14 Object & Wrapper Classes

1 Object class

Variables may be declared to be of a special type called **`Object`**. Such variables have the property that they may be assigned a reference to an object of *any* type. For example, the following statement is legitimate (recall that strings are treated as objects of type `String`):

```
Object obj = "Yahoo!";
Object obj1 = new Point(4,5);
Object obj2 = new Date(20,5,2001);
```

Here, we have used classes `Point` and `Date`:

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

   Point(int xx, int yy) {
      x = xx; y = yy;
   }

   public String toString() {
      return "(" + x + "," + y + ")";
   }

}
```

```
class Date {
   private int day, month, year;
   Date (int d, int m, int y) {
      day = d; month = m; year = y;
   }
   public String toString() {
      return day + "/" + month + "/" + year;
   }
   void show(String label) {
      System.out.println(label + toString());
   }
}
```

Remember that integers are not objects and so the following is an error:

```
Object obj = 23; // Error!
```

The main advantage of type `Object` is in declaring formal parameters. If a formal parameter is declared to be of type `Object`, the corresponding argument can be a reference to an object of any type. This allows us to write more widely applicable methods. Consider, for example, the following trivial program:

```
class GenericPut {
   static void putMany(Object p, int n) { // print p, n times (n>=0)
      for (int i=0; i<n; i++)
         System.out.println(p);
   }
   public static void main(String[] args) {
      putMany(new Point(4, 5), 2);
      putMany(new Date (15, 5, 2001), 2);
   }
}
```

The program generates the output

```
(4,5)
(4,5)
15/5/2001
15/5/2001
```

(Recall that an object `p` is printed by displaying the string returned by `p.toString()` – if none is included in the class definition, a default one is provided.) `putMany()` is a versatile method: it displays any type of object a number of times, made possible by declaring the formal

parameter `p` to be of type `Object`. We could even use `putMany()` to display a string several times, as in `putMany("Fire! ",3)` which causes "Fire! Fire! Fire! " to be displayed. Without the use of `Object` as a parameter type, we would have to write a version of `putMany()` for every type of object that is to be multiply printed. Each version would be identical but for the type of the first parameter, and that would breach one of the guiding principles is writing long-lived programs: try not to replicate code. When replicated code is to be amended (which is likely in long-lived programs), it is all too easy to change the code in one place but forget to change it in the other.

`putMany()` as we have written it is said to be *generic*, which means that it accommodates arguments of many types. The term *generic* applies not just to methods but to any program unit, including classes, that are accommodating of many types.

2 Type casting and Object

Although there are no restrictions on assigning a `Point` object, say, or a `Date` object to an `Object` variable, the compiler will complain if you try the reverse (we are continuing to use `Point` and `Date` as introduced in the preceding section). If you want to do this, and you probably will, you have to type cast the object as illustrated in the final two lines of the following code:

```
Object objP = new Point(3,4);
Object objD = new Date(6,7,2003);
Point p = (Point) objP;   // must type cast
Date d = (Date) objD;  // must type cast
```

The type casting in the final two assignments is obligatory. Of course, the type casting must be truthful. The following is erroneous, for example, if `objD` references a `Date` object:

```
Point q = (Point) objD; // Error!
```

When you invoke methods of an object referenced via a variable of type `Object`, type casting is again necessary. For example, given variable `objD` referencing an object of type `Date`, the following statement is legal but would not be so without the type casting:

```
((Date) objD).show("Today's date is ");
```

However, invoking methods `toString()` and `equals()` never requires type casting. For example, the assignment to `s` below is legitimate:

```
Object objP = new Point(3,4);
String s = objP.toString();
```

– it is not necessary to type cast as in `((Point)objP).toString()`. We exploited this in writing `putMany()` above. The statement `System.out.println(p)` in the body of `putMany()`, where `p` is of type `Object`, requires no type casting because it is equivalent to `System.out.println(p.toString())`.

3 Tailoring equals() for class Object OPTIONAL

Class `Object` defines a small collection of methods of which the most important are:

String toString()
boolean equals(*Object*)

These are in fact the default versions of `equals()` and `toString()` that are supplied whenever a class does not provide its own. The default **`equals()`** does not always behave as we would wish. As we have seen, it deems objects to be equal if and only if their references are equal, whereas we more typically want objects to be equal if their *contents* are equal. Hence we may need to provide our own version of `equals()`, such as the following for `Point` objects as introduced above:

```
boolean equals(Point p) {  // Is p equal to this point
    return x==p.x && y==p.y;
}
```

Unfortunately, this definition does not combine well with the use of class `Object`. In particular, if objects of type `Point` are referenced via variables of type `Object`, `equals()` as defined in class `Object` is applied, and this equality compares references. For example, the following code produces output `false` even if class `Point` includes `equals()` as defined above:

```
Object p = new Point(1,2);
Object q = new Point(1,2);
System.out.println(p.equals(q));
```

The solution is to define `equals()` for class `Point` as follows:

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

To guarantee that the new version of `equals()` will be invoked in place of the default in all circumstances, Java requires that the header of `equals()` be `public boolean equals(Object obj)` (the name of the parameter can be any identifier). Although the formal parameter `obj` is required to be of type `Object`, the corresponding argument must be (a reference to) a `Point` object, and so a type cast is needed in the body. An attempt to use the method to compare a `Point` object with a `Date` object, say, will not result in the outcome `false`, but in a run-time error. However, a null argument is accommodated. As another example, for class `Date` as introduced above, an appropriate definition of `equals()` is:

```
public boolean equals(Object obj) {
    if (obj== null) return false;
    Date d = (Date) obj;
    return day==d.day && month==d.month && year==d.year;
```

