Java Programming II

Collections

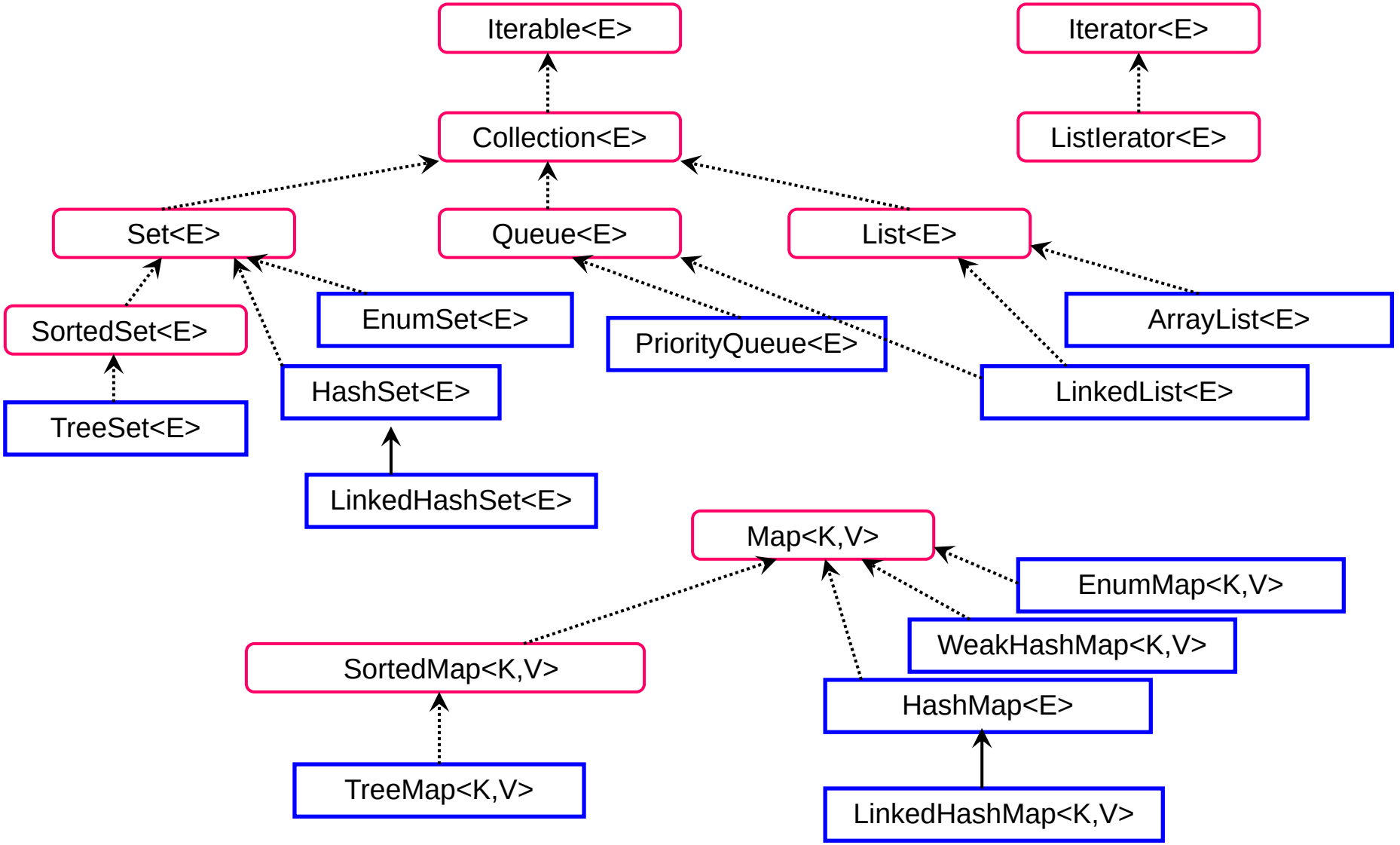
Contents

- Collections, Iteration
- The Collection Interface
- Set and SortedSet
- List
- Queue
- Map and SortedMap
- Enum Collections
- Wrapped Collections and the Collections Class
- Writing Iterator Implementations
- The Legacy Collection Types

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 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 particul ar order, but in an arbitrary fixed sequence with a beginning and an end.
  - Collections generally have the capability to expand to accommodate 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 together as a “name-value” pair.

