UNIT-3

Prepared By:
Deepak Kumar Sharma
Asst. Professor
UPES DEHRADUN

UNIT-III

Creating exception types, throw, throws Try, catch and finally, Custom exception, when to use exception
Wrapper classes, Loading classes
String operations, String comparisons, utility methods
Making related strings, string conversions, Strings and char arrays, string and byte arrays
StringBuffer, StringBuilder

By: Deepak Sharma, Asst. Professor, UPES Dehradun

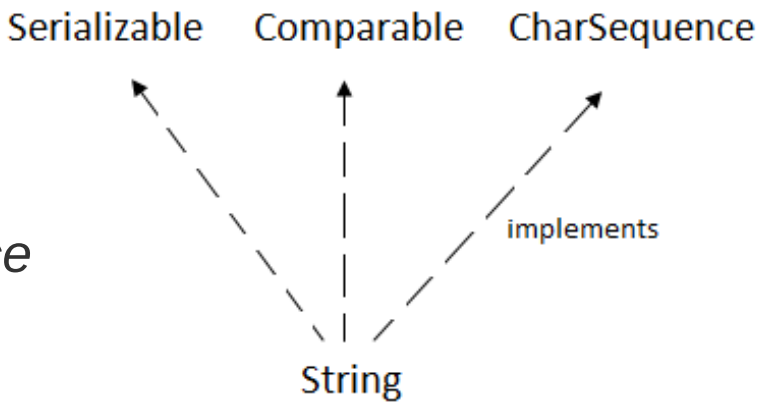
String Handling

String

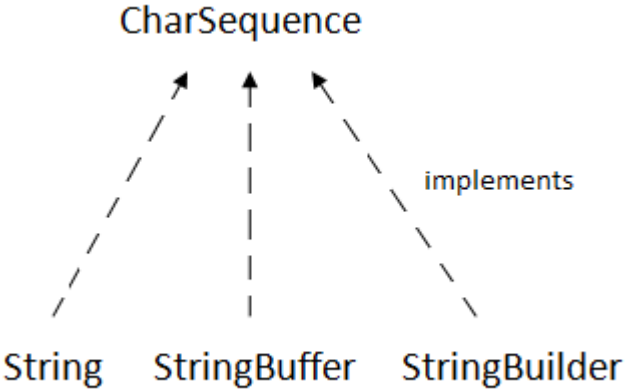
- String is basically an object that represents sequence of char values.
- An array of characters works same as Java string
- Once an object is created, no modifications can be done on that.

String

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



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



String

- In java a string is a sequence of characters. They are objects of type String.
- Once a String object has been created, we can not change the characters that comprise in the string.
- Strings are unchangeable once they are created so they are called as immutable.
- You can still perform all types of string operations. But, a new **String** object is created that contains the modifications. The original string is left unchanged.
- To get changeable strings use the class called StringBuffer.
- String and StringBuffer classes are declared final, so there cannot be subclasses of these classes.
- String class is defined in java.lang package, so these are available to all programmers automatically.

How to create a string object?

There are two ways to create String object:

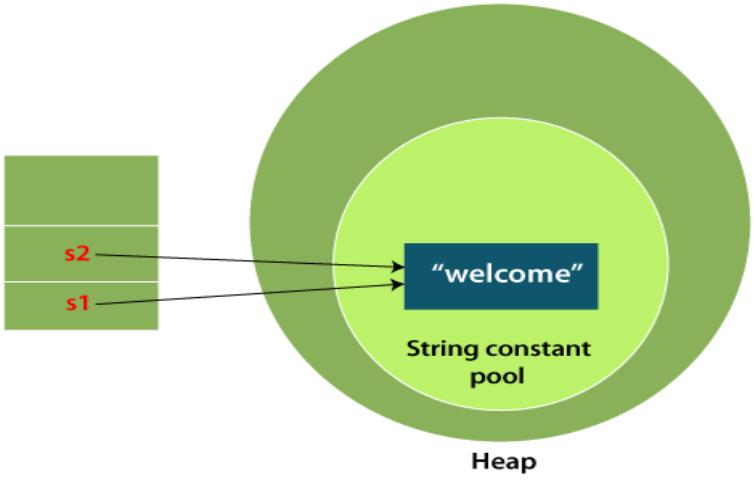
1.By string literal

2.By new keyword

```
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.

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



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

Why Java uses the concept of String literal?

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

How to create a string object?

2) By new keyword

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

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

```
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
```

String Constructors

- String s= new String();
- String(char chars[])
- String(char chars[], int startIndex, int numChars)
- String(String strobj)
- String str[]=new String[size];

String Constructors

- String s= new String(); //To create an empty String call the default constructor.

- String(char chars[]) //To create a string initialized by an array of characters.

  String str = "abcd"; is **equivalent** to

  char chars[]={‘a’,’b’,’c’,’d’};

  String s=new String(chars);

- String(char chars[], int startIndex, int numChars) //To create a string by specifying positions from an array of characters

  char chars[]={‘a’,’b’,’c’,’d’,’e’,’f’};

  String s=new String(chars,2,3); //This initializes **s** with characters “cde”.

- String(String str); //Construct a string object by passing another string object.

  String str = "abcd";

  String str2 = new String(str);

- String(byte asciiChars[ ]) // Construct a string from subset of byte array.
- String(byte asciiChars[ ], int startIndex, int numChars)

```
// 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);
   }
}
```

Output:

```
Java
Java
```



```
// 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);
   }
}
```

Output:

```
ABCDEF
CDE
```

String Length

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());        //it returns 3.
```

