27 Object & Wrapper Classes

1 Object class

Java includes a special built-in class called **`Object`**. Every class you introduce is deemed to be an extension of `Object`. This means, in effect, that the methods of `Object` are automatically included in the classes you define, the most important of these being `toString` and `equals`. Even if you do not use these methods explicitly, you will be almost certainly be making use of them indirectly through the use of methods in the Java library (for example, when you use the list classes to be introduced later).

As we saw, the default implementations of `toString` and `equals` don't usually work as we would wish, so Java allows us to *override* them. This means that if we explicitly include our own version of `toString`, say, it replaces or "overrides" the default version. We have seen how to do this.

Variables, in particular parameters, may be declared to be of a type `Object`. Such parameters have the property that the associated argument may be (a reference to) an object of *any* type. For example, the following method prints an object of any type multiply:

```
static void putMany(Object p, int n) { // print p, n times (n>=0)
   for (int i=0; i<n; i++)
```

```
        System.out.println(p);
    }
```

(Recall that an object `p` is printed by displaying the string returned by `p.toString()` – the default one if you haven't supplied one explicitly for `p`'s class.) The following, for example, are legitimate uses of `putMany`:

```
putMany(new Point(3,4), 5);
putMany("Fire!", 3);
```

Of course, `putMany(23,3)`, say, is not legitimate because integers are not objects (but see below).

2 Overriding equals()

To override the default equals for a class called `MyClass`, write an instance method with the following shape:

```
public boolean equals(Object obj) {
    if (obj== null) return false;
    MyClass p = (MyClass) obj;
    … compare contents of this object with those of p …
}
```

It is essential that the parameter be declared to be of type `Object`, even though we know in practice the argument will be of type `MyClass`; the type mismatch is fixed up by a type cast in the body. It is also necessary to include `public` in the header. Here is an example:

```
class Point {
   private int x, y;

  // more  …

   public boolean equals(Object obj) {
      if (obj== null) return false;
      Point p = (Point) obj;
      return x==p.x && y==p.y;
   }

}
```

Example 1: unique occurrences in an array

We write a generic method which counts the number of unique objects in an array, where uniqueness is with respect to object contents. By "generic" we mean that the same method will work for arrays of any kind of object. We illustrate its use assuming the availability of class `Point` above.

```
class Uniques {

    static int uniques(Object[] b) { // number of unique items in b
        int count = 0; // number of unique items encountered
        for (int i=0; i<b.length; i++) {
            int j=0;
            while (!b[i].equals(b[j])) {
                j++;
            }
            if (j==i) count++;
        }
        return count;
    }

    public static void main(String[] args) {
        Point[] ps = {new Point(4,5), new Point(3,4), new Point(4,5)};
        System.out.println(uniques(ps));  // 2 will appear
    }
}
```

Method `uniques()` compares each object `b[i]` in the array with every object `b[j]` that precedes it, i.e. it compares `b[i]` with `b[0]`, `b[1]`, `b[2]`, ..., `b[i]`, stopping when an object `b[j]` is found that is identical with `b[i]`. If none is found before `b[i]` itself, the loop will nevertheless terminate because eventually `j` equals `i` and trivially `b[i].equals(b[i])` is true. In that case `b[i]` has not occurred previously in `b` and so `count` is incremented. The final value of `count` evidently equals the number of unique objects in `b`. As an exercise, confirm that the method doesn't work if `equals` is not overridden, and explain the output you see.

Consistency requirements on equals()

It is important that `equals()` really does perform a genuine equality test. For example, although we do not offend the rules of Java if we define `equals()` for some class as follows

```
public boolean equals(Object obj) {
    return false;
}
```

we should not be surprised if we fail to get sensible behaviour from a program that invokes such a method. If you proceed sensibly following your intuitive understanding of equality you will not meet any difficulties. If you are in doubt, check that your definition satisfies the following requirements:

(i) `p.equals(p)` yields `true`.
(ii) `p.equals(q)` yields the same as `q.equals(p)`.
(iii) If both `p.equals(q)` and `q.equals(r)` yield `true`, so does `p.equals(r)`

