Understanding Instance and Class Members

When you declare a member variable such as `aFloat` in `MyClass`:

```
class MyClass {
    float aFloat;
}
```

you declare an *instance variable*. Every time you create an instance of a class, the runtime system creates one copy of each the class's instance variables for the instance. You can access an object's instance variables from an object as described in Using Objects.

Instance variables are in contrast to *class variables* (which you declare using the `static` modifier). The runtime system allocates class variables once per class regardless of the number of instances created of that class. The system allocates memory for class variables the first time it encounters the class. All instances share the same copy of the class's class variables. You can access class variables through an instance or through the class itself.

Methods are similar: Your classes can have instance methods and class methods. Instance methods operate on the current object's instance variables but also have access to the class variables. Class methods, on the other hand, cannot access the instance variables declared within the class (unless they create a new object and access them through the object). Also, class methods can be invoked on the class, you don't need an instance to call a class method.

By default, unless otherwise specified, a member declared within a class is an instance member. The class defined below has one instance variable--an integer named `x`--and two instance methods--`x` and `setX`--that let other objects set and query the value of `x`:

```
class AnIntegerNamedX {
    int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Every time you instantiate a new object from a class, you get a new copy of each of the class's instance variables. These copies are associated with the new object. So, every time you instantiate a new `AnIntegerNamedX` object from the class, you get a new copy of x that is associated with the new `AnIntegerNamedX` object.

All instances of a class share the same implementation of an instance method; all instances of `AnIntegerNamedX` share the same implementation of `x` and `setX`. Note that both methods, `x` and `setX`, refer to the object's instance variable `x` by name. "But", you ask, "if all instances of AnIntegerNamedX share the same implementation of `x` and `setX` isn't this ambiguous?" The answer is "no." Within an instance method, the name of an instance variable refers to the current object's

instance variable, assuming that the instance variable isn't hidden by a method parameter. So, within `x` and `setX`, `x` is equivalent to `this.x`.

Objects outside of `AnIntegerNamedX` that wish to access `x` must do so through a particular instance of `AnIntegerNamedX`. Suppose that this code snippet was in another object's method. It creates two different objects of type `AnIntegerNamedX`, sets their `x` values to different values, then displays them:

```
. . .
AnIntegerNamedX myX = new AnIntegerNamedX();
AnIntegerNamedX anotherX = new AnIntegerNamedX();
myX.setX(1);
anotherX.x = 2;
System.out.println("myX.x = " + myX.x());
System.out.println("anotherX.x = " + anotherX.x());
. . .
```

Notice that the code used `setX` to set the `x` value for `myX` but just assigned a value to `anotherX.x` directly. Either way, the code is manipulating two different copies of `x`: the one contained in the `myX` object and the one contained in the `anotherX` object. The output produced by this code snippet is:

```
myX.x = 1
anotherX.x = 2
```

showing that each instance of the class `AnIntegerNamedX` has its own copy of the instance variable `x` and each `x` has a different value.

You can, when declaring a member variable, specify that the variable is a class variable rather than an instance variable. Similarly, you can specify that a method is a class method rather than an instance method. The system creates a single copy of a class variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the class variable. Class methods can only operate on class variables--they cannot access the instance variables defined in the class.

To specify that a member variable is a class variable, use the `static` keyword. For example, let's change the `AnIntegerNamedX` class such that its `x` variable is now a class variable:

```
class AnIntegerNamedX {
    static int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Now the exact same code snippet from before that creates two instances of AnIntegerNamedX, sets their `x` values, and then displays them produces this, different, output.

```
myX.x = 2
```

```
anotherX.x = 2
```

The output is different because `x` is now a class variable so there is only one copy of the variable and it is shared by all instances of `AnIntegerNamedX`, including `myX` and `anotherX`. When you invoke `setX` on either instance, you change the value of `x` for all instances of `AnIntegerNamedX`.

You use class variables for items that you need only one copy of and which must be accessible by all objects inheriting from the class in which the variable is declared. For example, class variables are often used with `final` to define constants; this is more memory efficient than final instance variables because constants can't change, so you really only need one copy).

Similarly, when declaring a method, you can specify that method to be a class method rather than an instance method. Class methods can only operate on class variables and cannot access the instance variables defined in the class.

