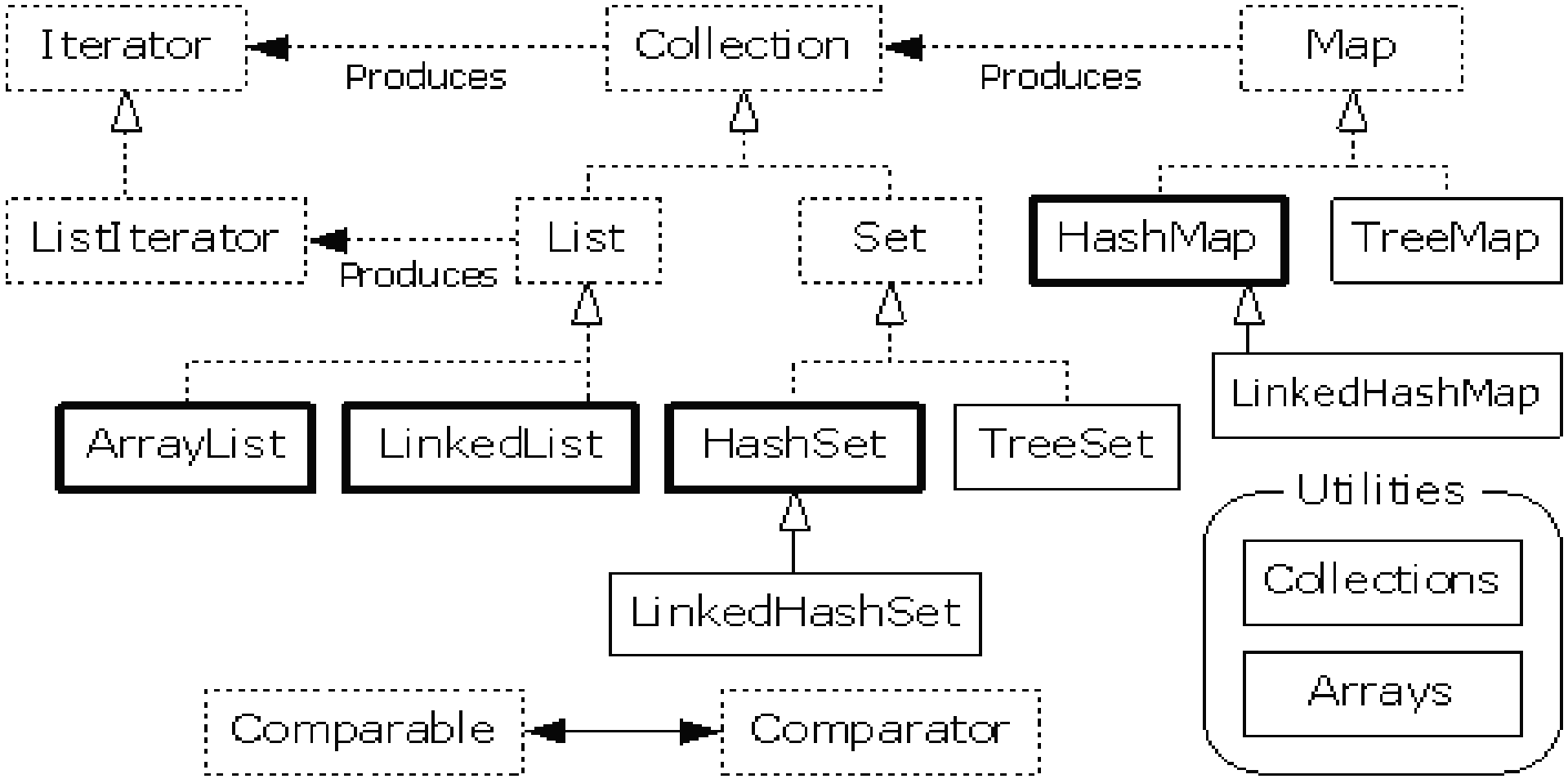
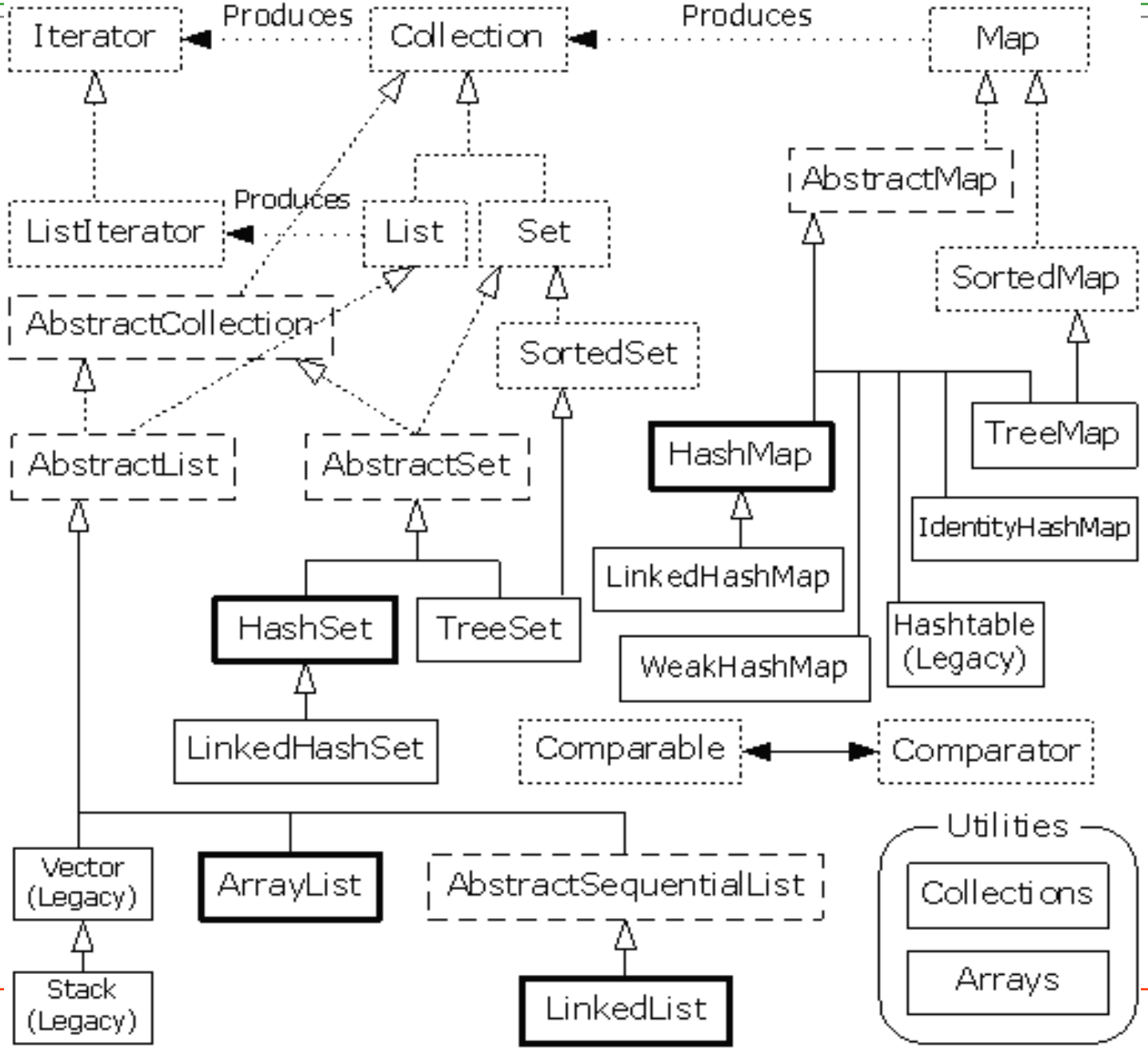
Chapter 5. Collection of objects

