Chapter 6 - Classes

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: 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.
- **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**.

- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

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 len, double wid){
                length = len;
                width = wid;
        }

        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 = ag;
	}

	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 default values

If you don't initialize a field to some value, when you try to run and print a getter method, what will happen?

You will print the default value of the data type. Here's a list of the values:

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

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.

When defining our Object class, however, we could set default values for fields.

```
class Car {
        private String make = "Ford";
```

```
        private String model = "Escape";
}
```

But this is not flexible. We are stuck with pre-defined values for our object (either the data type default, or hard-coded defaults like the Ford Escape above).

Let's say we write a Java class called Car and it had the following properties:

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

We can add Getter and Setter methods to access and mutate the object data. But we can also create an object constructor to set the values at instance creation. Here's a constructor for the Car Object class:

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

        public Car(String mk, String mdl, int yr, String col){
                make = mk;
                model = mdl;
                year = yr;
                color = col;
        }
}
```

Here is an alternative way to write the above constructor using the `this` keyword:

```
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.

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

A default constructor is not provided by Java if a constructor is already written.

Writing a no-arg constructor is equivalent to initializing the field values.

```
public Rectangle() {
  length = 1.0;
```

```
    width = 1.0;
}
```

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 say: `` 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

(Slide 37-38)
The most important distinction between methods (chapter 5) is between Static and Non-Static methods (which are known as INSTANCE methods).

Instance methods 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 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.

Method overloading is important because sometimes you need several different ways to perform the same operation.

```
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 the object in different ways.

In Class: Write various car constructors.

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);
        }
}
```

Passing Objects as Arguments

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 rollDice(Dice d) {}
```

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.

See: DieArgument.java

Bonus: Data Hiding: Why are Object fields private?

In Java, object class fields are typically made private to adhere to the principle of **encapsulation**, which is a fundamental concept in object-oriented programming. Here are a few reasons why this is important:

1. **Data Hiding**: By making fields private, you prevent direct access to them from outside the class. This helps protect the internal state of the object from unintended or harmful modifications.
2. **Controlled Access**: Private fields can only be accessed and modified through public methods (getters and setters). This allows you to control how the data is accessed and modified, ensuring that any necessary validation or processing is performed.
3. **Maintainability**: Encapsulation makes it easier to change the internal implementation of a class without affecting other parts of the code that rely on it. You can modify the private fields and the methods that access them without worrying about breaking external code.
4. **Improved Debugging**: When fields are private, you can more easily track and control how and when they are accessed or modified, which can simplify debugging and troubleshooting.
5. **Enhanced Security**: By restricting access to the fields, you reduce the risk of accidental or malicious interference with the object's state.

The difference between directly accessing a private field value and getting a private field using a method (like a getter) lies in the principles of encapsulation and control over the data. Here's a breakdown:

Direct Access (Not Recommended)

Directly accessing a private field value is generally not allowed in Java because private fields are not accessible outside their class. This is by design to enforce encapsulation. If you try to access a private field directly from outside the class, you'll get a compilation error.

Using a Getter Method (Recommended)

A getter method is a public method that provides controlled access to a private field. Here's why using a getter is beneficial:

1. **Encapsulation**: It maintains the principle of encapsulation by hiding the internal representation of the object.
2. **Control**: It allows you to control how the field is accessed. For example, you can add logic to the getter method to return a computed value or to log access.
3. **Validation**: You can include validation logic within the getter to ensure the field's value is appropriate before returning it.

Example: Direct access vs. using a Getter method

Here's a simple example to illustrate the difference:

```
public class Person {
    // Private field
    private String name;

    // Getter method
    public String getName() {
        return name;
    }

    // Setter method
    public void setName(String name) {
        this.name = name;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Alice");

        // Direct access (not allowed)
        // System.out.println(person.name); // This will cause a compilation
error

        // Access via getter method (allowed)
        System.out.println(person.getName()); // This will print "Alice"
    }
}
```

In this example, the `name` field is private and cannot be accessed directly from outside the `Person` class. Instead, you use the `getName()` method to access the value of `name`.