Inheritance and Polymorphism in Java

Introduction

In this article from my free Java 8 course, I will be discussing inheritance in Java. Similar to interfaces, inheritance allows a programmer to handle a group of similar objects in a uniform way which minimizes code duplication. However, if inheritance is not utilized properly and at the right time, it can make the code unmaintainable or even cause bugs in the long run.

A Word of Warning

Inheritance is one of the most powerful features of object-oriented languages. It sounds very promising and useful when first described. However, I think it is just a bit too powerful. If used incorrectly, it can damage your code base—leaving it very vulnerable to bad design, so take care to use it judiciously. There definitely are cases where the use of inheritance is justified, but only a few. It requires some experience on the part of the programmer to wield inheritance properly.

What is Polymorphism?

Polymorphism is a concept deeply related to interfaces and inheritance. The term “polymorphism” origins from the Greek roots “poly morphs” - *many forms*. Polymorphism allows us create different objects on the right side of a variable declaration, but assign them all to a uniform object type on the left side. Let’s take a look at the coding example below:

```
package com.marcusbiel.java8course;

import org.junit.Test;

public class ZooTest {

     @Test
     public void shouldFeedAllAnimals() {
          Zoo zoo = new Zoo();
          Animal[] animals = {new Dog(), new Gorilla(), new Tiger(), };
          zoo.feed(animals);
     }
}
```

Example 1

We are not actually testing something here; this is only a demonstration. This example is actually taken from the book *Head First Java* by Kathy Sierra. She's my favorite author for any Java-related book. I highly recommend that you read *Head First Java* and any of her other books as they're extremely creative and fun to read on top of being very informative.

Let's focus on our array of `Animal` objects. Our array has a `Dog`, a `Gorilla` and a `Tiger`. The usage here is indicative of polymorphism. We're using a reference variable of our supertype, or parent class, `Animal` on the left side of our statement and on the right side the instances are any subclass or subtype of `Animal`: `Dog`, `Gorilla`, and `Tiger` in our example. These different animals can be assigned to indices of `animals` because polymorphism means that we can have different implementations on the right side that are handled uniformly on the left.

If you look in our `Zoo` class, you can see the feed method below that applies the same code to any `Animal`, no matter whether it's a `Dog` or a `Tiger`.

```java
package com.marcusbiel.java8course;

public class ZooTest {

     public void feed(Animal[] animals){
          for (Animal animal : animals){
               animal.eat();
               animal.grow();
          }
     }
}
```
Example 2

We are iterating through the array of animals with an arbitrary example where each `animal` eats and grows. All of these animals probably eat in a different way, but as we can see, they are told to eat and grow no matter what. For example, a gorilla would eat leaves and insects while a tiger would eat meat. The idea is for each `Animal` subtype to have its own concrete implementation, its own way of eating as specified by its `eat()` method.
To reinforce the idea, we also have the `grow()` method. My idea is for `grow()` to be a concrete method, while `eat()` is an abstract method - with both methods residing in the `Animal` class. This would mean that all animals grow at the same rate while eating differently.

Inheritance

Let's jump to our first iteration of the `Animal` class:

```
package com.marcusbiel.java8course;

public class Animal {

     public void eat() {
          System.out.println(“Animal is eating”);
     }
     public void grow() {
          System.out.println(“Animal is growing”);
     }
}
```

Example 3

As you can see, our `Animal` class has two concrete methods: `eat()` and `grow()`. Later on we will make `grow()` abstract. We have both methods print out to the console, something we should generally avoid. However, for our example where we're demonstrating something, this would be an exception to the rule, but avoid doing it in real code. As you can see, this class looks very much like any regular class you have seen so many times before.

