Exploring java.lang

- Here, we will discuss the classes and interfaces defined by ***java.lang.***
- As you know, **java.lang** is automatically imported into all programs.
- It contains classes and interfaces that are fundamental to virtually all of Java programming.
- It is Java's most widely used package.
- Several of the classes contained in **java.lang** contain deprecated methods, most dating back to Java 1.0.
- These deprecated methods are still provided by Java to support an ever-shrinking pool of legacy code and are not recommended for new code.
- Most of the deprecations took place prior to Java SE 6

Classes provided by ***java.lang***

Boolean	Compiler	Integer	Package	String	Thread
Byte	Double	Long	Process	StringBuffer	ThreadGroup
Character	Enum	Math	ProcessBuilder	StringBuilder	ThreadLocal
Class	Float	Number	System	StrictMath	Throwable
ClassLoader	Runtime	Object	Short	Void	RuntimePermission
SecurityManager		StackTraceElement		InheritableThreadLocal	

Interfaces provided by ***java.lang***

Appendable	Comparable	CharSequence	Runnable
Cloneable	Iterable	Readable	

Primitive Type Wrappers

- Java uses primitive types, such as **int** and **char**, for performance reasons.
- These data types are not part of the object hierarchy.
- They are passed by value to methods and cannot be directly passed by reference.
- Also, there is no way for two methods to refer to the *same instance* of an **int**.
- At times, you will need to create an object representation for one of these primitive types.
- For example, there are collection classes that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class.
- To address this need, Java provides classes that correspond to each of the primitive types.
- In essence, these classes encapsulate, or *wrap,* the primitive types within a class.
- Thus, they are commonly referred to as ***type wrappers***.

Number

- The abstract class **Number** defines a superclass that is implemented by the classes that wrap the numeric types **byte**, **short**, **int**, **long**, **float**, and **double**.

- **Number** has abstract methods that return the value of the object in each of the different number formats.

- For example, **doubleValue( )** returns the value as a **double**, **floatValue( )** returns the value as a **float**, and so on.
- These methods are shown here:
  - byte byteValue( )
  - double doubleValue( )
  - float floatValue( )
  - int intValue( )
  - long longValue( )
  - short shortValue( )

- The values returned by these methods can be rounded.

- **Number** has six concrete subclasses that hold explicit values of each numeric type: **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long**.

Note:
- Students are advised to refer text book for a list of methods provided by each of the above classes.
- Few important methods are discussed hereunder.

Understanding isInfinite( ) and isNaN( ) :

- **Float** and **Double** provide the methods **isInfinite( )** and **isNaN( )**, which help when manipulating two special **double** and **float** values.

- These methods test for two unique values defined by the IEEE floating-point specification: infinity and NaN (not a number).

- **isInfinite( )** returns **true** if the value being tested is infinitely large or small in magnitude.

- **isNaN( )** returns **true** if the value being tested is not a number.

```
class InfNaN
{
    public static void main(String args[])
    {
          Double d1 = new Double(1/0.0);
          Double d2 = new Double(0/0.0);
          System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
          System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
    }
}
```

Output:

```
Infinity: true, false
NaN: false, true
```

Converting Numbers to and from Strings

- One of the most common programming chores is converting the string representation of a number into its internal, binary format.
- Java provides an easy way to accomplish this.
- The **Byte**, **Short**, **Integer**, and **Long** classes provide the **parseByte( )**, **parseShort( )**, **parseInt()**, and **parseLong( )** methods, respectively.
- These methods return the **byte**, **short**, **int**, or **long** equivalent of the numeric string with which they are called.
- Similar methods also exist for the **Float** and **Double** classes.
- The following program demonstrates **parseInt( )**. It sums a list of integers entered by the user. It reads the integers using **readLine( )** and uses **parseInt( )** to convert these strings into their **int** equivalents.

```
import java.io.*;
class ParseDemo
{
    public static void main(String args[]) throws IOException
    {
          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
          String str;
          int i, sum=0;

          System.out.println("Enter numbers, 0 to quit.");

          do
          {          str = br.readLine();
                     try
                     {
                              i = Integer.parseInt(str);
                     } catch(NumberFormatException e)
                     {
                              System.out.println("Invalid format");
                              i = 0;
                     }
                     sum += i;
          } while(i != 0);
          System.out.println("sum is: " + sum);
    }
```

- To convert a whole number into a decimal string, use the versions of **toString( )** defined in the **Byte**, **Short**, **Integer**, or **Long** classes.

- The **Integer** and **Long** classes also provide the methods **toBinaryString( )**, **toHexString()**, and **toOctalString( )**, which convert a value into a binary, hexadecimal, or octal string, respectively.

