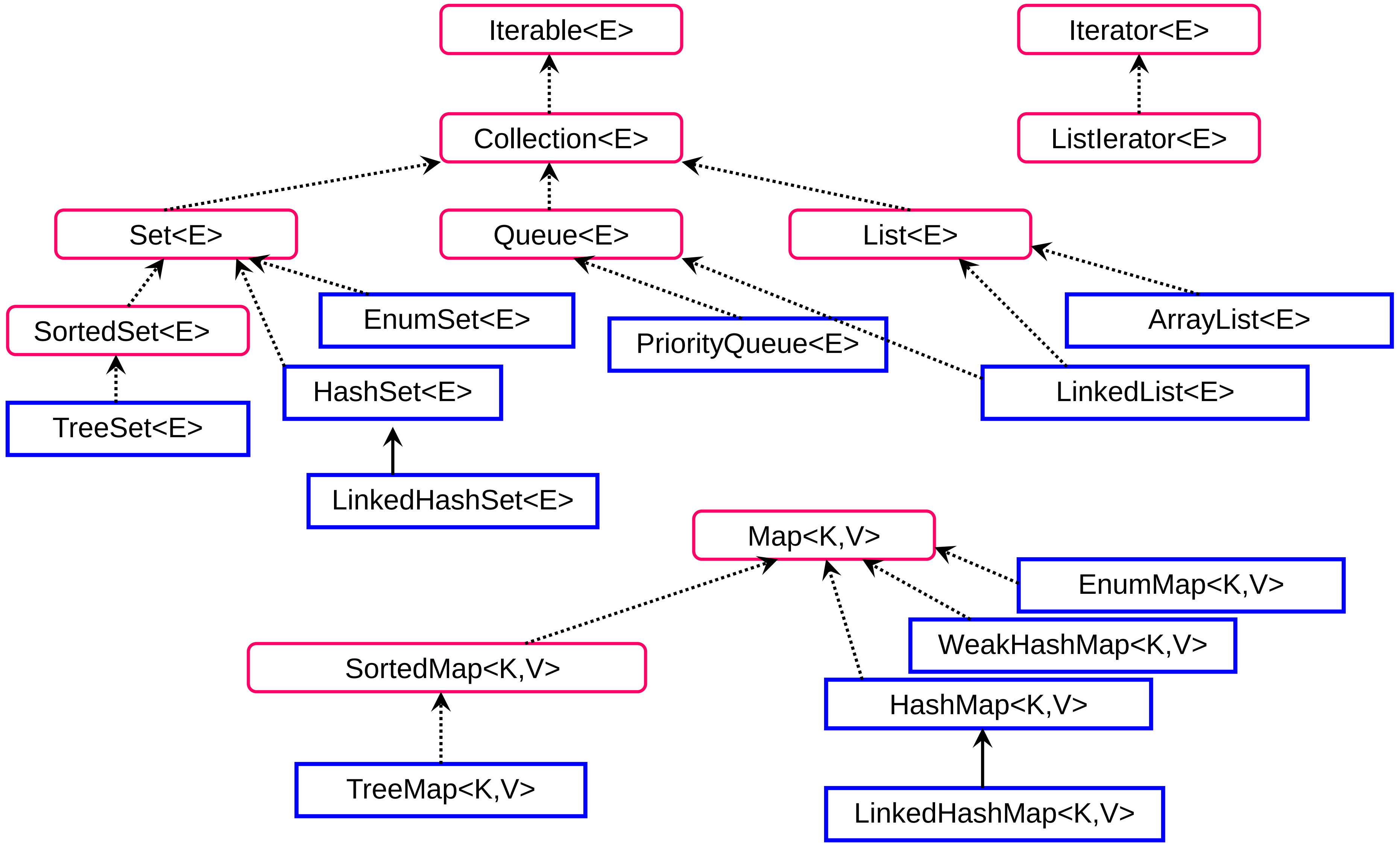
Chapter 12
Collections

Contents

Collections

- Collections are holders that let you store and organize objects in useful ways for efficient access.
- In the package java.util, there are interfaces and classes that provide a generic collection framework.
- The Collections interfaces: Collection<E>, Set<E>, SortedSet<E>, List<E>, Queue<E>, Map<K,V>, SortedMap<K,V>, Iterator<E>, ListIterator<E>, Iterable<E>
- Some useful implementations of the interfaces: HashSet<E>, TreeSet<E>, ArrayList<E>, LinkedList<E>, HashMap<K,V>, TreeMap<K,V>, WeakHashMap<K,V>
- Exception Convetions:
  - UnsupportedOperationException
  - ClassCastException
  - IllegalArgumentException
  - NoSuchElementException
  - NullPointerException