- **Special String Operations**
  - String Concatenation
  - String Concatenation with Other Data Types
  - String Conversion and toString( )
- **Character Extraction**
  - charAt( )
  - getChars( )
  - getBytes( )
  - toCharArray( )
- **String Comparison**
  - equals( ) and equalsIgnoreCase( )
  - regionMatches( ), startsWith( ) and endsWith( )
  - equals( ) Versus ==
  - compareTo( )
- **Searching Strings**
- **Modifying a String**
  - substring( )
  - concat( )
  - replace( )
  - trim( )
- **Changing the Case of Characters Within a String**

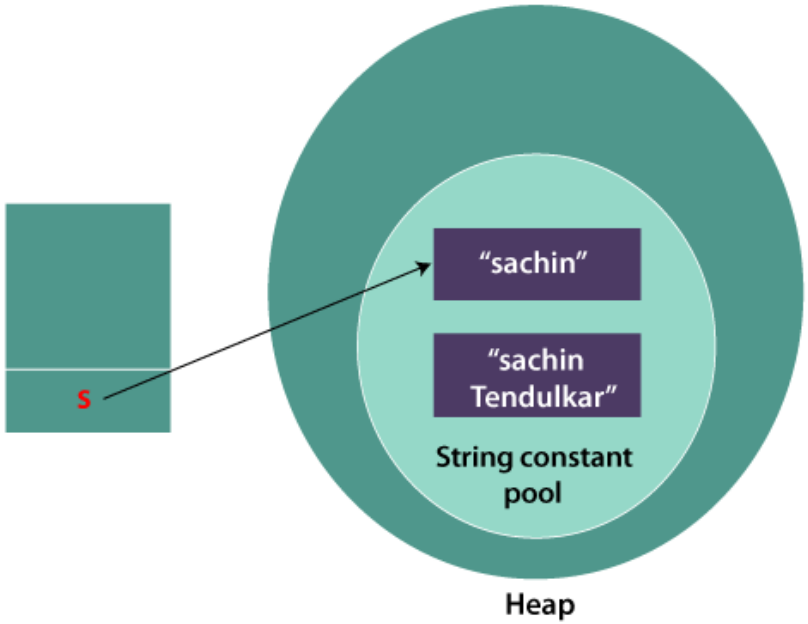
Strings are immutable

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

 }
}
```

Output:

Sachin



Special String Operations

String Concatenation

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);                    // He is 9 years old.
```

String Concatenation with Other Data Types

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);                    // He is 9 years old.


String s = "four: " + 2 + 2;
System.out.println(s);
        //This fragment displays four: 22  rather than the four: 4


String s = "four: " + (2 + 2);
        //Now s contains the string “four: 4”.
```

Special String Operations

Box's toString() method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

String Conversion using toString( )

```
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
}
    public String toString() {
    return "Dimensions are " + width + " by "
    +depth + " by " + height + ".";
    }
}
```

```
class toStringDemo {
public static void main(String args[]) {
            Box b = new Box(10, 12, 14);
            String s = "Box b: " + b;
            // concatenate Box object
            System.out.println(b);
            // convert Box to string
            System.out.println(s);
            }
}
```

The output of this program is shown here:

Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

char **charAt**(int *where*)

//To extract a single character from a **String**

```
char ch;
ch = "abc".charAt(1);
```

//assigns the value "**b**" to **ch**.

void **getChars**(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

//to extract more than one character at a time.

byte[ ] **getBytes**( )

//to store the characters in an array of bytes and it uses the default character-to-byte conversions provided by the platform.

//String source="hi"+"how"+"are"+"you."

//byte buf[] = source.getBytes();

char[ ] **toCharArray**( )

//to convert all the characters in a String object into a character array. It returns an array of characters for the entire string.

```
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

```
class GetBytesDemo{
    public static void main(String[] args){
        String str = "abc“ + ”ABC”;
            byte[] b = str.getBytes();
            //char[] c=str.toCharArray();
            System.out.println(str);
            for(int i=0;i<b.length;i++){
                System.out.print(b[i]+"  ");
                //System.out.print(c[i]+"  ");
            }
    }
}
```

Output:

97	**98**	**99**	**65**	**66**	**67**
//a	b	c	A	B	C

String Comparison

There are three ways to compare String in Java:

- **authentication** (by equals() method),
- **sorting** (by compareTo() method),
- **reference matching** (by == operator) etc.

String Comparison

- boolean **equals**(Object str)

  //To compare two strings for equality. It returns true if the strings contain the same characters in the same order, and false otherwise.

- boolean **equalsIgnoreCase**(String str)

  //To perform a comparison that ignores case differences.

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
        public static void main(String args[]) {
                String s1 = "Hello";
                String s2 = "Hello";
                String s3 = "Good-bye";
                String s4 = "HELLO";
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
```

```
System.out.println(s1.equals(s4));

System.out.println(s1.equalsIgnoreCase(s4));
        }
}
```

The output from the program is shown here:

```
true
false
false
true
```

String Comparison

boolean **regionMatches**(int startIndex, String str2, int str2StartIndex, int numChars)

boolean **regionMatches**(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

//The regionMatches( ) method compares a specific region inside a string with another specific region in another string.

//startIndex specifies the index at which the region begins within the invoking String object.

//The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex.

//The length of the substring being compared is passed in numChars.

String Comparison

```
class RegionTest{
        public static void main(String args[]){
                String str1 = "This is Test";
                String str2 = "THIS IS TEST";
                if(str1.regionMatches(5,str2,5,3)) {
                        // Case, pos1,secdString,pos1,len
                        System.out.println("Strings are Equal");
                }
                else{
                        System.out.println("Strings are NOT Equal");
                }
        }
}
```

Output:

Strings are NOT Equal

String Comparison

boolean **startsWith**(String str) //to determine whether a given String begins with a specified string.

boolean **endsWith**(String str) // to determine whether the String in question ends with a specified string.

Ex: "Football".endsWith("ball") and "Football".startsWith("Foot") are both **true**.

boolean startsWith(String str, int startIndex) // specifies the index into the invoking string at which point the search will begin.

Ex: "Football".startsWith("ball", 4) returns **true**.

equals( ) Versus ==

// It compares the characters inside a String object

//To compare two object references to see whether they refer to the same instance.

String Comparison

