Chapter 6 - Classes

Last updated: Wednesday, November 27, 2024

TL;DR

Every Java file is a class. We say it's either Executable or an Object class.

- Executable classes have a `main` method which executes sequentially. (`TestRectangle`, `TestStudent`). These runnable classes. Inside of executable classes, you can create as many objects as you want, including custom Objects that you write.
- Object classes don't have `main` method, not runnable. (`Rectangle`, `Student`)
  - Object classes have fields. Fields are object descriptors. When you create an object, it has those fields. You can set those fields by writing **setter** methods or **constructors**. Object Fields are private. You write a **getter** method to access a field.

IntelliJ Chapter 06 Source Code

1. Open `Source Code\Chapter 06` as a project in IntelliJ.
2. Open the `DiceDemo` file.
3. Define a Project JDK.
4. Delete folders **Rectangle Class Phase 1**, **Rectangle Class Phase 2**, **Rectangle Class Phase 3**, **Rectangle Class Phase 5**, and **Cho-Han**.
5. Run the `DiceDemo` file.

If you're able to run that file, you're good! 👍

IntelliJ Chapter 08 Source Code

Delete Stock Class Phase 1, 2, 3, and 4.

Recap: Static vs. instance methods

Differences between `static` and instance From the Java Language Specification:

> A method that is declared `static` is called a *class method*.

> A class method is **always invoked without reference to a particular object**. The declaration of a class method introduces a static context), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a

> static context, as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations.

> A method that is not declared `static` is called an *instance method*, and sometimes called a non-`static` method.

> An instance method is **always invoked with respect to an object**, which becomes the current object to which the keywords `this` and `super` refer during execution of the method body.

Recap: Classes vs. Objects

In lectures, I use the terms "Class" and "Object" interchangeably. I ask, "Can someone name a Java Object?" and say "Classes have methods", but I am referring to the same set of things.

Java Objects or Classes that we know include String, Scanner, File, PrintWriter, FileWriter, Random, Math, System, and (a bit of) the "wrapper" classes: Integer, Double, Boolean (which make Primitive Data Types behave like Java Objects).

In most cases it is okay that I use the terms interchangeably, but there is a technical difference between the two terms:

- **Class**: A blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) that the objects created from the class will have.
- **Object**: An instance of a class. It is created based on the class blueprint and has its own state and behavior as defined by the class.

The class is like a blueprint, and the object is like house we build from the blueprint. In Java, the blueprint is usually flexible (and we can write our own blueprints).

Terminology for two types of classes: Executable vs. Object

For our purposes, there are two types of Java classes:

- Those with a `main` method
- Those without a `main` method

I will call this distinction: **Executable classes vs. object classes**.

- An executable class has a main method. These types of classes are also referred to as a Main Class, an Entry Point Class, or a Test Class (think: **TestRectangle.java**).

- An object class is a blueprint that defines a Java object. It does not contain a main method. These types of classes are also referred to as a Helper Classes, Model Classes, or POJO (Plain Old Java Objects) (think: **Rectangle.java**).

Object Class: Fields and Methods

Object classes in Java have fields and methods.

- **Fields** are the data stored about an object (Rectangle has `length` and `width` fields). Fields are like variables except they have an access modifier (`private` or `public`). Fields should be `private` so that other classes don't access the data directly.
  - As said in the Java Language specification: "A `static` field, sometimes called a *class variable*, is incarnated when the class is initialized."
- **Methods** are the operations that an object can perform. We know methods from Chapter 5. Examples for object classes include a **set** and a **get** method for each **field** (For example, `setLength` and `getLength`).
  - A special type of method is a **Constructor**.

Here is the full `Rectangle.java` object class, which we will look at as we go:

```
public class Rectangle {
        private double length;
        private double width;

        public Rectangle(double length, double width){
                this.length = length;
                this.width = width;
        }

        public void setLength(double len){
                length = len;
        }

        public double getLength(){
                return length;
        }

        public void setWidth(double wid){
                width = wid;
        }

        public double getWidth(){
                return width;
        }
```

```
        public double getArea(){
                return length * width;
        }
}
```

Executable class: Using the Object class

Executable classes are no different than every class we've seen this semester: They are the ones that have a `main` method. The only difference is that, within them, we initialize our custom Object class.

