divide( ); catch(ArithmeticException ae)
             {
          }  System.out.println("Caught Exception " +ae);
             }


}
}
```

throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class.

// Java program that demonstrates the use of throw

```
class ThrowExcep
{
   static void fun()
   {
      try
      {
         throw new NullPointerException("demo");
      }
      catch(NullPointerException e)
      {
         System.out.println("Caught inside fun().");
         throw e; // rethrowing the exception
      }
   }
    public static void main(String args[])
   {
      try
      {
         fun();
      }
      catch(NullPointerException e)
      {
         System.out.println("Caught in main.");
      }
   }
}
```

User Defined Exceptions

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions. This can be done by extending the class Exception.

For example MyException in below code extends the Exception class. We pass the string to the constructor of the super class- Exception which is obtained using “getMessage()” function on the object created. In the below code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception’s constructor using super(). The constructor of Exception class can also be called without a parameter.

```
 // A Class that represents use-defined exception

class MyException extends Exception
{
   public MyException(String s)
   {
       super(s);  // Call constructor of parent Exception
   }
}
  // A Class that uses above MyException
public class Main
{
    public static void main(String args[])
   {
      try
       {
           throw new MyException("GeeksGeeks"); // Throw an object of user defined exception
       }
       catch (MyException ex)
       {
          System.out.println("Caught");     // Print the message from MyException object
          System.out.println(ex.getMessage());
       }
   }
}
```

Output

Caught
GeeksGeeks

Methods defined by Throwable class

Exception and all of it's subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable.

1. **String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.
2. **String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.
3. **synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
4. **String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
5. **void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

Java's Checked Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.