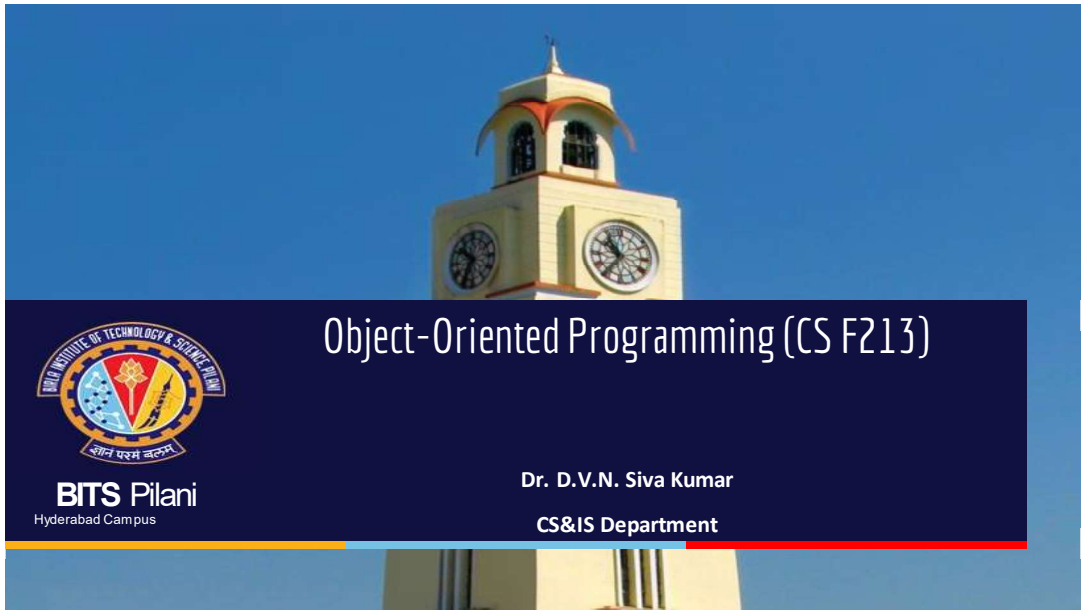


Object-Oriented Programming (CS F213)

BITS Pilani
Hyderabad Campus

Dr. D.V.N. Siva Kumar

CS&IS Department

Strings

In Java, a string is a sequence of characters.

1. Every string you create is actually an object of type **String**. Even string constants are actually **String** objects.

For example, in the statement

System.out.println("This is a String, too");

the string **"This is a String, too"** is a **String** object

2. String is immutable in nature. It means once a **String** object is created, its contents cannot be altered.

Whenever we change any string, a new instance is created. If you need to change a string, you can always create a new one that contains the modifications.

3. Mutable strings can be created by using **StringBuffer** and **StringBuilder** classes.

Strings Construction:

The easiest way is to use a statement like this:

String myString = "this is a test";

Agenda

- Strings
- StringBuffer and StringBuilder
- StringTokenizer
- Regular Expressions

String Constructors

The **String** class supports several constructors.

1. To create an empty **String**, call the default constructor. For example,

String s = new String();

2. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char *chars*[])

For example,

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string "abc".

3. We can specify a subrange of a character array as an initializer using the following constructor:

String(char *chars*[], int *startIndex*, int *numChars*)

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use.

For example,

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);     // s=cde
```

Example of creating a String object

Example:

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

        String s2=new String("example");    //creating Java string by new keyword

        System.out.println(s1);
        System.out.println(s2);
    }
}
```

- When we create a string literal, first it is checked in string constant pool. 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, in the above case if we create String s3 = “Hello”; the new string object will not be created and s3 will point to the same instance as that of s1.
- When we use **new** keyword, a new object is created every time.

Example program on array of Strings

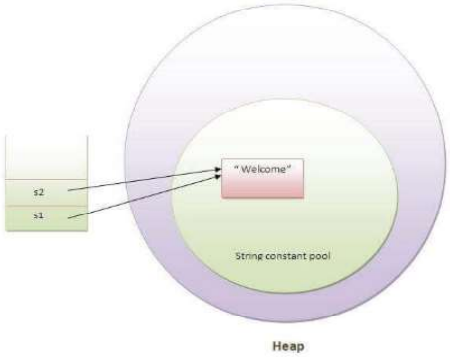
```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```



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



Some Methods of String class

1. **equals( )**: It is used to check the equality of two strings.
2. **length():** It is used to obtain the length of a string.
3. **charAt( )**: It is used to obtain the character at a specified index within a string.

The general form of the above methods is as follows:

boolean equals(*secondStr*)
int length( )
char charAt(*index*)

4. **valueOf():** It converts different types of values into string. We can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

For example,
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);//concatenating string with 10

Example

```java
import java.lang.String;
public class Example2
{
    public static void main(String args[])
    {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = new String("Hello");
        String s4 = new String("heLLo");

        System.out.println("s1 == s2: " + (s1 == s2));
        System.out.println("s1 == s3: " + (s1 == s3));
        System.out.println("s1.equals(s3): " + s1.equals(s2));
        System.out.println("s2.equals(s4): " + s2.equals(s4));
        System.out.println("s2.equalsIgnoreCase(s4): " + s2.equalsIgnoreCase(s4));
        System.out.println("s4.substring(2): "+s4.substring(2));
        System.out.println("s4.substring(0,3): "+s4.substring(0,3));
    }
}
```

Output:

```
$ java Example2
s1 == s2: true
s1 == s3: false
s1.equals(s3): true
s2.equals(s4): false
s2.equalsIgnoreCase(s4): true
s4.substring(2): LLo
s4.substring(0,3): heL
```


List of Some String Class Methods

- The String class has a set of built-in methods that you can use on strings. Some of them are mentioned below.

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
getChars()	Copies characters from a string to an array of chars	void
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int

compareTo( )

- It can be used to know which string is *less than*, *equal to*, or *greater than* the other string.
- The general form is provided below:

int compareTo(String *str*)

Note: Here, *str* is the **String** being compared with the invoking string.

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.

String Class Methods

Method	Description	Return Type
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
copyValueOf()	Returns a String that represents the characters of the character array	String
format()	Returns a formatted string using the specified locale, format string, and arguments	String
isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
getBytes()	Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array	byte[]

String Class Methods

Method	Description	Return Type
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lowercase letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

What could be the output?

```
class demo10 {
public static void main(String[] args) {
String s1 = "Hello";
String s2 = "Hello";
if(s1==s2)
{
System.out.println("Equal");
}
else
{
System.out.println("Not Equal");
}
}
}
```

Note: The **==** operator compares two object references to see whether they refer to the same instance

Output:
Equal

Few Questions

What could be the output?

```
class demo10 {
public static void main(String[] args) {
String s1 = new String("Hello“);
String s2 =new String("Hello“);
if(s1==s2)
{
System.out.println("Equal");
}
else
{
System.out.println("Not Equal");
}
}
}
```

Output:
Not Equal

What is the output of below program?

```
public class StringTest {

        public static void main(String[] args) {
                String s1 = new String("pankaj");
                String s2 = new String("PANKAJ");
                System.out.println(s1 = s2);
        }

}
```


Question :

Can you convert String to int and vice versa? If yes, then try to write the code.

Example :

Input : “1254” (string type) Output : 1254 (integer type)

Input : 7895 (integer type) Output : “7895” (string type)

Solution :

It’s a simple yet tricky program, it will print “PANKAJ” because we are assigning s2 String to String s1. Don’t get confused with == comparison operator.

Solution :

Yes, you can convert string to int and vice versa.You can convert string to an integer using the **valueOf()** method and the **parseInt()** method of the Integer class. Also, you can convert an integer to string using the valueOf() method of the String class. Below, I have given a program which shows the string to integer and integer to string conversion.

```
public class Conversion{
public static void main(String [] args){
String str = "1254";
int number = 7895;

// convert string to int using Integer.parseInt() method
int parseIntResult1 = Integer.parseInt(str);

// convert string to int using Integer.valueOf() method
int valueOfResult1 = Integer.valueOf(str);
System.out.println("Converting String to Integer:");
System.out.println("Using the Integer.parseInt() method : "+parseIntResult1);
System.out.println("Using the Integer.valueOf() method : "+valueOfResult1);
System.out.println("\n");

// convert integer to string using String.valueOf() method
String valueOfResult2 = String.valueOf(number);
System.out.println("Converting Integer to String :");
System.out.println("Using the String.valueOf() method : "+valueOfResult2); }
}
```

```
String firstString = "Gaurav";
String secondString = "Gaurav";
String thirdString = new String("Gaurav");
```


What is the output of the below code?

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

Output:
four: 22

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

Output:
four: 4

Solution :

two

Explanation :

Mutable Strings

In Java, there are two options by which strings are modifiable, i.e., mutable.

1. **StringBuffer**
2. **StringBuilder**

Note: String, StringBuffer, and StringBuilder classes are defined in java.lang. They are available to all programs automatically.
All are declared **final**, which means that none of these classes may be subclassed.

StringBuffer

- It supports a modifiable string. It represents growable and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

1. **StringBuffer( ):** The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
2. **StringBuffer(int *size*):** It accepts an integer argument that explicitly sets the size of the buffer.
3. **StringBuffer(String *str*):** It accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
4. **StringBuffer(CharSequence *chars*):** It creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

Few Methods of StringBuffer

- **insert(int offset, String s)** - It is used to insert the specified string with this string at the specified position. The insert() method is overloaded; like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double), etc.
- **replace(int startIndex, int endIndex, String str)** - It is used to replace the string from specified startIndex and endIndex.
- **reverse()** - It is used to reverse the string.

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

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

The output of this example is shown here:

```
I like Java!
```

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

Here is the output produced by the program:

```
abcdef
fedcba
```


StringBuffer

```
// 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
```

Note:
length() returns length of the StringBuffer.
capacity)() returns the total allocated capacity.
The above methods general form is provided below:

```
int length( )
int capacity( )
```


StringBuilder

- **StringBuilder** is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe.
- The advantage of **StringBuilder** is faster performance than StringBuffer.
- However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

Constructors of StringBuilder

1) **StringBuilder()** - Creates an empty string builder with a capacity of 16 (16 empty elements).
2) **StringBuilder(CharSequence cs)** - Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.
3) **StringBuilder(int initCapacity)** - Creates an empty string builder with the specified initial capacity.
4) **StringBuilder(String s)** - Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

Note: Methods are very similar to StringBuffer class.

Tokenization

- Tokenization is essentially splitting a **phrase, sentence, paragraph, or an entire text document** into smaller units, such as individual words or terms.
- Each of these smaller units are called **tokens**.

Some Use cases of Tokenization:
- To count the number of words in the text.
- To count the frequency of the word, that is, the number of times a particular word is present.
- It is useful in performing the most basic step in parsing the text (NLP). It is important because **meaning of the text could easily be interpreted by analyzing the words present in the text.** It is useful in language translation, speech processing, etc.
- It is useful in building indexes that would be used by search engines to retrieve the documents of users' interest.

String vs StringBuffer vs StringBuilder

When to use which one :

a. If a string is going to remain constant throughout the program, then use a String class object because a String object is immutable.
b. If a string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a single thread, using a StringBuilder is good enough.
c. If a string can change, and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous so you have thread-safety.

Tokenization in Java (StringTokenizer)

- The **java.util.StringTokenizer** class allows you to break a sentence or paragraph into tokens.
- **StringTokenizer** implements **Enumeration** interface with which we can enumerate the individual tokens contained in the given input string using.
- To use StringTokenizer, we specify an **input string** and a string that contains **delimiters**.
- Delimiters are characters **that separate tokens**. Each character in the delimiters string is considered a valid delimiter. For example, ",;:" sets the delimiters to a comma, semicolon, and colon.
- The default set of delimiters consists of the whitespace characters: space, tab, form feed, newline, and carriage return.

Constructors:
1)**StringTokenizer(String str)** – str is the string to be tokenized. Default delimiters are used.
2)**StringTokenizer(String str, String delimiters)** - delimiters is a string that specifies the delimiters.
3) **StringTokenizer(String str, String delimiters, boolean delIlmAsToken)** – str is the string to be tokenized, delimiter and **delIlmAsToken**. If **delIlmAsToken** value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of StringTokenizer:

Once the StringTokenizer object is created, the following methods are helpful in extracting the consecutive tokens.

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

Regular Expressions

- A regular expression is a string of characters that describes a character sequence, i.e., **pattern**.
- A pattern can be used to find matches in other character sequences. Also, can be used for '**text search**' as well as '**text replace**' operations.
- These are mainly used for creating passwords with required specifications and email validation in real life applications.
- Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, a regular expression represents a general form that can match several different specific character sequences.
- In Java, **java.util.regex** package supports regular expression processing.
- Two classes **Pattern** and **Matcher** work together to support regular expression processing.
- **Pattern** can be used to define a regular expression and **Matcher** can be used to match the pattern against another character sequence.

Example program of StringTokenizer class

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
    "author=Schildt;" +
    "publisher=Oracle Press;" +
    "copyright=2019";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) {
      String key = st.nextToken();
      String val = st.nextToken();
      System.out.println(key + "\t" + val);
    }
  }
}
```

The output from this program is shown here:

```
title Java: The Complete Reference
author Schildt
publisher Oracle Press
copyright 2019
```


Regular Expressions

- The java.util.regex package provides following classes and interfaces for regular expressions:-
  1. MatchResult interface
  2. Pattern class
  3. Matcher class (implements MatchResult interface)
  4. PatternSyntaxException class

Pattern class

- It defines no constructors.
- It is used to define a pattern for regex engine.
- A pattern is created by calling the **compile()** factory method. The compile() method transforms the string into a pattern that can be used for pattern matching by the Matcher class.
- Once a pattern object is created, we will use it to create a Matcher. This can be done by calling the **matcher().**

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Matcher class (Cont…)

3. **String group():** It can be used to obtain a string containing the last matching sequency by calling group(), i.e., it returns the matching string. If no match exists, IllegalStateException is thrown.
4. **String group(int group):** It returns the input subsequence captured by the given group during the previous match operation.
5. **int groupCount():** Returns the number of capturing groups in this matcher's pattern.
6. **int start():** It returns the start index of the previous match.
7. **int end():** It returns the offset after the last character matched.
8. **String replaceAll(String replacement):** It replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
9. **String replaceFirst(String replacement):** It replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.

Matcher Class

- It implements the MatchResult interface.
- Once we create a Matcher by calling the **matcher()** method defined by Pattern, we can use to Matcher methods to perform various pattern matching operations.

1. **boolean matches()**: It returns true if the sequence and the pattern match. Otherwise, it returns false. It is to be noted that the entire sequence must match the pattern.
2. **boolean find():** It can be used to determine if a subsequence of the input sequence matches the pattern. It returns true if there is a matching subsequence and false otherwise**. It can be called repeatedly**, allowing it find all matching subsequences. Each call to find() begins where the previous one left off.


```
import java.util.regex.*;
class RegExpr {
  public static void main(String args[]) {
    Pattern pat;
    Matcher mat;
    boolean found;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    found = mat.matches(); // check for a match

    System.out.println("Testing Java against Java.");
    if(found) System.out.println("Matches");
    else System.out.println("No Match");

    System.out.println();

    System.out.println("Testing Java against Java SE.");
    mat = pat.matcher("Java SE"); // create a new matcher

    found = mat.matches(); // check for a match

    if(found) System.out.println("Matches");
    else System.out.println("No Match");
  }
}
```

The output from the program is shown here:

```
Testing Java against Java.
Matches

Testing Java against Java SE.
No Match
```

Example program of demonstrating pattern matching using **matches()**

Flags of compile() method

Flags in the `compile()` method change how the search is performed. Here are a few of them:

- `Pattern.CASE_INSENSITIVE` - The case of letters will be ignored when performing a search.
- `Pattern.LITERAL` - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- `Pattern.UNICODE_CASE` - Use it together with the `CASE_INSENSITIVE` flag to also ignore the case of letters outside of the English alphabet


```
import java.util.regex.*;

class RegExpr3 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("test");
    Matcher mat = pat.matcher("test 1 2 3 test");

    while(mat.find()) {
      System.out.println("test found at index " +
                         mat.start());
    }
  }
}
```

Example program of using **find()** to multiple subsequences

The output is shown here:

```
test found at index 0
test found at index 11
```

Note: The **find( )** method can be used to search the input sequence for repeated occurrences of the pattern because each call to **find( )** picks up where the previous one left off.


```
// Use find() to find a subsequence.
import java.util.regex.*;

class RegExpr2 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java SE");
    System.out.println("Looking for Java in Java SE.");

    if(mat.find()) System.out.println("subsequence found");
    else System.out.println("No Match");
  }
}
```

Example program of demonstrating pattern matching using **find()**

The output is shown here:

```
Looking for Java in Java SE.
subsequence found
```


Constructing Regular Expressions (Patterns)

- It is to be noted that a regular expression can be comprised of **normal characters, character classes (sets of characters), wildcard characters, quantifiers and metacharacters.**

Normal Characters:

- **A normal character** is matched as-is. Thus, if a pattern consists of **"xy"**, then the only input sequence that will match it is "xy".
- Characters such as newline and tab are specified using the standard escape sequences, which begin with \. For example, a newline is specified by **\n**.

Note: In the language of regular expressions, a normal character is also called a ***literal***.

Character Classes and Wildcard

Character Classes:
A **character class** is a set of characters. **A character class is specified by putting the characters in the class between brackets**. For example, the class [wxyz] matches w, x, y, or z. Also, [a-zA-Z] matches a through z or A through Z, inclusive (range). For example, to match the lowercase characters a through z, use **[a-z]**.

- To specify an inverted set, precede the characters with a ^. For example, [^wxyz] matches any character except w, x, y, or z.
- We can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use [1-9].

Wildcard character :
The wildcard character is the . (dot) and it matches any character. Thus, a pattern that consists of "." will match any one of these input sequences: "A", "a", "x", and so on.

Example program of using Regular Expression

```
// Three ways of using regular expression
import java.util.regex.*;
public class RegexExample1{
public static void main(String args[]){
//1st way
Pattern p = Pattern.compile(".s");//. represents single character
Matcher m = p.matcher("as");
boolean b = m.matches();

//2nd way
boolean b2=Pattern.compile(".s").matcher("as").matches();

//3rd way
boolean b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);
}}
```


Example program using character classes and group()

```
import java.util.regex.*;

class RegExpr7 {
  public static void main(String args[]) {
    // Match lowercase words.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output is shown here:

```
Match: this
Match: is
Match: a
Match: test
```


Quantifiers

- A **quantifier determines how many times** an expression is matched. The basic quantifiers are shown in the below table:

+	Match one or more.
*	Match zero or more.
?	Match zero or one.

For example, "x+" will match "x", "xx", and "xxx", and so on.

Regular Expression	Description
x?	x occurs once or not at all
X*	X occurs zero or more times
X+	X occurs one or more times
X{n}	X occurs exactly n times.
X{n,}	X occurs at least n or more than n times.
X{n,m}	X occurs between n and m occurrences.

Example program using Quantifiers

```
import java.util.regex.*;
class RegExpr4{
public static void main(String args[]){
Pattern pat=Pattern.compile("W+");
Matcher mat=pat.matcher("W WW WWW");
while(mat.find())
System.out.println("Match: "+ mat.group());
}
}
```

The output from the program is shown here:

```
Match: W
Match: WW
Match: WWW
```

Note: "W+" here matches any arbitrary long sequence of **W**s.

Example program using Wildcard and Quantifier with ?

```
import java.util.regex.*;

class RegExpr6 {
  public static void main(String args[]) {
    // Use reluctant matching behavior.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");
    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

Output:

```
Match: extend
Match: end
```

Note: The pattern "**e.+?d**" will match the shortest sequence that begins with e and ends with d.

Example program using Wildcard and Quantifier

```
import java.util.regex.*;

class RegExpr5 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("e.+d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

Output:

```
Match: extend cup end
```

Note: The pattern "e.+d" will match the longest sequence that begins with e and ends with d. This is called greedy behavior.

Example Program Using replaceAll()

```
import java.util.regex.*;

class RegExpr8 {
  public static void main(String args[]) {
    String str = "Jon Jonathan Frank Ken Todd";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);

    System.out.println("Original sequence: " + str);

    str = mat.replaceAll("Eric ");

    System.out.println("Modified sequence: " + str);

  }
}
```

The output is shown here:

```
Original sequence: Jon Jonathan Frank Ken Todd
Modified sequence: Eric Eric Frank Ken Todd
```

Note: replaceAll() replaces all occurrences of sequences that begin with "Jon" with "Eric".

Example program to count number of occurrences of Hello

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexExample {
  public static void main(String args[]) {
    String regex = "Hello";
    String input = "Hello how are you Hello Hello";
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    int count = 0;
    while(m.find()) {
      count++;
    }
    System.out.println("Number of occurreneces of Hello: "+count);
  }
}
```

Output:
Number of occurrences of Hello: 3

Backslash (\)

- If your expression needs to search for one of the special characters, you can use a backslash (\ ) to escape them.
- In Java, backslashes in strings need to be escaped themselves, so **two backslashes are needed to escape special characters.**
- For example**, to search for one question mark,** you can use the following expression: **"\\?"**

Predefined Character Classes

Character	Matches
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
\b	A word boundary
\B	A non word boundary
\A	The beginning of the input or line
\z	The end of the input or line.