Metodología y Desarrollo de Programas

Collections' Outline

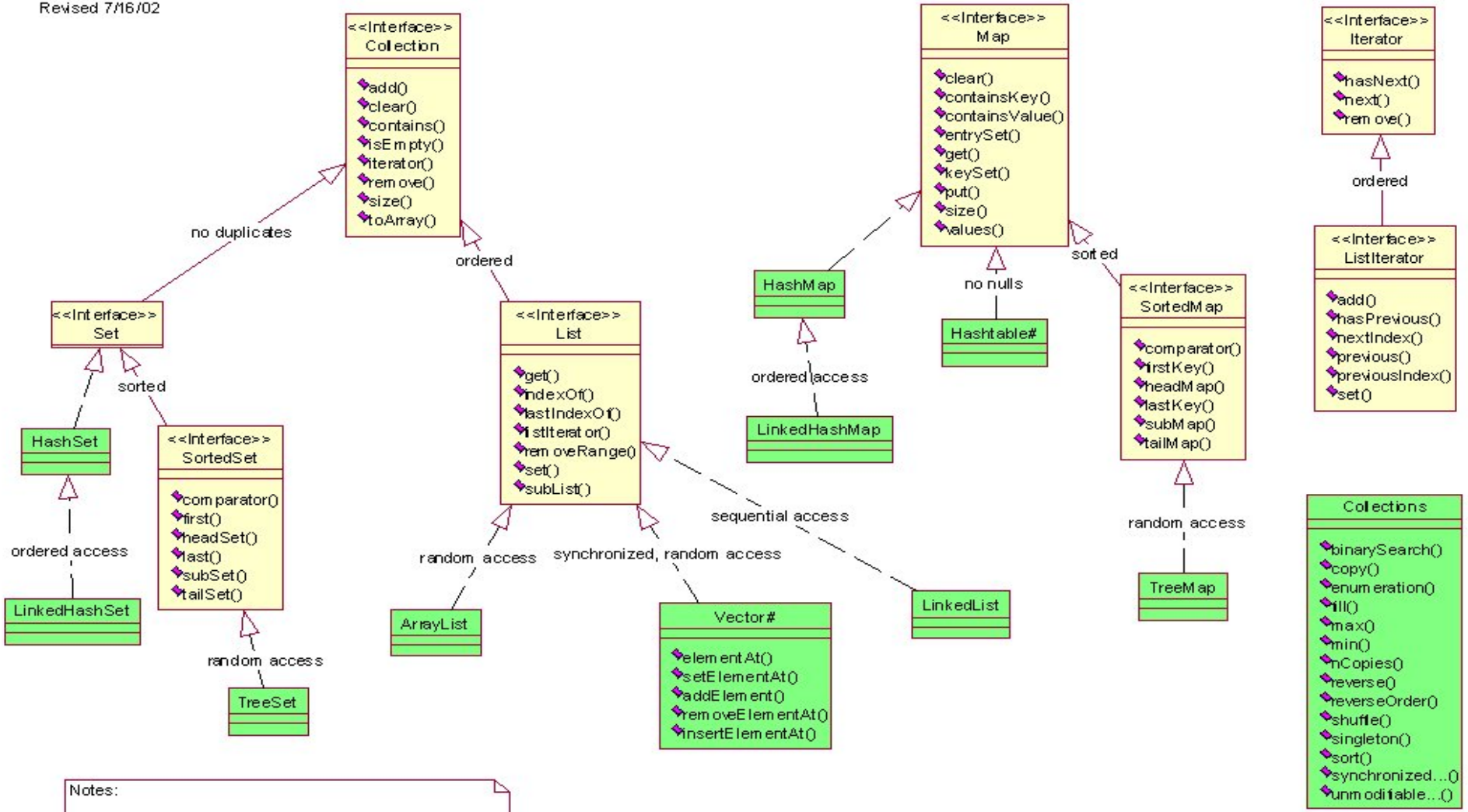


Collections' Outline in depth



Java Collections Framework

Revised 7/16/02



Notes:

# denotes class is a legacy class.

Differences between Java and Java2:
* Non-legacy collections are not synchronized.
* Iterator modifications change the backing collection.

Not all classes, interfaces, or methods are shown.

Chapter 6: Collection of objects

- In actual programs, always there is need to create and manipulate objects
  - Number of objects is not known at compile time
  - They may grow or shrink
- We need some utilities to store collection of objects
- Arrays are not a general solution
- Java provides utility classes which are containers for objects
- Arrays are efficient, typed and can store primitive types
- However,
  - Size is fixed (can't grow or shrink)
  - Arrays are useful when maximum number of objects can be guessed.