Now, let's take a look at the `Dog` class. Say we want to declare `Dog` as an `Animal`. If `Animal` were an `interface`, we would write `implements`. Since it's a `class`, we express its subclass relationship with the `extends` keyword instead. Quite simply, we say `Dog extends Animal` and voilà, we've just used inheritance.

```
package com.marcusbiel.java8course;

public class Dog extends Animal {

}
```

Example 4

Now it might look like the `Dog` class is empty but in fact we can already use `eat()` and `grow()` with `Dog`. We can also create the `Gorilla` and `Tiger` classes in the same way, but I won't show them here. Instead, let's move on and mix implementing interfaces and extending classes by implementing two interfaces: `Loggable` and `Printable`. The question now becomes, which comes first: `extends` or `implements`? There is a syntax rule stating that `extends` should come first followed by `implements` because of Java's single inheritance model. We can only extend a single class but we can implement any number of interfaces. We reflect this in a change to the declaration of the `Dog` class:

```
package com.marcusbiel.java8course;

public class Dog extends Animal implements Loggable, Printable{

}
```
Example 5

Note the comma-separated list of interfaces, namely `Loggable` and `Printable` which `Dog` declares as its interfaces. These interfaces tell us that our `Dog` class can play the role of a `Loggable`. For example, we can print and log our `Dog` object since it can play the role of a `Loggable`. You can learn more about this in my article about Interfaces in Java.

We write the same method signatures for `print()` and `message()` as declared in `Loggable` and `Printable`, respectively. We do it this way because there is a rule stating that when we implement the abstract methods of an `interface` or a `class`, the method signatures and return type must be exactly the same. We can't just write `public void message()`, for example, as `Loggable` explicitly requires a `String` return type.

```
package com.marcusbiel.java8course;

public class Dog extends Animal implements Loggable, Printable{
     public void print() {
          System.out.println("printing...");
     }
     public String message(){
          return "something";
     }
}
```
Example 6

Right now, only our `Dog` class implements stub implementations of `Loggable` and `Printable`. This means that it has basic general functionality that satisfies the interfaces, however it probably isn't the way we want to leave the code. If we wanted all animals to implement `Loggable` and `Printable`, we could implement these interfaces in the `Animal` class (and remove them in `Dog` to avoid duplicate code).

Multiple Inheritance

Although we can implement several interfaces, in Java we can only extend up to one class, so we can't say something like Pig extends Herbivore, Carnivore. Note that this works in C++ and can be even nastier than single inheritance. The use of multiple inheritance can lead to something called the **Deadly Diamond of Death**.

Abstract Classes

An abstract class is like a hybrid between an interface and a class. Simply put, we are allowed to create both abstract and non-abstract methods in an abstract class. Recall that for interfaces, we don't have to explicitly declare our methods `abstract` since all methods are `abstract` by default. Because an `abstract` class can mix both abstract and non `abstract` methods, however, we have to explicitly use the `abstract` modifier for `abstract` methods. Please note that, just like with interfaces, you cannot create an instance of an abstract class.

An `abstract` class can extend another `abstract` class, which means it can add further methods (both `abstract` and non-abstract), as well as implement or overwrite (`abstract`) methods of its superclass. The first concrete class in a hierarchy of classes extending one another must ensure that all `abstract` methods of the hierarchy are implemented.

Let's make `eat() abstract` in `Animal`. Since we know that each of our subclasses will implement this method differently, there's no reason to provide a concrete implementation of this method in the `Animal` class.

```
public abstract void eat();
```

Example 7

However, declaring `Animal abstract` changes its meaning and behavior. Since `abstract` classes cannot be instantiated, the compiler will no longer allow us to instantiate an `Animal` object with `new Animal()`. This is what our `Animal` class looks like now:

```
package com.marcusbiel.java8course;

public abstract class Animal {

     public abstract void eat();

     public void grow() {
          System.out.println("Animal is growing");
     }
}
```

Example 8

When Should I Use Abstract?

Firstly, I'd like to note that there is no "golden rule of abstract". Everything with inheritance is extremely case by case, and even modifying your program could change whether or not abstract should be used.

You should make a parent class `abstract` when you don't intend for it to be created. What is an animal? In this case you don't intend to create an `Animal` object; you're only going to be creating tangible animals like tigers, dogs and gorillas. That's when you should make a class abstract. When you only intend to use its children in your program.