Example program using a Predefined Class

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Main {
  public static void main(String args[]) {
    Pattern p = Pattern.compile("Java\\d");

    String candidate = "Java 4";
    Matcher m = p.matcher(candidate);

    if (m != null)
      System.out.println(m.find());
  }
}
```

Output:
true

Note: \d to match all digits.
\\d is used in the string to escape the \.

Another Example program using a Predefined Class (\W)

```
String regex = "\\W!";
    String input = "Hello how are you welcome !";
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    int count = 0;
    while(m.find()) {
      count++;
    }
    System.out.println("Number of matches: "+count);
```

Output:
Number of matches: 1

Metacharacters

Metacharacters are characters which are having special meanings in Java regular expression. This will not be counted as a regular character by the regex engine.
Following metacharacters are suppored in Java Regular expressions.
<([{\^-=$!|]})?*+.>

Metacharacter	Description
|	Find a match for any one of the patterns separated by | as in: cat|dog|fish
^	Find a match as the beginning of a string or Line as in:^Hello
$	Find a match as the end of a string or Line as in:Hello$
[0-9&&[4-8]]	Intersection of two ranges (4, 5, 6, 7, or 8)
[a-z&&[^aeiou]]	All lowercase letters minus vowels
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)

Regular Expressions

Example: Create a regular expression that accepts mobile numbers starting with 7 or 8 or 9 only.

```
import java.util.regex.*;
class RegexExample{
public static void main(String args[]){
System.out.println("by character classes and quantifiers ...");
System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true
System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
System.out.println("by metacharacters ...");
System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)
}}
```


Example Code Segment using Metacharacter (^)

```
String content =    "begin here to start, and go there to end\n" +
                    "come here to begin, and end there to finish\n" +
                    "begin here to start, and go there to end";
                     String regex    =   "^begin";
//OR
//String regex = "^begin";

Pattern pattern =   Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
Matcher matcher =   pattern.matcher(content);
while (matcher.find())
{
   System.out.print("Start index: " + matcher.start());
   System.out.print(" End index: " + matcher.end() + " ");
   System.out.println(matcher.group());
}
```

Note: We can use "\\A" as well in place of ^ in regular expression.

Output:
Start index: 0 End index: 5 begin

Example Code Segment using Metacharacter ($)

```
String content =    "begin here to start, and go there to end\n" +
                    "come here to begin, and end there to finish\n" +
                    "begin here to start, and go there to end";

String regex    =   "end$";
//String regex    =   "end\\z";

Pattern pattern =   Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
Matcher matcher =   pattern.matcher(content);
while (matcher.find())
{
   System.out.print("Start index: " + matcher.start());
   System.out.print(" End index: " + matcher.end() + " ");
   System.out.println(matcher.group());
}
```

Output:

Start index: 122 End index: 125 end

Note: We can use "\\z" as well in place of $ in regular expression.


```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
   public static void main( String args[] ) {
      // String to be scanned to find the pattern.
      String line = "This order was placed for QT3000! OK?";
      String pattern = "(.*)(\\d+)(.*)";

      // Create a Pattern object
      Pattern r = Pattern.compile(pattern);

      // Now create matcher object.
      Matcher m = r.matcher(line);

      if (m.find( )) {
         System.out.println("Found  value: " + m.group(0) );
         System.out.println("Found  value: " + m.group(1) );
         System.out.println("Found  value: " + m.group(2) );
      } else {
         System.out.println("NO  MATCH");
      }
   }
}
```

Output:

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

Note: group(0) always represents the entire expression. This group is not included in the total reported by groupCount.

m.group(3) would return **! OK?**

Capturing Groups and Backreferences

-
- You can create a group using ().
- For example, In the regular expression ((A)(B(C))), there are four such groups:
  1. ((A)(B(C)))
  2. (A)
  3. (B(C))
  4. (C)
-

- The portion of input String that matches the capturing group is saved into memory and can be recalled using Backreference.

Backreference:

- Backreference is a way to repeat a capturing group.
- backreference can be used inside a regular expression by inlining it's group number preceded by a single backslash.
- **Backreference**

Code segment with Backreferences

```
System.out.println(Pattern.matches("(\\w\\d)\\1", "a2a2")); // returns true
System.out.println(Pattern.matches("(\\w\\d)\\1", "a2b2")); //returns false
System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B2AB")); //returns true
System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B3AB")); //returns false
```

Difference with and without backreference:

Pattern.compile("([A-Za-z])[0-9]**\\1**").matcher("**a9a**").find(); //**matches: 'a9a'**

Pattern.compile("[A-Za-z][0-9][A-Za-z]").matcher("a9b").find(); //**matches: 'a9b'**

Ref: https://www.logicbig.com/tutorials/core-java-tutorial/java-regular-expressions/regex-backreferences.html#:~:text=Backreference%20is%20a%20way%20to,%5D)%5B0%2D9%5D%5C1%20.

Testing Prime Numbers using Regular Expressions

```
public static boolean prime(int n) {
  return !new String(new char[n]).matches(".?|(..+?)\\1+");
}
prime(5);// checking if 5 is a prime or not.
```

Output:
true

```
prime(6) // checking if 6 is a prime or not.
```
false

Ref: https://iluxonchik.github.io/regular-expression-check-if-number-is-prime/

Conclusion

- Strings
- StringTokenizer
- RegularExpressions

Agenda

- Arrays
- Multi-dimensional Arrays

Arrays

- Arrays are objects that help us organize large amounts of information.
- An array stores multiple values of the same type (the *element type*).
- The element type can be a primitive type or an object reference.
- Therefore, we can create an array of integers, or an array of characters, or an array of String objects, etc.
- In Java, the array itself is an object.
- A particular value in an array is referenced using the array name followed by the index in brackets.

  For example:

  Scores[2]; //(refers to the 3rd value in the array)

- That expression represents a place to store a single integer and can be used wherever an integer variable can be used.
- The values held in an array are called *array elements*.
- The name of the array is a object reference variable, and the array itself must be instantiated.

Bounds Checking

- Once an array is created, it has a fixed size
- An index used in an array reference must specify a valid element
- That is, the index value must be in bounds (0 to N-1)
- The Java interpreter throws an **`ArrayIndexOutOfBoundsException`** if an array index is out of bounds
- This is called *automatic bounds checking*
- For example, if the array `codes` can hold 100 values, it can be indexed using only the numbers 0 to 99
- If `count` has the value 100, then the following reference will cause an exception to be thrown:

```
System.out.println (codes[count]);
```


Declaring Arrays

The scores array could be declared as follows:

int[] scores = new int[10];

The type of the variable scores is int[] (an array of integers). The reference variable scores is set to a new array object that can hold 10 integers.

Note that the type of the array does not specify its size, but each object of that type has a specific size.

Some examples of array declarations:

```
String[] array = new String[10];   // int size = array.length;
        or
String array[]=new String[10];

float[] prices = new float[500];

boolean[] flags;

flags = new boolean[20];

char[] codes = new char[1750];
```


Bounds Checking (contd..)

The brackets of the array type can be associated with the element type or with the name of the array.

Therefore the following declarations are equivalent:

float[] prices;

float prices[];

- **The first format generally is more readable.**

- **An *initializer list* can be used to instantiate and initialize an array in one step**
- **The values are enclosed within braces and separated by commas**
- **Examples:**

```
int[] units = {147, 323, 89, 933, 540,
        269, 97, 114, 298, 476};

char[] letterGrades = {'A', 'B', 'C', 'D', 'F'};
```

Array as Parameters

- An entire array can be passed as a parameter to a method
- Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other
- Changing an array element within the method changes the original
- An array element can be passed to a method as well, and follows the parameter passing rules of that element's type

```
public class ArrayParam2
 {
   public static void updateParam( double[] x )
   {
          x[0] = 9999;  // Update one of the array element
   }

   public static void main(String[] args)
   {
          double[] a = { 2.3, 3.4 , 4.5 };

          System.out.println("Array before calling updateParam:");

          for (int i = 0; i < a.length; i++)
            System.out.println( a[i] );

          updateParam( a );      // Call updateParam
          System.out.println("Array AFTER calling updateParam:");

          for (int i = 0; i < a.length; i++)
            System.out.println( a[i] );
   }
 }
```


A *two-dimensional array* can be thought of as a table of elements, with rows and columns.

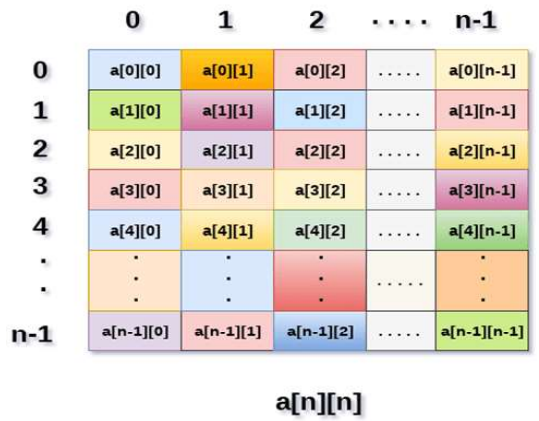
The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
int arr[max_rows][max_columns];
```

We can assign each cell of a 2D array to 0 by using the following code:

```
for ( int i=0; i<n ;i++)
{
  for (int j=0; j<n; j++)
  {
    a[i][j] = 0;
  }
}
```

Two-Dimensional Arrays



Arrays of objects

```
class JavaObjectArray{
public static void main(String args[]){
Account obj[] = new Account[1] ;
obj[0] = new Account();
obj[0].setData(1,2);
System.out.println("For  Array Element 0");
obj[0].showData();
}
}
class Account{
int a;
int b;
public void setData(int c,int d){
a=c;
b=d;
}
public void showData(){
System.out.println("Value  of a ="+a);
System.out.println("Value  of b ="+b);
}
}
```

- An object represents a single record in memory, and thus for multiple records, an array of objects must be created. It must be noted, that the arrays can hold only references to the objects, and not the objects themselves.
- The elements of an array can be object references
- The following declaration reserves space to store 25 references to String objects:

  String[] words = new String[25];

- It does NOT create the String objects themselves
- Each object stored in an array must be instantiated separately

To be precise, a two-dimensional array in Java is an array of arrays.

A two-dimensional array can be declared by specifying the size of each dimension separately:

- `int[][] scores = new int[12][50];`

A two-dimensional array element is referenced using two index values.

- `value = scores[3][6]`

The array stored in one row or column can be specified using one index.

Two-Dimensional Array Program

```
import java.util.Scanner;
Public class TwoDArray {
Public static void main(String[] args) {
   int[][] arr = newint[3][3];
   Scanner sc = new Scanner(System.in);
   for (inti =0;i<3;i++)
   {
      for(intj=0;j<3;j++)
      {
         System.out.print("Enter Element");
         arr[i][j]=sc.nextInt();
         System.out.println();
      }
   }
```

```
   System.out.println("Printing  Elements...");
    for(inti=0;i<3;i++)
    {
       System.out.println();
       for(intj=0;j<3;j++)
       {
          System.out.print(arr[i][j]+"\t");
       }
    }
}
}
```


Agenda

- Exceptions
- User-defined exceptions

Syntax Errors, Runtime Errors, and Logic Errors

There are three categories of errors:
syntax errors, runtime errors, and logic errors.

- ***Syntax errors*** arise because the rules of the language have not been followed. They are detected by the compiler.
- ***Runtime errors*** occur while the program is running if the environment detects an operation that is impossible to carry out.
- ***Logic errors*** occur when a program doesn't perform the way it was intended to.

Def (Exception)

An ***exception*** is an abnormal condition that arises in a code sequence at run time. An exception is also called as a **Run-time Error**.

- In computer languages that do not support exception handling, errors must be checked and handled manually— typically through the use of error codes, and so on.
- Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Uncaught Exceptions

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.
- This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- As we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Runtime Errors

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

    // Display the result
    System.out.println(
      "The number entered is " + number);
  }
}
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

```
Terminated.
```


Why should exceptions are to be handled?

- It allows us to fix the error.
- It prevents the program from automatically getting terminated during execution.

Handling Exceptions

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- **Java exception handling** is managed via five keywords: **try, catch, throw, throws, and finally**.

General Form of Exception Handling Block

```
try {
   // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
   // exception handler for ExceptionType2
}
// ...
finally {
   // block of code to be executed after try block ends
}
```


Handling Exceptions (Cont…)

- Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the try block, it is thrown.
- Your code can catch this exception (using **catch**) and handle it in some rational manner.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a try block completes is put in a **finally** block.

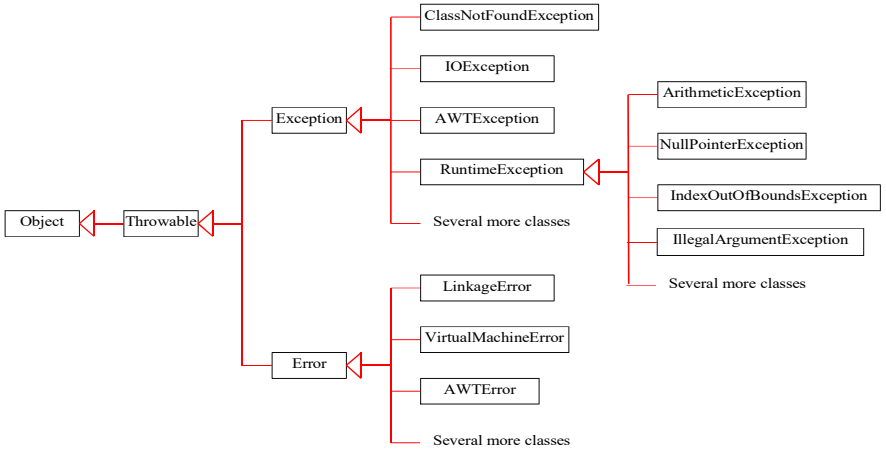
Note: System generated exceptions are automatically thrown by the Java run-time system.

Catch Runtime Errors

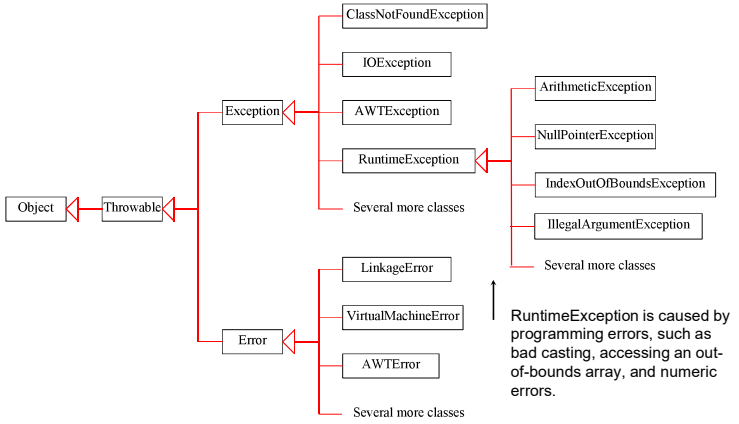
```
1  import java.util.*;
2
3  public class HandleExceptionDemo {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = scanner.nextInt();
12
13       // Display the result
14       System.out.println(
15         "The number entered is " + number);
16
17       continueInput = false;
18     }
19     catch (InputMismatchException ex) {
20       System.out.println("Try again. (" +
21         "Incorrect input: an integer is required)");
22       scanner.nextLine(); // discard input
23     }
24   } while (continueInput);
25 }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

Exception Classes

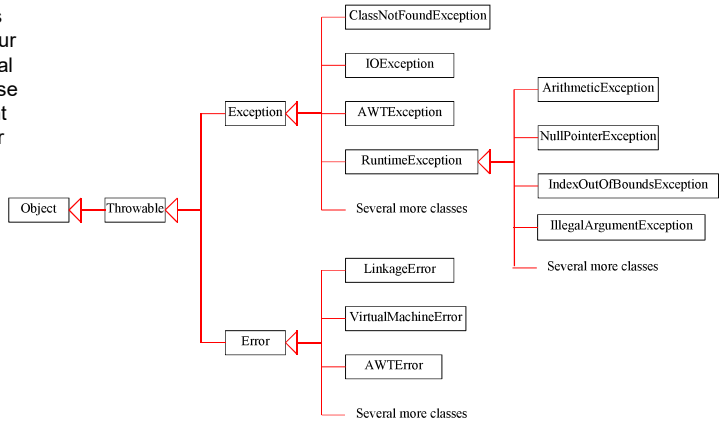


Runtime Exceptions



Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



try block and a catch clause that processes the ArithmeticException

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}

System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

Another example program on **ArithmeticException**

Note: The goal of the most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.


```
// Demonstrate multiple catch statements.
class MultipleCatches {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Multiple catch clauses example

Output

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
After try/catch blocks.
```


Multiple catch clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this situation, we can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- In multiple catch blocks, **after one catch statement executes, the others are bypassed**, and execution continues after the **try/catch** block.

Important Note:

- When we use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is essential because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. **This unreachable code causes an error, i.e., compiler error.**

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
     } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Example program with multiple catch blocks that has **unreachable catch block**. This causes compiler error.

Note: If we compile the above code, it displays an error message stating that the second **catch** statement is **unreachable** because the exception has already been caught.


```
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

       System.out.println("a = " + a);

       try { // nested try block
         /* If one command-line arg is used,
            then a divide-by-zero exception
            will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

         /* If two command-line args are used,
            then generate an out-of-bounds exception. */
        if(a==2) {
           int c[] = { 1 };
           c[42] = 99; // generate an out-of-bounds exception
         }
       } catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Array index out-of-bounds: " + e);
       }

     } catch(ArithmeticException e) {
       System.out.println("Divide by 0: " + e);
     }
  }
}
```

Example of nested try

Output

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:
  Index 42 out of bounds for length 1
```


Nested try Statements

- The **try** statement can be nested, i.e., a try statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

throw

- It is possible for our programs to throw an exception explicitly, using the **throw** statement.
- The general form of **throw** is shown below:

  throw ThrowableInstance

Here, **ThrowableInstance** must be an object of type **Throwable or** a subclass of **Throwable.**

Note: Primitive types, such as **int** or **char,** as well as non-**Throwable** classes such as **String** and **Object,** cannot be used as exceptions.

- There are two ways we can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

  - The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.

  - The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

Example program demonstration on usage of **throw** keyword

Output

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```


throws

- **throws** keyword is used when a method is capable of causing an exception that it does not handle.
- In such cases, the method must specify this behavior so that callers of the method can guard themselves against that exception. It can be done by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw.
- All other exceptions that a method can **throw** must be declared in the **throws** clause. If they are not, a compiler error will result.

The general form of a method that includes a throws clause:

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

Note: exception-list is a comma-separated list of exceptions that a method can throw.

Uses of throw

- **throw** clause can be used in software testing to test whether a program is handling all the exceptions as claimed by the programmer.
- **throw** can also be used to throw our the user-defined exceptions as well.

Example program with **throws**

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not.
- It is useful for closing files and freeing up any other resources that might have been allocated at the beginning with the intent to dispose them before returning.

General form of finally block:

```
try {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```


Question:

What is the output of following Java program?

