COMP241
Software Engineering Development
Lecture 9: Container Classes 1 (Collections)

Mark Hall

Readings: Horstman Chap 4 & 5

THE UNIVERSITY OF WAIKATO DEPARTMENT OF COMPUTER SCIENCE TARI ROROHIKO

- Introduction
- arrays
- Arrays class
- Copying arrays
- Comparing arrays
  - `Comparable`
  - `Comparator`
    - Strategy design pattern
- Sorting arrays
- Searching sorted arrays

Introduction

- Only the simplest of programs know apriori how many (or even what type of) objects will be needed
- Java has several ways to hold objects (actually references to objects)
  - Java's built in type is of course is the array
  - Also provides a reasonably complete set of *container classes*
    - Sophisticated ways to hold and manipulate objects
    - Not to be confused with GUI containers (components)
    - Sometimes called *collection classes*, however, Java 2 libraries use the name **Collection** to refer to a particular subset of the library

But first, Arrays

- Pretty familiar with arrays by now :-)
- Two features distinguish arrays from other container types
  - Efficiency — arrays are the most efficient way to store and randomly access a sequence of objects
    - The cost of speed is static size
  - Type — arrays are created to hold a specific type
    - You get compile-time type checking when storing/extracting
    - Other java containers are *generic,* i.e. they treat all types as **`Object`**
      - From Java 1.5 we have *generics*

Arrays

- Arrays can hold primitive types or objects
  - Regardless of the type of the array, the array identifier is actually a reference to a true object that is created on the heap

```
public class Test {} // does nothing

// arrays of objects
Test [] a; // null reference
Test [] b = new Test[5]; // null references
Test [] c = new Test[4];
for (int i = 0; i < c.length; i++) {
   c[i] = new Test();
}
// aggregate initialization (created and initialized)
Test [] d = {
   new Test(), new Test(), new Test()
};
```


Arrays

```
// dynamic aggregate initialization
a = new Test [] {
   new Test(), new Test()
};

// arrays of primitives
int [] e; // null reference
int [] f = new int [5]; // elements automatically = 0
int [] g = new int [4]; // elements automatically = 0
for (int i = 0; i < g.length; i++) {
   g[i] = i * i;
}
int [] h = { 11, 47, 93 };
e = new int [] { 1, 2 };
```


The Arrays class

- `java.util` has a utility class called `Arrays`
  - Holds a set of *static* methods that perform useful functions for arrays
- Four basic functions:
  - `equals` — compares two arrays for equality
  - `fill` — fill an array with a value
  - `sort` — sort an array
  - `binarySearch` — find an element in a **sorted** array
- All of these methods are overloaded for all the primitive types and `Objects`
- Also has a `asList` method that takes an array and turns it into a `List` container

The Arrays class

- `Arrays` is useful, but stops short of being fully functional
  - Eg. doesn't have a method for printing the elements of an array
  - `fill` method only takes a single value and places it into the array
    - What if we wanted to fill an array with randomly generated values?

Improving Arrays Example

- We could make a new, more useful version of `fill` and put it in our own utility class…
  - Fill an array with values or objects that are created by a user defined *generator*
  - Need a generator for each primitive type as well as Object

```
public interface Generator {
  Object next();
}

public interface IntGenerator {
   int next();
}

public interface CharGenerator {
   int next();
}
. . . // one for each primitive type
```

- Now make our own utility class (`Arrays2`) with new `fill` methods that use generators

```
public class Arrays2 {
  public static void fill(Object [] a, Generator g) {
    fill(a, 0, a.length, g);
  }
  public static void fill(Object [] a, int from, int to,
                          Generator g) {
     for (int i = from; i < to; i++) {
        a[i] = g.next();
     }
  }
  public static void fill(int [] a, IntGenerator g) {
     fill(a, 0, a.length, g);
  }
  public static void fill(int [] a, int from, int to,
                          IntGenerator g) {
     for (int i = from; i < to; i++) {
        a[i] = g.next();
     }
  }
  . . . // fill methods for every primitive type
```

