More on Classes

ArrayList, Wrappers, Enum, Class

Chapter 5, Core Java Volume I

Contents

- Generic Array Lists
- Object Wrappers and Autoboxing
- Enumeration Classes
- The Class Class

Generic Array Lists : ArrayList<T>

- The length of an array is fixed—inconvenient when it is not known in advance.
- ArrayList class manages an array that automatically grows and shrinks on demand.
- ArrayList is a generic class with a type parameter: ArrayList<T>
- Use a type parameter such as ArrayList<Employee> to specify element type.
- Declaring an ArrayList

```
ArrayList<Employee> staff = new ArrayList<Employee>();
ArrayList<Employee> staff = new ArrayList<>();  // diamond syntax
var staff = new ArrayList<Employee>();                    // inferencing the type
var staff = new ArrayList<Employee>(100);        // initial capacity, not size (0)
```

- To add or remove an element:

```
Employee e = new Employee("Harry Hacker", );
staff.add(e);              // append the object e to the end of list
staff.add(i, e);           // insert the object e at the specified position
staff.remove(i);           // remove the object at the specified position
staff.remove(e);           // remove the object e (the first occurrence of the equal element to e)
```

Generic Array Lists

- The call `staff.size()` yields the current size.
  - the size is the number of actual elements in the list, not the size of capacity of the list.
- Access and modify elements with the `get` and `set` methods:

```
Employee e = staff.get(i);
staff.set(i, tony);
```

- Can use "for each" loop to visit all elements:

```
for (Employee e : staff)
        System.out.println(e);
```

Example: Listing 5.11: ArraysList/ArrayListTest.java

```
package arrayList; // this program demonstrates "ArrayList"
import java.util.*
public class ArrayListTest {
  public static void main(String[] args)
  {
     ArrayList<Employee> staff = new ArrayList<>(); // fill the staff array list
      staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
      staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
      staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
      for (Employee e : staff)
          e.raiseSalary(5); // raise everyone's salary by 5%
     for (Employee e : staff)  // printout all information about all Employee objects
          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="+ e.getHireDay());
  }
} // end of ArrayListTest
```

Object Wrappers and Autoboxing

- We may need to convert a primitive type like **int** to an object type.
- All primitive types have "class" counterparts called wrapper classes, namely:
  - Integer, Long, Float, Double, Short, Byte, Character, and Boolean.
- The wrapper classes are *immutable* so we cannot change a wrapped value after the wrapper has been constructed.
- They are *final*, so we cannot subclass them.

```
java.lang.Object                    // root class
        java.lang.Character         // child class of Object class
        java.lang.Boolean           // child class of Object class
        java.lang.Number            // child class of Object class
                java.lang.Long      // child classes of Number  class
                java.lang.Integer
                java.lang.Short
                java.lang.Byte
                java.lang.Float
                java.lang.Double
```

Object Wrappers and Autoboxing

- ArrayList can only hold objects, not primitive values.
  - `ArrayList<int> list = new ArrayList<int>( );  // wrong array declaration`
- It is OK to declare an array list of Integer objects:

```
ArrayList<Integer> list = new ArrayList<Integer>(); // correct declaration
```

- Conversion between int and Integer is automatic:

```
list.add(3) ;          // same as  list.add(Integer.valueOf(3));      // autoboxing(autowrapping)
int n = list.get(i);  // same as int n = list.get(i).intValue();      //auto unboxing(auto unwrapping)
```

- Automatic boxing also works for arithmetic expressions

```
Integer n = 1000;          // auto boxing and n is wrapper reference
n++;                       // unboxing the object , incrementing the value  and auto boxing again
int k = n;                 // unboxing the object;  same as int k = n.intValue();
```

- Caution: the operator == doesn't work with wrappers as shown below

```
Integer a = 100;
Integer  b = 100;
System.out.println(a == b); // the output may be  false and we use  equals() method.
```

Enumeration Classes (enum Types)

- A **enum** type defines a set of instances (**enum** constants) represented with unique identifier.

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

- Declaring and using variables of **enum** types

```
Size s = Size.SMALL;
if(s == Size.SMALL) ...
```

- All **enum** types are reference types
- **enum** constants are implicitly static final.
- Cannot create an object of enum **type** with operator new

Enumeration Classes (enum Types)

- Can add constructors, methods, and fields:

```
public enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
   private String abbreviation;
   private Size(String abbreviation) { this.abbreviation = abbreviation; }  // always private
   public String getAbbreviation() { return abbreviation; }
}
```

- All enumeration classes are subclasses of `Enum<T>` class and inherit methods:
  - `toString` — yields the name "SMALL", "MEDIUM", ...
  - `ordinal` — yields the position 0, 1, ...

- Useful static methods:
  - `Enum.valueOf(Size.class, "SMALL")` yields `Size.SMALL` – `valueOf(Class enumClass, String name)`
  - `Size.values()` yields all values in an array of type `Size[]`

Listing 5.12: EnumTest

```
package enums; // this program demonstrate Enumerated types
import java.util.*;
class EnumTest {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
    String input = in.next().toUpperCase();
    Size size = Enum.valueOf(Size.class, input);
    System.out.println("size=" + size);          // size.toString();
    System.out.println("abbreviation=" + size.getAbbreviation());
    if (size == Size.EXTRA_LARGE)
      System.out.println("Good job--you paid attention to the _.");
  } // end of main()
}//end of EnumTest

enum Size{
  SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
  private String abbreviation;
  private Size(String abbreviation) { this.abbreviation = abbreviation; }
  public String getAbbreviation() { return abbreviation; }
}//end of Size
```

The Class Class

- The Java runtime system always maintaines what is called runtime type identification on all objects.
- The information keeps track of the class to which each object belongs.
- You can access this type information with a special Java class called ***Class***.

- Given an object, get its ***Class*** object: (first method)

```
Employee e = new Employee(...);
Class cl = e.getClass();
```

- Find the class name:

```
System.out.println(cl.getName());
```

- `newInstance` yields an instance constructed with the no-arg constructor:

```
Object newObj = cl.newInstance();
```

Obtaining Class Instances

- Can get a *Class* instance from a string: (second method)

```
String className =  "java.util.Random";
Class cl = Class.forName(className);
```

- Shorthand for class literals: (third method)

```
Class cl1 = Random.class; // if you import java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

  - Class describes a type which need not be a class.

- Class instances are unique:

```
if (obj.getClass() == Employee.class) // Ok
```