Labs are instructional guides for you to learn. The goal of each lab is to teach you what you will need to be successful in the assignments and course project.

This setup is required:

Install IntelliJ: https://www.jetbrains.com/idea/download/?section=windows
NOTE: You can install Java from within IntelliJ, or you can follow the below link for Java 17. Regardless of method, you must use Java v17.

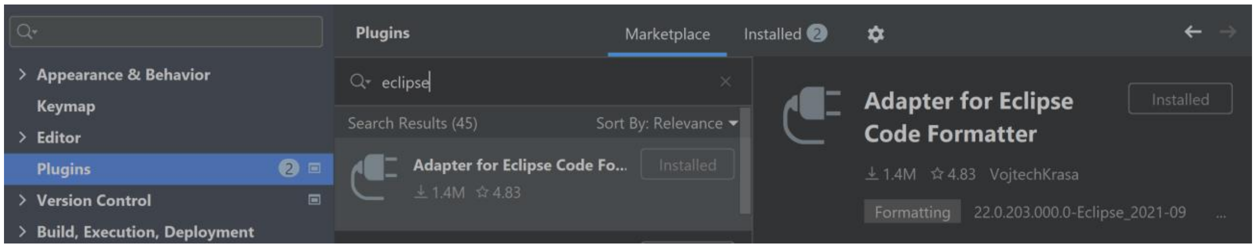
Download and install Java 17 JDK: https://www.oracle.com/java/technologies/downloads/

Note: The screenshots below are using an older version of IntelliJ. The UI has been updated but it is functionally the same. If there's any confusion, please ask the professor or TA, now, in class, where we can provide immediate help.

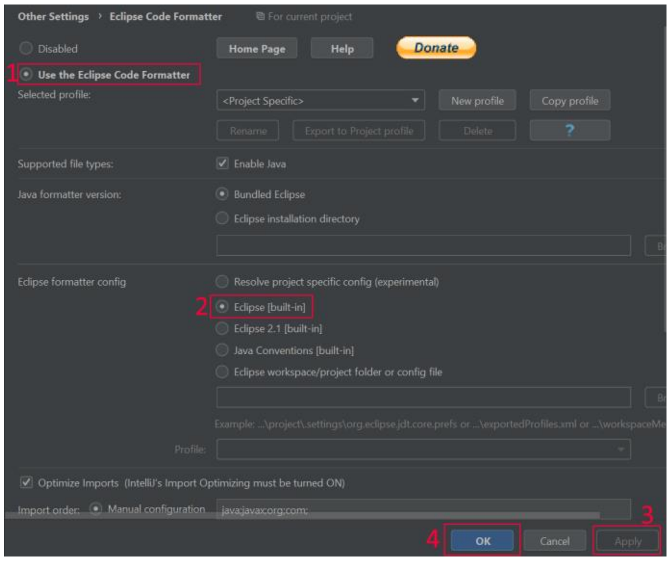
Create a new Java Project in IntelliJ:
File -> New Project -> Name it "Throwaway"
Under "JDK" use the dropdown to select "17"
Do not check "sample code." Click Create.

Next, open the Settings menu (File -> Settings, or CTRL + ALT + S). Click on Plugins on the left. Search for "save" and install the "Save Actions" plugin

Do the same thing, this time search for "eclipse" and install Adapter For Eclipse Code Formatter:



The save actions plugin will automatically trigger code formatting every time you make a change in any file you are coding.
The Eclipse code formatter will automatically format your code to the Java industry standard, which is required for all code in this course.
As such, make sure these plugins and all following settings are done before you start any development on any assignment in this course.
Restart IntelliJ. Go to File -> New Projects Setup -> Settings for New Projects…
Open the "Other Settings" section. Open "Adapter for Eclipse Code Formatter."
Following the image below, enable the formatter, select Eclipse [built-in], then Apply, then OK.



Other Settings › Eclipse Code Formatter — For current project

Disabled

1 [✓ Use the Eclipse Code Formatter]

Selected profile: <Project Specific>

Supported file types: ☑ Enable Java

Java formatter version: [✓ Bundled Eclipse] / Eclipse installation directory

Eclipse formatter config:
- Resolve project specific config (experimental)
- 2 [✓ Eclipse [built-in]]
- Eclipse 2.1 [built-in]
- Java Conventions [built-in]
- Eclipse workspace/project folder or config file

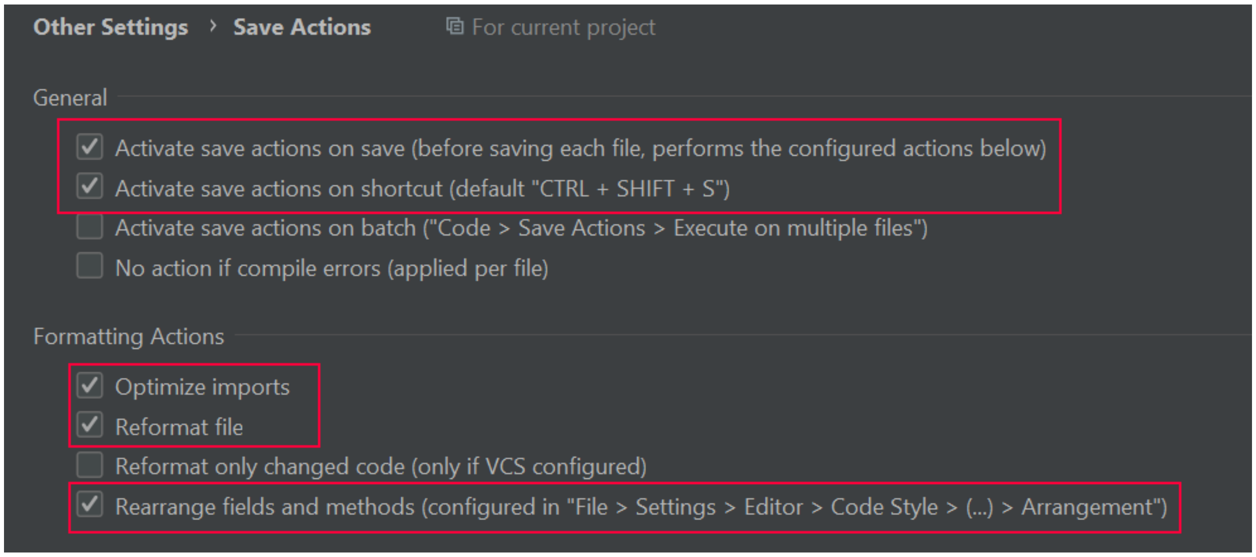
Example: ...\project\.settings\org.eclipse.jdt.core.prefs or ...\exportedProfiles.xml or ...\workspaceMe

Profile:

☑ Optimize Imports (IntelliJ's Import Optimizing must be turned ON)

Import order: [✓ Manual configuration] java;javax;org;com;

