Java and J2EE

(17IS6DEJVA)

Unit 4

String Handling

Strings in Java

- Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

- The *java.lang.String* class is used to create string object.

- Syntax for declaring a string **:**

  <String_Type> <string_variable> = "<sequence_of_string>";

Example:

```
String Str = “DSCE”;
```

. .

String object Creation

- There are two ways to create a String object:

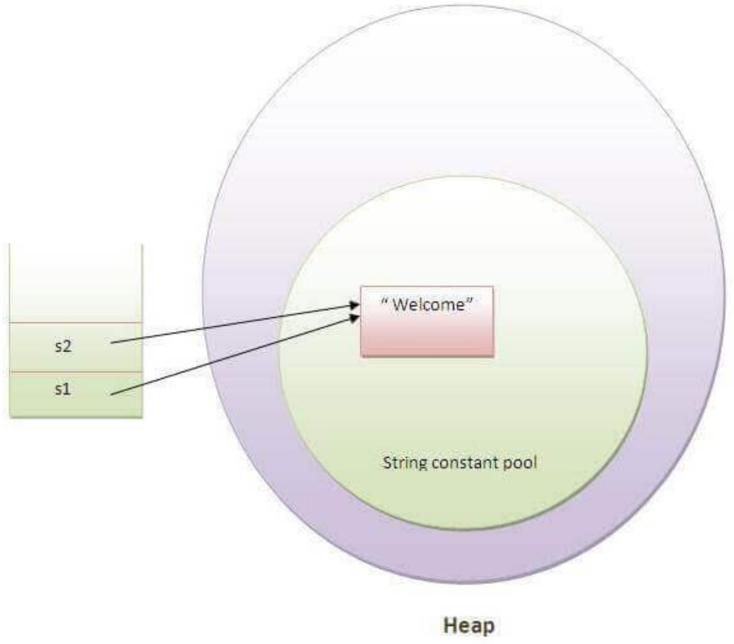
1) **By string literal** : Java String literal is created by using double quotes.

For Example: String s="Welcome";

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

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

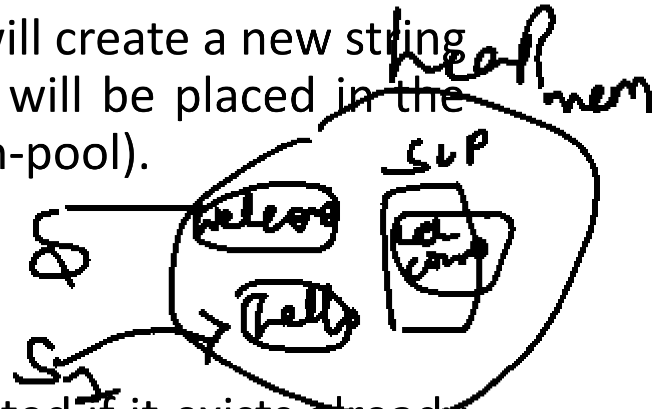
In the above example, only one object will be created. Firstly, JVM the reference to the same instance will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it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



String object Creation

2) **By new keyword** : Java String is created by using a keyword "new". For example: String s=new String("Welcome");

- Note: It creates 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).



- 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).

.

- The **String** class supports several constructors.
- The most commonly used constructors of String class are as follows:

1. **String():** To create an empty String, we will call a default constructor.

For example:

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

It will create a string object in the heap area with no value.

2. **String(String str):** construct a String object that contains the same character sequence as another String object .

For example:

```
// 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:

```
J
a
v
```

a
J
a
v
a

3. **String(char chars[ ]):** It will create a string object and stores the array of characters in it. For example:

```
        char chars[ ] = { ‘a’, ‘b’, ‘c’, ‘d’ };
        String s3 = new String(chars);
```

The object reference variable s3 contains the address of the value stored in the heap area.

4. **String(char chars[ ], int startIndex, int count):** It will create and initializes a string object with a subrange of a character array. The argument *startIndex* specifies the index at which the subrange begins and *count* specifies the number of characters to be copied.

For example:

```
        char chars[ ] = { ‘w’, ‘i’, ‘n’, ‘d’, ‘o’, ‘w’, ‘s’ };

        String str = new String(chars, 2, 3);
```

The object str contains the address of the value ”ndo” stored in the heap area because the starting index is 2 and the total number of characters to be copied is 3.

**** NOTE : The contents of the array are copied whenever you create a String object from an array. If you modify the contents of the array after you have created the string, the String will be unchanged.***

5. **String(byte byteArr[ ]):** It constructor a new string object by decoding the given array of bytes (i.e, by decoding ASCII values into the characters) according to the system's default character set.

6. **String(byte byteArr[ ], int startIndex, int count):** This constructor also creates a new string object

by decoding the ASCII values using the system's default character set.

```
// 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:

```
A
B
C
D
E
F
C
D
E
```

NOTE: 65 is the Unicode value of A,66 ➨ B, 67 ➨ C, 68 ➨ D…….

String Length

- **length()**: This method tells the length of the string. It returns count of total number of characters present in the String.

- For example:

```
public class Example{
        public static void main(String args[]{
                St
                ri
                ng
                s1
                ="
                he
                llo
                ";
                St
                ri
                ng
                s2
                ="
                w
                ha
                ts
                u
                p"
                ;
                System.out.println("string
                length is: "+s1.length());
```

String Length

```
System.out.println("string
length is: "+s2.length());

        }
    }
```

Output:

```
string length is: 5 string length is: 7
```

Special String Operations

- Java has added special support for several string operations within the syntax of the language. These operations include :

  - the automatic creation of new **String** instances from string literals,
  - concatenation of multiple **String** objects by use of the **+** operator, and
  - the conversion of other data types to a string representation.

- There are explicit methods available to perform all of these functions, but Java

  does them automatically as a convenience for the programmer and to add clarity.

Special String Operations : String Literals

- String Literals
  - A String instance can be created from an array of characters by using the new operator. However, there is an easier way to do this using a string literal.
  - For each string literal in the program, Java automatically constructs a **String** object.Thus, you can use a string literal to initialize a Stringobject.
  - Example:
    - char chars[] = { 'a', 'b', 'c' };
    - String s1 = new String(chars);
    - String s2 = "abc"; // use string literal

Special String Operations : String Literals

- Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object.
- For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows.
  - System.out.println("abc".length());

- It calls the **length( )** method on the string “abc”. As expected, it prints “3”.

Special String Operations : String Concatenation

- **String Concatenation**
  - Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result.
  - This allows you to chain together a series of + operations.
  - Example:
    - String age = "9";
    - String s = "He is " + age + " years old.";
    - System.out.println(s);

Output:

He is 9 years old.

  - One practical use of string concatenation is found when you are creating very long strings.
  - Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the **+** to concatenate them.

Special String Operations : String Concatenation

- Example:

```
class ConCat {
        public static void main(String args[]) {
        String longStr = "This could have been " +
        "a very long line that would have " +
        "wrapped around. But string concatenation " +
        "prevents this.";
        System.out.println(longStr);
    }
}
```

Special String Operations: String Concatenation with Other Data Types

- **String Concatenation with Other Data Types**

  - It is possible to concatenate strings with other types of data.

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

Output:

```
He is 9 years old.
```

  - This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of String.

  - Example:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

Output:four:22ra

therthanthefour:4

- Operator precedence causes the concatenation of “four” with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time.
- To complete the integer addition first, you must use parentheses, like this: String s = "four: " + (2 + 2);

  Now s contains the string “four: 4”.

Special String Operations: String Conversion and toString( )

- **String Conversion and toString( )**

  - When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method valueOf( ) defined by String. valueOf( ) is overloaded for all the simple types and for type Object.
    - valueOf( ) returns a string that contains the human-readable equivalent of the value with which it is called. For objects, valueOf( ) calls the toString( ) method on the object.
  - Every class implements toString( ) because it is defined by Object. However, the

    default implementation of toString( ) is seldom sufficient.
  - The **toString( )** method has this general form:

Special String Operations: String Conversion and toString( )

- String toString( )
  - To implement toString( ), simply return a Stringobject that contains the human readable string that appropriately describes an object of your class.
- By overriding toString( ) for classes that you create, you allow them to be fully integrated into Java’s programming environment. For example, they can be used in print( ) and println( ) statements and in concatenation expressions.