You should make your methods abstract when you intend every child to have a different implementation. If you know that there's no 'default' eating behavior for an `Animal`, there's no reason to write one that is going to be overridden by every single class. If some or all of your children will implement a method in the same way, you may want to write that method and override it when necessary. This after all is why inheritance is useful in the first place.

Implementing an Abstract Method

Going back to our classes, we still have a few more bumps to iron out. Now that Animal's `eat()` method is abstract, we are forced to implement the abstract `eat` method for the other subclasses. Let's start with Tiger:

```java
package com.marcusbiel.java8course;

public class Tiger extends Animal{
     public void eat() {
          System.out.println("Tiger is eating...");
     }
}
```

Example 9

Here, we have an example of a subclass' implementation of `eat()`. Each `Animal` subclass would have its own specific logic for that method. A real world implementation would probably not use a print line statement for implementing `eat()`, however, it's good enough for our demonstration.

All of the logic that is common for all `Animal` subclasses, like `grow()`, belongs in the `Animal` class. It's a double-edged sword, though. On one hand, it is very handy for code reuse, but on the other hand it makes our program more complicated to understand.

For example, someone reading the code can't tell for sure whether a `Gorilla` ages and eats with `Animal.grow()` and `Animal.eat()` or if it has its own overriding `Gorilla.grow()` and `Gorilla.eat()` methods like the ones below.

```
public class Gorilla extends Animal {

    @Override
    public void grow() {
        System.out.println("Gorilla is implementing the age by
                itself");
    }

    @Override
    public void eat() {
        System.out.println("Gorilla is eating...");
    }
}
```
Example 10

This is similar to how you can just define any arbitrary exception to some rule. You could say, "Well, my `Gorilla` doesn't use the `Animal.grow()` method but instead replaces it with its own." But you want to make sure you use this tool effectively. If your intent is to modify the functionality of a method that was written in `Animal`, you can override it. When overriding a method, it must have the same method signature as the parent's method. To do this, you can't change the method name or the parameter types as they form part of the method signature. Typically when overriding a method, you should use the `@Override` annotation which ensures that all these rules are followed by giving a warning when you aren't actually overriding a method.

If you are intending to have a new method of the same name, but with different parameters, you are not overriding, but "overloading" the method. For example we could have our `grow()` method that increases an animal's size based on a default value and a `grow(int amount)` method that grows by `amount`. Two methods with the same name can exist as long as they have different parameters.

Changing Visibility

There is yet another dimension to inheritance, and for this we have to introduce a new visibility modifier. The `protected` modifier gives visibility of a member to any class in the same package, as well as to any subclasses. `protected` is different from the package private modifier in that the `protected` modifier allows for a method to be accessed outside the package through inheritance. In this case let's apply it to `grow()`.

```
package com.marcusbiel.java8course;

public abstract class Animal{
     public abstract void eat();

     protected void grow() {
          System.out.println(“Animal is growing”);
     }
}
```

Example 11

In our example, the `protected` modifier implies that `Animal.grow()` is visible to anything in the `com.marcusbiel.java8course` package, as well as to any Animal subclass outside of that package. Both properties of `protected` make using it complex, so much so that we will be going into even more depth to understand why we should avoid using it. Among the visibility modifiers in Java, only public has more access privileges than protected.

When you override a method in a Java class, you **are** allowed to change its visibility. However, you can only increase the visibility of members in the child classes. This means that in this case, the only modifiers we can use for `grow()` in `Animal` subclasses are `protected` and `public`. In any `Animal` subclass, trying to make `age() private` will result in a compiler error which says, “Attempting to assign weaker access privileges.” In the end, our `abstract class Animal` defines a contract or a protocol, exactly like with interfaces. You are in fact promising any client class in the Java universe that the client can in fact use any `Animal` instances with all the `public` methods they provide. Just imagine how messy it would be if it were optional for a class and its subclasses to fulfill the visibility contract. You would have a practically broken inheritance model!