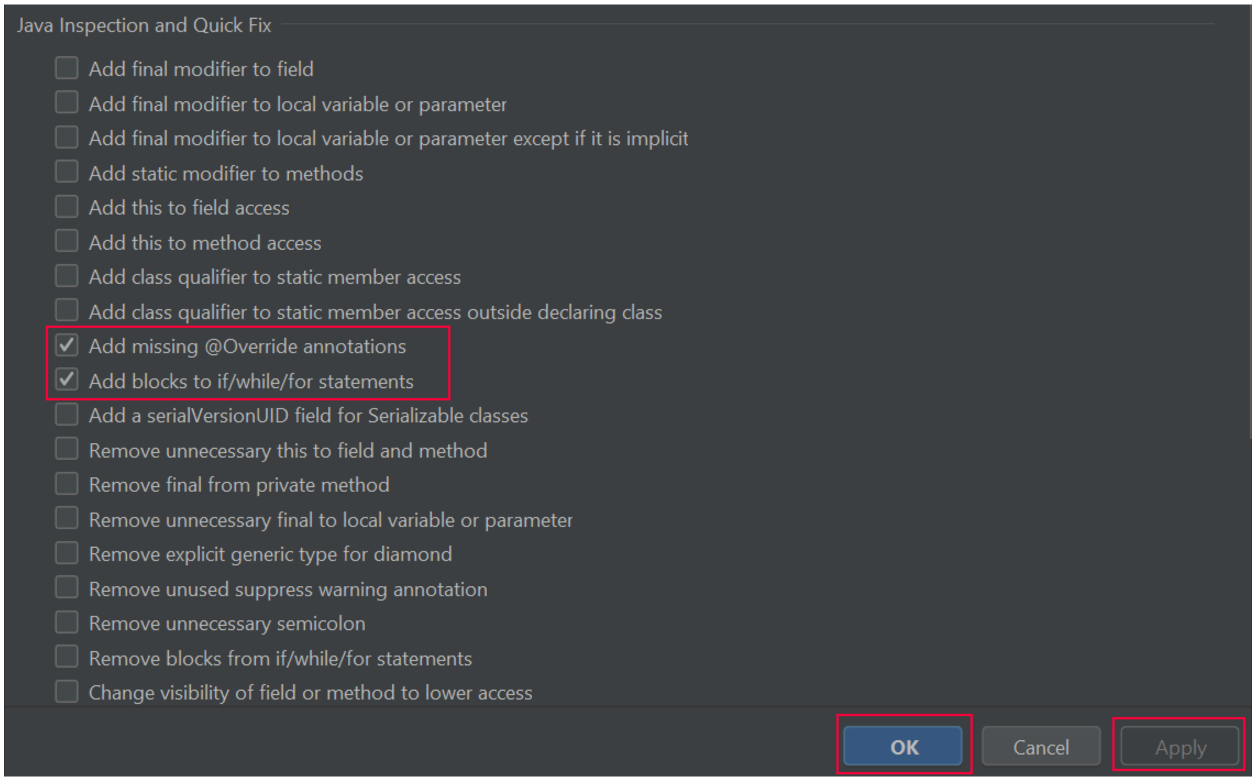
Under “Other Settings” click on “Save Actions”:



Other Settings › Save Actions — For current project

General
- ☑ Activate save actions on save (before saving each file, performs the configured actions below)
- ☑ Activate save actions on shortcut (default "CTRL + SHIFT + S")
- ☐ Activate save actions on batch ("Code > Save Actions > Execute on multiple files")
- ☐ No action if compile errors (applied per file)

Formatting Actions
- ☑ Optimize imports
- ☑ Reformat file
- ☐ Reformat only changed code (only if VCS configured)
- ☑ Rearrange fields and methods (configured in "File > Settings > Editor > Code Style > (...) > Arrangement")



Java Inspection and Quick Fix

- [ ] Add final modifier to field
- [ ] Add final modifier to local variable or parameter
- [ ] Add final modifier to local variable or parameter except if it is implicit
- [ ] Add static modifier to methods
- [ ] Add this to field access
- [ ] Add this to method access
- [ ] Add class qualifier to static member access
- [ ] Add class qualifier to static member access outside declaring class
- [✓] Add missing @Override annotations
- [✓] Add blocks to if/while/for statements
- [ ] Add a serialVersionUID field for Serializable classes
- [ ] Remove unnecessary this to field and method
- [ ] Remove final from private method
- [ ] Remove unnecessary final to local variable or parameter
- [ ] Remove explicit generic type for diamond
- [ ] Remove unused suppress warning annotation
- [ ] Remove unnecessary semicolon
- [ ] Remove blocks from if/while/for statements
- [ ] Change visibility of field or method to lower access

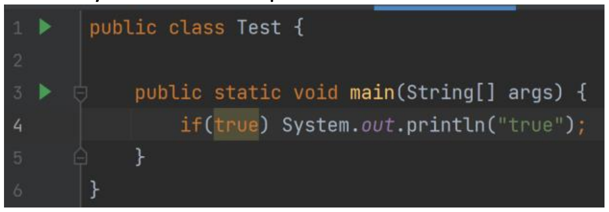
Select everything as it is above, then click Apply, then OK.

You will have to do all of this again if you reinstall IntelliJ or your OS

After you set up all plugins, make a **new** project for these changes to take effect!
File -> New Project -> Name it “OOP Lab” -> make sure JDK 11 is selected -> Create

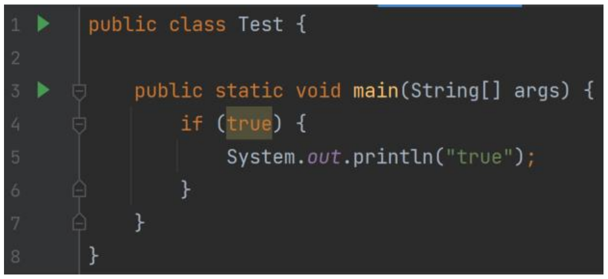
Making sure your plugins work

Let's say I coded a simple if statement on one line:



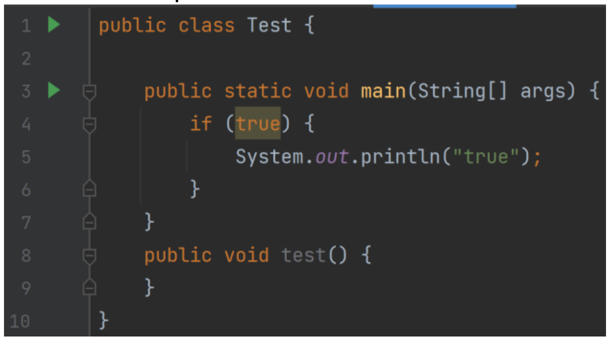
```
1   public class Test {
2
3       public static void main(String[] args) {
4           if(true) System.out.println("true");
5       }
6   }
```

When you save the file (CTRL+S), it will be automatically formatted. In addition, if you ALT+TAB to switch windows, this should also trigger a save, and thus automatically format your code for you. The formatted code looks like this:



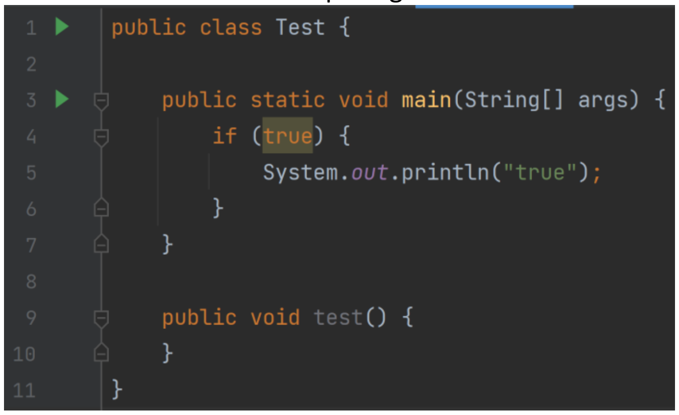
```
1   public class Test {
2
3       public static void main(String[] args) {
4           if (true) {
5               System.out.println("true");
6           }
7       }
8   }
```

Another example. Before:



```
1   public class Test {
2
3       public static void main(String[] args) {
4           if (true) {
5               System.out.println("true");
6           }
7       }
8       public void test() {
9       }
10  }
```

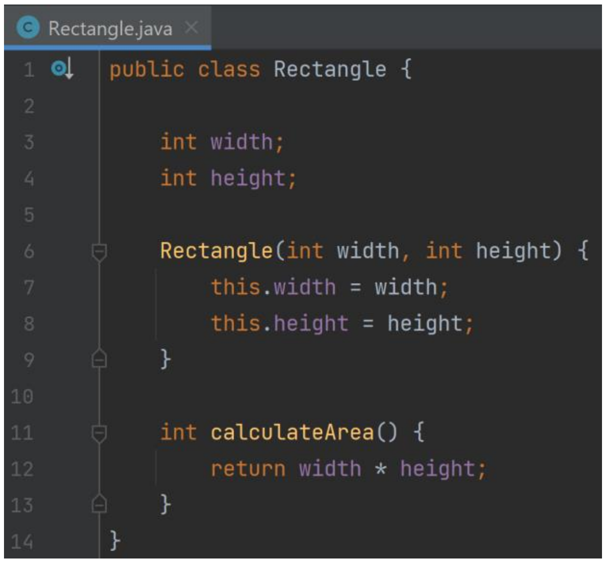
Notice the lack of vertical spacing between the two methods. After:



```
1   public class Test {
2
3       public static void main(String[] args) {
4           if (true) {
5               System.out.println("true");
6           }
7       }
8
9       public void test() {
10      }
11  }
```

Line 8 is automatically added.
These plugins are required to be set up **before** you start work on any assignments in this course!

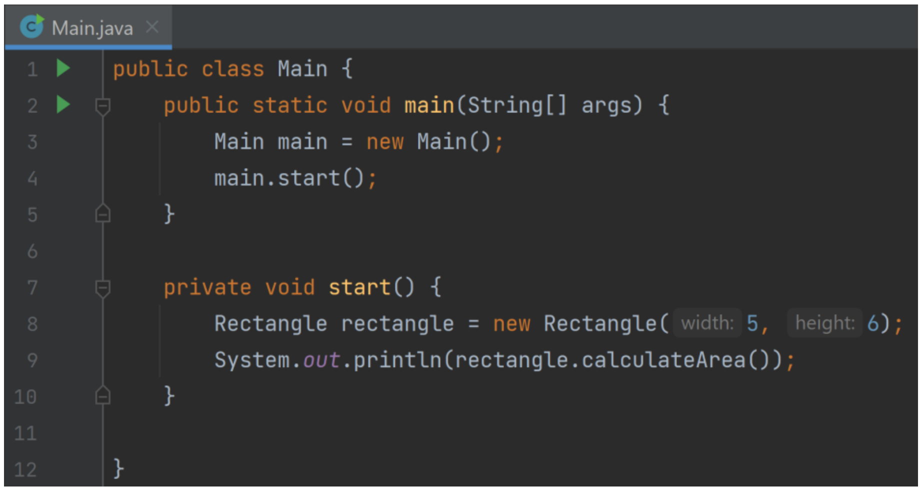
Let’s start by creating a simple Object with a field and a method:



```java
public class Rectangle {

    int width;
    int height;

    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    int calculateArea() {
        return width * height;
    }
}
```

Fields are the **data** that a class has. In this situation, a Rectangle **has** a width and a height. It’s important to note that a Rectangle **doesn’t have** an area field. Area is a **calculated** value. This is a critical software design distinction. If we had a field called area, then if width or height was ever changed, we would have to keep area updated along with it. If you forgot – that’s a bug! Since this happens all the time, it’s better design to not make fields for **calculated values**. Make **methods** that start with the word “calculated” in their name instead, to indicate that is a calculated value.
Methods are the **behaviors** that a class has. In this situation, a Rectangle can calculate its own area. Our Rectangle also features a **constructor** which sets the fields. In Java, using “this.field = field” is typical for a constructor or a setter method **only**. Don’t use “this.field” or “this.method()” outside a constructor or a setter.

Throughout the lab I will also demonstrate our classes being used by a very simple program:

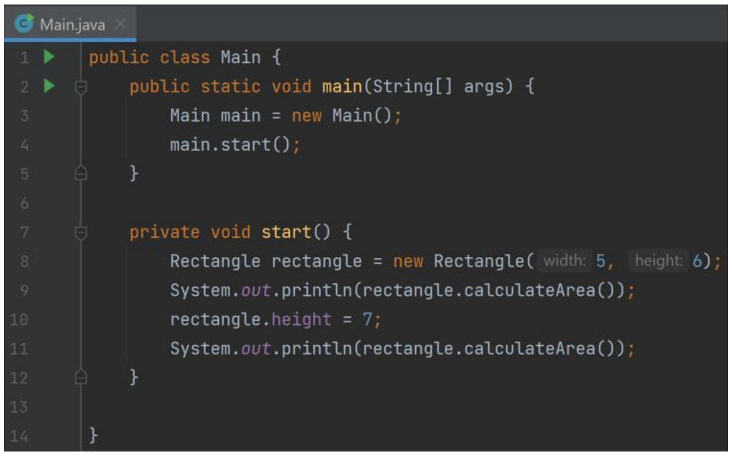


```java
public class Main {
    public static void main(String[] args) {
        Main main = new Main();
        main.start();
    }

    private void start() {
        Rectangle rectangle = new Rectangle( width: 5,  height: 6);
        System.out.println(rectangle.calculateArea());
    }

}
```

You might be wondering why bother to create “new Main()” in the screenshot above. Well, the alternative would be to make the start() method *static*. However, static methods are an **antipattern**. This is not something I teach heavily in this course, but if you’d like to learn more, the internet is full of resources on the topic (and later courses in our program cover this more thoroughly). What you must know: train yourself to avoid using static methods. Later in life you will learn design patterns that require the use of static – they are the only exception. But don't worry about that for now. If you ever want to use a static method, stop yourself and find another solution, as I did above. This is a minor learning outcome for this course.