```
// equals() vs ==
class EqualsNotEqualTo {
   public static void main(String args[]) {
         String s1 = "Hello";
         String s2 = new String(s1);
         //String s2 = s1;
         System.out.println(s1.equals(s2));
         System.out.println( s1 == s2);
   }
}
```

Output:

true
false

String Comparison

int **compareTo**(String str)

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.

int compareToIgnoreCase(String str)

String Comparison

```
class SortString {
    static String arr[] = {
            "Now", "is", "the", "time", "for", "all", "good", "men",
            "to", "come", "to", "the", "aid", "of", "their", "country“};
    public static void main(String args[]) {
            for(int j = 0; j < arr.length; j++) {
                    for(int i = j + 1; i < arr.length; i++) {
                            if(arr[i].compareTo(arr[j]) < 0) {
                                    String t = arr[j];
                                    arr[j] = arr[i];
                                    arr[i] = t;
                            }
                    }
                    System.out.println(arr[j]);
            }
    }
}
```

The output of this program is the list of words:

Now aid all come country for good is men of the the their time to to

Searching Strings

indexOf( ) //Searches for the first occurrence of a character or substring.

lastIndexOf( ) //Searches for the last occurrence of a character or substring.

int indexOf(int ch) //To search for the first occurrence of a character.

int lastIndexOf(int ch) //To search for the last occurrence of a character.

int indexOf(String str) //To search for the first or last occurrence of a substring.

int lastIndexOf(String str)

int indexOf(int ch, int startIndex)

int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)

int lastIndexOf(String str, int startIndex)

In all cases, the methods return the **index** at which the character or substring was found, or **–1** on failure.

```
// 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));
}
}
```

Here is the output of this program:

```
Now is the time for all good men
to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

String **substring**(int startIndex)

//To extract a substring, startIndex specifies the beginning index

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

// Here endIndex specifies the stopping point.

//endIndex-1

```
// Substring replacement.
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String result = org.substring(0, 2);
        System.out.println(result);
    }
}
```

Output:

Th

String **concat**(String str)

//To concatenate two strings, **concat( ) performs the same function as +.**

```
String s1 = "one";
String s2 = s1.concat("two");
String s1 = "one";
String s2 = s1 + "two";
```

String **replace**(char original, char replacement) //To replace all occurrences of one character in the invoking string with another character

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into s.

String **trim( )** //Removes leading and trailing whitespaces.

Here is an example:

```
String s = "   Hello World   ".trim();
//This puts the string “Hello World” into s.
```

Data Conversion Using valueOf( )

static String valueOf(double num)

// converts data from its internal format into a human-readable form.

static String valueOf(long num)

//each type can be converted properly into a string

static String valueOf(Object ob)

static String valueOf(char chars[ ])

Changing the Case of Characters Within a String

String toLowerCase( )

//converts all the characters in a string from uppercase to lowercase.

String toUpperCase( )

//converts all the characters in a string from lowercase to uppercase

```
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
    public static void main(String args[]){
        String s = "This is a test.";
        System.out.println("Original: " + s);
        String upper = s.toUpperCase();
        String lower = s.toLowerCase();
        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}
```

Output:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

StringBuffer

- A StringBuffer is like a String, but can be modified.
- StringBuffer represents growable and writeable character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

StringBuffer Constructors

- **StringBuffer( )**

// The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.

- **StringBuffer(int size)**

// accepts an integer argument that explicitly sets the size of the buffer.

- **StringBuffer(String str)**

// str + reserves room for 16 more characters without reallocation.

Methods

- length( ) and capacity( )
- setLength( )
- charAt( ) and setCharAt( )
- getChars( )
- append( )
- insert( )
- reverse( )
- delete( ) and deleteCharAt( )
- replace( )
- substring( )

int **length( )** //The current length of a StringBuffer can be found via the length( ) method
int **capacity( )** //The total allocated capacity can be found through the capacity( ) method.

```
// 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());
   }
}
```

Output:

```
buffer = Hello
length = 5
capacity = 21
```

void **setLength**(int len) //To set the length of the buffer within a StringBuffer object.

char **charAt**(int where)

//The value of a single character can be obtained from a StringBuffer via the charAt( ) method.

void **setCharAt**(int where, char ch)

//To set the value of a character, ch specifies the new value of that character.

```
// 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));
    }
}
```

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

- void **getChars**(int sourceStart, int sourceEnd, char target[ ], int targetStart)

  // To copy a substring of a StringBuffer into an array

- StringBuffer **append**(String str)

  // concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

- StringBuffer **append**(int num)

- StringBuffer **append**(Object obj)

- **String.valueOf( ) is called for each parameter to obtain its string representation. The result is appended to the current StringBuffer object.**

```
// 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);
   }
}
```

Output:

```
a = 42!
```

//The insert( ) method inserts one string into another.

StringBuffer **insert**(int index, String str)

StringBuffer **insert**(int index, char ch)

StringBuffer **insert**(int index, Object obj)

```
// Demonstrate insert().
class insertDemo {
   public static void main(String args[]) {
         StringBuffer sb = new StringBuffer("I Java!");
         sb.insert(2, "like ");
         System.out.println(sb);
   }
}
```

Output:

I like Java!

StringBuffer **reverse( )**

// reverse the characters within a StringBuffer object using 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);
   }
}
```

Output:

```
abcdef
fedcba
```

StringBuffer **delete**(int startIndex, int endIndex) // to delete characters.

StringBuffer **deleteCharAt**(int loc)

```
// 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);
   }
}
```

Output:

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

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

//It replaces one set of characters with another set inside a StringBuffer object.

// The replacement string is passed in str.

// 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);
    }
}
```

Output: After replace: This was a test.

