

Module 1: Collections and Framework

The collections and Framework: Collections Overview, The Collection Interfaces, The Collection Classes, accessing a collection Via an Iterator, Storing User Defined Classes in Collections, The Random Access Interface, Working with Maps, Comparators, The Collection Algorithms, Arrays, The legacy Classes and Interfaces, Parting Thoughts on Collections.

What is a Collection?

The term "Collection" in Java refers to a group of objects, known as elements, that are grouped together. It represents a general concept of a container for objects.

Example: Think of it as a basket that can hold fruits. You can put different types of fruits like apples, oranges, and bananas in this basket. Here, the basket is the "Collection" and the fruits are its elements.

In Java: In Java, Collection is an interface that defines the basic operations for working with collections. It provides methods like add(), remove(), contains(), etc., which are common to all collection types.

What are Collections?

What is it? On the other hand, "Collections" (plural) in Java refers to a utility class that provides various static methods to work with collections.

Example: Imagine you have several baskets (collections) filled with fruits (elements). Now, you want to perform operations like sorting the fruits in alphabetical order, finding the total number of fruits, or shuffling the fruits randomly among the baskets. The "Collections" utility class provides methods to perform these kinds of operations on your baskets of fruits.

In Java: The Collections class contains static methods for operations such as sorting, searching, shuffling, and synchronization. It helps you manipulate collections in various ways without having to write all the logic yourself.

Summary:

Collection (singular) refers to the concept of a group of objects.

Collections (plural) refers to the utility class in Java that provides methods to work with collections effectively.

What is the Java Collections Framework?

- The Java Collections Framework (JCF) is a set of classes and interfaces that provide a standard way to handle collections in Java.
- It's like a toolkit that Java developers use to work with groups of objects efficiently.

- The Java Collections Framework provides a unified architecture for handling collections of objects in Java.
- It includes interfaces, implementations, utility classes, and algorithms to make working with collections easier and more efficient.
- By using the Java Collections Framework, developers can write code that is more reusable, flexible, and maintainable.

Key Components of the Java Collections Framework

The Java Collections Framework (JCF) is a comprehensive set of classes and interfaces that provide reusable, type-safe, and efficient data structures to store and manipulate groups of objects. It's an essential part of the Java API, offering a unified architecture for working with collections.

a. Interfaces: These define the types of collections and the operations you can perform on them. They are like blueprints or contracts that specify what methods a collection must have.

Collection: The root interface that represents a group of objects.

List: An ordered collection (like an array but dynamic in size). Examples: ArrayList, LinkedList.

Set: A collection that does not allow duplicate elements. Examples: HashSet, TreeSet.

Queue: A collection designed to hold elements before processing. It typically follows the FIFO (First-In-First-Out) principle. Examples: LinkedList, PriorityQueue.

Map: A collection of key-value pairs, where each key is unique. Examples: HashMap, TreeMap.

2. Classes: These are concrete implementations of the interfaces. They provide the actual data structures to store the elements.

ArrayList: Implements the List interface using a dynamic array.

LinkedList: Implements the List and Deque interfaces using a doubly linked list.

HashSet: Implements the Set interface using a hash table.

HashMap: Implements the Map interface using a hash table.

PriorityQueue: Implements the Queue interface using a priority heap.

c. Algorithms: These are static methods provided by the Collections class to perform various operations on collections, such as sorting, searching, and shuffling.

Detailed Explanation

1. Interface

a. Collection Interface:

- Root interface of the collection hierarchy.

- Represents a group of objects, known as elements.
- Defines basic operations common to all collections: add(), remove(), contains(), size(), etc.
- Subinterfaces: List, Set, Queue, Deque.

b. List Interface:

- Represents an ordered collection of elements (sequence).
- Allows duplicate elements.
- Provides methods for positional access and manipulation (get, set, add, remove).
- Implementations: ArrayList, LinkedList, Vector.

c. Set Interface:

- Represents a collection that does not allow duplicate elements.
- Does not preserve insertion order.
- Provides methods for set operations such as union, intersection, and difference.
- Implementations: HashSet, LinkedHashSet, TreeSet.

d. Queue Interface:

- Represents a collection designed for holding elements before processing.
- Follows the FIFO (First-In-First-Out) principle.
- Provides methods for enqueue, dequeue, and peek operations.
- Implementations: LinkedList, PriorityQueue.

e. Deque Interface:

- Represents a double-ended queue.
- Allows insertion and removal of elements from both ends.
- Extends Queue and adds methods for stack operations.
- Implementations: LinkedList, ArrayDeque.

f. Map Interface:

- Represents a collection of key-value pairs.
- Does not allow duplicate keys.
- Provides methods for key-based access and manipulation.
- Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable.

2. Classes:

a. ArrayList:

- Implements the List interface using a dynamic array.
- Provides fast random access to elements.
- Efficient for retrieval but not for insertion/removal in the middle.
- Suitable for scenarios requiring frequent access by index.

b. LinkedList:

- Implements the List and Deque interfaces using a doubly linked list.
- Provides efficient insertion/removal at both ends of the list.
- Suitable for scenarios requiring frequent insertion/removal at the beginning/end.

c. HashSet:

- Implements the Set interface using a hash table.
- Does not allow duplicate elements.
- Provides constant-time performance for basic operations on average.
- Does not guarantee order of elements.

d. HashMap:

- Implements the Map interface using a hash table.
- Provides key-value mapping with constant-time performance for basic operations on average.
- Does not guarantee order of key-value pairs.

3. Algorithms:

- The Collections class provides various static methods for operating on collections.
- Includes algorithms for sorting (sort()), searching (binarySearch()), shuffling (shuffle()), reversing (reverse()), etc.
- These methods are useful for manipulating collections without the need for writing custom code.

Why Use the Collections Framework?

Efficiency: It provides optimized and well-tested data structures and algorithms, making your code more efficient.

Reusability: You can use the same collection interfaces and classes for different types of data, which makes your code more reusable.

Interoperability: Since collections in Java are standardized, they can easily work together and with other parts of the Java API.

Reduced Effort: The framework provides ready-to-use implementations, so you don't need to write your own data structures from scratch.

Say you want to store a list of student names and print them out in alphabetical order. Here's how you might do it using the Java Collections Framework:

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class Main

{
```

```
public static void main(String[] args)
{
    // Create a list to store student names

    List<String> students = new ArrayList<>();

    // Add some names to the list

    students.add("Alice");

    students.add("Bob");

    students.add("Charlie");

    // Sort the list alphabetically

    Collections.sort(students);

    // Print out the sorted list

    for (String student : students)
    {
        System.out.println(student);
    }
}
```

Output:

Alice

Bob

Charlie

1. import Statements:

```
import java.util.ArrayList;
```

```
import java.util.Collections;
```

```
import java.util.List;
```

These lines import the necessary classes from the Java Collections Framework:

ArrayList and **List** are used to create and manage a list of student names.

Collections is a utility class that provides static methods for operating on collections, like sorting.

2. Main Class and Method:

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

This is the entry point of the Java application. The main method is where the program starts executing.

3. Creating the List:

```
List<String> students = new ArrayList<>();
```

Here, a new ArrayList is created to store String objects. The List interface is used for the variable type to promote flexibility (you can easily switch to a different List implementation if needed).

4. Adding Elements to the List:

```
students.add("Alice");  
  
students.add("Bob");  
  
students.add("Charlie");
```

These lines add three student names to the students list. The add method appends each element to the end of the list.

5. Sorting the List:

```
Collections.sort(students);
```

This line sorts the students list in alphabetical order. The Collections.sort method uses the natural ordering of the elements (alphabetical for strings).

6. Printing the Sorted List:

```
for (String student : students)  
  
{  
  
    System.out.println(student);  
  
}
```

This for-each loop iterates over each element in the sorted students list and prints it to the console.

Explanation

Creating the List: We use ArrayList, which is a resizable array implementation of the List interface.

Adding Elements: We add some student names to the list using the add method.

Sorting the List: We use Collections.Sort to sort the list alphabetically.

Printing the List: We use a for-each loop to print each name in the sorted list.

The Java Collections Framework makes it easy to handle data efficiently and with minimal code. Whether you need a list, set, queue, or map, the framework provides you with the tools to manage collections of objects effectively.

The Collection Interface

Description: The root interface of the collection hierarchy.

Methods: Defines basic operations common to all collections, such as adding, removing, and querying elements.

add(E e): Adds the specified element to the collection.

remove(Object o): Removes the specified element from the collection.

contains(Object o): Returns true if the collection contains the specified element.

size(): Returns the number of elements in the collection.

isEmpty(): Returns true if the collection contains no elements.

iterator(): Returns an iterator over the elements in the collection.

Subinterfaces: List, Set, Queue, Deque

Give the syntax of collection interface. Explain the methods present in collection interface.

```
public interface Collection<E> extends Iterable<E>
{
    // Methods

    // Adds the specified element to the collection
    boolean add(E e);

    // Adds all of the elements in the specified collection to this collection
    boolean addAll(Collection<? extends E> c);

    // Removes all of the elements from this collection
    void clear();
```

// Returns true if this collection contains the specified element

boolean contains(Object o);

// Returns true if this collection contains all of the elements in the specified collection

boolean containsAll(Collection<?> c);

// Compares the specified object with this collection for equality

boolean equals(Object o);

// Returns the hash code value for this collection

int hashCode();

// Returns true if this collection contains no elements

boolean isEmpty();

// Returns an iterator over the elements in this collection

Iterator<E> iterator();

// Removes a single instance of the specified element from this collection, if it is present

boolean remove(Object o);

// Removes all of this collection's elements that are also contained in the specified collection

boolean removeAll(Collection<?> c);

// Retains only the elements in this collection that are contained in the specified collection

boolean retainAll(Collection<?> c);

// Returns the number of elements in this collection

```
int size();

// Returns an array containing all of the elements in this collection

Object[] toArray();

// Returns an array containing all of the elements in this collection; the runtime type of the
returned array is that of the specified array

<T> T[] toArray(T[] a);

}
```

Explanation of Methods:

`add(E e)`: Adds the specified element to the collection. Returns true if the collection changed as a result of the call.

`addAll(Collection<? extends E> c)`: Adds all of the elements in the specified collection to this collection. Returns true if this collection changed as a result of the call.

`clear()`: Removes all of the elements from this collection.

`contains(Object o)`: Returns true if this collection contains the specified element.

`containsAll(Collection<?> c)`: Returns true if this collection contains all of the elements in the specified collection.

`equals(Object o)`: Compares the specified object with this collection for equality.

`hashCode()`: Returns the hash code value for this collection.

`isEmpty()`: Returns true if this collection contains no elements.

`iterator()`: Returns an iterator over the elements in this collection.

`remove(Object o)`: Removes a single instance of the specified element from this collection, if it is present.

`removeAll(Collection<?> c)`: Removes all of this collection's elements that are also contained in the specified collection.

`retainAll(Collection<?> c)`: Retains only the elements in this collection that are contained in the specified collection.

`size()`: Returns the number of elements in this collection.

`toArray()`: Returns an array containing all of the elements in this collection.

`toArray(T[] a)`: Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Java program

```
import java.util.ArrayList;

import java.util.Collection;

import java.util.Arrays;

public class CollectionExample

{

    public static void main(String[] args)

    {

        // Create a collection

        Collection<String> collection = new ArrayList<>();

        // Add elements to the collection

        collection.add("Apple");

        collection.add("Banana");

        System.out.println("After add: " + collection);

        // Add all elements from another collection

        Collection<String> moreFruits = Arrays.asList("Cherry", "Date", "Elderberry");

        collection.addAll(moreFruits);

        System.out.println("After addAll: " + collection);

        // Check if collection contains an element

        boolean containsApple = collection.contains("Apple");

        System.out.println("Contains 'Apple': " + containsApple);

        // Check if collection contains all elements of another collection

        boolean containsAllFruits = collection.containsAll(Arrays.asList("Apple", "Banana"));

        System.out.println("Contains all [Apple, Banana]: " + containsAllFruits);
```

```
// Remove an element from the collection

collection.remove("Banana");

System.out.println("After remove 'Banana': " + collection);


// Remove all elements from another collection

collection.removeAll(Arrays.asList("Cherry", "Date"));

System.out.println("After removeAll [Cherry, Date]: " + collection);


// Check if collection is empty

boolean isEmpty = collection.isEmpty();

System.out.println("Is collection empty: " + isEmpty);


// Get the size of the collection

int size = collection.size();

System.out.println("Size of collection: " + size);


// Retain only elements that are also in another collection

collection.retainAll(Arrays.asList("Apple", "Elderberry"));

System.out.println("After retainAll [Apple, Elderberry]: " + collection);


// Clear the collection

collection.clear();

System.out.println("After clear: " + collection);


// Check if collection is empty after clear

isEmpty = collection.isEmpty();

System.out.println("Is collection empty after clear: " + isEmpty);

    }

}
```

Output:

After add: [Apple, Banana]

After addAll: [Apple, Banana, Cherry, Date, Elderberry]

Contains 'Apple': true

Contains all [Apple, Banana]: true

After remove 'Banana': [Apple, Cherry, Date, Elderberry]

After removeAll [Cherry, Date]: [Apple, Elderberry]

Is collection empty: false

Size of collection: 2

After retainAll [Apple, Elderberry]: [Apple, Elderberry]

After clear: []

Is collection empty after clear: true

Simple Java program demonstrating the use of the equals method with collections. This program will show how to compare two collections for equality.

```
import java.util.ArrayList;
import java.util.Collection;

public class EqualsExample
{
    public static void main(String[] args)
    {
        // Create two collections
        Collection<String> collection1 = new ArrayList<>();
        Collection<String> collection2 = new ArrayList<>();

        // Add elements to the first collection
        collection1.add("Apple");
        collection1.add("Banana");
        collection1.add("Cherry");
```

```
// Add elements to the second collection

collection2.add("Apple");

collection2.add("Banana");

collection2.add("Cherry");

// Check if the two collections are equal

boolean areEqual = collection1.equals(collection2);

System.out.println("Are collection1 and collection2 equal? " + areEqual);

// Modify one collection

collection2.add("Date");

// Check if the collections are still equal

areEqual = collection1.equals(collection2);

System.out.println("Are collection1 and collection2 equal after modification? " +
areEqual);

// Create a third collection with different elements

Collection<String> collection3 = new ArrayList<>();

collection3.add("Apple");

collection3.add("Banana");

// Check if collection1 is equal to collection3

areEqual = collection1.equals(collection3);

System.out.println("Are collection1 and collection3 equal? " + areEqual);

}

}
```

Output:

Are collection1 and collection2 equal? true

Are collection1 and collection2 equal after modification? false

Are collection1 and collection3 equal? False

Simpler Java program that demonstrates the use of the **hashCode method** with collections. This program focuses on creating two collections, comparing their hash codes, and observing the effect of modifying one of the collections.

```
import java.util.ArrayList;
```

```
import java.util.Collection;

public class SimpleHashCodeExample
{
    public static void main(String[] args)
    {
        // Create two collections

        Collection<String> collection1 = new ArrayList<>();
        Collection<String> collection2 = new ArrayList<>();

        // Add elements to the first collection
        collection1.add("Apple");
        collection1.add("Banana");
        collection1.add("Cherry");

        // Add elements to the second collection
        collection2.add("Apple");
        collection2.add("Banana");
        collection2.add("Cherry");

        // Compute and print hash codes for both collections

        System.out.println("Hash code of collection1: " + collection1.hashCode());
        System.out.println("Hash code of collection2: " + collection2.hashCode());

        // Check if the hash codes are equal
        if (collection1.hashCode() == collection2.hashCode())
        {
            System.out.println("The hash codes of collection1 and collection2 are equal.");
        }
        else
        {
            System.out.println("The hash codes of collection1 and collection2 are not equal.");
        }
    }
}
```

```
}

// Modify the second collection

collection2.add("Date");

// Compute and print hash codes again after modification

System.out.println("Hash code of collection1 after modification: " +
collection1.hashCode());

System.out.println("Hash code of collection2 after modification: " +
collection2.hashCode());

// Check if the hash codes are still equal after modification

if (collection1.hashCode() == collection2.hashCode())

{

    System.out.println("The hash codes of collection1 and collection2 are still equal after
modification.");

}

else

{

    System.out.println("The hash codes of collection1 and collection2 are not equal after
modification.");

}

}
```

Output:

Hash code of collection1: 2066939

Hash code of collection2: 2066939

The hash codes of collection1 and collection2 are equal.

Hash code of collection1 after modification: 2066939

Hash code of collection2 after modification: 1357183

The hash codes of collection1 and collection2 are not equal after modification.

1. Explain brief about collection frame work.

- The Java Collections Framework standardizes the way in which groups of

- objects are handled by your programs.
- The framework had to be high-performance.
- The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) 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.
- 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.
- 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.

2. What are the recent changes to collection framework?

Recently, the Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use. The changes were the addition of generics, autoboxing/unboxing, and the for-each style for loop.

Generics

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework has been reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Generics add 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. Thus, it was possible to accidentally store incompatible types in a collection.

Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Autoboxing/unboxing

Autoboxing facilitates the Use of Primitive Types.

Autoboxing/unboxing facilitates the storing of primitive types in collections.

As you will see, 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

collection can be cycled through by use of the for-each style for loop.

Earlier it was done with Iterable interface. For each loop is easier than the earlier iterator.

3. List the Collection Interfaces?

- The Collections Framework defines several interfaces. This section provides an overview of each interface. 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
- 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.

4. Give the syntax of collection interface. Explain the methods present in collection interface.

interface Collection<E>

E specifies the type of objects that the collection

Collection extends the Iterable interface.

Iterating through the list can be done through the iterable interface.

Methods in collection interface

add

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.

addAll

boolean addAll(Collection<? extends E> c)

Adds all the elements of c to the invoking collection.

Returns true if the operation succeeded

(i.e., the elements were added). Otherwise, returns false

clear

void clear()

Removes all elements from the invoking collection.

contains

boolean contains(Object obj)

Returns true if obj is an element of the invoking collection.

Otherwise, returns false.

containsAll

boolean containsAll(Collection<?> c)

Returns true if the invoking collection contains all elements of c.

Otherwise, returns false.

equals

boolean equals(Object obj)

Returns true if the invoking collection and obj are equal.

Otherwise, returns false.

hashCode

int hashCode() Returns the hash code for the invoking collection.

isEmpty

boolean isEmpty()

Returns true if the invoking collection is empty.

Otherwise, returns false.

iterator

Iterator<E> iterator() Returns an iterator for the invoking collection.

remove

boolean remove(Object obj)

Removes one instance of obj from the invoking collection.

Returns true if the element was removed. Otherwise, returns false.

removeAll

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.

retainAll

boolean retainAll(Collection<?> c)

Removes all elements from the invoking collection except those in c.

Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

size

int size() Returns the number of elements held in the invoking collection.

toArray

Object[] toArray()

Returns an array that contains all the elements stored in the invoking collection.

The array elements are copies of the collection elements.

The array elements are copies of the collection elements.

If the size of array equals the number of elements, these are returned in array.

5. Explain the methods present in List interface.

List interface extends collection interface. It includes new method. Which are given below.

void add(int index , E obj)

Inserts obj into the invoking list at the index passed in index.

Any pre existing elements at or beyond the point of insertion are shifted up.

boolean addAll(int index , Collection<? extends E> c)

Inserts all elements of C into the invoking list at the index passed in

index . Any pre existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.

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.

int lastIndexOf(Object obj)

Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

ListIterator<E> listIterator()

Returns an iterator to the start of the invoking list.

ListIterator<E> listIterator(int index)

Returns an iterator to the invoking list that begins at the specified index.

E remove(int index)

Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.

E set(int index , E obj)

Assigns obj to the location specified by index within the invoking list.

List<E> subList(int start , int end)

Returns a list that includes elements from start to end –1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

6. Explain Set Interface and set method:

The Set interface defines a set.

It extends Collection and declares the behaviour of a collection that does not allow duplicate elements.

Therefore, the add() method returns false if an attempt.

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 defined by Set, the SortedSet interface declares the methods.

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.

Several methods throw a NoSuchElementException when no items are contained in the invoking set.

A ClassCastException is thrown when an object is incompatible with the elements in a set.

A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set.

An IllegalArgumentException is thrown if an invalid argument is used.

7. **NavigableSet Interface and method**

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 is a generic interface that has this declaration:

interface NavigableSet<E>

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

NavigableSet adds the following

E ceiling(E obj)

Searches the set for the smallest element

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.

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 lower(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.

E pollLast()

Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest 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.

NavigableSet<E> tailSet(E lowerBound , boolean incl)

Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound . If incl is true, then an element equal to lowerBound is included. The resulting set is backed by the invoking set

8. The Queue Interface and methods

The Queue interface extends Collection and declares the behaviour 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>
```

```
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.

9. Deque interface

It 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, first-out stacks.

Module 1: Collections and Framework

Deque is a generic interface that has this declaration:

```
interface Deque<E>
```

```
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.

```
E getFirst( )
```

Returns the first element in the deque. The object is not removed from the deque. It throws `NoSuchElementException` if the deque is empty.

```
E getLast( )
```

Returns the last element in the deque.

The object is not removed from the deque. It throws `NoSuchElementException` if the deque is empty.

```
boolean offerFirst(E obj )
```

Attempts to add obj to the head of the deque. Returns true if obj was added and false otherwise. Therefore, this method returns false when an attempt is made to add obj to a full, capacity-restricted deque.

```
boolean offerLast(E obj )
```

Attempts to add obj to the tail of the deque. Returns true if obj was added and false otherwise.

```
E peekFirst( )
```

Returns the element at the head of the deque. It returns null if the deque is empty. The object is not removed.

```
E peekLast( )
```

Returns the element at the tail of the deque. It returns null if the deque is empty. The object is not removed.

```
E pollFirst( )
```

Returns the element at the head of the deque, removing the element in the process. It returns null if the deque is empty.

`E pollLast( )` Returns the element at the tail of the deque, removing the element in the process. It returns null if the deque is empty.

`E pop( )` Returns the element at the head of the deque, removing it in the process. It throws `NoSuchElementException` if the deque is empty.

`void push(E obj )` Adds obj to the head of the deque. Throws an `IllegalStateException` if a capacity-restricted deque is out of space.

Module 1: Collections and Framework

`E removeFirst( )` Returns the element at the head of the deque, removing the element in the process. It throws `NoSuchElementException` if the deque is empty.

`boolean removeFirstOccurrence(Object obj )`

Removes the first occurrence of `obj` from the deque. Returns true if successful and false if the deque did not contain `obj`.

`E removeLast( )`

Returns the element at the tail of the deque, removing the element in the process. It throws `NoSuchElementException` if the deque is empty.

`boolean removeLastOccurrence(Object obj )`

Removes the last occurrence of `obj` from the deque. Returns true if successful and false if the deque did not contain.

10. The Collection Classes with example code

AbstractCollection

Implements most of the Collection interface.

AbstractList

Extends **AbstractCollection** and implements most of the List interface.

Queue interface.

AbstractSequentialList

Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.

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**.

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.

ArrayList

ArrayList class extends **AbstractList** and implements the **List** interface

ArrayList is a generic class that has this declaration:

```
class ArrayList<E>
```

ArrayList has the constructors shown here:

```
ArrayList( )
```

constructor builds an empty array list

```
ArrayList(Collection<? extends E> c)
```

Module 1: Collections and Framework

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.

```
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());
    }
}
```

Converting ArrayList to Array

```
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);
    }
}
```

LinkedList

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

Module 1: Collections and Framework

constructors

`LinkedList()`

`LinkedList(Collection<? extends E> c)`

The first constructor builds an empty linked list.

The second constructor builds a linked list that is initialized with the elements of the collection *c*.

Example code:

```
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);
    }
}
```

HashSet

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage.

Java HashSet class is used to create a collection that uses a hash table for storage.

It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.

HashSet is a generic class that has this declaration:

```
class HashSet<E>
```

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

Constructor

`HashSet()`

`HashSet(Collection<? extends E> c)`

`HashSet(int capacity)`

Module 1: Collections and Framework

HashSet(int *capacity*, float *fillRatio*)

Example:

```
import java.util.*;  
class HashSetDemo {  
    public static void main(String args[]) {  
        HashSet<String> hs = new HashSet<String>();
```

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

}
```

output

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

LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Contains unique elements only like HashSet.
- Provides all optional set operations, and permits null elements.
- Maintains insertion order.

The **LinkedHashSet** class extends **HashSet** and adds no members of its own.

It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

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

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.

To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

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

which is the order in which the elements were inserted.

TreeSet

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()  
TreeSet(Collection<? extends E> c)  
TreeSet(Comparator<? super E> comp)  
TreeSet(SortedSet<E> ss)
```

Example

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

The output from this program is shown here:
[A, B, C, D, E, F]

PriorityQueue

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator.

PriorityQueue is a generic class that has this declaration:

```
class PriorityQueue<E>
```

Here, **E** specifies the type of objects stored in the queue.

PriorityQueues are dynamic, growing as necessary.

PriorityQueue defines the six constructors shown here:

```
PriorityQueue()
```

```
PriorityQueue(int capacity)
```

```
PriorityQueue(int capacity, Comparator<? super E> comp)
```

```
PriorityQueue(Collection<? extends E> c)
```

```
PriorityQueue(PriorityQueue<? extends E> c)
```

```
PriorityQueue(SortedSet<? extends E> c)
```

ArrayDeque

Java SE 6 added the **ArrayDeque** class, which 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 is a generic class that has this declaration:

```
class ArrayDeque<E>
```

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

```
ArrayDeque()
```

```
ArrayDeque(int size)
```

```
ArrayDeque(Collection<? extends E> c)
```

Example:

```
import java.util.*;
class ArrayDequeDemo {
    public static void main(String args[]) {
        ArrayDeque<String> adq = new ArrayDeque<String>();
        adq.push("A");
        adq.push("B");
        adq.push("D");
        adq.push("E");
        adq.push("F");
        System.out.print("Popping the stack: ");
        while(adq.peek() != null)
            System.out.print(adq.pop() + " ");
        System.out.println();
    }
}
```

The output is shown here:

Popping the stack: F E D B A

Accessing a collection Via 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. Element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these 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()**.
4. For collections that implement **List**, you can also obtain an iterator by calling **listIterator()**.
5. As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element.
6. Otherwise, **ListIterator** is used just like **Iterator**.

```
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 + "+");
}

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();
}

}

```

Output:

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+

For Each loop for iterating through collection:

```

import java.util.*;
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("Original 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);
}

}

```

Output:

Original contents of vals: 1 2 3 4 5
Sum of values: 15

Storing User Defined Classes in Collections:

collections are not limited to the storage of built-in objects.

The power of collections is that they can store any type of object, including objects of classes that you create.

User defined objects stored in **LinkedList** to store mailing addresses:

```

import java.util.*;
class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    private String code;
    Address(String n, String s, String c,
        String st, String cd) {
        name = n;
        street = s;
        city = c;
        state = st;
        code = cd;
    }

    public String toString() {
        return name + "\n" + street + "\n" +
            city + " " + state + " " + code;
    }
}

class MailList {
    public static void main(String args[]) {
        LinkedList<Address> ml = new LinkedList<Address>();
        ml.add(new Address("J.W. West", "11 Oak Ave",
            "Urbana", "IL", "61801"));
        ml.add(new Address("Ralph Baker", "1142 Maple Lane",
            "Mahomet", "IL", "61853"));
        ml.add(new Address("Tom Carlton", "867 Elm St",
            "Champaign", "IL", "61820"));
        for(Address element : ml)
            System.out.println(element + "\n");
    }
}

```

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

The output from the program is shown here:

```
J.W. West  
11 Oak Ave  
Urbana IL 61801  
Ralph Baker  
1142 Maple Lane  
Mahomet IL 61853  
Tom Carlton  
867 Elm St  
Champaign IL 61820
```

Random Access Interface:

RandomAccess interface contains no members.

However, by implementing this interface, a collection signals that it supports efficient random access to its elements.

By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections.

RandomAccess is implemented by **ArrayList** and by the legacy **Vector** class, among others.

Working With Maps:

A *map* is an object that stores associations between keys and values, or *key/value pairs*.

Keys and values are objects. Keys must be unique, but the values may be duplicated.

Some maps can accept a **null** key and **null** values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map.

However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them.

The following interfaces support maps:

The 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 date. 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.

Map is generic:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.
The methods declared by **Map**.

Several methods
throw a **ClassCastException** when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the map.

An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Maps revolve around two basic operations: **get()** and **put()**. To put a value into a map, use **put()**, specifying the key and the value.

To obtain a value, call **get()**, passing the key as an argument. The value is returned.
maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet()** method. It returns a **Set** that contains the elements in the map.

To obtain a collection-view of the keys, use **keySet()**.

To get a collection-view of the values, use **values()**.

Collection-views are the means by which maps are integrated into the larger Collections Framework.

SortedMap

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending order based on the keys.

SortedMap is generic and is declared as shown here:

```
interface SortedMap<K, V>
```

K specifies the type of keys,

V specifies the type of values.

Several methods throw a **NoSuchElementException** when no items are in the invoking map.
A **ClassCastException** is thrown when an object is incompatible with the elements in a map.
A **NullPointerException** is thrown if an attempt is made to use a **null** object when **null** is not allowed in the map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Sorted maps allow very efficient manipulations of *submaps*

To obtain a submap, use **headMap()**, **tailMap()**, or **subMap()**.

To get the first key in the set, call **firstKey()**.

To get the last key, use **lastKey()**.

NavigableMap Interface

The **NavigableMap** interface was added by Java SE 6.

It extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:

```
interface NavigableMap<K,V>
```

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

Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.
equal to start.

Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet()** method declared by the **Map** interface returns a **Set** containing the map entries.

Each of these set elements is a **Map.Entry** object. **Map.Entry** is generic and is declared like this:

```
interface Map.Entry<K, V> Here, K specifies the type of keys, and V specifies the type of values.  
the methods declared by Map.Entry.
```

Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

HashMap:

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.

The following constructors are defined:

```
HashMap( )  
HashMap(Map<? extends K, ? extends V> m)  
HashMap(int capacity)  
HashMap(int capacity, float fillRatio)
```

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.

The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16.

The default fill ratio is 0.75.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

```
import java.util.*;  
class HashMapDemo {  
    public static void main(String args[]) {  
        HashMap<String, Double> hm = new HashMap<String, Double>();  
        hm.put("John Doe", new Double(3434.34));  
        hm.put("Tom Smith", new Double(123.22));  
        hm.put("Jane Baker", new Double(1378.00));  
        hm.put("Tod Hall", new Double(99.22));  
        hm.put("Ralph Smith", new Double(-19.08));  
        Set<Map.Entry<String, Double>> set = hm.entrySet();  
        for(Map.Entry<String, Double> me : set) {  
            System.out.print(me.getKey() + ": ");  
            System.out.println(me.getValue());  
        }  
  
        System.out.println();  
        double balance = hm.get("John Doe");  
        hm.put("John Doe", balance + 1000);  
        System.out.println("John Doe's new balance: " +  
            hm.get("John Doe"));  
    }  
}
```

Output from this program is shown here (the precise order may vary):

```
Ralph Smith: -19.08  
Tom Smith: 123.22  
John Doe: 3434.34  
Tod Hall: 99.22  
Jane Baker: 1378.0  
John Doe's new balance: 4434.34
```

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling **entrySet()**. The keys and values are displayed by calling the **getKey()** and **getValue()** methods

that are defined by **Map.Entry**. Pay close attention to how the deposit is made into John Doe's

account. The **put()** method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

TreeMap

The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface. It creates maps stored in a tree structure.

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 ascending key order.

TreeMap is a generic class that has this declaration:
class **TreeMap**<K, V>

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

The following **TreeMap** constructors are defined:

TreeMap()

TreeMap(**Comparator**<? super K> *comp*)

TreeMap(**Map**<? extends K, ? extends V> *m*)

TreeMap(**SortedMap**<K, ? extends V> *sm*)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*. (Comparators are discussed later in this chapter.) The third form initializes

a tree map with the entries from *m*, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*.

TreeMap has no methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;
class TreeMapDemo {
    public static void main(String args[]) {
        // Create a tree map.
        TreeMap<String, Double> tm = new TreeMap<String, Double>();
        // Put elements to the map.
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));
```

```
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = tm.entrySet();
// Display the elements.
for(Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}

System.out.println();
double balance = tm.get("John Doe");
tm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
tm.get("John Doe"));
}

}
```

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22
John Doe's current balance: 4434.34
TreeMap sorts the keys.
```

However, in this case, they are sorted by first name instead of last name.

You can alter this behavior by specifying a comparator when the map is created, as described shortly.

LinkedHashMap

LinkedHashMap extends **HashMap**.

It maintains a linked list of the entries in the map, in the 1order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

LinkedHashMap that returns its elements in the order in which they were last accessed.

LinkedHashMap is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

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

LinkedHashMap defines the following constructors:

```
LinkedHashMap( )
```

```
LinkedHashMap(Map<? extends K, ? extends V> m)
```

```
LinkedHashMap(int capacity)
```

`LinkedHashMap(int capacity, float fillRatio)`

`LinkedHashMap(int capacity, float fillRatio, boolean Order)`

The first form constructs a default **LinkedHashMap**.

The second form initializes the **LinkedHashMap** with the elements from *m*. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capacity is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access.

IdentityHashMap

IdentityHashMap extends **AbstractMap** and implements the **Map** interface.

It is similar to **HashMap** except that it uses reference equality when comparing elements.

IdentityHashMap is a generic class that has this declaration:

```
class IdentityHashMap<K, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

```
class EnumMap<K extends Enum<K>, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

```
EnumMap(Class<K> kType)
```

```
EnumMap(Map<K, ? extends V> m)
```

```
EnumMap(EnumMap<K, ? extends V> em)
```

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em*.

Comparator interface

Comparator is a generic interface that has this declaration:

```
interface Comparator<T>
```

Here, **T** specifies the type of objects being compared.

The **Comparator** interface defines two methods: **compare()** and **equals()**. The **compare()**

method, shown here, compares two elements for order:

```
int compare(T obj1, T obj2)
```

obj1 and *obj2* are the objects to be compared.

This method returns zero if the objects are equal.

It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned.

ClassCastException if the types of the objects are not compatible for comparison.

By overriding **compare()**, you can alter the way that objects are ordered.

For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison. The **equals()** method, shown here, tests whether an object equals the invoking comparator:

`boolean equals(Object obj)`

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

```
import java.util.*;
class MyComp implements Comparator<String> {
public int compare(String a, String b) {
String aStr, bStr;
aStr = a;
bStr = b;
return bStr.compareTo(aStr);
}

}

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

Output:

F E D C B A

The Collection Algorithms:

Collections Framework defines several algorithms that can be applied to collections and maps.

algorithms are defined as static methods within the **Collections** class.

`static <T> Boolean addAll(Collection <? super T> c, T ... elements)`

Inserts the elements specified by elements into the collection specified by c. Returns true if the elements were added and false otherwise.

`static <T> Queue<T> asLifoQueue(Deque<T> c)`

Returns a last-in, first-out view of c.

`static <T>int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)`

Searches for value in list ordered according to c. Returns the position of value in list, or a negative value if value is not found.

`static <T> int binarySearch(List<? extends Comparable<? super T>> list,T value)`

Searches for value in list. The list must be sorted. Returns the position of value in list, or a negative value if value is not found.

`static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)`

Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a `ClassCastException`.

`static <E> List<E> checkedList(List<E> c, Class<E> t)`

Returns a run-time type-safe view of a List. An attempt to insert an incompatible element will cause a `ClassCastException`.

`static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)`

Returns a run-time type-safe view of a Map.

An attempt to insert an incompatible element will cause a `ClassCastException`.

`static <E> List<E> checkedSet(Set<E> c, Class<E> t)`

Returns a run-time type-safe view of a Set. An

attempt to insert an incompatible element will cause a `ClassCastException`.

`static int frequency(Collection<?> c, Object obj)`

Counts the number of occurrences of obj in c and returns the result.

`static int indexOfSubList(List<?> list, List<?> subList)`

Searches list for the first occurrence of subList.

Returns the index of the first match, or -1 if no match is found.

`static int lastIndexOfSubList(List<?> list, List<?> subList)`

Searches list for the last occurrence of subList.

Returns the index of the last match, or -1 if no match is found.

```
import java.util.*;
class AlgorithmsDemo {
    public static void main(String args[]) {
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.add(-8);
        ll.add(20);
        ll.add(-20);
        ll.add(8);
        Comparator<Integer> r = Collections.reverseOrder();
        Collections.sort(ll, r);
        System.out.print("List sorted in reverse: ");
        for(int i : ll)
            System.out.print(i+ " ");
        System.out.println();
        Collections.shuffle(ll);
        System.out.print("List shuffled: ");
        for(int i : ll)
            System.out.print(i + " ");
        System.out.println();
        System.out.println("Minimum: " + Collections.min(ll));
    }
}
```

```

System.out.println("Maximum: " + Collections.max(l1));
}

}

```

Output:

List sorted in reverse: 20 8 -8 -20

List shuffled: 20 -20 8 -8

Minimum: -20

Maximum: 20

Notice that **min()** and **max()** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Why Generic Collections?

As mentioned at the start of this chapter, the entire Collections Framework was refitted for generics when JDK 5 was released.

Furthermore, the Collections Framework is arguably the single most important use of generics in the Java API.

The reason for this is that generics add type safety to the Collections Framework. Before moving on, it is worth taking some time to examine in detail the significance of this improvement.

```

import java.util.*;
class OldStyle {
public static void main(String args[]) {
    ArrayList list = new ArrayList();
    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");
    Iterator itr = list.iterator();
    while(itr.hasNext()) {
        String str = (String) itr.next(); // explicit cast needed here.
        System.out.println(str + " is " + str.length() + " chars long.");
    }
}
}

```

Prior to generics, all collections stored references of type **Object**.

This allowed any type of reference to be stored in the collection.

The preceding program uses this feature to store references to objects of type **String** in **list**, but any type of reference could have been stored. Unfortunately, the fact that a pre-generics collection stored **Object** references could easily lead to errors.

First, it required that you, rather than the compiler, ensure that only objects of the proper type be stored in a specific collection. For example, in the preceding example, **list** is clearly intended to store **Strings**, but there is nothing that actually prevents another type of reference from being added to the collection.

For example, the compiler will find nothing wrong with this line of code:
`list.add(new Integer(100));`

Because **list** stores **Object** references, it can store a reference to **Integer** as well as it can store a reference to **String**.

However, if you intended **list** to hold only strings, then the preceding statement would corrupt the collection. Again, the compiler had no way to know that the preceding statement is invalid.

The second problem with pre-generics collections is that when you retrieve a reference from the collection, you must manually cast that reference into the proper type. This is why the preceding program casts the reference returned by `next()` into **String**. Prior to generics, collections simply stored **Object** references. Thus, the cast was necessary when retrieving objects from a collection.

Aside from the inconvenience of always having to cast a retrieved reference into its proper type, this lack of type safety often led to a rather serious, but surprisingly easy-to-create, error. Because **Object** can be cast into any type of object, it was possible to cast a reference obtained from a collection into the *wrong type*. For example, if the following statement were added to the preceding example, it would still compile without error, but generate a run-time exception when executed:

```
Integer i = (Integer) itr.next();
```

- Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.
- Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects.

When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces.

Thus, they are fully compatible with the framework. While no classes have actually been deprecated, one has been rendered obsolete.

Of course, where a collection duplicates the functionality of a legacy class, you will usually want to use the collection for new code. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the collection classes are synchronized, but all the legacy classes are synchronized.

This distinction may be important in some situations. Of course, you can easily synchronize collections, too, by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary
Hashtable
Properties
Stack
Vector

There is one legacy interface called **Enumeration**.

The following sections examine **Enumeration** and each of the legacy classes, in turn.

The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**.

```
interface Enumeration<E>
```

where **E** specifies the type of element being enumerated.

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections.

Vector is declared like this:

```
class Vector<E>
```

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

```
Vector()
```

```
Vector(int size)
```

```
Vector(int size, int incr)
```

```
Vector(Collection<? extends E> c)
```

- The first form creates a default vector, which has an initial size of 10.
- The second form creates a vector whose initial capacity is specified by *size*.
- The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr*.

The increment specifies the number of elements to allocate each time that a vector is resized upward.

The fourth form creates a vector that contains the elements of collection *c*.

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

```
class Stack<E>
```

Here, **E** specifies the type of element stored in the stack.
Stack includes all the methods defined by **Vector**.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**.

Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is fully discussed here.

class Dictionary<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 17-17.

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a Dictionary.

HashMap, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table. **Hashtable** was made generic by JDK 5.

It is declared like this: **class Hashtable<K, V>**

```
Hashtable()  
Hashtable(int size)  
Hashtable(int size, float fillRatio)  
Hashtable(Map<? extends K, ? extends V>  
m)
```

The first version is the default constructor.

The second version creates a hash table that has an initial size specified by *size*.

The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hashtable multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used.

Finally, the fourth version creates a hash table that is initialized with the elements in m . The capacity of the hash table is set to twice the number of elements in m . The default load factor of 0.75 is used.

Properties

Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**.

The **Properties** class is used by many other Java classes. For example, it is the type of object returned by **System.getProperties()** when obtaining environmental values.

Although the **Properties** class, itself, is not generic, several of its methods are. **Properties** defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties**

defines these constructors:

Properties()

Properties(Properties *propDefault*)