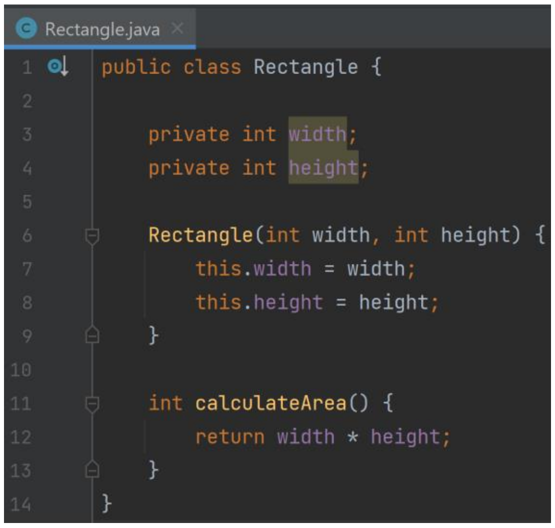
In the start() method, we create a Rectangle with a width of 5 and a height of 6. Our area gets calculated and printed correctly if you run the code.

However, even our basic Rectangle has a problem, and this introduces the first major learning outcome of the lab: **encapsulation**! If we wanted to, we could manipulate width or height as such:


Main.java

```
1   public class Main {
2       public static void main(String[] args) {
3           Main main = new Main();
4           main.start();
5       }
6
7       private void start() {
8           Rectangle rectangle = new Rectangle( width: 5,  height: 6);
9           System.out.println(rectangle.calculateArea());
10          rectangle.height = 7;
11          System.out.println(rectangle.calculateArea());
12      }
13
14  }
```

Since our main method (lines 2-5) will never change, I will drop it from future screenshots throughout this lab.
This is important: line 10 is **wrong**. Don't get me wrong – the code works. It compiles. It runs. The result is correct. But it violates **encapsulation**. Manipulating a field by accessing it directly means that the field is being exposed. This highlights an issue with our Rectangle design. It is important to make fields **private** to prevent this from ever happening:


Rectangle.java

```
public class Rectangle {

    private int width;
    private int height;

    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    int calculateArea() {
        return width * height;
    }
}
```

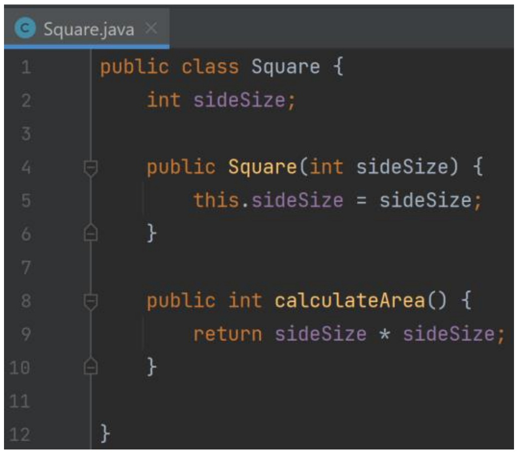
We add the modifier "private" on lines 3 and 4. This makes sure that no developer can ever access those fields directly. If we look at our start() method now, you'll see that previous line 10 does not compile anymore. We follow encapsulation. Why is this so important?? Software design. This course is the beginning of your journey of making your own design decisions. Fields can be set using setters and retrieved using getters. But the design question, for each field, is should they be? Should width be able to be changed after the Rectangle is instantiated? If yes, add a setter for it. Will width ever need to be retrieved for use? If yes, add a getter for it. **Do not just add setters and getters by habit**. They are design decisions.

No setter example: A University has Students and Students have a unique 8 digit ID. That ID is assigned when the Student is instantiated and is never allowed to change. Creating a setter for this would be **incorrect design.** Allow it to be set once in the constructor, but do not create a setter for it. And remember to mark it **private**, so no one will ever be able to change its value.

No getter example: We don't actually need access to a Rectangle's width or height. We only need its area. So we will abstain from adding getters for width and height. Adding a getter for a field tells the reader of your code "this field is used elsewhere by other classes." And that would be wrong with our current design – they are not being used anywhere aside from getArea().

Next, let's move on to our next major learning outcome: **inheritance**.

Let's say we want to add a Square to our software system. We might code something like this:

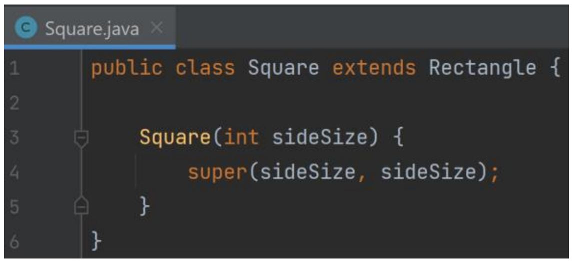

Square.java

```java
public class Square {
    int sideSize;

    public Square(int sideSize) {
        this.sideSize = sideSize;
    }

    public int calculateArea() {
        return sideSize * sideSize;
    }

}
```

At this point in the lab, we're going to start playing a game called "spot the duplicate code." Compare this code to our Rectangle.

Before we made this Square, we should have remembered the first step of any design decision: think! What is a Square in relation to a Rectangle? Remember that inheritance means an "is a" relationship. For example, a Car is a Vehicle. Is a Rectangle a Square? No. Is a Square a Rectangle? YES! So, Square can inherit from Rectangle to gain many benefits. A child class will inherit the parent's fields and method implementations.

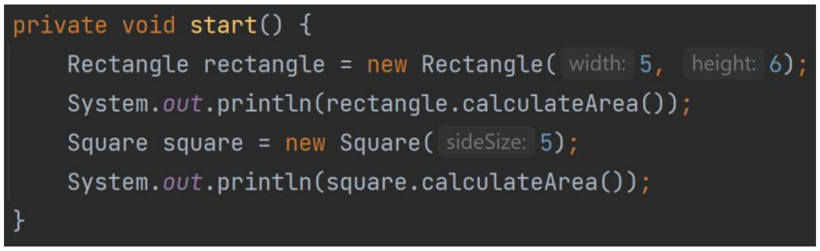
If we use inheritance to our advantage, here's what our Square would look like:



Square.java

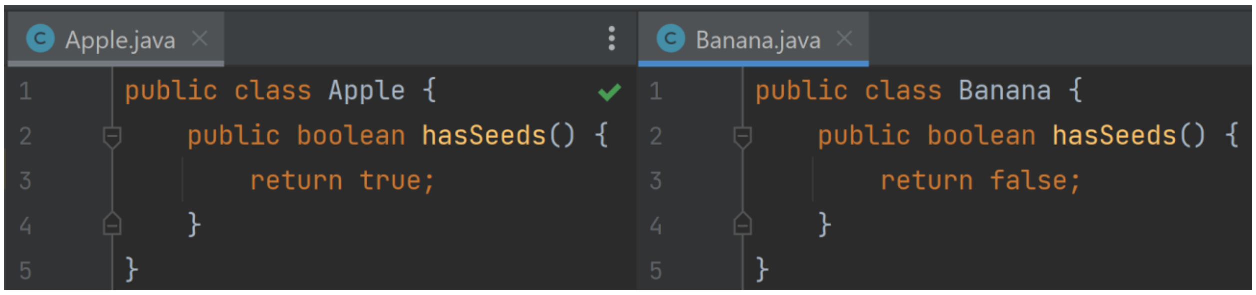
```java
public class Square extends Rectangle {

    Square(int sideSize) {
        super(sideSize, sideSize);
    }
}
```