```
class Test
{
public static void main (String[] args)
{
try
{
int a = 0; System.out.println ("a = " + a);
int b = 20 / a;
System.out.println ("b = " + b);
}
catch(ArithmeticException e)
{
System.out.println ("Divide by zero error");
}
finally
{
System.out.println ("inside the finally block");
}
}
}
```

(A) Compile error (B) Divide by zero error
(C) a = 0 Divide by zero error inside the finally block (D) a = 0 (E) inside the finally block

finally can be used:

1. **When an exception does not rise:** The program runs fine without throwing any exception and finally block execute after the try block.
2. **When the exception rises and handled by the catch block:** In this case, the program throws an exception but handled by the catch block, and finally block executes after the catch block.
3. **When exception rise and not handled by the catch block:** In this case, the program throws an exception but not handled by catch so finally block execute after the try block and after the execution of finally block program terminate abnormally, But finally block execute fine.

Solution :

Answer: (C)

Explanation: On division of 20 by 0, divide by zero exception occurs and control goes inside the catch block. Also, the finally block is always executed whether an exception occurs or not.

When exception does not occur

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

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

Output

```
5
finally block is always executed
Rest of the code…
```


User-defined Exceptions

- **Usage Context:** To create our own exception types to handle situations specific to our applications.
  - We can create our own exception class by extending **Exception** or any **subclass of Exception**.
  - The user-defined exceptions don't need to actually implement anything (it allows us to use them as exceptions in some real-world scenarios).
  - It is to be noted that the Exception class does not define any methods of its own, but inherit those methods provided by **Throwable**.
  - Thus, all exceptions, including those that we create, have the methods defined by **Throwable.**

Java's Built-in Exceptions

- The standard package **java.lang** defines several exception classes.
- Most of these exceptions are subclasses of the standard type **RuntimeException,** which was already discussed.

Java's Built-in Exceptions can be classified further into:

- **Unchecked Exceptions:** Unchecked exceptions are not checked at compile-time, but they are checked at runtime. The compiler does not check to see if a method handles or throws these exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
- **Checked Exceptions:** These exceptions are checked at compile time. The classes that directly inherit the **Throwable** class except **RuntimeException and Error** are known as checked exceptions. For example, ClassNotFoundException, IOException, SQLException, etc.

Few methods of **Throwable**

Method	Description
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

The following example demonstrates how to create user-defined exceptions

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}
```


Question

What is the output of the following code?

```
class Base extends Exception {}
class Derived extends Base  {}
public class Main
{
        public static void main(String args[])
        {
                try
                {
                   // Some code
                   throw new Derived();
        }
        catch(Base b)
                {
                   System.out.println("Caught base class exception");
        }
        catch(Derived d)
                {
           System.out.println("Caught derived class exception");
        }
         }
}
```

a. Caught base class exception
b. Caught derived class exception
c. Compiler Error because derived is not throwable
d. Compiler Error because base class exception is caught before derived class

Solution: d


```
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```


Agenda

- Object class
- Type Wrappers
- Java Collection Framework

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled. (Deprecated by JDK 9.)
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( )	Waits on another thread of execution.
void	
void	

Note: getClass(), notify(), notifyAll(), and wait() are declared as **final** and we can override the other methods.
Classes that implement the **Cloneable** interface can only be cloned.
toString() method is automatically called when an object is output using **println()**.

Object Class

- Object class is a special class defined by Java.
- All other classes are subclasses of **Object** (Object is a superclass of all other classes).
- With this, a reference variable of type **Object** can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.
- Object methods are available in every object.

Type Wrappers

- Java uses primitive types such as **int, double, etc.,** to hold the basic data types supported by the language. These primitive types are used for the sake of performance.
- Primitive types are not part of the object hierarchy and they do not inherit **Object.**

There are times when we need an object representation:
•Wrapper Class will **convert primitive data types into objects**. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are **passed by value**).
•To perform operations on objects: Standard data structures implemented by the Java operate on objects, which means we can't use these data structures to store primitive types.
•The object is needed to support **synchronization** in **multithreading**.

To handle these situations, Java provides **Type Wrappers,** which are classes that encapsulate a primitive type within an object.

The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character,** and **Boolean.** These classes offer us a set of methods that allow us to full integrate primitives types into Java's object hierarchy.

Character

- **Character** is a wrapper around a **char.**
- The constructor for **Character** is

  Character(char ch)

  Here, **ch** specifies the character that will be wrapped by the **Character** object being created.
- To obtain a **char** value contained in a **Character** object, call charValue(), which is shown below:

  char charValue()

  It returns the encapsulated character.

The Numeric Type Wrappers

- **Byte, Short, Integer, Long, Float,** and **Double** are the most commonly used type wrappers.
- These number type wrappers inherit the abstract class **Number**.
- **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown below:

  byte byteValue()
  double doubleValue()
  float floatValue()
  int intValue()
  long longValue()
  short shortValue()
- For example, **doubleValue()** returns the value of an object as a **double**, **floatValue()** returns the value as a **float,** and so on.

Boolean

- Boolean is a wrapper around **boolean** values.
- It defines these constructors:

  (i) **Boolean(Boolean boolValue)**

  (ii) **Boolean(String boolString)**

In the first one, boolValue must be either **true** or **false.** In the second one, If boolString contains the string **"true"** (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

- To obtain a **boolean** value from Boolean object, use **boolValue()**, which is defined below:

  Boolean booleanValue()

  It returns the **boolean** equivalent of the invoking object.

Numeric Type Wrappers (Cont…)

- All the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.
- From JDK9, **valueOf( )** method can be used to obtain a wrapper object
- The **valueOf( )** method is a static member of all of the numeric wrapper classes and all numeric classes support forms that convert a numeric value or a string into an object.
- For example, two of the forms supported by **Integer**:

  static Integer valueOf(int *val*)

  static Integer valueOf(String *valStr*) throws NumberFormatException.

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form.

- Each returns an **Integer** object that wraps the specified value. Here is an example:

Integer iOb = Integer.valueOf(100);

Numeric Type Wrappers (Cont…)

- If str does not contain a valid numeric value, then **NumberFormatException** is thrown.
- All these wrappers override **toString()**, i.e., returns the human readable form of the value contained within the wrapper. This allows us to output the value by passing a type wrapper object to **println().**

Autoboxing and Auto-unboxing

Autoboxing: It is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

- For example, to construct an **Integer** object that has the value 100:

Integer iOb = 100; // autobox an int

Auto-unboxing: Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue()**.

- To unbox an object, simply assign that object reference to a primitive-type variable.
- For example, to unbox **iOb**, you can use this line:

int i = iOb; // auto-unbox

Note: In general, we should restrict use of the type wrappers to only those cases in which an object representation of a primitive type is required.


```
// Demonstrate a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = Integer.valueOf(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```



```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);  // displays 100 100
  }
}
```

```
class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
```

Output

Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134

Example program dealing with group of objects using Array

```
import java.util.*;
class Employee{
//intance variables
int id;
String name;
Employee(int i, String n)
{
id=i;
name=n;
}
//a method to display data
void displayData()
{
System.out.println(id+"\t"+name);
}
}
```

```
class Group{
public static void main(String args[]){
Scanner sc=new Scanner(System.in);
//create Employee type array  with size 5
Employee arr[]=new Employee[5];
for(int i=0;i<5;i++){
System.out.println("Enter  id: ");
int id=Integer.parseInt(sc.next());
System.out.println("Enter  name: ");
String name=sc.next();
arr[i]=new Employee(id,name);
}
System.out.println("The  employee data is");
for(int i=0;i<5;i++){
arr[i].displayData();
}
}
}
```


How can we handle a group of objects?

Can we use an array to store a group of objects?
Yes, an array can be used to store a group of objects.

Issues when working with Arrays

- We cannot store different class objects into the same array because the array can store only one data type of elements.
- Adding the objects at the end of an array is easy. But, inserting and deleting the elements in the middle of the arrays difficult. In this case, we have to re-arrange all the elements of the array.
- Retrieving the elements from an array is easy but after retrieving the elements, if we want to process them, then there are no methods available to carry out this.

Due to the above reasons, programmers want a better mechanism to store a group of objects. To meet this need**, we have a Java Collection framework.**

Java Collections

- Java collection framework standardizes the way in which groups of objects are handled by our programs.
- It is aimed to meet the following goals:
  - **High-performance:** The implementations for fundamental collections (Dynamic arrays, linked lists, trees and hash tables) are highly efficient.
  - To allow **different types of collections** to work in a similar manner and with a high degree of interoperability.
  - Extending a collection needs to be easy.

- To achieve these goals, we have the java Collections Framework.

What is Java Collection Framework?

- A collection framework is a class library to handle groups of objects. The collection framework is implemented in **java.util** package.

•It has a set of **classes and interfaces (Iterators)** to facilitate us to perform searching, sorting, insertion, manipulation, and deletion on collection.
•A collection represents a group of objects, known as its *elements*.
•Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

Collection Interface

- The **Collection** interface is the foundation upon which the **Collections Framework is built** because it must be implemented by any class that defines a collection.
- Collection is generic interface that has this declaration:
  interface Collection<E>
  Here, E specifies the type of objects that the collection will hold.

- Collection extends Iterable interface. It means all collection objects can be cycled through by use of the for-each style **for** loop.

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

int size( )	Returns the number of elements held in the invoking collection.
	Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elemen were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.

Method	Description
<T> T[ ] toArray(T *array*[ ])	Returns an array of the elements from the invoking collection. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not compatibl with the array type.

default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtaine from the *opToApply* function.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within th invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*.
	list changes and returns **false** otherwise.
static <E> List<E> copyOf(Collection<? extends E> *from*)	Returns a list that contains the same elements as that specified by *from*. The returned list is unmodifiable. Null values are not allowed.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.

List interface

- The **List** interface extends **Collection** and declares the behavior of the collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using zero-based index.
- A list may contain duplicate elements.
- List declaration:
  interface List <E>
  Here, E specifies the type of objects that the list will hold.
- In addition to the methods defined by **Collection**, List defines of some of its own, which are explained in the next slide.

Set Interface

- The **Set** interface defines a set.
- It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements.
- The **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.
- **Set** is a generic interface that has this declaration:
  interface Set<E>
  Here, **E** specifies the type of objects that the set will hold.

The SortedSet Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.
- **SortedSet** is a generic interface that has this declaration:
  interface SortedSet<E>

Here, **E** specifies the type of objects that the set will hold.

- In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized in the next slide.

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

The ArrayList Class

- The **ArrayList** class extends **AbstractList** and implements the **List** interface.
- **ArrayList** is a generic class that has this declaration:
  class ArrayList<E>
  Here, **E** specifies the type of objects that the list will hold.
- **ArrayList** supports dynamic arrays that can grow as needed.
- an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size.

Note: Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

Constructors of ArrayList

ArrayList has the constructors shown here:

1. **ArrayList( )** // builds an empty array list.
2. ArrayList(Collection<? extends E> *c*)// builds an array list that is initialized with the elements of the collection c.
3. ArrayList(int *capacity*)// builds an array list that has the specified initial *capacity*.

The capacity is the size of the underlying array that is used to store the elements.
The capacity grows automatically as elements are added to an array list.


```
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

```
rgs[]) {
    // Create an array list.
    ArrayList<String> al = new ArrayList<String>();

    System.out.println("Initial size of al: " +
                       al.size());

    // Add elements to the array list.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");

    System.out.println("Size of al after additions: " +
                       al.size());

    // Display the array list.
    System.out.println("Contents of al: " + al);

    // Remove elements from the array list.
    al.remove("F");
    al.remove(2);

    System.out.println("Size of al after deletions: " +
                       al.size());
```

Another Example Program to demonstrate ArrayList

Output

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

Example

```
cars.get(0);
```

Change an Item

To modify an element, use the set() method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the remove() method and refer to the index number:

Example

```
cars.remove(0);
```

Looping though an arraylist

```
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new
ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```


Important Note

- **ensureCapacity( )**.: This method can be used to increase the capacity of an **ArrayList** object manually by calling this method.

Note:
The signature for **ensureCapacity( )** is shown here:

void ensureCapacity(int *cap*)
Here, ***cap*** specifies the new minimum capacity of the collection

- **trimToSize( )**: It can be used to reduce the size of the array that underlies an **ArrayList** object so that it is precisely as large as the number of items that it is currently holding.

```
import java.util.ArrayList;
public class Demo1 {
   public static void main(String[] arg) throws Exception
   {
      try {

         // Creating object of ArrayList of String of
         // size = 3
         ArrayList<String> numbers
            = new ArrayList<String>(3);

         // adding element to Arrlist numbers
         numbers.add("10");
         numbers.add("20");
         numbers.add("30");

         // Print the ArrayList
         System.out.println("ArrayList: " + numbers);

         // using ensureCapacity() method to
         // increase the capacity of ArrayList
         // numbers to hold 500 elements.
         System.out.println(
            "Increasing the capacity of ArrayList numbers to store upto 500 elements.");
         numbers.ensureCapacity(500);

         System.out.println(
            "ArrayList numbers can now store upto 500 elements.");
      }

      catch (NullPointerException e){
         System.out.println("Exception thrown : " + e);
      }
   }
}
```

Example Program with **ensureCapacity()**

Output

```
ArrayList: [10, 20, 30]
Increasing the capacity of ArrayList numbers to store upto 500 elements.
ArrayList numbers can now store upto 500 elements.
```



```
Contents of al: [1, 2, 3, 4]
Sum is: 10
   class ArrayListToArray {
     public static void main(String args[]) {
       // Create an array list.
       ArrayList<Integer> al = new ArrayList<Integer>();

       // Add elements to the array list.
       al.add(1);
       al.add(2);
       al.add(3);
       al.add(4);

       System.out.println("Contents of al: " + al);

       // Get the array.
       Integer ia[] = new Integer[al.size()];
       ia = al.toArray(ia);

       int sum = 0;

       // Sum the array.
       for(int i : ia) sum += i;

       System.out.println("Sum is: " + sum);
     }
   }
```

Example Program to convert an ArrayList to an Array

Output:

Note: As collections store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type **int** to **add( )** without having to manually wrap them within an **Integer**


```
object[ ] toArray( )
<T> T[ ] toArray(T array[ ])
default <T> T[ ] toArray(IntFunction<T[ ]> arrayGen)
```

Obtaining an Array from an ArrayList

Several reasons for why we need to convert a Collection into array:
-To obtain faster processing times for certain operations
-To pass an array to a method that is not overloaded to accept a collection
-To integrate collection-based code with legacy code that does not understand collections

The below versions can be used to convert into array:

Note: The first returns an array of **Object**. The second and third forms return an array of elements that have the same type as **T.**

One More Example Program

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;
class Demo{
   public static void main(String[] args)
   {
      List<Integer> al = new ArrayList<Integer>();
      al.add(10);
      al.add(20);
      al.add(30);
      al.add(40);

      Object[] objects = al.toArray();

      // Printing array of objects
      for (Object obj : objects)
         System.out.print(obj + " ");
   }
}
```

Output
10 20 30 40

Accessing a Collection via an Iterator

- Iterator helps us to cycle through the elements of a collection so that we can access one element at a time.
- **Iterator** is an interface that has methods that enables us to cycle through a collection for obtaining or removing elements
- Iterator declaration:

  interface Iterator<E>

  Here, E specifies the type of objects being iterated.

How to access Iterator?

- Before we can access a collection through iterator, we must obtain it. Each of the collection classes provides an **iterator()** method to return an iterator to the start of the collection.
- By using this iterator object, we can access each element in the collection, one element at a time.

Sequence of steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.


```
import java.util.*;
class Demo{
   public static void main(String[] args)
   {
      ArrayList<Integer> al = new ArrayList<Integer>();
      al.add(10);
      al.add(20);
      al.add(30);
      al.add(40);
      Iterator<Integer> itr=al.iterator();
       while(itr.hasNext()){
              System.out.println(itr.next());
              }
   }
}
```

Program to demonstrate **Iterator** with the help of iterator()

Output:

10
20
30
40

ListIterator

- **ListIterator** extends **Iterator** to allow **bidirectional traversal** of a list, and the modification of elements.
- The declaration of ListIterator is shown below:

  interface ListIterator <E>

  Here, E specifies the type of objects being iterated.


```
import java.util.*;
class Demo3{
   public static void main(String[] args)
   {
      ArrayList<String> al = new ArrayList<String>();
      al.add("A");
      al.add("B");
      al.add("C");
      al.add("D");
      ListIterator<String> itr=al.listIterator();
            System.out.println("Elements  in forward direction");
      while(itr.hasNext()){
            String element=itr.next();
            System.out.print(element+" ");//displaying the list in forward direction.
            itr.set(element+"+");
            }
            System.out.println("\nElements  in reverse direction");
      while(itr.hasPrevious()){
            System.out.print(itr.previous()+" ");//displaying the list in reverse direction
            }
   }
}
```

Program to demonstrate **ListIterator** with the help of **listIterator()**

Output

Elements in forward direction
A B C D
Elements in reverse direction
A+ B+ C+ D+

boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

e element that will be


```
Contents of vals: 1 2 3 4 5
Sum of values: 15
```

```
tring args[]) {
    // Create an array list for integers.
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // Add values to the array list.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Contents of vals: ");
    for(int v : vals)
      System.out.print(v + " ");

    System.out.println();

    // Now, sum the values by using a for loop.
    int sum = 0;
    for(int v : vals)
      sum += v;

    System.out.println("Sum of values: " + sum);
  }
}
```

For-each

the contents of a collection or obtaining

For-each version of for loop cycle through a collection

Output:

HashSet Class

- **HashSet** extends **AbstractSet** and implements the **Set** interface.
- **HashSet** is a generic class that has this declaration:
  class HashSet<E>
  Here, E specifies the type of objects that the set will hold.
- It creates a collection that uses a hash table for storage.
- The hash table stores information by using a mechanism called **hashing**.
- In hashing, the information content of the a key is used to determine the unique value, called its **hash code**.
- The hash code is then used as the index at which the data associated with the key is stored.
- It is to be noted that the transformation of the **key (or element)** into its hash code is performed automatically (we never see the hash code itself and our code can't directly index the hash table).
- The advantage of hash is that the execution time of add(), contains(), remove() and size() would be constant even for large sets.


```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
class HashSetDemo {
  public static void main(String args[]) {
    // Create a hash set.
    HashSet<String> hs = new HashSet<String>();

    // Add elements to the hash set.
    hs.add("Beta");
    hs.add("Alpha");
    hs.add("Eta");
    hs.add("Gamma");
    hs.add("Epsilon");
    hs.add("Omega");

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

Program to demonstrate **HashSet**

Output:

Constructors of HashSet

1. **HashSet():** It constructs a default hash set.
2. **HashSet(Collection<? Extends E> c) :** It initializes the hash set by using the elements of c.
3. **HashSet(int capacity):** It initializes the capacity of the hash set to capacity. The default capacity is 16.
4. **HashSet(int capacity, float fillRatio):** The fourth form initializes both the capacity and the fill ratio (also called the load capacity) of the hash set from its arguments.
   - The fillRatio must be between 0.0 and 1.0 and it determines how full the hash set can be before it is resized upward.
   - When the number of elements is greater than the capacity of the hash set **multiplied** by its fill ratio, the hash set is expanded.

Note: For constructors that do not take a fill ratio, o.75 is used.

Few important points:

- HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.
- HashSet does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets.
- If you need sorted storage, then another collection, such as TreeSet, is better choice.

Map Interface

- The **Map** interface maps unique keys to values.
- A *key* is an object that you use to retrieve a value at a later time.
- Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.
- Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps can accept a **null** key and **null** values, others cannot.
- **Map** is generic and is declared as shown here:
  interface Map<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

- Methods of **Map** can be broken down into three groups:
  - querying
  - altering
  - obtaining different views

Few Map Methods

¨Here is a list of the Map methods:

– void clear( )

Removes all key/value pairs from the invoking map.

- boolean containsKey(Object k)

Returns true if the invoking map contains k as a key. Otherwise, returns false.-

- boolean containsValue(Object v)

Returns true if the map contains v as a value. Otherwise, returns false.

- Set entrySet( )

Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.

- boolean equals(Object obj)

Returns true if obj is a Map and contains the same entries. Otherwise, returns false.

The HashMap Class

- The **HashMap** class extends **AbstractMap** and implements the **Map** interface.
- It uses a hash table to store the map. This allows the execution time of **get( )** and **put( )** to remain constant even for large sets.
- **HashMap** is a generic class that has this declaration:

  class HashMap<K, V>

  Here, **K** specifies the type of keys, and **V** specifies the type of values.

Few Map Methods (Cont…)

–Object get(Object k)

Returns the value associated with the key k.

-Set keySet( )

Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

-Object put(Object k, Object v)

Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.

-Object remove(Object k)

Removes the entry whose key equals k.

-int size( )

Returns the number of key/value pairs in the map.

HashMap Constructors

1. **HashMap( ):** It constructs a default hash map.
2. **HashMap(Map<? extends K, ? extends V> *m*):** It initializes the hash map by using the elements of *m*.
3. **HashMap(int *capacity*):** It initializes the capacity of the hash map to *capacity*.
4. **HashMap(int *capacity*, float *fillRatio*):** It initializes both the capacity and fill ratio of the hash map by using its arguments.

Note:

- The meaning of capacity and fill ratio is the same as for **HashSet**, described in earlier slides.
- The default capacity is 16. The default fill ratio is 0.75.
- It does not add any methods of its own.
- It is to be noted that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

```
import java.util.*;
public class Demo4 {
public static void main(String[] args) {
    HashMap<String,Double> m1 = new
HashMap<String, Double>();
    m1.put("Zara", 8.4);
    m1.put("Mahnaz", 31.5);
    m1.put("Ayan", 12.6);
    m1.put("Daisy", 14.5);
    System.out.println(" Map Entires");
    System.out.println(m1);
  }
}
```

Example program to demonstrate HashMap

Output:

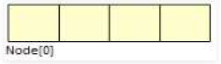
Map Entries
{Daisy=14.5, Ayan=12.6, Zara=8.4, Mahnaz=31.5}

Internal Structure of HashMap

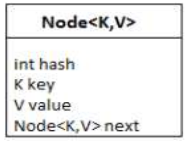
Internally HashMap contains an array of Node and a node is represented as a class which contains 4 fields:

1. int hash
2. K key
3. V value
4. Node next

It can be seen that node is containing a reference of its own object. So it's a linked list.
HashMap:



Node:




```
import java.util.*;
public class Demo4 {
public static void main(String[] args) {
    HashMap<Integer,String> m1 = new
HashMap<Integer, String>();
    m1.put(1, "Arjun");
    m1.put(2, "John");
    m1.put(3, "Raghu");
    m1.put(4, "Abdul");
    System.out.println(" Map Entires");
    System.out.println(m1);
  }
}
```

Another Example program to demonstrate HashMap

Output:

Map Entries
{1=Arjun, 2=John, 3=Raghu, 4=Abdul}

How Hashmap works?

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Aman", 19);
map.put("Sunny", 29);
map.put("Ritesh", 39);
```

When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

The Formula for calculating the index is:

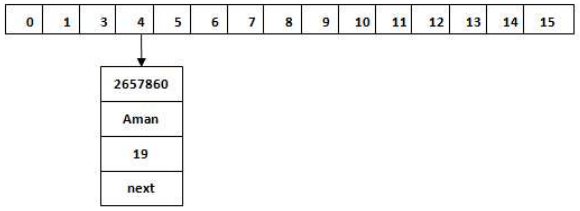
1. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

1. Index = 2657860 & (16-1) = 4

How Hashmap works?
With an example

The value 4 is the computed index value where the Key and value will store in HashMap.



What is the output?

```
import java.util.*;
public class Demo
{
 public static void main(String[] args)
 {
                    Map<Integer, Object> sampleMap = new TreeMap<Integer, Object>();
                    sampleMap.put(1, null);
                    sampleMap.put(5, "Hello");
                    sampleMap.put(3, 1);
                    sampleMap.put(2, null);
                    sampleMap.put(4, null);

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

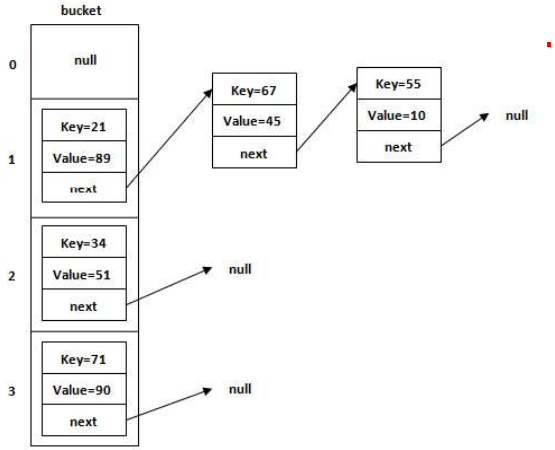



Figure: Allocation of nodes in Bucket

Output:

{1=null, 2=null,3=1,4=null,5=Hello}

What is the output?

```
import java.io.*;
import java.util.*;
class Demo1 {
    public static void main(String args[])
   {
      HashMap<Integer, String> hm1 = new HashMap<>(5, 0.75f);

      HashMap<Integer, String> hm2 = new HashMap<Integer, String>(3, 0.5f);
      hm1.put(1, "one");
      hm1.put(2, "two");
      hm1.put(3, "three");
      hm2.put(4, "four");
      hm2.put(5, "five");
      hm2.put(6, "six");
      hm2.put(6,"SIX");
      System.out.println("Mappings of HashMap hm1 are : " + hm1);
      System.out.println("Mapping of HashMap hm2 are : " + hm2);
   }
```


Initial capacity and Load Factor (or FillRatio) of HashMap

There are two factors which affect the performance of *HashMap*. They are **initial capacity and load factor**. We have to choose these two factors very carefully while constructing an *HashMap* object.