Special String Operations: String Conversion and toString( )

Example:

```
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);
    }
}
```

Output:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

Special String Operations: String Conversion and toString( )

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

Character Extraction

- The **String** class provides a number of ways in which characters can be extracted from a **String** object. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation.

- Like arrays, the string indexes begin at zero.

Character Extraction: charAt( )

- charAt( )
  - To extract a single character from a String, charAt( ) method is used. It has this general form:

    char charAt(int where)

    - Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location.

- Example:

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

Eg assigns the value “b” to ch.

ab String S, = "abc";

S. charAt(1)

"abc" b

Character Extraction : getChars( )

- getChars( )
  - Used to extract more than one character at a time.
  - It has this general form:
    - void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
      - Here, sourceStart specifies the index of the beginning of the substring, and
      - sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1.
      - The array that will receive the characters is specified by target.
      - The index within target at which the substring will be copied is passed in targetStart.
      - Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

Example:

```
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);
```

14 - 10 = 4

buf[] 4

d	e	m	o

Character Extraction : getChars( )

```
        }
    }
```

Output:

```
demo
```

Character Extraction : getBytes( )

Achi

Character → Bytes

A → 65

B — 66

- getBytes( )
  - There is an alternative to getChars( ) that stores the characters in an array of bytes. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.
  - Here is its simplest form:
    - byte[ ] getBytes( )
  - getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.
  - Example:

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

65 66 67

Output:

```
65

66

67

68

69
```

Character Extraction : getBytes()

- 70 71 6

}}

Character Extraction: toCharArray( )

- toCharArray( )

  - To convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ).
  - It returns an array of characters for the entire string.
  - It has this general form:
    - char[ ] toCharArray( )

"abc"   abc

  - This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.
  - **Example:**

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

Character Extraction: toCharArray()

```
String s1="hello";
char[] ch=s1.toCharArray(); for(int
i=0;i<ch.length;i++){
System.out.print(ch[i]);
        }
    }}
```

Output:

```
hello
```

ch = hello

s1.toCharArray()

String Comparison

- The **String** class includes several methods that compare strings or substrings within strings.
- **equals( ) and equalsIgnoreCase( )**
  - To compare two strings for equality, use **equals( )**. It has this general form:
    - boolean equals(Object *str*)
      - Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.
  - To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:
    - boolean equalsIgnoreCase(String *str*)
      - Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns

        true if the strings contain the same characters in the same order, and **false** otherwise.

String Comparison: equals( ) and equalsIgnoreCase( )

```
// 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 + " -> " +

                        s1.equals(s2));
                        System.out.println(s1 + " equals " + s3 + " -> " +
                        s1.equals(s3));
                        System.out.println(s1 + " equals " + s4 + " -> " +
```

S1.Equals(S2)

```
s1~~.equals(s4));~~
System.out.println(s1 + " equalsIgnoreCase " + s4 +
" -> " +
s1.equalsIgnoreCase(s4));
```

Output :

```
        }

}
```

```
Hello equals Hello -> true Hello
equals Good-bye -> false Hello equals
HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

String Comparison : regionMatches( )

- **regionMatches( )**
  - The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.
  - There is an overloaded form that allows you to ignore case in such comparisons.
  - Here are the general forms for these two methods:

    s1 s2

    - boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
    - boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*,int *str2StartIndex*, int *numChars*)
      - For both versions, *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.* In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

String Comparison : regionMatches( )

- Example:

```
public class
  RegionMatchesExample{
  public static void
  main(String args[]){
     String str1 = new String("Hello, How are
     you"); String str2 = new String("How");
     String str3 = new String("HOW");

     System.out.print("Result of Test1: " );
     System.out.println(str1.regionMatches(7,
     str2, 0, 3));

     System.out.print("Result of Test2: " );
     System.out.println(str1.regionMatches(7,
     str3, 0, 3));
```

str2,)

str1.regionMatches(7, 0,3

Output:

```
Result of Test1: true Result of
Test2: false Result of Test3:
true
```

String Comparison : regionMatches(

)

```
        System.out.print("Result of Test3: " );
        System.out.println(str1.regionMatches(true, 7, str3, 0, 3));
    }

}
```

boolean

String Comparison: startsWith( ) and endsWith( )

- **startsWith( ) and endsWith( )**
  - **String** defines two routines that are, more or less, specialized forms of **regionMatches( )**.
  - The **startsWith( )** method determines whether a given **String** begins with a specified string.
  - Conversely, **endsWith( )** determines whether the **String** in question ends with a specifiedstring.
  - They have the following general forms:
    - boolean startsWith(String *str*)
    - boolean endsWith(String *str*)
      - Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned.
  - For example,
    - "Foobar".endsWith("bar") and [illegible] String s1 = "Foobar"
      s1.endsWith( )
    - "Foobar".startsWith("Foo") are both **true**.

String Comparison: startsWith( ) and endsWith( )

- A second form of **startsWith( )**, shown here, lets you specify a starting point:
  - boolean startsWith(String *str*, int *startIndex*)
    - Here, *startIndex* specifies the index into the invoking string at which point the search will begin.
  - For example,
    - "Foobar".startsWith("bar", 3) returns **true**.

String Comparison: equals( ) Versus ==

- **equals( ) Versus ==**
  - the **equals( )** method and the **==** operator perform two different operations.
  - The **equals( )** method compares the characters inside a **String** object.
  - The **==** operator compares two object references to see whether they refer to the same instance.

- The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==

class EqualsNotEqualTo {

        public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);
        String s3="welcome";
        System.out.println(s1 + "
        equals " + s2 + " -> " +
        s1.equals(s2));
        System.out.println(s1 + " == " + s2 + "
        -> " + (s1 == s2));
    }

}
```

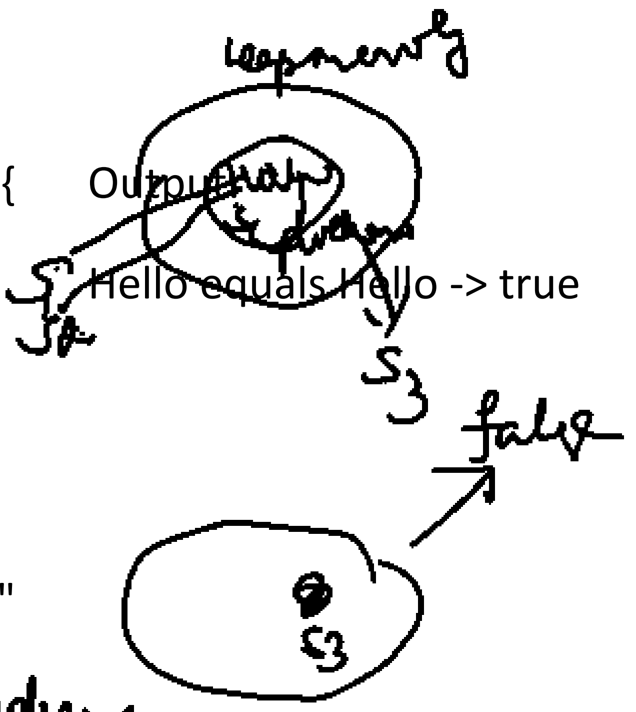
Hello

true

Address

Output:

```
Hello equals Hello -> true
```



String Comparison: compareTo( )

- **compareTo( )**
  - This method is used for comparing two strings lexicographically(dictionary order).
  - Each character of both the strings is converted into a Unicode value for comparison.
    - If both the strings are equal then this method returns 0 else it returns positive or negative value.
    - The result is positive if the first string is lexicographically greater than the second string else the result would be negative.
  - Syntax:
    - int compareTo(String str)
      - Here the comparison is between string literals. For example string1.compareTo(string2) where

String Comparison: compareTo(

string1 and string2 are String literals.

- int compareTo(Object obj)
  - Here the comparison is between a string and an object. For example string1.compareTo("Just a String object") where string1 is a literal and it’s value is compared with the string specified in the method argument.
- If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:
  - int compareToIgnoreCase(String *str*)
    - This method returns the same results as **compareTo( )**, except that case differences are ignored.