Much smaller, right? That’s because we’re using the power of inheritance to remove the need for **duplicate code**. “Square extends Rectangle” means “Square is a Rectangle”. The Square gains width, height, the constructor, and getArea() from its parent. It can use the parent’s constructor to create itself with a width and height being the same value. We can use our Rectangle and Square classes:



```java
private void start() {
    Rectangle rectangle = new Rectangle( width: 5,  height: 6);
    System.out.println(rectangle.calculateArea());
    Square square = new Square( sideSize: 5);
    System.out.println(square.calculateArea());
}
```

Let’s continue our game. Please see the below screenshot:



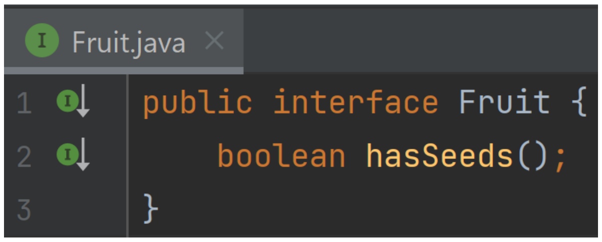
Apple.java

```java
1  public class Apple {
2      public boolean hasSeeds() {
3          return true;
4      }
5  }
```

Banana.java

```java
1  public class Banana {
2      public boolean hasSeeds() {
3          return false;
4      }
5  }
```

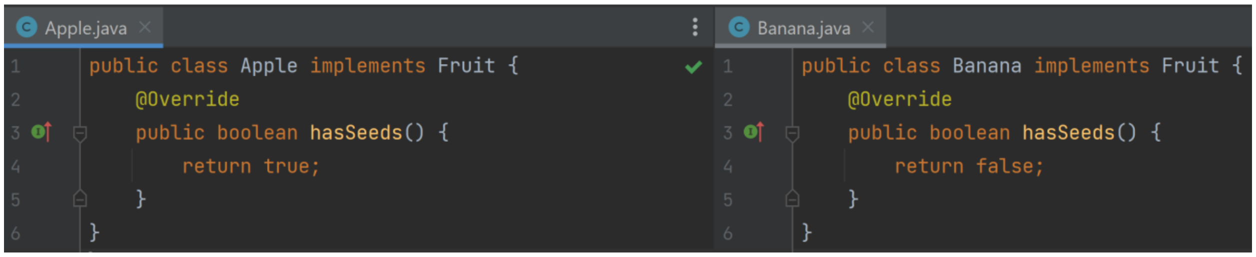
Apples have seeds. Bananas are seedless. (Google told me so). Where’s the duplicate code?
It might not seem like much, but line 2 is in fact a duplicated line of code. This example brings us to our next major learning outcome: **abstraction.**
In Java, there are 2 types of **abstractions,** abstract classes and interfaces.
Interfaces: use these when there are common methods, but not common code inside the method and no common fields.
Abstract classes: use these when there is even a simple common field or a single common line of code.
In this case, line 3 is not the same and both classes have no fields. So the correct **abstraction** type is an interface:



Fruit.java

```java
public interface Fruit {
    boolean hasSeeds();
}
```

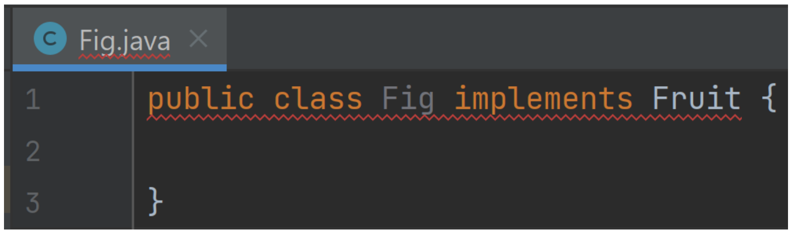
Now that we have a parent abstraction, we can use it in our children. Note that interfaces are **implemented:**



```java
1  public class Apple implements Fruit {
2      @Override
3      public boolean hasSeeds() {
4          return true;
5      }
6  }
```

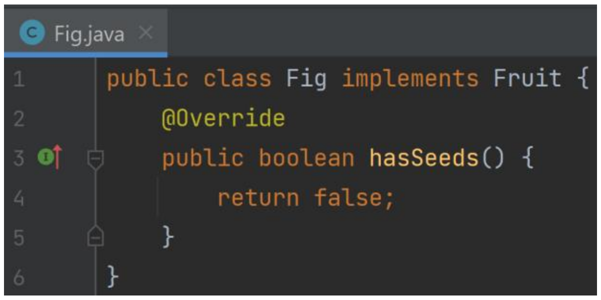
```java
1  public class Banana implements Fruit {
2      @Override
3      public boolean hasSeeds() {
4          return false;
5      }
6  }
```

Both Apple and Banana now **implement** Fruit. Indeed line 3 is still exactly the same, but the benefit is having an abstraction type for maintainability of your software system later. For example, let's create a new Fruit type:



```java
1  public class Fig implements Fruit {
2
3  }
```

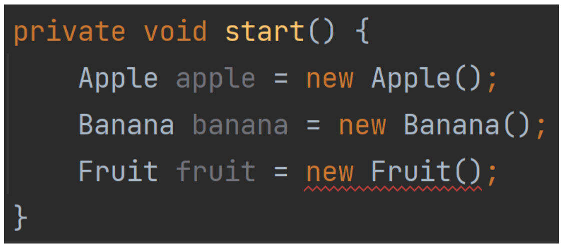
Because you have a parent abstraction, any decent modern IDE will help you fill in the blanks for the class. Hover over the red line and click "implement methods" (if a popup window comes up, just click OK). The IDE will do the rest of the work for you:



```java
1  public class Fig implements Fruit {
2      @Override
3      public boolean hasSeeds() {
4          return false;
5      }
6  }
```

So while line 3 is still repeated in every Fruit, **it's not a line of code you ever have to type out again**. And there's a lesson here. One of my favorite Ron Jeffries quotes: "Reduced duplication, high expressiveness, and **early building of simple abstractions**." The lesson here is that if we created the Fruit class **first**, we never would have had to copy paste that line of code when creating Apple and Banana. The longer you wait to create an abstraction, the more painful the **refactoring** process will be.