- **Solutions**: For more general situations we need to use other form of containers like **Lists**, **Sets** and **Maps**

Arrays are first-class objects

```
    // Arrays of primitives:
    int[] e; // Null reference
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Compile error: variable e not initialized:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // The primitives inside the array are
    // automatically initialized to zero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
  }
} ///:~
```

```
public class ArraySize {
  public static void main(String[] args) {
    // Arrays of objects:
    Weeble[] a; // Local uninitialized variable
    Weeble[] b = new Weeble[5]; // Null references
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      if(c[i] == null) // Can test for null reference
        c[i] = new Weeble();
    // Aggregate initialization:
    Weeble[] d = {
      new Weeble(), new Weeble(), new Weeble()
    };
    // Dynamic aggregate initialization:
    a = new Weeble[] {
      new Weeble(), new Weeble()
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // The references inside the array are
    // automatically initialized to null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);
  }
} ///:~
```

Returning an array

- As like any other object, a method can return an array as its result
- The method can create array and initialize it
- Remember that garbage collector takes care of objects on the heap, so it does not matter if the array is created inside a method, when there is no need for that, garbage collector destroys it

Returning an array - example

```
import java.util.*;

public class IceCream {
  private static Random rand = new Random();
  public static final String[] flavors = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl","Mint Chip", "Mocha Almond Fudge",
"Rum Raisin", "Praline Cream", "Mud Pie"
  };
  public static String[] flavorSet(int n) {
    String[] results = new String[n];
    boolean[] picked = new boolean[flavors.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(flavors.length);
      while(picked[t]);
      results[i] = flavors[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String[] fl = flavorSet(flavors.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

Arrays class

- To support some operations on arrays, Java provides a utility class named **Arrays (java.util.Arrays)**
- It has methods like:
  - *fill()* to fill an array with a value
  - *sort()* to sort an array
  - *binarySearch()* to search an array for an element
  - *Equals()* to test two arrays for having same elements
  - *asList()* to convert an array into a *List* container.

- Eckel provides Arrays2 class that provides some other usfull operations on arrays like toString().
- We can use System.arrayCopy() to copy contents of one array to another one

Arrays operations example

– Example: **Fill** and **System.arrayCopy()**

```
import java.util.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    System.out.println("i = " + Arrays2.toString(i));
    System.out.println("j = " + Arrays2.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    System.out.println("j = " + Arrays2.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    System.out.println("k = " + Arrays2.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    System.out.println("i = " + Arrays2.toString(i));
    // Objects:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    System.out.println("u = " + Arrays.asList(u));
    System.out.println("v = " + Arrays.asList(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    System.out.println("u = " + Arrays.asList(u));
  }
} ///:~
```

Inverse an Array

```
import java.util.*;

public class InvertirLista {
  public static void main(String[] args) {
    List l = Arrays.asList(args);
    Collections.reverse(l);
    System.out.println(l);
  }
}
```

Searching a sorted array

- Once an array is sorted, you can perform a fast search for a particular item by using **Arrays.binarySearch( ).**

```
import java.util.*;

public class ArraySearching {
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2.RandIntGenerator gen =
      new Arrays2.RandIntGenerator(1000);
    Arrays2.fill(a, gen);
    Arrays.sort(a);
    System.out.println(
      "Sorted array: " + Arrays2.toString(a));
    while(true) {
      int r = gen.next();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        System.out.println("Location of " + r +
          " is " + location + ", a[" +
          location + "] = " + a[location]);
        break; // Out of while loop
      }
    }
  }
} ///:~
```

Sorting arrays

- This code uses default **compareTo** of String class (of **comparable interface)**
- Default compareTo for String class uses lexicographic ordering:all uppercase letters first and then all lower case letters

```
public class StringSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa, new Arrays2.RandStringGenerator(5));
    System.out.println("Before sorting: " + Arrays.asList(sa));
    Arrays.sort(sa);
    System.out.println("After sorting: " + Arrays.asList(sa));
  }
} ///:~
```

Comparable Interface

- In some occasions, it is necessary to be able to compare (to sort) the information.
- With the simple datatype, this order is made by Java, however for each class in our system, we have to specify its order by means of a **Comparable Interface.**
- It is formed by a method called **compareTo**, that returns an integer value indicating if an object is equal, high or small than another object.

```
public interface Comparable {
    public int compareTo(Object o);
}
```

Sorting arrays. Comparable Interface

```
public class Persona implements Comparable{
  protected int dni;
              .....
  public int compareTo (Object otro){
      Persona otroP = (Persona)otro;
      if (dni >otroP.dni) return 1;
              else if (dni==otroP.dni) return 0
                        else return -1;
  }
}
public int compareTo(Object o) { //Definido como String
        Persona persona = (Persona)o;
        return this.apellidos.compareTo(persona.apellidos);
    }
```

```
//Main Programa
Persona[] plantilla;
 … //add information
Arrays.sort(Plantilla);
```

Sorting arrays – different ordering. Interface `Comparator`

- However, in some ocassions it is necessary to specify another kind of comparison, different from the natural one.--> **Comparator Interface** by means of the **compare** method.
- java.util.comparator

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

Sorting arrays – different ordering

```
public class AlphabeticComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(s2.toLowerCase());
  }
} ///:~
```

```
  public class AlphabeticSorting {
    public static void main(String[] args) {
      String[] sa = new String[30];
      Arrays2.fill(sa, new Arrays2.RandStringGenerator(5));
      System.out.println("Before sorting: " + Arrays.asList(sa));
      Arrays.sort(sa, new AlphabeticComparator());
      System.out.println("After sorting: " + Arrays.asList(sa));
    }
  } ///:~
```

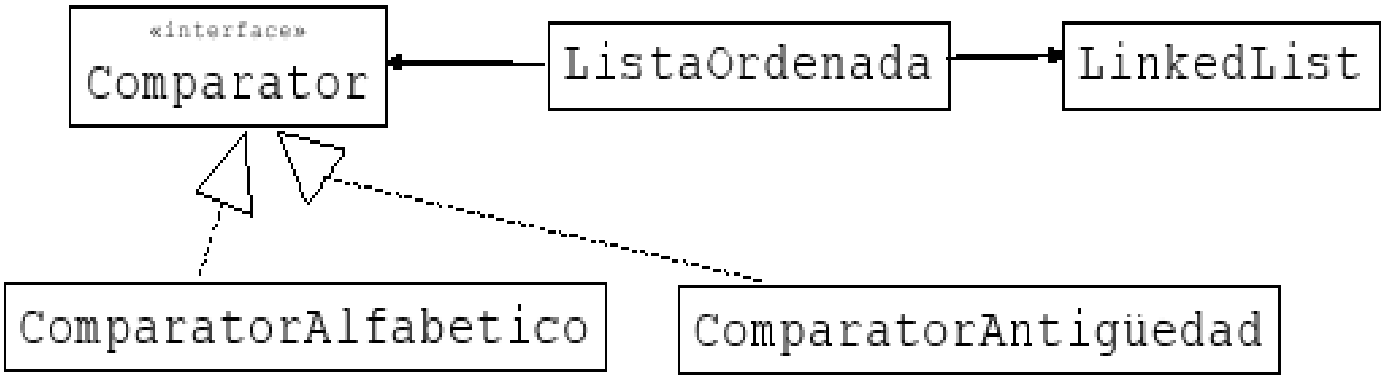
Interfaz `Comparator`

- If we want to sort by Birthdate

```
public class NacimientoPersonaComparator implements Comparator {

public int compare(Object o1, Object o2) {
        Persona persona1 = (Persona)o1;
        Persona persona2 = (Persona)o2;
        return persona1.getFechaNacimiento().
                compareTo(persona2.getFechaNacimiento());