Type Trees for Collections



The Collection Interface

- **The Collection Interface**
  - The basis of much of the collection system is the Collection interface.
- **Methods:**
  - *public int size()*
  - *public boolean isEmpty()*
  - *public boolean contains(Object elem)*
  - *public Iterator<E> iterator()*
  - *public Object[] toArray()*
  - *public <T> T[] toArray(T[] dest)*
  - *public boolean add(E elem)*
  - *public boolean remove(Object elem)*
  - *public boolean containsAll(Collection<?> coll)*
  - *public boolean addAll(Collection<? extends E> coll)*
  - *public boolean removeAll(Collection<?> coll)*
  - *public boolean retainAll(Collection<?> coll)*
  - *public void clear()*

Safe in case the size of the collection has increased since the array was allocated

```
String[] strings = new
    String[collection.size()];
strings =
    collection.toArray(strings);

String[] strings =
    collection.toArray(new
    String[0]);
```

Allocate an array of exactly the right size

The Collections Framework

- The Java collection framework is a set of generic types that are used to create collection classes that support various ways to store and manage objects of any kind in memory.
- A generic type for collection of objects: To get static checking by the compiler for whatever types of objects to want to manage.

Generic Types

Generic Class/Interface Type	Description
The Iterator<T> **interface type**	**Declares methods for iterating through elements of a collection, one at a time.**
The Vector<T> **type**	**Supports an array-like structure for storing any type of object. The number of objects to be stored increases automatically as necessary.**
The Stack<T> **type**	**Supports the storage of any type of object in a pushdown stack.**
The LinkedList<T> **type**	**Supports the storage of any type of object in a doubly-linked list, which is a list that you can iterate though forwards or backwards.**
The HashMap<K,V> **type**	**Supports the storage of an object of type V in a hash table, sometimes called a map. The object is stored using an associated key object of type K. To retrieve an object you just supply its associated key.**

Collections of Objects

- Three Main Types of Collections
  - Sets
  - Sequences
  - Maps

- Sets
  - The simple kinds of collection
  - The objects are not ordered in any particular way.
  - The objects are simply added to the set without any control over where they go.

Collections of Objects

- Sequences
  - The objects are stored in a linear fashion, not necessarily in any particular order, but in an arbitrary fixed sequence with a beginning and an end.
  - Collections generally have the capability to expand to accommod ate as many elements as necessary.
  - The various types of sequence collections
    - Array or Vector
    - LinkedList
    - Stack
    - Queue

Collections of Objects

- Maps
  - Each entry in the collection involves a pair of objects.
  - A map is also referred to sometimes as a **dictionary**.
  - Each object that is stored in a map has an associated **key** object, and the object and its key are stored togeth er as a “name-value” pair.

Classes for Collections

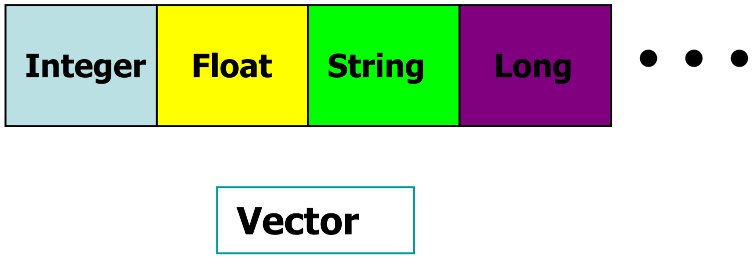
- **Classes in <u>Sets</u>:**
  - *HashSet<T>*
  - *LinkedHashSet<T>*
  - *TreeSet<T>*
  - *EnumSet<T extends Enum<T>>*
- **Classes in <u>Lists</u>:**
  - *To define a collection whose elements have a defined order-each element exists in a praticular poistion the collection.*
  - *Vector<T>*
  - *Stack<T>*
  - *LinkedList<T>*
  - *ArrayList<T>*
- **Class in <u>Queues</u>:**
  - *FIFO ordering*
  - *PriorityQueue<T>*
- **Classes in <u>Maps</u>:**
  - *Does not extend Collection because it has a contract that is different in important ways: do not add an element to a Map(add a key/value pair), and a Map allows looking up.*
  - *Hashtable<K,V>*
  - *HashMap<K,V>*
  - *LinkedHashMap<K,V>*
  - *WeakHashMap<K,V>*
  - *IdentityHashMap<K,V>*
  - *TreeMap<K,V> : keeping its keys sorted in the same way as TreeSet*

