Access and Non-Access Modifiers

Introduction

In this article from my free Java 8 Course, I will be discussing visibility modifiers. There are four access modifiers that exist in Java, three of which I will cover in this article: `public`, `private`, and *default*. The fourth modifier is `protected`, which is related to a more advanced topic (inheritance), so I will skip it for now. There are also many non-access modifiers. In this article, I will only be focusing on the `static` modifier.

The Public Access Modifier

The public modifier signifies that a method, variable or class is accessible from ***any*** other class. For example, the `Person` class that we've used in previous examples can access and use the `Name` class because the `Name` class is `public`. Please note that I've placed the `Name` class in a separate `package` for demonstration purposes only.

```java
package com.marcusbiel.java8course;

import com.marcusbiel.java8course.attributes.Name;

public class Person {
     private Name personName;
}
```

Example 1

The Private Access Modifier

The `private` modifier signifies that the method or variable is only accessible from the class where it was declared. Make all variables and methods `private` until you absolutely need to make them `public`. This is very important, especially for variables. You don't want other classes to poke around in your `Person` class and have them changing your `Person` object's name whenever they feel like it. Generally, if you want to expose some variables to outside classes, you should not make the variables `public`. This allows outside classes to not only see them, but also modify them without any restrictions.

A Data Centric Approach

A commonly used alternative to making all your attributes `public`, is to provide so-called `public` "getter and setter" methods, that allow other objects to directly access and change your private attributes. This approach is taught as "object oriented programming" by many Java books and courses. To me, this always felt like having a locked door, with a key and a note saying, "Please don't open this door" stuck on it. It took me many years to realize that I wasn't the only one who was confused, but that actually all those "smart books" and teachers were wrong! **Don't let them fool you!** It is fallacy to believe that you can effectively encapsulate a class while still providing `public` methods that allow one to ***directly*** operate on its internals. (For more details, read my article *Getters and Setters are Evil*). Avoid this data-centric approach.

An Object Oriented Approach

Instead, use an object-oriented approach. Focus on providing functionality from a business point of view, independent of the internal details of a class.

When designing your class, put yourself in the shoes of someone who will have to use your class. Make it as simple as possible for the caller to use the class's methods. *To achieve this, focus on what the class should do, and not on how this will be achieved.*

After careful consideration, offer only a small set of well defined public methods, independent of the internal details of a class. **Generally, less is more.**

The less the client "knows", the more flexible your code stays - every method that is not `public` can easily be changed, without affecting other code.

As an analogy, a house also has a well defined number of doors, and they are usually closed. The house owner decides **if**, **when,** and **how** he wants to open them. For example, he's not going to open the safe door when a delivery person comes to the door, but he might open the front door for the person, so they can carry his package inside.

See every `public` method as an open safe door; as a potential **threat** to your class.

Finally, you should also always validate incoming arguments. If the package the delivery person brought was supposed to be a new book, but it was ticking, the house owner probably wouldn't let it come inside. The same goes for your `public` methods. As a Software Craftsman, you must make sure that each class doesn't cause harm to the system, even when it's used beyond its intended purpose.

I will continue to talk about these object-oriented principles throughout the course as they are very important to good code design.

The Package-Private Modifier

The third modifier I'm going to discuss is the "package-private" modifier. It is also called the "default" modifier, *because this modifier is never declared*. Instead, it is used as a *fallback* when no other visibility modifiers are declared.

A class, method or variable with this visibility is accessible from the `package` in which it is declared, **but from nowhere else**. In other words, for classes that reside in the same `package` as a package-private class, it's as though the class is `public`; however, for classes belonging to other packages, a package-private class acts as though it was a `private` class.

```
package com.marcusbiel.java8course.attributes;

class Name {
      /*
       * Package-Private Modifier for Class Name
       */
}
```
Example 2

This modifier is only sparsely used by Java developers, *without good reason*. Generally speaking, use the default level modifier whenever you need classes of the same `package` to use a method, but you don't want classes outside of this `package` to use the method. For example, imagine a class `Car` has a method `diagnose` that you want a class `Mechanic` of the same `package` to be able to use. But the sales-oriented car company you are coding for doesn't want class `Customer` to fiddle around with this method, because that would hurt its earnings.

