

In this lesson we will just about finish up with object-oriented programming. There will be a couple small things left to cover in lesson 15 but you're almost done. The two big topics in this lesson are **abstract classes** and **interfaces**. Both of these concepts are somewhat similar to what you learned in the inheritance lesson, but they have slightly different use cases.

One quick note – a class can only inherit from, or extend, one class. E.g. you can't do `class Dog extends Animal, extends Pet, etc.` You have to choose one.

Watch this video – it does a good job of explaining abstract classes. Pay close attention to the example with the array at the end. Being able to group different kinds of objects together and perform operations on all of them because it's known they extend the same base class is critically important. This ability to group things based on parent class (or interface) is one of the biggest advantages to using inheritance, abstract classes, and interfaces, as opposed to just typing out the methods, but NOT creating the parent class/abstract class/interface. Here's the video:  
<https://www.youtube.com/watch?v=pt1S11yX-7k>

Here's a link to the other site we've been referencing's explanation of abstract classes. It's not bad (I think the shape example is a good one to show how sometimes you need to define that methods exist, but can't logically be implemented in the parent class), but you'll have to read through the code a little bit. For abstract classes, read sections 5.1 through 5.3.  
[https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b\\_OOPInheritancePolymorphism.html#zz-5](https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html#zz-5)

For interfaces, watch this video: <https://www.youtube.com/watch?v=NnZQ-C0x4hs> and read sections 5.4 through 5.10 of the document linked above.

There are a few more important points about interfaces that warrant some extra explanation. First and foremost, interfaces are similar in concept to abstract classes, so if you've noticed some overlapping concepts, you're right. But they have slightly different applications. Classes can *implement* any number of interfaces, whereas like said, they can only *extend* one. This is useful because classes can *be* some individual thing, but *do* multiple things. For example, a class `Dog` might extend `Animal`, but implement `Pettable`, `CanWalk`, `CanMakeNoise`, etc. In Java people often name interfaces either by ending them with `able`, or prefacing them with `Can`. You'll see this if you look at some of the interfaces in the RomiReference project, for example the `CommandScheduler` class implements two `able` interfaces:

```
38 public final class CommandScheduler implements Sendable, AutoCloseable {
```

You might notice that `pet`, `walk`, and `bark` are all verbs. So are `send` and `auto-close`. This isn't a coincidence. As stated, inheritance/extension is often used to specify what something is, and interfaces to designate what it can do. If you're confused about whether inheritance or an interface is a better option for any given task, this is probably the best way to think about it. Is your class a more specific instance of something, for example a dog being an animal, or is it something that can do something, but that thing isn't necessarily related to what it is? For example, a ton of things make noise, but that's not something that is related to being a dog (or an animal.) In addition to a dog, a robot could also implement a `CanMakeNoise` interface. (If your Romi has ever gotten low on batteries, you may have noticed that it will start singing.)

This might sound a bit repetitive, but it's important to understand the advantages of interfaces, abstract classes, and inheritance as opposed to just typing out all the methods you need in each class. One big advantage is organizationally, it makes your code easier to think about and organizes your classes and methods. Especially when you're working with other people on a project, this makes life easier. However, it also gives you the ability to write more functional code.

Interfaces and parent classes (either abstract or concrete) are types that the compiler recognizes. The compiler is the program that reads your code and turns it into something that a computer or robot can execute. The compiler recognizing a type lets you group by that type – for example, you can make an array of `GamePieceManipulator` objects, even if each of those objects belongs to a different class. (E.g. one belongs to `WobblyArm`, one belongs to `BrokenShooter`, etc.) Even if `WobblyArm` and `BrokenShooter` have the exact same methods, if you make an array of `WobblyArm` objects, you can't add any `BrokenShooter` objects to that array. But by making the array of `GamePieceManipulator` objects, you can add both those types, and use all the common functionality. So the interface or parent class is where you define what the common functionality is. You've heard the term *polymorphism* – this is one of the prime examples. If this is confusing, go back and watch 9:04 to 13:37 of the abstract classes video from this lesson. You'll gain better understanding over time, but that at least illustrates the concept.