String Comparison: compareTo(
)

- Example:

```
public class CompareToExample{
        public static
                void
                main(Strin
                g args[]){
                String
                s1="hello"
                ;
                S
                t
                r
                i
                n
                g
                s
                2
                =
```

```
"
h
e
l
l
o
"
;
S
t
r
i
n
g
s
3
=
"
m
```

s1 s3
hijklm ng

>0

lm
s1 s4
h f
s1 s3

String Comparison: compareTo(

- e
k
l
o
"
;
S
t
r
i
n
g
s
4
=
"
h
e
m
l

o
"
;
S
t
r
i
n
g
s
5
=
"
f
l
a
g
"
;
System.out.println(s1.compareTo(s2));//0

String Comparison: compareTo(

because both are equal

Output:
0
-5

-1

2

```
System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m" System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m" System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
        }
    }
```

String Comparison: compareTo( )

- Example:

```
    // A bubble sort
        for Strings.
        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]);
}
```

String Comparison: compareTo()

Output: Now aid
all come country

for good
is men of the
the their time
to
to

}

}

Modifying a String

- Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods,which will construct a new copy of the string with your modifications complete.
  - **substring( )**
  - **concat( )**
  - **replace( )**
  - **trim( )**

Modifying a String :

substring

- A part of string is called **substring**. In other words, substring is a subset of another string.
- In case of substring startIndex is inclusive and endIndex is exclusive.
- You can get substring from the given string object by one of the two methods:

  String s= "hello";

  - **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).

    String s= "World")

  - **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex, but not including the endIndex (exclusive).

    0

    He

Modifying a String :

substring

- Example:

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

        String s="Sachin Tendulkar";
        System.out.println(s.substring(6));
        //Tendulkar
        System.out.println(s.substring(0,6)
        );//Sachin

      }
```

Output:

```
Tendulkar Sachin
```

S ea

Q S

Modifying a String :

}

substring

Modifying a String : concat( )

- The Java String concat() method concatenates one string to the end of another string. This method returns a string with the value of the string passed into the method, appended to the end of the string.
- Syntax:
  - **public** String concat(String anotherString)
  - **anotherString** : another string i.e. to be combined at the end of this string.

- Example:

```
public class ConcatExample{
public static void main(String args[]){
        String s1="java string";
        String s2 = "so assign it explicitly"; String str1 =
      s1.concat("is immutable");
      System.out.println(str1);
      String str2 = str1.concat(s2);
    System.out.println(str2);


}}
```

str1

str2 = str1.conct(s2)

Output:

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

Modifying a String : replace( )

- The **replace( )** method has two forms.
- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:
  - String replace(char *original*, char *replacement*)
    - Here, *original* specifies the character to be replaced by the character specified by

      replacement. The resulting string is returned.

- For example,
  - String s = "Hello".replace('l', 'w’); puts the string “Hewwo” into **s**.
- The second form of **replace( )** replaces one character sequence with another. It has this general form:
  - String replace(CharSequence *original*, CharSequence *replacement*)

Modifying a String : replace( )

- Example1:

```
public class ReplaceExample1{
        public static void main(String
        args[]){ String s1="hello how are
        you";
        String
        replaceString=s1.replace('h','t');
        System.out.println(replaceString);

    }}
```

Output:

```
tello tow are you
```

- Example2:

```
public class ReplaceExample2{
    public static void main(String
    args[]){ String s1="Hey, welcome
    to DSCE";
```

```
String

replaceString=s1.replace(“DSCE",“ISE");

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

Modifying a String : replace( )

Hey, welcome to ISE

Output:

}}

Modifying a String :trim( )

- **trim( )**
  - The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:
    - String trim( )
  - Here is an example:
    - String s = " Hello World ".trim();
      - This puts the string "Hello World" into **s**.
  - The **trim( )** method is quite useful when you process user commands.
    - For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

Modifying a String: trim( )

```
// Using trim() to process commands.
import java.io.*;
class UseTrim {
        public static void main(String args[])
                throws IOException
                {
                // create a BufferedReader using System.in
                BufferedReader br = new
                BufferedReader(new InputStreamReader(System.in));
                String str;
                System.out.println("Enter 'stop' to quit.");
                System.out.println("Enter State: ");
                do {
                str = br.readLine();
                str = str.trim(); // remove whitespace
                if(str.equals("Illinois"))
                System.out.println("Capital is Springfield.");
                else if(str.equals("Missouri"))
                System.out.println("Capital is Jefferson City.");
                else if(str.equals("California"))
                System.out.println("Capital is Sacramento.");
                else if(str.equals("Washington"))
                System.out.println("Capital is Olympia.");
                // ...
                } while(!str.equals("stop"));
        }
}
```

Event Handling

Introductio n

- Event handling is fundamental to java programming because it is integral to the creation of applets and other types of GUI-based programs.

- Most of the events are generated when the user interacts with a GUI-based program.

- There are several types of events including those generated by the mouse, the keyboard, and various GUI controls, such as push button, scroll bar or checkbox.

The delegation event model

- The modern approach to handling events is based on the *delegation event model, which* defines standard and consistent mechanisms to generate and process events.

- **Concept :**
  - A source generates an event and sends it to one or more listeners.
  - In this scheme, the listener simply waits until it receives an event.
  - Once received, the listener processes the event and then returns.

The delegation event model (contd..)

- **Advantage** : the application logic that processes events is cleanly separated from the user interface logic that generates those events.

- A user interface element is able to "**delegate**" the processing of an event to a separate piece of code.

- In the delegation event model, listeners must register with a source in order to

  receive an event notification.

The delegation event model (contd..)

- This provides an important benefit: notifications are sent only to listeners that want to receive them.

Components of Event Handling

- Event handling has three main components:
  - **Events:** An event is a change in state of an object.
  - **Events Source:** Event source is an object that generates an event.
  - **Listeners:** A listener is an object that listens to the event. A listener gets notified when an event occurs.

Event

- An event is an object that describes a state change in a source.

It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

For example pressing a button, clicking a mouse etc.

Events may also occur that are not directly caused by interactions with a user interface.

For example, an event may be generated when a timer expires, a counter exceeds a value etc.

Event Sources

- A source is an object that generates an event. This occurs when the internal state of that object changes in some way.

- Sources may generate more than one type of event.

- A source must register listeners in order for the listeners to receive notifications about a specific type of event. The general form:

  public void add*Type*Listener*(Type*Listener *el)*

  - Here, *Type* is the name of the event and *el* is a reference to the event listener.
  - *For example, the* method that registers a keyboard event listener is called **addKeyListener( ).**
  - When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting*

the event. In all cases, notifications are sent only *to* listeners that register to receive them.

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

  public void remove*Type*Listener*(Type*Listener *el)*

  - Here, *Type is the name of the event and el is a reference to the event listener.*
  - *For example,* to remove a keyboard listener, call **removeKeyListener( ).**

Event Listeners

- A listener is an object that is notified when an event occurs.

- It has two major requirements:

  - First, it must have been registered with one or more sources to receive notifications about specific types of events.
  - Second, it must implement methods to receive and process these notifications.

- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event.**
  - For example, the MouseMotionListener interface defines 2 methods to