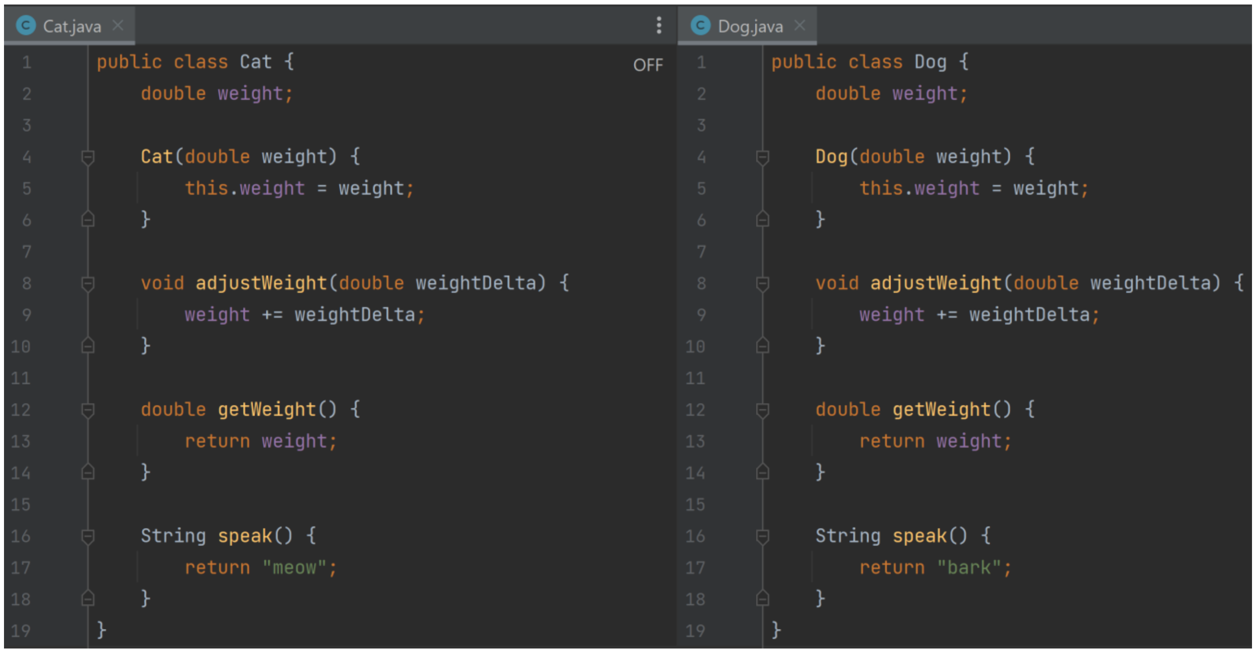
Abstractions are a type of **inheritance.** There's a critical difference, however. With simple inheritance, like Square and Rectangle above, both of these classes can be instantiated. Abstractions can't be instantiated. Abstractions are conceptual representations. What would you get if you walked into a grocery store and asked "can I get 1 fruit please?" That's not a thing. Not really. You can instantiate an Apple. You can instantiate a Banana. You can't instantiate a Fruit. This is critical to software design – to stop other developers from instantiating a concept that should not exist. Here you can see that instantiating a new Fruit does not compile:



```java
private void start() {
    Apple apple = new Apple();
    Banana banana = new Banana();
    Fruit fruit = new Fruit();
}
```

Since interfaces are a type of abstraction, they are never allowed to be instantiated. This is correct design.

Let's move on to a more complex example of our game. Please see the below screenshot:



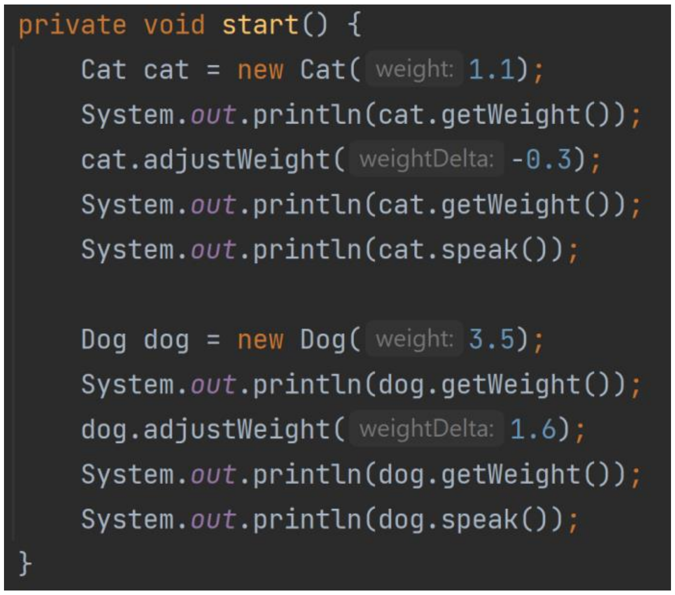
Cat.java

```java
1  public class Cat {
2      double weight;
3
4      Cat(double weight) {
5          this.weight = weight;
6      }
7
8      void adjustWeight(double weightDelta) {
9          weight += weightDelta;
10     }
11
12     double getWeight() {
13         return weight;
14     }
15
16     String speak() {
17         return "meow";
18     }
19 }
```

Dog.java

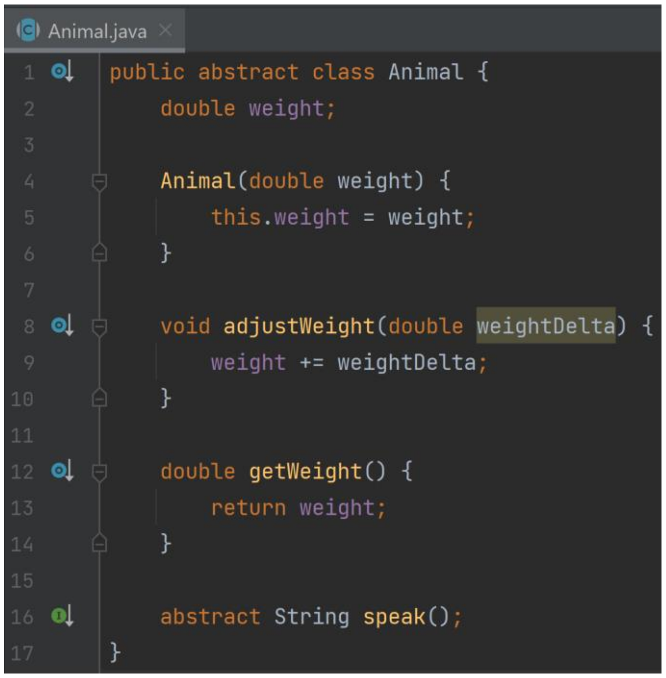
```java
1  public class Dog {
2      double weight;
3
4      Dog(double weight) {
5          this.weight = weight;
6      }
7
8      void adjustWeight(double weightDelta) {
9          weight += weightDelta;
10     }
11
12     double getWeight() {
13         return weight;
14     }
15
16     String speak() {
17         return "bark";
18     }
19 }
```

Hopefully you can easily spot most of the duplicated code. Let's go through:
Line 2 – exactly the same field.
Line 4 – exactly the same constructor code, except for the Object type.
Line 8-10 – exactly the same method definition and code.
Line 12-14 – exactly the same method definition and code.
Line 16 – exactly the same method definition. But the code on line 17 is not common.
Should we use an abstract class or an interface to create our **abstraction** and remove our duplicate code?
Also, in case you were curious, this is how these classes can be used:



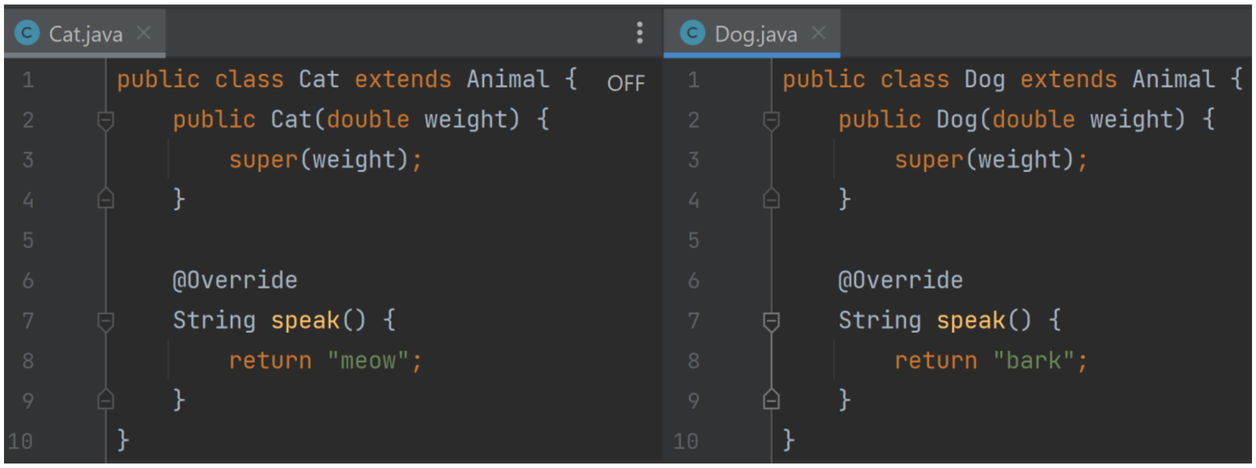
```
private void start() {
    Cat cat = new Cat( weight: 1.1);
    System.out.println(cat.getWeight());
    cat.adjustWeight( weightDelta: -0.3);
    System.out.println(cat.getWeight());
    System.out.println(cat.speak());

    Dog dog = new Dog( weight: 3.5);
    System.out.println(dog.getWeight());
    dog.adjustWeight( weightDelta: 1.6);
    System.out.println(dog.getWeight());
    System.out.println(dog.speak());
}
```

An **abstract class –** because there are common fields and common lines of code inside methods.



```
public abstract class Animal {
    double weight;

    Animal(double weight) {
        this.weight = weight;
    }

    void adjustWeight(double weightDelta) {
        weight += weightDelta;
    }

    double getWeight() {
        return weight;
    }

    abstract String speak();
}
```

Nearly all our duplicated code can make its way into the parent Animal. The only thing that is unique to each animal is how they speak, so we make that an **abstract method**. That's similar to how interfaces define **common behavior** without actually defining the exact behavior itself.



```java
public class Cat extends Animal {
    public Cat(double weight) {
        super(weight);
    }

    @Override
    String speak() {
        return "meow";
    }
}
```

```java
public class Dog extends Animal {
    public Dog(double weight) {
        super(weight);
    }

    @Override
    String speak() {
        return "bark";
    }
}
```

Look how clean! Note that abstract classes are **extended**, not implemented like interfaces are.
A common mistake new developers make is leaving "double weight" in the children (Cat/Dog) as a field. This is in fact a **mistake**. Many of you may even make this mistake throughout your journey in this course. Remember that a child class will inherit fields and methods from its parent. So, creating a "double weight" field in the child would **overwrite that**, and would cause some very nasty **bugs.**

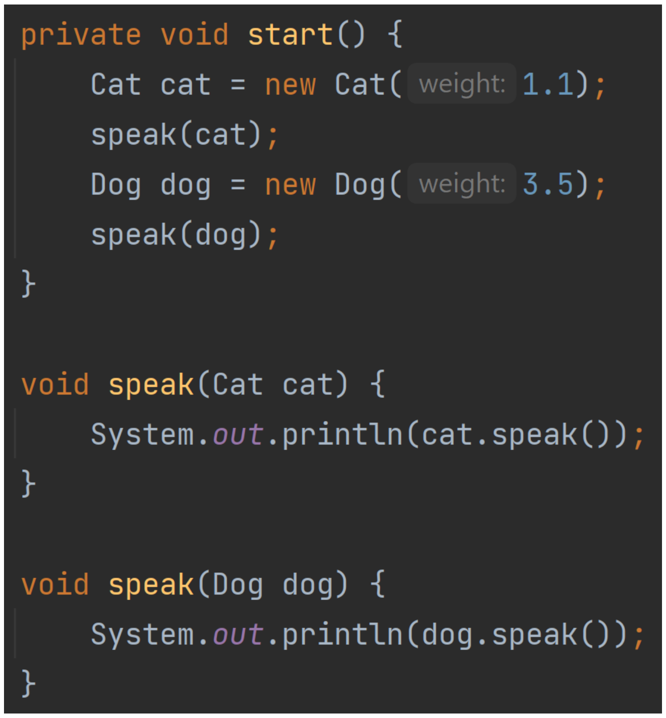
Remember throughout this course that both **abstract classes** and **interfaces** are forms of **abstraction** in Java – they can't be instantiated. The term **abstraction** is language-agnostic. In Java, these are the 2 types. Other languages behave differently.

Our goal is always to remove duplicate code. When doing so, we have options. If the **parent** is not a concept but rather something that can and should be instantiated, then simple inheritance is the way to go. If the **parent** is a concept that should not ever be allowed to be instantiated, then **abstraction** is the way to go. In Java, we can choose between **interfaces** and **abstract classes**. Java has a rule that a class can only **extend** one other class. Whereas a class can **implement** as many classes as it wants to. For this reason, **interfaces** are useful if you only have common behaviors to abstract. An abstract class provides shared method implementation while an interface provides an outline. If an outline is not enough, specifically if the **children** share common fields and method code, then use an **abstract class**.

These are important design decisions to make. You will be graded on the decisions you make.

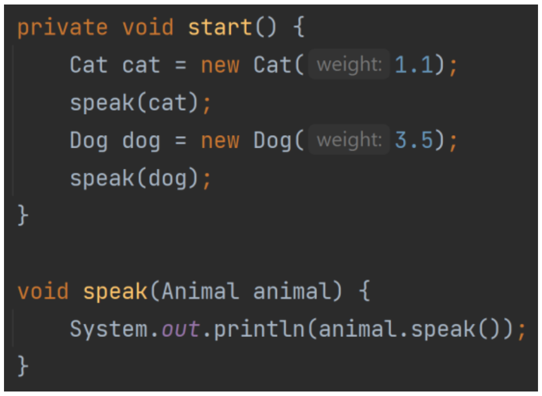
The last thing to cover is **polymorphism**.

