An introduction to writing classes in java.

What is a class

Classes are the building blocks of our code in object oriented languages because our programs are written as systems of classes. A class is essentially a blueprint for creating similar objects. For example you could have a class "fruit" and then have instances of that class to represent "apple" and "orange".

Classes define data

Classes let us define complex datatypes to represent things in our program, one part of how we do this is through attributes. Attributes are like internal variables which let us store values. Objects are *stateful* meaning that the values they contain can be modified without creating an entire new object. An example of this would be if you had a "car" class, you could have attributes like "speed" and "make" and "model"

Classes define behavior

Classes do not just define data, unlike primitive types that are simple datapoints, classes encapsulate data and behavior. This means objects can have data, and also do things with that data internally. When writing a class we write these as methods; they are essentially functions that let us use or modify the internal state of an object. So if you had a "car" class you could have methods like "accelerate" and "brake" to represent things that the car can do.

The syntax of classes

Classes are defined using the class keyword, attributes and methods can be defined in any order but it is standard to

```
class Example {
    // attributes
    int x = 3;
    int y = 2;

    // methods
    public int sum() {
        return x+y;
    }

    public int difference() {
        return x-y;
    }
}
```

An Example

Let's create a simple example project based on the car example from earlier, you can play with the code snippets below and try to get the example working.

trinket: run code anywhere

Java in the browser. No installation required.

trinket.io

Attributes of Classes

Attributes are an important part of classes because they define the data we work upon, while attributes are very basic and similar to variables, they have some important differences which should be noted. Attributes must have defined types, but it can be a generic type, they are always accessible everywhere within the class they are defined, but the behavior can be modified using many modifier keywords.

Public and Private Attributes

The two most common attribute modifiers are public and private. These concepts are very simple, but very important. With a public attribute, the value is accessible to any external object, for example if I have an object with an attribute “a” any other object could read or change the value of “a” This is often not desirable because we want to encapsulate our state away from the rest of the program. For this reason most attributes in java should be private, and we usually write methods to read and change those attributes as we need to. This prevents unexpected behavior which can compromise security or cause unwanted effects.

Static Attributes

While regular attributes belong to the objects which are created from the class, static attributes belong to the class itself. This way you can have values that synchronize across all objects of the same type. For example if you wanted to give each object of a certain type a unique ID, you could have a static attribute for ID, and then use that and increment it when you create an object of that type. This way each object would have a single unique ID with no gaps or overlap.

Final Attributes

Final attributes are Java’s version of an immutable attribute. While normal attributes can be assigned an arbitrary number of times, final attributes can only be assigned once over the lifetime of the object. For example if you wanted a unique ID, you would want that to be a final variable since the ID of that object should likely never change. It is important to mention that final variables can still point to mutable objects, so for example if you have a final ArrayList, the values within that List can still be changed, you just can’t assign it to an entirely new object. (I tend to write all my attributes as final by default to prevent unwanted behavior and because of my rust background, but this is not universally accepted across the java community.)

The Syntax of Attributes

```
public class Example {
    // this variable can be accessed by the outs
    public int a = 5;

    // this variable can never be reassigned
    private final String b = "Something";

    // this attribute is the same across all obj
    private static double c = 19.2;

    // modifiers can be combined to enrich behav
    // except private and public which are mutua
    public static final boolean d = true;
}
```

Example

Lets refine that example earlier by adding

Methods

Methods in Java are what make classes especially useful, in other languages like C and Go we have complex types called structs, but they are only datatypes. By defining methods we are able to add built in behavior to our objects, allowing them to act on the data we defined. Methods allow a few different things, like parameters and return values, which allow us to extend them to be extremely flexible. Like attributes, methods can be public, private, or static. There are also special methods called constructors, which are used to run code when an object is initialized from the class.

