

# Chapter 6 - Classes

## 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.java` ). 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` , `TestStudent` )
  - 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: 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;  
    }  
}
```

Additionally, you can also create a multi-arg constructor like this:

```
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, model, year, 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 remedied 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 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 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.

## 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.

There's a Java design pattern called the Builder pattern that provides a more flexible and readable way to create objects with multiple optional parameters.

## 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 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.

## 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);
    }
}
```

## LET'S BUILD A SOCIAL NETWORK

1. Add a new package to Chapter 06 called `FriendBook`.
2. Create a `User` object class with these fields:

```
private int id = generateID();
private String username;
private String password;
private String school;
private String quote;
private List<User> friends
```

3. Write getter methods for all fields.
4. Write setter methods for quote.
5. Write a four-arg constructor to set the username, password, school, and quote.
6. Write a `generateID()` method that returns a `Random` int between 9999 and 99999.
7. Make a file in the same package called `TestFriendBook`. Run the file:

```
// TestFriendbook.java
package FriendBook;

public class TestFriendBook {
    public static void main(String[] args) {
        User john = new User("John Doe", "password", "Springfield
High", "I finally learned how to write without a pencil.");
        User jane = new User("Jane Smith", "password", "Riverview
Academy", "I'm outta here! See you all at the 10-year reunion.");
        User alex = new User("Alex Johnson", "password", "Mountainview
High", "I'm not a complete idiot. Some parts are missing.");
        User emily = new User("Emily Brown", "password", "Lakeside
School", "I'm not late; I'm just early for tomorrow.");
        User michael = new User("Michael Davis", "password", "Greenwood
High", "I didn't choose the school life; the school life chose me.");
        User sarah = new User("Sarah Wilson", "password", "Sunnydale
High", "I'm 100% certain that I am 0% sure of what I'm doing.");
        User david = new User("David Martinez", "password", "Riverside
High", "I'm not arguing; I'm just explaining why I'm right.");
        User laura = new User("Laura Garcia", "password", "Hilltop
Academy", "I'm not lazy; I'm on energy-saving mode.");
```

```

        User james = new User("James Lee", "password", "Westside High",
        "I'm not a procrastinator; I'm just extremely productive at unimportant
        things.");

        User sophia = new User("Sophia Kim", "password", "Eastwood
        High", "I'm not weird; I'm limited edition.");

        System.out.println("ID: " + james.getId());
        System.out.println("Username: " + james.getUsername());
        System.out.println("School: " + james.getSchool());
        System.out.println("Quote: " + james.getQuote());
        //User.printUserProfile(sophia);
        //System.out.println(sophia.toString());
    }
}

```

8. Move the print statements into the `User` class as a static void method called `printUserProfile` that takes `User u` as an input parameter. Uncomment the relevant line in `TestFriendbook` are see if it runs.

## 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

You can also override the default `.equals()` method of the object, which compares memory locations. You can set it to a value that makes sense within the context of the object or business requirements:

```
public boolean equals(User user2){  
    if(username.equals(user2.username)){  
        return true;  
    } else {  
        return false;  
    }  
}
```



}

}