```
    }
```

Again, note that parameter `obj` is of type `Object`, not `Date`.

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. We illustrate its use assuming the availability of classes `Point` and `Date` as above, each with a tailored `equals()` as described 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)};
        Date[] ds = {new Date(1,2, 2002), new Date(1,2,2002)};
        System.out.println(uniques(ps));  // 2 will appear
        System.out.println(uniques(ds)); // 1 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`.

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 true;
    }
```

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)`

Suppose, for example, that we had defined `equals()` in class `Date` as follows:

```
public boolean equals(Object obj) {
    if (obj== null) return false;
    return year<=((Date)d).year;
}
```

– date equality is determined in a silly way by testing whether the year of the first date is less than or equal to the year of the second date. Although this `equals()` passes tests (i) and (iii), it fails test (ii) and so is not acceptable. The definition we gave earlier passes all three tests.

4 Wrapper classes and autoboxing

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`). This is a pity: we may invest a large effort in writing a generic method, and it is reasonable that we would wish it to be as widely applicable as possible. Java recognises this and supplies *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(Integer)
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`, recall the method:

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

We can use `putMany()` 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 do this, you have to create 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(*Double*)	boolean equals(*Character*)	boolean equals(*Boolean*)
int compareTo(*Double*)	int compareTo(*Character*)	int compareTo(*Boolean*)
static double parseDouble(*String*)		

For example, `putMany(new Double(3.14),5)` prints 3.14 five times, and if string `s` contains "3.14" then `Double.parseDouble(s)` yields 3.14 as a value of type `double`.

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. Note that although the second statement is neater, it's execution time is no different from that of 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.

5 Type Parameters OPTIONAL

We may parametrise a class definition with a type. Each time we create an instance of the class we must supply a real type as argument. Consider:

```
class Pair<T> {

   private T first;  // first item
   private T second; // second item

   Pair(T x, T y) {
```

```
        first = x; second = y;
    }

    T getFirst() {
     return first;
    }

    T getSecond() {
     return second;
    }

}
```

By writing `<T>` after the class name we introduce `T` as a *type parameter*. The name `T` is freely chosen according to the standard rules for choosing identifiers (although it is conventional to use single uppercase letters to stand for types). `T` may be used as a type in the definition of `Pair`, and indeed we have used it several times in this way. Each time we make use of type `Pair` outside its definition we must supply an actual type for `T`, as typified by the following:

```
Pair<String> pairString = new Pair<String>("Hi", "There");
Pair<Integer> pairInt = new Pair<Integer>(new Integer(3),  new Integer(4));
```

The actual type must be supplied both when `Pair` is used to introduce a variable and when we write a `Pair` constructor. The return type of `pairString.getFirst()` is `String`: this is because `getFirst()` is declared to have a return type of `T` and we instantiated `T` as `String` when we created `pairString`. Similarly, the return type of `pairInt.getFirst()` is `Integer`:

```
class PairTest {
   public static void main(String[] args) {
      Pair<String> pairString = new Pair<String>("Hi", "There");
      String s = pairString.getFirst();
      Pair<Integer> pairInt = new Pair<Integer>(new Integer(3),  new Integer(4));
      Integer k = pairInt.getSecond();
      System.out.println(s + " " +  k);
   }
}
```

The above program displays

```
Hi 4
```

It is not possible to supply a primitive type as type argument. For example, the following is illegal:

```
Pair<int> pairint;
```

A class definition can have several type parameters, as in:

```
class Pair2<T,U> {

   private T first;  // first item
   private U second; // second item

   Pair2(T x, U y) {
    first = x; second = y;
   }

   T getFirst() {
    return first;
   }

   U getSecond() {
    return second;
   }

}
```

The following are illustrative uses of `Pair2`:

```
Pair2<String,Integer> pairSI = new Pair2<String,Integer>("Hi", new Integer(4));
Pair2<String,String> pairSS = new Pair2<String,String>("Hi", "There");
```

6 Example: a generic stack OPTIONAL

A *stack* is a list of items to which elements are added and from which elements are removed at the end of the list only. Items are never inserted at the front or the middle of a stack, nor removed from the front or the middle, only at the end. Adding an item to a stack is called *pushing* the item onto the stack, and removing an item (from the end of the stack, of course) is called *popping* the stack. The final item in the stack is called the *top* of the stack. It turns out that stacks occur quite frequently in programming. A *generic stack* is one which accommodates items of some arbitrary type. The following encodes a generic stack:

```
class Stack<T> {

   private Object[] s = new Object[10000]; // max size of stack is 10000 ...
   private int n = 0; // ... only first n items significant

   void push(T item) { // push item (assume space available in s)
      s[n] = item;  n++;
   }

   T pop() { // pop (assume s not empty)
      n--;
```

```
        return (T)(s[n]);
    }

    boolean isEmpty() { // Is s empty?
        return n==0;
    }
}
```

When `Stack` is compiled we get a warning message with respect to the type casting in `(T)(s[n])` (in method `pop()`). All use of type parameters is eliminated by the compiler – it actually translates the program we write into a similar one that makes no use of type parameters. Type casting is a run-time operation, however, and the best the compiler can do when the type casting uses a type parameter it just to ignore it. The compiler warns that it is doing so.

As an example of using `Stack`, the following program reads a sequence of integers from the keyboard and displays them in reverse order, one integer per line:

```
class ReverseStrings {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<Integer>();
        while (!Console.endOfFile()) {
            int m = Console.readInt();
            stack.push(m);  // using autoboxing
        }
        while (!stack.isEmpty()) {
            int m = stack.pop(); // using autounboxing
            System.out.println(m);
        }
    }
}
```