# 10.2: Basics of collection framework and Scanner class

IT1406

Level I - Semester 1

## 10.2.1. Using Scanner class

- Scanner reads formatted input and converts it into its binary form.
- It can be used to read input from the console, a file, a string, or any source that implements the *Readable* interface or *ReadableByteChannel***.**
- For example, you can use Scanner to read a number from the keyboard and assign its value to a variable.
- In general, a Scanner reads tokens from the underlying source that you specified when the Scanner was created.
- a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default.

## 10.2.1. Using Scanner class

- Scanner defines the constructors shown in the following Table.

| Method | Description |
|---|---|
| Scanner(File *form*) throws FileNotFoundException | Creates a scanner that uses the file specified by *from* as a source for input |
| Scanner(File *from*, String *charset*) throws FileNotFoundException | Creates a Scanner that uses the file specified by *from* with the encoding specified by charset as a source for input |
| Scanner(InputStream *from*) | Creates a Scanner that uses the stream specified by *from* as a source for input |
| Scanner(InputStream *from*, String *charset*) | Creates a Scanner that uses the stream specified by from with the encoding specified by *charset* as a source for input. |
| Scanner(Path *from*) throws IOException | Creates a Scanner that uses the file specified by *from* as a source for input |

## 10.2.1. Using Scanner class

- Scanner defines the constructors shown in the following Table.

| Method | Description |
|---|---|
| Scanner(Path *from*, String *charset*) throws IOException | Creates a Scanner that uses the file specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(Readable *from*) | Creates a Scanner that uses the Readable object specified by *from* as a source for input |
| Scanner (ReadableByteChannel *from*) | Creates a Scanner that uses the ReadableByteChannel specified by *from* as a source for input. |
| Scanner(ReadableByteChannel *from*, String *charset*) | Creates a Scanner that uses the ReadableByteChannel specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(String *from*) | Creates a Scanner that uses the string specified by *from* as a source for input. |

## 10.2.1. Using Scanner class

- To use Scanner, follow this procedure:

1. Determine if a specific type of input is available by calling one of Scanner's **hasNextX** methods, where X is the type of data desired.
2. If input is available, read it by calling one of Scanner's **nextX** methods.
3. Repeat the process until input is exhausted.
4. Close the Scanner by calling close( ).

- Scanner defines two sets of methods that enable you to read input.
- The first are the **hasNextX** methods, which determine if the specified type of input is available.
- For example, calling hasNextInt( ) returns true only if the next token to be read is an integer.
- If the desired data is available, then you read it by calling one of Scanner's **nextX** methods.

## 10.2.1. Using Scanner class

```
Scanner conin = new Scanner(System.in);
int i;
while(conin.hasNextInt()) {
        i = conin.nextInt();
}
```

- The while loop stops as soon as the next token is not an integer.
- Thus, the loop stops reading integers as soon as a non-integer is encountered in the input stream.
- If a next method cannot find the type of data it is looking for, it throws an **InputMismatchException**.
- A **NoSuchElementException** is thrown if no more input is available.
- For this reason, it is best to first confirm that the desired type of data is available by calling a method before calling its corresponding next method

## 10.2.1. Using Scanner class –Example 01

```
// Use Scanner to compute an average of the values.
import java.util.*;
class AvgNums {
  public static void main(String args[]) {
      Scanner conin = new Scanner(System.in);
      int count = 0;
      double sum = 0.0;
      System.out.println("Enter numbers to average.");


     while(conin.hasNext()) {
         if(conin.hasNextDouble()) {
              sum += conin.nextDouble();
             count++;
         }
         else {
             String str = conin.next();
             if(str.equals("done")) break;
             else {
                 System.out.println("Data format error.");
                 return;
             }
          }
      }
      conin.close();
      System.out.println("Average is " + sum / count);
   }
}
```

## 10.2.1. Using Scanner class –Example 01

- The program reads numbers from the keyboard, summing them in the process, until the user enters the string "done".
- It then stops input and displays the average of the numbers. Here is a sample run:

```
Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65
```

## 10.2.1. Using Scanner class –Example 01

- Notice that the numbers are read by calling nextDouble( ).
- This method reads any number that can be converted into a double value, including an integer value, such as 2, and a floating-point value like 3.4.
- Thus, a number read by nextDouble( ) need not specify a decimal point. This same general principle applies to all next methods. They will match and read any data format that can represent the type of value being requested.