Vector (Before 1.5)

```
class VectorDemo {

  public static void main(String args[]) {

    // Create a vector and its elements
    Vector vector = new Vector();
    vector.addElement(new Integer(5));
    vector.addElement(new Float(-14.14f));
    vector.addElement(new String("Hello"));
    vector.addElement(new Long(120000000));
    vector.addElement(new Double(-23.45e-11));

    // Display the  vector elements
    System.out.println(vector);

    // Insert an element into the vector
    String s = new String("String to be inserted");
    vector.insertElementAt(s, 1);
    System.out.println(vector);

    // Remove an element from the vector
    vector.removeElementAt(3);
    System.out.println(vector);
  }
}
```

Result :

```
[5, -14.14, Hello, 120000000, -2.345E-10]
[5, String to be inserted, -14.14, Hello,
120000000, -2.345E-10]
[5, String to be inserted, -14.14, 120000000, -
2.345E-10]
```



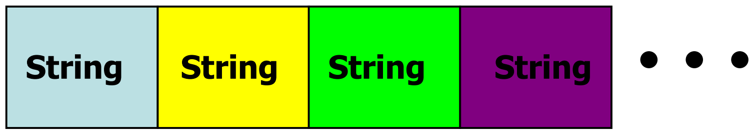
Vector (Using Generic Type) After 1.5

```
import java.util.*;

public class VectorDemo2 {
  public static void main(String args[]) {

    // Create a vector and its elements
    Vector<String> vector = new Vector<String>();
    vector.addElement(new String("One"));
    vector.addElement(new String("Two"));
    vector.addElement(new String("Three"));
    vector.addElement(new String("Four"));
    vector.addElement(new String("Five"));

    // Display the  vector elements
    System.out.println(vector);

    // Insert an element into the vector
    String s = new String("String to be inserted");
    vector.insertElementAt(s, 1);
    System.out.println(vector);

    // Remove an element from the vector
    vector.removeElementAt(3);
    System.out.println(vector);
  }
}
```

- The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
- Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

Result :

```
[One, Two, Three, Four, Five]
[One, String to be inserted, Two, Three, Four, Five]
[One, String to be inserted, Two, Four, Five]
```



Vector<String>

Using ArrayList

```
import java.util.*;

public class ArrayListDemo {
  public static void main(String args[]) {
    // Create a vector and its elements
    ArrayList<String> alist = new ArrayList<String>();
    alist.add(new String("One"));
    alist.add(new String("Two"));
    alist.add(new String("Three"));
    alist.add(new String("Four"));
    alist.add(new String("Five"));

    // Display the  vector elements
    System.out.println(alist);

    // Insert an element into the vector
    String s = new String("String to be inserted");
    alist.add(1, s);
    System.out.println(alist);

    // Remove an element from the vector
    alist.remove(3);
    System.out.println(alist);
  }
}
```

- Resizable-array implementation of the List interface.
- Each ArrayList instance has a capacity. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
- An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

Result :

```
[One, Two, Three, Four, Five]
[One, String to be inserted, Two, Three, Four, Five]
[One, String to be inserted, Two, Four, Five]
```

Hashtable (Before 1.5)

```
class HashtableDemo {

  public static void main(String args[]) {

    Hashtable hashtable = new Hashtable();
    hashtable.put("apple", "red");
    hashtable.put("strawberry", "red");
    hashtable.put("lime", "green");
    hashtable.put("banana", "yellow");
    hashtable.put("orange", "orange");

    Enumeration e = hashtable.keys();
    while(e.hasMoreElements()) {
      Object k = e.nextElement();
      Object v = hashtable.get(k);
      System.out.println("key = " + k +
         "; value = " + v);
    }
    System.out.print("\nThe color of an apple is: ");
    Object v = hashtable.get("apple");
    System.out.println(v);
  }
}
```

Key

Value

The Hashtable<k,V> class inherits from the Dictionary class, and implement the Map interface. All methods of it are synchronized, unlike HashMap.

