12 Classes: Class Parameters

1 Class parameters

Class `Point` (say) may be freely used within the definition of class `Point`. It is especially common for instance methods in a class `Point` (say) to have parameters of type `Point`. For example:

```
class Point {
        double x, y;  // cooordinates

        void getPoint() {
                x = Console.readDouble();
                y = Console.readDouble();
        }

        double distance(Point p) { // distance of this point from point p
                return (Math.sqrt((x - p.x) * (x - p.x) + (y - p.y) * (y - p.y)));
        }

}
```

Observe that `distance` takes a parameter of type `Point`. In the body of `distance`, `x` unadorned with a prefix denotes that `x` in the same object as the current incarnation of `distance`, and `p.x` denotes that `x` in the object referenced by parameter `p`. If `p1` and `p2` are each variables of type `Point`, then the distance between them is obtained by invoking either `p1.distance(p2)` or `p2.distance(p1)`. The following example assumes `Point` as defined above.

```
class PointsDist {
        public static void main(String[] args) {
                Point p1 = new Point();
                Point p2 = new Point();
                p1.getPoint();
                p2.getPoint();
                System.out.print("The distance between them is " + p1.distance(p2));
        }
}
```

We might equally well have written `p2.distance(p1)` in place of `p1.distance(p2)`.

2 The redundant parameter trap

Remember that an instance method belongs to an object, and has free access to the object's instance variables. It is very important to grasp the significance of this: *every instance method has free access to the instance variables of the object to which it belongs*. Many beginners forget this and introduce a redundant object parameter. For example, they may try to write a method `distance2` in place of `distance` as follows:

```
        double distance2(Point p, Point q) { // distance between two points, but BAD!
                return (Math.sqrt((p.x - q.x) * (p.x - q.x) + (p.y - q.y) * ( p.y -q.y)));
        }
```

Method `distance2` above determines the distance between two points, but it is poor coding aninstance method (although technically correct) in that both points are passed as parameters. `distance2` is an instance method and so has free access to a point, i.e. to the `x` and `y` variables in the object in which it resides. Therefore it only needs access to one other point, and that is passed as a single parameter. Method `distance` written earlier exhibits the correct object-oriented style of coding.

In general, instance methods tend to have one less parameter than you might think at first, because every invocation of an instance method has free access to everything in the object to which it belongs.

3 Describing the effect of an instance method

Comments in instance methods often refer to the object to which it belongs as "me" or "I" or "this point", "this employee", "this date" etc., or "the point", "the employee", "the date" etc. For example, the behaviour of `distance(Point p)` may be described by any of the following comments

- // distance of this point from point P
- // distance from point P
- // my distance from point P
- // how far am I from point P
- // distance of the point from point P

4 Class result types

An instance method may return a value of a class type, even of the same class to which the method belongs. For example, an instance method in class `Point` (say) may return (a reference to) an object of type `Point`. Consider the following example which is a program to read a point and compute its reflection in the x-axis (the reflection of (x,y) in the x-axis is (x,-y)).

```
class Point {
        double x, y;  // coordinates

        void getPoint() { // read coordinates from keyboard
                x = Console.readDouble();
                y = Console.readDouble();
        }

        void putPoint() { // display point
                System.out.print("("+ x + "," + y + ")");
        }

        Point xReflect() { // create and return x-reflection of this point
                Point pt = new Point();
                pt.x = x; pt.y = -y;
                return pt;
```

```
    }
}
```

In this case we have designed `xReflect` to create and return a *new* Point object, rather than changing the current one. In other words, `xReflect` is a function, not a procedure – the point object is not changed, but rather a new one is created. (In another design, we might have been required to write a similar method as a procedure which changes the object to represent the x-reflection of the original point.)

We might have included an all-args constructor for class Point above:

```
    Point(double x0, double y0) {
        x = x0; y = y0;
    }
```

Had we done so, we could use it to write an alternative and very succinct version of `xReflect`:

```
    Point xReflect() { // create and return x-reflection of this point
        return new Point(x, -y);
    }
```

An example of a program which uses class point is:

```
class PointsDist{
    public static void main(String[] args) {
        Point p = new Point();
        p.getPoint();
        Point r = p.xReflect();
        System.out.print("The x-reflection of ");
        p.putPoint();
        System.out.print(" is ");
        r.putPoint();
    }
}
```

Note that we wrote `Point r = p.xReflect;` and not `Point r = new Point(); r = p.xReflect;`. While the latter is not incorrect, it needlessly creates a Point object that is immediately discarded. The code can be written more compactly by removing the declaration and assignment to `r`, and replacing r.putPoint() with (p.xReflect()).putPoint().