String **substring**(int startIndex) //to return sub strings.

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

StringBuilder

- Java StringBuilder class is used to create mutable (modifiable) String.
- The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.
- It is available since JDK 1.5.

StringBuilder Constructors

StringBuilder() Constructs a string builder with no characters in it and an initial capacity of 16 characters.
StringBuilder(**CharSequence** seq) Constructs a string builder that contains the same characters as the specified CharSequence.
StringBuilder(int capacity) Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
StringBuilder(**String** str) Constructs a string builder initialized to the contents of the specified string.

Methods

- length( ) and capacity( )
- setLength( )
- charAt( ) and setCharAt( )
- getChars( )
- append( )
- insert( )
- reverse( )
- delete( ) and deleteCharAt( )
- replace( )
- substring( )

StringBuilder- append method

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

Methods are similar to StringBuffer

For complete list of methods:

https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

String vs StringBuffer

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

Performance check

```
public class ConcatTest{
   public static String concatWithString()    {
      String t = "Java";
      for (int i=0; i<10000; i++){
         t = t + “Example";
      }
      return t;
   }
   public static String concatWithStringBuffer(){
      StringBuffer sb = new StringBuffer("Java");
      for (int i=0; i<10000; i++){
         sb.append(“Example");
      }
      return sb.toString();
   }
   public static void main(String[] args){
      long startTime = System.currentTimeMillis();
      concatWithString();
      System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-startTime)+"ms");
      startTime = System.currentTimeMillis();
      concatWithStringBuffer();
            System.out.println("Time taken by Concating with StringBuffer: "+(System.currentTimeMillis()-
startTime)+"ms");
   }
}
```

Output:
Time taken by Concating with String: 356ms
Time taken by Concating with StringBuffer: 2ms

StringBuffer vs StringBuilder

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

Performance check

```
//Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
public class ConcatTest{
   public static void main(String[] args){

long startTime = System.currentTimeMillis();
        StringBuffer sb = new StringBuffer("Java");
        for (int i=0; i<100000; i++){
            sb.append("Programming");
        }
      System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() -
 startTime) + "ms");

startTime = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder("Java");
        for (int i=0; i<100000; i++){
            sb2.append("Programming");
        }
              System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() -
 startTime) + "ms");
   }
}
```

Output:
Time taken by StringBuffer: 79ms
Time taken by StringBuilder: 5ms

StringTokenizer

- The **StringTokenizer** class provides the first step in parsing process, often called the lexer (lexical analyzer) or scanner.

- Parsing is the division of text into a set of discrete parts, or tokens.

To use StringTokenizer

- Specify an input string and a delimiter string.

- Delimiters are characters that separate tokens.

- Each character in the delimiters string is considered a valid delimiter.

  Ex: **“ , ; : a”**

- The default set of delimiters consists of the whitespace characters:

  Ex: space, tab, newline.

StringTokenizer Constructors

- StringTokenizer(String *str*)
- StringTokenizer(String *str*, String *delimiters*)
- StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

Methods

- int **countTokens( )** //returns number of tokens in the string.
- boolean **hasMoreTokens( )** //checks whether tokens are there or not
- String **nextToken( )** //returns the token in the string

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;
class STDemo{
   //static String str = "Hello Welcome to Java Programming";
   static String str = "Hello,Welcome,to,Java,Programming";
   public static void main(String args[]) {
         //StringTokenizer st = new StringTokenizer(str);
         //StringTokenizer st = new StringTokenizer(str,",");
         StringTokenizer st = new StringTokenizer(str,",",true);
         while(st.hasMoreTokens()) {
               String tokens = st.nextToken();
               System.out.println(tokens + "\n");
         }
   }
}
```

Output:
Hello
,
Welcome
,
to
,
Java
,
Programming

Exercise1:

Write a java program that reads a line of integers and then displays each integer and find the sum of the integers (using StringTokenizer)