Result :

```
Result #2
key = lime; value = green
key = strawberry; value = red
The color of an apple is: red
```

Here, you will meet warning message of unchecked type. How can we solve this?

Hashtable<K,V> (1.5)

```
import java.util.*;
class HashtableDemoGen {
  public static void main(String args[]) {
    Hashtable<String,String> hashtable = new
Hashtable<String,String>();
    hashtable.put("apple", "red");
    hashtable.put("strawberry", "red");
    hashtable.put("lime", "green");
    hashtable.put("banana", "yellow");
    hashtable.put("orange", "orange");

    for (Enumeration<String> e = hashtable.keys() ;
e.hasMoreElements() ;) {
      String k = e.nextElement();
      String v = hashtable.get(k);
      System.out.println("key = " + k +
         "; value = " + v);
     }

    System.out.print("\nThe color of an apple is: ");
    String v = hashtable.get("apple");
    System.out.println(v);
  }
}
```

Parameterized Type

The **Hashtable<k,V>** class inherits from the Dictionary class, and implement the Map interface. All methods of it are synchronized, unlike HashMap.

Enhanced For Loop

- **Using the enhanced for statement**
  for (Type iterate-variable : set-expression) statement;
- Here, the *set-expression* must either evaluate to an array instance, or an object implement the interface **java.lang.Iterable**.

```
class EnhancedForDemo {
  public static void main(String[] args){
     int[] numbers = {1,2,3,4,5} ;
      for (int item : numbers) {
         System.out.println(“Count : “ + item);
      }
  }
}
```

- **Class Vector<E>**

java.lang.Object -->
java.util.AbstractCollection<E> -->
java.util.AbstractList<E> -->
java.util.Vector<E>

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

```
import java.util.Vector;
public class EnhanceForVector {
  public static void main(String[] args){
     Vector<String> vec = new
      Vector<String>();
     vec.addElement("Apple");
      vec.addElement("Orange");
     vec.addElement("Grape");
      for (String item : vec) {
         System.out.println("Item : " + item);
      }
  }
}
```

- **Is the below possible?**

```
MyVector myvec = MyVector();
for (String item : vec) { …. }
```

→ **It should be “Iterable” type**

Iterable and Iterator Interface, and Creating Iterable Class

- **Iterable<T> interface**
  - T iterator()
- **Iterator<E> interface**
  - *E next()*
  - *boolean hasNext()*
  - *void remove()*

- **To create an "Iterable Class"**
  - An Iterable class implements the "Iterable" interface
  - The "iterator" method should be implemented in the class
  - An Iterator class should be provided for the iterator. The Iterator class has 3 methods, hasNext, next, and remove to access elements of the class

```
class Cage<T> implements Iterable<T> {
    fields and methods for the Cage class

    public Iterator<T> iterator() {
      return new SomeIterator();
    }
}

class SomeIterator implements Iterator<T>
     {
     public T next() {
      //  body of the next method
     }

     public boolean hasNext() {
      //  body of the hasNext method
     }

     public void remove() {
      //  body of the remove method
     }
}
```

Writing Iterator Implementations

"ShortStrings.java"

```
import java.util.*;

public class ShortStrings implements Iterator<String>
       {
  private Iterator<String> strings ;  // source for strings
  private String nextShort;  // null if next not known
  private final int maxLen;  // only return strings <=

  public ShortStrings(Iterator<String> strings, int maxLen)
       {
    this.strings = strings;
    this.maxLen = maxLen;
    nextShort = null;
  }

  public boolean hasNext() {
    if (nextShort != null)  // found it already
      return true;
    while (strings.hasNext()) {
      nextShort = strings.next();
      if (nextShort.length() <= maxLen) return true;
    }
    nextShort = null;   // did not find one
    return false;
  }
```

