9 Classes: common errors

1 The big traps

The no-object trap

A declaration of a variable of a class type does not create an instance of the class. For example, the declaration

```
Point p;
```

only introduces a variable of type reference to Point, but it does not create a Point object. Hence the code

```
Point p; p.x = 1.2;
```

is erroneous because p.x does not exist at that point (p has the initial default value of *null*). Remember that variable p does not refer to an object until it is assigned a reference to an object. This is typically done by executing `p = new Point()` which leaves p referring to a newly created object.

The needless-object trap

Sometimes, we needlessly create an object when a variable of a class type is declared, as illustrated by the following example. Suppose we have two point objects referenced by variables `p` and `q`:

```
Point p = new Point(); p.x = 1.2;  p.y = ...;
Point q = new Point(); q.x = ...;  q.y = ...;
```

Suppose that later in the program, we want to interchange `p` and `q`; for this we need a temporary variable of type `Point`. It is tempting to write:

```
Point t = new Point(); // silly!
```

```
t = p; p = q; q = t;
```

Although this code will work, it is silly to initialise `t` by creating a new Point object and placing a reference to it in `t` – because in the very next statement we assign `p` to `t`, and so the object we created plays no role at all. Instead `t` should be introduces as either

```
Point t;
```

or

```
Point t = null;
```

The instance variable trap

The variables in an object are called *instance* variables (because they belong to an instance of the class). Whenever you refer to an instance variable, you must indicate the particular object in which the variable sits. The following code contains errors often made by beginners:

```
class MyClass {
        int x;
        ...;
}

class MainClass {
    public static void main(String[] args) {
        MyClass p = new MyClass (); // ok
        ....
        p.x = 7; // ok
        MyClass.x = 5; // Error!
        x = 3; // Error!
        .....
    }
}
```

It is crucial to get it absolutely clear that instance variables belong to objects, and do not exist merely by their occurrence in the class definition. Let x be an instance variable declared in class MyClass: *No variables called x are created until the programmer creates an object of type MyClass, and as many variables x are created as objects of type MyClass.* Whenever we refer to

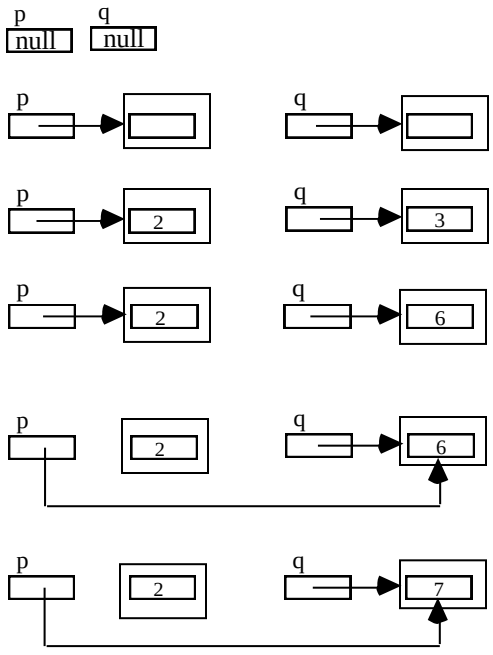
variable x, we must indicate *which* x, and we do this by prefixing it with a reference to the object of which it is a component (as in `p.x` where `p` is a variable of type `MyClass`).

Variables introduced in a class that are not marked `static` are said to be *instance* or *dynamic* variables; this is rather lazy terminology as strictly speaking these variables exist in objects of the class rather than the class.

2 Orphans and aliases

An orphan is an object which is no longer accessible to the program because we no longer retain a reference to it. Orphans arise naturally. The memory they occupy is released for later use by a component of the Java runtime system called the *garbage collector* which runs in the background. We illustrate with a (completely useless) program:

```
class Triv {
       int x;
}

class Silly {

  public static void main(String[] s) {
   Triv p, q;


   p = new Triv(); q = new Triv();


   p.x = 2; q.x = 3;


   q.x = p.x + 4;


   p = q;


   q.x = 7;


   System.out.print(p.x);

}
```



7 appears on the screen

The assignment p = q above creates an "orphan" (the object containing 2); the memory it occupies will be released eventually by the garbage collector.

In contrast to orphans (for which we have no access), some objects may have many access paths. When there is more than one way to access an object, the object is said to be *aliased*. In the trivial program above, for example, the object containing 7 can be accessed via variables p or q and so is aliased. Usually aliasing arises intentionally and presents no problems. However, occasionally it catches us out as explained below.