Method Parameters

Sometimes we want to use data other than our object attributes in our methods, we accomplish this by using method parameters, parameters allow us to pass in a value to be temporarily used by the method, this is done with the following syntax:

```
class MyClass {
    // this is the syntax for a method
    public void doSomething(String word) {
        // do something with the word parameter
        System.out.println(word);
    }
}

// the parameter is passed into the parameters
new MyClass().doSomething("cat");
```

Method Return Values

Methods can also return values when they are called, this allows us to get information from the object. Return values must be specified in the method declaration, and if the Return value is not "void" then all possible execution paths must return a value, even if it is Null. You can see the syntax of method returns here:

```
class MyClass {
    // the int type in front of the method name
    // if this is void there is no return value
    public int getSomething() {
        return 5; // the return statement return
    }
}

// we can get the value by calling the method on
int myInt = new MyClass().getSomething();
```

Accessors and Mutators

Because attributes are usually private, we need some way to still work with them, this is done with methods, called accessors and mutators, an accessor is a method which gets some information from inside the class and gives it to the caller, and a mutator is a method which changes something about the internal state of the object. The two most simple kinds of accessors and mutators are called getters and setters, where a getter simply fetches an attribute value and returns it to the caller, and a setter takes a value as a parameter and sets an internal attribute to that value. Here is an example of getters and setters:

```
class Example {
    // this private attribute cannot be accessed
    private int a = 5;

    // this is a getter accessor
    public int getA() {
        return a;
    }

    // this is a setter mutator
    public void setA(int newValue) {
        a = newValue
    }
}
```

Constructor Methods

Constructors are methods which are run when a new object is created. These methods are commonly used to set up internal state for the object so you don't have to call a bunch of setter methods right after creating an object. One common issue with constructors and setters is we want to use the same name for our parameter as we have for our attribute. We can solve this using the "this" keyword, "this" refers to the current object always, so we can still access those values even if the attribute name has been temporarily overwritten by a parameter. The syntax is as follows:

```
class Point {
    private int x;
    private int y;

    // this is a constructor method
    // it sets the values of the private attribu
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

// constructor parameters are passed like this
Point myPoint = new Point(3, 7);
```

Static Methods

Static Methods are similar to static attributes in the sense that they belong to the class and not the object. Static methods are behaviors that are related to the function of that type, but don't have to do with the internal state of any specific object. For example if we have a class "Car" we could have a static method "mphToKph" which converts miles per hour to kilometers per hour, this would be a good use case for a static method because it is related to the "Car" class without necessarily needing to know about the state of any given car object. Here is an example:

```
class Car {
    // this is a static method
    public static int mphToKph(int mph) {
        // implementation can only access parame
        return mph * 1.609;
    }
}
```

```
// we call the static method with the class name
int kph = Car.mphToKph(25);
```

Method Overloading

Method overloading is a common pattern to introduce static polymorphism into our java code. When we use method overloading, we assign different behaviors to methods depending on their behavior. This is a very common pattern for constructors because we may want to create objects in different ways based on what we provide them. It is however not exclusive to constructors and can be used with any kind of method. Here is an example:

```
class Car {
    private int speed = 0;

    // default behavior
    public void accelerate() {
        speed += 5;
    }

    // parameterized becavior
    public void accelerate(int amount) {
        speed += amount;
    }
}
```

Example

<ELEPHANT> trinket example

Extending Another Class

Inheritance is one of the 4 pillars of OOP, it allows us to extend the behavior of another class so we can reduce the amount of code we have to write. In java the behavior of this is that any given subtype inherits the methods and attributes of the parent class, an example of this would be a plain class “Animal” with subtypes for different manufacturers, like having a “Dog” or “Bird” class. These subtypes might be distinct types, but they all share very similar properties, because of this we can inherit all the common methods from the “Animal” class. Additionally We do not have to be as specific when we are constructing subtypes, as we can just use a supertype as a type parameter, if that is specific enough. An example is shown here:

```
class Car {
    private int speed = 0;

    public void move() {
        speed += 5
   }
}

// the extends keyword denotes inheritance
class SportsCar extends Car {
    public void makeEngineNoises() {
        System.out.println("Vrooom");
    }
}

Car s2000 = new Car();
s2000.move();
s2000.makeEngineNoises();
```

Final Methods

Final is a special modifier for methods because it prevents a child from modifying the implementation of that method, this is used to make a method consistent across a class hierarchy.

