The Object.finalize() Method

Introduction

In this article from my free Java 8 course, I will be discussing the `Object finalize()` method in Java. The class `Object`, which is the superclass of all classes, defines the `finalize()` method as well as other methods including clone, toString, hashCode and equals.

What is Finalize?

`finalize()` is a hook method of the class `Object`.

A hook method is an empty method in a base class that you can override to provide functionality that will be called in specific situations by the existing process. In other words you have a mechanism to "hook" into the existing process and extend its functionality. This concept exists independently of any specific programming language. Using inheritance, this can be realized in Java by providing an empty method in a parent class. By overwriting this method in a child class, the code provided will automatically be called by the surrounding logic / process.

According to the Java 8 documentation, it is called by the **Garbage Collector** when there are no more references to the object. The usual purpose of this method is to perform cleanup actions just before the object is irrevocably discarded.

The Garbage Collector

The garbage collector, as you can imagine, collects the 'garbage' in your program. The garbage that's being cleaned up are the objects that you've created, processed on, and then later put aside when you no longer needed them. The garbage collector's main responsibility is to free up memory resources, so that your program hopefully never runs out of memory. However, the garbage collector runs asynchronously, and we have no control or influence over if and when it will be running. We can give it recommendations, but they are just that - recommendations. The garbage collector is not bound by them.

A computer uses several types of hardware resources, and those are physically limited by nature. Therefore, we have to use those resources wisely. In Java, unlike in older programming languages like C++, memory is automatically managed by the garbage collector; however, there are other resources, usually "IO resources", that we need to manage by ourselves.

The "I" of "IO" stands for input, and the "O" stands for output. Therefore, IO is a general term for **i**nput or output functionality (also referred to as communication) from a resource such as a file, an external system, or a database. Input is **read into** our system, while output is **written from** our system. A typical name for a class or interface that is used to read input is "Reader", while a typical name for a class or reader that is used to write output is "Writer".

Those resources must be requested when needed, and released when not needed anymore. As this is rather tedious, the idea was to automatically release them just before the objects

that are using them are discarded, and hence, the finalize() method was born. However, as there is no guarantee that the finalize() method is ever called, there is no guarantee that those resources are ever released either, and therefore, the finalize() method is useless. If you're interested in learning more about this you could read Effective Java by Joshua Bloch, where he's done a lot of interesting research on the subject. For example, he's tested and found that creating an object and destroying it using an overridden `finalize()` method is 430 times slower than using the inherited method.

Overriding finalize()

In order to have the `finalize()` method run, let's implement a `Porsche` class, in which we will override it. To override the method, we must make sure that the name and signature of the overridden method are the exactly same in both the superclass and subclass. The only change I'm going to make, and this is allowed, is that I'm going to make it `public`, because `public` is more visible than `protected`. This allows the method to be called outside of our `Porsche` class.

```
public class Porsche {

     public void finalize(){

     }

}
```

Example 1

The @Override Annotation

The @Override annotation helps us be sure that we are overriding the method and not just writing a new method accidentally, as you can only override a method if the method signature is the same as in the superclass (a method's signature is its name as well as the number and type of its parameters). If we add an argument to our `finalize()` method, this is no longer overriding, but it is overloading the method. (It's merely a method that has the same name, but with different parameters), and it will give us a compile time error.

This helps us in identifying immediately at compile time, while still in our IDE, a problem that could result in unexpected results at runtime (such as our "overridden" function not being called). That's why whenever you override a method, you should always add an `@Override` annotation.

```
public class Porsche {

     IOReader ioReader = new IOReader();

     @Override
     public void finalize(){
          ioReader.close();
     }
}
```

Example 2

Inside our finalize method, I've written code that calls the `close()` method on all the objects that an object of class `Porsche` would create. Let's say, for instance, that we have an `IOReader` which is reading a character stream from a file. Once we are finished, we would like to close this reader by calling the `close()` method provided by `IOReader`. Now we have our overwritten `finalize()` method.

However, there are problems with our implementation. First, as already mentioned, we do not have any guarantee that this method will ever be called. It is totally under the JVM's control, and outside of our influence. The second problem is that if within this code we have any exception that is not handled, the process will stop and the objects will remain in a weird "zombie" state, which slows down garbage collection.

The Alternative to Finalize

The recommended alternative would be to create our own `close()` method which cleans up/closes all the resources no longer in use, in this case `IoReader`. This way we have more control over our resources, and aren't depending on the garbage collector.

```
public class Porsche {

     IOReader ioReader = new IOReader();

     public void close(){
          ioReader.close();
     }

}
```

Example 3

Let's also write a draft of closing an object in the `CarSelector` class. First, let's create the `CarSelector` class and add a Porsche object with the following line: `Porsche porsche = new Porsche();` and use our new `close()` method. We'll surround this in a try, catch, finally block and use our finally block to clean up our Porsche. The critical point here is that you have a finally section at the end, because the cool thing about finally is that, even if an exception occurs, it is guaranteed to always be executed. Which is exactly what we need to solve our problem and make sure all non-memory resources are always freed. Here is how you can properly do it in Java:

```
package com.marcusbiel.java8course;

public class CarSelector {

     public static void main(String[] arguments) {
          Porsche porsche = new Porsche();

          try {
               // some code
          } finally {
               porsche.close();
          }
     }
}
```

Example 4

This way we guarantee that once we are done with our `porsche` object, we will close all of the affiliated resources involved, without overwriting `finalize()`. The finally block does exactly what we wanted our `finalize()` method to do.

The `finalize()` method is extremely flawed. The garbage collector has a mind of its own and we can't truly control when and how it operates. Since overriding the `finalize()` method doesn't effectively solve our problem, you can instead use a try-(catch)-finally block to close any extra resources when you're done with an object. Since the close method is our own method, unaffiliated with the garbage collector, it works exactly as intended, every time.

Thanks for reading!