When we inherit the members of a class, we inherit both instance methods and instance variables. This means that if `Animal` had a `protected double weight` variable, all its subclasses would inherit it. Say for instance that when our `Tiger` grows, it adds a kilogram to its weight. It would look something like our example below:

```
public class Tiger extends Animal{

     public void grow() {
          super.grow();
          weight += 1.0;
     }
}
```

Example 12

One of the issues with `protected` it that it has incredibly complicated rules that make it very easy to mess up while programming. It also violates **encapsulation** because Animal should be the only one with access to its internal variables. Even though it’s possible, I wouldn’t make use of `protected`.

Now that we’ve completed our classes, we are finally in a position where we can run our ZooTest. When we run it, each Animal subclass implements its own `eat()` method and if it lacks its own `grow()` behavior it uses the one specified in Animal.

Weaknesses in Inheritance

Inheritance can easily become extremely confusing and difficult to use. For example, say you have five classes inheriting from one another in a hierarchy. To understand the behavior of an object originating from this class hierarchy, you will have to understand the entire network of classes as well as its interwoven dependencies. Every single method could be overwritten in any or all of the classes in the hierarchy. Even worse, a method of a child class can call all non-private methods of its parent class by putting a preceding “super” before the method name of the parent class. This allows the behavior of a (supposedly simple) method to turn into a very tight sticky mush, and turn basic inheritance into an intransparent network of classes, highly dependent on one another. But what if your client only needs you to create a few classes? Well, even that could end up not working out.

Let me illustrate this briefly: Imagine you have an `abstract Duck` class with several variants to implement, like `MallardDuck` and `RedHeadDuck`. Both are Ducks, both `fly()` and `swim()` in the same way, so this could be a justified reason to use inheritance. We provide a default implementation of `fly()` and `swim()` in the `Duck` class for both `MallardDuck` and `RedHeadDuck`, and an abstract method `quack()` that both implement in their own unique way.

Months later, we extend our system adding a `RubberDuck` and a `DecoyDuck`. The abstract method `quack()` has to be implemented for both, however, `RubberDuck` will internally `squeak()` instead. `DecoyDuck` neither quacks nor squeaks, so we provide an empty implementation `quack()` in `DecoyDuck`. We also don’t want our `RubberDuck` and `DecoyDuck` to fly. All of these changes are implementable, but our design that once shone with beauty and flexibility is starting to be more of a pain then a help.

There is a better solution. We could use a more flexible approach, known as the *Strategy Pattern*! To start, you replace abstract class `Duck` with an `interface Duck`. Since it’s an `interface` it can only contain abstract methods for `fly()`, `swim()` and `quack()`. Second, you define concrete classes for `MallardDuck`, `RedHeadDuck`, `RubberDuck` and `DecoyDuck`, each implementing the interface `Duck`.

Now we define three more interfaces: `FlyingBehavior`, `SwimmingBehavior`, and `QuackingBehavior` - and provide various implementations for each. For example `FlyingWithWings`, `Quacking` and `Squeaking`, and so on. As `RubberDuck` and `DecoyDuck` both don't `fly()` and `DecoyDuck` doesn't even `quack()` we also have to provide a `NotFlying` and `NotQuacking` implementation. Still not very nice, but much nicer than using inheritance. Now, all ducks won't have a default implementation.

Using the strategy pattern, it will never happen that a `Duck` is suddenly flying on production, when it should not. Things are more clear. On top of this, this design is way more flexible - you can exchange the behavior at runtime.

```
if(summer) {
      flyBehavior = new FlyingWithFastWings();
} else {
      flyBehavior = new FlyingWithWings();
}
```

Example 13

In conclusion, there are sometimes justified cases of inheritance, but later the requirements change, and using inheritance can quickly become a maintenance nightmare. On the other hand, even in cases where using inheritance is advised, it usually doesn't hurt to still decide against its usage. I wanted to illustrate for you how problematic inheritance is, although it may not seem so at first. I also wanted to show that there are great alternatives to using it. The above example of a strategy pattern was taken from *Head First Design Patterns*, a book I highly recommend that you read.

As a rule of thumb, don't use inheritance. For the last three years of Java development, I haven't used inheritance a single time. Instead you should utilize interfaces, as I have been doing in my work, whenever possible.

Thanks for reading!