Improving Arrays Example

- To fill an array of elements, the fill method takes a reference to the appropriate generator **interface**
  - Repeatedly calls the `next` method that somehow (depending on the implementation) produces an object of the right type
- Now we can create any generator by implementing the appropriate interface
  - e.g. `RandomIntGenerator implements IntGenerator`

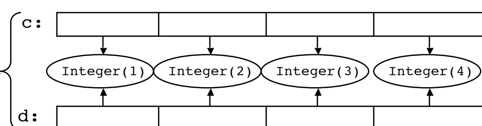
Copying an array

- The `System` class provides an overloaded `arraycopy` method for making fast copies of arrays
  - Can copy both primitive arrays and object arrays
  - If you copy arrays of objects then only the *references* get copied, not the objects themselves — this is called a *shallow copy*

Copying an Array

```
int [] a = {1,2,3,4,5,6,7,8,9,10};
int [] b = new int [5];
System.arraycopy(a, 5, b, 0, 5);
System.arraycopy(b, 0, a, 0, 5);

Integer [] c = {new Integer(1), new Integer(2),
                new Integer(3), new Integer(4)};
Integer [] d = new Integer [4]; // null references
System.arraycopy(c, 0, d, 0, c.length);
for (int i = 0; i < c.length; i++) {
   if (c[i] == d[i]) { System.out.println("Same object!"); }
}
```



Comparing Arrays

- `Arrays` provides the method `equals` to compare entire arrays for equality
  - Again these are overloaded for all the primitives and `Object`
- To be equal:
  - The arrays must have the same number of elements
  - Each element must be equivalent to each corresponding element in the other array, using the `equals` method for each element
    - For primitives, the corresponding wrapper class equals is used (e.g. Integer.equals())

Array Element Comparisons

- Sorting was one thing that was missing in early versions of Java (1.0 and 1.1)
  - Since Java 1.2, sorting has been included (eg. Arrays has a `sort` method)
- To have generic sorting code you need to be able to perform comparisons on the actual type of an object
  - One approach is to write a different sorting method for every type—but this does not produce code that is reusable for new types

Array Element Comparisons

- Instead of hardwiring the comparison code into many different sort routines, a *callback* can be used

  With a callback, the part of the code that varies from case to case is encapsulated inside its own class; the part of the code that stays the same will ***call back*** *to the code that changes*

  A callback is an example of using the ***Strategy*** *design pattern*
- This way you can make different objects to express different ways of comparing and feed them to the same sorting code
- The `Generator` example shown earlier is an example of using the *Strategy* design pattern

Array Element Comparisons

- In Java there are two ways to provide comparison functionality
- One approach is for a class to implement `java.lang.Comparable`
  - Specifies a single `compareTo(Object cmp)` method
    - Must return -1, 0 or 1 to indicate that this object is less than, equal or greater than cmp
    - Must ensure transitivity (x.compareTo(y) > 0 && y.compareTo(z) >0 => x.compareTo(z) > 0)
  - Some classes that implement Comparable: Integer, Float, Double, etc. String

Comparable Example

```
public class CompType implements Comparable<CompType> {
  public int mI;
  public int mJ;
  public CompType(int n1, int n2) {
    mI = n1; mJ = n2;
  }
  public String toString() {
    return "[i = "+mI+", j = "+mJ+"]";
  }
  public int compareTo(CompType rv) {
    int rvi = rv.mI;
    return (mI < rvi ? -1 : (mI == rvi ? 0 : 1));
  }
}
```

Performs its comparisons using only the mI member variable

Comparable Example

- Now if we had an array of `CompType` objects we could sort them using `Arrays.sort()`

```
CompType [] a = {new CompType(3,3), new CompType(1,2),
                 new CompType(0,5), new CompType(7,6)};
Arrays.sort(a);
```