Assume `Rectangle.java` and the following class (`TestRectangle.java`) are in the same directory (this way, we won't have to import anything):

```
public class TestRectangle {
        public static void main(String[] args) {
                Rectangle box = new Rectangle(8.5, 12.5);
                System.out.println(box.getLength());
                System.out.println(box.getWidth());

                box.setLength(10);
                box.setWidth(10);
                System.out.println(box.getArea());
        }
}
```

We initialize our Rectangle object like we would with a Scanner, File, or Random object. Then we run methods on the object. We also make use of the two argument constructor to set the default length and width of the rectangle.

Object methods: Getters and Setters

Although object fields are private, we may want to get and set their values. For every field you want to modify, write a get and set function for that field. Once you learn the "recipe", all getters and setters have the same syntax. For example:

```
public class Thing {

        private String name;
        private int age;

        public void setName(String nm){
```

```
                name = nm;
        }

        public String getName(){
                return name;
        }

        public void setAge(int ag){
                age = age;
        }

        public int getAge(){
                return age;
        }
}
```

The input parameter `String nm` would be used as the argument when you call the `setName()` method, like so:

```
public class TestThing{
        public static void main(String[] args){
                Thing a = new Thing();
                a.setName("Arnold");
                System.out.println(a.getName());
                a.setAge(10);
                System.out.println(a.getAge());
        }
}
```

An alternative way to write setter methods is like so:

```
public class Thing {
    private double gpa;

    public void setGpa(double gpa){
        this.gpa = gpa;
    }
}
```

Using the `this.{fieldname} = {fieldname}` syntax eliminates the need for misspelling or writing unclear variables.

Constructors: Set initial values

The default constructor

When an object is created, its constructor is always called.

If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the **default constructor**.

- It sets all of the object's numeric fields to 0.
- It sets all of the object's boolean fields to false.
- It sets all of the object's reference variables to the special value null.

Here are the defaults described for each data type:

Data Type	Default Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0`
`char`	`'\u0000'` (null character)
`boolean`	`false`
`String`	`null`

Let's say we write a Java class called Car, which uses only the default constructor:

```
public class Car {
        private String make;
        private int year;
        private double gasTankMax;
        private char grade;

        public String getMake() { return make; }
        public int getYear() { return year; }
        public double getGasTankMax() { return gasTankMax; }
        public char getGrade() { return grade; }

        public static void getCarInfo(Car c){
                System.out.println(c.make);
                System.out.println(c.year);
```

```
                System.out.println(c.gasTankMax);
                System.out.println(c.grade);
        }

        public static void main(String[] args) {
                Car defaultCar = new Car();
                getCarInfo(defaultCar);
        }
}
```

Here's what is printed when `getCarInfo(defaultCar)` runs:

```
null
0
0.0
  //(There is a special character called NUL at the beginning of this line)
```

Writing our own constructors

We can add Getter and Setter methods to access and mutate the object data. But we can also create object constructors to set the values at instance creation.

Here's an example constructor for the Car Object class:

```
public class Car {
        private String make;
        private String model;
        private int year;
        private String color;

        public Car(String make, String model, int year, String color){
                this.make = make;
                this.model = model;
                this.year = year;
                this.color = color;
        }
}
```

If we want to use the Car object in another class, we'd instantiate a car with the arguments in the specified order, like so:

```
public class TestCar {
        public static void main(String[] args) {
```

```
                Car myCar = new Car("Toyota", "Corolla", 2024, "red");
                Car yourCar = new Car("Ford", "Escape", 2020, "blue");
        }
}
```

One great thing about objects is that you can instantiate (create) as many as you need. In the above example, we've created two car objects, `myCar` and `yourCar`. If we have getter and setter methods, we can access and change information about each car separately.

Constructor properties

Constructors have a few special properties that set them apart from normal methods.