```
import java.util.Scanner;
import java.util.*;
class Token{
        static int sum=0;
        public static void main(String sree[]){
                Scanner s=new Scanner(System.in);
                System.out.print("Enter sum of integers: ");
                String str=s.next();
                StringTokenizer st=new StringTokenizer(str,"+");
                while(st.hasMoreTokens()){
                        sum=sum+Integer.parseInt(st.nextToken());
                }
                System.out.println("Sum of "+str+“is: "+sum);
        }
}
```

Output:
Enter sum of integers: 10+20+30+40
Sum of 10+20+30+40 is: 100

Exception Handling in Java

exception

- an exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.

Exception Handling:

- mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Following are some scenarios where an exception occurs:

• 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.

Advantages of Exception Handling

- to maintain the normal flow of the application.

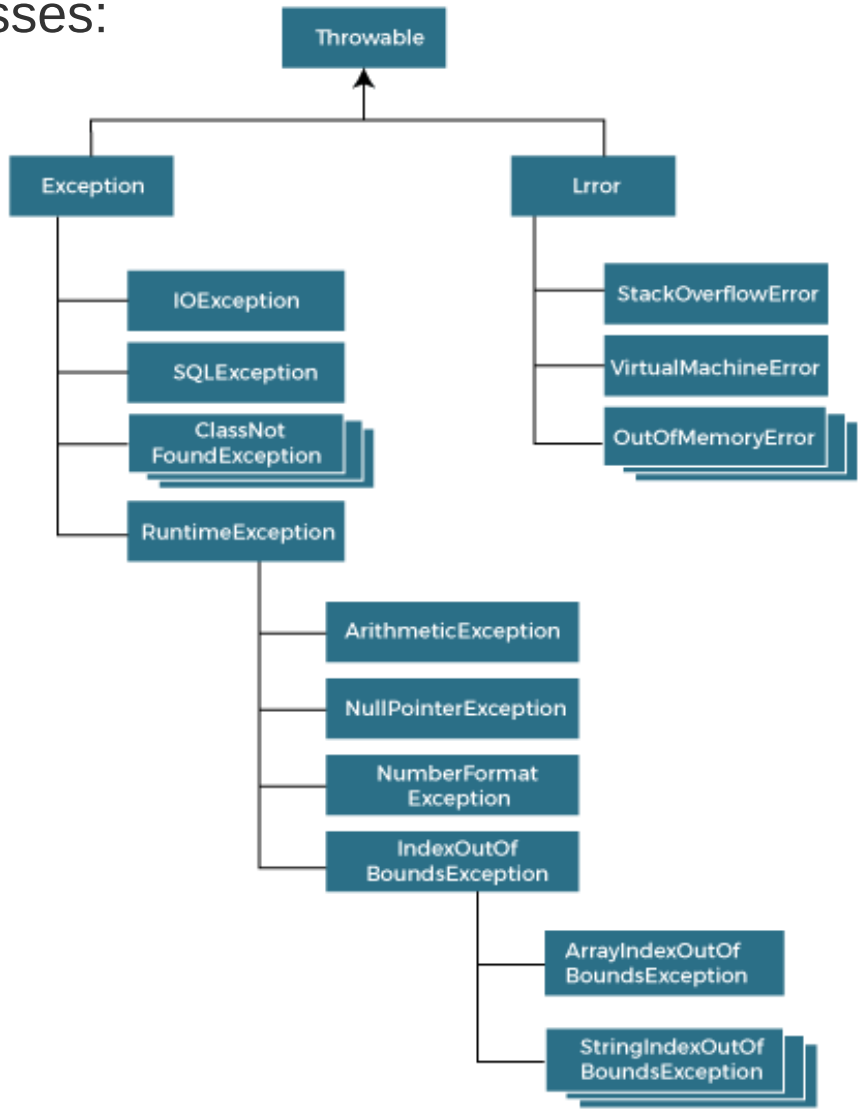
```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- If an exception occurs at statement 5; the rest of the code will not be executed.
- exception handling in Java will ensure that remaining statements (6 to 10) are executed.

Hierarchy of Java Exception classes

- The **java.lang.Throwable** class is the root class of Java Exception hierarchy inherited by two subclasses:
  - Exception and Error

hierarchy of Java Exception classes



Types of Java Exceptions

1) Checked Exception

- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.
- For example, IOException, SQLException, etc.
- Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes that inherit the RuntimeException are known as unchecked exceptions.
- For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

- Error is irrecoverable.
- Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

```
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...

Common Scenarios of Java Exceptions

1) A scenario where **ArithmeticException** occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where **NullPointerException** occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where **NumberFormatException** occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where **ArrayIndexOutOfBoundsException** occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

- Java **try** block is used to enclose the code that might throw an exception.
- It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., Exception) or the generated exception type.
- However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only.
- You can use multiple catch block with a single try block.

The JVM firstly checks whether the exception is handled or not.

If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

Problem without exception handling

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception


        System.out.println("rest of the code");


    }

}
```

- all the code below the exception won't be executed.

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

Solution by exception handling

```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example

- the code in a try block that will not throw an exception

```
public class TryCatchExample3 {

  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
                   // if exception occurs, the remaining statement will not exceute
    System.out.println("rest of the code");
    }
      // handling the exception
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }

  }
}
```

Output:

java.lang.ArithmeticException: / by zero

Example

- Handling the exception using the parent class exception.

```
public class TryCatchExample4 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example

- to print a custom message on exception.

```
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
                // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }

}
```

Output:

Can't divided by zero

Example

Let's see an example to resolve the exception in a catch block.

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

25

Example

In this example, along with try block, we also enclose exception code in a catch block.

```
public class TryCatchExample7 {

    public static void main(String[] args) {

        try
        {
        int data1=50/0; //may throw exception

        }
            // handling the exception
        catch(Exception e)
        {
            // generating the exception in catch block
        int data2=50/0; //may throw exception

        }
    System.out.println("rest of the code");
    }
}
```

Output:

Exception in thread "main"
java.lang.ArithmeticException: / by zero

enclose exception code within a try block and use catch block only to handle the exceptions.

Java Catch Multiple Exceptions

Points to Note:

•At a time only one exception occurs and at a time only one catch block is executed.

•All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)  {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)  {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)  {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Output:

Arithmetic Exception occurs
rest of the code

Example

```
public class MultipleCatchBlock2 {
   public static void main(String[] args) {
        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Output:

ArrayIndexOutOfBounds Exception occurs
rest of the code

```
public class MultipleCatchBlock3 {
   public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Predict output?

Output:

Arithmetic Exception occurs
rest of the code

Example

- Generated NullPointerException, but didn't provided the corresponding exception type.
- In such case, the catch block containing the parent exception class **Exception** will invoked.

```
public class MultipleCatchBlock4 {

  public static void main(String[] args) {
      try{
          String s=null;
          System.out.println(s.length());
        }
        catch(ArithmeticException e) {
          System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e) {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
        }
        catch(Exception e) {
          System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
  }
}
```

Output:

Parent Exception occurs
rest of the code

Example

- Handling the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5{
 public static void main(String args[]){
  try{
   int a[]=new int[5];
   a[5]=30/0;
  }
  catch(Exception e){System.out.println("common task completed");}
  catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 complet
ed");}
  System.out.println("rest of the code...");
 }
}
```

Output:

Predict output?

Compile-time error

Java Nested try block

- A try block inside another try block -> nested try block

Scenario :

- the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException**
- while the **outer try block** can handle the **ArithemeticException**

```java
public class NestedTryBlock{
 public static void main(String args[]){
//outer try block
 try{
 //inner try block 1
   try{
    System.out.println("going to divide by 0");
    int b =39/0;
  }
   //catch block of inner try block 1
   catch(ArithmeticException e)  { System.out.println(e);  }
   //inner try block 2
   try{
   int a[]=new int[5];
    a[5]=4;
    }
   //catch block of inner try block 2
   catch(ArrayIndexOutOfBoundsException e)  {  System.out.println(e);  }
   System.out.println("other statement");
 }
 //catch block of outer try block
 catch(Exception e)  {  System.out.println("handled the exception (outer catch)"); }
 System.out.println("normal flow..");
}    }
```

```
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not.
- The finally block follows the try-catch block.