Result:

```
Short String = First String
Short String = Second Second String
Short String = Third Third Third String
```

```
public String next() {
    if (nextShort == null && !hasNext())
      throw new NoSuchElementException();
    String n = nextShort;  // remember nextShort
    nextShort = null;
    return n;
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

"ShortStringsTest.java"

```
import java.util.*;
public class ShortStringsTest {
  public static void main(String[] args) {
    LinkedList<String> myList = new LinkedList<String>();
    myList.add("First String");
    myList.add("Second Second String");
    myList.add("Third Third Third String");
    myList.add("Fourth Fourth Fourth Fourth String");
    myList.add("Fifth Fifth Fifth Fifth Fifth String");

    ShortStrings myShort = new ShortStrings(myList.iterator(),
       25);
    // for (String val : myShort) // Why not able ?
    while(myShort.hasNext()) {
      System.out.println("Short String = " + myShort.next());}
    } }
}
```

Example of Creating Iteratable Class

```
class MyArray implements Iterable<String> {

    private String[] v = new String[10];
    private int ptr = 0;

    public void add(String t) {
        v[ptr++] = t;
    }

    public String get(int i) {
        String a = v[ptr];
        return a;
    }

    public int getSize() {
        return ptr;
    }

    public Iterator<String> iterator() {
        return new MyIterator();
    }

  private class MyIterator implements Iterator<String> {
      int idx;
      // Constructor
      public MyIterator() {
        idx = 0;
      }

      public boolean hasNext() {
        return idx < ptr ;
      }

      public String next() {
        if(idx >= ptr)
          throw new java.util.NoSuchElementException();

        String element = v[idx];
        idx++;
        return element;
      }

      public void remove() {
        // we think there will not be remove invocation.
      }
  } // end of MyIterator

} // end of MyArray

public class IteratorExample {
     public static void main(String[] args) {
      MyArray str = new MyArray();

      str.add("This");      str.add("is");
      str.add("a");         str.add("test");
      str.add("string.");

      for(String s : str)
        System.out.print(s + " ");
      System.out.println();
    }
}
```

Comparable and Comparator

- The interface java.lang.Comparable<T> can be implemented by any class whose objects can be sorted.
- This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's ***natural ordering***, and the class's compareTo method is referred to as its ***natural comparison method***.
  - public int compareTo (T other): return a value that is less than, equal to, or greater than zero as this object is less than, equal to, or greater than the other object.
- If a given class does not implement Comparable or if its natural ordering is wrong for some purpose, java.util.Comparator object can be used
  - public int compare(T o1, T o2)
  - boolean equals(Object obj)

Sortable Class: TryManSortable Example

```
class SortableMen implements
    Comparable<SortableMen> {
  private String name;
  private Integer age;
  public SortableMen (String n, Integer a)
  {   name = n;   age = a;  }
  public String getName() {  return name;  }
  public Integer getAge() {   return age;  }
  public String toString() {   return new
    StringBuffer(128).append("Name:
    ").append(name).append(", Age:
    ").append(age.toString()).toString();
  }
  public void setName(String n) {  name =
    n;  }
  public void setAge(Integer a) {   age =
    a;  }
  public int compareTo(SortableMen v)
  {   int result;
    result =
    name.compareTo(v.getName());
    return result == 0 ?
    age.compareTo(v.getAge()) : result ;
  }
}
```

```
import java.util.Arrays;
public class TryMenSort {
public static void main(String[] args) {
SortableMen[] srtm = {
new SortableMen("Man D",25),
new SortableMen("Man A",25),
new SortableMen("Man A",30),
new SortableMen("Man B",25)
                    };
System.out.println("Original Order: ");
for (SortableMen m : srtm)
    System.out.println(m);
Arrays.sort(srtm);
System.out.println("\nOrder after sorting
    using Comparable method: ");
for (SortableMen m : srtm)
    System.out.println(m);
}
}
```

- **In Arrays class, a sort method**
  - *public static <T> void sort(T[] a, Comparator<? super T> c)*