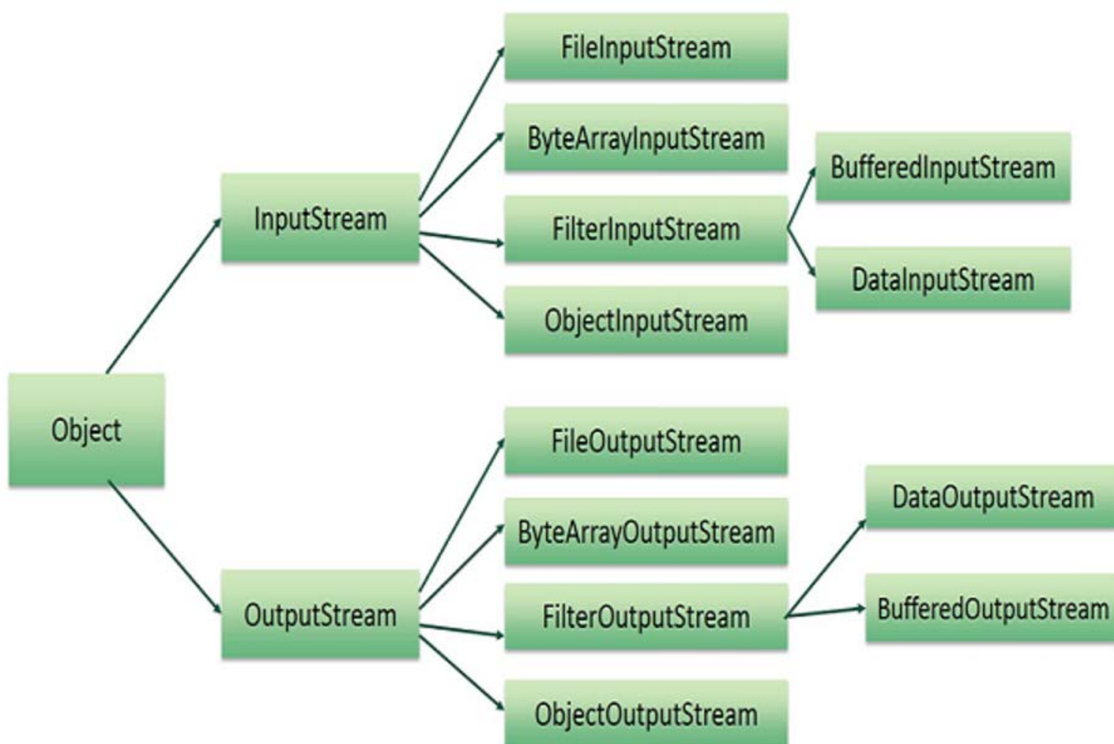
# Week 8-1: I/O Library

## Part1. Reading and Writing Files

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.




- **FileInputStream**

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file -

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows -

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

- **FileOutputStream**

This stream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output. Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file -

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows -

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

- **Example**

Following is the example to demonstrate InputStream and OutputStream

```
import java.io.*;
public class fileStreamTest {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      }catch(IOException e) {
         System.out.print("Exception");
      }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

# Week 8-2: Containers

## Part1. Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

### - TreeMap

The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.

You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in an ascending key order.

The following program illustrates several of the methods supported by this collection −

```java
import java.util.*;
public class TreeMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
}
```

```
Ayan: 1378.0

Daisy: 99.22

Mahnaz: 123.22

Qadir: -19.08

Zara: 3434.34


Zara's new balance: 4434.34
```

- **TreeSet**

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.
Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

The following program illustrates several of the methods supported by this collection −

```
import java.util.*;
public class TreeSetDemo {

   public static void main(String args[]) {
      // Create a tree set
      TreeSet ts = new TreeSet();

      // Add elements to the tree set
      ts.add("C");
      ts.add("A");
      ts.add("B");
      ts.add("E");
      ts.add("F");
      ts.add("D");
      System.out.println(ts);
   }
}
```

```
[A, B, C, D, E, F]
```