- Constructors have the same name as the class.
- Constructors have no return type (not even void).
- Constructors may not return any values.
- Constructors are typically public.

No-arg Constructors

The default constructor is a constructor with no parameters, used to initialize an object in a default configuration. This is seen in an Executable class when you run this statement (for example):

```
Rectangle box = new Rectangle();
```

There are no arguments in the parentheses because you are creating a default object.

The only time that Java provides a default constructor is when you do not write any constructor for a class.

When you write your own constructor, for example, with two args (as you might see in Rectangle), like so:

```
public Rectangle(double length, double width){
        this.length = length;
        this.width = width;
}
```

If you try to create a Rectangle object with no-args, you will get an error! However, this can be fixed by adding a no-arg constructor to the Rectangle object class (like so):

```
public Rectangle() {
}
```

See **Overloading classes** for more info.

The String class constructor

One of the String class constructors accepts a string literal as an argument.
This string literal is used to initialize a String object.
For instance:
`String name = new String("Michael Jordan");`

This creates a new reference variable name that points to a String object that represents the name “Michael Jordan”. Because they are used so often (String is a special object), we usually just write: `` String name = "Michael Jordan";` ``

Calculated methods: Prevent Stale Data

Above in our Rectangle class, we have a fifth method, `getArea()`, which looks like this:

```
public double getArea(){
        return length * width;
}
```

This method calculates the area at the time it's called. This allows us to return a dynamic value of the rectangle’s area when the method is called. For example:

```
public class TestRectangle {
        public static void main(String[] args) {
                Rectangle box = new Rectangle(8.5, 12.5);
                System.out.println(box.getArea());
                //Returns 106.25

                box.setLength(10);
                box.setWidth(10);
                System.out.println(box.getArea());
                //Returns 100.0
        }
}
```

It would be impractical to use an `area` field in the Rectangle class because data that requires the calculation of various factors has the potential to become stale.

To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a variable.

Instance Fields and Methods

Constructors allow you to create numerous instances of an Object class, and work with them independently. For example:

```
Rectangle kitchen = new Rectangle(12.5, 12.5);
Rectangle mainBedroom = new Rectangle(18.0, 13.0);
Rectangle kidBedroom = new Rectangle(14.0, 8.0);
```

Those three "rooms" hold different addresses in memory -- three different objects, three different instances of a "room" object class. In the memory location for each room is a field for length and width (slide 37-39).

Overloading methods

Two or more methods in a class may have the same name as long as their parameter lists are different.

An example is `Random.nextInt()`. It behaves different based on whether you give it zero input parameters, one input parameter, or two.

When this occurs, it is called **method overloading**. This also applies to constructors but then it is called **class overloading**.

Method overloading is important because sometimes you need several different ways to perform the same operation. You may want a user to be able to add two ints, or two doubles. So you can write a method for each data type:

```
public int add(int num1, int num2) {
   return num1 + num2;
}

public double add(double num1, double num2) {
   return num1 + num2;
}
```

A method signature consists of the method's name and the data types of the method's parameters, in the order that they appear. The return type is not part of the signature.

The process of matching a method call with the correct method is known as **binding**. The compiler uses the method signature to determine which version of the overloaded method to bind the call to.

Overloading classes

If you write multiple Object class constructors, you can initialize objects with different default values.

Let's say you had the Car class with four fields.
You could write as many object constructors as you wanted from no-arg, to 1-arg, 2-arg, 3-arg, and 4-arg. You could also write the parameters in different orders depending on the business logic needed.

```
public class Car {
    private String make;
    private String model;
    private int year;
    private String color;

    // Constructor with all parameters
    public Car(String make, String model, int year, String color) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
    }

    // Constructor with default color
    public Car(String make, String model, int year) {
        this(make, model, year, "Unknown");
    }

    // Constructor with default year and color
    public Car(String make, String color) {
        this(make, "Unknown", 2024, color);
    }

        public Car(){
                this.make = "Unknown";
                this.model = "Unknown";
                this.year = 0000;
                this.color = "Unknown";
        }
```

```
    // Getters and setters...
}
```

This would give you the flexibility when creating a Car object to specify as many values as you know at instance creation time:

```
Car profCar = new Car("Toyota", "Corolla", 2024, "red");
Car cooperCar = new Car("Vovlo", "S60", 2007);
Car taylorCar = new Car("McLaren", "Light blue");
Car seamusCar = new Car();
```

With Java, your constructor arguments must be in the correct order of an existing constructor. See the "Optional reading: Optional values" section for more about this.