To specify that a method is a class method, use the `static` keyword in the method declaration. Let's change the `AnIntegerNamedX` class such that its member variable `x` is once again an instance variable, and its two methods are now class methods:

```
class AnIntegerNamedX {
    int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

When you try to compile this version of AnIntegerNamedX, the compiler displays an error like this one:

```
AnIntegerNamedX.java:4: Can't make a static reference to
nonstatic variable x in class AnIntegerNamedX.
        return x;
               ^
```

This is because class methods cannot access instance variables unless the method created an instance of `AnIntegerNamedX` first and accessed the variable through it.

Let's fix `AnIntegerNamedX` by making its `x` variable a class variable:

```
class AnIntegerNamedX {
    static int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
```

```
}
```

Now the class will compile and the same code snippet from before that creates two instances of `AnIntegerNamedX`, sets their `x` values, and then prints the `x` values produces this output:

```
myX.x = 2
anotherX.x = 2
```

Again, changing `x` through `myX` also changes it for other instances of `AnIntegerNamedX`.

Another difference between instance members and class members is that class members are accessible from the class itself. You don't need to instantiate a class to access its class members. Let's rewrite the code snippet from before to access `x` and `setX` directly from the `AnIntegerNamedX` class:

```
. . .
AnIntegerNamedX.setX(1);
System.out.println("AnIntegerNamedX.x = " + AnIntegerNamedX.x());
. . .
```

Notice that you no longer have to create `myX` and `anotherX`. You can set `x` and retrieve `x` directly from the `AnIntegerNamedX` class. You cannot do this with instance members, you can only invoke instance methods from an object and can only access instance variables from an object. You can access class variables and methods either from an instance of the class or from the class itself.

Initializing Instance and Class Members

You can use static initializers and instance initializers to provide initial values for class and instance members when you declare them in a class:

```
class BedAndBreakfast {
    static final int MAX_CAPACITY = 10;
    boolean full = false;
}
```

This works well for members of primitive data type. Sometimes, it even works when creating arrays and objects. But this form of initialization has limitations, as follows:

1. Initializers can perform only initializations that can be expressed in an assignment statement.
2. Initializers cannot call any method that can throw a checked exception.
3. If the initializer calls a method that throws a runtime exception, then it cannot do error recovery.

If you have some initialization to perform that cannot be done in an initializer because of one of these limitations, you have to put the initialization code elsewhere. To initialize class members, put the initialization code in a static initialization block. To initialize instance members, put the initialization code in a constructor.

Using Static Initialization Blocks

Here's an example of a static initialization block:

```
import java.util.ResourceBundle;
class Errors {
    static ResourceBundle errorStrings;
    static {
        try {
            errorStrings = ResourceBundle.
                                getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

The `errorStrings` resource bundle must be initialized in a static initialization block. This is because error recovery must be performed if the bundle cannot be found. Also, `errorStrings` is a class member and it doesn't make sense for it to be initialized in a constructor. As the previous example shows, a static initialization block begins with the `static` keyword and is a normal block of Java code enclosed in curly braces {}.

A class can have any number of static initialization blocks that appear anywhere in the class body. The runtime system guarantees that static initialization blocks and static initializers are called in the order (left-to-right, top-to-bottom) that they appear in the source code.

Initializing Instance Members

If you want to initialize an instance variable and cannot do it in the variable declaration for the reasons cited previously, then put the initialization in the constructor(s) for the class. Suppose the `errorStrings` bundle in the previous example is an instance variable rather than a class variable. Then you'd use the following code to initialize it:

```
import java.util.ResourceBundle;
class Errors {
    ResourceBundle errorStrings;
    Errors() {
        try {
            errorStrings = ResourceBundle.
                                getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

The code that initializes `errorStrings` is now in a constructor for the class.

Sometimes a class contains many constructors and each constructor allows the caller to provide initial values for different instance variables of the new object. For example, `java.awt.Rectangle` has these three constructors:

```
Rectangle();
Rectangle(int width, int height);
Rectangle(int x, int y, int width, int height);
```

The no-argument constructor doesn't let the caller provide initial values for anything, and the other two constructors let the caller set initial values either for the size or for the origin and size. Yet, all of the instance variables, the origin and the size, for `Rectangle` must be initialized. In this case, classes often have one constructor that does all of the work. The other constructors call this constructor and provide it either with the values from their parameters or with default values. For example, here are the possible implementations of the three `Rectangle` constructors shown previously (assume `x`, `y`, `width`, and `height` are the names of the instance variables to be initialized):

```
Rectangle() {
    this(0,0,0,0);
}
Rectangle(int width, int height) {
    this(0,0,width,height);
}
Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

The Java language supports instance initialization blocks, which you could use instead. However, these are intended to be used with anonymous classes, which cannot declare constructors.

The approach described here that uses constructors is better for these reasons:

- All of the initialization code is in one place, thus making the code easier to maintain and read.
- Defaults are handled explicitly.
- Constructors are widely understood by the Java community, including relatively new Java programmers, while instance initializers are not and may cause confusion to others reading your code.