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Case 1: When an exception does not occur

```
1.class TestFinallyBlock {
2.  public static void main(String args[]){
3.  try{
4.//below code do not throw any exception
5.   int data=25/5;
6.   System.out.println(data);
7.  }
8.//catch won't be executed
9.  catch(NullPointerException e){
10.System.out.println(e);
11.}
12.//executed regardless of exception occurred or not
13. finally {
14.System.out.println("finally block is always executed");
15.}
16.
17.System.out.println("rest of phe code...");
18.  }
19.}
```

```
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

```
public class TestFinallyBlock1{
    public static void main(String args[]){
    try {
      System.out.println("Inside the try block");
      //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
    //can only accept Null Pointer type exception
    catch(NullPointerException e){
      System.out.println(e);
    }
    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
  }
```

```
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

the finally block is executed after the try block and then the program terminates abnormally.

Case 3: When an exception occurs and is handled by the catch block

```
public class TestFinallyBlock2{
    public static void main(String args[]){

    try {

      System.out.println("Inside try block");

      //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
      System.out.println("Exception handled");
      System.out.println(e);
    }
    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }
    System.out.println("rest of the code...");
    }
  }
```

the finally block is executed after the try-catch block.

```
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

- Rule: For each try block there can be zero or more catch blocks, but only one finally block.
- Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has **some message** with it that provides the error description.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.
- We can also define our own set of conditions and throw an exception explicitly using throw keyword.
  - For example, we can throw ArithmeticException if we divide a number by another number.

The syntax of the Java throw Instance

```
throw new exception_class("error message");
throw new IOException("sorry device error");
```

the Instance must be of type Throwable or subclass of Throwable.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");

      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
 }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

Example 2: Throwing Checked Exception

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

```
import java.io.*;
public class TestThrow2 {
   //function to check if person is eligible to vote or not
   public static void method() throws FileNotFoundException {
      FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
      BufferedReader fileInput = new BufferedReader(file);
      throw new FileNotFoundException();
   }
   //main method
   public static void main(String args[]){
      try
      {
         method();
      }
      catch (FileNotFoundException e)
      {
         e.printStackTrace();
      }
      System.out.println("rest of the code...");
  }
}
```

```
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java.
A checked exception is everything else under the Throwable class.

Example 3: Throwing User-defined Exception

```
// class represents user-defined exception
class UserDefinedException extends Exception
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    } }
// Class that uses above MyException
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        } } }
```

```
Caught the exception
This is user-defined exception
```

Java Exception Propagation

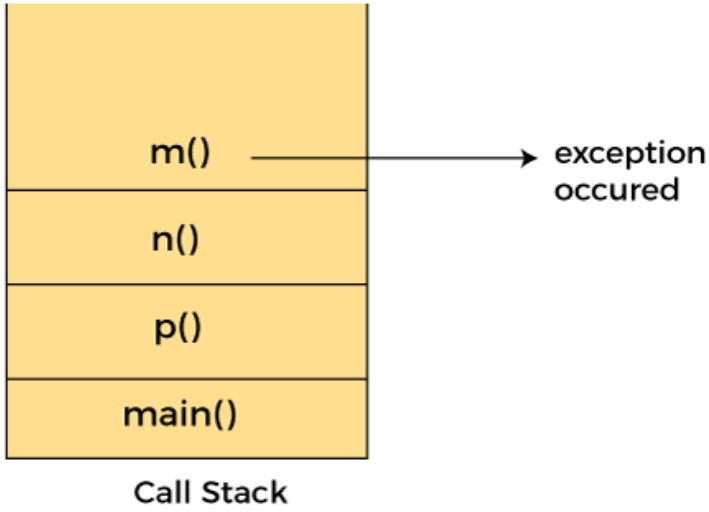
- An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method.
- If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.
- This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Exception Propagation Example

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```



Output:

```
exception handled
normal flow...
```

Exception Propagation Example

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handeled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation2 obj=new TestExceptionPropagation2();
   obj.p();
   System.out.println("normal flow");
  }
}
```

Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Output:

Compile Time Error

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

• **unchecked exception:** under our control so we can correct our code.

•**error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Java throws Example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

Output:

exception handled
normal flow...

Cont..

There are two cases:

- **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
- **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

- In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
      normal flow...
```

Case 2: Declare Exception

•In case we declare the exception, if exception does not occur, the code will be executed fine.
•In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:

device operation performed
normal flow...

Cont..

