Arrays of Objects, Array Lists, Wrapper Classes

2013-03-10
Arrays of Objects

• The elements of an array can be object references
• The following declaration reserves space to store 5 references to String objects

\[
\text{String[]} \ \text{words} = \text{new String}[5];
\]
• It does NOT create the String objects themselves
• Initially an array of objects holds null references
• Each object stored in an element of an array must be instantiated separately
Arrays of Objects

• The **words** array when initially declared:

  ![Diagram](image.png)

  - A reference to `words.length` is OK (= 5)
  - However, the following reference will throw a `NullPointerException`:
    ```java
    System.out.println(words[0].length());
    ```
Arrays of Objects

• To create some `String` objects and store them in elements of the array:

```java
words[0] = new String("friendship");
words[1] = "loyalty";
words[2] = "honor";
```
Arrays of Objects

• String objects can be created using literals

• The following declaration creates an array object called verbs with a length of 4 and fills it with references to four String objects created using string literals

```java
String[] verbs = {"play", "work", "eat", "sleep"};
```
Arrays of Objects

• To use one of the methods of an object element of an array:
  ```java
  verbs[2].equals("eat");  // true
  ```

• To pass one of the object elements of an array as a parameter to a method:
  ```java
  "eat".equals(verbs[2]);  // true
  ```

• To return an element of an array:
  ```java
  public String methodName(String[] verbs)
  {
    return verbs[2];  // "eat"
  }
  ```
Command-Line Arguments

• Your program’s main method is defined as:
  
  ```java
  public static void main(String[] args)
  ```

• The signature of the `main` method indicates that it takes an array of `String` objects as a parameter

• These values come from *command-line arguments* that are provided when the interpreter is invoked

• In Dr Java interactions pane, this invocation of the JVM passes three `String` objects (or tokens) as arguments to the `main` method of `StateEval`:
  
  ```java
  > java StateEval pennsylvania texas arizona
  ```
Command Line Arguments

- These strings are stored at indexes 0–2 in the array args for the main method
- The array args will contain:

```
for (String arg : args)
    System.out.println(arg);
```

- Code in main can print the arguments:

```
for (String arg : args)
    System.out.println(arg);
```
The ArrayList Class

• **The ArrayList class is in java.util package**

• **Instantiating an empty ArrayList**

  ```java
  ArrayList<String> myList = new ArrayList<String>();
  ```

• **Like an array:**
  - ArrayList can store a list of object references
  - You can access each one using a numeric index

• **Unlike an array:**
  - ArrayList object grows and shrinks as needed
  - You don’t use [ ] syntax with an ArrayList object
  - Cannot store primitive types (Use Wrapper classes)
What's a Wrapper?

The `java.lang` package contains a *wrapper class* that corresponds to each primitive type:

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>void</td>
<td>Void</td>
</tr>
</tbody>
</table>
Wrapper Classes

- The following declaration creates an `Integer` object which is a reference to an object with the integer value 40
  
  ```java
  Integer age = new Integer(40);
  ```

- An object of a wrapper class is used in situations where a primitive value will not suffice

- For example, some objects (like ArrayLists) serve as containers for Objects only.

- Primitive values are not objects.
Wrapper Classes

The following declaration creates an `Integer` object which is a reference to an object with the integer value 40.

```java
Integer age = new Integer(40);
```

An object of a wrapper class is used in situations where a primitive value will not suffice.

For example, some objects serve as containers of other objects. Primitive values could not be stored in such containers, but wrapper objects could be.
Wrapper Classes

• Wrapper classes may contain static methods that help manage the associated type
  – For example, the Integer class contains a method to convert digits stored in a String to an int value:
    
    ```java
    num = Integer.parseInt(str);
    ```

• Wrapper classes often contain useful constants
  – For example, the Integer class contains MIN_VALUE and MAX_VALUE for the smallest and largest int values
Autoboxing

• Autoboxing is the automatic conversion of a primitive value to a corresponding wrapper object:

```java
Integer obj;
int num = 42;
obj = num;
```

• The assignment creates the appropriate `Integer` object wrapping a value of 42

• The reverse conversion (called unboxing) also occurs automatically as needed
The ArrayList Class

- The `ArrayList` class is available in the `java.util` package
- Instantiating an empty `ArrayList`:
  ```java
  ArrayList<String> myList = new ArrayList<String>();
  ```
- An `ArrayList` stores references to the class inside the `< >` which allows it to store objects of that class only
- This is a part of Java’s generics capability which you will study further in CS210
The ArrayList Class

• Strings are inserted with a method invocation
  
  ```java
  boolean b = myList.add(string); // to end
  myList.add(index, string); // at index
  ```

• When an element is inserted at a specific index, the other elements are "moved aside" to make room

• If `index > myList.size()`, the method throws an `IndexOutOfBoundsException` exception

• Elements are removed with a method invocation
  
  ```java
  String s = myList.remove(index);
  ```

• When an element is removed, the list "collapses" to close the gap and maintain contiguous indexes
The `ArrayList` class is implemented using an underlying array. The array is manipulated so that indexes remain contiguous as elements are added or removed. If elements are added to and removed from the end of the list, this processing is fairly efficient. But as elements are inserted and removed from the front or middle of the list, the remaining elements are shifted.
Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions.
- Overloading is what allows us to call `myList.add(string)` and `myList.add(index, string)` above.
- If a method is overloaded, the method name is not sufficient to determine which method is being called.
Method Overloading

- The *signature* of each overloaded method must be unique:

- Method Signature includes:
  - The number of parameters
  - The type of each parameter
  - The order of the parameters
Method Overloading

• The `println` method is overloaded:

```java
println (String s)
println (int i)
println (double d)
```

and so on...

• The following lines invoke different versions of the `println` method:

```java
System.out.println ("The total is:");
System.out.println (3);
```
Method Overloading

• The compiler determines which method is being invoked by analyzing the parameters

Invocation

```java
result = tryMe(25, 4.32)
```

```java
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```
Method Overloading

• The return type of the method is not part of the signature
• Overloaded methods cannot differ only by their return type
• Constructors can be overloaded and often are
• Overloaded constructors provide multiple ways to initialize a new object