Sunday, December 30, 2018 2:01 PM

Anatomy of an Object

Parts of an Object Definition

```
public class Student {
   // fields
   String name;
   int grad;
   int ID;
   double GPA;
   int abs;

   // constructor
   public Student(String name, int grad, int ID, double GPA, int abs) {
      this.name = name;
      this.grad = grad;
      this.ID = ID;
      this.GPA = GPA;
      this.abs = abs;
   }

   // behavior
   public boolean isGraduating() {
      return (GPA > 2.0 && abs < 10 && grad == 12);
   }
}
```

In the above you can see there are generally three distinct parts to this code. In Java, objects are composed of three distinct parts. In your Java class, they are usually organized like this

In the following sections we'll discuss each of these pieces in detail.

State

The state of an object is defined by the collection of instance variables who's scope

is that of the entire object. We call these "fields". This collection of variables holds all the data associated with an object instance. Together, they represent the state of an object, everything that distinguishes it from other objects of the same type, or from itself at different points in time. If you created another object with exactly the same state, it would behave identically, but still be considered a separate object. That is because each time you make a new object you create a space for it in memory, so even if that memory stores the same values they are still distinct allocations of data.

Constructors

Constructors are a special type of method that is invoked by the "new" operator. This is what creates and initialize instances of the object and typically sets the initial value for all the fields.

Behavior

The behavior of an object is dictated by what methods you define in the object class. These methods represent the things a client class can do with an objects of this type. These methods typically modify the state, but they can also interact other classes or with the user like a Scanner or a PrintStream does. (System.out is actually an object of the PrintStream class.)

State

The state of an object is the collection of values that are stored within an object's fields.

Defining Fields

You define fields the same way define any other variable, but you define them outside of a method, traditionally right after the class header.

```
public class MyObject {
   String field1 = "";
   int field2 = 0;
   double field3 = 0.0;

   ...
}
```

By defining the fields outside any method that gives them the scope of the entire class. This means that they can be used by any method within the class, so you should try to limit them to those values that are strictly necessary.

You need to determine what Java data type to use and they can be a primitive data types, or any type of object including other classes that you have defined. Here’s what our Bicycle object might have:

```
public class Bicycle {
   String brand = "";
   String model = "";
   double listPrice = 0.0;
   int tireSize = 24;
   boolean isSold = false;
   double salePrice = 0.0;
   ...
```

These are the aspects of a bicycle that are most important for the shop owner, so they are created as fields.

Fields can be objects as well. For example, you could create a list of all the different accessories that come installed on the bike:

```
ArrayList<String> accessoryList = new ArrayList<String>();
```

These accessories might even be objects themselves, defined in Accessory.java; if so, the field would look like:

```
ArrayList<Accessory> accessoryList = new ArrayList<Accessory>();
```

As shown, the naming convention for fields is to use nouns in camelCase.

Using fields

Once you've created a field it works just like any other variable. You refer to it by name whenever you need to use it within your object's code.

```
public class Bicycle {
   double listPrice = 899.99;
   double salePrice = 599.99;

   // Updates the price of the bicycle to be the // reduced sale price
   public void setSale() {
      listPrice = salePrice;
   }
}
```

You can also access an object instance's fields from the client class using the name of the variable like so:

```
public class BicycleManager {
   public static void main(String[] args) {
      Bicycle myBike = new Bicycle();
      myBike.listPrice = 1000.00;
   }
}
```

However keep in mind that directly accessing the fields from the client code is not considered good style. We will talk about how to improve this in the encapsulation section.

This Keyword

Sometimes it can get confusing between what parameters are inside your class and which are being passed in from an external source. To help make this more clear you can add the "this" keyword in front of your object fields.

For example, you might have a method that takes in a parameter from the client that you want to save into the value of your field. You can use the this keyword to make it easier to distinguish between the parameter and the field.

```
public void setPrice(double newPrice) {
   this.listPrice = newPrice;
}
```

We will learn more situations in which to use the "this" keyword in the following sections.

Behavior

The behavior of an object is the collection of methods within the object.

These methods will represent the different things you want your client class to be able to do with the state stored in an object's fields. You should think through how your client class will use your object and that should help you determine which methods to include in your object.

Defining Methods

You write methods for an object the same way you've written other methods, except you omit the static property.

```
public returnType methodName(parameters) {
   statements;
}
```

You will want to define your methods after your fields within your class, like so:

```
public class MyClass {
   String field1 = "";
   int field2 = 0;

   public void updateField(int param) {
      field2 = param;
   }
}
```

The job of these methods is to execute some action, so often they will have names that begin with verbs like "get", "set", "calculate" etc...

As this work is typically performed on the fields, you will usually want to combine your verb with which field this method acts upon.

Let's look at our Bicycle example. Our Bicycle field has the following fields:

```
public class Bicycle {
   String brand = "";
   String model = "";
   double listPrice = 0.0;
   int tireSize = 24;
   boolean isSold = false;
   double salePrice = 0.0;
   ...
```

This Bicycle class is intended to be used by a shop keeper who sells Bicycles, so some things they might want to do are:

```
// returns a String with basic information about
// the bicycle
public String getInfo() {
   return brand + " " + model + " $" + price;
}

// marks the bicycle as sold and returns
// the asking price for the customer
public double makeFullPriceSale() {
   isSold = true;
   return listPrice;
}

// marks the bicycle as sold and returns
// the asking price for the customer
// during a sale
public double makeReducedPriceSale() {
   isSold = true;
   return salePrice;
}
```

As you can see we can create behaviors based on what a user might want to be able to do with the fields.