B) If exception occurs

Testthrows4.java

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

Difference between throw and throws in Java

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

```
public class TestThrow {
   //defining a method
   public static void checkNum(int num) {
      if (num < 1) {
         throw new ArithmeticException("\nNumber is negative, cannot calculate square");
      }
      else {
         System.out.println("Square of " + num + " is " + (num*num));
      }
   }
   //main method
   public static void main(String[] args) {
         TestThrow obj = new TestThrow();
         obj.checkNum(-3);
         System.out.println("Rest of the code..");
   }
}
```

```
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

        System.out.println("Rest of the code..");
    }
}
```

```
Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

```
public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmeticException
    static void method() throws ArithmeticException
    {
        System.out.println("Inside the method()");
        throw new ArithmeticException("throwing ArithmeticException");
    }
    //main method
    public static void main(String args[])
    {
        try
        {
            method();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught in main() method");
        }
    }
}
```

```
Inside the method()
caught in main() method
```

Java Custom Exception

- we can create our own exceptions that are derived classes of the Exception class.
- Java custom exceptions are used to customize the exception according to user need.
- Using the custom exception, we can have your own exception and message.

Why use custom exceptions?

- Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.
- In order to create custom exception, we need to **extend Exception** class that belongs to java.lang package.

Example: Custom Exception

```
public class WrongFileNameException extends Exception {
    public WrongFileNameException(String errorMessage) {
    super(errorMessage);
    }
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example: Custom Exception

```
class InvalidAgeException  extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}
// class that uses custom exception InvalidAgeException
public class TestCustomException1
{
    // method to check the age
    static void validate (int age) throws InvalidAgeException{
       if(age < 18){
        // throw an object of user defined exception
        throw new InvalidAgeException("age is not valid to vote");
    }
       else {
        System.out.println("welcome to vote");
        }
    }
```

……Cont. on next Page

```java
// main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");
            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }
        System.out.println("rest of the code...");
    }
}
```

```
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example: Custom Exception

```
// class representing custom exception
class MyCustomException extends Exception
{

}
// class that uses custom exception MyCustomException
public class TestCustomException2
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new MyCustomException();
        }
        catch (MyCustomException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
        System.out.println("rest of the code...");
    }
}
```

```
Caught the exception
null
rest of the code...
```

Understanding throw, throws and custom exception step wise

By: Deepak Sharma, Asst. Professor, UPES Dehradun

Example: throw, throws and custom exception

```
public class Throws_example
{
   public static double divide(int n,int d) //Made by Team2
{
            double res=n/d;
            return res;
}

public static void main(String[] args)
   {
// To be called by Team1
    double result=divide(20,0);
    System.out.println("Result: " + result);
   }
}
```

Without any exception Handling


```
PS D:\java_prog> java Throws_example
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Throws_example.divide(Throws_example.java:5)
        at Throws_example.main(Throws_example.java:11)
```

Example: throw, throws and custom exception

```
public class Throws_example
{
  public static double divide(int n,int d) throws ArithmeticException
{
        double res=n/d;
        return res;
}
        public static void main(String[] args)
  {
     double result=divide(20,0);
     System.out.println("Result: " + result);
  }
}
```

- Team 2 telling everyone that function may raise the given exception.
  Team 1 use try catch for this exception or handle it properly.

```
PS D:\java_prog> java Throws_example
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Throws_example.divide(Throws_example.java:5)
        at Throws_example.main(Throws_example.java:11)
```

Example: throw, throws and custom exception

```
public class Throws_example
{
   public static double divide(int n,int d) throws ArithmeticException
{
            double res=n/d;
            return res;
}
            public static void main(String[] args)
    {
                        try{
                        double result=divide(20,0);
        }catch(Exception e)
                        {
                        System.out.println("Denominator cant be negative");
                        }
                        System.out.println("Result: " + result);

    }
}
```

Exception Handled

```
PS D:\java_prog> java Throws_example
Denominator cant be negative
```

Example: throw, throws and custom exception

```
class neg_radiusexception extends Exception{

public String toString()
{
        return "radius can't be negative";
}
public String getMessage()
{
        return "radius can't be negative";
}

}
```

Cont..

```
public class Throws_example{
        public static double area_circle(int r) throws neg_radiusexception{
                if(r<0){
                        throw new neg_radiusexception(); }
                double res=3.14*r*r;
                return res;
        }
  public static double divide(int n,int d) throws ArithmeticException
{
        double res=n/d;
        return res;
}
        public static void main(String[] args)
  {
                try{
                double result=divide(20,0);
    System.out.println("Result: " + result);
                }catch(Exception e){System.out.println("Denominator can't be negative");}
            try{
                double ar=area_circle(-10);
                System.out.println(ar);
      }catch(neg_radiusexception ne){ System.out.println(ne);}
      }
}
```

Output:
Denominator can't be negative
radius can't be negative

Wrapper classes in Java

- A Wrapper class is a class whose object wraps or contains primitive data types.
- When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.
- Wrapper class in java provides the mechanism to convert primitive data type into object is called **boxing** and object into primitive data type is called **unboxing**.
- Since J2SE 5.0, auto boxing and unboxing feature converts primitive data type into object and object into primitive data type automatically. The automatic conversion of primitive data type into object is known as **auto-boxing** and vice-versa **auto-unboxing**.

Wrapper Classes

- The eight classes of the *java.lang* package are known as wrapper classes in Java.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Example

```
// Wrapper class Example: Primitive to Wrapper

import java.lang.*;

public class WrapperExample1
{
    public static void main(String args[])
    {
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
        System.out.println(a+" "+i+" "+j);
    }
```

Example

```
// Wrapper class Example: Wrapper to Primitive

import java.lang.*;

public class WrapperExample2
{
    public static void main(String args[])
    {
        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue();//unboxing i.e converting Integer to int
        int j=a;//auto unboxing, now compiler will write a.intValue() internally

        System.out.println(a+" "+i+" "+j);
    }
}
```

Need of Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Autoboxing

- Automatic conversion of primitive types to the object of their corresponding wrapper classes
- For example – conversion of int to Integer, long to Long, double to Double etc.

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output: 20 20 20

Unboxing

- Automatic conversion of wrapper type into its corresponding primitive type
- For example – conversion of Integer to int, Long to long, Double to double…

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally


System.out.println(a+" "+i+" "+j);
}}
```