- The following program demonstrates binary, hexadecimal, and octal conversion:

```
class StringConversions
{
    public static void main(String args[])
    {
            int num = 19648;
            System.out.println(num + " in binary: " +  Integer.toBinaryString(num));
            System.out.println(num + " in octal: " + Integer.toOctalString(num));
            System.out.println(num + " in hexadecimal: " + Integer.toHexString(num));
    }
}
```

The output of this program is shown here:

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

System

- The **System** class holds a collection of static methods and variables.
- The standard input, output, and error output of the Java run time are stored in the **in**, **out**, and **err** variables.
- Many of the methods throw a **SecurityException** if the operation is not permitted by the security manager.

Note: Refer textbook for list of methods supported by ***System*** class.

Object

- **Object** is a super class of all other classes.
- **Object** defines the various methods which are available to every object.

Note: Refer textbook for list of methods supported by ***Object*** class.

Class

- **Class** encapsulates the run-time state of an object or interface.
- Objects of type **Class** are created automatically, when classes are loaded.
- You cannot explicitly declare a **Class** object.
- Generally, you obtain a **Class** object by calling the **getClass()** method defined by **Object**.
- **Class** is a generic type that is declared as shown here:
  class Class<T>
- Here, **T** is the type of the class or interface represented.
- The methods defined by **Class** are often useful in situations where run-time type information about an object is required.
-
- The methods are provided that allow you to determine additional information about a particular class, such as its public constructors, fields, and methods.
- Among other things, this is important for the Java Beans functionality.

```
class X
{    int a; }
class Y extends X
{    double c; }

class RTTI
{    public static void main(String args[])
     {        X x = new X();
              Y y = new Y();

              Class<?> clObj;
              clObj = x.getClass();
              System.out.println("x is object of type: " + clObj.getName());

              clObj = y.getClass();
              System.out.println("y is object of type: " + clObj.getName());

              clObj = clObj.getSuperclass();
              System.out.println("y's superclass is " + clObj.getName());
     }
}
```

Output:

```
x is object of type: X
y is object of type: Y
y’s superclass is X
```

Class Loader

- The abstract class **ClassLoader** defines how classes are loaded.
- Your application can create subclasses that extend **ClassLoader**, implementing its methods.
- Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system.
- However, this is not something that you will normally need to do.

Math

- The **Math** class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods.
- **Math** defines two **double** constants: **E** (approximately 2.72) and **PI** (approximately 3.14).
- The trigonometric functions includes *sin, cos, tan, asin* (sine-inverse), *acos, atan, sinh* (sine-hyperbolic), *cosh, tanh* etc.
- The exponential functions include *cbrt, exp, log, pow, sqrt* etc.
- The rounding functions include *abs, ceil, floor, max, min, round* etc.

Thread, ThreadGroup and Runnable

- The **Runnable** interface and the **Thread** and **ThreadGroup** classes support multithreaded programming.

The Runnable Interface

- The **Runnable** interface must be implemented by any class that will initiate a separate thread of execution.
- **Runnable** only defines one abstract method, called **run( )**, which is the entry point to the thread.
- It is defined like this:

```
void run( )
```

- Threads that you create must implement this method.

Thread

- **Thread** creates a new thread of execution. It defines the following commonly used constructors:
  - Thread( )
  - Thread(Runnable *threadOb*)
  - Thread(Runnable *threadOb*, String *threadName*)
  - Thread(String *threadName*)
  - Thread(ThreadGroup *groupOb*, Runnable *threadOb*)
  - Thread(ThreadGroup *groupOb*, Runnable *threadOb*, String *threadName*)
  - Thread(ThreadGroup *groupOb*, String *threadName*)
- Here, *threadOb* is an instance of a class that implements the **Runnable** interface and defines where execution of the thread will begin.
- The name of the thread is specified by *threadName*.
- When a name is not specified, one is created by the Java Virtual Machine.
- *groupOb* specifies the thread group to which the new thread will belong.
- When no thread group is specified, the new thread belongs to the same group as the parent thread.
- The following constants are defined by **Thread**: MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY.

ThreadGroup

- **ThreadGroup** creates a group of threads.
- It defines these two constructors:
  - ThreadGroup(String *groupName*) : creates a new group that has the current thread as its parent
  - ThreadGroup(ThreadGroup *parentOb*, String *groupName*) : the parent is specified by *parentOb*
- For both forms, *groupName* specifies the name of the thread group.
- Thread groups offer a convenient way to manage groups of threads as a unit.
- This is particularly valuable in situations in which you want to suspend and resume a number of related threads.
- For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file.
- If printing is aborted, you will want an easy way to stop all threads related to printing.
- Thread groups offer this convenience.

Throwable

- The **Throwable** class supports Java's exception-handling system and is the class from which all exception classes are derived.

