11 Classes: instance methods

1 Introduction to instance methods

Methods are either static or *instance*. Instance methods are also called *dynamic* or *non-static* methods. They are easily recognised – they don't have the keyword `static` in the header. Otherwise, they look similar. For example:

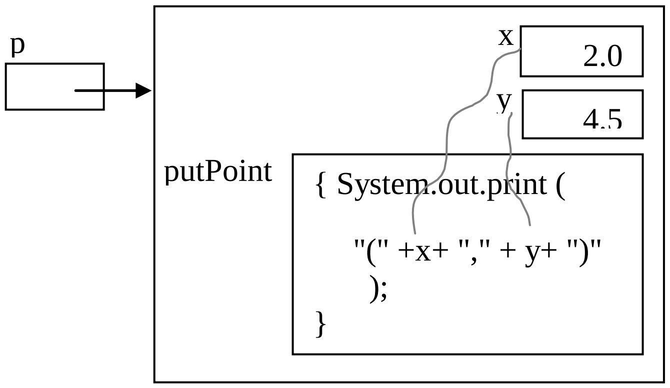
```
class Point {
    double x, y;

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

There are two key points to observe in method `putPoint` above: (i) no `static` in the header which marks it out as an instance method, and (ii) `x` and `y` are referred to without indicating the intended object (as in, say, `p.x` or `p.y`). Point (ii) illustrates the crucial advantage of instance over static methods: *instance methods may refer to the instance variables of the class in which they are defined without qualification*.

To understand what this means, suppose we create an object of type `Point`, as follows, say:

```
Point p = new Point();
p.x = 2.0; p.y = 4.5;
```

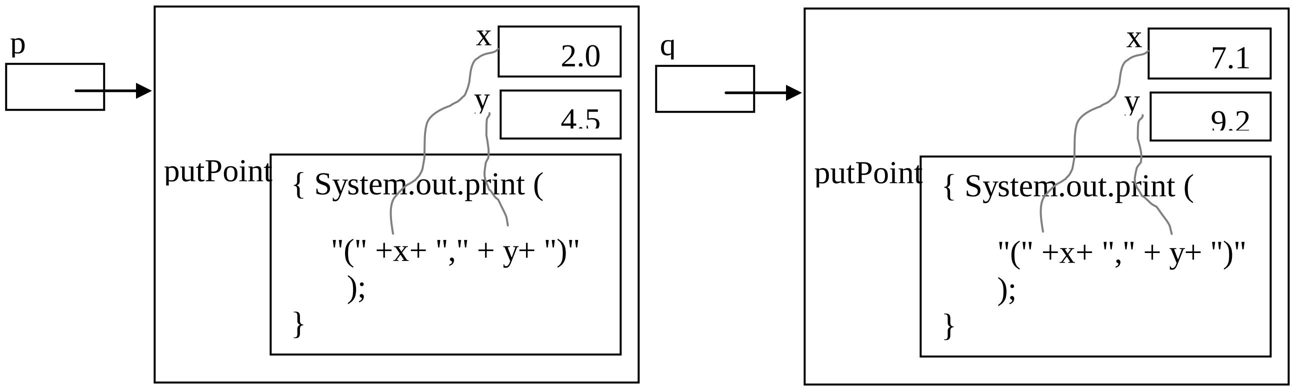
then we can picture the effect as:



Every object we create has a copy of each instance method in the class. In contrast, only one copy of a static method exists, and it exists from the time the class is loaded at run time. So every `Point` object has a version of `putPoint`. To emphasise that we really do get a copy of `putPoint` inside each `Point` object, suppose we created two `Point` objects:

```
Point p = new Point(); p.x = 2.0; p.y = 4.5;
Point q = new Point(); q.x = 7.1; q.y = 9.2;
```

The situation created is:



Observe that each copy of `putPoint` differs slightly from the others in that each mention of `x` and `y` in the body of `putPoint` is locked onto the `x` and the `y` *in the object where the copy lives*. This is indicated by the wavy lines in the diagram above.

Like instance variables, an instance method only comes to life when an object of the class is created. Indeed, there will be precisely as many versions of `putPoint` in existence as there are objects of type `Point`. And each version will be slightly different, because each is locked onto the instance variables in its own object.

When we invoke an instance method we must indicate which copy we are invoking – by prefixing the method with an object reference. In the example above, we might invoke `p.putPoint()` – which prints (2.0,4.5) – or `q.putPoint()` which prints (7.1,9.2).

2 Elementary examples

The following trivial program displays (2.0,4.5):

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

        Point(double xval, double yval) { // constructor
                x = xval; y = yval;
        }

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

}

class PointDemo2 {
        public static void main(String args[] ) {
                Point p = new Point(2.0,4.5);
                p.putPoint();
        }
}
```

Could we have made `putPoint` a static method in the above program, by including `static` in its header? No we couldn't, because `putPoint` uses the freedom to refer to instance variables nakedly (in writing `x` rather than `p.x`, say), and that can only happen in instance methods.

It is important to recognise that the point printed by `putPoint` is not supplied as an argument; rather, it is accessed directly within the object to which this version of `putPoint` belongs. Once you decide which version of `putPoint` to invoke, you have also committed to the point that will be printed.

We can make the program a little more general by reading the coordinates of each new point. This version has a second instance method.

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

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

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

class PointDemo3 {
        public static void main(String args[] ) {
                Point p = new Point();
                p.getPoint();
                p.putPoint();
        }
}
```

Note that when we invoke an instance method such as `getPoint` to initialise an object with values entered at the keyboard, we have to create the object before invocation – otherwise the necessary instance method doesn’t exist.

The local variable trap

When writing an instance method, be careful not to hide the instance variables by introducing local variables of the same name. The following coding of `getPoint` illustrates this.

```
        void getPoint() {  // read coordinates
                double x = Console.readDouble(); //WRONG
                double y =  Console.readDouble(); // WRONG
        }
```

It is seriously wrong to have declared `x` and `y` above by prefixing their use with `double`. The effect is to introduce two additional `x` and `y` variables local to `getpoint` where the point coordinates read from the keyboard will be placed, rather than placing them in the instance variables where they should be.

The instance method trap

Remember that instance methods belong to objects, and so if no objects of the class have been created, then there are no copies of the method. You cannot use an instance method without having first created at least one instance of the class where it is defined. Hence the following code contains errors:

```
class MyClass {
        int x;
        void  inc() { x = x+1;}
        ...;
}

