18 More on Classes

1 toString

Java allows us to supply an object wherever a string is expected. The run-time system will automatically apply a conversion function to create a string representation of the object. However, the generated string is neither elegant nor informative. We can instead provide a tailored conversion routine for each class, if we wish, as a string-valued instance function called toString(). This is illustrated below:

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

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

   public String toString() { // header must be exactly as written, incl. public
      return "(" + x + "," + y + ")";
   }
}

class ToStringTest {
        public static void main(String[] args) {
                Point p = new Point(3,4);
                System.out.println("My point is " + p);
        }
}
```

The effect of the println statement is exactly the same as

```
System.out.println("My point is " + p.toString());
```

– the system substitutes p.toString() for p. Hence the following message appears:

```
My point is (3,4)
```

The only requirement on the definition of toString() is that it return a string and be explicitly marked public. Note that toString() must include `public` explicitly in its declaration. Remember also that toString is a function, not a procedure (so the string representation of the object is not printed inside toString, but computed and returned).

2 equals for objects

Equality testing of objects using == is not terribly useful as it tests for equality of their references, which is almost certainly *not* what you want. Consider the following program:

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

    Point(int xval, int yval) {
        x = xval; y = yval;
    }
}

class Equality {

    public static void main(String args[] ) {
        Point p = new Point(3,4);
        Point q = new Point(3,4);
        if (p==q)  System.out.println("They're equal!");
        else System.out.println("They're not equal!");
    }
}
```

This produces the somewhat surprising output `They're not equal`. Even if you write the equality test as

```
if (p.equals(q))  System.out.println("They're equal!");
```

the program will still not behave as expected because the default behaviour for .equals on objects also compares object references rather than object contents. If you need to test for equality of objects (of a given class), include a boolean-valued method in the class definition. It is usual to call the method "equals".

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

        Point(int xval, int yval) {
                x = xval; y = yval;
        }

        boolean equals(Point p) {
        // Points are equal if their respective components are equal
           return (p!=null && x == p.x && y == p.y);
        }
}

class Equality2 {

        public static void main(String args[] ) {
                Point p = new Point(3,4);
                Point q = new Point(3,4);
                if (p.equals(q))  System.out.println("They're equal!");
                else System.out.println("They're not equal!");
        }
}
```

When the above program is run, `They're equal!` will be output, as we would wish. We will see a more sophisticated version of `equals` later.

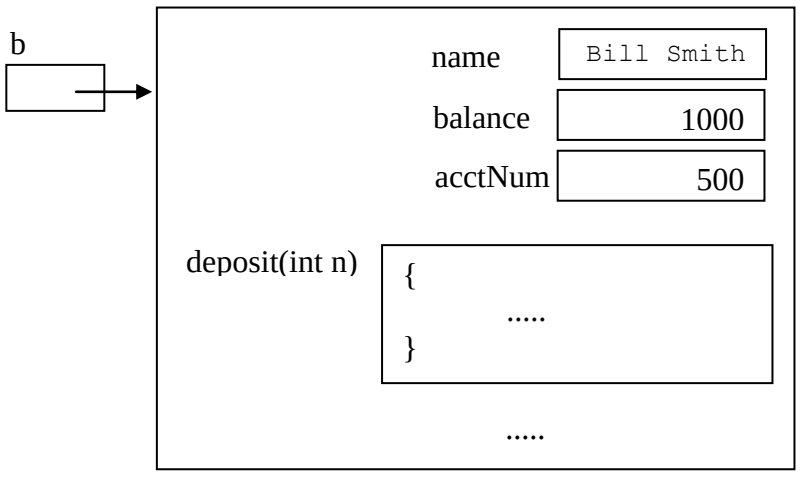
3 Static variables

Static variables (as previously described) have the property that *they are created just once* when the program is loaded and live as long as the program. Unlike instance variables, they do not live

in objects, but in the program as a whole. A class may contain both static and instance variables. For example, consider the following simple class for bank accounts;

```
class BankAcct {

        private String name;       // name of account holder
        private int balance;       // amount in account (cents)
        private int acctNum;       // unique account number

        private static nextFreeNum = 1;

        BankAcct (String name0, int balance0) {
                name = name0; balance = balance0;
                acctNum = nextFreeNum; nextFreeNum++;
        }

        void deposit (int n){
                ...
        }
        ...
}
```

Note that variable `nextFreeNum` is static, and that it must be so (what would happen if it wasn't?). `nextFreeNum` is a variable of the program; there isn't a version of it in every object. For example, an execution of `BankAcct b = new BankAcct("Bill Smith",1000)` creates (assuming this is the 500th time that an account has been created):



– note there is no occurrence of `nextFreeNum` in the created object.

(Technical aside: Variables of `String` type contain references to strings rather than a string itself. However, it is a consequence of the fact that strings are immutable (i.e. they cannot be changed once created) that we may picture string variables as containing the actual string. Hence, the representation of "Bill Smith" above.)

All the methods in a class (whether instance or static) can refer freely to the static variables of the class. A static variable (that is not marked as private) can be accessed by methods in other classes, but only by using its full name. The full name of a static variable x defined in class MyClass is MyClass.x (note the period).

Note that variables declared inside a method cannot be marked `static` – variables can only be marked static when declared directly in a class definition.