In my opinion there is a flaw in the package-private modifier: Since there is no keyword, it is unclear whether a missing modifier is an error on the part of the programmer, or a *planned* package-private modifier. If you forget to put a modifier in, you are going to cause issues for your program down the road, but you won't know. There will be no warning from the compiler that there's a "missing modifier", since it is legal coding practice to leave it out. If you had wanted your class or members to be public, when you try to access them outside of the package you can't. Even more dangerous, is when you've forgotten to set a `private` modifier and months later, your method or variable is used somewhere else, without your noticing.

On the other hand, if you intended to use the package-private modifier, you're intentionally leaving the visibility modifier out. Another programmer might not realize this and try to "fix" your code by adding in a modifier that they assume you wanted. That's why I recommend that if you are *on purpose* using the package-private modifier (which in some cases is **very useful**), then leave a comment denoting your intent.

Coding Example

Now that you know about visibility modifiers, let's apply them to a coding example. First, we are going to create a new `@Test` method in our `PersonTest` class. This method will be called `shouldReturnNumberOfPersons` and will contain three objects of type Person named "person1", "person2", and "person3".

```
package com.marcusbiel.java8course;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

     @Test
     public void shouldReturnNumberOfPersons {
          Person person1 = new Person();
          Person person2 = new Person();
          Person myPerson = new Person();
          assertEquals(3, myPerson.numberOfPersons());
     }
}
```

Example 3

Next we're going to use an `assertEquals()` method to check if the number of `Person` objects created is equal to 3.

Let's begin to write the code to make this method work in our `Person` class. In our `Person` class, I've created an instance variable of type int called `personCounter`. Then, in the default constructor, we will add `1` to `personCounter` each time this constructor is called. Logically, every time a `Person` is created this constructor is going to be called, so if we add `1` to `personCounter` each time the constructor is called, we count the number of `Person` objects we have created. We never initialized `personCounter`, but this should still work because the default value for an `int` is `0`. (If you'd like to learn more about default values, you can take a look at this article).

```
public class Person {
      private int personCounter;

      public Person() {
            personCounter = personCounter + 1;
      }
}
```

Example 4

As an added note, there are actually three ways to add 1 to `personCounter`. The first is the way we did above. The second is:

```
personCounter += 1;
```

Example 5

which can increment `personCounter` by any value we wish. The third option is the shortest, but only works if you want to increment by `1`:

```
personCounter++;
```

Example 6

All three of these options take the value of `personCounter`, increase it by `1`, and then store that new value in a new version of `personCounter`. Now let's write our `numberOfPersons()` method to return `personCounter`:

```
public static int numberOfPersons() {
     return personCounter;
}
```

Example 7

If we execute this code, our test fails because our `numberOfPersons()` method returned 1. Can you guess why?

Each time we created a new `Person` object, we stored the object and all its values into a separate `person` variable. Therefore, each time we create a new object, all of its instance variables are reset by the constructor and stored as part of a new object. So for each of our `Person` objects, the value of `personCounter` got initialized to `0`, and then incremented by `1`.

The Static Modifier

This brings us to our solution, the static modifier. As you might remember from the last article, the static modifier associates the method or variable with the class as a whole instead of with each individual object. Normally if you create a hundred Person objects, each will have its own personCounter variable, but with this modifier, all one hundred objects will share one common personCounter variable. This way, our personCounter will retain the same value, no matter how many Person objects we create.

First, we add this modifier to our personCounter variable and we're also going to add it to our numberOfPersons() method, as we should never have an instance method return a static variable and vice versa.

```java
public class Person {
     private Name personName;
     private static int personCounter;

     [...]

     public static int numberOfPersons() {
          return personCounter;
     }
}
```

Example 8

By making the method and variable static, we can now accurately count and return the number of Person objects created. Our variable is associated with the class, and since it can't be accessed due to the private tag, we have the public method numberOfPersons() which allows outside code to access, but not modify, the value of personCounter.

Thanks for reading!