5 Example: Date of following day

We write a program to read a date and print the following day's date. A typical input/output is:

```
Enter day month year: 30 4 2011
The day after 30/4/2011 is 1/5/2011
```

```
class Date {

    int day; int month; int year;

    Date(int day0, int month0, int year0) {
        day = day0; month = month0; year = year0;
    }

    Date(){}

    void getDate() {
        System.out.print("Enter day month year: ");
        day = Console.readInt();
        month = Console.readInt();
        year = Console.readInt();
    }

    void putDate() {
        System.out.print(day + "/" + month + "/" + year);
    }
```

```
    int daysInMonth() { // Number of days in month
        if (month==9 || month==4 || month==6 || month==11)  return(30);
        else if (month==2) {
            if (year%4==0 && year!=1900) return(29);
            else return(28);
        }
        else return(31);
    }

    Date nextDay() { // day after this day
        if (day<daysInMonth()) return new Date(day+1, month, year);
        else if (month<12) return new Date(1, month+1, year);
        else return new Date(1, 1, year+1);
   }
}

class Tomorrow {
   public static void main(String[] args) {
        Date d1 = new Date(); d1.getDate();
        System.out.print("The day after "); d1.putDate();
        System.out.print(" is ");
        Date d2 = d1.nextDay(); d2.putDate();
    }
}
```

The final line in the preceding method may be written more succinctly as

```
    (d1.nextDay()).putDate();
```

6 Example: least name

Let us write a program to read a list of persons' names, one name per line of input. Each name consists of a forename followed by a space followed by a surname. The program should display the name which comes first in the usual phone-book ordering of names, output as surname followed by forename separated by a comma. For example, the input

```
Charles Darwin
Marie Curie
Neils Bohr
Albert Einstein
```

should give rise to the output

```
Bohr, Neils
```

The input has at least one name. The program follows. Note that `lte` takes just a single `Name` parameter although it compares two names.

```
class Name {  // The name of a person

   String forename, surname; // first and second names

   void get() {  // Read name from keyboard
      forename = Console.readToken(); // forename is one word
      surname = Console.readString(); // surname is rest of line
   }

   void put() { // Write name in form "Smith, Fred"
      System.out.println(surname + ", " + forename);
   }

   boolean lte(Name s) { // My name precedes or equals s?
      return(surname.compareTo(s.surname)<0 ||
         (surname.equals(s.surname) && forename.compareTo(s.forename)<=0));
   }
```

```
}

class LeastName {

   public static void main(String[] args) {
      Name least = new Name();
      least.get(); // least name so far
      while(!Console.endOfFile()) {
        Name tempName = new Name();  // Big banaskin dodged – see below
        tempName.get();
        if (tempName.lte(least))
           least = tempName;
      }
      least.put();
   }
}
```

In the above program, we only retain access to (at most) two objects, one accessed via `least` and the other accessed via `tempName`, and yet we create an object each time a name is read. Hence the program creates lots of orphans, but this is not a problem.

Big banana skin! You may be tempted to avoid the creation of a name object for each name read. If so, you may effect this by placing the declaration `Name tempName = new Name();` alongside the declaration of `least` (removing it from the body of the loop). Can you see why this will fail?

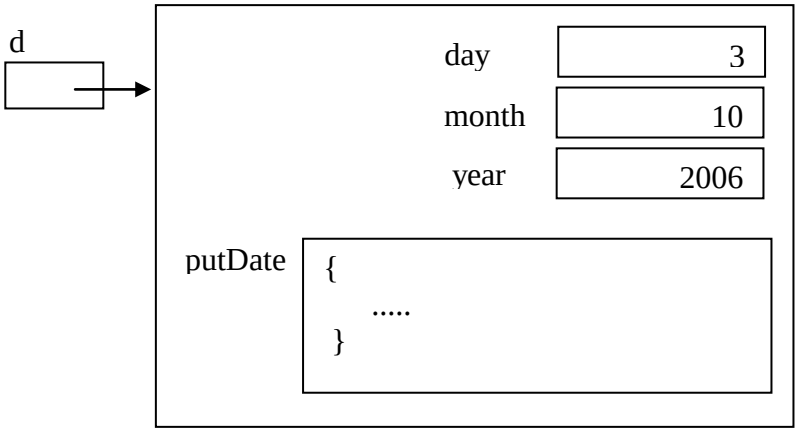
7 Mixing static and instance methods

It is common for classes to include both instance and static methods. Remember that static methods exist as soon as the class is loaded – they are not included in objects. A method should be static if it has no good reason to access instance variables or instance methods nakedly. Of course, a correct static method would not be made wrong by changing it to an instance method (by just omitting `static` from its header), but that would be needlessly expensive computationally. It would mean that a copy of the method would be created in every object of the class, even though the extra power available to instance methods is not used.

Consider the following class regarding dates. The class includes an instance method `putDate` which prints the date in a form typified by `03-10-06` (day followed by month followed by year, each occupying two digits).

```
class Date {

  int day; int month; int year;

  Date(int d, int m, int y) {
        day = d; month = m; year = y;
  }

  static String dig2(int n) { // n as 2-digit string, 0<=n<100
        return("" +  n/10 + n%10);
  }

  void putDate() {
        System.out.println(dig2(day) + "-" + dig2(month) + "-" + dig2(year%100));
  }

}
```

`putDate` uses *static* method `dig2` to compute a 2-character string representation of the rightmost digits of its argument. `dig2` makes no reference to any instance variables in the class and so it would be both misleading and computationally expensive to write it as an instance method. Remember that static methods exist as soon as the program is loaded – they are not included in objects. An execution of `Date d = new Date(3,10,2006)`, for instance, creates:



– note that the object does *not* contain `dig2`.

A static method may be invoked by an instance method. When both are defined in the same class, they are invoked by using just the simple name of the static method (as in the example above). If the static method is defined in a different class, then the name of the static method must be prefixed with the name of the class in which it is defined, as in `Character.isDigit(c)`.

Static methods may call instance methods. They must *always* identify the object in which the instance method resides using the usual prefix notation.