Event Listeners

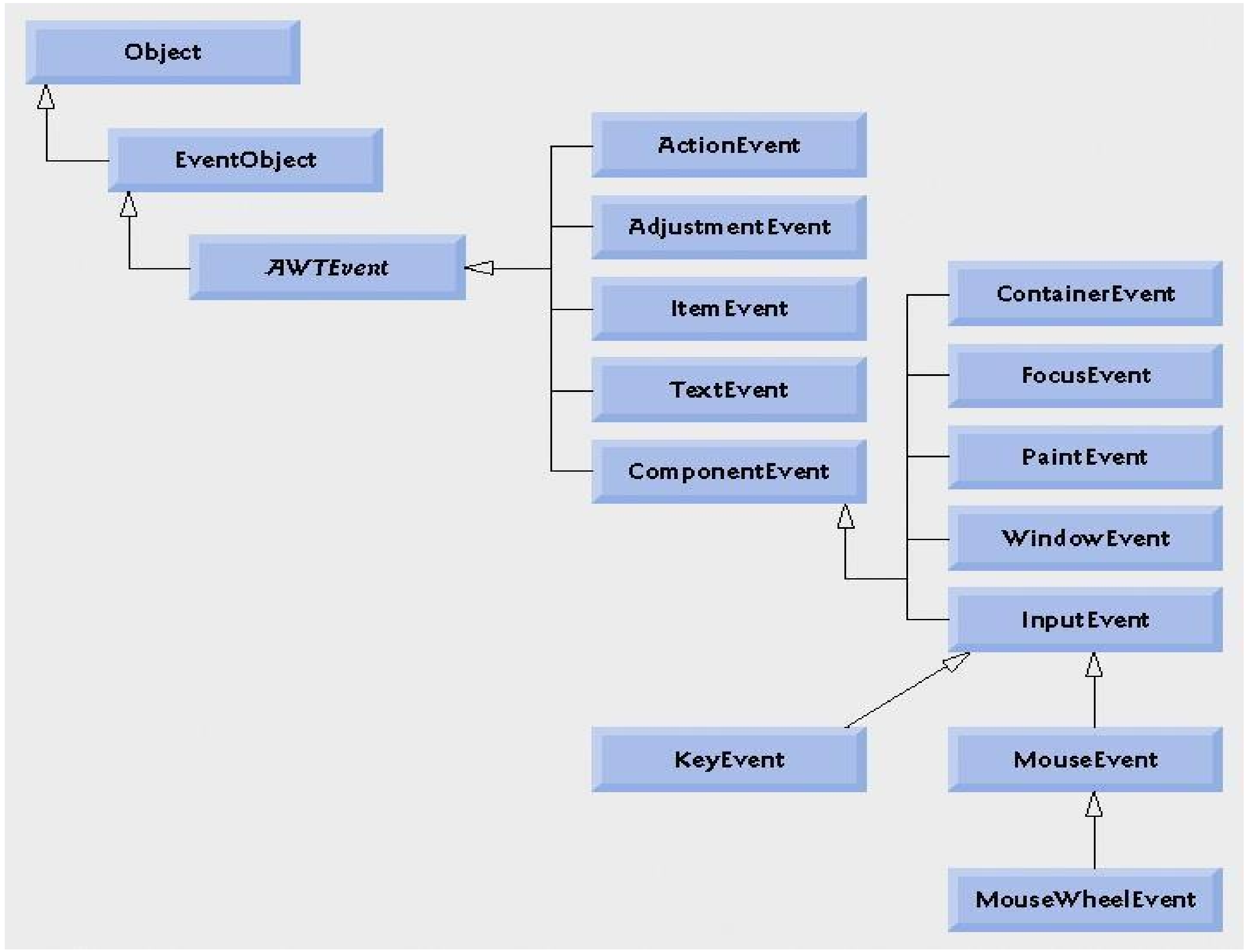
receive notifications when the mouse is dragged or moved.

Event class

- At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. Its one constructor is:
  - **EventObject(Object *src)***
    - Here, *src is the object that generates this event.*

Event class (contd..)

- The class AWTEvent, defined within the java.awt package, is a subclass of EventObject.
  - It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.
  - Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here:
    - int getID( )

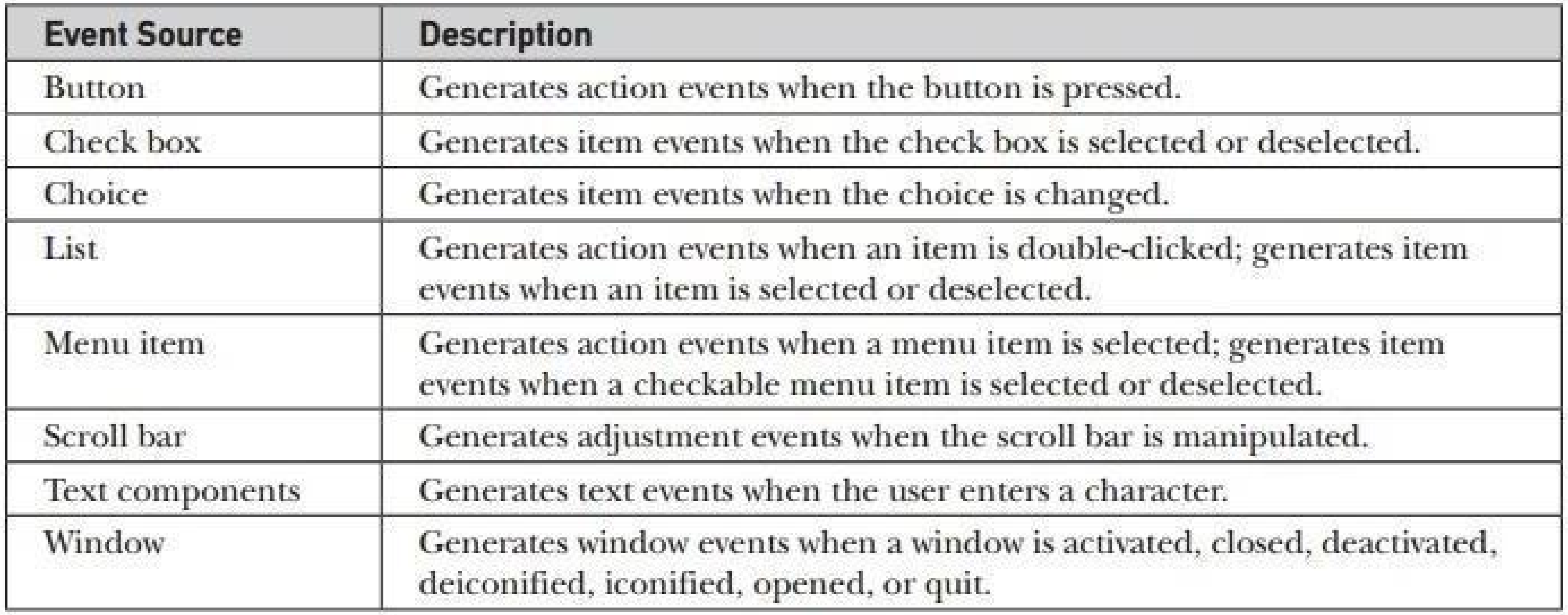

Object
EventObject
AWTEvent
ActionEvent
AdjustmentEvent
ItemEvent
TextEvent
ComponentEvent
ContainerEvent
FocusEvent
PaintEvent
WindowEvent
InputEvent
KeyEvent
MouseEvent
MouseWheelEvent

Important Event Classes and Interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener, and MouseMotionListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener

ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Sources of Events



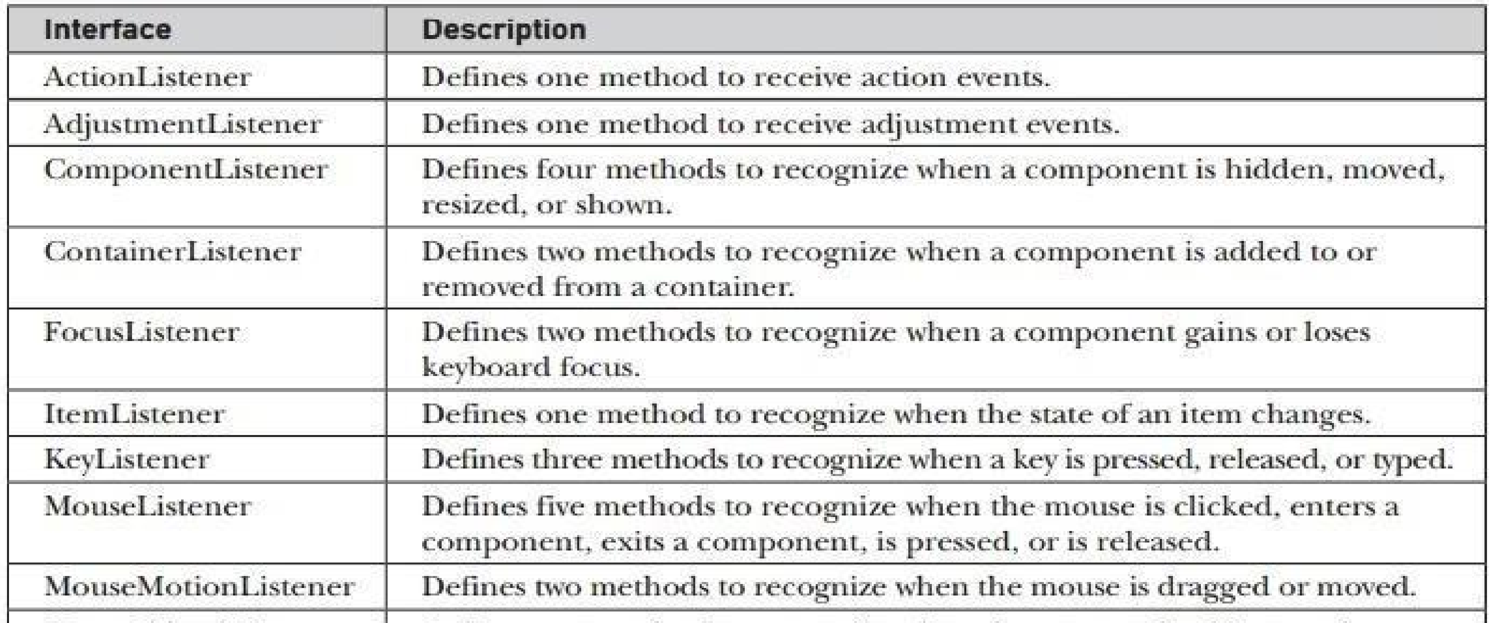
Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

- The delegation event model has two parts: sources and listeners.
- Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.
- When an event occurs, the event source invokes the appropriate method defined

  by the listener and provides an event object as its argument.