## 10.2.1. Using Scanner class –Example 02

```
import java.util.*;
import java.io.*;

class AvgFile {
    public static void main(String args[])
        throws IOException {
        int count = 0;
        double sum = 0.0;
        FileWriter fout = new FileWriter("test.txt");
        fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
        fout.close();
        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);

    while(src.hasNext()) {
        if(src.hasNextDouble()) {
            sum += src.nextDouble();
            count++;
        }
        else {
          String str = src.next();
          if(str.equals("done")) break;
          else {
                System.out.println("File format error.");
                return;
          }
       }
    }
    src.close();
    System.out.println("Average is " + sum / count);
  }
}
```

## 10.2.1. Using Scanner class –Example 02

Here is the output:

```
Average is 6.2
```

- The preceding program illustrates another important feature of Scanner.
- Notice that the file reader referred to by fin is not closed directly.
- Rather, it is closed automatically when src calls close( ).
- When you close a Scanner, the Readable associated with it is also closed.
- Therefore, in this case, the file referred to by fin is automatically closed when src is closed.

## 10.2.2 Collections Framework

- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Collections were not part of the original Java release, but were added by J2SE 1.2.
- Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.
- Although these classes were quite useful, they lacked a central, unifying theme.
- Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these (and other) problems.

## 10.2.2.1 Collections Overview

- The Collections Framework was designed to meet following goals.
  - The framework had to be high-performance
  - The implementations for the fundamental collections are highly efficient.
  - The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Extending and/or adapting a collection had to be easy.
- Several standard implementations (such as LinkedList, HashSet, and TreeSet) of these interfaces are provided that you may use as-is.
- You may also implement your own collection, if you choose.
- Mechanisms were added that allow the integration of standard arrays into the Collections Framework.

## 10.2.2.1 Collections Overview

- **Algorithms** are another important part of the collection mechanism.
- Algorithms operate on collections and are defined as static methods within the **Collections** class.
- Thus, they are available for all collections.
- 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 provides an iterator, the elements of any collection class can be accessed through the methods defined by **Iterator**.

## 10.2.2.1 Collections Overview

- JDK 8 adds another type of iterator called a **spliterator**, which are iterators that provide support for parallel iteration.
- The interfaces that support spliterators are Spliterator and several nested interfaces that support primitive types.
- In addition to collections, the framework defines several map interfaces and classes.
- Maps store **key/value pairs**.
- Although maps are part of the Collections Framework, they are not "collections" in the strict use of the term.
- You can, however, obtain a collection-view of a map.
- The collection mechanism was retrofitted to some of the original classes defined by java.util so that they too could be integrated into the new system.

## 10.2.2.1 Collections Overview

- When JDK 5 was released, following fundamental changes were made to the Collections Framework

- Generics Fundamentally Changed the Collections Framework
  - All collections are now generic, and many of the methods that operate on collections take generic type parameters.
  - Generics added the one feature that collections had been missing: type safety.
  - Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.
  - Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods.

## 10.2.2.1 Collections Overview

- Autoboxing Facilitates the Use of Primitive Types
  - Autoboxing/unboxing facilitates the storing of primitive types in collections.
  - A collection can store only references, not primitive values.
  - In the past, if you wanted to store a primitive value, such as an int, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type.
  - Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.
- The For-Each Style for Loop
  - All collection classes in the Collections Framework were retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the **foreach** style for loop.

## 10.2.2.2 Collections Interfaces

- 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; that all collections can be cycled through by use of the foreach style for loop.

.

## 10.2.2.2 Collections Interfaces

- Collection declares the core methods that all collections will have.
- Some of these methods are summarized in the following Table.

| Method | Description |
|---|---|
| boolean add(E obj) | Adds obj to the invoking collection.<br>Returns true if obj was added to the collection.<br>Returns false if obj is already a member of the collection and the collection does not allow duplicates. |
| boolean addAll(Collection<? extends E> c) | Adds all the elements of c to the invoking collection.<br>Returns true if the collection changed.<br>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.<br>Otherwise, returns false. |

## 10.2.2.2 Collections Interfaces

| Method | Description |
|---|---|
| int size( ) | Returns the number of elements held in the invoking collection. |
| Object[ ] toArray( ) | Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements |
| 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. |
| Iterator<E> iterator( ) | Returns an iterator for the invoking collection |
| int hashCode( ) | Returns the hash code for the invoking collection. |
| boolean retainAll(Collection<?> c) | Removes all elements from the invoking collection except those in c.<br>Returns true if the collection changed.<br>Otherwise, returns false |

## 10.2.2.2 Collections Interfaces

- 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 declaration:
  interface List<E>
- In addition to the methods defined by Collection, List defines some of its own.