3 Wrapper classes

When a formal parameter is of type `Object`, the corresponding argument must be a reference to an object, and not a value of a primitive type (recall that the primitive types are `int`, `long`, `short`, `byte`, `double`, `float`, `char`, and `boolean`). Java helps to overcome this by supplying *wrapper* classes for dressing up primitive values as objects. There is a wrapper class for each primitive type. The wrapper for type `int` is called **`Integer`**. It has the following constructor and methods:

Integer(*int*)
int intValue()
String toString()
boolean equals(*Object*)
int compareTo(*Integer*)
static int parseInt(*String*)

For `k` denoting any integer, **`new Integer(k)`** creates an `Integer` object which encapsulates the integer `k` (actually, class `Integer` merely has an instance variable of type `int` whose value is initialised to `k`). For `p` a reference to an object of type `Integer`, **`p.intValue()`** returns the integer encapsulated by `p` as a value of type `int`, and **`p.toString()`** returns it as a string. For `p` and `q` references to `Integer` objects, **`p.equals(q)`** compares the two encapsulated integers for equality, and **`p.compareTo(q)`** tests them for less-than, equality or greater-than. A negative integer is returned by `p.compareTo(q)` if the integer in object `p` is less than that in `q`, 0 is returned if they are equal, and otherwise a positive is returned. For example, `(new Integer(2)).compareTo(new Integer(5))` yields a negative number because 2 is less than 5. **`Integer.parseInt(s)`** returns the integer equivalent of string `s` (`s` must consist of digits only, possibly prefixed by `+` or `−`). As an example of the usefulness of `Integer`, we can use `putMany()introduced above` to print twenty-five 17's, as follows:

```
putMany(new Integer(17), 25);
```

`Integer` objects are immutable, i.e. no methods are provided which change the value of the encapsulated integer. If you want to effect a change, you have to do so by creating a new `Integer` object, as in the following assignment where `p` is a variable of type `Integer`:

```
p = new Integer(p.intValue() + 1);
```

This assigns to `p` a reference to a new `Integer` object representing a value one greater than the old value. This is clearly cumbersome, and so it is not attractive to use class `Integer` for doing arithmetic.

The wrapper classes for `double`, `char`, and `boolean` are called **`Double`**, **`Character`**, and **`Boolean`**, respectively, summarised in the following table:

Double	Character	Boolean
Double(*double*)	Character(char)	Boolean(boolean)
double doubleValue()	char charValue()	boolean booleanValue()
String toString()	String toString()	String toString()
boolean equals(*Object*)	boolean equals(*Object*)	boolean equals(*Object*)
int compareTo(*Double*)	int compareTo(*Character*)	int compareTo(*Boolean*)
static double parseDouble(*String*)		

4 Autoboxing & autounboxing

Actually in most situations, you may supply a item of type `int` where one of type `Integer` is expected, and analogously for the other primitive types. The system will automatically carry out the necessary wrapping (this is called *autoboxing* although *autowrapping* would be more appropriate). For example, the following statements have identical effects (they each print 17 a total of 25 times):

```
putMany(new Integer(17), 25);
putMany(17, 25);
```

In fact, the compiler automatically translates the second statement to the first.

The compiler will automatically unwrap an `Integer` object that is used where a value of type `int` is expected. For example, if `myInt` references an object of type `Integer` whose constituent value is 3, then the statement

```
int k = myInt + 1
```

leaves variable `k` with the value 4. Strictly this is not type correct, but the compiler will automatically translate it to the following:

```
int k = myInt.intValue() + 1
```

This is called *auto-unboxing*. It is even legal to write

```
myInt = myInt + 1
```

But be very sure of what's really going on here: the compiler supplies the automatic boxing and unboxing so that the statement is nothing but a neat shorthand for

```
myInt = new Integer(myInt.intValue() + 1);
```

If you need to carry out other than trivial arithmetic, it is silly to use type `Integer` – use type `int`. Note that it is an error to rely on autounboxing when the `Integer` argument is null – it has to be a reference an object of type `Integer`. Auto-unboxing applies similarly to all the primitive types.