Commonly Used Event Listener Interfaces



Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.

Using the Delegation Event Model

- Using the delegation event model is actually quite easy. Just follow these two steps:
  - Implement the appropriate interface in the listener so that it can receive the type of event

    desired.
  - Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- Remember that a source may generate several types of events. Each event must be registered separately.
- Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

MouseListener and MouseMotionListener in Java

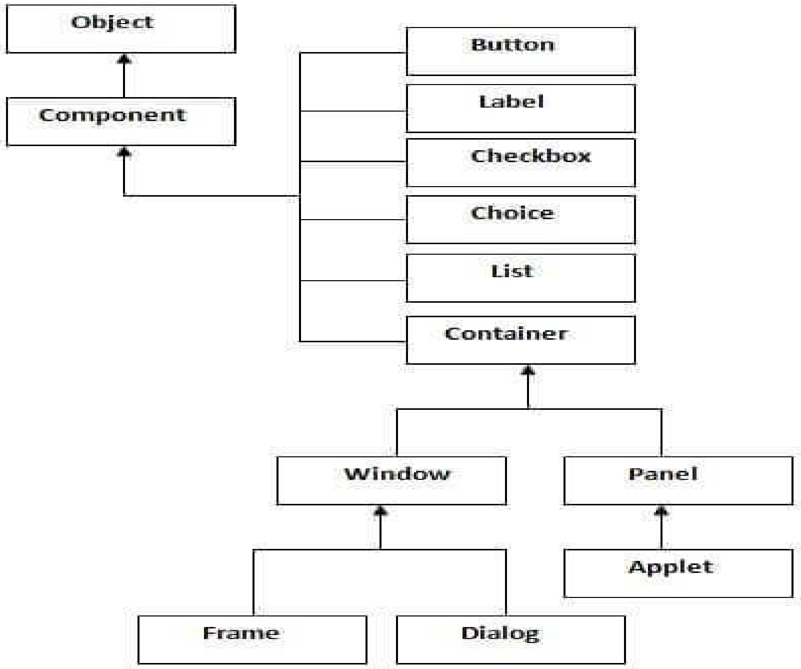
- MouseListener and MouseMotionListener is an interface in java.awt.event package .
- Mouse events are of two types.
  - MouseListener handles the events when the mouse is not in motion.
  - While MouseMotionListener handles the events when mouse is in motion.
- There are five types of events that MouseListener can generate. There are five abstract functions that represent these five events. **The abstract functions are :**
  - **void mouseReleased(MouseEvent e)** : Mouse key is released
  - **void mouseClicked(MouseEvent e)** : Mouse key is pressed/released

- **void mouseExited(MouseEvent e)** : Mouse exited the component
- **void mouseEntered(MouseEvent e)** : Mouse entered the component
- **void mousepressed(MouseEvent e)** : Mouse key is pressed

- There are two types of events that MouseMotionListener can generate. There are two abstract functions that represent these events. **The abstract functions are :**
    - **void mouseDragged(MouseEvent e)** : Invoked when a mouse button is pressed in the component and dragged. Events are passed until the user releases the mouse button.
    - **void mouseMoved(MouseEvent e)** : invoked when the mouse cursor is moved from one point to another within the component, without pressing any mouse buttons.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.


Object
Component
Button
Label
Checkbox
Choice
List
Container
Window
Panel
Applet
Frame
Dialog

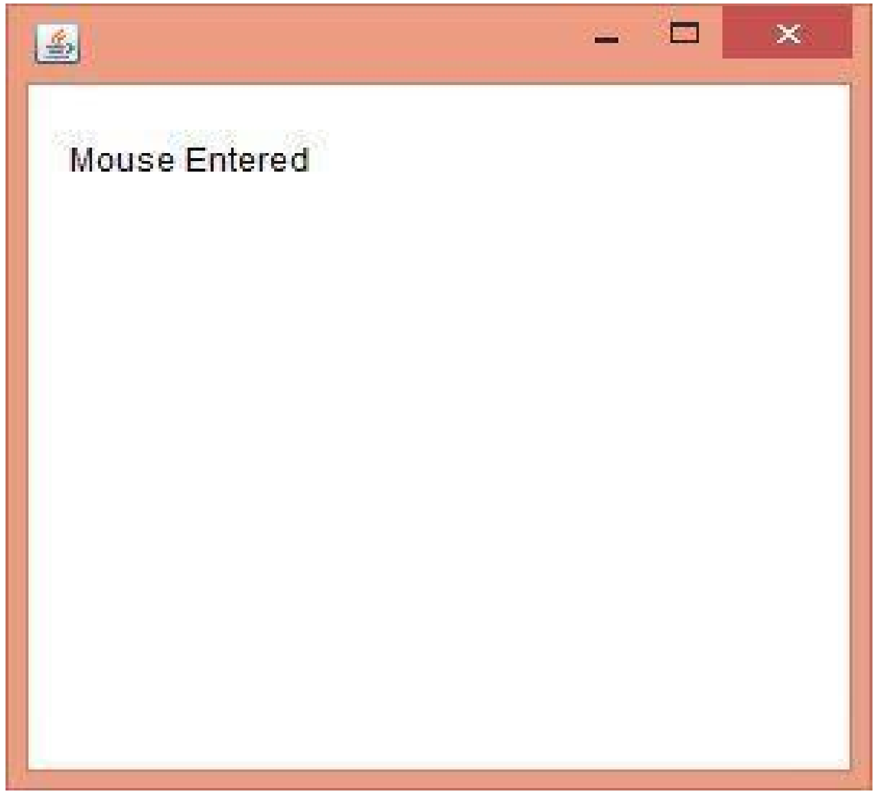
Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame impleme nts
MouseListener
{
  MouseListenerExample()
  {
    addMouseListener(this);/*registering component with the
    Listener.*/
    Label l=new Label();
    l.setBounds(20,50,100,20);
    add(l);//inserts the component
    setSize(300,300);
    setLayout(null);
    setVisible(true);
  }

public void
    mouseClicked(MouseEvent e) {
    l.setText("Mouse Clicked");
  }

  public void
    mouseEntered(MouseEvent e) {
    l.setText("Mouse Entered");
  }

  public void
    mouseExited(MouseEvent e) {
    l.setText("Mouse Exited");
  }

  public void
    mousePressed(MouseEvent e) {
    l.setText("Mouse Pressed");
  }
```

```
    public void mouseReleased(MouseEvent e) {

        l.setText("Mouse Released");

