Access Attributes

In Java you can declare properties such as variables and methods to be

- public
- protected
- private

You also have the option of saying nothing about access.

To see the difference in these, you need to be aware that classes in Java can be collected into groups called *packages*.

public properties can be seen and modified anywhere in the program.

protected properties can be seen and modified anywhere within their package. They can also be inherited by subclasses. So if you are writing a system for a bank and you have a protected variable balance within a class called UserAccount, even if you don't let anyone see your UserAccount code, someone can get access to the balance variable by making a subclass of UserAccount.

private properties are visible only within their classes. They can't be inherited by subclasses.

The default access level (when you say nothing about access) is to make properties visible within their package, but not to subclasses outside their package. This is the level of protection you get if you don't say the property is public, protected or private.

For what we'll be doing this year (all of the code for a project is contained in one package) the public, protected and default access types are the same: protected objects can be seen or modified anywhere in the program, private objects can only be seen or modified within their own class.

We will generally use the attributes private and public whenever we want to be explicit about access.

static is an attribute of java objects that causes a lot of confusion. When an object is declared to be static that means it belongs to a class rather than to any particular object of that class.

For example consider this class:

```
public class Person {
        private String name;
        public int populationCount = 0;
        public Person( String who ) {
                name = who;
                populationCount += 1;
        }
}
```

Every Person has their own name, of course. The way this class is written every Person also has their own populationCount. That isn't what we want.

If we make populationCount static:

```
public class Person {
        private String name;
        public static int popCount = 0;
        public Person( String who ) {
                name = who;
                popCount += 1;
        }
}
```

then there is just one popCount for the class, and it keeps track of how many Persons have been constructed.

Methods can also be declared static. A static method of a class can be called without constructing an object of the class. Static methods are called with className.methodName() rather than objectName.methodName()

Static methods can't refer to any non-static properties of their class and can't call any other non-static methods without constructing objects for them.

Consider this variation on the Person class:

```
public class Person {
        static int popCount = 0;
        private String name;
        int myCount;

        public Person(String who) {
                name = who;
                popCount += 1;
                myCount = popCount;
        }

        public void history( ) {
                System.out.printf("When %s was created they were Person number %d\n",
                        name, myCount);
        }
```

The only difference is that Persons now have a variable *myCount* that remembers which element of the population they are.

Here is a new main( ) method for class Person:

```
public static void main(String[] args) {
        Person x = new Person("bob");
        Person y = new Person("mary");
        x.history();
        y.history();
}

} // closes the person class
```

We don’t need to construct an object of class Person for main() to make sense, so main( ) can be declared to be static.

If we added a line to main(v):

```
System.out.println( myCount );
```

then main( ) could no longer be static because we would have to know which Person's myCount this refers to.

Suppose we changed the program so that myCount was always 23:

```
class Person {
        int myCount;
        …
        public Person() {
                myCount = 23;
                …
        }
        …
        public static void main(String [ ] args) {
                …
                System.out.println(myCount);
        }
}
```

This still wouldn't be valid because Java isn't smart enough to figure out that myCount is always 23. If main( ) refers to myCount than myCount must be static.

A program's main( ) method must be declared to be static. If you forget to declare it as static Eclipse won't recognize it as a true main method.

There is one more attribute that we need. If a variable is declared to be *final* then it can only be assigned to once. This is used to make constants in Java. For example, the following constructs an array of 10 ints:

```
public static final int SIZE = 10;
int [ ] data = new int[SIZE];
```