    }
}

Persona[] plantilla;
 … //add information
Arrays.sort(Persona,new NacimientoPersonaComparator());
```

Interfaz Comparable-Comparator

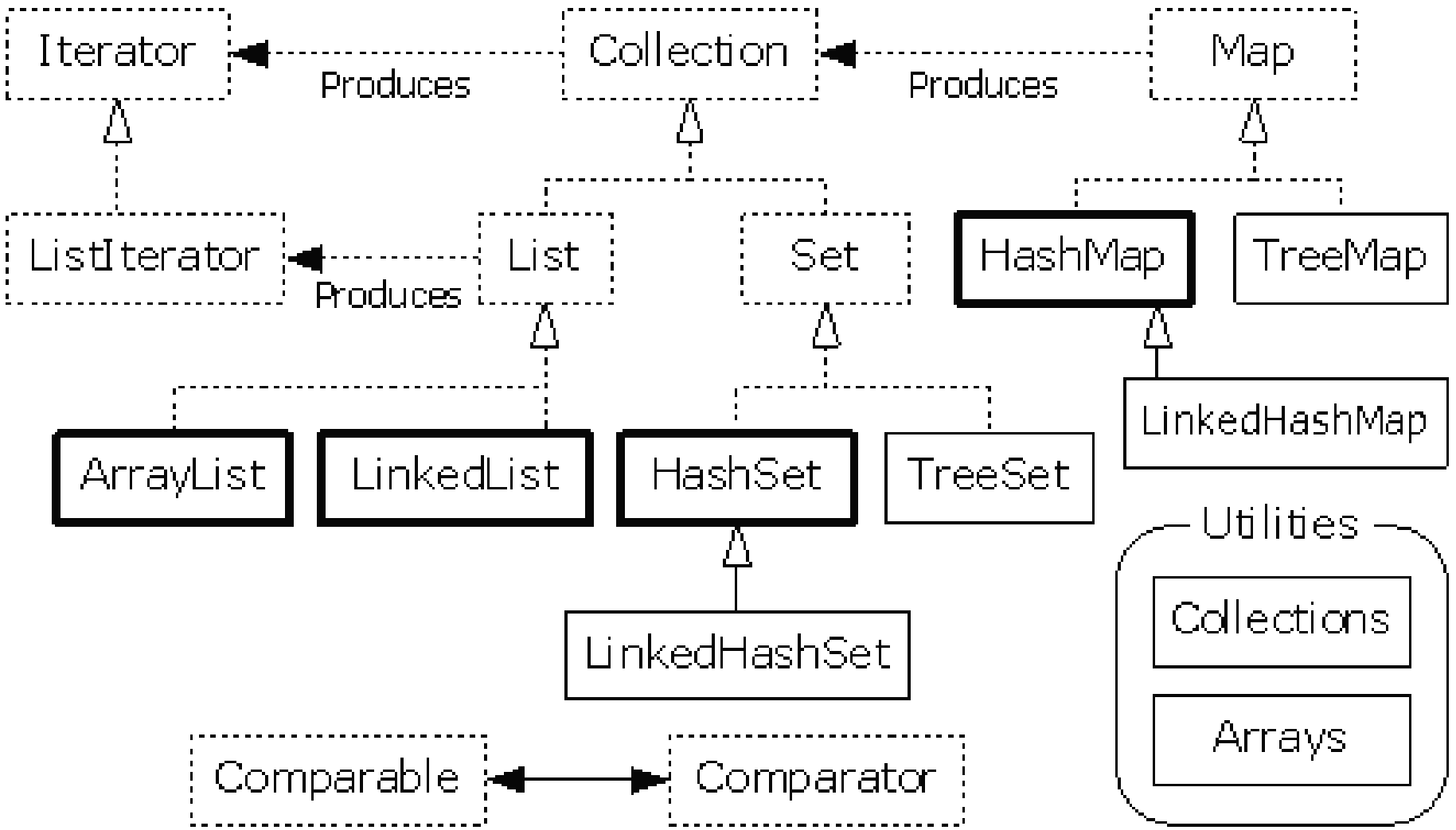
- You specify sort criteria just on SortList creation time.
- Different sort criteria can be defined: i.e:
  - Alphabetically
  - Salary
  - etc



Containers

- Java containers are divided into two groups
  - **Collection**: a group of individual elements, often with some rule applied to them:
    - ✓ A List must hold the elements in a particular sequence (ArrayList and LinkedList)
    - ✓ A Set cannot have any duplicate elements (HashSet, TreeSet and LinkedHashSet)
    - ✓ A queue with a special order, a “first-in, first-out” (FIFO) container
  - Map: a group of key-value object pairs.
    - ✓ HashMap
    - ✓ TreeMap
    - ✓ LinkedListMap
- They are in java.util and are formed by:
  - interfaces
  - classes.
  - iterators

Container taxonomy



Container taxonomy

- Interface Collection

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Collection functionality

Method	Meaning
boolean **add**(Object)	Ensures that the container holds the argument. Returns false if it doesn't add the argument. (This is an "optional" method, described later in this chapter.)
boolean **addAll**(Collection)	Adds all the elements in the argument. Returns trueif any elements were added. ("Optional.")
void **clear**( )	Removes all the elements in the container. ("Optional.")
boolean **contains**(Object)	true if the container holds the argument.
boolean containsAll(Collection)	true if the container holds all the elements in the argument.
boolean **isEmpty**( )	true if the container has no elements.
Iterator **iterator**( )	Returns an Iteratorthat you can use to move through the elements in the container.
boolean **remove**(Object)	If the argument is in the container, one instance of that element is removed. Returns trueif a removal occurred. ("Optional.")
boolean **removeAll**(Collection)	Removes all the elements that are contained in the argument. Returns trueif any removals occurred. ("Optional.")
boolean **retainAll**(Collection)	Retains only elements that are contained in the argument (an "intersection" from set theory). Returns trueif any changes occurred. ("Optional.")
int **size**( )	Returns the number of elements in the container.
Object[] **toArray**( )	Returns an array containing all the elements in the container.
Object[] **toArray**(Object[] a)	Returns an array containing all the elements in the container, whose type is that of the array a rather than plain Object(you must cast the array to the right type).

Container taxonomy

- Example: Simple Collection. Adding information