Optional reading: Optional values

Optional or named input parameters are not supported by default in Java! 😔🤧 This is not cool! and definitely something I will look at after I am elected president of Java.

See other-diversions/optional-and-named-arguments for an overview and workaround.

Uninitialized Local Reference Variables

You can declare a variable in an Executable class of your custom type without initializing it. •A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
public class MyClass{
        public static void main(String[] args) {
                Rectangle box;
                box = new Rectangle(7.0, 14.0);
        }
}
```

Important! Passing a Custom Object as a Method Input Parameter

Remember when we said that Methods take input parameters in parentheses? Example:

```
public static void setLength(int length) {}
public static void setName(String name) {}
```

You can do the same with your custom objects:

```
public static void displayRectangle(Rectangle r) {}
```

When you pass an object as an argument, the thing that is passed into the parameter variable is the object's memory address. As a result, parameter variable references the object, and the receiving method has access to the object.

```
// Rectangle.java class
public static void displayRectangle(Rectangle r){
        System.out.println("Length: " + r.getLength() +
                                                "Width: " + r.getWidth());
}
```

When you pass an object as an argument to a method, the method will almost always be **static** and all the instance methods are available. You can use this to access and mutate information about the particular object.

Methods that take custom objects as input parameters can access the fields and methods of the object. You can leverage this fact to make repeatable code that works on ANY instance of the object. For example:

```
public class Car {
        // ... fields, methods, constructors ...

        public static void printAutoInfo(String carOwner, Car c){
                System.out.println(carOwner.toUpperCase() + "'s DREAM
AUTOMOBILE: ");
                System.out.println("Car Make: " + c.make);
                System.out.println("Car Model: " + c.model);
                System.out.println("Car Year: " + c.year);
                System.out.println("Car Color: " + c.color);
                System.out.println();
        }
}
```

In our `TestCar` class, we can create as many car objects as we want, and then call the `printAutoInfo()` method and supply it the car owner and Car object.

```
Car firstCar = new Car("BMW", "3-series", 1997, "blue");
Car secondCar = new Car("Tesla", "Cybertruck", 2024, "matte black");
Car thirdCar = new Car("Toyota", "Prius", 2024, "matte black");
```

```
Car fourthCar = new Car("Nissan", "Rogue", 2024, "white");

Car.printAutoInfo("Michael", firstCar);
Car.printAutoInfo("Rudra", secondCar);
Car.printAutoInfo("Nolan", thirdCar);
Car.printAutoInfo("Dre", fourthCar);
```

Returning objects from methods

Methods are not limited to returning the primitive data types. Methods can return references to objects as well. Just as with passing arguments, a copy of the object is not returned, only its address.

It is mostly used in Factory design patterns, where you have a Factory class that handles the creation of object instances.

We won't cover this topic in any greater detail in this course, but here is an example of two methods that return custom objects. They act like constructor methods, but they allow you to set different default values depending on your business needs:

```
public class Car {
        private String type;
        private String model;
        private String year;

        public static Car createElectricCar(String model, int year) {
                return new Car("Electric", model, year);
        }
        public static Car createGasCar(String model, int year) {
                return new Car("Gas", model, year);
        }

    public static void main(String[] args) {
        Car electricCar = Car.createElectricCar("Model S", 2024);
        Car gasCar = Car.createGasCar("Jeep Wrangler", 2002);
    }
}
```

Custom object .toString() method

All objects have a toString() method that returns the class name and a hash of the memory address of the object. It is also called implicitly whenever you try to print the object:

```
PrintWriter pw = new PrintWriter("Filename.txt");
System.out.println(pw.toString());
// java.io.PrintWriter@5b480cf9

Die d = new Die(45);
System.out.println(d);
// Die@10f87f48
```

Some objects override the default toString() method, like `File` and `String`:

```
File f = new File("Filename.txt");
System.out.println(f);
// Filename.txt

String s = "Sarah";
System.out.println(s);
// Sarah
```

When we create our custom objects, we get the default toString() method that Java provides:

```
User sarah = new User("Sarah", "password", "Sunnydale High", "I'm 100% certain
that I am 0% sure of what I'm doing.");
System.out.println(sarah);
// User@2f4d3709
```

But we can override the default method with our own to print out more useful information. Here's an example from the `User` class:

```
@Override
public String toString(){
        String str = String.format("ID: %-7d Username: %s %nSchool: %s
%nQuote: \"%s\" ", id, username, school, quote);
        return str;
}
```

Then, when we call the `User sarah` object again, the output is printed as we expect to see it.

```
User sarah = new User("Sarah", "password", "Sunnydale High", "I'm 100% certain
that I am 0% sure of what I'm doing.");
System.out.println(sarah);

// ID: 39930    Username: Sarah Wilson
```

```
// School: Sunnydale High
// Quote: "I'm 100% certain that I am 0% sure of what I'm doing."
```

Custom object .equals() method

The default .equals() method for a Java Object compares memory locations. We see this when we print a class object:

```
public class EqualsCompareObjects {
        public static void main(String[] args) {
                Car myCar = new Car("Honda", "Civic", 2020, "blue");
                Car yourCar = new Car("Honda", "Civic", 2020, "blue");

                System.out.println(myCar);
                //Car@5f184fc6
                System.out.println(yourCar);
                //Car@3feba861
        }
}
```

Notice how above says `Car@xxxxx`. The JVM is printing the object type and the memory location in hexadecimal.

If you run `myCar.equals(yourCar)`, it evaluates to `false` because the objects are in different memory locations.

You can overwrite the equals() method by adding logic that makes sense within the context of the object or business requirements. For example, if you wanted to define "equal" for a User object as having the same username and school, you could write an equals method like so:

```
public class User {
        private String username;
        private String school;

        // ...  constructors, setter, getter methods here ...

        public boolean equals(User user2){
                return username.equals(user2.username) &&
school.equals(user2.school);
        }
}
```

As an input parameter, `.equals()` takes another instance of the same object. The `equals()` method now checks to see if both User fields are the same in user A and user B. If they are, this could tell us that we have a duplicate entry. For example:

```
User jill = new User("Jill Cooley", "Bentley University");
User jillian = new User("Jill Cooley", "Bentley University");

if (jill.equals(jillian)){
        System.out.println("Possible duplicate entry");
}
```

You aren't required to write a custom .equals() method on the final homework assignment. But if you play around with this, you may notice inconsistent results. Because sometimes the JVM compiler will use the default equals() method rather than your custom method, when you call the .equals method on your custom objects, the comparison may be by memory location.

To write a more comprehensive custom .equals() method, you must use the @Override annotation, but the syntax is a bit more complicated. If you find yourself having issues, try to edit the following code to suit your purposes. If you are doing things the proper way, you must also override the hashCode() method, also included:

```
@Override
public boolean equals(Object obj) {
        // first, if the memory locations match, return true.
        // they are the same object.
        if (this == obj) {
                return true;
        }

        // next, if the comparison object is null,
        // or the two class types aren't the same, return false.
        // the objects cannot be equal.
        if (obj == null || getClass() != obj.getClass()) {
                return false;
        }

        // finally, cast the comparison object to your custom type
        // then evaluate your object fields for equality
        // here, usernames and passwords must match.
        User comparisonObject = (User) obj;
        return username.equals(comparisonObject.username)
                        && password.equals(comparisonObject.password);
}
```

```java
@Override
public int hashCode() {
        // Import java.util.Objects
        // and provide the comparison field values
        // this ensure that equal objects have the same hash code.
        return Objects.hash(username, password);
}
```