The aliasing trap

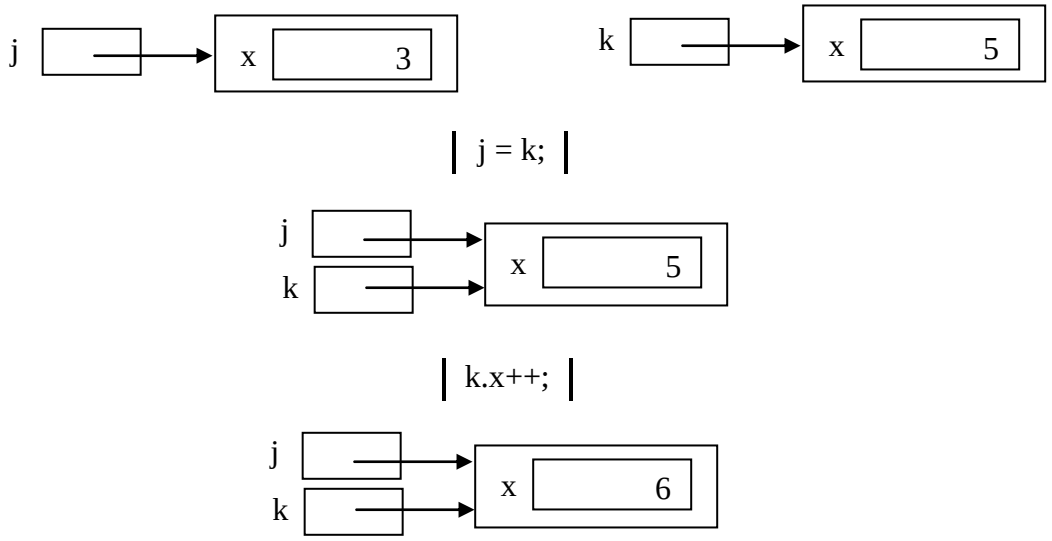
Recall that integer variables actually *contain* their integer values, and similarly for reals, chars, booleans, and floats. All other types of variable contain *references to* objects (references are also called "pointers" or "handles"). It is important to understand the distinction between them, because it affects the way assignment behaves. Consider the following code fragment

```
int j, k
j = 3; k = 5;
j = k; // integer copied!
k++;
System.out.println(j);
```

Obviously this code displays 5 – the assignment k++ has no effect on the value of j. Now consider the following which is similar to the above, but involves assignment of objects rather than integers

```
class Triv {
    int x;
}
....
Triv j, k;
j = new Triv(); j.x = 3;
k = new Triv(); k.x = 5;
j = k; // reference copied!
k.x++;
System.out.print(j.x);
```

Surprisingly for some, 6 is output rather than 3. The point to understand is that the assignment `j=k;` is effected by copying references, not values, leaving j and k referring to the same object – see the picture below.



Subsequently, any change to the state of the object accessed via j is concomitantly a change to the object accessed via k. If you do not want this, then the assignment j = k should be replaced with code which makes an exact copy of the object referenced by k, and assigns a reference to the new object to j:

```
j = new Triv(); j.x = k.x;  // new copy of k's object created!
```

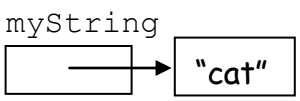
Actually, it turns out in practice that making copies of objects is not much needed – copying references usually suffices. We will see a more systematic way to copy objects later.

3 Strings are immutable objects

It so happens that strings are treated as objects in Java. The `String` type in Java is a built-in class, and hence variables of type `String` contain not an actual string but a reference to a string. For example, the declaration

```
String myString = "cat";
```

creates the situation depicted below:

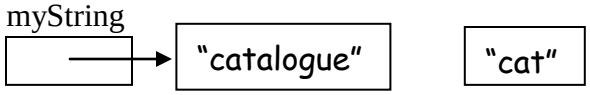


Hence, a variable of type `String` may be assigned the value *null*.

You never have to worry about inadvertently aliasing strings. This is because strings are "immutable", i.e. we can never change the value of a string. Of course, we can write an assignment such as

```
myString = myString + "alogue";
```

This does indeed change the string referred to by `myString`, but the change is effected not by modifying the original string, but by creating a new string similar to the original one but with "alogue" appended. The situation can be depicted as follows:



The original string object (containing "cat") becomes an orphan, which will eventually be discarded by the garbage collector.