class MyClassTest {
        public static void main(String[] args) {
                MyClass t = new MyClass();
                ....
                inc();                  // Error – must prefix inc() with an object
                MyClass.inc();          // Error –inc() prefixed with class name
                t.inc();                // ok
                .....
        }
}
```

3 Invoking instance methods locally

Just as instance methods may refer nakedly to instance variables in the same object, so they may also refer nakedly to methods in the same object. Whereas outside a class (class `Point`, say) we invoke method `p.putPoint()`, say, where `p` identifies a `Point` object, code inside class `Point` can invoke `putPoint()` – in this case, the local version of `putDate()` is invoked, i.e. the one that resides in the same object. This is illustrated below. Of course, `putPoint()` in some other object can be invoked but then the other object must be identified in the usual way.

4 Object-oriented programming

The example that follows illustrates the object-oriented programming style of programming. This entails identifying the important concepts in the problem statement, and incorporating them into the solution as types encapsulated as classes.

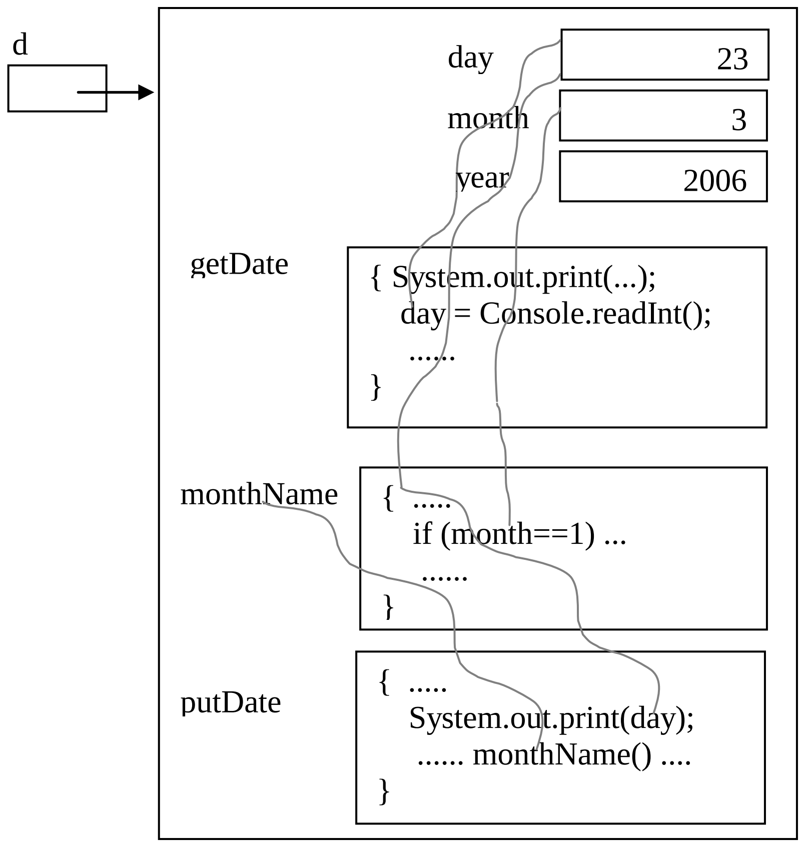
We write a program to read a date – typified by 23 3 2006 , say – and print it in a form typified by 23rd March 2006. We identify the notion of “date” as central and so begin by writing a corresponding type called Date.

```
class Date {

  int day, month, year;

   void getDate() {
        day = Console.readInt();
        month = Console.readInt();
        year = Console.readInt();
   }

   String monthName() {
        String[] name = ; {"February", "March", "April", "May", "June",
                        "July", "August", "September", "October", "November",
                        "December"};
        return name[month-1];
   }

   void putDate() {  // in form e.g. 23rd March 2006
        System.out.print(day);
        if (day%10==1 && day!=11) System.out.print("st");
        else if (day%10==2 && day!=12)) System.out.print("nd");
        else if (day%10==3 && day!=13) System.out.print("rd");
        else System.out.print("th");
        System.out.print(monthName() + "  " + year);  // Note monthName()!
   }
}

class PrintDate {
   public static void main(String[] args) {
        Date d = new Date();
```

```
            d.getDate();
            d.putDate();
        }
    }
```

Note the invocation of `monthName()` in `putDate()` – it invokes that version of `monthName()` *in the same object*. After `d.getDate()` has completed, and supposing the user enters 23 3 2006, we have the situation depicted below. When `d.putDate()` is invoked, it in turn invokes `monthName()`. The use of `monthName` within `putDate` does not need a prefix to identify it as long as our intention is to invoke the version of `monthName` that resides within the same object.

Remember that when we invoke an instance method such as `getDate` to initialise an object with values entered at the keyboard, we have to create the object before invocation – otherwise the necessary instance method doesn't exist. Observe also that `getDate` and `putDate` have no parameters – all the variables they need reside in the object to which they belong; this is often the case with instance methods.


d
day
23
month
3
year
2006
getDate
{ System.out.print(...);
day = Console.readInt();
......
}
monthName
{ .....
if (month==1) ...
......
}
putDate
{ .....
System.out.print(day);
...... monthName() ....
}