The first thing I want to do is demonstrate simple polymorphism. Let's create a method that takes an Animal and makes it speak:



```java
private void start() {
    Cat cat = new Cat( weight: 1.1);
    speak(cat);
    Dog dog = new Dog( weight: 3.5);
    speak(dog);
}

void speak(Cat cat) {
    System.out.println(cat.speak());
}

void speak(Dog dog) {
    System.out.println(dog.speak());
}
```

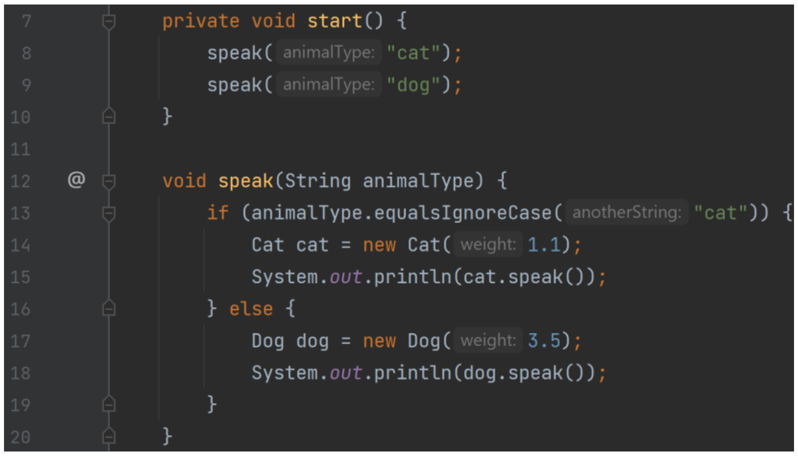
Did you think we were done playing the game? We're never done. Throughout this course, if you are not constantly playing "spot the duplicate code", you will consistently lose portions of your assignment grade! Hopefully you all spotted that the 2 methods are basically… the same. This is where we can use our OOP basics reduce our duplicate code. (Note, the exact weight is irrelevant – weight could be any value).
What's the parent to both Cat and Dog? Here ya go:



```java
private void start() {
    Cat cat = new Cat( weight: 1.1);
    speak(cat);
    Dog dog = new Dog( weight: 3.5);
    speak(dog);
}

void speak(Animal animal) {
    System.out.println(animal.speak());
}
```

The code above does the same thing. By using the **parent**, we allow our code to behave **polymorphically.**

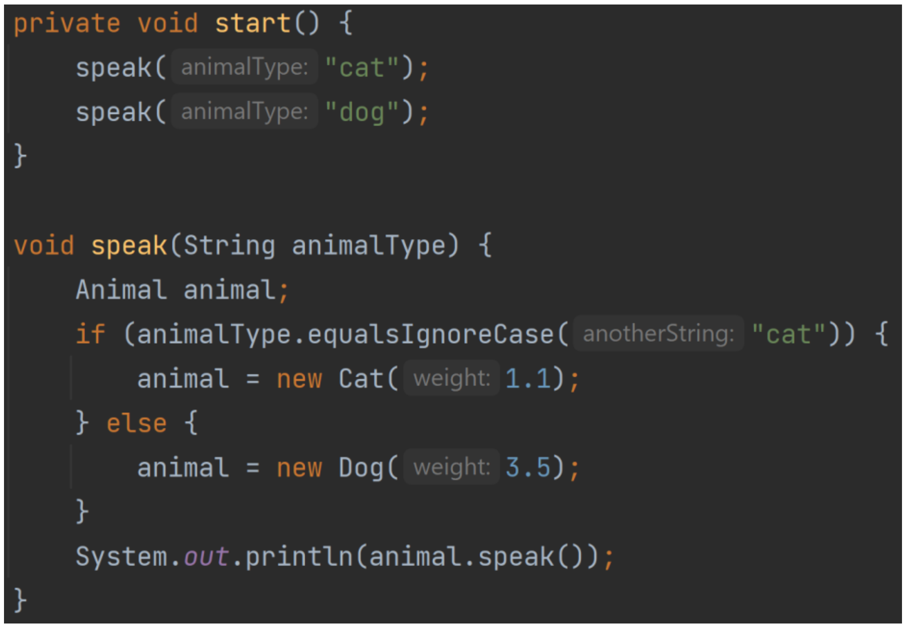
Let’s say we coded a simple method that takes a String and uses it to print whatever the animal speaks:



```
 7      private void start() {
 8          speak( animalType: "cat");
 9          speak( animalType: "dog");
10      }
11
12  @   void speak(String animalType) {
13          if (animalType.equalsIgnoreCase( anotherString: "cat")) {
14              Cat cat = new Cat( weight: 1.1);
15              System.out.println(cat.speak());
16          } else {
17              Dog dog = new Dog( weight: 3.5);
18              System.out.println(dog.speak());
19          }
20      }
```

Again, can you spot the duplicate code?

It’s not much, but line 15 and 18 are awful similar… if only I could remove that duplication by using the **parent:**



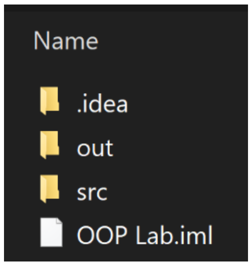
```
private void start() {
    speak( animalType: "cat");
    speak( animalType: "dog");
}

void speak(String animalType) {
    Animal animal;
    if (animalType.equalsIgnoreCase( anotherString: "cat")) {
        animal = new Cat( weight: 1.1);
    } else {
        animal = new Dog( weight: 3.5);
    }
    System.out.println(animal.speak());
}
```

Woo! We did it. By using the **parent** we are able to polymorphically use the Animal class, regardless of which type it ends up being. Please note that in the example above **Animal** is an abstract class. This code could work exactly the same if it was an interface instead. Those are the only 2 types of **abstraction** in Java.

Submission Requirements

When you are done, your submission must be a **zip** (not rar or any other type) file as documented below. This will be the same process you follow for all assignment submissions.

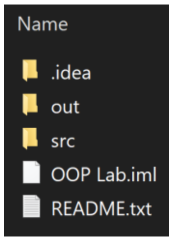
Right click on the project -> Open in -> Explorer, or just find the directory where your project resides. You can now right click on your project folder "OOP Lab" and create a zip using any tool of your choice. The folder structure inside should look like this:



```
Name
.idea
out
src
OOP Lab.iml
```

If you are on a Mac, the .idea folder will be hidden by default. Type "CMD + Shift + ." to show this folder in Finder.

If you have anything additional to say, please place a README.txt in that directory, parallel to TDD.iml, as such:



```
Name
.idea
out
src
OOP Lab.iml
README.txt
```

Zip file name should consist of your name and id (ex: A1-Boris-Valerstein-bv49.zip).

Grading Rubric:
If you do not submit a zip file or follow the zip file naming convention throughout this course, your submission will not be graded. **Straight 0, fair warning!**

- 50% plugins
    You did not set up the plugins correctly ***before*** you started development
-50% Completed the work
    All the Java files should be in the submission