Intro to OOP: Java scope rules

Scope rules, constructors, object instantiation. These are all important parts of object-oriented programming as you study Java. Learn about these topics in this installment of our series on learning OOP with Java.

By Jeff Hanson

Before delving too deep into scope rules with Java, you should keep in mind the benefits of restricting access to an object's data using encapsulation techniques. In addition, you should be aware that by limiting access to data and methods, you can modify the implementation of our object's infrastructure without changing the public interfaces. This is important because it allows future enhancements to be made with very little impact on existing programs, thus avoiding backward-compatibility problems. If you need a refresher on these topics, I recommend you take a look at "Intro to OOP: Restricting access to object properties."

Here, we will further explore Java's scope features and begin looking at how Java objects are constructed and initialized.

Catch up on past articles covering the transition to OOP with Java
"Transitioning into object-oriented programming using Java"
"Intro to OOP: Understanding classes and objects"
"Intro to OOP: A closer look at inheritance and polymorphism"
"Intro to OOP: Restricting access to object properties"

A deeper look into scope

The scope modifiers provided by Java give us the ability to limit the access to our methods and fields as we see fit. This enables us to encapsulate the data within our classes and to hide the data from outside access. Remember the difference between `public`, `private`, `protected`, and `package` scope. `Public` scope exposes a method or field to all outside access, and `private` scope completely hides a method or field from all outside access. `Protected` scope allows limited access from outside. `Package` scope groups related classes into a package using a kind of reverse domain-naming mechanism. Let's take a look at how we can use packages to encapsulate groups of classes.

Java restricts each source file to one public class. The source file must be named the same as the public class. Thus, the class `HelloWorld` should be placed in the file `HelloWorld.java`. With this in mind, a company named *Acme* might place the `HelloWorld` class in a package named `com.acme.apps`. This would make the full, or qualified, name of the `HelloWorld` class `com.acme.apps.HelloWorld`.

Let's assume that the *Acme Company* wants to create some utility classes for the `HelloWorld` application to use. They might place the utility classes in a package named `com.acme.utils`. Perhaps the *Acme Company* needs some specialized input/output classes. They could place them in a file named `AcmeIO.java` inside the `com.acme.utils` package. The public class inside the `AcmeIO.java` file would have the qualified name `com.acme.utils.AcmeIO`. Now, the `HelloWorld` class could use the `AcmeIO` class to perform specialized input/output. Since the `AcmeIO` class belongs to a different package than `HelloWorld`, the `AcmeIO` class must be "imported" into the `HelloWorld` class before `HelloWorld` can use it. This is accomplished either by referring to the `AcmeIO` class by its qualified name, `com.acme.utils.AcmeIO`, or placing an import statement at the top of the file, as follows:

```
import com.acme.utils.AcmeIO;
```

This import statement imports one class from the `com.acme.utils` package. We could use the import statement to import all classes from the `AcmeIO` package by using a wildcard, as follows:

```
import com.acme.utils.*;
```

The name space created by Java's package and import mechanisms allow the *Acme Company* and any other company to define classes with the same name without running into name conflicts.

Remember that `private` scope restricts outside callers from accessing the data of an object and that `public` scope allows all outside callers access to our data or methods. Let's assume that we want all outside callers to have access to a particular method, but we don't want them to override the method. Java offers the final modifier that enables this very thing. Defining a method or field as `final` restricts a caller of the method or variable from overriding or changing the method or variable, as follows:

```
public final void addCustomer(CustomerData customer)
{
      customers.add(customer);
}
```

The need arises, at times, for a method or field to be offered globally without the presence of an instantiated object. For this, Java offers the `static` modifier. Defining a method or field as `static` makes the method or field globally available to callers without creating an instance of the class. The `Singleton` pattern defines a class that instantiates, at most, one instance. This is a good case for a static or, as it is called, class method, as follows:

```
public class Singleton
{
      protected Singleton()
      {
      }

      private static Singleton _instance = null;
```

```
      public static Singleton getInstance()
      {
            if (null == _instance) {
               _instance = new Singleton();
            }
            return _instance;
      }
}
```

To call a static method or access a static field, you must precede the method or field with the class name, as follows:

```
Singleton   localField = Singleton.getInstance();
```

Construction and initialization

So far, we've talked about objects, classes, operations on objects, properties of objects, and so on. Let's backtrack a little and discuss how Java objects are created and initialized.

In Java, before an object can be used, it must be created, or *instantiated*. The process of object instantiation in Java is handled in two steps. First, the Java Virtual Machine (JVM) allocates the storage area for the object when the new operator is called. This is demonstrated as follows:

```
_instance = new Singleton();
```

Second, the matching constructor is called on the object. For the above example, the constructor with no arguments is called. The code for this constructor is as follows:

```
protected Singleton()
{
}
```

Constructors are special code blocks that look a lot like methods but must follow certain guidelines, among them:

- Constructors can have no return type.
- Constructors cannot be called explicitly.
- Constructors must be named the same as the class they belong to.
- Constructors cannot be overridden.

Constructors do share some similarities with methods, however, including the following:

- Constructors can have `public`, `package`, `protected`, or `private` scope
- Constructors can have an arbitrary list of arguments or no arguments at all

The following examples are all legal constructor definitions for the Singleton class:

```
public Singleton ()
{
}

private Singleton(String name)
{
      _name = name;
}

Singleton(int ID, String name)
{
      _ID = ID;
      _name = name;
}
```

The above examples would be called, respectively, when the following code is executed:

```
Singleton   firstObj = new Singleton();
Singleton   secondObj = new Singleton("secondObj");
Singleton   thirdObj = new Singleton(45, "thirdObj");
```

Moving on

This article has been our look at how to use Java scope rules to limit access to data and methods. We also took a look at object instantiation and constructors. In our next article, we'll talk about the intricacies and dangers of constructing and initializing Java objects.