```
import java.util.*;

public class FillingLists {
   private static Test monitor = new Test();
   public static void main(String[] args) {
      List list = new ArrayList();
      for(int i = 0; i < 10; i++)
         list.add("");
      Collections.fill(list, "Hello");
      System.out.println(list);
   }
} ///:~
```

Moving through a container elements using Iterators

- Containers lets us traversing their elements sequentially by using an Iterator object
- Iterator related Operations:
  1- Ask a container to hand you an Iterator using a method called **iterator( )**. This Iterator will be ready to return the first element in the sequence on your first call to its **next( )** method.
  2- Get the next object in the sequence with **next( )**.
  3- See if there are any more objects in the sequence with **hasNext( )**.
  4- Remove the last element returned by the iterator with **remove( ).**

- **Notice:**
  - Containers start from **position 0**.
  - It is not necessary to know the size of the container.

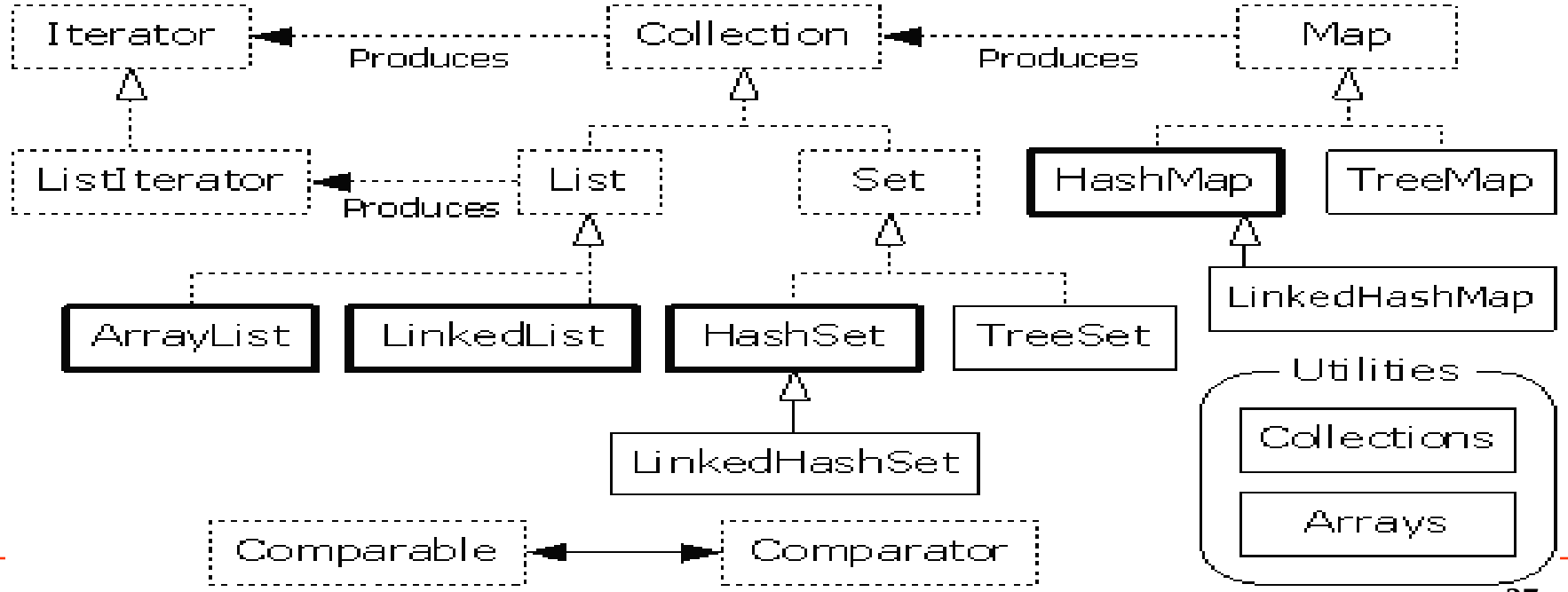
Iterators usage examples

```
import java.util.*;

public class SimpleCollection {
   private static Test monitor = new Test();
   public static void main(String[] args) {
      // Upcast because we just want to
      // work with Collection features
      Collection c = new ArrayList();
      for(int i = 0; i < 10; i++)
         c.add(Integer.toString(i));
      Iterator it = c.iterator();
      while(it.hasNext())
         System.out.println(it.next());
   }
} ///:~
```

- Exercises: Use a collection to store Persons
  - 1. Add persons to the collection
  - 2. Make a method to know how many odds age there are
  - 3. Make a method to look for a given name in the collection

Container taxonomy

- Containers are formed by interfaces, classes and iterators.
  - Interface Collection (to hold element)
    - ✓ Interface List (ArrayList and LinkedList classes)
    - ✓ Interface Set (HashSet and TreeSet classes)
  - Interface Map (key-value pair)
    - ✓ TreeMap and HashMap Classes
  - Interface Iterator (to iterate)



List functionality

- List allows us to add new object to a container, increasing its size automatically.
- List is quite simple to use
  - **add**( ) or **set (int pos)** to insert objects,
  - **get**(**int pos**) to get them out one at a time,
  - and **iterator**( ) to get an Iterator for the sequence,
- However, a set of other methods can be used
  - contains
  - remove
  - indexOf
  - equals
  - isEmpty
  - lastIndexOf
  - subList
  - size
  - .....
- Different types of List can be defined.

List functionality

List (interface)	Order is the most important feature of a **List**; it promises to maintain elements in a particular sequence. **List** adds a number of methods to **Collection** that allow insertion and removal of elements in the middle of a **List**. (This is recommended only for a **LinkedList**.) A **List** will produce a **ListIterator**, and by using this you can traverse the **List** in both directions, as well as insert and remove elements in the middle of the **List**.
ArrayList*	A **List** implemented with an array. Allows rapid random access to elements, but is slow when inserting and removing elements from the middle of a list. **ListIterator** should be used only for back-and-forth traversal of an **ArrayList**, but not for inserting and removing elements, which is expensive compared to **LinkedList**.
LinkedList	Provides optimal sequential access, with inexpensive insertions and deletions from the middle of the **List**. Relatively slow for random access. (Use **ArrayList** instead.) Also has **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, and **removeLast( )** (which are not defined in any interfaces or base classes) to allow it to be used as a stack, a queue, and a deque.

List functionality example

- Interface List

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);            // Optional
    void add(int index, Object element);              // Optional
    Object remove(int index);                         // Optional
    abstract boolean addAll(int index, Collection c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

List functionality example

- Two implementations
  - ArrayList
    - ✓ Same methods than List, however they can iterate by ListIterator.
  - LinkedList (Very useful for stack and queue)
    - ✓ getFirst
    - ✓ getLast
    - ✓ removeFirst
    - ✓ removeLast
    - ✓ addFirst
    - ✓ addLast

List functionality example

- ListIterator
  - It allows us to move forward and backward.

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();              // Optional
    void set(Object o);         // Optional
    void add(Object o);         // Optional
}
```

Coding for interfaces (not implementations)

It is a good programming practice to write codes to work with interfaces.