    }

public static void main(String[] args)
    { new MouseListenerExample();
}

}
```

- Note: Label does not generate any Event. It is just used to display the content on the Frame.
- Add(l)// Adds the object label the frame.
- setLayout(**null**); null layout means absolute positioning - you have to do all the work in your code.
- The object of Label class is a component for placing text in a container
- **setSize(300,300);//frame size 300 width and 300 height**

Output:

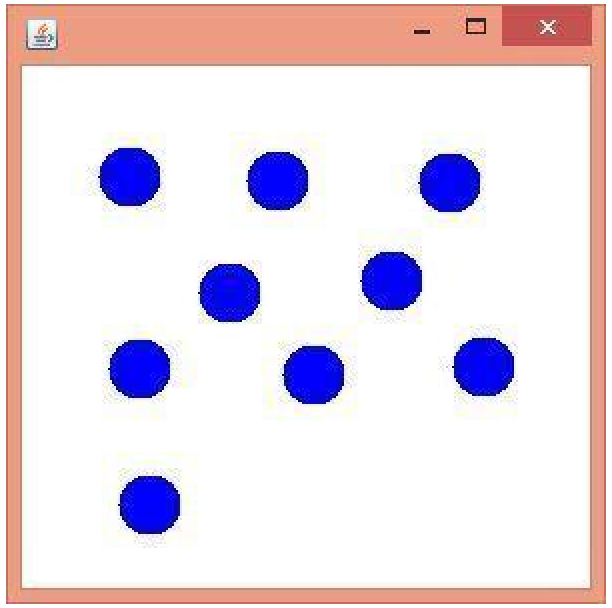


```
Mouse Entered
```

Java MouseListener Example 2

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample2 extends Frame implements
  MouseListener{ MouseListenerExample2(){
    addMouseListener(this);



    setSize(300,300);
    setLayout(null);
    setVisible(true);
  }
  public void mouseClicked(MouseEvent e) {
    Graphics g=getGraphics();
    g.setColor(Color.BLUE);
    g.fillOval(e.getX(),e.getY(),30,30);
  }
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}

public static void main(String[] args) {
  new MouseListenerExample2();
}
```

Output:



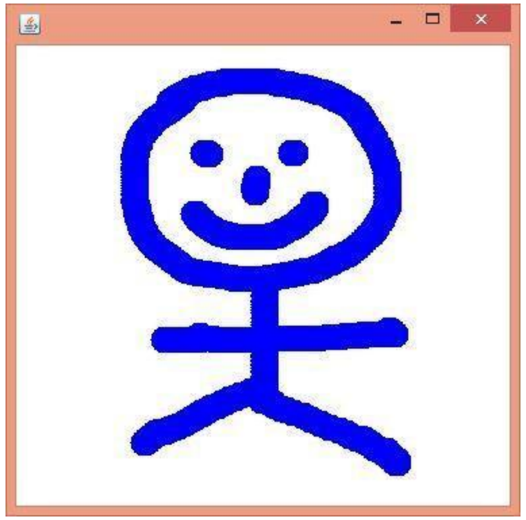
Java MouseListener Example

```
}
```

Java MouseMotionListener Example

```
import java.awt.*;

import java.awt. event.*;

public class MouseMotionListenerExample extends Frame implements MouseMotionListener
{
      MouseMotionListenerExample()
   {
      addMouseMotionListener(this);

      setSize(300,300);
      setLayout(null);
      setVisible(true);
   }


public void mouseDragged(MouseEvent e)
{
    Graphics g=getGraphics();
   g.setColor(Color.BLUE);
   g.fillOval(e.getX(),e.getY(),20,20);
}

public void mouseMoved(MouseEvent e) {}
```

Output:



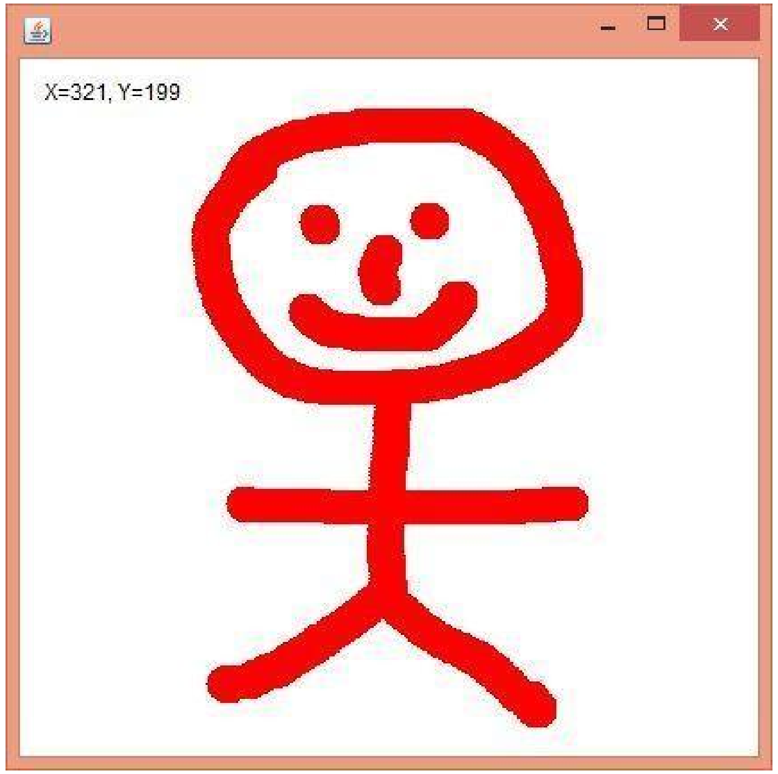
Java MouseMotionListener Example

```
public static void main(String[] args) {
    new MouseMotionListenerExample();
}
}
```

Java MouseMotionListener Example 2

```
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
public class Paint extends Frame implements MouseMotionListener{ Label l;
   Color c=Color.BLUE;
   Paint(){
   l=new Label();
   l.setBounds(20,40,100,20); add(l);

   addMouseMotionListener(this);


   setSize(400,400);
   setLayout(null);
   setVisible(true);
}
public void mouseDragged(MouseEvent e) {
   l.setText("X="+e.getX()+", Y="+e.getY());
   Graphics g=getGraphics();
   g.setColor(Color.RED);
   g.fillOval(e.getX(),e.getY(),20,20);
}
public void mouseMoved(MouseEvent e) {
   l.setText("X="+e.getX()+", Y="+e.getY());
}
public static void main(String[] args) {
   new Paint();
}
}
```

Out
put
:



```
X=321, Y=199
```

Write a java program for handling keyboard events

Write a java program for handling keyboard events

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){

        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        add(l);add(area);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
public void keyPressed(KeyEvent e) {
l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e) {
l.setText("Key Released");
}
public void keyTyped(KeyEvent e) {
l.setText("Key Typed");
                              }

public static void main(String[] args) {
new KeyListenerExample();
                              }
                           }
```

Adapter Classes

- Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

- An adapter class basically provides an empty implementation of all methods in an listener interface.

- Adapter classes are useful when we want to receive and process only some of the events

  that are handled by a particular event listener interface.

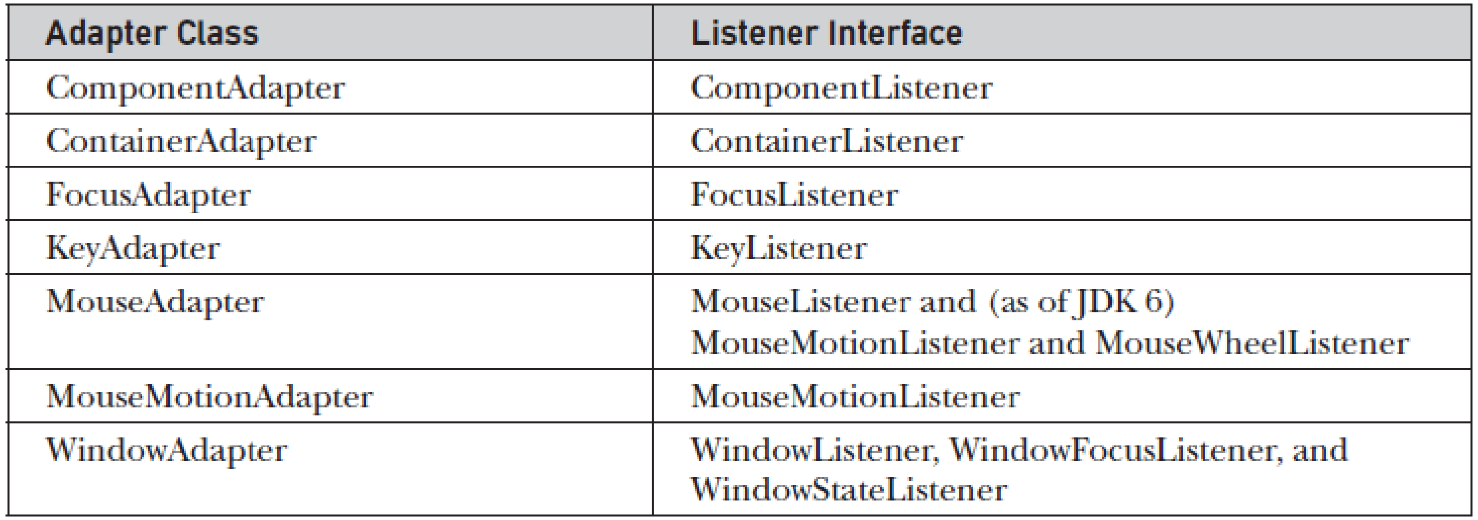
- We can define a new class to act as an event listener by extending one of the adapter

Adapter Classes

classes and implementing only those events in which we are interested.

Adapter Classes

- Some of the adapter classes available in the *java.awt.event* package are listed below:

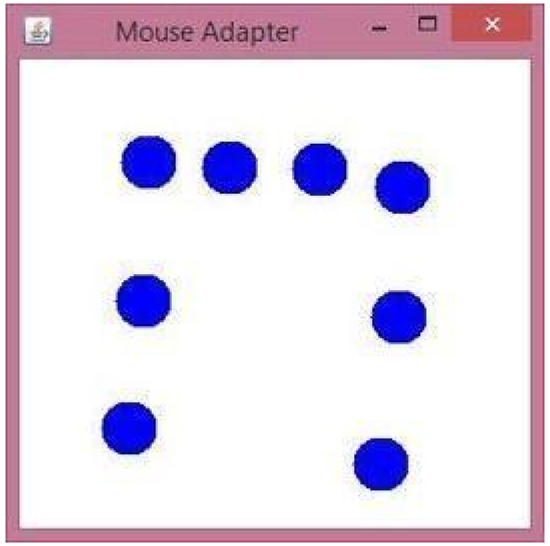


Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

Adapter Class – Example: Mouse Adapter