## 10.2.2.2 Collections Interfaces

- Some of the methods are summarized in the following Table.

| Method | Description |
|---|---|
| void add(int index, E obj) | Inserts obj into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| 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. |
| default void sort(Comparator<? super E> comp) | Sorts the list using the comparator specified by comp. |
| ListIterator<E> listIterator(int index) | Returns an iterator to the invoking list that begins at the specified index |

## 10.2.2.2 Collections Interfaces

- The **Set** interface defines a set. It extends Collection and specifies the behavior of a collection that does not allow duplicate elements.
- Set declaration:
  interface Set<E>

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

## 10.2.2.2 Collections Interfaces

- In addition to those methods provided by Set, the SortedSet interface declares some other methods.
- Those methods are summarized in the following table.

| 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. |
| 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. |

## 10.2.2.2 Collections Interfaces

- The **NavigableSet** interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- NavigableSet has this declaration:

  interface NavigableSet<E>

| Method | Description |
|---|---|
| E ceiling(E obj) | Searches the set for the smallest element e such that e >= obj. If such an element is found, it is returned. Otherwise, null is returned. |
| Iterator<E> descendingIterator( ) | Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator. |
| NavigableSet<E> descendingSet( ) | Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set. |

## 10.2.2.2 Collections Interfaces

| Method | Description |
|---|---|
| E floor(E obj) | Searches the set for the largest element e such that e <= obj. If such an element is found, it is returned. Otherwise, null is returned. |
| NavigableSet<E> headSet(E upperBound, boolean incl) | Returns a NavigableSet that includes all elements from the invoking set that are less than upperBound. If incl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set. |
| E higher(E obj) | Searches the set for the largest element e such that e > obj. If such an element is found, it is returned. Otherwise, null is returned. |
| E pollFirst( ) | Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty. |
| NavigableSet<E> subSet(E lowerBound, boolean lowIncl, E upperBound, boolean highIncl) | Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound and less than upperBound. If lowIncl is true, then an element equal to lowerBound is included. If highIncl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set. |

## 10.2.2.2 Collections Interfaces

- 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 has this declaration:

  interface Queue<E>
- The methods declared by Queue are shown in the below table.

| Method | Description |
|---|---|
| E element( ) | Returns the element at the head of the queue. |
| 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. |
| 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. |

## 10.2.2.2 Collections Interfaces

- The **Dequeue** interface extends Queue and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, firstout stacks.
- Deque has this declaration :

  interface Dequeue<E>
- Some of the methods declared by Dequeue are shown in the below table.

| Method | Description |
|---|---|
| void addFirst(E obj) | Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space. |
| void addLast(E obj) | Adds obj to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space. |
| Iterator<E> descendingIterator( ) | Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator |
| void addFirst(E obj) | Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space. |
| void addLast(E obj) | Adds obj to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space. |

## 10.2.2.3 Collections Classes

- The **Collection Classes** are the standard classes that implement collection interface.
- Some of the classes provide full implementations that can be used as-is.
- Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections.
- The core collection classes are summarized in the following table:

| 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 |

## 10.2.2.3 Collections Classes

| Class | Description |
|---|---|
| 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. |

## 10.2.2.3 Collections Classes

- The **ArrayList** class extends **AbstractList** and implements the List interface.
- ArrayList has this declaration:

```
class ArrayList<E>
```

- ArrayList supports dynamic arrays that can grow as needed.
- In Java, standard arrays are of a fixed length.
- Sometimes, you may not know until run time precisely how large an array you need.
- To handle this situation, the Collections Framework defines ArrayList.
- 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.

## 10.2.2.3 Collections Classes

- **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.

## 10.2.2.3 Collections Classes – example 01

```
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);
   }
}
```

## 10.2.2.3 Collections Classes

- The output from this program is shown here:

```
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]
```

- The above program shows a simple use of ArrayList.
- An array list is created for objects of type String, and then several strings are added to it.
- The list is then displayed. Some of the elements are removed and the list is displayed again.

## 10.2.2.3 Collections Classes

- Obtaining an Array from an ArrayList
  - you will sometimes want to obtain an actual array that contains the contents of the list.
  - You can do this by **toArray( )**, which is defined by **Collection**.
  - 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**
  - Normally, the second form is more convenient because it returns the proper type of array.

## 10.2.2.3 Collections Classes

```
import java.util.*;

class ArrayListToArray {
    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) sum += i;
            System.out.println("Sum is: " + sum);
    }
}
```

- The output from the program is shown here:

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

## 10.2.2.3 Collections Classes

- The program begins by creating a collection of integers.
- Next, toArray( ) is called and it obtains an array of Integers.
- Then, the contents of that array are summed by use of a for-each style for loop.
- There is something else of interest in this program. As you know, collections can 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, as the program shows.
- Autoboxing causes them to be automatically wrapped.
- In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

## 10.2.2.3 Collections Classes

- 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>
    ```
  -

    There are 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.

## 10.2.2.3 Collections Classes

-
  ```
  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 = 11.get(2);
         ll.set(2, val + " Changed");
        System.out.println("ll after change: " + ll);
      }
  }
  ```

# 10.2.2.3 Collections Classes

- The output from this program is:

```
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]
```

- Because LinkedList implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast( ).**
- To insert items at a specific location, use the **add(int, E)** form of **add( ),** as illustrated by the call to add(1, "A2") in the example.
- Notice how the third element in **ll** is changed by employing calls to **get( )** and **set( ).**
- To obtain the current value of an element, pass **get( )** the index at which the element is stored.
- To assign a new value to that index, pass **set( )** the index and its new value.

## 10.2.2.3 Collections Classes

- The HashSet Class
  - **HashSet** extends **AbstractSet** and implements the **Set** interface.
  - It creates a collection that uses a hash table for storage.
  - HashSet has this declaration:

    class HashSet<E>

    There are 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.
  - 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.

## 10.2.2.3 Collections Classes

- The transformation of the key into its hash code is performed
- 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.
  **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.
- HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.
- 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.

## 10.2.2.3 Collections Classes

- The LinkedHashSet Class
  - The **LinkedHashSet** class extends **HashSet** and adds no members of its own.
  - It has this declaration:

    class LinkedHashSet<E>
  - LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted.
  - This allows insertion-order iteration over the set.
  - That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
  - This is also the order in which they are contained in the string returned by **toString( )** when called on a LinkedHashSet object.

## 10.2.2.3 Collections Classes

- 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 has this declaration:
    class TreeSet<E>
  - 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):** constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements
    **TreeSet(Comparator<? super E> comp):** 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*

## 10.2.2.3 Collections Classes

- The PriorityQueue Class
  - **PriorityQueue** extends **AbstractQueue** and implements the Queue interface.
  - It creates a queue that is prioritized based on the queue's comparator.
  - PriorityQueue has this declaration:
    class PriorityQueue<E>

  - PriorityQueue defines the six constructors shown here:
    **PriorityQueue( ):** builds an empty queue. Its starting capacity is 11.
    **PriorityQueue(int capacity):** builds a queue that has the specified initial capacity.
    **PriorityQueue(Comparator<? super E> comp) :** specifies a comparator
    **PriorityQueue(int capacity, Comparator<? super E> comp):** builds a queue with the specified capacity and comparator
    **PriorityQueue(Collection<? extends E> c)**
    **PriorityQueue(PriorityQueue<? extends E> c)**
    **PriorityQueue(SortedSet<? extends E> c)**
    } create queues that are initialized with the elements of the collection passed in *c.* Capacity grows as the elements are added.

## 10.2.2.3 Collections Classes

- If no comparator is specified when a PriorityQueue is constructed, then the default comparator for the type of data stored in the queue is used.
- The default comparator will order the queue in ascending order.
- However, by providing a custom comparator, you can specify a different ordering scheme.
- You can obtain a reference to the comparator used by a PriorityQueue by calling its comparator( ) method, as shown below.

  **Comparator<? super E> comparator( )**
- It returns the comparator. If natural ordering is used for the invoking queue, null is returned.
- One word of caution: Although you can iterate through a PriorityQueue using an iterator, the order of that iteration is undefined.
- To properly use a PriorityQueue, you must call methods such as offer( ) and poll( ), which are defined by the Queue interface.

## 10.2.2.3 Collections Classes

- The ArrayDequeue Class
  - The **ArrayDeque** class extends AbstractCollection and implements the Deque interface.
  - It adds no methods of its own.
  - ArrayDeque creates a dynamic array and has no capacity restrictions.
  - ArrayDeque has this declaration:
    class ArrayDeque<E>

  - ArrayDeque defines the following constructors:
    **ArrayDeque( ):** builds an empty deque. Its starting capacity is 16.
    **ArrayDeque(int size):** builds a deque that has the specified initial capacity.
    **ArrayDeque(Collection<? extends E> c) :** creates a deque that is initialized with the elements of the collection passed in c
  - The capacity grows as needed to handle the elements added to the deque.

## 10.2.2.3 Collections Classes

- The EnumSet Class
  - **EnumSet** extends **AbstractSet** and implements Set.
  - It is specifically for use with elements of an enum type.
  - It has this declaration:

    class EnumSet<E extends Enum<E>>
  - EnumSet defines no constructors.
  - Instead, it uses the factory methods. (Some are shown in the following table) to create objects.
  - Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.

| **Method** | **Description** |
|---|---|
| static <E extends Enum<E>> EnumSet<E> allOf(Class<E> t) | Creates an EnumSet that contains the elements in the enumeration specified by t. |
| static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> e) | Creates an EnumSet that is comprised of those elements not stored in e. |
| static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> c) | Creates an EnumSet from the elements stored in c. |
| static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c) | Creates an EnumSet from the elements stored in c. |

## 10.2.2.4 Accessing a collection via an iterator

- you will want to cycle through the elements in a collection.
- One way to do this is to employ an iterator, which is an object that implements either the **Iterator** or the **ListIterator** interface.
- Iterator enables you to cycle through a collection,
- ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Iterator and ListIterator are generic interfaces which are declared as follows:
  interface Iterator<E>    interface ListIterator<E>
- The methods declared by the **Iterator** interface are:

| Method | Description |
|---|---|
| default void forEachRemaining(Consumer<? super E> action) | The action specified by action is executed on each unprocessed element in the collection. |
| boolean hasNext( ) | Returns true if there are more elements. Otherwise, returns false. |
| E next( ) | Returns the next element |
| default void remove( ) | Removes the current element. |

## 10.2.2.4 Accessing a collection via an iterator

- The methods declared by **ListIterator** are shown in the following table:

| Method | Description |
|---|---|
| void add(E obj) | Inserts obj into the list in front of the element that will be returned by the next call to next( ). |
| default void forEachRemaining( Consumer<? super E> action) | The action specified by action is executed on each unprocessed element in the collection. |
| 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. |
| int nextIndex( ) | Returns the index of the next element. |
| E previous( ) | Returns the previous element. |
| int previousIndex( ) | Returns the index of the previous element. |
| void remove( ) | Removes the current element from the list |
| void set(E obj) | Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ). |

## 10.2.2.4 Accessing a collection via an iterator

- Using an Iterator
  - Before you can access a collection through an iterator, you must obtain one.
  - Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection.
  - By using this iterator object, you can access each element in the collection, one element at a time.
  - In general, to use an iterator to cycle through the contents of a collection, these steps should be followed:
    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( ).

## 10.2.2.4 Accessing a collection via an iterator – Example 01

```
import java.util.*;
class IteratorDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("C");    al.add("A");   al.add("E");   al.add("B");  al.add("D");  al.add("F");
        System.out.print("Original contents of al: ");
         Iterator<String> itr = al.iterator();
         while(itr.hasNext()) {
              String element = itr.next();
              System.out.print(element + " ");
         }
         System.out.println();
         ListIterator<String> litr = al.listIterator();
         while(litr.hasNext()) {
              String element = litr.next();
              litr.set(element + "+");
        }
```

## 10.2.2.4 Accessing a collection via an iterator – Example 01

```
        System.out.print("Modified contents of al: ");
         itr = al.iterator();
         while(itr.hasNext()) {
              String element = itr.next();
              System.out.print(element + " ");
         }
         System.out.println();
         System.out.print("Modified list backwards: ");
          while(litr.hasPrevious()) {
                   String element = litr.previous();
                   System.out.print(element + " ");}
           System.out.println();}
}
```

## 10.2.2.4 Accessing a collection via an iterator

- The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

- After the list is modified, litr points to the end of the list.
- litr.hasNext( ) returns false when the end of the list has been reached.
- To traverse the list in reverse, the program continues to use litr, but this time it checks to see whether it has a previous element.
- As long as it does, that element is obtained and displayed.

## 10.2.2.4 Accessing a collection via an iterator

- The For-each alternative to Iterators
  - If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the for loop is often a more convenient alternative to cycling through a collection than is using an iterator.
  - Recall that the for can cycle through any collection of objects that implement the Iterable interface.
  - Because all of the collection classes implement this interface, they can all be operated upon by the for.
  - The **for** loop is substantially shorter and simpler to use than the iteratorbased approach.
  - However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

## 10.2.2.4 Accessing a collection via an For each– Example 01

```
class ForEachDemo {
      public static void main(String args[]) {
             ArrayList<Integer> vals = new ArrayList<Integer>();
             vals.add(1);  vals.add(2);  vals.add(3);  vals.add(4);  vals.add(5);
             System.out.print("Contents of vals: ");
             for(int v : vals)
                System.out.print(v + " ");
             System.out.println();
             int sum = 0;
             for(int v : vals)
                 sum += v;
              System.out.println("Sum of values: " + sum);
   }
}
```

- The output from the program is shown here:

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