This way the only place you care about implementation is at the point of creation:

```
List x = new LinkedList();
```

In other parts of code, you work with List. This has the benefit that if you decide to change implementation (for example to ArrayList), other parts of code is not affected

List functionality

```
import java.util.*;
public class Main {

 public static List fill(List a) {
  Random rand=new Random();
  for(int i = 0; i < 10; i++)
    a.add(Integer.toString(rand.nextInt(100)));
  return a;
 }

public static int sum(List a) {
 int suma=0;
 Iterator it=a.iterator();
 while (it.hasNext())
     suma=suma+Integer.parseInt(it.next().toString());
 return suma;
 }

 public static void main(String[] args) {
  // Make and fill a new list each time:

  List a=new ArrayList();
  System.out.println(sum(fill(a)));
 }
} ///:~
```

List functionality example

```
import java.util.*;
import com.bruceeckel.util.*;

public class List1 {
  public static List fill(List a) {
       Random rand=new Random();
       for(int i = 0; i < 10; i++)
            a.add(Integer.toString(rand.nextInt(100)));
        return a;
  }
   private static boolean b;
   private static Object o;
   private static int i;
   private static Iterator it;
   private static ListIterator lit;

   public static void main(String[] args) {
     // Make and fill a new list each time:
     basicTest(fill(new LinkedList()));
     basicTest(fill(new ArrayList()));
     iterMotion(fill(new LinkedList()));
     iterMotion(fill(new ArrayList()));
     iterManipulation(fill(new LinkedList()));
     iterManipulation(fill(new ArrayList()));
     testVisual(fill(new LinkedList()));
     testLinkedList();
   }
} ///:~
```

List functionality example

```
public static void basicTest(List a) {
        a.add(1, 14); // Add at location 1
        a.add(12); // Add at end
        // Add a collection:
        a.addAll(fill(new ArrayList()));
        // Add a collection starting at location 3:
        a.addAll(3, fill(new ArrayList()));
        b = a.contains(1); // Is it in there?
        // Is the entire collection in there?
        b = a.containsAll(fill(new ArrayList()));
        // Lists allow random access, which is cheap
        // for ArrayList, expensive for LinkedList:
        o = a.get(1); // Get object at location 1
        i = a.indexOf(1); // Tell index of object
        b = a.isEmpty(); // Any elements inside?
        it = a.iterator(); // Ordinary Iterator
        lit = a.listIterator(); // ListIterator
        lit = a.listIterator(3); // Start at loc 3
        i = a.lastIndexOf(1); // Last match
        a.remove(1); // Remove location 1
        a.remove("3"); // Remove this object
        a.set(1, 12); // Set location 1 to "y"
        // Keep everything that's in the argument
        // (the intersection of the two sets):
        a.retainAll(fill(new ArrayList()));
        // Remove everything that's in the argument:
        a.removeAll(fill(new ArrayList()));
        i = a.size(); // How big is it?
        a.clear(); // Remove all elements
      }
```

List functionality example (cont.)

```
public static void iterMotion(List a) {
        ListIterator it = a.listIterator();
        b = it.hasNext();
        b = it.hasPrevious();
        o = it.next();
        i = it.nextIndex();
        o = it.previous();
        i = it.previousIndex();
      }
      public static void iterManipulation(List a) {
        ListIterator it = a.listIterator();
        it.add("47");
        // Must move to an element after add():
        it.next();
        // Remove the element that was just produced:
        it.remove();
        // Must move to an element after remove():
        it.next();
        // Change the element that was just produced:
        it.set("47");
      }
```

List functionality example (cont.)

```
// There are some things that only LinkedLists can do:
public static void testLinkedList() {
  LinkedList ll = new LinkedList();
  fill(ll);
  System.out.println(ll);
  // Treat it like a stack, pushing:
  ll.addFirst("one");
  ll.addFirst("two");
  System.out.println(ll);
  // Like "peeking" at the top of a stack:
  System.out.println(ll.getFirst());
  // Like popping a stack:
  System.out.println(ll.removeFirst());
  System.out.println(ll.removeFirst());
  // Treat it like a queue, pulling elements
  // off the tail end:
  System.out.println(ll.removeLast());
  // With the above operations, it's a dequeue!
  System.out.println(ll);
}
```

List exercises

- Given a list of Integer (as ArrayList and LinkedList)
  - Work out
    - ✓ the average of the List
    - ✓ the bigger number of the container
    - ✓ if the list is ordered
  - Make the same exercise using Set

- Given a List of Students formed by
  - Personal Information (name, identity card and age)
  - Course information (list of courses)
  - Where each course has a list of subject and mark

  - Fill the structure with information

Making a stack from a linkedList

- A **stack** is sometimes referred to as a "last-in, first-out" (LIFO) container. That is, whatever you "push" on the stack last is the first item you can "pop" out.
- The **LinkedList** has methods that directly implement stack functionality, so you can also just use a LinkedList rather than making a stack class.

```
public class StackL {
  private LinkedList list = new LinkedList();

  public Object top() { return list.getFirst(); }

  public void push(Object v) { list.addFirst(v); }
  public Object pop() { return list.removeFirst(); }

  public static void main(String[] args) {
    StackL stack = new StackL();
    Random rand=new Random();
    for(int i = 0; i < 10; i++)
      stack.push(rand.nextInt());
    System.out.println(stack.top());
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
  }
} ///:~
```

Making a queue from a linkedList

- A **queue** is sometimes referred to as a “first-in, first-out” (FIFO) container. That is, whatever you “push” on the stack first is the first item you can “pop” out.
- The **LinkedList** has methods that directly implement queue funcionality, so you can also just use a LinkedList rather than making a queue class.
- Queue container is quite important in **concurrency** and for this reason is included in Java Api.

```
import java.util.*;

public class Queue {
   private LinkedList list = new LinkedList();

   public void put(Object v) { list.addFirst(v); }
   public Object get() { return list.removeLast(); }
   public boolean isEmpty() { return list.isEmpty(); }

   public static void main(String[] args) {
      Queue queue = new Queue();
      for(int i = 0; i < 10; i++)
         queue.put(Integer.toString(i));
      while(!queue.isEmpty())
         System.out.println(queue.get());
   }
```