1. **Initial Capacity Of HashMap :**
   - The capacity of an *HashMap* is the number of buckets in the hash table.
   - The initial capacity is the capacity of an *HashMap* at the time of its creation.
   - The default initial capacity of the *HashMap* is 2^4 i.e 16.
   - The capacity of the *HashMap* is doubled each time it reaches the threshold,. i.e., the capacity is increased to 2^5=32, 2^6=64, 2^7=128
2. **Load Factor Of HashMap :**
   - Load factor is the measure which decides when to increase the capacity of the *HashMap*.
   - The default load factor is 0.75f.

Output

```
Mappings of HashMap hm1 are :  {1=one, 2=two, 3=three}
Mappings of HashMap hm2 are :  {4=four,  5=five, 6=SIX}
```


How Threshold Is Calculated?

The threshold of an *HashMap* is the product of current capacity and load factor.

Threshold = (Current Capacity) * (Load Factor)

For example, if the *HashMap* is created with initial capacity of 16 and load factor of 0.75f, then threshold will be,
Threshold = 16 * 0.75 = 12

When to use Hash Map and ArrayList?

ArrayList: Consumes low memory, but similar to LinkedList, and takes extra time to search when large.

HashMap: Can perform a jump to the value, making the search time constant for large maps. Consumes more memory and takes longer to find the value than small lists.

The beauty of **hashing** is that although you sacrifice some extra time searching for the element, the time taken does not grow with the size of the map. This is because the HashMap finds information by converting the element you are searching for, directly into the index, so it can make the jump.

```
import java.util.Hashtable;

public class HashMapArrayListExample {

   public static void main(String[] args) {
  ArrayList<String> arrlistobj = new
ArrayList<String>();
      arrlistobj.add("1. Alive is awesome");
       arrlistobj.add("2. Love yourself");
       System.out.println("ArrayList object output
:"+ arrlistobj);

      HashMap hashmapobj = new HashMap();
       hashmapobj.put("Alive is ", "awesome");
       hashmapobj.put("Love", "yourself");
       System.out.println("HashMap object output
:"+hashmapobj);

 }
}
```

Similarities between HashMap and ArrayList

1. Allows null : Both HashMap and ArrayList allows null. HashMap can have one null key and any number of null values.

2. Synchronized : Both HashMap and ArrayList are not synchronized. One needs to avoid using them in multithreading environment.

3. Traversal : Both HashMap and ArrayList can be traversed through iterator in java.

4. Iterator : Both HashMap and ArrayList classes iterator are fail-fast i.e they will throw ConcurrentModificationException as soon as they detect any structural change in ArrayList or HashMap.

Difference between Hash Map and ArrayList

	HashMap	ArrayList
Implementation	Map	List
Storing Objects	two objects key and value	store only one object
Duplicates	Unique key,duplicate values	permit duplicates
Ordering	Unordered	Maintains order
get() performance	Worst O(n) best O(1)	O(1)

Agenda

- Multithreading
- Synchronization

Process and Thread

- **Process (A program under execution is called a process) :**
  - *Heavy weight - Separate Address space.*
  - *Context Switching - More Expensive.*
  - *Inter Process Communication.*
- **Thread (A segment of a process) :**
  - *Lightweight - Share common Address space.*
  - *Context Switching - Less Expensive.*
  - *Inter Thread Communication.*

Single Tasking and Multi-tasking

Def(Task): A task means doing some calculation, processing, etc. A task involves execution of group of statements, for example executing a program.

Single Tasking:
- It involves execution of single task at a time.
- In this environment, only one task is given to the processor at a time.
- The problem with approach is that the processor sits idle most of the time waiting for the next task to execute.

Multi-tasking:
- In this environment, several tasks can be given to a processor at a time for execution. Here, the processor executes tasks one by one so quickly that we feel all the tasks are executed by processor at a time.
- It allows a user to perform more than one computer task simultaneously.
- Processor time would be used effectively as the processor will have at least one task to execute if one task is busy waiting for I/O activity or any other thing.
- **Two types of Multi-tasking:**
  - **i) Process-based Multi-tasking:** Several programs are executed at a time by the processor.
  - **ii) Thread-based Multi-tasking:** Several parts of the same program are executed at a time by the processor. Each part represent a separate block of code or function and is aimed to perform a separate task.

Main Thread

Importance of Main Thread:
- All Java Programs must have at least one thread called main thread created by JVM. Whenever the main method is invoked, main thread is created.
- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Important points:
- Main thread can be controlled through a thread object. To do so, we must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.
- Thread.sleep(1000) // This method can be used to pause the execution of current thread for specified time in milliseconds.
- The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. Hence, this need to be encapsulated in try/catch block.

Program to find the thread used by JVM

```
class Current{
public static void main(String args[]){
System.out.println("Let us find current thread");
Thread t=Thread.currentThread();
System.out.println("Current thread= "+t);
System.out.println("Current Thread name= "+t.getName());
System.out.println("Current Thread priority= "+t.getPriority());
}
}
```

Output

```
Let us find current thread
Current thread= Thread[main,5,main]
Current Thread name= main
Current Thread priority= 5
```


Uses of Threads

Some of the uses of threads are:

- Threads are used in server-side programs to serve the needs of multiple clients on network or internet.
- Threads are also used to create games and animation (moving objects from one place to another). Here, we have to perform more than one task at a time. For example, a flight may move from left to right and a machine gun should shoot it., releasing the bullets at the flight. These two tasks should happen simultaneously with the help of two threads.

Multithreaded Programming

- Java provides built-in support for *multithreaded programming.*
- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a ***thread,*** and each thread defines a separate path of execution. Thus, multithreading is a specialized form of **multitasking**.
- **Thread** follows independent path of execution within a program and it is smallest executable unit.
- Examples of real-world multithreaded applications: Word processor, Computer games, etc.

Note: In Java, support for creating threads and managing them are specified in **java.lang** package.

Creating threads and running

1. In every Java program **main thread** is available already, apart from the main thread, we can create our own threads in two ways :
   i) By extending **Thread** class, e.g., **class Myclass extends Thread**
   ii) By implementing **Runnable** interface, e.g., **class Myclass implements Runnable.**
2. Inside the class, we need to define run() method which is recognized and executed by a thread.

   Syntax : **public void run()**
   {
   // Statements
   }

   Note: Inside **run( )**, we can write the code that constitutes the new thread.
   It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
3. Create an object to Myclass, so that the run() method is available for execution.
   Myclass obj=new Myclass();

Creating threads and running (Cont...)

4. Create a thread and attach the thread to the object **obj**.
   Thread t=new Thread(obj);
   or
   Thread t=new Thread(obj, "Thread name");

5. Run the thread. For this purpose, we need to use start() method of Thread class.
   t .start();

Creating a thread using **Thread**

Ex :
```
import java.io.*;
import java.util.*;
class MyThread extends Thread{
   public void run(){
      for(int i=1;i<=10;i++){
         System.out.println("i value: "+i);
      }
   }
}
public class MyClass{
   public static void main(String args[]) {
      MyThread obj = new MyThread();
      Thread t = new Thread(obj);
      t.start();
   }
}
```


Thread Class Methods

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

setPriority(int priority_no): To set the priority of a thread.
Note that the priorities range from 1 to 10. Main thread priority number is by default 5.
setName("New name"): To set the new name to a created thread.

Output :

```
i value: 1
i value: 2
i value: 3
i value: 4
i value: 5
i value: 6
i value: 7
i value: 8
i value: 9
i value: 10
```

Creating Multiple Threads using **Runnable**

```
class MultithreadingDemo implements Runnable {
public void run()
{
try {
// Displaying the thread that is running
System.out.println(
"Thread " + Thread.currentThread().getId() + " is
running");
    }
catch (Exception e) {
// Throwing an exception
System.out.println("Exception is caught");
}
}
}
```

```
class Multithread {
        public static void main(String[] args)
        {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
Thread object= new Thread(new
MultithreadingDemo());
        object.start();
                }
        }
}
```


Creating Multiple Threads (two) using **Thread** class

```
class MyThread extends Thread
{
String message;
MyThread(String message)
{
this.message = message;
}
public void run()
{
try
{
for(int i=1; i<=5; i++)
{
System.out.println(message + "-" + i);
Thread.sleep(5000); //sleep for 5 seconds
}
}
catch(InterruptedException ie) { }
}
}
```

```
public class MultipleThreadDemo
{
 public static void main( String[] args)
 {
 MyThread t1 = new MyThread("One"),
MyThread t2 = new MyThread("Two");
System.out.println(t1);
System.out.println(t2);
t1.start();
t2.start();
 }
}
```


Output

```
Thread 15 is running
Thread 17 is running
Thread 16 is running
Thread 13 is running
Thread 18 is running
Thread 12 is running
Thread 14 is running
Thread 11 is running
```


Why does Java have two ways to create child threads?

- When a Java class that is aimed at creating a thread needs to inherit another class, then **implements Runnable** can be used
- When a Java class that is aimed at creating a thread but does not inherit other classes, then it can create a thread by extending **Thread** class or by implementing **Runnable** interface.

Terminating the Thread

- It is to be noted that a thread will terminate automatically when it comes out of **run()** method.
- We can terminate the thread on our own by using the sequence of below steps:
  - Create a boolean type variable and initialize it to **false**.
    boolean stop=false;
  - Now, assume that we want to terminate the thread when the user presses <Enter> key. So, when the user presses that button, make the **boolean** type variable as **true**.
    stop=true;
  - Check this variable in run() method and when it is true, make the thread return from the run() method

```
public void run(){
if(stop==true) return
}
```


Single Tasking using a Thread

```
class MyThread implements Runnable{
public void run()
{
task1();
task2();
task3();
}
void task1(){
System.out.println("This is task 1");
}
void task2(){
System.out.println("This is task 2");
}
void task3(){
System.out.println("This is task 3");
}
}
```

```
class SingleThread{
public static void main(String args[]){
MyThread obj=new MyThread();
Thread t1=new Thread(obj);
t1.start();
}
}
```

Output

```
This is task 1
This is task 2
This is task 3
```


Example program to terminate the thread by pressing the Enter button

```
import java.util.*;
import java.io.*;
class Myclass extends Thread{
        boolean stop=false;
        public void run()
        {
        for(int i=1;i<=10000;i++)
        {
        System.out.println(i);
        if(stop)return; // come out of run()
        }
        }
}
class Demo{
public static void main(String args[]){
Myclass obj=new Myclass();
Thread t=new Thread(obj);
t.start();
Scanner in=new Scanner(System.in);
in.nextLine();
obj.stop=true;
}
}
```


Multitasking using Two Threads

```
class MyThread implements Runnable{
String str;
MyThread(String str){
this.str=str;
}
public void run()
{
for(int i=1;i<=10;i++){
System.out.println(str+" : "+i);
try{
Thread.sleep(2000);
//ceases execution of a thread for 2000
milliseconds.
}
catch(InterruptedException e){
        e.printStackTrace();
        }
}
}
}
```

```
class Theatre{
public static void main(String
args[]){
MyThread obj1=new
MyThread("Generate
Ticket");
MyThread obj2=new
MyThread("Show the Seat");
Thread t1=new
Thread(obj1);
Thread t2=new
Thread(obj2);
t1.start();
t2.start();
}
}
```

Output

```
Show the Seat : 1
Generate Ticket : 1
Generate Ticket : 2
Show the Seat : 2
Generate Ticket : 3
Show the Seat : 3
Generate Ticket : 4
Show the Seat : 4
Generate Ticket : 5
Show the Seat : 5
Generate Ticket : 6
Show the Seat : 6
Generate Ticket : 7
Show the Seat : 7
Generate Ticket : 8
Show the Seat : 8
Generate Ticket : 9
Show the Seat : 9
Generate Ticket : 10
Show the Seat : 10
```

Multitasking (different tasks) by two threads

```
class Simple1 extends Thread{
 public void run(){
            int sum=0;
            for(int i=1;i<=10;i++)
            sum=sum+i;
            System.out.println("Sum of 10 numbers
is " + sum);
 }
}
class Simple2 extends Thread{
 public void run(){
            int fact=1;
   for(int i=1;i<=5;i++)
            fact=fact*i;
            System.out.println("Factorial of 5 is "+
fact);
 }
}
```

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

  Simple1 s1=new Simple1();
  Simple2 s2=new Simple2();
  Thread t1=new Thread(s1);
  Thread t2=new Thread(s2);
  t1.start();
  t2.start();
 }
}
```

Output:

```
C:\Users\BITS-PC\Desktop\Java\Multi-threading>java MultitaskingDifferent
Sum of 10 numbers is 55
Factorial of 5 is 120
```


Another Example of Multitasking with two threads (Thread.sleep())

```
class Simple1 extends Thread{
 public void run(){
            for(int i=1;i<=10;i=i+2){
            System.out.println("Odd number "
+ i);
            try{
            Thread.sleep(1000);
            }catch(Exception e){}
            }
 }
}
class Simple2 extends Thread{
 public void run(){
  for(int i=0;i<=9;i=i+2){
            System.out.println("Even number "
+ i);
            try{
            Thread.sleep(1000);
            }catch(Exception e){}
            }
 }
}
```

```
class MultitaskingDifferent{
 public static void main(String args[]){
  Simple1 t1=new Simple1();
  Simple2 t2=new Simple2();
 t2.start();
 t1.start();
 }
}
```

Output

```
Even number 0
Odd number 1
Even number 2
Odd number 3
Even number 4
Odd number 5
Even number 6
Odd number 7
Even number 8
Odd number 9
```


Another Example of Multitasking with two threads

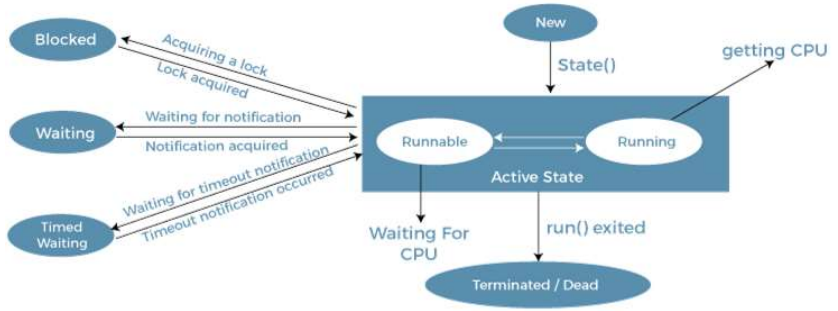
```
class SimpleOne implements Runnable{
 public void run(){
            try{
            for(int i=1;i<=10;i=i+2)
            System.out.println("Odd number " + i);
            }catch(Exception e){}
 }
}
class SimpleTwo implements Runnable{
 public void run(){
            try{
            for(int i=0;i<=9;i=i+2)
            System.out.println("Even number " + i);
            }catch(Exception e){}
 }
}
```

```
class MultitaskDifferent{
 public static void main(String args[]){
  SimpleOne s1=new SimpleOne();
  SimpleTwo s2=new SimpleTwo();
 Thread t1=new Thread(s1);
 Thread t2=new Thread(s2);
 t2.start();
 t1.start();
 }
}
```

Output

```
Even number 0
Even number 2
Even number 4
Even number 6
Even number 8
Odd number 1
Odd number 3
Odd number 5
Odd number 7
Odd number 9
```


Life Cycle of a Thread :



Obtaining a Thread's State

- We can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**.
- It is shown here:

  Thread.State getState( )

  It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. The values that can be returned are given in the below table.

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority (For example, how an operating system implements multitasking can affect the relative availability of CPU time).
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting for on I/O), it will preempt the lower-priority thread.
- In theory, threads of equal priority should get equal access to the CPU, but depends on other factors of OS.
- **setPriority():** Thread class of this method can be used to set a thread's priority**.** The general form is given below:

  final void setPriority(int level)

  Note: The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY,** which are 1 and 10 respectively. The default value, i.e., **NORM_PRIORITY** is **5.**
- **getPriority():** It can be used to obtain the current priority value of a thread.

  Its general form is shown below:

  final int getPriority()

Role of Thread Scheduler

- A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.
- In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.
- However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.
- There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Importance of isAlive() and join() methods of Thread class

- We often may want the main thread to finish last.
- Two ways are there:

  i) isAlive(): The general form of isAlive() is given here:

  final Boolean isAlive()

  It **returns** true if the thread upon which it is called is still running. It returns **false** otherwise

  ii) join(): When this method is called, then it waits until a thread on which it is called finishes its execution. The general form is shown below:

  final void join() throws InterruptedException

  Note: We can also specify the amount time that we want to wait for the specified thread to terminate.

Example program to demonstrate isAlive() and join()

```
class NewThread implements Runnable{
String name;
Thread t;
NewThread(String threadname){
name=threadname;
t=new Thread(this,name);
System.out.println("New thread:"+t);
t.start();//start the thread
}
public void run(){
try{
for(int i=5;i>0;i--){
System.out.println(name+" : "+i);
Thread.sleep(1000);
}
}catch(InterruptedException e)
{
System.out.println(name+" Interrupted");
}
System.out.println(name+" exiting");
}
}
```

```
class ThreadDemojoin{
public static void main(String args[]){
NewThread ob1=new NewThread("One");
NewThread ob2=new NewThread("Two");
NewThread ob3=new NewThread("Three");
System.out.println("Thread one is alive: "+ob1.t.isAlive());
System.out.println("Thread two is alive: "+ob2.t.isAlive());
System.out.println("Thread three is alive: "+ob3.t.isAlive());
//wait for threads to finish
try{
System.out.println("Waiting for threads to finish");
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e){
System.out.println("Main thread Interrupted");
}
System.out.println("Thread one is alive: "+ob1.t.isAlive());
System.out.println("Thread two is alive: "+ob2.t.isAlive());
System.out.println("Thread three is alive: "+ob3.t.isAlive());
}
}
```


Multiple Threads Acting on Single Object with an Example Scenario (Ticket Booking)

- Let us assume that only one berth is available in a train, and two passengers (threads) are asking for that berth.
- In reservation counter 1, the clerk has sent a request to the server to allot that berth to his passenger.
- Similarly, in reservation counter 2, the second clerk has also sent a request to the server to allot that berth to his passenger.

What happens when two threads act upon a single object?

We may get undesired results (This is also referred to as inconsistent state).

Output

```
New thread:Thread[One,5,main]
New thread:Thread[Two,5,main]
New thread:Thread[Three,5,main]
Thread one is alive: true
Thread two is alive: true
Thread three is alive: true
Waiting for threads to finish
Two : 5
Three : 5
One : 5
Three : 4
One : 4
Two : 4
One : 3
Three : 3
Two : 3
One : 2
Three : 2
Two : 2
One : 1
Three : 1
Two : 1
Three exiting
One exiting
Two exiting
Thread one is alive: false
Thread two is alive: false
Thread three is alive: false
```



```
class Reserver implements Runnable{
//available berths are 1
int available=1;
int wanted;
//accept wanted berths at run time
Reserver(int i)
{
wanted=i;
}
public void run(){
//display available berths
System.out.println("Available Berths= "+available);
//if available berths are more than wanted berths
if(available>=wanted){
//get teh name of passenger
String name=Thread.currentThread().getName();
// allot the berth to him
System.out.println(wanted+" Berths reserved for "+name);
try{
Thread.sleep(1500);//wait for printing teh ticket
available=available-wanted;
//update the no. of available berths
}catch(InterruptedException ie){}
}
//if available berths are less, display sorry
else
System.out.println("Sorry, no berths");
}
}
```

Ticket Booking Example without synchronization

```
class unsafe{
public static void main(String args[])
{
Reserver obj=new Reserver(1);//tell that 1 berth is
needed
//attach first thread to the object
Thread t1=new Thread(obj);
//attach second thread to the same object
Thread t2=new Thread(obj);
t1.setName("First Person");
t2.setName("Second Person");
//send the requests for berths
t1.start();
t2.start();
}
}
```

Output

```
Available Berths= 1
Available Berths= 1
1 Berths reserved for First Person
1 Berths reserved for Second Person
```

From this output,
It can be observed that same berth has been allotted to both the passengers. **This is inconsistent and is not expected and it is wrong.**

The result is unreliable since both the threads are acting on the same object simultaneously.

What is the solution?.
Let us make thread t2 to wait until the thread t1 completes and comes out. We should not allow any other thread to enter the object till t1 comes out. We need to aim at preventing the threads to act on the same object simultaneously. This is called **Thread Synchronization** or **Thread Safe.**

Synchronization in Two ways

1. **synchronized Method:**
   - All objects in java have their own implicit monitor associated with them.
   - To enter an object's monitor, it is required to just call a synchronized method that has been modified with **synchronized** keyword.
   - While a thread is inside a synchronized method, all other threads that try to call (or any other synchronized method) on the same instance have to wait.
   - To exit the monitor and relinquish the control of the object to the next waiting thread, the owner of the monitor simply **returns** from the synchronized method.
   - An **example of synchronized** method is shown below:

```
synchronized void display(){
Statements;
}
```


Synchronization

- When two or more threads need access to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time.
- When a thread is already acting on an object, preventing any other thread from acting on the same object is called Thread Synchronization or Thread safe.
- Synchronization ensures that only one thread only will be allowed to use the shared resource at a time.
- Synchronization object is like a locked object, locked on thread.
- Key element of synchronization is Monitor (think of it as a small box that can hold only one thread).
- A monitor is an object that is used as a mutually **exclusive lock**.
- Once a thread a enter a monitor, all other threads must wait until that thread exists the monitor. Only one thread can own a monitor at a given time. ` In this way, a monitor can be used to protect a shared data from being manipulated by more than one thread at a time.

Note: There is no class called **Monitor,** but each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called**.** Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Synchronization in Two ways (Cont…)

2. **synchronized Block:**

- We can also synchronize methods by putting in synchronized block.
- The general form of the synchronized block is

```
synchronized(object){
//statements to be synchronized
}
```

Thread Synchronization using Synchronized Block

Ticket Booking Example with synchronized Block

```
class Reserver implements Runnable{
int available=1; //available berths are 1
int wanted;
//accept wanted berths at run time
Reserver(int i)
{
wanted=i;
}
public void run(){
synchronized(this){//synchronize the current object
//display available berths
System.out.println("Available Berths= "+available);
//if available berths are more than wanted berths
if(available>=wanted){
//get teh name of passenger
String name=Thread.currentThread().getName();
// allot the berth to him
System.out.println(wanted+" Berths reserved for "+name);
try{
Thread.sleep(1500);//wait for printing teh ticket
available=available-wanted;
//update the no. of available berths
}catch(InterruptedException ie){}
}
//if available berths are less, display sorry
else
System.out.println("Sorry, no berths");
}//end of synchronized block
}
}
```

```
class safe{
public static void main(String args[])
{
Reserver obj=new Reserver(1);//tell that 1 berth is needed
//attach first thread to the object
Thread t1=new Thread(obj);
//attach second thread to the same object
Thread t2=new Thread(obj);
t1.setName("First Person");
t2.setName("Second Person");
//send the requests for berths
t1.start();
t2.start();
}
}
```


Deadlock Problem

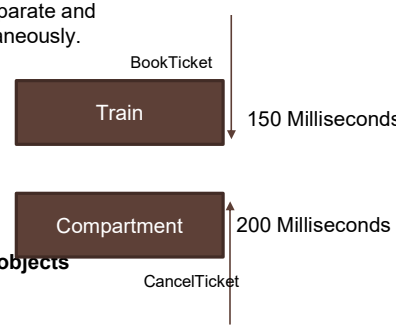
- When we synchronize the threads, there is a possibility of 'deadlock' to occur.
- **Def(Thread Deadlock):** When a thread has locked an object and waiting for another object to be released by another thread and the other thread is also waiting for the first thread to release the lock on first object, both the threads will **continue waiting forever. This is called deadlock.**

Output

```
Available Berths= 1
1 Berths reserved for First Person
Available Berths= 0
Sorry, no berths
```


Deadlock Example

- For example, **Trian Ticket Booking and Cancellation,** thousands of people book tickets in trains and cancel tickets also.
- The developer here may visualize booking tickets and cancellation them are **Reverse procedures.** Hence, he/she may write these two tasks as separate and Opposite tasks and assign 2 different threads to do these tasks simultaneously.
- **BookTicket:**
  - i) The thread will enter the train object to verify that the ticket is available or not. If so, it updates the available number of tickets in the train object. For this, it takes **150 milliseconds.**
  - ii) Then, it enters the compartment object, here, it should allot the ticket for the passenger and update its status **'reserved'**



Note: Thread should first go through the Train and Compartment objects

Thread Deadlock

Deadlock Example (Cont…)

-CancelTicket:

- i) The thread here first enters the Compartment object, and update the status of the ticket as available. **For this, it may take 200 milliseconds.**
- ii) Then it enters train object and updates the available number of tickets there.

Note: Here, t**he thread would go through the Compartment and Train objects.**

- For both booking a ticket and cancelling a ticket, a thread enter both Train and Compartment objects. **Here, more than one thread will access these threads, we need to synchronize them.**

Deadlock happens:

- After 150 milliseconds, the **BookTicket** thread tries to come out of train object and wants to lock on Compartment object, by entering into it. At this time, it finds that the compartment object is already locked by another thread(**CancelTicket**) and hence it wait.
- After 200 milliseconds, the **CancelTicket** thread which is in compartment object completes its execution and wants to enter and lock on train object. But it will find that the train object is already locked by **BookTicket** thread and hence is not available. The CancelTicket thread will wait for the train object which should be unlocked by the **BookTicket**.
- Here, **BookTicket** thread keeps on waiting for the **CancelTicket** thread to unlock the compartment object and the **CancelTicket** thread keeps on waiting for the **BookTicket** thread to unlock the train object.
- Each thread is expecting the other thread to release the object first, and then only it is willing to release its own object.. This way both the threads will wait forever suspending any further execution.

Program to demonstrate Deadlock (Cont…)

```
public class Deadlock{
public static void main(String args[]){
//take train, compartment as objects of Object class
Object train=new Object();
Object compartment=new Object();
//create objects to BookTicket, CancelTicket classes.
BookTicket obj1=new BookTicket(train, compartment);
CancelTicket obj2=new CancelTicket(train, compartment);
//attach 2 threads to these objects
Thread t1=new Thread(obj1);
Thread t2=new Thread(obj2);
//run the threads on the objects
t1.start();
t2.start();
}
}
```


Program to demonstrate Deadlock

```
class BookTicket extends Thread{
Object train, comp;//asume that train and compartment as objects.
BookTicket(Object train, Object comp){
this.train=train;
this.comp=comp;
}
public void run(){
//lock on train
synchronized(train){
System.out.println("BookTicket locked on train");
try{
Thread.sleep(150);
}catch(InterruptedException e){}
System.out.println("BookTicket now waiting to lock on compartment...");
synchronized(comp){
System.out.println("BookTicket locked on Compartment");
}
}
}
}
```

```
class CancelTicket extends Thread{
Object train, comp;//asume that train and compartment as objects.
CancelTicket(Object train, Object comp){
this.train=train;
this.comp=comp;
}
public void run(){
//lock on compartment
synchronized(comp){
System.out.println("CancelTicekt locked on compartment");
try{
Thread.sleep(200);
}catch(InterruptedException e){}
System.out.println("CancelTicket now waiting to lock on train...");
synchronized(train){
System.out.println("CancelTicket locked on train");
}
}
}
}
```


Output

```
BookTicket locked on train
CancelTicekt locked on compartment
BookTicket now waiting to lock on compartment...
CancelTicket now waiting to lock on train...
```

How do you avoid deadlocks?

- There is no specific solution for the problem of deadlocks.
- It depends on the logic used by the programmers. The programmer should design his program in such a way that it does not form any deadlock.
- In the preceding example, if the programmer used the threads in such a way that the **CancelTicket** thread follows the **BookTicket,** then deadlock situation would have been avoided.

Output

BookTicket locked on train
BookTicket now waiting to lock on compartment…
BookTicket locked on compartment
CanceltTicket locked on train
CancelTicket now waiting to lock train…
CancelTicket locked on compartment

The run() method inside the CancelTicket should be changed as shown below to avoid the deadlock

```
public void run()
{
//first lock on train like the BookTicket does
synchronized(train)
{
System.out.println("CancelTicket locked on train");
try{
Thread.sleep(200);
}
catch(InterruptedException e){}
System.out.println("CancelTicket now waiting to lock on compartment...");
synchronized(comp)
{
System.out.println("CancelTicket locked on compartment");
}
}
}
```


Cooperation (Inter-thread communication)

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:
    1. wait()
    2. notify()
    3. notifyAll()

1. Wait() method:

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

Synchronization

Method	Description
public final void wait() throws InterruptedException	It waits until object is notified.
public final void wait(long timeout) throws InterruptedException	It waits for the specified amount of time.

2. notify() method:

The notify() method wakes up a single thread that is waiting on this object's monitor. If multiple threads are waiting for notification, and we use the notify() method, only one thread will receive the notification. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax**: public final void notify()**

Example: Producer-Consumer Problem

- A Consumer thread waits for a Producer thread to produce the data (or some goods).
- When the Producer thread completes production of data, then the Consumer thread should take that data and use it.
- The Producer take a StringBuffer object to store data, i.e., numbers from 1 to 10.

How can Consumer take the data from the StringBuffer of the Producer and use it?
How can Producer be sure that the data produced by the producer is consumed by the Consumer?

Synchronization

3. notifyAll() method

Wakes up all threads that are waiting on this object's monitor. One of the threads will be granted access.

Syntax: **public final void notifyAll()**

Program to demonstrate Producer-Consumer Problem

```
class Producer extends Thread{
//to add data, we use String buffer object
StringBuffer sb;
//dataprodover will be true when data production is over
boolean dataprodover=false;
Producer(){
sb=new StringBuffer();//allot memory
}
public void run(){
//go on appending data (numbers) to string buffer
//go on appending data (numbers) to string buffer
for(int i=1;i<=10;i++)
{
try{
sb.append(i+":");
Thread.sleep(100);
System.out.println("appending");
}catch(Exception e){}
}
dataprodover=true;
}
```

```
class Consumer extends Thread{
//create producer reference to refer to producer object from Consumer
class
Producer prod;
Consumer(Producer prod){
this.prod=prod;
}
public void run(){
//wait till the sb object is released and a notification is sent
try{
while(!prod.dataprodover)
Thread.sleep(10);
}catch(Exception e){}
System.out.println(prod.sb);
}
}
```

Program to demonstrate Producer-Consumer Problem (Cont…)

```
public class Communicate{
public static void main(String args[]) throws Exception{
//Producer produces some data which consumer consumes
Producer obj1=new Producer();
//Pass producer object to consumer so that is then available to consumer.
Consumer obj2=new Consumer(obj1);
//create two threads and attach to Producer and Consumer
Thread t1=new Thread(obj1);
Thread t2=new Thread(obj2);
//Run the threads
t2.start();//Consumer waits
t1.start();//Producer starts production
}
}
```


Observation of Previous Program

- The program was not an efficient.
- Why?.
- The consumer check the **dataprodover** at some point of time, and finds it false. So it goes into sleep for the nest 10 milliseconds and then only it can find **dataprodover** is true. It means there may be **a time delay of 1 to 9 milliseconds** to receive the data after its production is completed.

This wastage of time can be avoided by allowing threads to communicate with each other.

Output

```
appending
appending
appending
appending
appending
appending
appending
appending
appending
appending
1:2:3:4:5:6:7:8:9:10:
```


Producer-Consumer Program without Time Delay (with the help of wait() and notify())

```
class Producer extends Thread{
//to add data, we use String buffer object
StringBuffer sb;
//dataprodover will be true when data production is over
boolean dataprodover=false;
Producer(){
sb=new StringBuffer();//allot memory
}
public void run(){
//go on appending data (numbers) to string buffer
synchronized(sb)
{
//go on appending data (numbers) to string buffer
for(int i=1;i<=10;i++)
{
try{
sb.append(i+":");
Thread.sleep(100);
System.out.println("appending");
}catch(Exception e){}
}
sb.notify();
}
}
}
```

```
class Consumer extends Thread{
//create producer reference to refer to producer object from Consumer
class
Producer prod;
Consumer(Producer prod){
this.prod=prod;
}
public void run(){
synchronized(prod.sb)
{
//wait till the sb object is released and a notification is sent
try{
prod.sb.wait();
}catch(Exception e){}
System.out.println(prod.sb);
}
}
}
```

```
class CommunicateWithoutTimeDelay{
public static void main(String args[]) throws Exception{
//Producer produces some data which consumer consumes
Producer obj1=new Producer();
//Pass producer object to consumer so that is then available to consumer.
Consumer obj2=new Consumer(obj1);
//create two threads and attach to Producer and Consumer
Thread t1=new Thread(obj1);
Thread t2=new Thread(obj2);
//Run the threads
t2.start();//Consumer waits
t1.start();//Producer starts production
}
}
```

Output

```
appending
appending
appending
appending
appending
appending
appending
appending
appending
appending
1:2:3:4:5:6:7:8:9:10:
```


Daemon Threads

- *These threads will continuously execute to provide services to other threads.*
- *For example, oracle.exe is a program (a thread) that continuously executes in a computer.*
- *Garbage collection is another example for daemon threads.*
- *These are low priority threads which provides background processing.*
- *To make a thread t as a daemon thread, we can use* ***setDaemon().*** *Example, t.setDaemon(true)*
- *We can check whether thread is deamon thread or not.*

  boolean b = t.isDeamon();

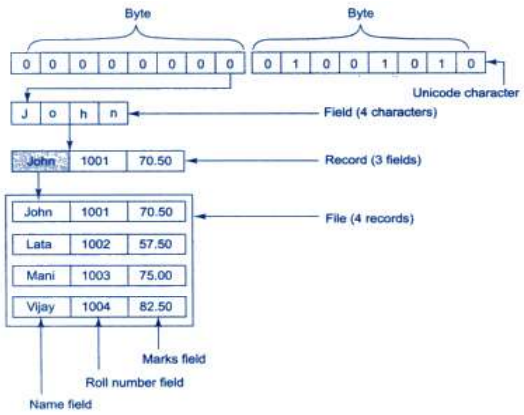
Agenda (java.io)

- File
- Byte Streams
- Character Streams
- Object Serializability

java.io

- The **java.io** supports basic techniques for reading and writing files, and handling I/O Exceptions, and closing a file.
- It is to be noted that most programs cannot accomplish their goals **without accessing external data**.
- The data is retrieved from **an input source**.
- The results of a program are sent to **an output destination.**
- In Java, these sources or destinations are defined very broadly, e.g., a network connection, memory buffer, or disk file. Although these are physical different, but the same abstraction: the **stream.**

Structure



What is a file?

Defintion: A file is a collection of related records placed in a particular area on the disk.

- A record is composed of several fields.
- A field is a group of characters or bytes or class objects.

Storing and processing files is called as File processing

- Example: Creating files, updating files, and manipulation of data.

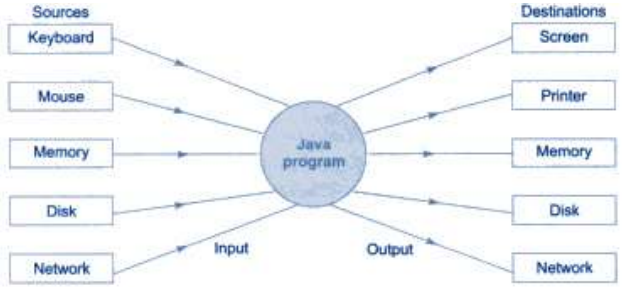
Reading and writing of data in a file can be done

- at the level of bytes or
- characters or
- fields depending on the application requirement.

File Processing

Input: flow of data into a program

Output: flow of data out of a program



File

- The File class does not operate on streams and it directly with files and the file system.
- It describes the properties of a file itself.
- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.
- Files are a primary source and destination for data within many programs.
- A directory in Java is treated simply as a **File** with one additional property: a list of filenames that can be examined by the **list( )** method.

File Constructors

1. File(String *directoryPath*)
2. File(String *directoryPath*, String *filename*)
3. File(File *dirObj*, String *filename*)
4. File(URI *uriObj*)

Here, ***directoryPath*** is the path name of the file; ***filename*** is the name of the file or subdirectory**;** ***dirObj*** is a **File** object that specifies a directory; and ***uriObj*** is a **URI** object that describes a file.

Example Code segment

File f1 = new File("/"); // includes directory path only
File f2 = new File("/",“f1.txt"); // includes the path and the filename
File f3 = new File(f1,“f1.txt");
// the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

Note: Java allows us to work UNIX and Windows path conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. If you are using the Windows convention of a backslash character (\), you will need to use its **escape sequence (\\)** within a string.

Obtaining Properties of a File Object

- **getName( )** returns the name of the file.
- **getParent( )** returns the name of the parent directory.
- **exists( )** returns **true** if the file exists, **false** if it does not.
- **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.
- The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.
- **renameTo(*newName*):** *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed.
- **delete( )**: It deletes the disk file represented by the path of the invoking **File** object. It returns **true** if it deletes the file and **false** if the file cannot be removed. We can also use **delete( )** to delete a directory if the directory is empty.

Few more methods of File class

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
long getUsableSpace( )	Returns the number of usable free bytes of storage available on the partition associated with the invoking object.
boolean isHidden( )	Returns **true** if the invoking file is hidden. Returns **false** otherwise.
boolean setLastModified(long *millisec*)	Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

Directory

- A directory is a **File** that contains a list of other files and directories
- When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**.
- **list( )** method on file object can be used to extract the list of other files and directories inside.
- list() has two forms:
  - **String[ ] list( ):** The list of files is returned in an array of **String** objects.
  - **String[ ] list(FilenameFilter *FFObj*):** It can be used to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*.

Note: **FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)


```
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

Example program to demonstrate File Properties

Output

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory

is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes
```


Program to Demonstrate Directory

```
import java.io.*;
class DirList{
public static void main(String args[]){
File f1=new File("java");
if(f1.isDirectory()){
System.out.println("Directory is "+f1.getName());
String s[]=f1.list();
System.out.println("The contents of " +f1.getName()+" are");
for(int i=0;i<s.length;i++){
System.out.println(s[i]);
}
}
else{
System.out.println("Not a directory");
}
}
}
```

Output

```
Directory is java
The contents of java are
copyright.txt
hello.txt
```

Creating Directories

- There are two more useful **File** utility methods: **mkdir( )** and **mkdirs( )**.
- The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure.
- **mkdirs( )** method can be used to create a directory for which no path exists. Here, it creates both a directory and all the parents of the directory.

AutoCloseable, Closeable, and Flushable Interfaces

- **Closeable** and **Flushable** are defined in **java.io.** The third, **AutoCloseable**, is packaged in **java.lang**.
- **AutoCloseable** provides support for the **try**-with-resources statement, which automates the process of closing a resource.
  - Only objects of classes that implement **AutoCloseable** can be managed by **try**-with-resources.
  - The **AutoCloseable** interface defines only the **close( )** method:

  void close( ) throws Exception

  This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try**-with-resources statement, thus eliminating the need to explicitly call **close( )**.
- The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed.
  - **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

Flushable interface

- Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

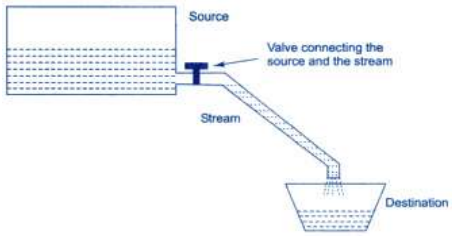
  void flush( ) throws IOException
- Flushing a stream typically causes buffered output to be physically written to the underlying device.
- This interface is implemented by all of the I/O classes that write to a stream.

Stream

A stream in JAVA is a path along which data flows.

It has a source (of data) and a destination (of data).

Both, the source and the destination may be a physical device or programs or other streams in a same program.



Streams (Cont…)

- Streams **facilitates transporting** data from one place to another. Different streams are need to send or receive data through different sources, such as to receive data from keyboard, we need a stream and to send data to a file, we need another streams.
- Without streams, it is not possible to move data in java.
- A stream is a logical entity that either produces or consumes information and it is linked to a physical devices by the Java I/O subsystem.

Java Streams can also be classified as **Byte Streams** and **Character Streams**.

Streams (Cont…)

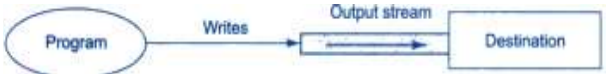
Java streams are classified into:

- Input Stream: These are the streams that receive or read data.
- Output Stream: These are the streams that send or write the data.

All streams are represented by classes in **java.io** (input-output) package.

An input stream extracts (or reads) data from the source (file) and sends it to the program.

An output stream takes data from the program and sends (writes) into the destination (file).



Byte Streams

- Byte streams represent data in the form of individual bytes. They are used to handle any **characters (text), images, audio, and video files**.
- If a class name ends with the word 'Stream', then it comes under byte streams.
- **InputStream** reads bytes and **OutputStream** writes bytes. For example, FileInputStream, FileOutputStream, BufferedInputStream, BufferedOuputStream.

Character Streams

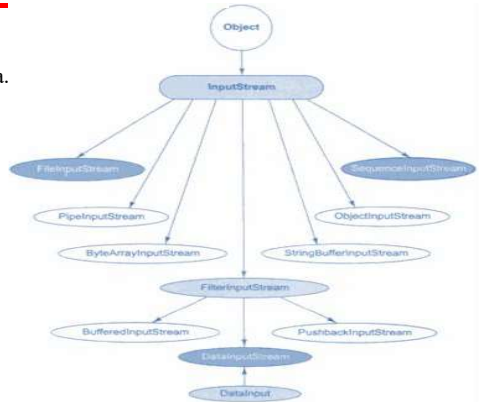
- They represent data as characters of each 2 bytes. They can always store and retrieve data in the **form of characters (or text) only.** It means text streams are more suitable for handling text files and they are **not suitable to handle the images, audio, or video files**.
- If a class name ends with the word 'Reader' or 'Writer' then it taken as a text stream.
- Reader reads text and Writer writes text. For example: FileReader, FileWriter, BufferedReader, BufferedWriter

InputStream Classes

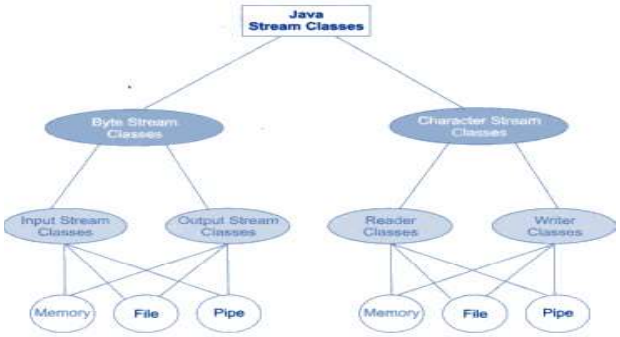
Input stream classes are used to read 8-bit bytes.

InputStream is the super class of input stream classes in Java.

- It is a abstract class
- Defines methods for performing:
  - Reading bytes.
  - Closing streams.
  - Marking positions in streams.
  - Skipping ahead in a stream.
  - Finding number of bytes in a stream.



Java Stream Classification

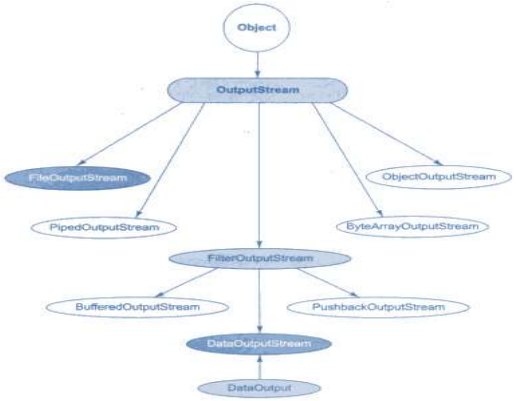


Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream.
static InputStream nullInputStream( )	Returns an open, but null input stream, which is a stream that contains no data. Thus, the stream is always at the end of the stream and no input can be obtained. The stream can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when an attempt is made to read at the end of the stream.
byte[ ] readAllBytes( )	Beginning at the current position, reads to the end of the stream, returning a byte array that holds the input.
byte[ ] readNBytes(int *numBytes*)	Attempts to read *numBytes* bytes, returning the result in a byte array. If the end of the stream is reached before *numBytes* bytes have been read, then the returned array will contain less than *numBytes* bytes. (Added by JDK 11.)
int readNBytes(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.
long transferTo(OutputStream *strm*)	Copies the bytes in the invoking stream into *strm*, returning the number of bytes copied.

Methods Defined by **InputStream** Class

Output Stream Classes

Output stream classes are derived from the **OutputStream** class.
Like InputStream class, OutputStream class is also an abstract class.



OutputStream Class

OutputStream is the super class of output stream classes in Java.

- It is a abstract class
- Defines methods for performing:
  1. Writing bytes
  2. Closing streams
  3. Flushing streams

Methods Defined by OutputStream Class

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an IOException.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
static OutputStream nullOutputStream( )	Returns an open, but null output stream, which is a stream to which no output is actually written. Thus, its output methods can be called but don't actually produce output. The stream can, however, be closed. (Added by JDK 11.)
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

FileOutputStream

- **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file.
- It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- Four of its constructors are shown here:

FileOutputStream(String *filePath*)
FileOutputStream(File *fileObj*)
FileOutputStream(String *filePath*, boolean *append*)
FileOutputStream(File *fileObj*, boolean *append*)

Note: The above throw a **FileNotFoundException**. Here, ***filePath*** is the full path name of a file, and ***fileObj*** is a **File** object that describes the file. If ***append*** is **true**, the file is opened in append mode.