Method Overriding

If you want to change the behavior of a method inherited from a parent class, you can still do this using decorators. (A decorator is an annotation in java which changes the behavior of a method) An example of when you would want to use override would be if you had a class “dog” which extends “animal” and “dog” inherited a method “walk”, you might want to change the implementation of “walk” to be more specific to how a dog walks. Here is an example:

```
class Car {
    private speed = 0;

    // default implementation for accelerate
    public void accelerate() {
        speed += 5
    }
}

class SportsCar extends Car {
    // we inherit

    // we can override the accelerate method to
    @Override
    public void accelerate() {
        speed += 10
    }
}

Car s2000 = new SportsCar();
s2000.move();
```

Pitfalls of Inheritance

While inheritance is an important part of object oriented programming, it is also a controversial tool among programmers. Complex inheritance hierarchies can make code far more difficult to understand, and can introduce unexpected and unwanted behavior, additionally inheritance creates a dependency between child and parent classes, which is something we generally seek to avoid when writing our programs. Due to these issues inheritance is not widely recommended to be used, and some modern languages like Go and Rust have even abandoned it completely. Java specifically also does not support multiple inheritance, which means a class can only ever inherit from a single other class, which can limit the flexibility of inheritance.

Composition

To deal with the issues of inheritance in OOP, most programmers today recommend to use composition as an alternative to inheritance. Composition represents a "has a" relationship instead of a "is a child of" relationship, which means we think about the components of an object which make it do what we want, instead of just inheriting from a parent with similar behavior. This way results in much looser coupling of components, and more versatility. Composition is also very easy, we simply assign objects of other classes as attributes of our class, this allows us to access the behavior of that class without the deep coupling and other pitfalls of inheritance. This is not to say that inheritance never has its uses, but composition can be better in many circumstances when you want to modify and extend the behavior of an existing class.

```java
class Car { // car does not extend anything

    // composing this class with other types
    private Wheels wheels = new Wheels();

    public void move() {
        wheels.turn();
    }
}
```

Programming to an interface

Interfaces in object oriented programming are important because they allow us to separate and loosely couple our code. A great example of this is the builtin "java.util.Collection" interface. This interface describes the methods and constants that a collection must have. Because of this we can trust that we can interact with all collections in similar ways, meaning we don't have to learn the methods for each specific implementation of a collection, and we can be generic when we are constructing collections, reducing dependencies on specific types. Here is an example:

```
interface Vehicle {
    // methods to not need bodies in interfaces
    void speedUp();
    void slowDown();
}

// implements shows that this class follows that
class Car implements Vehicle {
    private int speed = 0;

    // both of the methods defined in the interf
    // we also use the
    @Override
    public void speedUp() {
        speed += 1;
    }
    @Override
    public void slowDown() {
        speed -= 1;
    }
}
```

Other benefits to programming to an interface

Some other benefits to programming to an interface include: Testability, by making many of our classes follow a common interface, we can more easily write unit tests to make sure they work properly, and it makes human testing much simpler. In a business setting it also lets us test multiple implementations of an idea without fully committing to them, which can be extremely useful when getting feedback from a client, or when you need to change your implementation.

Test what you’ve learned

Conclusion

In this article we covered how to do effective object oriented programming in java through writing classes, and covered how classes work, their attributes, methods, inheritance, composition, and programming to an interface. Using all of these tools, you should be able to improve the quality of your code, and understand how code interacts.

About this article

This is my semester 1 final for my AP Computer Science class, feel free to respond with any feedback or corrections!