- When you design the comparison function, you are responsible for deciding what it means to compare one of your objects to another
  - In this example only the `mI` values are used in the comparison

Array Element Comparisons

- What if a class doesn't implement `Comparable`? Or, if it does, you don't like how the `compareTo` method works?
- In this case we can use the second approach for comparing objects
  - Create a separate class that implements an interface called `Comparator`
  - Implementing `Comparator` is an example of using the *callback* or *Strategy* approach

Comparator

- Defines two methods — `compare` and `equals`
  - `compare` is for comparing two objects
  - `equals` is for testing if one `Comparator` is equal to another
    - Don't actually have to implement this as the equals inherited from `Object` is sufficient
- The `java.util.Collections` class has a static `reverseOrder` method that returns a `Comparator` for reversing the natural sorting order of objects that implement `Comparable`

Example: Reverse (sorting an array of strings in reverse lexicographic order)

```
public class Reverse {
  public static void main(String [] args) {
    String [] a = new String [] {"Fred", "george", "Bob",
                                 "Zaphod", "Mary", "sue"};
    System.out.println("Before sorting: ");
    for (int i = 0; i < a.length; i++) {
       System.out.println(a[i]);
    }
    // sort contents of a into reverse lexicographic order
    Arrays.sort(a, Collections.reverseOrder());
    System.out.println("After sorting: ");
    for (int i = 0; i < a.length; i++) {
       System.out.println(a[i]);
    }
  }
}
```


Example: comparing CompType Objects according to their mJ value

```
public class CompTypeComparator implements Comparator<CompType> {
  public int compare(CompType o1, CompType o2) {
    int j1 = o1.mJ;
    int j2 = o2.mJ;
    // compare according to j values rather than i
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String [] args) {
    CompType [] a = {new CompType(3,3), new CompType(1,2),
                     new CompType(0,5), new CompType(7,6)};
    System.out.println("before sorting: ");
    for (int i = 0; i < a.length; i++) {
      System.out.println(a[i]);
    }
    Arrays.sort(a, new CompTypeComparator());
    System.out.println("after sorting: ");
    . . . // print out array again
}
```


Searching a sorted array

- Once an array is sorted, you can perform a fast search for an item using `Arrays.binarySearch()`
  - Returns a value >= 0 if the search item is found
  - If not found, returns a *negative* value representing the place that the element should be inserted [-(insertion point) - 1]
  - Must not use `binarySearch` on an **unsorted** array as the results will be unpredictable
- If you have sorted an array using a `Comparator`, you must pass in the `Comparator` to `binarySearch`
  - Note that primitive arrays do not allow sorting with a `Comparator`

Example: AlphabeticSearch (search alphabetically sorted strings)

- First we need a `Comparator` to ensure that Strings are sorted in *alphabetical* rather than *lexicographic* order

```
public class AlphabeticComparator implements
Comparator<String> {
  public int compare(String s1, String s2) {
    // make case irrelevant
    return s1.toLowerCase().compareTo(s2.toLowerCase());
  }
}
```

Example: AlphabeticSearch (search alphabetically sorted strings)

```
public class AlphabeticSearch {
  public static void main(String [] args) {
    String [] a = new String [] {"Fred", "george", "Bob",
                                  "Zaphod", "Mary", "sue"};
    String searchItem = new String(a[1]);
    AlphabeticComparator comp = new AlphabeticComparator();
    Arrays.sort(a, comp); //sort with respect to the Comparator

    // search for an element that is already in the array
    int index = Arrays.binarySearch(a, searchItem, comp);
    System.out.println("Index = " + index);

    // search for something that is not present
    index = Arrays.binarySearch(a, "Craig", comp);
    System.out.println("Index = " + index);
  }
}
```


Beyond Arrays

- Arrays should always be considered first for holding a group of items because of their efficiency
  - Forced to use arrays if you want to hold a group of primitives
- However, sometimes arrays are not sophisticated enough for a particular problem
- Java provides a number of powerful container classes with sufficient functionality to cover most problems