The Collections Framework

- The Collections Framework is one of the most important subsystems of Java, included within ***java.util*** package.
- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

Collections Overview

- Collections were added to J2SE 1.2.
- Prior to this, there were ad hoc classes such as Dictionary, Stack etc. but they lacked uniform usage.
- Moreover, those classes were not extendable.
- To avoid such problems, Collection class and interfaces have been introduced.

- The Collections Framework was designed to meet several goals.
- First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
- We need not code these data structures manually.
- Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Third, extending and/or adapting a collection had to be easy.
- Toward this end, the entire Collections Framework is built upon a set of standard interfaces.
- We can use several standard implementations (such as **LinkedList**, **HashSet**, and **TreeSet**) as they are.
- You may also implement your own collection, if you choose.
- Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier.
- Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

- *Algorithms* are another important part of the collection mechanism.
- Algorithms operate on collections and are defined as static methods within the **Collections** class.
- The algorithms provide a standard means of manipulating collections.
- Another item closely associated with the Collections Framework is the **Iterator** interface.
- An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.
- Thus, an iterator provides a means of *enumerating the contents of a collection*.
- Because each collection implements **Iterator**, the elements of any collection class can be accessed through the methods defined by **Iterator**.
- Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

Collections Interfaces

- The Collections Framework defines several interfaces.
- Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes.
- Put differently, the concrete classes simply provide different implementations of the standard interfaces.
- To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional.
- The optional methods enable you to modify the contents of a collection.
- Collections that support these methods are called *modifiable*.
- Collections that do not allow their contents to be changed are called *unmodifiable*.
- If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown.
- All the built-in collections are modifiable.

Interfaces available in Collections Framework

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. (Added by Java SE 6.)
List	Extends Collection to handle sequences (lists of objects).
NavigableSet	Extends SortedSet to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
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.

The 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 a generic interface that has this declaration:
  interface Collection<E>
- Here, **E** specifies the type of objects that the collection will hold.

- **Collection** extends the **Iterable** interface.

- This means that all collections can be cycled through by use of the for-each style **for** loop.

- **Collection** declares the core methods that all collections will have.

Few important Methods available in Collection Interface

Method	Description
boolean add(E obj)	Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection and the collection does not allow duplicates.
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 equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
boolean isEmpty( )	Returns true if the invoking collection is empty. Otherwise, returns false.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object obj)	Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
int size( )	Returns the number of elements held in the invoking collection.

The List Interface

- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- **List** is a generic interface that has this 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 some of its own methods.

The Queue Interface

- The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list.

- However, there are types of queues in which the ordering is based upon other criteria.

- **Queue** is a generic interface that has this declaration:

  interface Queue<E>

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

- The methods defined by Queue are shown in the following Table.

Methods available in Queue Interface

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
boolean offer(E obj)	Attempts to add obj to the queue. Returns true if obj was added and false otherwise.
E peek( )	Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

The Collection Classes

- Here, we will examine the standard classes that implement various interfaces discussed earlier.
- Some of the classes provide full implementations that can be used directly.
- Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections.
- The standard collection classes are summarized in the following table:

Collection Classes

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.
AbstractSequentialLi st	Extends AbstractList for use by a collection that uses sequential rather than randomaccess 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.
TreeSet	Implements a set stored in a tree. Extends AbstractSet.

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.
- In Java, standard arrays are of a fixed length.
- After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- But, sometimes, you may not know until run time precisely how large an array you need.
- To handle this situation, the Collections Framework defines **ArrayList**.
- In essence, an **ArrayList** is a variable-length array of object references.
- That is, an **ArrayList** can dynamically increase or decrease in size.

- 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.
- **ArrayList** has the constructors shown here:
- ArrayList( ) : builds an empty array list.
- ArrayList(Collection<? extends E> *c*): builds an array list
  that is initialized with the elements of the collection *c*.
- 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.

```
import java.util.*;

class ArrayListDemo
{
    public static void main(String args[])
    {
        ArrayList<String> al = new ArrayList<String>();
        System.out.println("Initial size of al: " + al.size());

        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());

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

        al.remove("F");
        al.remove(2);
        System.out.println("Size of al after deletions: " + al.size());
        System.out.println("Contents of al: " + al);
    }
}
```

Output:

```
Initial size of al: 0
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]
```

- Notice that **a1** starts out empty and grows as elements are added to it.
- When elements are removed, its size is reduced.
- Although the capacity of an **ArrayList** object increases automatically as objects are stored in it, you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity()**.
- By increasing its capacity at the beginning, you can prevent several reallocations later.
- Since reallocations are costly in terms of time, prevention will improve the performance.
- The signature for **ensureCapacity( )** is shown here:

  void ensureCapacity(int *cap*)
- Here, *cap* is the new capacity.
- Conversely, if you want 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, call **trimToSize( )**, shown here:

  void trimToSize( )

Obtaining an Array from an ArrayList

- When working with **ArrayList**, you will sometimes want to obtain an actual array that contains the contents of the list.