Output: 3 3 3

Methods Supported by the Wrapper Class

Method	Method Description
typeValue()	Converts the value of this Number object to the specified primitive data type returned
compareTo()	Compares this Number object to the argument
equals()	Determines whether this Number object is equal to the argument
valueOf()	Returns an Integer object holding the value of the specified primitive data type value
toString()	Returns a String object representing the value of specified Integer type argument
parseInt()	Returns an Integer type value of a specified String representation
decode()	Decodes a String into an integer
min()	Returns the smaller value after comparison of the two arguments
max()	Returns the larger value after comparison of the two arguments
round()	Returns the closest round off long or int value as per the method return type

Custom Wrapper Class in Java

- We can create custom wrapper class in Java which wraps a primitive data type.

```
class SpeedWrapperClass{
private int speed;
SpeedWrapperClass(){}
SpeedWrapperClass(int speed){
this.speed=speed;
}
public int getVehicleSpeed(){
return speed;
}
public void setVehicleSpeed(int speed){
this.speed=speed;
}
public String toString() {
  return Integer.toString(speed);
}
}
//Testing the custom wrapper class
public class TestJavaWrapperClass{
public static void main(String[] args){
SpeedWrapperClass speedValue =new SpeedWrapperClass(100);
System.out.println(speedValue);
}}
```

Points to Remember

- Wrapper classes in Java wraps the primitive data type in its class object.
- Java wrapper classes are provided in the java.lang package.
- Autoboxing and unboxing converts the primitive into objects and objects into primitives automatically.
- We can also create custom Wrapper classes which wraps a primitive data type.

Java ClassLoader

- Java ClassLoader is an abstract class.
- It belongs to a **java.lang** package.
- It loads classes from different resources.
- Java ClassLoader is used to load the classes at run time. In other words, JVM performs the linking process at runtime.
- Classes are loaded into the JVM according to need. If a loaded class depends on another class, that class is loaded as well.
- When we request to load a class, it delegates the class to its parent.

Java ClassLoader principles:

•**Delegation principle:** It forwards the request for class loading to parent class loader. It only loads the class if the parent does not find or load the class.

•**Visibility principle:** It allows child class loader to see all the classes loaded by parent ClassLoader. But the parent class loader cannot see classes loaded by the child class loader.

•**Uniqueness principle:** It allows to load a class once. It is achieved by delegation principle. It ensures that child ClassLoader doesn't reload the class, which is already loaded by the parent.

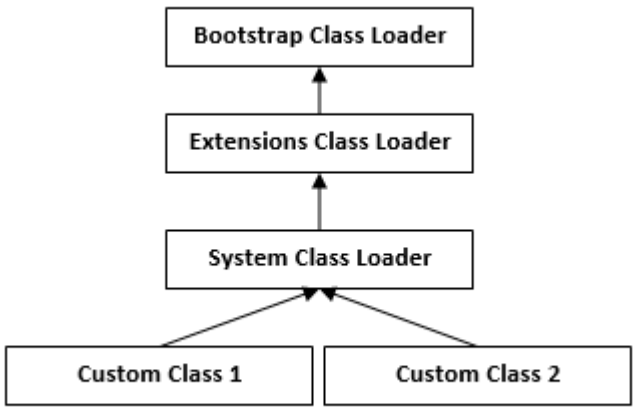


Figure: Hierarchy of Class Loader

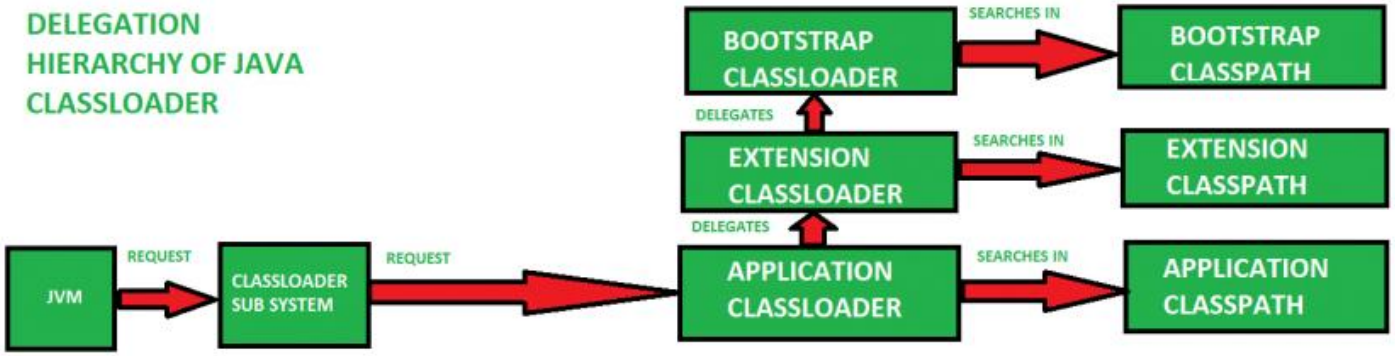
Types of ClassLoader

- **Bootstrap Class Loader:** It loads standard JDK class files from rt.jar and other core classes. It is a parent of all class loaders. It doesn't have any parent. When we call String.class.getClassLoader() it returns null, and any code based on it throws NullPointerException.

- **Extensions Class Loader:** It delegates class loading request to its parent. If the loading of a class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory as java.ext.dirs.

- **System Class Loader:** It loads application specific classes from the CLASSPATH environment variable. It can be set while invoking program using -cp or classpath command line options. It is a child of Extension ClassLoader.

How it Works



Class loader follows the following rule:

- It checks if the class is already loaded.
- If the class is not loaded, ask parent class loader to load the class.
- If parent class loader cannot load class, attempt to load it in this class loader.

```
public class Demo
{
public static void main(String args[])
{
System.out.println("How are you?");
}
}
```

Compile and run the above code by using the following command:

```
javac Demo.java
java -verbose:class Demo
```

-verbose:class: It is used to display the information about classes being loaded by JVM.

It is useful when using class loader for loading classes dynamically.