```
import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();
    FileOutputStream f0 = null;
    FileOutputStream f1 = null;
    FileOutputStream f2 = null;

    try {
      f0 = new FileOutputStream("file1.txt");
      f1 = new FileOutputStream("file2.txt");
      f2 = new FileOutputStream("file3.txt");

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
      }
    }
  }
}
```

Java Program to create three files using **FileOutputStream**

FileInputStream

- The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file.
- Two commonly used constructors are shown here:

  FileInputStream(String *filePath*)

  FileInputStream(File *fileObj*)

Note: Either of the above can throw a **FileNotFoundException**. Here, ***filePath*** is the full path name of a file, and ***fileObj*** is a **File** object that describes the file.

- When a **FileInputStream** is created, it is also opened for reading.
- **FileInputStream** overrides several of the methods in the abstract class **InputStream**.

Output (Files content)

Here are the contents of each file after running this program. First, **file1.txt**:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```



```
import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    // Use try-with-resources to close the stream.
    try ( FileInputStream f =
          new FileInputStream("FileInputStreamDemo.java") ) {

      System.out.println("Total Available Bytes: " +
                         (size = f.available()));

      int n = size/40;
      System.out.println("First " + n +
                         " bytes of the file one read() at a time");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }

      System.out.println("\nStill Available: " + f.available());
      System.out.println("Reading the next " + n +
                         " with one read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }

      System.out.println(new String(b, 0, n));
      System.out.println("\nStill Available: " + (size = f.available()));
      System.out.println("Skipping half of remaining bytes with skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());

      System.out.println("Reading " + n/2 + " into the end of array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
      }

      System.out.println(new String(b, 0, b.length));
      System.out.println("\nStill Available: " + f.available());
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Java Program to read byte, array of bytes, subrange of an array of bytes using **FileInputStream**

Output

```
Total Available Bytes: 1714
First 42 bytes of the file one read() at a time
// Demonstrate FileInputStream.

impor
Still Available: 1672
Reading the next 42 with one read(b[])
t java.io.*;

class FileInputStreamD

Still Available: 1630
Skipping half of remaining bytes with skip()
Still Available: 815
Reading 21 into the end of array
t java.io.*;

c n) {
        Syst

Still Available: 794
```


Reading Single character using FileInputStream

```
import java.io.FileInputStream;
public class FISExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("input.txt");
          int i=fin.read();
          System.out.print((char)i);
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

Input.txt: hello world 123

Output:
h

Reading all characters using FileInputStream

```
import java.io.FileInputStream;
public class FISExample2 {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("input.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

Input.txt: hello world 123

Output
hello world 123


```
import java.io.*;
public class iobasic {
  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Another Example Program to demonstrate **FileInputStream** and **FileOutputStream**

Note: Here, it creates a file named output.txt in which the sample content in input.txt is copied.

Filtered Byte Streams

- They are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality.
- Typical extensions are buffering, character translation, and raw data translation.
- The filtered byte streams are:
  FilterInputStream and **FilterOutputStream**.
- Their constructors are shown here:
  FilterOutputStream(OutputStream *os*)
  FilterInputStream(InputStream *is*)
- The methods provided in these classes are identical to those in **InputStream** and **OutputStream**.

Buffered Byte Streams

- For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream.
- This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance.
- Because the buffer is available, skipping, marking, and resetting of the stream become possible.
- The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**.

BufferedInputStream

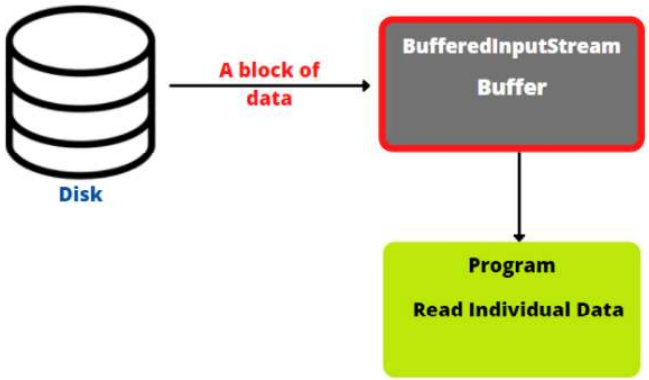
- Buffering I/O is a very common performance optimization
- Java's **BufferedInputStream** class allows you to "wrap" any **InputStream** into a buffered stream to improve performance.
- It has two constructors:
  BufferedInputStream(InputStream *inputStream*)
  BufferedInputStream(InputStream *inputStream*, int *bufSize*)

Note: The first form creates a buffered stream using a **default buffer size**. The default internal buffer size is **8192** bytes.

In the second, the size of the buffer is passed in bufSize. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance.

- **BufferedInputStream** also supports the **mark( )** and **reset( )** methods in addition to the methods implemented in any **InputStream.**

Internal Working of BufferedInputStream



Java Program to read data of file using BufferedInputStream

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("input.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Output
hello world 123

BufferedOutputStream

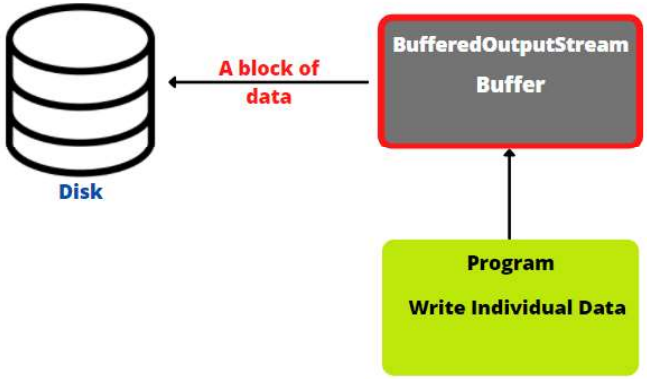
- A **BufferedOutputStream** is similar to any **OutputStream** with the exception that the **flush( )** method is used to ensure that data buffers are written to the stream being buffered.
- Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.
- Two available constructors:

  BufferedOutputStream(OutputStream *outputStream*)

  BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

Note: The first form creates a buffered stream using the default buffer size. In the second form, the size of the buffer is passed in bufSize.

Internal Working of BufferedOutputStream



```
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;

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

      String data = "This is a line of text inside the file";

      try {
         // Creates a FileOutputStream
         FileOutputStream file = new FileOutputStream("output.txt");

         // Creates a BufferedOutputStream
         BufferedOutputStream output = new BufferedOutputStream(file);

         byte[] array = data.getBytes();

         // Writes data to the output stream
         output.write(array);
         output.close();
      }

      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Program to Demonstrate **BufferedOutputStream**

Output: It creates output.txt file

SequenceInputStream

- The **SequenceInputStream** class allows you to concatenate multiple **InputStream**s
- A SequenceInputStream **constructor** uses either a pair of **InputStream**s or an Enumeration of InputStreams as its argument:

  SequenceInputStream(InputStream *first*, InputStream *second*)

  SequenceInputStream(Enumeration <? extends InputStream> *streamEnum*)

Note:

-This class fulfills read requests from the first **InputStream** until it runs out and then switches over to the second one.

- In the case of an **Enumeration**, it will continue through all of the **InputStream**s until the end of the last one is reached.
- When the end of each file is reached, its associated stream is closed. Closing the stream created by **SequenceInputStream** causes all unclosed streams to be closed.

Example Program

```
import java.io.*;
class ISExample {
  public static void main(String args[])throws Exception{
   FileInputStream input1=new FileInputStream("input1.txt");
   FileInputStream input2=new FileInputStream("input2.txt");
   SequenceInputStream inst=new SequenceInputStream(input1, input2);
   int j;
   while((j=inst.read())!=-1){
    System.out.print((char)j);
   }
   inst.close();
   input1.close();
   input2.close();
  }
}
```

Here: It will display the content of both the files

DataOutputStream and DataInputStream

- **DataOutputStream** and **DataInputStream** enable you to write or read primitive data to or from a stream.
- They implement the **DataOutput** and **DataInput** interfaces, respectively.
- These interfaces define methods that convert primitive values to or from a sequence of bytes.
- These streams make it easy to store binary data, such as integers or floating-point values, in a file.
- **DataOutputStream** extends **FilterOutputStream**, which extends **OutputStream**.
- In addition to implementing **DataOutput**, **DataOutputStream** also implements **AutoCloseable**, **Closeable**, and **Flushable**.

DataOutputStream Constructor

- **DataOutputStream(OutputStream *outputStream*)**

Here, ***outputStream*** specifies the output stream to which data will be written.

Note: When a **DataOutputStream** is closed (by calling **close( )**), the underlying stream specified by *outputStream* is also closed automatically.

- **DataOutputStream** supports all of the methods defined by its superclasses.
- **DataOutput** defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream.

Few Methods:

final void **writeDouble**(double *value*) throws IOException

final void **writeBoolean**(boolean *value*) throws IOException

final void **writeInt**(int *value*) throws IOException.

Here, *value* is the value written to the stream.

The methods of DataInput interface read a sequence of bytes and convert them into values of a primitive type.

Few Methods:

final double **readDouble( )** throws IOException

final boolean **readBoolean( )** throws IOException

final int **readInt( )** throws IOException

DataInputStream

- DataInputStream is the complement of the DataOutputStream.
- It extends **FilterInputStream**, which extends **InputStream**
- In addition to implementing the **DataInput** interface, **DataInputStream** also implements **AutoCloseable** and **Closeable**.
- Here is its only constructor:

  DataInputStream(InputStream inputStream)

  Here, inputStream specifies the input stream from which data will be read.
- When a **DataInputStream** is closed (by calling **close( )**), the underlying stream specified by *inputStream* is also closed automatically.
- Like DataOutputStream, DataInputStream supports all of the methods of its superclasses, but it is the methods defined by the DataInput interface that make it unique.


```
import java.io.* ;
public class ReadWriteFilter {
public static void main(String args[]) throws IOException {
FileOutputStream fos = new FileOutputStream( "input.txt");
DataOutputStream dos = new DataOutputStream(fos);
dos.writeInt(120);
dos.writeDouble(375.50);
dos.writeInt('A'+ 1);
dos.writeBoolean(true);
dos.writeChar('X');
dos.close();
fos.close();
FileInputStream fis = new FileInputStream( "input.txt");
DataInputStream dis = new DataInputStream(fis);
System.out.println(dis.readInt());
System.out.println(dis.readDouble());
System.out.println(dis.readInt());
System.out.println(dis.readBoolean());
System.out.println(dis.readChar());
dis.close();
fis.close();
}
}
```

Java Program to demonstrate **DataOutputStream** and **DataInputStream**

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java ReadWriteFilter
120
375.5
66
true
X
```

RandomAccessFile

- **RandomAccessFile** encapsulates a random-access file.
- **RandomAccessFile** is special because it supports positioning requests—that is, you can position the file pointer within the file.
- It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods.
- It also implements the **AutoCloseable** and **Closeable** interfaces.
- **RandomAccessFile** implements the standard input and output methods,

which you can use to read and write to random access files.

RandomAccessFile Constructors

- It has two constructors, which are shown below:

RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException
RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

Note: In the first form, ***fileObj*** specifies the file to open as a **File** object.
In the second form, the name of the file is passed **in *filename***.

- In both the constructors, the *access* determines what type of file access is permitted, which is explained as follows:
  - **"r",** the file can be read, but not written
  - **"rw",** then the file is opened in read-write mode
  - **"rws",** the file is opened for read-write operations and every change to the file's data or metadata will be immediately written to the physical device.
  - **"rwd",** the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

Methods

- **seek( )**: It can be used to set the current position of the file pointer within the file.

  - **void seek(long *newPos*) throws IOException**

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file.

Note: After a call to **seek( )**, the next read or write operation will occur at the new file position.

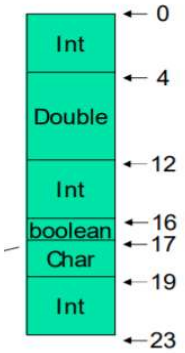
- **setLength( )**:

  void setLength(long *len*) throws IOException

This method sets the length of the invoking file to that specified by *len*. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

Java Program to demonstrate **RandomAccess**

```
import java.io.* ;
public class RandomAccess {
public static void main(String args[]) throws IOException {
// write primitive data in binary format to the "mydata" file
RandomAccessFile myfile = new RandomAccessFile( "input.txt", "rw");
myfile.writeInt(120);
myfile.writeDouble(375.50);
myfile.writeInt('A'+ 1);
myfile.writeBoolean(true);
myfile.writeChar('X');
// set pointer to the beginning of file and read next two items
myfile.seek(0);
System.out.println(myfile.readInt());
System.out.println(myfile.readDouble());
// set pointer to the 4th item and read it
myfile.seek(16);
System.out.println(myfile.readBoolean());
myfile.seek(myfile.length());
myfile.writeInt(2003);
// read 5th and 6th items
myfile.seek(17);
System.out.println(myfile.readChar());
System.out.println(myfile.readInt());
System.out.println( "File length: "+ myfile.length());
myfile.close();
}
}
```



Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java RandomAccess
120
375.5
true
X
2003
File length: 23
```

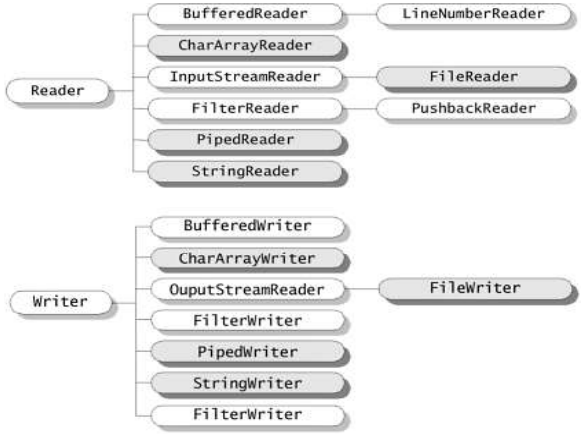

Few Methods of Reader

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
static Reader nullReader( )	Returns an open, but null reader, which is a reader that contains no data. Thus, the reader is always at the end of the stream and no input can be obtained. The reader can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when an attempt is made to read at the end of the stream.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
int read(CharBuffer *buffer*)	Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.

Character Streams

Similar to Byte stream classes, there are two kind of character stream classes.

- Character or text streams are used to store and retrieve data in the **form of characters (or text) only**
  - **Reader** class: It is an abstract class and defines Java's model of streaming character input.
  - **Writer** class: It is an abstract class that defines streaming character output.



Few Methods of Writer

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*.

FileWriter

- It is useful to create a file by writing content (characters) into it.

The constructors of FileWriter are provided below:

- FileWriter(Sting filePath)
- FileWriter(String filePath, boolean append)
- FileWriter(File fileObj)
- FileWriter(File fileObj,boolean append)

Here, **filePath** is the full path name of a file, and **fileObj** is a **File** object that describes the file. If append is **true,** then output is appended to the end of the file.

Note: An IOException is thrown when you make an attempt to open a read-only file.

Example program to demonstrate FileWriter

```
import java.io.*;
class CreateFile{
public static void main(String args[]){
String s="This is an example";
 try{
 FileWriter fw=new FileWriter("FWtest.txt");
 for(int i=0;i<s.length();i++)
 fw.write(s.charAt(i));
 fw.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Output

Fwtest.txt file will be created.

FileReader

- It is useful to read data in the form of characters from a text file.
- It has two most commonly used constructors:

  FileReader(String filePath)

  FileReader(File fileObj)

Note: Either of the above throw a **FileNotFoundException. filePath** is the full path name of a file and the **fileObj** is a File object that describes the file.

Example program to demonstrate FileReader

```
import java.io.*;
class ReadFile{
public static void main(String args[]){
int ch;
 try{
 FileReader fr=new FileReader("FWtest.txt");
 while((ch=fr.read())!=-1){
   System.out.print((char)ch);}
 fr.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java ReadFile
This is an example
```

BufferedWriter

- It buffers output.
- It improves the performance of writing by reducing the number of times data is actually physically written to the output device.
- It has two constructors:
  - BufferedWriter(Writer outputStream): It creates a buffer with default size
  - BufferedWriter(Writer outputStream, int bufSize): It creates a buffer with the specified size.
- Few Methods of BufferedWriter:

void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.

BufferedReader

- It improves performance by buffering input.
- It has two constructors:

i) BufferedReader(Reader inputstream): It creates a buffer with default size.

ii) BufferedReader(Reader inputstream, int bufSize): It creates a buffer with the specified size.

- Few Methods of BufferedReader:

int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array.
String readLine()	It is used for reading a line of text.

Example of BufferedWriter and BufferedReader

```
import java.io.*;
class CreateFile{
public static void main(String args[]){
String s="This is an example";
 try{
 FileWriter fw=new FileWriter("FWtest.txt");
 BufferedWriter bw=new BufferedWriter(fw);
 bw.write(s);
 bw.close();
 fw.close();
}catch(Exception e){System.out.println(e);}
}
}
```

```
import java.io.*;
class ReadFile{
public static void main(String args[]){
int ch;
 try{
 FileReader fw=new FileReader("FWtest.txt");
 BufferedReader br=new BufferedReader(fw);
 while((ch=br.read())!=-1){
                System.out.print((char)ch);
}
 br.close();
 fw.close();
}catch(Exception e){System.out.println(e);}
}
}
```

output

```
C:\Users\BITS-PC\Desktop\Java\IO>java ReadFile
This is an example
```


Reading Data from Keyboard

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Scanner
- DataInputStream, ((we have discussed it already)), etc.

Reading Data from Keyboard

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

Reading Input from Keyboard

```
import java.io.*;
class InputKeyboard{
public static void main(String args[])throws Exception{
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

Output

```
Enter your name
karan
Welcome karan
```


Reading input from keyboard until user writes stop

```
import java.io.*;
class InputKeyboard2{
public static void main(String args[])throws Exception{

 InputStreamReader r=new InputStreamReader(System.in);
 BufferedReader br=new BufferedReader(r);

 String name="";

  while(!name.equals("stop")){
   System.out.println("Enter data: ");
   name=br.readLine();
   System.out.println("data is: "+name);
  }

 br.close();
 r.close();
 }
}
```


Serialization

- *Serialization* is the process of storing object contents into a file.
- The stored objects can later be retrieved and used as and when needed. This is called **deserialization.**
- The class whose objects are to be stored in the file should implement '**Serializable'** interface of **java.io** package.
- Serialization is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects through deserialization.
- Serialization is also needed to implement ***Remote Method Invocation (RMI)***. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Note: **static** and **transient** variables cannot be serialized.
For example, (static int x=15;
transient String str="text";) cannot be serializable.