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

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

Comparable and Comparator

- The interface java.lang.Comparable<T> can be implemented by any class whose objects can be sorted.
  - 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)

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)*

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

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

  - *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()*

Collection Classes

- **Classes in Sets:**
  - *HashSet<T>*
  - *LinkedHashSet<T>*
  - *TreeSet<T>*
  - *EnumSet<T extends Enum<T>>*
- **Classes in Lists:**
  - *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 Queues:**
  - *FIFO ordering*
  - *PriorityQueue<T>*
- **Classes in Maps:**
  - *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*

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;
  }
  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();
  }
}
```

Result:

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

"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 Iterator 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();
   }
}
```

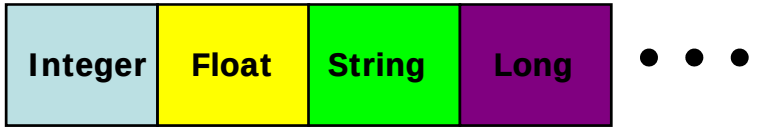
The Legacy Collection Types

- Enumeration
  - Analogous to Iterator.
- Vector
  - Analogous to ArrayList, maintains an ordered list of elements that are stored in an underlying array.
- Stack
  - Analogous of Vector that adds methods to push and pop elements.
- Dictionary
  - Analogous to the Map interface, although Diectionary is an abstract class, not an interface.
- Hashtable
  - Analogous HashMap.
- Properties
  - A subclass of Hashtable. Maintains a map of key/value pairs where the keys and values are strings. If a key is not found in a properties object a “default” properties object can be searched.

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

Vector (Using Generic Type)

```
import java.util.Vector;
import java.util.ListIterator;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

class Person {
  // Constructor
  public Person(String firstName, String surname) {
    this.firstName = firstName;
    this.surname = surname;
  }

  public String toString() {
    return firstName + " " + surname;
  }

  private String firstName;          // First name of person
  private String surname;            // Second name of person
}


public class TryVector {
  public static void main(String[] args) {
    Person aPerson = null;           // A person object
    Vector<Person> filmCast = new Vector<Person>();

    // Populate the film cast
    for( ; ; ) {                     // Indefinite loop
      aPerson = readPerson();        // Read in a film star
      if(aPerson == null) {          // If null obtained...
        break;                       // We are done...
      }
      filmCast.add(aPerson);         // Otherwise, add to the cast
    }

    int count = filmCast.size();
    System.out.println("You added " + count +
      (count == 1 ? " person":  " people") + " to the cast.¥n");
    System.out.println("The vector currently has room for "
            + (filmCast.capacity() - count) + " more people.¥n");

// Show who is in the cast using an iterator
    ListIterator<Person> thisLot = filmCast.listIterator();

    while(thisLot.hasNext()) {       // Output all elements
      System.out.println( thisLot.next());
    }
  }

  // Read a person from the keyboard
  static Person readPerson() {
    // Read in the first name and remove blanks front and back
    String firstName = null;
    String surname = null;
    System.out.println(
                  "¥nEnter first name or ! to end:");
    try {
      firstName = keyboard.readLine().trim();    // Read and trim a string

      if(firstName.charAt(0) == '!') {           // Check for ! entered
        return null;
      }                                          // If so, we are done...

      // Read in the surname, also trimming blanks
      System.out.println("Enter surname:");
      surname = keyboard.readLine().trim();      // Read and trim a string
    } catch(IOException e) {
      System.err.println("Error reading a name.");
      e.printStackTrace();
      System.exit(1);
    }
    return new Person(firstName,surname);
  }


  static BufferedReader keyboard = new BufferedReader(new
          InputStreamReader(System.in));
}
```

Try TryVector.java

What are differences to those of the V1.4?

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.

Miscellaneous Utilities

- **Formatter** – A class for producing formatted text.
- **BitSet** – A dynamically sized bit vector
- **Observer/Observable** – An interface/class pair that enables an object to be observable by having one or more Observer objects that are notified when something interesting happens in the Observable object.
- **Random** – A class to generate sequences of pseudorandom numbers.
- **Scanner** – A class for scanning text and parsing it into values of primitive types or strings, based on regular expression patterns.
- **StringTokenizer** – A class that splits a string into tokens based on delimiters(by default, whitespace)
- **Timer/TimerTask** – A way to schedule tasks to be run in the future.
- **UUID** – A class that represents a universally unique identifier(UUID)
- **Math** – A class performing basic mathematical operations, such as trigonometric functions, exponentiation, lograithms, and so on.
- **StricMath** – Defines the same methods as Math but guarantees the sue of specific algorithms that ensure the same results on every virtual machine.