Using a LinkedList to implement the queue interface

- **Interface Queue**. Methods
  - offer(): Adds an element at the end of the queue
  - peek() and element(): Return the first element.
    - ✓ peek(): Return null if the queue is empty
    - ✓ element (): Throw NoSuchElementException if it is empty
  - pool() and remove(): Delete and Return the 1st el.
    - ✓ pool()--> Null
    - ✓ remove() -->NoSuchElementException

```
import java.util.*;
public class QueueDemo {
  public static void printQ(Queue queue) {
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> queue = new LinkedList<Integer>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      queue.offer(rand.nextInt(i + 10));
    printQ(queue);
    Queue<Character> qc = new LinkedList<Character>(
    for(char c : "Brontosaurus".toCharArray())
      qc.offer(c);
    printQ(qc);
```

PriorityQueue. Java 5

- What is a PriorityQueue???

```
import java.util.*;
public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<Integer> priorityQueue = new PriorityQueue<Integer>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      priorityQueue.offer(rand.nextInt(i + 10));
    QueueDemo.printQ(priorityQueue);
    List<Integer> ints = Arrays.asList(25, 22, 20, 18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
    priorityQueue = new PriorityQueue<Integer>(ints);
    QueueDemo.printQ(priorityQueue);

    priorityQueue = new PriorityQueue<Integer>(ints.size(), Collections.reverseOrder());
    priorityQueue.addAll(ints);
    QueueDemo.printQ(priorityQueue);

  }
} /* Output:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
*///:~
```

Set Functionality

- **Set** has exactly the same interface as Collection
- A **Set** refuses to hold more than one instance of each object value

Set (interface)	Each element that you add to the **Set** must be unique; otherwise, the **Set** doesn't add the duplicate element. **Object**s added to a **Set** must define **equals( )** to establish object uniqueness. **Set** has exactly the same interface as **Collection**. The **Set** interface does not guarantee that it will maintain its elements in any particular order.
HashSet*	For **Set**s where fast lookup time is important. **Object**s must also define **hashCode( )**.
TreeSet	An ordered **Set** backed by a tree. This way, you can extract an ordered sequence from a **Set**.
LinkedHashSet (JDK 1.4)	Has the lookup speed of a **HashSet**, but maintains the order in which you add the elements (the insertion order), internally using a linked list. Thus, when you iterate through the **Set**, the results appear in insertion order.

Hashing

- Hashing is a common way to build lookup tables
- For each key a hash value is calculated
- Objects are inserted in locations that are specified by their hash value
- For lookup, again hash code is used to find the place the object is in
- The root class Object in java defines a hashCode method which simply returns address of the object as hashCode
- It is in-appropriate in many situations (e.g. two keys may have same value but have different addresses!)
- Subclasses like String overdies hashCode to have appropriate behavior
- For user-defined classes therefore we need to implement hasCode
- HashMap also uses equals() method. So for user defined classes we need to overide equals() too.

Set Functionality example

- Notice that the order maintained by the **HashSet** is different from **TreeSet** and **LinkedHashSet**

```
public class Set1 {
  private static Test monitor = new Test();
  static void fill(Set s) {
    s.addAll(Arrays.asList(
      "one two three four five six seven".split(" ")));
  }
  public static void test(Set s) {
    // Strip qualifiers from class name:
    System.out.println(
      s.getClass().getName().replaceAll("\\w+\\.", ""));
    fill(s); fill(s); fill(s);
    System.out.println(s); // No duplicates!
    // Add another set to this one:
    s.addAll(s);
    s.add("one");
    s.add("one");
    s.add("one");
    System.out.println(s);
    // Look something up:
    System.out.println("s.contains(\"one\"): " +
      s.contains("one"));
  }
  public static void main(String[] args) {
    test(new HashSet());
    test(new TreeSet());
    test(new LinkedHashSet());
    monitor.expect(new String[] {
      "HashSet",
      "[one, two, five, four, three, seven, six]",
      "[one, two, five, four, three, seven, six]",
      "s.contains(\"one\"): true",
      "TreeSet",
      "[five, four, one, seven, six, three, two]",
      "[five, four, one, seven, six, three, two]",
      "s.contains(\"one\"): true",
      "LinkedHashSet",
      "[one, two, three, four, five, six, seven]",
      "[one, two, three, four, five, six, seven]",
      "s.contains(\"one\"): true"
    });
  }
} ///:~
```

Set Functionality example

- Notice that the order maintained by the HashSet is different from TreeSet and LinkedHashSet
  - **TreeSet** keeps elements sorted into a red-black tree data structure
  - **HashSet** uses a hashing function, which is designed specifically for rapid lookups.
  - **LinkedHashSet** uses hashing internally for lookup speed, but appears to maintain elements in insertion order using a linked list.

  - When creating your own types, be aware that a Set needs a way to maintain a storage order, which means that you must implement the Comparable interface and define the compareTo( ) method.

Sorted set

- If you have a **SortedSet** (of which TreeSet is the only one available), the elements are guaranteed to be in sorted order, which allows additional functionality to be provided with these methods in the SortedSet interface
  - Comparator comparator( ): Produces the Comparator used for this Set, or null for natural ordering.
  - Object first( ): Produces the lowest element.
  - Object last( ): Produces the highest element.
  - SortedSet subSet(fromElement, toElement): Produces a view of this Set with elements from fromElement, inclusive, to toElement, exclusive.
  - SortedSet headSet(toElement): Produces a view of this Set with elements less than toElement.
  - SortedSet tailSet(fromElement): Produces a view of this Set with elements greater than or equal to fromElement..

Sorted set

```
public class SortedSetDemo {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    SortedSet sortedSet = new TreeSet(Arrays.asList(
    "one two three four five six seven eight".split(" ")));
    System.out.println(sortedSet);
    Object
      low = sortedSet.first(),
      high = sortedSet.last();
    System.out.println(low);
    System.out.println(high);
    Iterator it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedSet.subSet(low, high));
    System.out.println(sortedSet.headSet(high));
    System.out.println(sortedSet.tailSet(low));
    monitor.expect(new String[] {
      "[eight, five, four, one, seven, six, three, two]",
      "eight",
      "two",
      "one",
      "two",
      "[one, seven, six, three]",
      "[eight, five, four, one, seven, six, three]",
      "[one, seven, six, three, two]"
    });
  }
} ///:~
```

Templates or Generic in Java

- Container disadvantages: unknown types
- Because the type information is thrown away when you put an object reference into a container, there's no **restriction on the type** of object that can be put into your container.
- Because the type information is lost, the only thing the container knows that it *holds is a reference* to an object.
- For this reason, **you must perform a cast** to the correct type before you use it.

