Array-Lists, Iterators, Comparators, and Collections

Weiss, section 2.4, sections 4.1.9 through 4.1.11, and from the beginning of chapter 6 through section 6.4 (pages 37–47, 121–124, and 229–248)
Arrays

• Have a *length* field
• Use reference rules for array assignment and parameter passing
• Use different syntax for declaration and construction

• Length of an array is known when creating the array.
• Each element is accessed by its numerical index.
Arrays

public class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray; // declares an array. anArray is null.
        anArray = new int[2]; // allocates memory
        anArray[0] = 100;
        anArray[1] = 200;
        System.out.println("Element 1 at index 0: " + anArray[0]);
        System.out.println("Element 1 at index 0: " + anArray[1]);
        int[] otherArray = {100, 200};

        Dog[] dogs1 = new Dog[5];
        for (int i = 0; i < dogs1.length; i++) { dogs1[i] = new Dog(); }
        Dog[] dogs2 = new Dog[5];
        dogs2 = dogs1;
    }
}

Dog[] dogs1 = new Dog[5];
for (int i = 0; i < dogs1.length; i++) { dogs1[i] = new Dog(); }
Dog[] dogs2 = new Dog[5];
dogs2 = dogs1;
Compatibility of Array Types

- Array of type T [ ] may contain objects of type T or any subtype of T
  - Arrays are covariant.

```java
Number[ ] numbers = new Numbers[2];
numbers[0] = new Integer(10);
numbers[1] = new Double(3.14);

Integer[ ] myInt = {1, 2, 3};
Number[ ] myNumber = myInt;

myNumber[0] = 3.14; // throws ArrayStoreException at runtime
// myNumber[0] references an Integer object!
```
Array Expansion

• An array is resized because you may need to put more data into array.

• Compiler needs to know the new size of the array to allocate enough memory.
Array Expansion

```java
int [ ] arr = new int[ 10 ];

int [ ] original = arr;
arr = new int[ 12 ];
for( int i = 0; i < 10; i++ )
    arr[ i ] = original[ i ];
```

- Copying array elements is an expensive operations.
- Expand array to make it some multiplicative constant times as large.
- Cost of copying is paid off by adding elements without expansion.
ArrayList

ArrayList tracks its capacity. If capacity is reached, it automatically resizes an array.

ArrayList is a generic type. It implements List interface.

Bear momma = new Bear();
Bear poppa = new Bear();

List<Bear> bears = new ArrayList<Bear>();
bears.add(momma);
bears.add(poppa);

add method does not copy momma Bear object to ArrayList. It adds a reference for that object to ArrayList.

Bear firstBear = bears.get(0);
Bear secondBear = bears.get(1);
Multidimensional Arrays

• It is commonly used as a matrix.
• Components are themselves arrays.
  
  $$\text{int}[\ ][\ ] \ x = \text{new} \ \text{int}[3][4];$$
• Rows are allowed to vary in length.
  
  $$\text{int}[\ ][\ ] \ a = \{ \{1, 2, 3\}, \{4\}, \{5, 6, 7, 8\}\}$$
• An element is accessed by more than one index.
  
  ```java
  System.out.println(a[0][2]); // 3
  a.length returns the number of rows.
  a[0].length returns the number of columns (3).
  ```
• *Collection* is an object that groups multiple elements into a single unit. Classes that implement Collection are data structures that
  ▪ may allow duplicate elements.
  ▪ may have ordered elements.
• Collection provides methods for:
  ▪ Counting the number of elements in the collection.
  ▪ Checking whether an element is in the collection.
  ▪ Adding and removing elements.
  ▪ Iterating (traversing) through elements.
• Benefits of Collection
  ▪ Reduces programming effort.
  ▪ Fosters software reuse.
Iterator

- Traversal is the most common operation over a List.
- Walk a list and do something to each element, e.g., printing the value of that element.
- Iteration is a form of traversal. An **iterator** object is used for traversing.
- Iterator is like a *cursor*. It maintains the current position in the list.

```
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("world");
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

*hasNext* returns **true** if cursor (index) is less than the size of list.

*next* retrieves the element under the cursor and advances the cursor one position.
For-each Loops

• Traversing with an iterator is very common in Java.
• Java provides a special syntax for the iteration.

List<String> list = new ArrayList<String>();
list.add(“Hello”);
list.add(“world”);
for (String str : list) {
    System.out.println(str);
}

str is a variable that each element of the list is bound to.
list is the reference to a list that is traversed.
Comparable

• Comparable\(<T>\) is an interface in the Java Standard Library.

```java
public interface Comparable\(<T>\) {
    public int compareTo(\(T o\));
}
```

• Objects of *type parameter* implement `compareTo` to define comparison.

• The return value is
  - A negative integer if this object is less than the other object.
  - 0 if objects are equal.
  - A positive integer if this object is greater than the other object.

• `compareTo` imposes *ordering* among objects of the *same type*. 
public class Dog extends Animal implements Comparable<Dog> {

    private String name;

    public Dog(int age) {
        super(age);
    }

    @Override
    public void makeSound() {
        System.out.println("the Dog barks");
    }

    @Override
    public int compareTo(Dog other) {
        if (this.getAge() < other.getAge()) {
            return -1;
        } else if (this.getAge() > other.getAge()) {
            return 1;
        } else {
            return 0;
        }
    }

    @Override
    public String toString() {
        return getAge() + "";
    }
}
Comparable

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestComparable {

    public static void main(String[] args) {
        List<Dog> dogList = new ArrayList<Dog>(){
            dogList.add(new Dog(5));
            dogList.add(new Dog(3));
        }
        Collections.sort(dogList);
        System.out.println(dogList);
    }
}
```

`sort` orders Dog objects based on the natural order. Output: [3, 5]
Comparator

- Order objects based on **different properties**.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- An object of Comparator encapsulates an ordering.
- Returns a negative integer, 0, or positive integer depending on whether o1 is than, equal to, or greater than o2.
public class ageComparator implements Comparator<Dog> {
    @Override
    public int compare(Dog o1, Dog o2) {
        if (o1.getAge() < o2.getAge()) {
            return -1;
        } else if (o2.getAge() > o2.getAge()) {
            return 1;
        } else {
            return 0;
        }
    }
}

public class nameComparator implements Comparator<Dog> {

    @Override
    public int compare(Dog o1, Dog o2) {
        int res = o1.getName().compareTo(o2.getName());
        return res;
    }
}
Comparator

```java
public class TestComparable {

    public static void main(String[] args) {
        List<Dog> dogList = new ArrayList<Dog>();
        dogList.add(new Dog(5, "a"));
        dogList.add(new Dog(3, "b"));
        Collections.sort(dogList, new nameComparator());
        System.out.println(dogList);
    }
}
```

Sort orders Dog objects based on names
Output: [5:a, 3:b]