Serializable Interface

- It defines no members.
- It is simply used to indicate that a class may be serialized
- If a class is serializable, all of its subclasses are also serializable.
- If Serializable interface is not implemented by a class, then writing that class objects into a file will lead to **NotSerializable** exception.

ObjectOutput interface and ObjectInput interface

- **WriteObject()** is one of the methods defined by **ObjectOutput** interface that can be used to serialize an object. This method will throw an **IOException** on error conditions. This method accepts object to be serialized as an argument.
- **ReadObject()** is one of the methods defined by **ObjectInput** interface that can be used to deserailize an object, i.e., object can be read from a file. It may throw **IOException** on error conditions and also **ClassNotFoundException.**

ObjectOutputStream and ObjectInputStream classes

1. **ObjectOutputStream:**
   - It extends the Outputstream and implements **ObjectOutput** interface.
   - It is responsible for writing objects to a stream.
   - The **constructor** of the this class is provided below:
   ObjectOutputStream(OutputStream outStream) throws IOException

One of the important methods of this class is provided below:
final void WriteObject(Object obj): it writes obj to the invoking stream.

ObjectOutputStream and ObjectInputStream classes (Cont…)

2. **ObjectInputStream:**
   - It extends the InputStream class and implements the ObjectInput interface.
   - It is responsible for reading objects from a stream.
   - The **constructor** of this class is shown here:
   ObjectInputStream(InputStream instream) throws IOException

One of the important methods of this class is provided below:
Object readObject(): Reads an object form the invoking stream.

Java program to demonstrate serializing Employee Objects

```
import java.io.*;
import java.util.Date;
class Employee implements Serializable{
//instance variables
private int id;
private String name;
private float sal;
private Date doj;
//constructor
Employee(int i,String n, float s, Date d){
id=i;
name=n;
sal=s;
doj=d;
}
//to display employee details
void display(){
System.out.println(id+"\t"+name+"\t"+sal+"\t"+doj);
}
}
```

```
class StoreObj{
public static void main(String args[]){
try{
//to accept data from keyboard
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
//to store objects into objfile
FileOutputStream fos=new FileOutputStream("objfile");
ObjectOutputStream oos=new ObjectOutputStream(fos);
//ask how many objects to store
System.out.println("How many objects?");
int n=Integer.parseInt(br.readLine());
for(int i=0;i<n;i++){
//accept employee id number, name and salary
System.out.print("Enter emp id:");
int id=Integer.parseInt(br.readLine());
System.out.print("Enter name:");
String name=br.readLine();
System.out.print("Enter salary:");
float sal=Float.parseFloat(br.readLine());
//take current system date and time as for joining
Date d=new Date();
Employee eobj=new Employee(id,name,sal,d);
oos.writeObject(eobj);
}
oos.close();
}catch(Exception e){};
}
}
```


Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java StoreObj
How many objects?
3
Enter emp id:1
Enter name:arun
Enter salary:10000.23
Enter emp id:2
Enter name:john
Enter salary:11234.2
Enter emp id:3
Enter name:vishal
Enter salary:12000.2
```


Java Program to demonstrate de-serializing Employee objects from objfile

```
import java.io.*;
class ReadObjfile{
public static void main(String args[]){
//to read objects from objfile
try{
FileInputStream fis=new FileInputStream("objfile");
ObjectInputStream ois=new ObjectInputStream(fis);
// read objects and display till a null object is read
Employee e;
while((e=(Employee)ois.readObject())!=null)
{
e.display();
}
}catch(Exception e){}
}
}
```

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java ReadObjfile
1       arun    10000.23        Tue Nov 30 21:07:00 IST 2021
2       john    11234.2 Tue Nov 30 21:07:09 IST 2021
3       vishal  12000.2 Tue Nov 30 21:07:19 IST 2021
```


Discussion on Performance improvement using **BufferedOutputStream** and also on **flush()** method.

Total time of writing some content without using BufferedOutputStream

```
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
public class DirectWriteTime {
    public static void main(String[] args) {

        String data = "This is a line of text inside the file some random text
textaadfafdafafdafaddsafaddadadfadafafaddas";
        try {
            // Creates a FileOutputStream
            FileOutputStream file = new FileOutputStream("output.txt");
                    long start=System.currentTimeMillis();
                for (int i=0;i<data.length();i++){
                    file.write(data.charAt(i));
                    }
                    file.close();
                    long end=System.currentTimeMillis();
                    System.out.println("Time of write "+(end-start)+" milliseconds");
        }
        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java DirectWriteTime
Time of write 5 milliseconds
```


Total time of writing the same content using BufferedOutputStream

```
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
public class BOSExampleTime {
    public static void main(String[] args) {

        String data = "This is a line of text inside the file some random text
textaadfafdafafdafaddsafaddadadfadafafaddas";
        try {
            // Creates a FileOutputStream
            FileOutputStream file = new FileOutputStream("output.txt");

            // Creates a BufferedOutputStream
            BufferedOutputStream output = new BufferedOutputStream(file);
                    long start=System.currentTimeMillis();
                for (int i=0;i<data.length();i++){
                    output.write(data.charAt(i));
                    }
                output.close();
                    long end=System.currentTimeMillis();
                    System.out.println("Time of write "+(end-start)+" milliseconds");
        }
        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java BOSExampleTime
Time of write 2 milliseconds
```



```
import java.io.* ;
public class CombineStreams {
public static void main(String args[]) throws IOException {
// declare file streams
FileInputStream file1 = new FileInputStream("file1.txt");
FileInputStream file2 = new FileInputStream("file2.txt");
// declare file3 to store combined streams
SequenceInputStream file3 = null;
// concatenate file1 and file2 streams into file3
file3 = new SequenceInputStream(file1, file2);
BufferedInputStream inBuffer = new BufferedInputStream(file3);
BufferedOutputStream outBuffer = new BufferedOutputStream(System.out);
// read and write combined streams until the end of buffers
int ch;
while((ch = inBuffer.read()) != -1)
outBuffer.write(ch);
outBuffer.flush(); // check out the output by removing this line
System.out.println( "\nHello, This output is generated by CombineFiles.java program");
inBuffer.close();
outBuffer.close();
file1.close();
file2.close();
file3.close();
}
}
```

Example program to demonstrate the usage of flush()

file1.txt content:
hello world

File2.txt content:
welcome to the lab session

Output

```
C:\Users\BITS-PC\Desktop\Java\IO>java CombineStreams
hello world
welcome to the lab session
Hello, This output is generated by CombineFiles.java program
```



```
import java.io.* ;
public class CombineStreams {
public static void main(String args[]) throws IOException {
// declare file streams
FileInputStream file1 = new FileInputStream("file1.txt");
FileInputStream file2 = new FileInputStream("file2.txt");
// declare file3 to store combined streams
SequenceInputStream file3 = null;
// concatenate file1 and file2 streams into file3
file3 = new SequenceInputStream(file1, file2);
BufferedInputStream inBuffer = new BufferedInputStream(file3);
BufferedOutputStream outBuffer = new BufferedOutputStream(System.out);
// read and write combined streams until the end of buffers
int ch;
while((ch = inBuffer.read()) != -1)
outBuffer.write(ch);
//outBuffer.flush(); // check out the output by removing this line
System.out.println( "\nHello, This output is generated by CombineFiles.java program");
inBuffer.close();
outBuffer.close();
file1.close();
file2.close();
file3.close();
}
}
```

Example program to demonstrate the without flush()

Output

```
Hello, This output is generated by CombineFiles.java program
hello world
welcome to the lab session
```

Object-Oriented Programming (CS F213)

Dr. D.V.N. Siva Kumar
CS&IS Department

BITS Pilani
Hyderabad Campus

Graphical User Interface (GUI)

innovate achieve lead

- GUIs provide an environment in which the **users interact with the application through graphics or images**.
- **GUI has the following advantages:**
  - It is user-friendly. The user need not worry about any commands. Even a layman will be able to work with the application developed using GUI.
  - It adds attraction and beauty to any application by adding pictures, colors menus, animation, etc.
  - It is possible to simulate the real life objects using GUI. For example, a calculator program may actually display a real calculator on the screen. The user feels that he is interacting with a real calculator and he would be able to use it without any difficulty.
  - GUI helps us to create graphical components like push buttons, radio buttons, check boxes, etc., use them effectively in our programs.

Agenda

innovate achieve lead

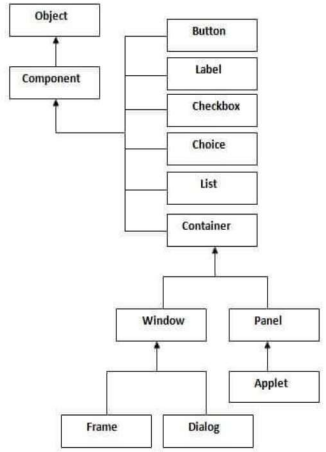
- Abstract Window Toolkit (AWT)
- Event Handling

Java AWT

innovate achieve lead

- **Java AWT** (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.
- **java.awt** package has got classes and interfaces to develop GUI and lets the users interact in a more friendly with the applications.
- Java AWT components are platform-dependent, i.e., components are displayed according to the view of operating system. AWT is heavyweight, i.e., its components are using the resources of OS.

The hierarchy of Java AWT classes



Component

Component class is at the top of AWT hierarchy. It allows a user to interact with your application in various ways—for example, a commonly used control is the push button.

- It is an abstract class that encapsulates all of the attributes of a visual component.
- In general, a Component represents an object which is displayed pictorially on the screen. For example, Button b=new Button();
  Here, b is an object of Button class. If we display this **b** on the screen, it displays a push button. Similarly, radio buttons, check boxes, etc., are all components.
- **Except for menus,** all user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component.**
- It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Layout Managers

- A container has a layout manager **to arrange its components**.
- A container has a **setLayout()** method to set its layout manager: // java.awt.Container

public void setLayout(LayoutManager mgr)

Few layouts (in package java.awt) are provided below:

• **FlowLayout:** Components are arranged from left-to-right inside the container in the order that they are added

• **Gridlayout:** Components are arranged in a grid (matrix) of rows and columns inside the Container

• **BorderLayout:** Here, container is divided into 5 zones: EAST, WEST, SOUTH, NORTH, and CENTER. Components are added using method a Container.add(acomponent, zone), where zone is either BorderLayout.NORTH, and so on

• **BoxLayout:** To arrange the components either vertically or horizontally

Container

- The **Container** class is a subclass of **Component.**
- It has additional methods that allow other **Component** objects (like buttons, textfields, labels etc.) to be nested within it.
- A Container is responsible for laying out (positioning) any components that it contains.
- A ***layout manager*** automatically positions components within a container
- The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window and Panel

Window

The **Window** class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop.

Note: We won't create Window objects directly, but we use a subclass of **Window** called **Frame.**

The window is the container that have no borders and menu bars.

Frame

- It represents what we commonly thought of as a "window".
- It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. It can have other components like button, textfield etc.

Panel

- The **Panel** class is a concrete subclass of **Container.**
- A panel may be thought of as a recursively nestable, concrete screen component.
- Panel is the superclass of **Applet.** An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
- When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.
- It is to be noted that Panel window does not contain a title bar, menu bar, or border.
- The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Note: The applets are deprecated now.
(https://docs.oracle.com/javase/10/docs/api/java/applet/Applet.html)

Working with Frame Windows (Cont..)

- A frame window can also include a standard-style ***menu bar***.
- Each entry in a menu bar activates **a drop-down menu** of options from which the user can choose. This constitutes the *main menu* of an application.
- In general, a **menu bar is positioned at the top of a window**.

Working with Frame Windows

- **Frame** is the type of the window we often create.
- We will use it to crate child windows within top-level or child windows for standalone applications (**runs locally on the device** and do not require anything else to be functional).
- Two of the **Frame's constructors** are shown below:
  - i) Frame() throws headlessException: It creates a standard windows that does not contain a title.
  - ii) Frame(String title) throws HeadlessException: It creates a window with the title specified by **title.**

  Note:
  - We cannot specify the dimensions of the window. We must set the size of the window after it has been created.
  - A **HeadlessException** is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.
- The appearance of a window is determined by a combination of the **controls that it contains and the layout manager used to position them**. The layout manager automates the task of positioning the components within a window.

Setting the Window's Dimensions using Component class methods

- **setSize():** It is used to set the dimensions of the window.
  It is shown here:
  void setSize(int *newWidth*, int *newHeight*)
  Note: The dimensions are specified in terms of pixels.
  void setSize(Dimension *newSize*)

Note:. The coordinates are specified in pixels. The origin of the each window is at the top-left corner and is 0,0

- **getSize( ):** It is used to obtain the current size of a window.
  One of its forms is shown here:
  Dimension getSize( )
  This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

What is a pixel?

- A pixel (short for **picture element**) represents any single point or dot on the screen.
- Any data or pictures displayed on the screen are composed of several dots called **pixels.**
- Monitors accommodate various size of pixels horizontally and vertically.
  - For example, 800 pixels horizontally and 600 pixels vertically.
    The total pixels seen on the screen would be **800*600=480000 pixels**. This is called **screen resolution.** The more the screen resolution the more clarity a picture will have on the screen when displayed.
- Most of laptops use 1024*768 pixels resolution.

Closing a Frame Window using Component class method

- **setVisible():** This method can be used to close the window.
  - When using a frame window, our program must remove that window from the screen when it is closed.
    If it is not the top-level window of your application, this is done by calling **setVisible(false)**.

Hiding and Showing a Window using Component class methods

setVisible(): It is used to make the created window to be visible after a frame window has been created.
Its signature is shown here:
void setVisible(boolean *visibleFlag*)
It is visible if the argument to this method is **true**. Otherwise, it is hidden.

Setting the Window's Title

- **setTitle( )**: It can be used to set the title of a frame window.
  Its general form:
  void setTitle(String *newTitle*)
  Here, *newTitle* is the new title for the window.

Creating a Frame

A frame becomes the basic component in AWT.
The frame has to be created before any other component as all other components can be displayed in a frame.

There are 3 ways to create a frame, which are as follows:

1. Create a Frame class object,
   Frame f=new Frame();
2. Create a Frame class object and pass its title also,
   Frame f=new Frame("My Frame");
3. Create a subclass MyFrame to the Frame class and create object to the subclass as :
   class MyFrame extends Frame
   MyFrame f=new MyFrame();
   Here, As MyFrame is subclass of Frame class, its object f contains a copy of Frame class and hence f represents the frame.

Note: In all these 3 cases, a frame with initial size of 0 pixelz width and 0 pixels height will be created, which won't be visible on screen. Hence, we need specify the size of the frame using **setSize()** method and it then needs to be displayed on screen using **setVisible(true)**.

Java example programs to create a Frame

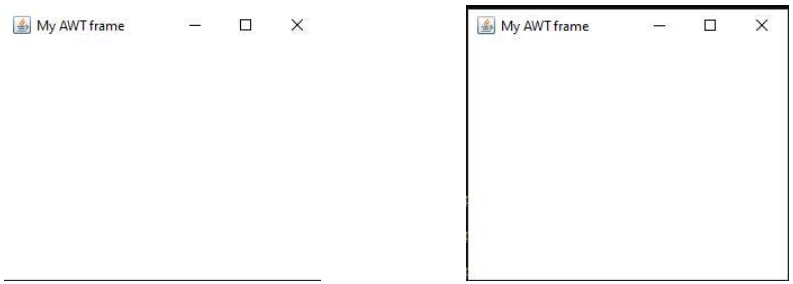
```
import java.awt.*;
class MyFrame{
public static void main(String args[]){
Frame f=new Frame("My AWT frame");
f.setSize(300,250);
f.setVisible(true);
}
}
```

```
import java.awt.*;
class MyFrame extends Frame{
MyFrame(String str){
super(str);
}
public static void main(String args[]){
MyFrame f=new MyFrame("My AWT frame");
f.setSize(300,250);
// 300 is the width and 250 is the height in terms of pixels.
f.setVisible(true);
}
}
```


Uses of a Frame

- With the help of **paint()** method, we can draw some graphical shapes like dots, lines, rectangles, etc., in the frame.
  For example,
  drawLine(int x1, int y1, int x2, int y2), drawRect(int x, int y, int w, int h).
- To display some text in the frame.
- To display pictures and images in the frame.
- To display components like push buttons, radio buttons, etc., in the frame.

Outputs



Note: The frames can be minimized, maximized and resized, but cannot be closed.

The paint( ) Method

- An output to a window typically occurs when the **paint( )** method is called by the run-time system.
- This method is called automatically when a frame is created and displayed.
- This method is defined by **Component** and overridden by **Container** and **Window.** Thus, it is available to instances of **Frame**.
- The **paint( )** method is called each time an AWT-based application's output must be redrawn.
- Whatever the cause, whenever the window must redraw its output, **paint( )** is called.
- The **paint( )** method's general form is shown here:
  void paint(Graphics *context*)
  This parameter will contain the graphics context, which describes the graphics environment in which the program is running. This context is used whenever output to the window is required.

Drawing in the Frame with the help of Graphics methods

Graphics class of java.awt package has the following methods which help us to draw various shapes.

- **drawLine(int x1, int y1, int x2, int y2):** To draw a line connecting (x1,y1) and (x2, y2).
- **drawRect(int x, int y, int w, int h):** to draw a rectangle. The left corner of the rectangle starts at (x,y), the width is w, and the height is h.
- **drawOval(int x, int y, int w, int h):** To draw a circle or ellipse bounded in the region of a rectangle starting at (x,y), with width w and height h.
- **drawArc(int x, int y, int w, int h, int sangle, int aangle):** To draw an arc bounded by a rectangle region of (x,y), with width w and height h. Here, sangle represents the beginning angle measured from 3 o'clock position in the clock. aangle represents the arc angle relative to start angle.

  sangle+aangle=end angle. If the sangle is positive, the arc is drawn counter clock wise and if it is negative, then it is drawn clockwise.

  For example, g.drawArc(50,50,150, 150,90,180)
- drawRoundRect(int x, int y, int w, int h, int arcw, int arch): To draw the outline of a rectangle with rounded corners. The rectangle's top corner starts at (x,y), the width is w, height is h. arcw represents horizontal diameter of the arc at the corner and arch represents vertical diameter of the arc at the corner.

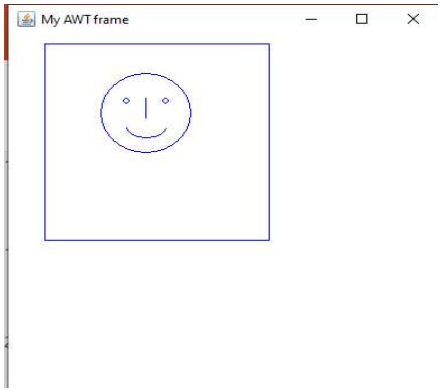
Specifying Colors (class)

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Program to draw a smiling face using the methods of Graphics class

```
import java.awt.*;
class Draw extends Frame{
Draw(String str){
super(str);
}
public void paint(Graphics g){
g.setColor(Color.blue); //sets blue color for drawing
g.drawRect(40,40,200,200);//display a rectangle to contain
drawing
g.drawOval(90,70,80,80);// to draw oval (face)
g.drawOval(110,95,5,5);// to draw a left eye
g.drawOval(145,95,5,5);// to draw a right eye
g.drawLine(130,95,130,115);// to draw a nose
g.drawArc(113,115,35,20,0,-180);// to draw a mouth
}
public static void main(String args[]){
Draw d=new Draw("My AWT frame");
d.setSize(400,400);// 300 is the width and 250 is the height in
terms of pixels.
d.setVisible(true);// to display the frame
}
}
```

Output



Displaying a String using Graphics class

- **drawString( ):** It can be used to output a string to a frame.
- It is a member of the **Graphics** class.
- Its form is shown here:

  void drawString(String *message*, int *x*, int *y*)

  Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0.

  Note: **setColor()** can be used to set text color and **setFont()** to set the text font.

Setting the Foreground and Background Colors

- **setBackground( )**: To set the background color of a frame

  Its general form is:

  void setBackground(Color *newColor*)
- **setForeground():** To set the foreground color of a frame. Its general form is:

  void setForeground(Color *newColor*)

For example, setBackground(Color.green);
setForeground(Color.red);

Java Program display some text

```
import java.awt.*;
import java.awt.event.*;
class MessageFrame extends Frame{
MessageFrame(){
//close the frame when close button is clicked
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we){
System.exit(0);
}
});
}
public void paint(Graphics g){
//set background color for frame
this.setBackground(Color.green);
Font f=new Font("Arial",Font.BOLD+Font.ITALIC,30);
g.setFont(f);
g.setColor(Color.yellow);
g.drawString("Hello",100,100);
}
public static void main(String args[]){
MessageFrame m=new MessageFrame();
m.setSize(400,300);
m.setTitle("This is my text");
m.setVisible(true);
}
}
```

Output

Event Delegation Model

- Event represents a specific action done on a component.
- Event happens when an user interacts with the GUI or due to some other reasons, e.g., timer expires.
- In general, a user expects an action to perform some action when he interacts the GUI by clicking on button, moving mouse pointer, pressing a keyboard, etc..
- When an event is generated on the component, the component will not know about it because it cannot listen to the event.
- To let the component understand that an event is generated on it, we should **add some listener** to the components.
- **A listener is an interface** which listens to an event coming from a component and it will have some abstract methods which need to be implemented.
- When an event is generated by the user on the component, the event is not handled by the component. On the other hand, **the component sends(delegates) that event to the listener attached to it**.
- **The listener will not handle the event. It hands over (delegates) the event to an appropriate method**.
- Finally, the method is executed and the event is handled. **This is called event delegation model.**

Note: All listeners are available in java.awt.event package.

drawPolygon() of Graphics class

- A polygon has several slides and to each side, we need x and y coordinates to connect by a straight line.
- All x and y coordinates can be stored in two arrays as:
  int x[]={40,200,40,100}
  int y[]={40,40,200,200};
- If a polygon is using **drawPolygon(x,y,4),** it starts at (40,40) and connects with (200,40), from a there a line connects it to (40,200), and finally ends at (100, 200).
  here, number 4 indicates four pairs coordinates.