Working with other Objects

Your object's methods can access the field in other objects of the same type. For example, in our Bicycle object we could implement an equals method that determines whether two Bicycle objects have the same brand and model like so:

```
public boolean equals(Bicycle other) {
   return brand.equals(other.brand) && model.equals(other.model);
}
```

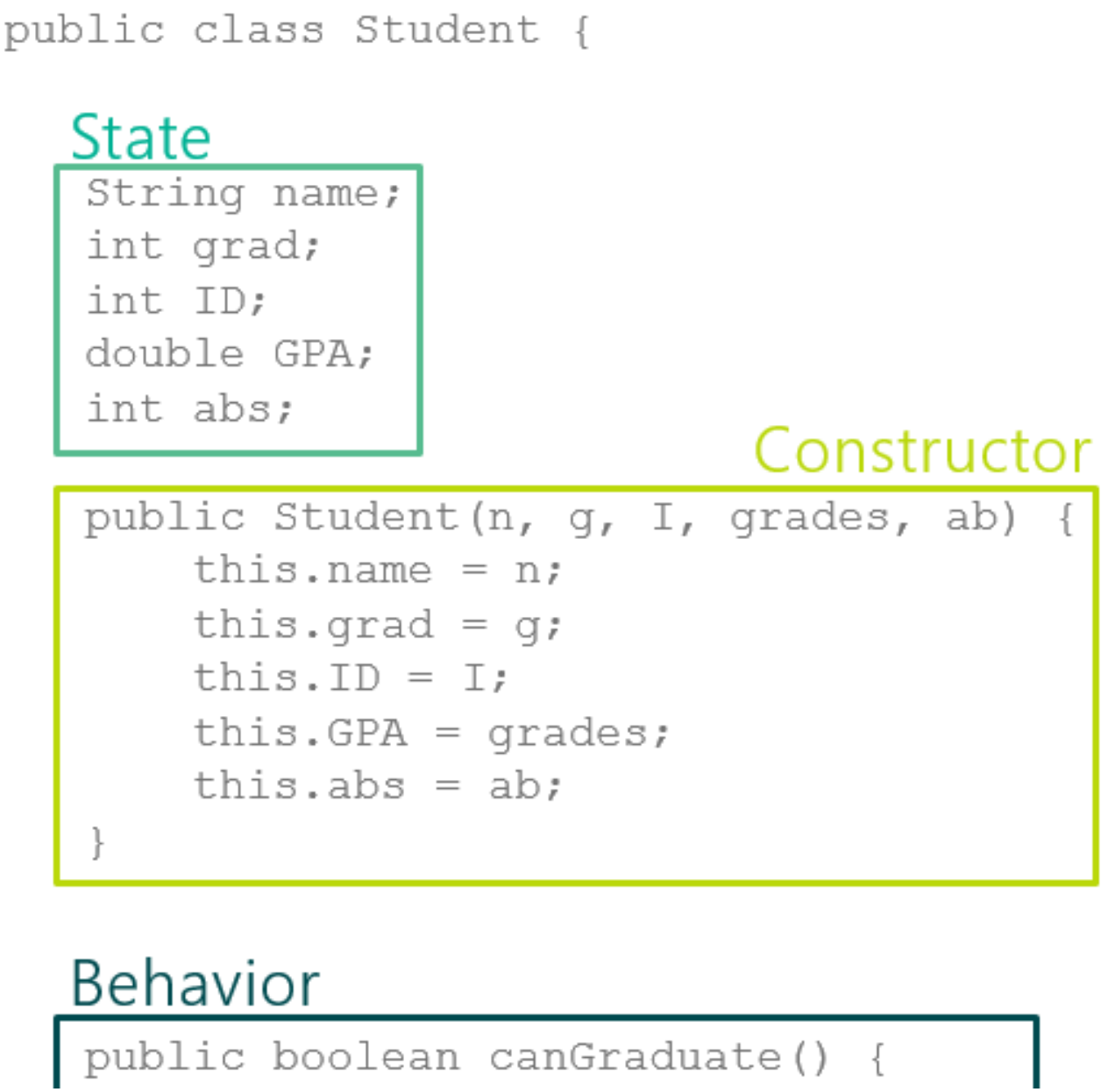
This is another case where the "this" keyword could help make things more clear. The above method could be rewritten:

```
public boolean equals(Bicycle other) {
   return this.brand.equals(other.brand) && this.model.equals(other.model);
}
```

The client code would use this method on two separate Bicycle variables, like so:

```
Bicycle bike1 = new Bicycle("Schwinn", "Cruiser");
Bicycle bike2 = new Bicycle("Giant", "Racer");

if (bike1.equals(bike2)) { // in this case would return false
   bike2.listPrice = bike1.listPrice;
}
```



```
    return (GPA >= 2.0 && abs >= 5);
  }
}
```