- You can do this by calling **toArray( )**, which is defined by **Collection**.

- Several reasons for converting a collection into an 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 (older versions of Java code) that does not understand collections

- There are two versions of **toArray()**:
  - Object[ ] toArray( )   : returns an array of **Object**

  - <T> T[ ] toArray(T *array*[ ]) : returns an array of elements that have the same type as **T**. This form is more convenient because it returns the proper type of array.

```
import java.util.*;

class ToArrayDemo
{
    public static void main(String args[])
    {
        ArrayList<Integer> al = new ArrayList<Integer>();

        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);
        System.out.println("Contents of al: " + al);

        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);
        int sum = 0;

        for(int i : ia)    //note autoboxing
                sum += i;
        System.out.println("Sum is: " + sum);
    }
}
```

Output:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

The LinkedList Class

- The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.
- It provides a linked-list data structure.
- **LinkedList** is a generic class that has this declaration:
  class LinkedList<E>
- Here, **E** specifies the type of objects that the list will hold.
- **LinkedList** has the two constructors shown here:
  - LinkedList( ) : builds an empty linked list
  - LinkedList(Collection<? extends E> *c*) : builds a linked list that is initialized with the elements of the collection *c*.

```
import java.util.*;

class LinkedListDemo
{   public static void main(String args[])
    {
        LinkedList<String> ll = new LinkedList<String>();

        ll.add("F");   ll.add("B");
        ll.add("D");   ll.add("E");
        ll.add("C");   ll.addLast("Z");
        ll.addFirst("A");  ll.add(1, "A2");

        System.out.println("Original contents of ll: " + ll);

        ll.remove("F");
        ll.remove(2);
        System.out.println("Contents of ll after deletion: " + ll);

        ll.removeFirst();
        ll.removeLast();
        System.out.println("ll after deleting first and last: " + ll);
        String val = ll.get(2);
        ll.set(2, val + " Changed");
        System.out.println("ll after change: " + ll);
    }
```

Output:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

The HashSet Class

- **HashSet** extends **AbstractSet** and implements the **Set** interface.
- It creates a collection that uses a hash table for storage.
- **HashSet** is a generic class that has this declaration:
  class HashSet<E>
- Here, **E** specifies the type of objects that the set will hold.
- A hash table stores information by using a mechanism called hashing.
- In *hashing*, the informational content of a key is used to determine a 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.
- The transformation of the key into its hash code is performed automatically— you never see the hash code itself.
- Also, your code can't directly index the hash table.
- The advantage of hashing is that it allows the execution time of **add( )**, **contains()**, **remove( )**, and **size( )** to remain constant even for large sets.

- The following constructors are defined:
  - HashSet() : constructs a default hash set
  - HashSet(Collection<? extends E> *c*) :initializes the hash set by using the elements of *c*
  - HashSet(int *capacity*) : initializes the capacity of the hash set to *capacity*. (The default capacity is 16.)
  - HashSet(int *capacity*, float *fillRatio*) : initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.
- It is important to note that **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.

```
import java.util.*;

class HashSetDemo
{
    public static void main(String args[])
    {
        HashSet<String> hs = new HashSet<String>();

        hs.add("A");
        hs.add("B");
        hs.add("C");
        hs.add("D");
        hs.add("E");
        hs.add("F");
        System.out.println(hs);
    }
}
```

Output:

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

Note: Output may vary.

The TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.
- It creates a collection that uses a tree for storage.
- Objects are stored in sorted, 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.
- **TreeSet** is a generic class that has this declaration:

  ```
  class TreeSet<E>
  ```
- Here, **E** specifies the type of objects that the set will hold.

- **TreeSet** has the following constructors:
  - TreeSet( ) : constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.
  - TreeSet(Collection<? extends E> *c*) : builds a tree set that contains the elements of *c*.
  - TreeSet(Comparator<? super E> *comp*) : constructs an empty tree set that will be sorted according to the comparator specified by *comp*.
  - TreeSet(SortedSet<E> *ss*) : builds a tree set that contains the elements of *ss*.

```
import java.util.*;

class TreeSetDemo
{
    public static void main(String args[])
    {
          TreeSet<String> ts = new TreeSet<String>();
          ts.add("F");
          ts.add("C");
          ts.add("D");
          ts.add("E");
          ts.add("A");
          ts.add("B");
          System.out.println(ts);
    }
}
```

Output:

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

Note:

- As **TreeSet** stores its elements in a tree, they are automatically arranged in sorted order (You can think of in-order traversal of binary search tree)