What is event delegation model?

- It represents that when an even is generated by the user on a component, it is delegated to a listener interface and the listener calls a method in response to the event. Finally, the event is handled by the method.

Approaches to handle events:

- Within the same class
- Using other class
- Using an anonymous class

Note: We will be required to place the event handling code in the above places.

Steps involved in Event Delegation Model

1. Attach an appropriate listener to a component. This is done using **addXXXListener()** method. Similarly, we can remove a listener from a component, we can use **removeXXXListener()** method.
2. Implement the methods of the listener, especially the method which handles the event.
3. When an event is generated on the component, then the method in step 2 will be executed and the event is handled.

Advantages of Event Delegation Model

- The component and the action parts can be developed in two separate environments.
- We can modify the code for creating the component without modifying the code for action part of the component. Similarly, we can modify the action part without modifying the code for the component. Thus, we can modify one part without effecting the any modification to other part. This makes debugging and maintenance of code very easy.

Commonly Used Event Classes in java.awt.event

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Commonly Used 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.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Closing the Frame

Three steps are given below that describe how to use delegation model to do this.

1. Attach a listener to the frame component. 'window listener' is most suitable listener to the frame in this context. It can be attached using **addWindowListner()** method.
   For example, f.addWindowListner(WindowLisnter obj)
   Note: We should create an object to the implementation class of interface and pass it to the method.

2. Implement all the methods of the **WindowListener** interface. The following are found in WindowListner interface:

- public void windowActivated(WindowEvent e)
- public void windowClosed(WindowEvent e)
- **public void windowClosing(WindowEvent e)**
- public void windowDeactivated(WindowEvent e)
- public void windowDeiconified(WindowEvent e)
- public void windowIconified(WindowEvent e)
- public void windowOpened(WindowEvent e)

Among the above all, windowClosing() method is suitable to close the frame. So, implementing this is enough.

Closing the Frame (Cont…)

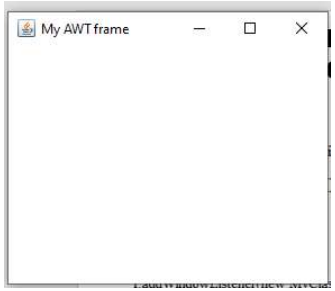
windowClosing() method is implemented as follows:
public void windowClosing(WindowEvent e){
System.exit(0); //close the application.
}

Note: For the remaining methods of WidowListener interface, we need to provide an empty body.

Java program to close the frame when clicking the close button. (Other class is used)

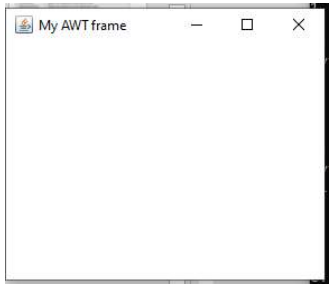
```
import java.awt.*;
import java.awt.event.*;
//the below is aimed to handle closing the window event
class MyFrame extends Frame{
public static void main(String args[]){
//create a frame with title
MyFrame f=new MyFrame();
f.setTitle("My AWT frame");
f.setSize(300,250);
f.setVisible(true);
f.addWindowListener(new MyClass());
}
}
class MyClass implements WindowListener{
public void windowActivated(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowClosing(WindowEvent e){
System.exit(0);
}
public void windowDeactivated(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public  void windowOpened(WindowEvent e){}
}
```

Output:



Closing a window using the same class

```
import java.awt.*;
import java.awt.event.*;
class MyFrame4 extends Frame implements WindowListener{
public void windowActivated(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowClosing(WindowEvent e){
System.exit(0);
}
public void windowDeactivated(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public  void windowOpened(WindowEvent e){}
public static void main(String args[]){
//create a frame with title
MyFrame4 f=new MyFrame4();
f.setTitle("My AWT frame");
f.setSize(300,250);
f.setVisible(true);
f.addWindowListener(new MyClass());
}
}
```



Observation of the Previous Program

The MyClass had to implement all the methods of **WindowListener** interface just for the sake of one method,i.e., **windowClosing().**

This issue can be resolved with the help of **Adapter class**, i.e., with the help of WindowAdapter class in **java.awt.event** package.

Adapter class

An adapter class is an implementation class of a listener which contains all methods implemented with empty body. Adapter class reduces overhead on programming while working with listener interfaces.

For example, **WindowAdapter** is an adapter class of WindowListener interface.

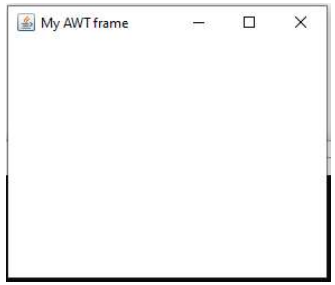
Closing a frame using Anonymous Inner class

Anonymous inner class is an inner class **whose name is not mentioned** and for which only **one object is created.**

Java program to close the frame using an anonymous inner class

```
import java.awt.*;
import java.awt.event.*;
//the below is aimed to handle closing the window event
class MyFrame2 extends Frame{
public static void main(String args[]){
//create a frame with title
MyFrame f=new MyFrame();
f.setTitle("My AWT frame");
f.setSize(300,250);
f.setVisible(true);
f.addWindowListener(new MyClass(){
public void windowClosing(WindowEvent e){
System.exit(0);
}
});
}
}
```

Output



Closing a frame using WindowAdapter class

```
import java.awt.*;
import java.awt.event.*;
//the below is aimed to handle closing the window event
class MyFrame1 extends Frame{
public static void main(String args[]){
//create a frame with title
MyFrame f=new MyFrame();
f.setTitle("My AWT frame");
f.setSize(300,250);
f.setVisible(true);
f.addWindowListener(new MyClass());
}
}
class MyClass extends WindowAdapter{
public void windowClosing(WindowEvent e){
System.exit(0);
}
}
```

Output

AWT Controls

The AWT supports the following types of controls:

- Push buttons
- Labels
- Check boxes
- TextField
- Choice lists
- Lists
- Scroll bars

All the above AWT controls are subclasses of **Component**.

Adding and Removing Controls using Container class methods

- First create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container.**
- **add():** One of the general forms of the add() method is shown below:
  Component add(Component *compRef*)
  Here, ***compRef*** is a reference to an instance of the control that you want to add.
  A reference to the object is returned.
- Once a control has been added, it will automatically be visible whenever its parent window is displayed.

- **remove( )**: This method can be used to remove a control from a window when the control is no longer needed. The general form of remove() is shown below:
  void remove(Component *compRef*)

Labels

- The object of Label class is a component for placing text in a container.
- It is used to display a single line of read only text.
- The text can be changed by an application but a user cannot edit it directly.
- Labels are passive controls that do not support any interaction with the user.
- **Label** defines the following constructors:

- **Label( ) throws HeadlessException** // creates a blank label
- **Label(String *str*) throws HeadlessException** // creates a label that contains the string specified by *str (left-justified)*.
- **Label(String *str*, int *how*) throws HeadlessException** // It creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

Methods of Label:
- void **setText(String *str*):** To set or change the text in a label.
- String **getText( ):** To obtain the label text.

Responding to Controls

- Except for labels, which are passive, all other controls generate events when they are accessed by the user.

HeadlessException: AWT controls throw when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).
- We use this exception to write code that can adapt to non-interactive environments.

Java Program to demonstrate Label

```
import java.awt.*;
import java.awt.event.*;

public class LabelDemo extends Frame {
  public LabelDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");

    // Add labels to frame.
    add(one);
    add(two);
    add(three);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    LabelDemo appwin = new LabelDemo();

    appwin.setSize(new Dimension(300, 100));
    appwin.setTitle("LabelDemo");
    appwin.setVisible(true);
  }
}
```

Output

Buttons

- A *push* ***button*** is a component that contains a label and generates an event when it is pressed.
- Push buttons are objects of type **Button**.
- **Button** defines these two constructors:

- **Button( ) throws HeadlessException** // creates an empty button
- **Button(String *str*) throws HeadlessException** // creates a button that contains *str* as a label.

Methods:

- void **setLabel**(String *str*): It can be used to set the label of the button.
- String **getLabel( ):** It can be used to retrieve the label of the button.

Java Program to Demonstrate Button

```
import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends Frame implements ActionListener {
  String msg = "";
  Button yes, no, maybe;

  public ButtonDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create some buttons.
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

    // Add them to the frame.
    add(yes);
    add(no);
    add(maybe);

    // Add action listeners for the buttons.
    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Handle button action events.
  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
    }
    else {
      msg = "You pressed Undecided.";
    }

    repaint();
  }

  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
  }

  public static void main(String[] args) {
    ButtonDemo appwin = new ButtonDemo();

    appwin.setSize(new Dimension(250, 150));
    appwin.setTitle("ButtonDemo");
    appwin.setVisible(true);
  }
}
```

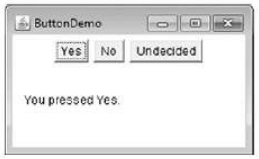

Handling Buttons:

- Each time a button is pressed, an **action event** is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.
- That interface defines the **actionPerformed( )** method, which is called when an event occurs.
- An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button.

- **getActionCommand()**: It is used to obtain label of the button that helps to determine which button has been pressed.

Note: We can also determine which button has been pressed by comparing the object obtained from the **getSource( )** method to the button objects that you added to the window

Output



Check Boxes

- A *check box* is a control that is used to turn an **option on or off**.
- It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- We can change the state of a check box by clicking on it.
- Check boxes can be used individually or as part of a group.
- Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

- Checkbox( ) throws HeadlessException
- Checkbox(String *str*) throws HeadlessException
- Checkbox(String *str*, boolean *on*) throws HeadlessException
- Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
- Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

Handling Check Boxes

- An item event is generated each time a check box is selected or deselected.
- This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.
- Each listener implements the **ItemListener** interface. This interface defines the **itemStateChanged( )** method, which is called when this event is generated.
- An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

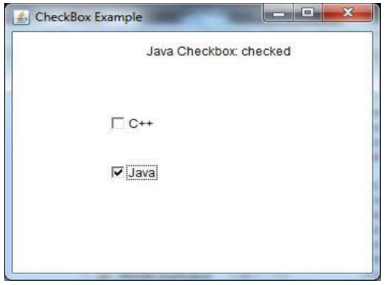
Few Methods of Checkbox

- **boolean getState( ):** To retrieve the current state of a check box.

- **void setState(boolean *on*)**: To set the state of the check box.
- **String getLabel( ):** To obtain the current label associated with a check box.
- **void setLabel(String *str*):** To set the label.

T Checkbox

```
import java.awt.*;
import java.awt.event.*;
public class CheckboxExample
{
    CheckboxExample(){
      Frame f= new Frame("CheckBox Example");
      final Label label = new Label();
      label.setAlignment(Label.CENTER);
      label.setSize(400,100);
      Checkbox checkbox1 = new Checkbox("C++");
      checkbox1.setBounds(100,100, 50,50);
      Checkbox checkbox2 = new Checkbox("Java");
      checkbox2.setBounds(100,150, 50,50);
      f.add(checkbox1); f.add(checkbox2); f.add(label);
      checkbox1.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
            label.setText("C++ Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));
          }
        });
      checkbox2.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {

            label.setText("Java Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));

          }
        });
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
  new CheckboxExample();
}
}
```



CheckboxGroup

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called **radio buttons**, because they act like the station selector on a car radio—only one station can be selected at any one time.
- To create a set of mutually exclusive check boxes, We must first define the group to which they will belong and then specify that group when you construct the check boxes.
- Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.
- We can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**
- We can set a check box by calling setSelectedCheckbox( ).

TextField

- The TextField class implements a single-line text-entry area, usually called an edit control.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- **TextField defines the following constructors:**
- TextField( ) throws HeadlessException // a default text field.
- TextField(int numChars) throws HeadlessException // creates a text field that is numChars characters wide.
- TextField(String str) throws HeadlessException //creates a text field that is numChars characters wide.
- TextField(String str, int numChars) throws HeadlessException // initializes a text field and sets its width

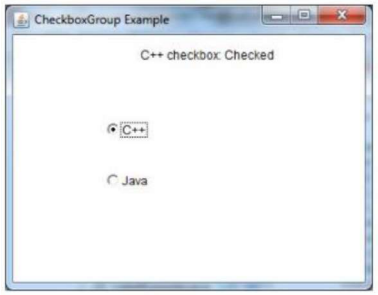
Few Methods of TextField:

- **String getText( ) :** To obtain the string currently contained in the text field
- **void setText(String str):** To obtain the string currently contained in the text field


```
import java.awt.*;
import java.awt.event.*;
public class CheckboxGroupExample
{
   CheckboxGroupExample(){
    Frame f= new Frame("CheckboxGroup Example");
    final Label label = new Label();
    label.setAlignment(Label.CENTER);
    label.setSize(400,100);
    CheckboxGroup cbg = new CheckboxGroup();
    Checkbox checkBox1 = new Checkbox("C++", cbg, false);
    checkBox1.setBounds(100,100, 50,50);
    Checkbox checkBox2 = new Checkbox("Java", cbg, false);
    checkBox2.setBounds(100,150, 50,50);
    f.add(checkBox1); f.add(checkBox2); f.add(label);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    checkBox1.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        label.setText("C++ checkbox: Checked");
      }
    });
    checkBox2.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        label.setText("Java checkbox: Checked");
      }
    });
  }

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

Program to demonstrate CheckboxGroup

Output



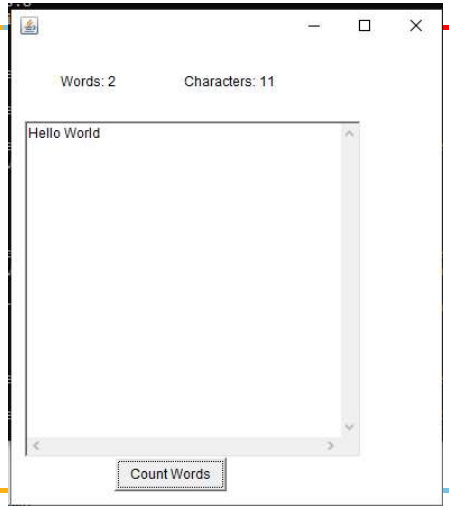
Java program to demonstrate TextField with Button

```
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
b.addActionListener(this);//passing current instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

Few More Controls with their Registration methods for event handling

- **TextArea**
  - public void addTextListener(TextListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}

Output



Java Program to Demonstrate TextArea

The object of a TextArea class is a multi line region that displays text. It allows the editing of multiple line text.

```
public class TextAreaExample2 extends Frame implements ActionListener {
// creating objects of Label, TextArea and Button class.
Label l1, l2;
TextArea area;
Button b;
// constructor to instantiate
TextAreaExample2() {
// instantiating and setting the location of components on the frame
    l1 = new Label();
    l1.setBounds(50, 50, 100, 30);
    l2 = new Label();
    l2.setBounds(160, 50, 100, 30);
    area = new TextArea();
    area.setBounds(20, 100, 300, 300);
    b = new Button("Count Words");
    b.setBounds(100, 400, 100, 30);
// adding ActionListener to button
    b.addActionListener(this);
// adding components to frame
    add(l1);
add(l2);
add(area);
add(b);
// setting the size, layout and visibility of frame
    setSize(400, 450);
    setLayout(null);
    setVisible(true);
}
// generating event text area to count number of words and characters
public void actionPerformed(ActionEvent e) {
    String text = area.getText();
    String words[]=text.split("\\s");
    l1.setText("Words: "+words.length);
    l2.setText("Characters: "+text.length());
}
// main method
public static void main(String[] args) {
    new TextAreaExample2();
}
}
```


Java AWT Choice

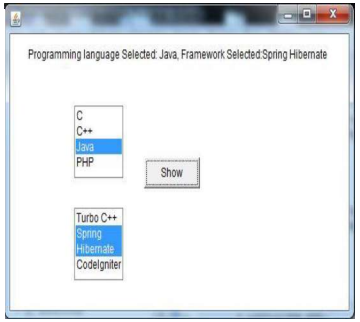
The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

```
import java.awt.*;
import java.awt.event.*;
public class ChoiceExample
{
    ChoiceExample(){
    Frame f= new Frame();
    final Label label = new Label();
    label.setAlignment(Label.CENTER);
    label.setSize(400,100);
    Button b=new Button("Show");
    b.setBounds(200,100,50,20);
    final Choice c=new Choice();
    c.setBounds(100,100, 75,75);
    c.add("C");
    c.add("C++");
    c.add("Java");
    c.add("PHP");
    c.add("Android");
    f.add(c);f.add(label); f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
      String data = "Programming language Selected: "+ c
.getItem(c.getSelectedIndex());
        label.setText(data);
      }
      });
      }
public static void main(String args[])
{
  new ChoiceExample();
}
}
```

Java AWT List

The object of List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It inherits Component class..

```
import java.awt.*;
import java.awt.event.*;
public class ListExample
{
    ListExample(){
      Frame f= new Frame();
      final Label label = new Label();
      label.setAlignment(Label.CENTER)
      label.setSize(500,100);
      Button b=new Button("Show");
      b.setBounds(200,150,80,30);
      final List l1=new List(4, false);
      l1.setBounds(100,100, 70,70);
      l1.add("C");
      l1.add("C++");
      l1.add("Java");
      l1.add("PHP");
      final List l2=new List(4, true);
      l2.setBounds(100,200, 70,70);
      l2.add("Turbo C++");
      l2.add("Spring");
      l2.add("Hibernate");
      l2.add("CodeIgniter");
      f.add(l1); f.add(l2); f.add(label); f.add(b);
      f.setSize(450,450);
      f.setLayout(null);
      f.setVisible(true);
      b.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
        String data = "Programming language Selected: "+l1.getIte
m(l1.getSelectedIndex());
        data += ", Framework Selected:";
        for(String frame:l2.getSelectedItems()){
              data += frame + " ";
        }
        label.setText(data);
        }
       });
}
public static void main(String args[])
{
  new ListExample();
}
}
```



Important note about AWT GUI

GUIs created using AWT are seldom used in today's world. However, it is important for the following reasons:

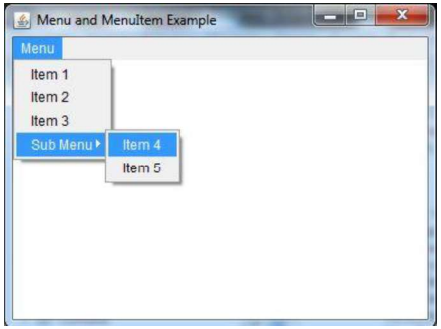
- Most of the information and many of the techniques related to controls and event handling are generalizable to Swing.
- The layout managers of AWT are also used by Swing.
- For some small applications, the AWT components might be the appropriate choice.
- We may encounter legacy code that uses the AWT.

Therefore, a basic understanding of the AWT is important for all Java programmers and hence, covered fully in lectures.

Java AWT MenuItem and Menu

- The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.
- The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

```
import java.awt.*;
class MenuExample
{
    MenuExample(){
      Frame f= new Frame("Menu and MenuItem Example");
      MenuBar mb=new MenuBar();
      Menu menu=new Menu("Menu");
      Menu submenu=new Menu("Sub Menu");
      MenuItem i1=new MenuItem("Item 1");
      MenuItem i2=new MenuItem("Item 2");
      MenuItem i3=new MenuItem("Item 3");
      MenuItem i4=new MenuItem("Item 4");
      MenuItem i5=new MenuItem("Item 5");
      menu.add(i1);
      menu.add(i2);
      menu.add(i3);
      submenu.add(i4);
      submenu.add(i5);
      menu.add(submenu);
      mb.add(menu);
      f.setMenuBar(mb);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
    public static void main(String args[])
    {
    new MenuExample();
    }
    }
```



Swing: Why, when you already have AWT?

- **Swing was a response to deficiencies present** in Java's original GUI subsystem: the Abstract Window Toolkit.

AWT Issues:

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

MVC Architecture

A visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

Swing Components

- Swing components are derived from the **JComponent** class.
- **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel.
- **JComponent** inherits the AWT classes **Container** and **Component**. Due to this, Swing component is built on and compatible with an AWT component.
- All of Swing's components are represented by classes defined within the package **javax.swing**.

Swing Components and Containers

- A Swing GUI consists of two key items: ***components* and *containers***.
- A *component* is an independent **visual control**, such as a push button or slider. A container holds a **group of components**. Thus, a container is a special type of component that is designed to hold other components.
- Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container.

Swing Classes for Components

JApplet (Deprecated)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
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	JWindow		

Swing Containers

Swing defines two types of containers.
1. **The first are top-level containers**: **JFrame**, **JApplet**, **JWindow**, and **JDialog**.
-These containers do not inherit **JComponent**. However, they inherit the AWT classes **Component** and **Container**.
-As the name implies, a top-level container must be at the top of a containment hierarchy. The top level containers are heavyweight.
-A top-level container is not contained within any other container.
-The one most commonly used for applications is **JFrame**.

Swing Packages

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

Swing Containers (Cont…)

2**. Lightweight containers**.
- Lightweight containers *do* inherit **Jcomponent.**
  An example of a lightweight container is **JPanel**, which is a general-purpose container.
- Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.
- Thus, we can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

A Simple Swing Application

```
import javax.swing.*;

class SwingDemo {

  SwingDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("A Simple Swing Application");

    // Give the frame an initial size.
    jfrm.setSize(275, 100);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a text-based label.
    JLabel jlab = new JLabel(" Swing means powerful GUIs.");

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

Output: