Java Collection Framework

Chapter Topics

- Introduction to the Java collections Framework
- Lists
- Sets
- Maps
- The Collections Class

The Java Collection Framework

The Java Collections Framework is a library of classes and interfaces for working with collections of objects.

A collection is an object which can store other objects, called elements. Collections provide methods for adding and removing elements, and for searching for a particular element within the collection.

The Main Types of Collections

- Lists
- Sets
- Maps

Lists

Lists: List type collections assign an integer (called an index) to each element stored.

Indices of elements are 0 for the element at the beginning of the list, 1 for the next element, and so on.

Lists permit duplicate elements, which are distinguished by their position in the list.

Sets

Set: a collection with no notion of position within the collection for stored elements. Sets do not permit duplicate elements.

Maps

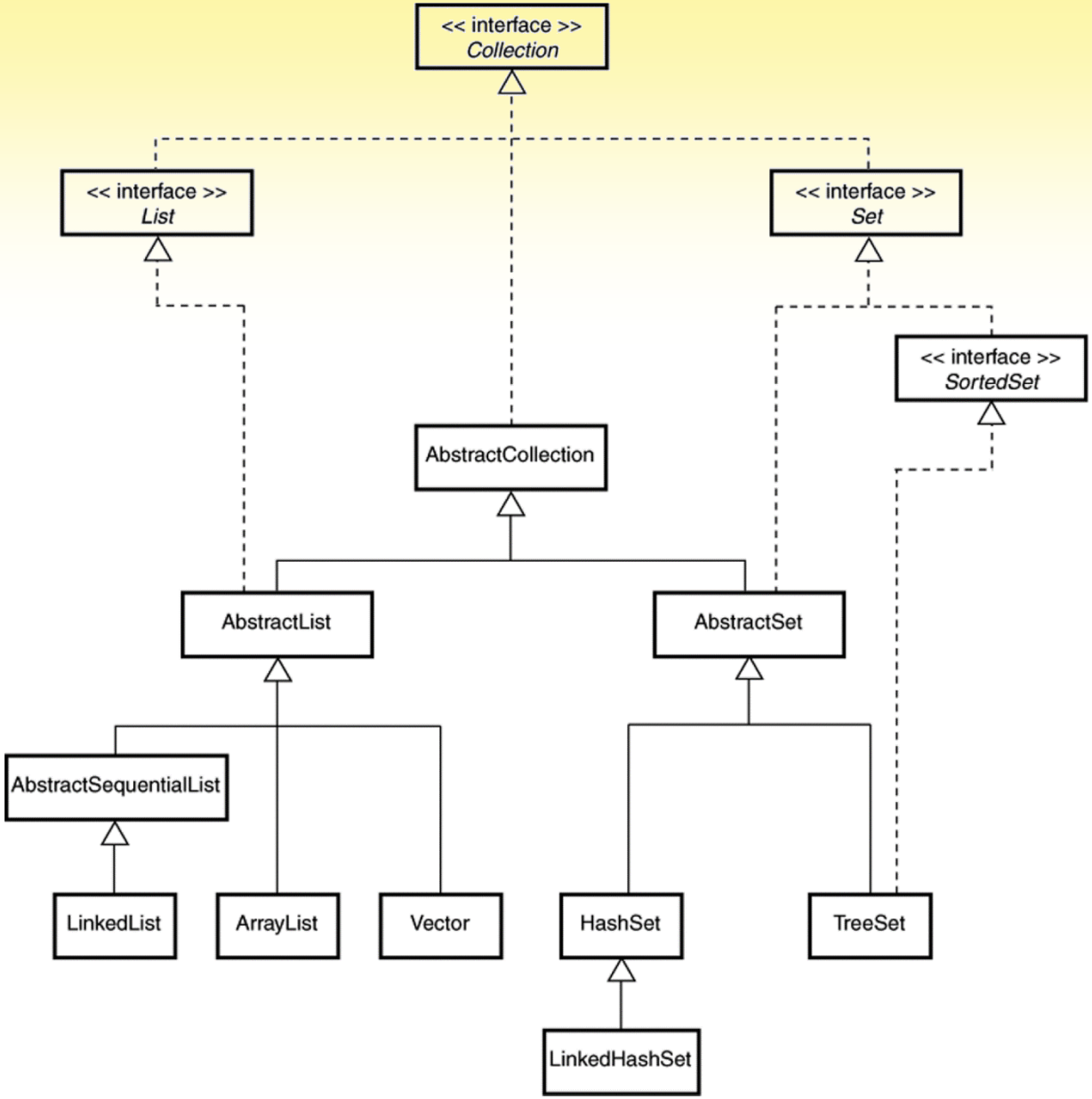
A map is a collection of pairs of objects:

1. A value: this is the object to be stored.
2. A key: this is another object associated with the value, and which can be used to quickly find the value within the collection.

A map is really a set of keys, with each each key having a value attached to it.

Maps do not allow duplicate keys.

Part of the JCF Hierarchy


<< interface >>
Collection
<< interface >>
List
<< interface >>
Set
<< interface >>
SortedSet
AbstractCollection
AbstractList
AbstractSet
AbstractSequentialList
LinkedList
ArrayList
Vector
HashSet
TreeSet
LinkedHashSet

The Collection Interface

- Lists and Sets are similar in many ways.
- The Collection Interface describes the operations that are common to both.
- Maps are fundamentally different from Lists and Sets and are described by a different interface.

Some Methods in the Collection Interface

Method	Description
add(o : E) : boolean	Adds an object o to the Collection. The method returns true if o is successfully added to the collection, false otherwise.
clear() : void	Removes all elements from the collection.
contains(o : Object): boolean	Returns true if o is an element of the collection, false otherwise.
isEmpty() : boolean	Returns true if there are no elements in the collection, false otherwise.
iterator() : Iterator<E>	Returns an object called an iterator that can be used to examine all elements stored in the collection.
remove(o : Object) : boolean	Removes the object o from the collection and returns true if the operation is successful, false otherwise.
size() : int	Returns the number of elements currently stored in the collection.

AbstractCollection

The AbstractCollection class provides a skeleton implementation for a Collection class by implementing many of the methods of the Collection interface.

Programmers can create a working collection class by providing implementations for iterator(), size(), and overriding add(o : Object).

Iterators

An iterator is an object that is associated with a collection. The iterator provides methods for fetching the elements of the collection, one at a time, in some order.

Iterators have a method for removing from the collection the last item fetched.

The Iterator Interface

Iterators implement the Iterator interface. This interface specifies the following methods:

hasNext() : boolean

next() : E

remove() : void

The remove() method is optional, so not all iterators have it.

Methods of the Iterator Interface

Method	Description
hasNext() : boolean	Returns true if there is at least one more element from the collection that can be returned, false otherwise.
next() : E	Returns the next element from the collection.
remove() : void	Removes from the collection the element returned by the last call to next(). This method can be called at least one time for each call to next().

The List Interface

The List interface extends the Collection interface by adding operations that are specific to the position-based, index-oriented nature of a list.

List Interface Methods

The methods in the List interface describe operations for adding elements and removing elements from the list based on the index of the element.

There are also methods for determining the index of an element in the list when the value of an element is known.

The List Interface Methods

add(index:int, el:E) : void	Adds the element el to the collection at the given index. Throws IndexOutOfBoundsException if index is negative, or greater than the size of the list.
get(index:int):E	Returns the element at the given index, or throws IndexOutBoundsException if index is negative or greater than or equal to the size of the list.
indexOf(o:Object):int	Returns the least (first) index at which the object o is found; returns -1 if o is not in the list.
lastIndexOf(o:Object):int	Returns the greatest (last) index at which the object o is found; returns -1 if o is not in the list.
listIterator():ListIterator<E>	Returns an iterator specialized to work with List collections.
remove(index:int):E	Removes and returns the element at the given index; throws IndexOutOfBoundsException if index is negative, or greater than or equal to the size of the list.
set(index:int, el:E):E	Replaces the element at index with the new element el.

AbstractList

This is an abstract class that provides a skeletal implementation of a List collection.

It extends AbstractCollection and implements the List interface.

It serves as the abstract superclass for the concrete classes ArrayList and Vector.

ArrayList and Vector

ArrayList and Vector are array-based lists.

Internally, they use arrays to store their elements: whenever the array gets full, a new, bigger array is created, and the elements are copied to the new array.

Vector has higher overhead than ArrayList because Vector is synchronized to make it safe for use in programs with multiple threads.

AbstractSequentialList and LinkedList

Array-based lists have high overhead when elements are being inserted into the list, or removed from the list, at positions that are not at the end of the list.

LinkedList is a concrete class that stores elements in a way that eliminates the high overhead of adding to, and removing from positions in the middle of the list.

LinkedList extends AbstractSequentialList, which in turn, extends AbstractList.

Using the Concrete List Classes

- The concrete classes ArrayList, Vector, and LinkedList work in similar ways, but have different performance characteristics.
- Because they all implement the List interface, you can use List interface references to instantiate and refer to the different concrete classes.
- Using a List interface instead of the concrete class reference allows you to later switch to a different concrete class to get better performance.

Example: ArrayList

```
import java.util.*;
public class Test
{
  public  static void main(String [ ] args)
   {
     List<String> nameList = new ArrayList<String> ();
     String [ ] names = {"Ann", "Bob", "Carol"};

     // Add to arrayList
     for (int k = 0; k < names.length; k++)
        nameList.add(names[k]);

     // Display name list
     for (int k = 0; k < nameList.size(); k++)
        System.out.println(nameList.get(k));
   }
}
```

An Example: LinkedList

Because we used a List reference to refer to the concrete class objects, we can easily switch from an ArrayList to a LinkedList : the only change is in the class used to instantiate the collection.

Example: LinkedList

```
import java.util.*;
public class Test
{
  public  static void main(String [ ] args)
   {
     List<String> nameList = new LinkedList<String> ();
     String [ ] names = {"Ann", "Bob", "Carol"};

     // Add to arrayList
     for (int k = 0; k < names.length; k++)
        nameList.add(names[k]);

     // Display name list
     for (int k = 0; k < nameList.size(); k++)
        System.out.println(nameList.get(k));
   }
}
```

Using an Iterator

To use an iterator with a collection,

1. Call the iterator():Iterator<E> method of the collection to retrieve an iterator object.
2. Use the hasNext():boolean method to see if there still remain elements to be returned, and the next():E method to return the next available element.
3. If desired, use the remove():void method to remove the element returned by next().

The Iterator remove() method

- The remove() method removes the element returned by the last call to next().

- The remove() method can be called at most one time for each call to next().

Using an Iterator

```
List<String> nameList = new ArrayList<String>();
  String [ ] names = {"Ann", "Bob", "Carol"};
  // Add to arrayList
  for (int k = 0; k < names.length; k++)
    nameList.add(names[k]);

  // Display name list using an iterator
  Iterator<String> it = nameList.iterator();     // Get the iterator
  while (it.hasNext())                           // Use the iterator
    System.out.println(it.next());
```

ListIterator

The ListIterator extends Iterator by adding methods for moving backward through the list (in addition to the methods for moving forward that are provided by Iterator)

hasPrevious() : boolean

previous() : E

Some ListIterator Methods

Method	Description
add(el:E):void	Adds el to the list at the position just before the element that will be returned by the next call to the next() method.
hasPrevious():boolean	Returns true if a call to the previous() method will return an element, false if a call to previous() will throw an exception because there is no previous element.
nextIndex():int	Returns the index of the element that would be returned by a call to next(), or the size of the list if there is no such element.
previous():E	Returns the previous element in the list. If the iterator is at the beginning of the list, it throws NoSuchElementException.
previousIndex():int	Returns the index of the element that would be returned by a call to previous(), or -1.
set(el:E):void	Replaces the element returned by the last call to next() or previous() with a new element el.

Iterator Positions

Think of an iterator as having a cursor position that is initially just before the element that will be returned by the first call to next().

A call to next() puts the cursor just after the element returned, and just before the element that will be returned by the next call to next().

At any time, in a ListIterator, the cursor is in between two list elements: A call to previous() will skip backward and return the element just skipped, a call to next() will skip forward and and return the element just skipped.

Iterator and ListIterator Exceptions

A call to previous() throws NoSuchElementException when there is no element that can be skipped in a backward move.

A call to next() throws NoSuchElementException when there is no

element that can be skipped in a forward move.

Example Use of a ListIterator

```
public  static void main(String [ ] args)
 {
    List<String> nameList = new ArrayList<String>();
    String [ ] names = {"Ann", "Bob", "Carol"};

    // Add to arrayList using a ListIterator
    ListIterator<String> it = nameList.listIterator();
    for (int k = 0; k < names.length; k++)
       it.add(names[k]);

    // Get a new  ListIterator for printing
    it = nameList.listIterator();
    while (it.hasNext())
       System.out.println(it.next());
 }
```

Enhanced For Loop

The enhanced for loop can be used with any collection.

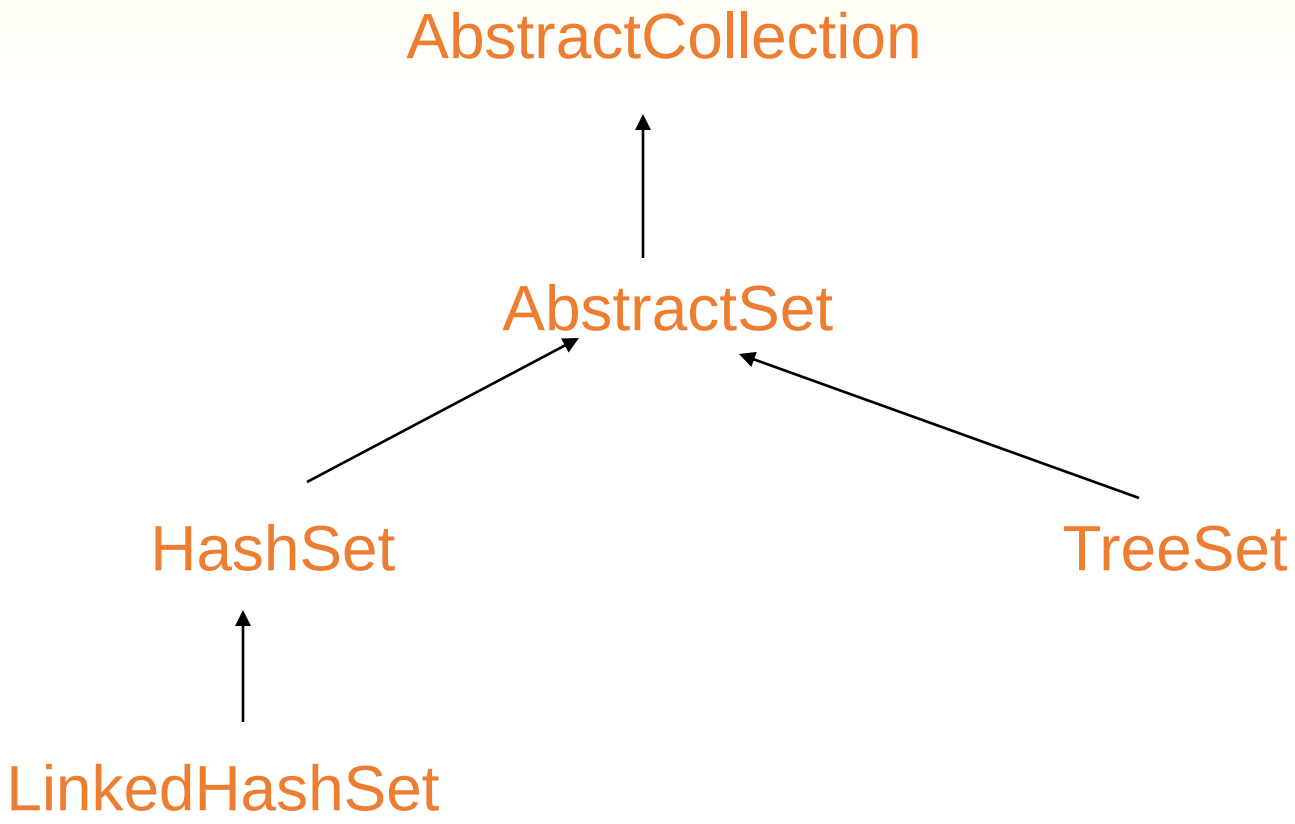
The compiler converts the enhanced for loop into a traditional loop that uses the collection's iterator.

Sets

Sets are collections that store elements, but have no notion of a position of an element within the collection.

The distinguishing feature of a set as a collection is that it does not allow duplicates.

The Set Part of the JCF Hierarchy



The Set Part of the JCF

AbstractSet implements the Set Interface.

TreeSet implements the SortedSet interface, which has methods for working with elements that have an order that allows them to be sorted according to their value.

HashSet

- HashSets store elements according to a hash code.
- A hash code of an element is an integer computed from the value of the element that can be used to help identify the element.
- The procedure used to compute the hash code of an element is called the hashing function or the hashing algorithm.

Examples of Hashing Functions

- For Integer objects, you can use the integer value of the object (or its absolute value).
- For Character objects, you can use the UNICODE value for the character.
- For String objects, you can use a function that takes into account the UNICODE values of the characters that make up the string, as well as the position occupied by each character.

A Simplistic Hashing Function

A very simple (but not very good) hashing function for strings might assign to each string the UNICODE value of its first character.

Note that all strings with the same first character are assigned the same hash code.

When two distinct objects have the same hash code, we say that we have a collision.

Implementation of a HashSet

- A HashSet can be regarded as a collection of "buckets."
- Each bucket corresponds to a hash code, and stores all objects in the set that have that particular hash code.
- Some buckets will have just one element, whereas other buckets may have many.
- A good hashing scheme should distribute elements among the buckets so that all buckets have approximately the same number of elements.

Implementation of a HashSet

The HashSet is a collection of buckets, and each bucket is a collection of elements.

The collection of buckets is actually a list of buckets, perhaps an ArrayList.

Each bucket may also be a list of elements, usually a linked list.

How a HashSet Works

- To add an element *X*, the hash code for *X* is used (as an index) to locate the appropriate bucket. *X* is then added to the list for that bucket. If *X* is already in the bucket (The test is done using the equals method), then it is not added.

- To remove an item *X*, the hash code for *X* is computed. The corresponding bucket is then searched for *X*, and if it is found, it is removed.

Efficiency of HashSet Operations

Given an item *X*, computing the hash code for *X* and locating the corresponding bucket can be done very fast.

The time to search for, or remove *X* from the bucket depends on how many elements are stored in the bucket.

More collisions mean more elements in some buckets, so we try to find a hashing scheme that minimizes collisions.

HashSet Performance Considerations

To have good performance with a HashSet:

1. Have enough buckets: fewer buckets means more collisions.
2. Have a good hashing function that spreads elements evenly among the buckets. This keeps the number of elements in each bucket small.

HashSet Capacity and Load Factor

- The load factor of a HashSet is the fraction of buckets that must be occupied before the number of buckets is increased.
- The number of buckets in a HashSet is called its capacity.

Some HashSet Constructors

HashSet()	Creates an empty HashSet object with a default initial capacity of 16 and load factor of 0.75.
HashSet(int initCapacity, float loadFactor)	Creates an empty HashSet object with the specified initial capacity and load factor.
HashSet(int initCapacity)	Creates an empty HashSet object with the specified initial capacity and a load factor of 0.75.

The hashCode() method

The Java Object class defines a method for computing hash codes

```
int hashCode()
```

This method should be overriden in any class whose instances will be stored in a HashSet.

The Object class's hashCode() method returns a value based on the memory address of the object.

Overriding the hashCode() Method

Observe these guidelines:

1. Objects that are equal according to their equals method should be assigned the same hash code.
2. Because of 1), whenever you override a class's equals() method, you should also override hashCode().
3. Try to minimize collisions.

HashSet Example 1

```
import java.util.*;

   /**
     This program demonstrates how to add elements
     to a HashSet. It also shows that duplicate
     elements are not allowed.
   */

   public class HashSetDemo1
   {
     public static void main(String[] args)
     {
       // Create a HashSet to hold String objects.
       Set<String> fruitSet = new HashSet<String>();

       // Add some strings to the set.
       fruitSet.add("Apple");
       fruitSet.add("Banana");
       fruitSet.add("Pear");
       fruitSet.add("Strawberry");

       // Display the elements in the set.
       System.out.println("Here are the elements.");
       for (String element : fruitSet)
         System.out.println(element);

       // Try to add a duplicate element.
       System.out.println("\nTrying to add Banana to " +
                          "the set again...");
       if (!fruitSet.add("Banana"))
         System.out.println("Banana was not added again.");

       // Display the elements in the set.
       System.out.println("\nHere are the elements once more.");
       for (String element : fruitSet)
         System.out.println(element);
     }
   }
```

A Car Class for Use With a HashSet

```
class Car
{
   String vin,  description;
   public boolean equals(Object other)       // Depends on vin only
   {
      if (!(other instanceof Car))
          return false;
      else
          return  vin.equalsIgnoreCase(((Car)other).vin);
   }

  public int hashCode() { return vin.hashCode();} // Depends on vin only

  public Car(String v, String d)  {  vin = v; description = d;  }
  public String toString()   {  return vin + " " + description;  }
}
```

A Car Class for use with a HashSet

Note that the Car class overrides both equals() and hashCode().

Use of the Car Class with a HashSet

```
public  static void main(String [ ] args)
 {
   Set<Car> carSet = new HashSet<Car>();
   Car [ ] myRides = {
                        new Car("TJ1", "Toyota"),
                        new Car("GM1", "Corvette"),
                        new Car("TJ1", "Toyota Corolla")
                      };
   // Add the cars to the HashSet
   for (Car c : myRides)
      carSet.add(c);

   // Print the list using an Iterator
   Iterator it = carSet.iterator();
   while (it.hasNext())
      System.out.println(it.next());
 }
```

HashSet<Car> Program Output

GM1 Corvette

TJ1 Toyota

Note:

- The iterator does not return items in the order added to the HashSet.
- The entry of the Toyota Corolla is rejected because it is equal to an entry already stored (same vin).

HashSet Example 2

```
import java.util.*;

/**
   This program creates a HashSet, adds some
   names to it, gets an iterator for the set,
   and searches the set for names.
*/

public class HashSetDemo2
{
   public static void main(String[] args)
   {
      // Create a HashSet to hold names.
      Set<String> nameSet = new HashSet<String>();

      // Add some names to the set.
      nameSet.add("Chris");
      nameSet.add("David");
      nameSet.add("Katherine");
      nameSet.add("Kenny");

      // Get an iterator for the set.
      Iterator it = nameSet.iterator();
```

HashSet Example 2

```
// Display the elements in the set.
        System.out.println("Here are the names in the set.");
        while (it.hasNext())
          System.out.println(it.next());

        System.out.println();

        // Search for "Katherine". We should find this
        // name in the set.
        if (nameSet.contains("Katherine"))
          System.out.println("Katherine is in the set.");
        else
          System.out.println("Katherine is NOT in the set.");

        // Search for "Bethany". We should not find
        // this name in the set.
        if (nameSet.contains("Bethany"))
          System.out.println("Bethany is in the set.");
        else
          System.out.println("Bethany is NOT in the set.");
    }
}
```

HashSet Example 3

```
/**
   The Car class stores a VIN (Vehicle Identification
   Number) and a description for a car.
*/

public class Car
{
   private String vin;         // Vehicle Identification Number
   private String description;  // Car description

   /**
      Constructor
      @param v The VIN for the car.
      @param desc The description of the car.
   */

   public Car(String v, String desc)
   {
      vin = v;
      description = desc;
   }

   /**
      getVin method
      @return The car's VIN.
   */

   public String getVin()
   {
      return vin;
   }
```

HashSet Example 3

```
/**
   getDescription method
   @return The car's description.
*/

public String getDescription()
{
   return description;
}

/**
   toString method
   @return A string containing the VIN and description.
*/

public String toString()
{
   return "VIN: " + vin +
          "\tDescription: " +
          description;
}

/**
   hashCode method
   @return A hash code for this car.
*/

public int hashCode()
{
   return vin.hashCode();
}
```

HashSet Example 3

```
/**
      equals method
      @param obj Another object to compare this object to.
      @return true if the two objects are equal, false otherwise.
   */

   public boolean equals(Object obj)
   {
      // Make sure the other object is a Car.
      if (obj instanceof Car)
      {
         // Get a Car reference to obj.
         Car tempCar = (Car) obj;

         // Compare the two VINs. If the VINs are
         // the same, then they are the same car.
         if (vin.equalsIgnoreCase(tempCar.vin))
            return true;
         else
            return false;
      }
      else
         return false;
   }
}
```

HashSet Example 3

```
import java.util.*;

/**
   This program stores Car objects in a HashSet and then
   searches for various objects.
*/

public class CarHashSet
{
   public static void main(String[] args)
   {
      // Create a HashSet to store Car objects.
      Set<Car> carSet = new HashSet<Car>();

      // Add some Car objects to the HashSet.
      carSet.add(new Car("227H54", "1997 Volkswagen"));
      carSet.add(new Car("448A69", "1965 Mustang"));
      carSet.add(new Car("453B55", "2007 Porsche"));
      carSet.add(new Car("177R60", "1980 BMW"));

      // Display the elements in the HashSet.
      System.out.println("Here are the cars in the set:");
      for (Car c : carSet)
         System.out.println(c);

      System.out.println();
```

HashSet Example 3

```
// Search for a specific car. This one is in the set.
      Car mustang = new Car("448A69", "1965 Mustang");
      System.out.println("Searching for " + mustang);

      if (carSet.contains(mustang))
        System.out.println("The Mustang is in the set.");
      else
        System.out.println("The Mustang is NOT in the set.");

      // Search for another car. This one is not in the set.
      Car plymouth = new Car("911C87", "2000 Plymouth");
      System.out.println("Searching for " + plymouth);

      if (carSet.contains(plymouth))
        System.out.println("The Plymouth is in the set.");
      else
        System.out.println("The Plymouth is NOT in the set.");
    }
  }
```

LinkedHashSet

A linkedHashSet is just a HashSet that keeps track of the order in which elements are added using an auxiliary linked list.

TreeSet

A TreeSet stores elements based on a natural order defined on those elements.

The natural order is based on the values of the objects being stored .

By internally organizing the storage of its elements according to this order, a TreeSet allows fast search for any element in the collection.

Order

An order on a set of objects specifies for any two objects *x* and *y*, exactly one of the following:

x is less than *y*

x is equal to *y*

x is greater than *y*

Examples of Natural Orders

Some classes have a “natural” order for their objects:

- Integer, Float, Double etc has the obvious concept of natural order which tells when one number is less than another.
- The String class has a natural alphabetic order for its objects.

The Comparable Interface

In Java, a class defines its natural order by implementing the Comparable interface:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The compareTo method returns a negative value, or zero, or a positive value, to indicate that the calling object is less than, equal to, or greater than the other object.

Using a TreeSet with Comparable Elements

1. Make sure the class of your objects implements Comparable.
2. Create an instance of TreeSet specialized for your class

   Set<String> mySet = new TreeSet<String>();
3. Add elements.
4. Retrieve elements using an iterator. The iterator will return elements in sorted order.

Sorting Strings Using a TreeSet

```
import java.util.*;
public class Test
{
   public  static void main(String [ ] args)
    {
      // Create TreeSet
      Set<String> mySet = new TreeSet<String>();
      // Add Strings
      mySet.add("Alan");
      mySet.add("Carol");
      mySet.add("Bob");
      // Get Iterator
      Iterator it = mySet.iterator();
      while (it.hasNext())
      {
         System.out.println(it.next());
      }
    }
}
```

The SortedSet Interface

TreeSet implements the SortedSet interface.

SortedSet methods allow access to the least and greatest elements in the collection.

SortedSet methods allow various views of the collection, for example, the set of all elements greater than a given element, or less than a given element.

Comparators

A comparator is an object that can impose an order on objects of another class.

This is different from the Comparable interface, which allows a class to impose an order on its own objects.

The Comparator Interface

```
Interface Comparator <T>
{
   int compare(T obj1, T obj2);
   boolean equals(Object o);
}
```

The compare(x, y) method returns a negative value, or zero, or a positive value, according to whether x is less than, equal to, or greater than y.

The equals method is used to compare one comparator object to another. It does not have to be implemented if the equals inherited from Object is adequate.

Using TreeSets with Comparators

A TreeSet that stores objects of a class that does not implement Comparable must use a comparator to order its elements.

The comparator is specified as an argument to the TreeSet constructor.

A comparator can also be used to make a TreeSet order its elements differently from their natural order.

A Comparator for Ordering Strings in Reverse Alphabetic Order

```
import java.util.*;
class RevStrComparator implements Comparator<String>
{
   public int compare(String s1, String s2)
   {
      return  - s1.compareTo(s2);     // Note the negation operator
   }
}
```

Using a TreeSet to Sort Strings in Reverse Alphabetic Order

```
public class Test
{
   public  static void main(String [ ] args)
    { // Create Comparator
       RevStrComparator comp = new RevStrComparator();
       Set<String> mySet = new TreeSet<String>(comp);
       // Add strings
       mySet.add("Alan");
       mySet.add("Carol");
       mySet.add("Bob");
       // Get Iterator
       Iterator it = mySet.iterator();
       while (it.hasNext())
       {
          System.out.println(it.next());
       }
    }
}
```

TreeSet Example 1

```
import java.util.*;

/**
   This program demonstrates how a TreeSet
   sorts its elements in ascending order.
*/

public class TreeSetDemo1
{
   public static void main(String[] args)
   {
      // Create a TreeSet and store some values in it.
      SortedSet<String> mySet = new TreeSet<String>();
      mySet.add("Pear");
      mySet.add("Apple");
      mySet.add("Strawberry");
      mySet.add("Banana");

      // Display the elements in the TreeSet.
      System.out.println("Here are the TreeSet elements " +
                         "in ascending order:");
      for (String str : mySet)
         System.out.println(str);

      // Add a new element to the TreeSet.
      System.out.println("\nAdding Blueberry to the set.");
      mySet.add("Blueberry");

      // Display the elements again.
      System.out.println("\nHere are the TreeSet elements " +
                         "again:");
      for (String str : mySet)
         System.out.println(str);
   }
}
```

TreeSet Example 2

```
import java.util.Comparator;

 public class CarComparator<T extends Car>
              implements Comparator<T>
{
   public int compare(T car1, T car2)
   {
     // Get the two cars' VINs.
     String vin1 = car1.getVin();
     String vin2 = car2.getVin();

     // Compare the VINs and return the
     // result of the comparison.
     return vin1.compareToIgnoreCase(vin2);
  }
}
```

TreeSet Example 2

```
import java.util.*;

/**
   This program demonstrates how a TreeSet
   can use a Comparator to sort its elements.
*/

public class TreeSetDemo2
{
   public static void main(String[] args)
   {
      // Create a TreeSet and pass an instance of
      // CarComparator to it.
      SortedSet<Car> carSet =
          new TreeSet<Car>( new CarComparator<Car>() );

      // Add some Car objects to the TreeSet.
      carSet.add(new Car("227H54", "1997 Volkswagen"));
      carSet.add(new Car("453B55", "2007 Porsche"));
      carSet.add(new Car("177R60", "1980 BMW"));
      carSet.add(new Car("448A69", "1965 Mustang"));

      // Display the elements in the TreeSet.
      System.out.println("Here are the cars sorted in " +
                         "order of their VINs:");
      for (Car car : carSet)
        System.out.println(car);
   }
}
```

Maps

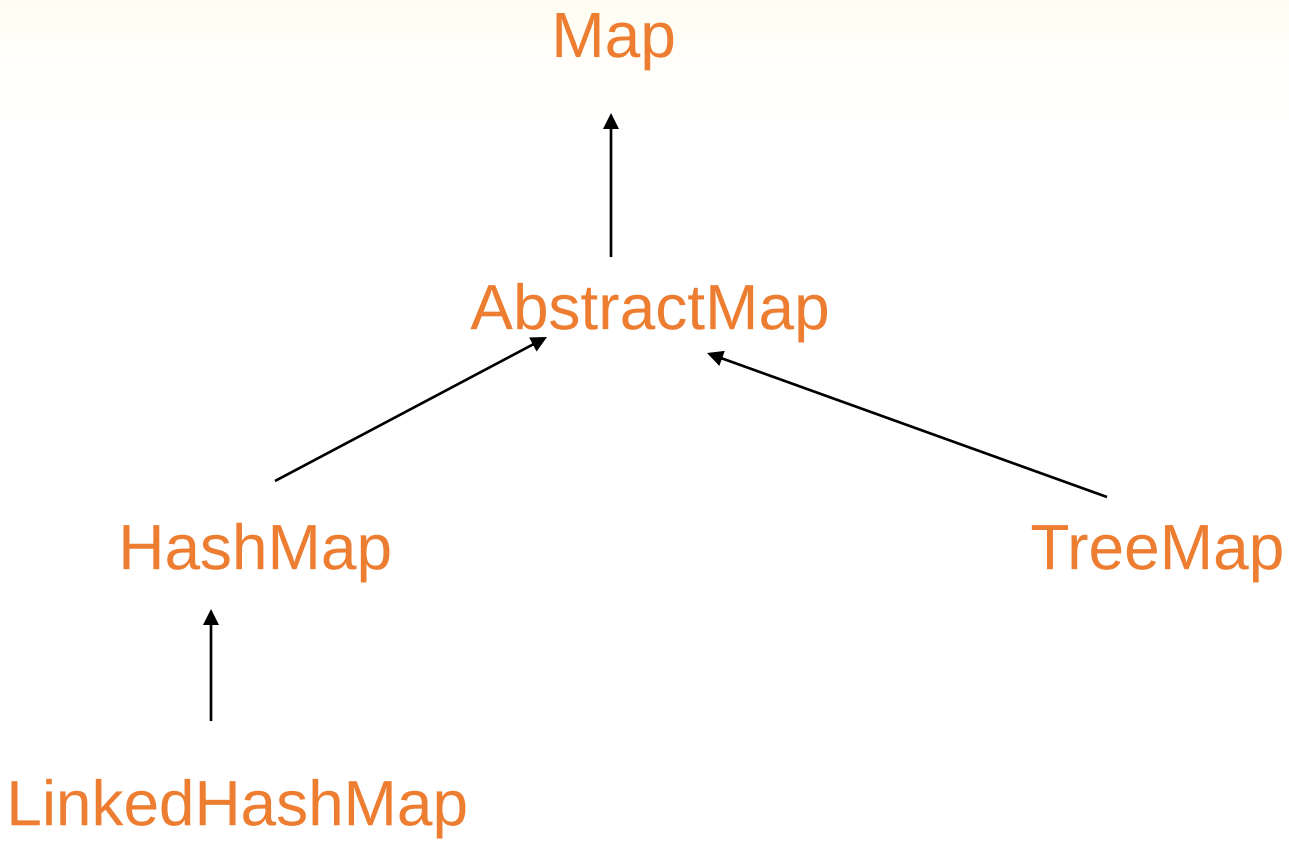
A map is a collection whose elements have two parts: a key and a value.

The combination of a key and a value is called a mapping.

The map stores the mappings based on the key part of the mapping, in a way similar to how a Set collection stores its elements.

The map uses keys to quickly locate associated values.

The Map Part of the JCF Hierarchy



The Map Interface

Map is a generic interface Map<K, V>

Map specifies two type parameters, K for the key, and V for the value part of the mapping.

Some Methods of the Map Interface

clear() : void	Removes all elements from the map.
containsValue(value: Object):boolean	Returns true if the map contains a mapping with the given value.
containsKey(key : Object) : boolean	Returns true if the map contains a mapping with the given key.
get(key : Object) : V	Returns the value associated with the specified key, or returns null if there is no such value.
isEmpty() : boolean	Returns true if the key contains no mappings.
keySet() : Set<K>	Returns the set of all keys stored in the map.

Some Methods of the Map Interface

put(key : K, value : V) : V	Adds a mapping that associates V with K, and returns the value previously associated with K. Returns null if there was no value associated with K.
remove(key : Object) : V	Removes the mapping associated with the given key from the map, and returns the associated value. If there is not such mapping, returns null.
size() : int	Returns the number of mappings in the map.
values() : Collection<V>	Returns a collection consisting of all values stored in the map.

Concrete Map Classes

Maps store keys with attached values. The keys are stored as sets.

- HashMap stores keys according to their hash codes, just like HashSet stores its elements.
- LinkedHashMap is a HashMap that can iterate over the keys in insertion order (order in which mappings were inserted) or in access order (order of last access).
- TreeMap stores mappings according to the natural order of the keys, or according to an order specified by a Comparator.

HashMap Example 1

```
import java.util.*;

/**
   This program stores mappings in a HashMap and then
   searches for various objects.
*/

public class CarHashMap1
{
   public static void main(String[] args)
   {
      // Create a HashMap to store Car objects.
      Map<String, Car> carMap =
            new HashMap<String, Car>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69", "1965 Mustang");
      Car porsche = new Car("453B55", "2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the HashMap. In each
      // mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);
```

HashMap Example 1

```
// Search for the Mustang by its VIN.
        System.out.println("\nSearching for the car with " +
                           "VIN " + mustang.getVin());
        Car foundCar = carMap.get(mustang.getVin());

        // If the car was found, display it.
        if (foundCar != null)
          System.out.println(foundCar);
        else
          System.out.println("The Mustang is NOT in the set.");

        // Search for another VIN. This one is not in the set.
        System.out.println("\nSearching for the car with " +
                           "VIN 911C87");
        foundCar = carMap.get("911C87");

        // If the car was found display it.
        if (foundCar != null)
          System.out.println(foundCar);
        else
          System.out.println("That car is NOT in the set.");
    }
  }
```

HashMap Example 2

```
import java.util.*;

/**
   This program retrieves a set of keys and a
   collection of values from a HashMap.
*/

public class CarHashMap2
{
   public static void main(String[] args)
   {
      // Create a HashMap to store Car objects.
      Map<String, Car> carMap =
            new HashMap<String, Car>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69","1965 Mustang");
      Car porsche = new Car("453B55","2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the HashMap. In each
      // mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);
```

HashMap Example 2

```
// Get a set containing the keys in this map.
      Set<String> keys = carMap.keySet();

      // Iterate through the keys, printing each one.
      System.out.println("Here are the keys:");
      for (String k : keys)
        System.out.println(k);

      // Get a collection containing the values.
      Collection<Car> values = carMap.values();

      // Iterate through the values, printing each one.
      System.out.println("\nHere are the values:");
      for (Car c : values)
        System.out.println(c);
   }
}
```

HashMap Example 3

```
import java.util.*;

/**
   This program retrieves the mappings from a HashMap
   as a Set of Map.Entry objects.
*/

public class CarHashMap3
{
   public static void main(String[] args)
   {
      // Create a HashMap to store Car objects.
      Map<String, Car> carMap =
            new HashMap<String, Car>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69", "1965 Mustang");
      Car porsche = new Car("453B55", "2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the HashMap. In each
      // mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);
```

HashMap Example 3

```
// Get a set containing the mappings in this map.
        Set<Map.Entry<String, Car>> cars = carMap.entrySet();

       // Iterate through the mappings, printing each one.
        System.out.println("Here are the mappings:");
        for (Map.Entry<String, Car> entry : cars)
        {
          System.out.println("Key = " + entry.getKey());
          System.out.println("Value = " + entry.getValue());
          System.out.println();
        }
    }
  }
```

HashMap Example 4

```
import java.util.*;

  /**
    This program retrieves the mappings from a
    LinkedHashMap as a Set of Map.Entry objects.
  */

  public class CarHashMap4
  {
    public static void main(String[] args)
    {
      // Create a LinkedHashMap to store Car objects.
      Map<String, Car> carMap =
            new LinkedHashMap<String, Car>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69", "1965 Mustang");
      Car porsche = new Car("453B55", "2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");
```

HashMap Example 4

```
// Put some mappings into the LinkedHashMap. In
      // each mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);

      // Get a set containing the mappings in this map.
      Set<Map.Entry<String, Car>> cars = carMap.entrySet();

      // Iterate through the mappings, printing each one.
      System.out.println("Here are the mappings:");
      for (Map.Entry<String, Car> entry : cars)
      {
        System.out.println("Key = " + entry.getKey());
        System.out.println("Value = " + entry.getValue());
        System.out.println();
      }
    }
  }
```

HashMap Example 5

```
import java.util.*;

/**
   This program displays the mappings stored in a
   TreeMap. The mappings are displayed in ascending
   key order.
*/

public class CarHashMap5
{
   public static void main(String[] args)
   {
      // Create a TreeMap to store Car objects.
      SortedMap<String, Car> carMap =
            new TreeMap<String, Car>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69","1965 Mustang");
      Car porsche = new Car("453B55","2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the TreeMap. In each
      //  mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);
```

HashMap Example 5

```
// Get a set containing the mappings in this map.
      Set<Map.Entry<String, Car>> cars = carMap.entrySet();

      // Iterate through the mappings, printing each one.
      System.out.println("Here are the mappings:");
      for (Map.Entry<String, Car> entry : cars)
      {
        System.out.println("Key = " + entry.getKey());
        System.out.println("Value = " + entry.getValue());
        System.out.println();
      }
    }
  }
```