- This problem is fixed in Java5 using **GENERIC**

Inability to control type of objects we put in a container

```
//: c11:Cat.java
package c11;

public class Cat {
  private int catNumber;
  public Cat(int i) { catNumber = i; }
  public void id() {
    System.out.println("Cat #" + catNumber);
  }
} ///:~
```

```
//: c11:Dog.java
package c11;

public class Dog {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id() {
    System.out.println("Dog #" + dogNumber);
  }
} ///:~
```

```
package c11;
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    List cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).id();
      // Dog is detected only at run time
  }
} ///:~
```

Making a type conscious ArrayList

- To avoid this situation (typed problem), to store a list of Mouse, we can add a List in MouseList as attribute and implements **add, get and size** methods.
- Using generic, this problem is solved (java 1.5).

```
//: c11:MouseList.java
// A type-conscious List.
import java.util.*;

public class MouseList {
  private List list = new ArrayList();
  public void add(Mouse m) { list.add(m); }
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size() { return list.size(); }
} ///:~
```

```
import com.bruceeckel.simpletest.*;

public class MouseListTest {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    MouseList mice = new MouseList();
    //ArrayList<Mouse> Mice = newArrayList<Mouse>();//Generic
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++)
      MouseTrap.caughtYa(mice.get(i));
  }
} ///:~
```

Map functionality

- To store a sequence of objects using other criterion non-based on its position.
- It is called as a **map**, a **dictionary**, or an **associative array**.
- Maintains key-value associations (pairs) so you can look up a value using a key.
- **Methods**
  - Object **put**(Object key, Object value);
  - Object **remove**(Object key);
  - Object **get**(Object key);
  - **containsKey**, **containsValue**, **isEmpty**,**size**

- A map object can only store a value for each key. If we want to store more than one value, it has to define as a list of value

Map functionality

Map (interface)	Maintains key-value associations (pairs) so you can look up a value using a key.
HashMap*	Implementation based on a hash table. (Use this instead of **Hashtable**.) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table.
LinkedHashMap (JDK 1.4)	Like a **HashMap**, but when you iterate through it, you get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than a **HashMap**, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.
TreeMap	Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order (determined by **Comparable** or **Comparator**, discussed later). The point of a **TreeMap** is that you get the results in sorted order. **TreeMap** is the only **Map** with the **subMap( )** method, which allows you to return a portion of the tree.
WeakHashMap	A map of *weak keys* that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.
IdentityHashMap (JDK 1.4)	A hash map that uses == instead of **equals( )** to compare keys. Only for solving special types of problems; not for general use.

Map usage example

- Maps do not provide an iterator() method as do Lists and Sets.
- A Set of either keys (keySet()) or key-value Map.Entry elements (entrySet()) can be obtained from the Map, and one can iterate over that.

```
HashMap hashMap = new HashMap();
hashMap.put(“51”, new Alumno(“Luis”,”51”,...));
hashMap.put("102",new Alumno(“Juan”,”102”,....));
hashMap.put("404",new Alumno(“Ana”,”404”,.....));
Iterator it = hashMap.entrySet().iterator();
while (it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();
        System.out.println(e.getKey() + " " + e.getValue().toString());
}
```

Map usage example

```
import java.util.*;

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

//       HashMap global = new TreeMap();
         //Sort by String

//NO sort
Map<String,String> global = new
                    HashMap<String,String>();

        // Insertar valores "key"-"value" al HashMap
        global.put("Doctor", "(+52)-4000-5000");
        global.put("Casa", "(888)-4500-3400");
        global.put("Hermano", "(575)-2042-3233");
        global.put("Hermana", "(421)-1010-0020");
        global.put("Suegros", "(334)-6105-4334");
        global.put("Oficina", "(304)-5205-8454");
        global.put("Ana C.", "(756)-1205-3454");
        global.put("Luis G.", "(55)-9555-3270");
        global.put("Oficina 2", "(874)-2400-8600");

// Iterator to retrieve/print its values
for( Iterator it = global.keySet().iterator(); it.hasNext();) {
            String s = (String)it.next();
            String s1 = (String)global.get(s);
            System.out.println(s + " : " + s1);
}

// Define an array with some values
 String args1[] = {
          "Casa", "Ana C.", "Luis G."
 };

// Remove HaspMap item which are in the array HashMap con
en el array
for(int i = 0; i < args1.length; i++) {
          global.remove(args1[i]);
}


// Iterator to retrieve/print its values
for( Iterator it2 = global.keySet().iterator(); it2.hasNext();) {
            String s = (String)it2.next();
            String s1 = (String)global.get(s);
            System.out.println(s + " : " + s1);    }

    }
}
```

Choosing a Container

- We have seen a number of containers. We should know in what situations to use each of them
- We now when we need a List or a Set or a Map
- List is used when the order of elements are important
- Set is used when we don't care about order but we want to be sure there is no repeatition
- Map is used for making associations between keys and values (lookup tables)

Choosing between List implementations

Type	Get	Iteration	Insert	Remove
array	172	516	na	na
ArrayList	281	1375	328	30484
LinkedList	5828	1047	109	16
Vector	422	1890	360	30781

Choosing between Set implementations

Type	Test size	Add	Contains	Iteration
	10	25.0	23.4	39.1
TreeSet	100	17.2	27.5	45.9
	1000	26.0	30.2	9.0
	10	18.7	17.2	64.1
HashSet	100	17.2	19.1	65.2
	1000	8.8	16.6	12.8
	10	20.3	18.7	64.1
LinkedHashSet	100	18.6	19.5	49.2
	1000	10.0	16.3	10.0

Choosing between Maps

Type	Test size	Put	Get	Iteration
	10	26.6	20.3	43.7
TreeMap	100	34.1	27.2	45.8
	1000	27.8	29.3	8.8
	10	21.9	18.8	60.9
HashMap	100	21.9	18.6	63.3
	1000	11.5	18.8	12.3
	10	23.4	18.8	59.4
LinkedHashMap	100	24.2	19.5	47.8
	1000	12.3	19.0	9.2
	10	20.3	25.0	71.9
IdentityHashMap	100	19.7	25.9	56.7
	1000	13.1	24.3	10.9
	10	26.6	18.8	76.5
WeakHashMap	100	26.1	21.6	64.4
	1000	14.7	19.2	12.4
	10	18.8	18.7	65.7
Hashtable	100	19.4	20.9	55.3
	1000	13.1	19.9	10.8

Bibliography

- Piensa en Java. 4ª Edición. Pearson Prentice Hall. ISBN 13: 9788489660342
- Core Java 2 Vol I. Fundamentos. Pearson Prentice Hall/Sun. ISBN 13: 9788420548326
- Core Java 2. Vol II. Características Avanzadas. Pearson Prentice Hall/Sun. ISBN 13: 9788483223109
- Programación, Algoritmos y ejecicios resultos en Java. Pearson Prentice Hall. ISBN 13: 9788420540245
- Estructuras de datos con Java. Diseño de estructuras y algoritmos. Pearson Addison Wesley. ISBN 13: 9788420550343