```
 import java.awt.*;

import java.awt.event.*;

public class MouseAdapterExample extends
    MouseAdapter{ Frame f;
    MouseAdapterExample(){

        f=new Frame("Mouse
        Adapter");
        f.addMouseListener(this);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);

    }

    public void mouseClicked(MouseEvent e) {
        Graphics g=f.getGraphics();
        g.setColor(Color.BLUE);
```

Output:

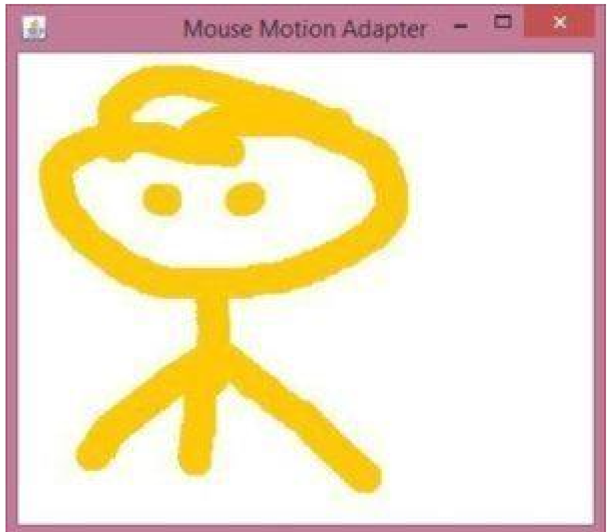


```
        g.fillOval(e.getX(),e.getY(),30,30);
    }

public static void main(String[] args) {
    new MouseAdapterExample();
}

}
```

Adapter Class – Example: MouseMotion Adapter

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionAdapterExample extends
   MouseMotionAdapter{ Frame f;
   MouseMotionAdapterExample(){
      f=new Frame("Mouse Motion
      Adapter");
      f.addMouseMotionListener(this);
      f.setSize(300,300);
      f.setLayout(null);
      f.setVisible(true);

   }
public void mouseDragged(MouseEvent e) {
   Graphics g=f.getGraphics();
   g.setColor(Color.ORANGE);
   g.fillOval(e.getX(),e.getY(),20,20);
```

Output:



```
}
public static void main(String[] args) {
   new MouseMotionAdapterExample();
}
}
```

Inner Classes

- An inner class is a class which is defined inside another class.
- The inner class can access all the members of an outer class, but vice-versa is not true.
- The mouse event handling program which was simplified using adapter classes can further be simplified using inner classes.
- **Syntax:**

```
class
{
class
{}
//other attributes and methods
}
```

For example: Suppose you have a class that extends the **Frame** class. Suppose now you want to use a **WindowAdapter** class to handle window events. Since Java does not support multiple inheritance so your class cannot extend both the **Frame** and **WindowAdapter** class. A solution to this problem is to define an inner class i.e. a class inside the Frame subclass that extends the WindowAdapter class.

```
import javax.swing.*;
  import java.awt.event.*;
  import java.awt.*;
  class NamedInnerClass  extends JFrame
  {
       NamedInnerClass()
       {
         addWindowListener(new MyAdapterClass());
       }
       class MyAdapterClass extends WindowAdapter
      {
         public void windowClosing(WindowEvent e)
```

```
        {
          System.exit(0);
        }
      }
  }
    class InnerEventClassJavaExample
   {
      public static void main(String args[])
      {
        NamedInnerClass frame = new NamedInnerClass();
        frame.setTitle("Named Inner Class Java Example");
        frame.setBounds(200,150,180,150);
        frame.setVisible(true);
      }
   }
```

Swing

Introductio
n

- Swing in Java is a lightweight GUI toolkit which has a wide variety of widgets for building optimized window based applications.

- It is a part of the JFC(Java Foundation Classes).

  - The Java Foundation Classes are a set of GUI components which simplify the development of desktop applications.

- It is build on top of the AWT API and entirely written in java.

- It is platform independent unlike AWT and has lightweight components.

Origin of swing

- The AWT defines a set of controls , windows and dialog boxes that support a usable, but limited graphical interface.
- The reason for this limited nature of the AWT is that it translates its various visual components into their corresponding platform-specific equivalents, or peers.
- This means that look and feel of a component is defined by the platform, not by java.
- Because AWT components use native code resources, they are referred to as heavyweight.

Disadvantages of AWT

The use of native peers has following problems.

1) Because of variation between operating systems, a component might look, or even act, differently on different platforms.
2) The look and feel of each component was fixed(because it is defined by the platform) and could not be(easily) changed.

3) The use of heavy weight components caused some frustrating restrictions. For

   They need to access more resources.

Swing is built on the AWT

Although swing elements eliminates a number of the limitations inherent in the AWT, swing does not replace it. Instead swing is built on the foundation of the AWT.

Swing also uses the same event handling mechanism as the AWT.

Two key swing features

1)Swing components are lightweight

- Written entirely in java.
- Do not translate into platform specific peers.
- Rendered using graphical primitives
- Lightweight components are efficient and flexible.
- The look and feel of each component is determined by Swing, not by the underlying operating system.

2) Swing supports a pluggable look and feel.

Two key swing features

- Look and feel is under control of swings

- It is possible to change the way the component is rendered without affecting any of its other aspects.

Advantages of Swing

- Look and feel is consistent across all platforms.
- It is possible to create a look and feel that acts like a specific platform.
- It is also possible to design a custom look and feel.
- The look and feel can also be changed dynamically at run time.

Note: The look-and-feels, such as metal and Motif, are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel.

Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.(Need to access more resources depending on the compiler)	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel.**(If an application is developed in one OS, Same look and feel will not be in another OS)	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.

Components and Containers

A swing GUI consists of two key items : components and containers.

All containers are also components.

Component: is an independent visual control, such as push button or slider.

Container: holds the group of components.

Container is a special type of components that is designed to hold other components. In order for component to be displayed, it must be held within a container. Thus all swing GUI will have a least one container.

Because containers are components, a container can also hold other containers.

Component
s

- A component is an independent visual control.
- Swing Framework contains a large set of components which provide rich

  functionalities and allow high level of customization.

- They all are derived from JComponent class. All these components are lightweight components. This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

Components

- All of Swing’s components are represented by classes defined within the package **javax.swing**.

