Intro to OOP: Restricting access to object properties

Learn how to limit scope and restrict access to an object's properties. This article uses Java to illustrate these object-oriented techniques to access your data.

By Jeff Hanson

In "Intro to OOP: A closer look at inheritance and polymorphism," we continued to explore the benefits of inheritance and polymorphism. Among other things, we learned:

- While Java supports inheritance from only one superclass, it supports a form of multiple inheritance using interfaces.
- Interfaces really deal with polymorphism to enable us to give multiple personalities to our objects to suit specific needs.
- You can use polymorphism to allow methods with the same name, but different parameter lists to perform slightly different functions.
- Dynamic/Runtime Binding allows an object to be cast at runtime to the object that fits your needs, provided the object implements the desired interface or extends the desired superclass.

Now we are going to talk about the purpose and practicality of restricting access to the properties and operations of our objects in a manner that enforces proper use of multiple interface implementation and superclass extending.

In search of the black box: Encapsulation

A fundamental object-oriented concept is encapsulation—the isolation of data that represents the state of an object from other objects. This is accomplished using a concept generally referred to as scope. Scope refers to the ability of a programming language to enforce rules that limit access to the fields and methods of a class or structure. Most object-oriented languages support some type of scope mechanism, which is usually enforced using special keywords like `public`, `protected`, and `private`.

Java provides four different scope realms: `public`, `package`, `protected`, and `private`. Any class, method, or field can be protected explicitly using the keywords `public`, `protected`, and `private`. Any class, method, or field not qualified with one of these keywords is implicitly given `package` scope. And this brings us to the concept of name spaces in Java.

Name spaces and packages

A name space can be thought of as a group of related names or identifiers within a given context. Name spaces prevent entities with the same name or identifier from existing within the same context. The implication of this is that entities with the same name or identifier can exist as long as they are in different name spaces. Java uses the package concept to implement name spaces and scope control.

A package is a set of classes and interfaces grouped under a common name. Every Java class or interface exists within a name space defined by a package declaration using the `package` keyword. For example, the declaration

```
package com.mycompany.apps.HelloWorld;
```

declares a class or interface named HelloWorld that exists within the `com.mycompany.apps` package. A package declaration is placed at the top of a file containing the definition of a class or interface.

Within the Java community, the current recommendation for package naming is to use the domain name of your company or organization, in reverse, as the first part of your package names. Since domain names are globally unique, using your domain name to name your packages makes your packages globally unique as well.

If a Java class or interface does not include a package declaration, it belongs to the "unnamed package," which is simply a package with no name. The unnamed package should only be used for test programs, code prototypes, etc.

A little privacy, please

In any programming style, especially object-oriented programming, keeping the actual implementation of the exposed programming interfaces hidden is essential. This allows the underlying implementation to change without affecting existing clients of the programming interface and enables objects to take complete responsibility for managing their own state.

The first step in separating the interface and implementation is to hide the internal data of the class. To hide a field or method from all potential clients in Java, declare it as private using the `private` keyword, as in the following example:

```
private int   customerCount;
```

To hide a field or method from all potential clients except descendants of the class that the field or method belongs to, declare it as protected using the `protected` keyword, as in the following example:

```
protected int   customerCount;
```

To hide a field or method from only the potential clients outside the scope of the class that the field or method belongs to, declare it without using any scope keyword, as in the following example:

```
int   customerCount;
```

To expose a field or method to all potential clients of the class that the field or method belongs to, declare it as public using the `public` keyword, as in the following example:

```
public int   customerCount;
```

Just a peek, please

No matter how well-hidden the data of an object might be, access to some of the hidden data is usually still needed by clients. This is accomplished by the use of function calls or methods. In Java, access to hidden data is made possible using special methods known as *property accessors*. There is essentially no difference between a property accessor and a regular method in Java. The only thing that turns an ordinary method into a property accessor is the adherence of the method to certain naming recommendations/patterns.

The pattern for read accessor methods is to name the method the same as the data field, capitalize the first letter, and then prepend the word get or is to the method name. The pattern for write accessor methods is to name the method the same as the data field, capitalize the first letter, and then prepend the word set to the method name. The following examples demonstrate read and write accessor methods.

This is a read accessor method:

```
          public int getCustomerCount()
          {
                return(customerCount);
          }
```

Here’s another example of a read accessor method:

```
          public int isCustomerActive()
          {
                return(customerActive);
          }
```

This is a write accessor method:

```
          public void setCustomerCount(int newValue)
          {
                customerCount = newValue;
          }
```

Using accessor methods allows other objects access to an object's hidden data without having direct access to the data fields. This allows the object that owns the hidden data to do validity checks on the data before changing it and to control whether the data should be changed to the desired new value.

Now let’s apply some of these concepts by modifying our example program, shown in Listing A.

The output from our program when executed should appear as follows:

```
A comforted dog says Bark
A scared dog says Growl
Is a dog carnivorous? true
Is a dog a mammal? true
A comforted cat says Purr
A scared cat says Hiss
Is a cat carnivorous? true
Is a cat a mammal? true
A comforted duck says Quack
A scared duck says Quack, Quack, Quack
Is a duck carnivorous? false
Is a duck a mammal? false
```

Summary

The use of data hiding/encapsulation is a powerful means of maintaining control over an object's data and state. It allows an object to determine if and how data is changed. This allows the implementation of an object change while keeping the exposed interfaces consistent. In our next article, we’ll further explore the scope rules Java provides and begin looking at how Java objects are constructed and initialized.