Containers

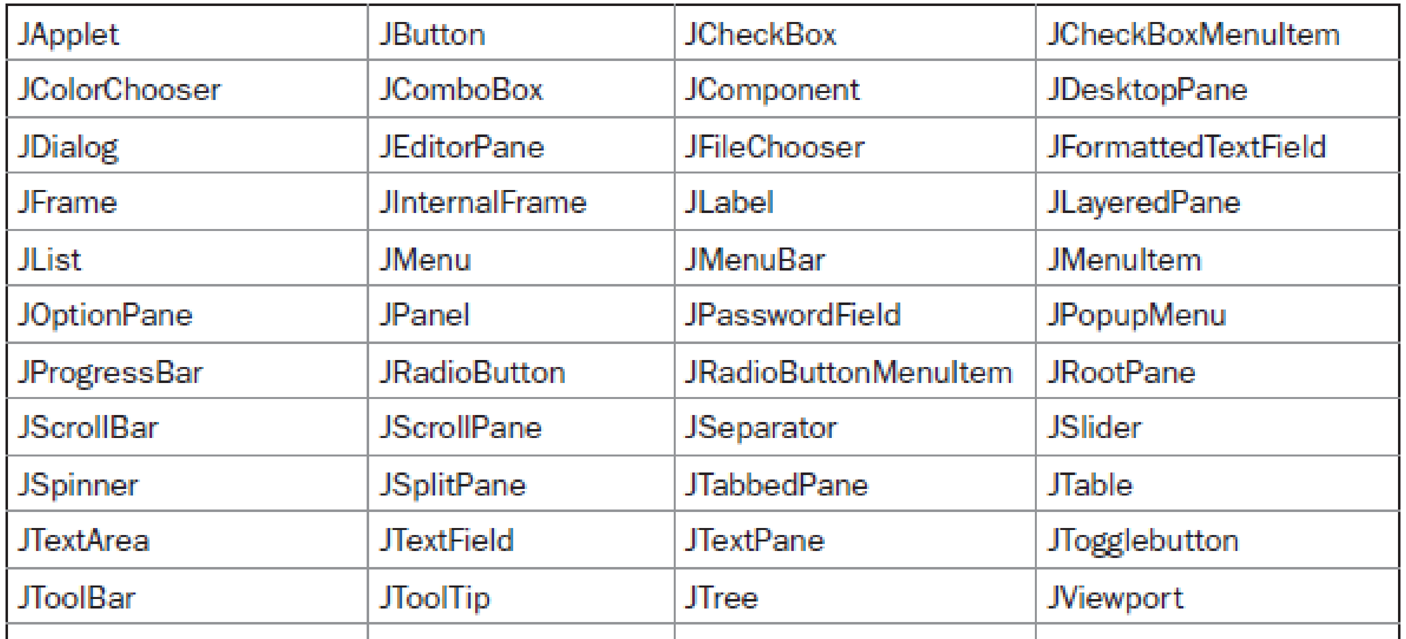
- A container holds a group of components. It provides a space where a component can be managed and displayed.

- Containers are of two types:

1) **Top level Containers**
   - It inherits Component and Container of AWT.
   - It cannot be contained within other containers.
   - Heavyweight.
   - Example: JFrame, JDialog, Japplet,Jwindow
   - Every containment hierarchy must begin with a top-level container.
   - The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

2) **Lightweight Containers**
   - It inherits JComponent class.
   - It is a general purpose container.
   - It can be used to organize related components together.
   - Example: JPanel

Components in swing



JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport

SWING Containers

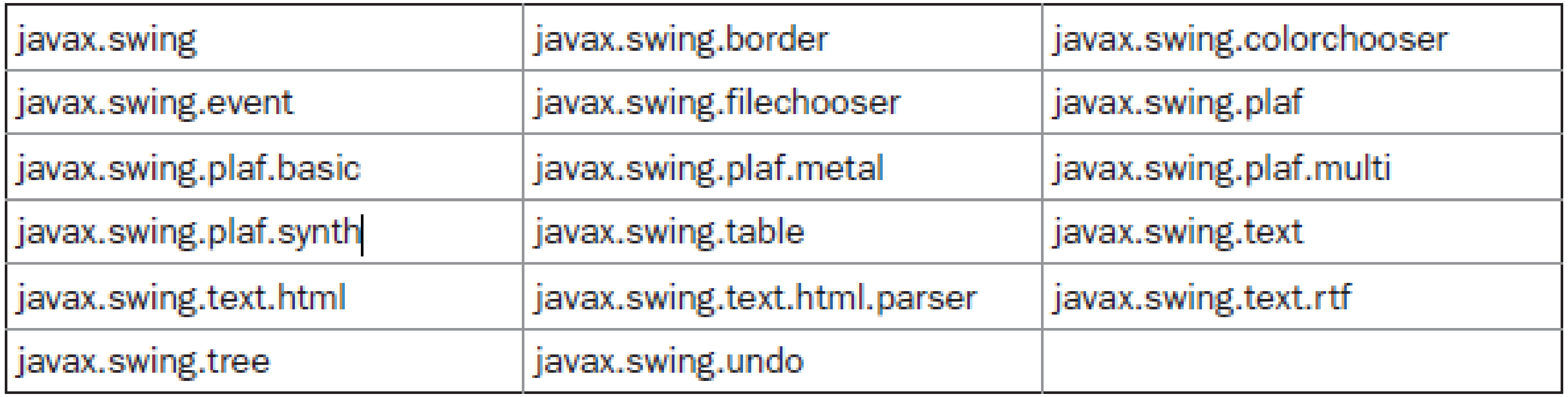
- Following is the list of commonly used containers while designing GUI using SWING

Sl.No.	Container & Description
1	Panel: JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.
2	Frame: A JFrame is a top-level window with a title and a border.

3	Window: A JWindow object is a top-level window with no borders and no menubar.

The Swing Packages

- Swing is a very large subsystem and makes use of many packages.
- The main package is **javax.swing**.
  - This package must be imported into any program that uses Swing.
  - It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.



javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

A simple swing application1

```
import javax.swing.*;
Class SwingDemo
{
   SwingDemo()
   {
          JFrame f = new JFrame(“A simple swing
          application”); f.setSize(275,100);
          f.setDefaultCloseOperation(JFrame.EXIT_ON_CL
          OSE); JLabel l = new JLabel(“swing means
          powerfull GUI”); f.add(l);
           f.setVisible(true);
   }


  public static void main(String args[])
   {
              SwingDemo s =    new SwingDemo();
   }
}
```

Output:

A simple swing application 2

import javax.swing.*;

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.*;
import java.awt.*;
public class AddGui
{

  public static void main(String[] args)
  {
         Addition obj=new Addition();
  }
}
 class Addition extends JFrame implements ActionListener
{
  JTextField t1;
  JTextField t2;
  JButton b;
  JLabel l1;

  public Addition()
  {
         t1=new JTextField(20);
      t2=new JTextField(20);
         b=new JButton("ok");
```

```
        setLayout(new FlowLayout());
        l1=new JLabel("Result");
        add(l1);
        add(t1);
        add(t2);
        add(b);
        b.addActionListener(this);

        setVisible(true);
        setSize(400,400);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent ae)
{
        int num1=Integer.parseInt(t1.getText());
        int num2=Integer.parseInt(t2.getText());
        int value=num1+num2;
        l1.setText(value+"");
}



}
```

A simple swing application 3

```
import java.awt.FlowLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
```

```
public class FirstGui
{

    public static void main(String[] args)
    {
                Abc obj=new Abc();
    }
}
class Abc extends JFrame
{
    public Abc()

    {
       setLayout(new FlowLayout());
                JLabel l1=new JLabel("Hello World");
                JLabel l2=new JLabel("Welcome");
                add(l1);
                add(l2);
                setVisible(true);
                setSize(400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }
}
```

A simple swing application 4

```
import javax.swing.*;
class SwingDemo
{
```

```
SwingDemo()
{
JFrame f = new JFrame("A simple swing application");
f.setSize(275,100);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JLabel l = new JLabel("swing means powerfull GUI");
f.add(l);
f.setVisible(true);
}
public static void main(String args[])
{
SwingDemo s =  new SwingDemo();
}
}
```

A simple swing application 5

```
import javax.swing.JFrame;
import javax.swing.JLabel;
class SwingDemo1 extends JFrame
{

SwingDemo1()
{

//f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JLabel f = new JLabel("swing means powerfull GUI");
setSize(275,100);
add(f);
setVisible(true);
}
public static void main(String args[])
{
    new SwingDemo1();
}
}
```

Thank you