Lastly, read sections 6.1 and 6.2 here:

[https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b\\_OOPInheritancePolymorphism.html#zz-6](https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html#zz-6)

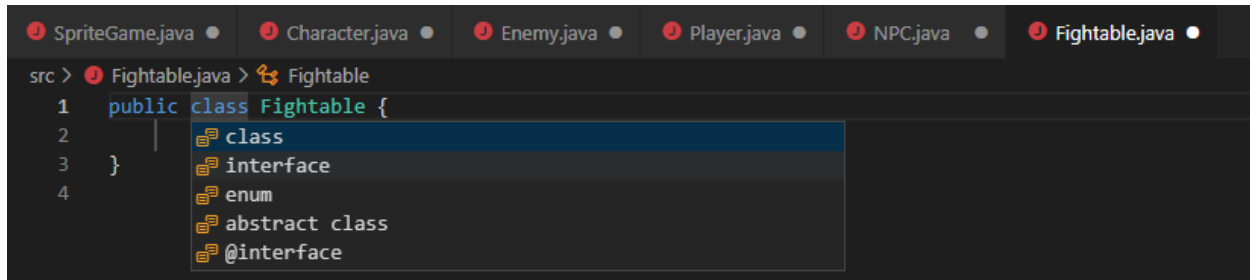
"Encapsulation" is a fancy word but the concept should look familiar (think getters and setters). "Coupling" is new but the explanation they give makes sense. Same with "cohesion", but phrased differently, cohesion means that if your class shouldn't represent multiple things – if it does, it should be split up into multiple classes until each class only represents one thing. For example you would not make a class called `DogAndCat`, you would make a `Dog` class and a `Cat` class.

--

For this lesson you'll set up classes and interfaces for a two-dimensional sprite game. We're not actually going to program the game (although you could do this on your own if you want), but we'll use it as an example of how these concepts interact.

1. Create a new desktop Java project called `SpriteGame`. If you do this in VSCode and VSCode adds an `App.java` file, you can delete that file. Add the following classes:
  - a. `SpriteGame` (your main method will go here)
  - b. `Character`
  - c. `Enemy`
  - d. `Player`
  - e. `NPC`
2. Create two *interfaces*. You can do this the same way you create a class – right click, add a Java file, and type in the name of the file, followed by `.java`. VSCode will default to creating a class but it will give you the option to create an interface right after you create the file. You can click "interface" in the drop down to change it. Alternatively, simply replace the word "class" with the word "interface". Name your two interfaces as follows:

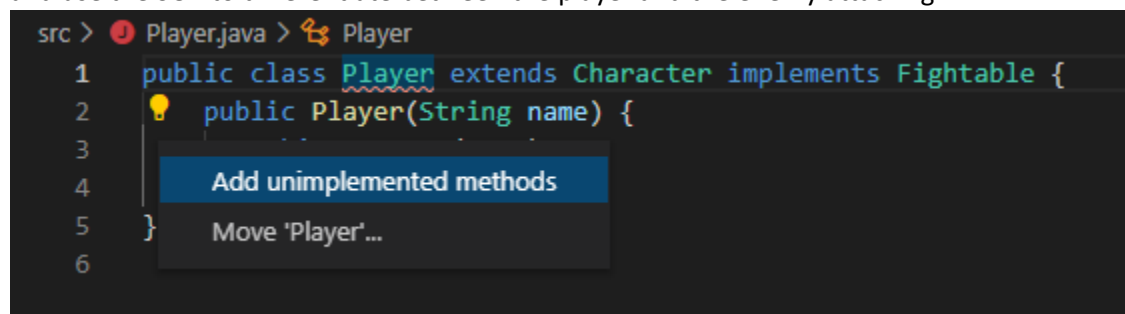
- a. Fightable
- b. Movable



- 3. Let's start by fleshing out what our interfaces do. Then we can ensure that our classes comply. This is a handy way of thinking about/designing your program. Give the fightable interface the following methods. Remember, in interfaces you just create stubs with semicolons, you don't fully define the method.
  - a. `public void attack();`
  - b. `public void takeDamage(int damage);`
  - c. `public int getMaxHp();`
  - d. `public int getHpRemaining();`
- 4. Now let's define movable. Since we're making a 2d sprite game, "movable" just means up, down, left, and right, or x and y coordinates. Let's make the following methods:
  - a. `public void moveNorth();`
  - b. `public void moveEast();`
  - c. `public void moveSouth();`
  - d. `public void moveWest();`
  - e. `public int getXPosition();`
  - f. `public int getYPosition();`
- 5. Now let's set up our Player and Enemy classes so that they can fight each other. We have a "Character" class, and both the player and enemies are characters. So let's make Player and Enemy extend Character. While we're at it, make NPC do the same.
- 6. Right now extending Character doesn't do much, since there's nothing in Character. So let's flesh that out. All characters are going to have a name, so let's make a private String class field called "name". Since this is private, let's also add a method called `printName` that SOP's the name field. Let's also add a method called `setName` that takes a string and sets the name field.
- 7. Since we added a name field and a couple methods to our Character class, and Player, Enemy, and NPC all extend Character, we have access to the new Character methods in those classes. Let's create constructor methods for these three subclasses that take advantage of this to set the name. In each of these classes, create a constructor method that takes a string and calls the `setName` method. **Protip:** while in a subclass, you can type "this." to bring up an intellisense menu that includes parent class methods. So if you type "this." in Player or Enemy, the Character class `setName` will appear on the menu.
- 8. Now that our Character class does something, let's go to our main method in SpriteGame and write some code to test that Character is working. Create a player object, two enemy objects,

and an NPC object using the constructors that set the names. You can name them whatever you want.

9. Let's create a method that outputs the names of all our characters, even though we have different kinds of characters. To do this we're going to need to store our characters in an accessible way. In your main method, create an `ArrayList<Character>` and add your four characters to it. You will need to import `ArrayList` from `java.util`.
10. Now create a method called "outputCharacterNames" that takes an `ArrayList<Character>` as a parameter, and prints all their names. This will be a public static void method in your `SpriteGame` class. (We'll talk about static in the next lesson.) Go ahead and add a call to your new method in your main method, and confirm that it outputs three names. The method is working on objects of different classes, pretty cool!
11. Now that we have characters that are working, let's get back to our interfaces. NPCs don't fight, but the player and enemies do. So go into player and enemy and make them both **implement** `Fightable`. This will give you a syntax error on the class name. Mouse over and select "add unimplemented methods." Easy! Since we're not fully programming this game, we don't actually care about implementing these methods for now. But, just so we can demo that they run, add a SOP to the `attack()` method in each that says "Now attacking." You could go one step further and use the SOP to differentiate between the player and the enemy attacking.



The screenshot shows a code editor with a file named `Player.java`. The code defines a `Player` class that extends `Character` and implements `Fightable`. A syntax error is highlighted on the `implements Fightable` line. A context menu is open, showing the option "Add unimplemented methods" which has been selected.

```
src > Player.java > Player
1 public class Player extends Character implements Fightable {
2     public Player(String name) {
3
4
5     }
6
```

12. This example illustrates how interfaces are different than subclasses. Enemies, Players, and NPCs are all characters, but they don't all fight. So if we had put the `Fightable` methods inside of `Character`, we would have a bunch of methods for NPCs that wouldn't make any sense to call. And if we had put the `Fightable` methods in *both* `Player` and `Enemy`, we'd have code duplication.
13. Go into your main method and create another `ArrayList`, this time of type `Fightable`, and put your player and enemies in there. Make another method like you did with `outputCharacterNames` but instead this one calls the `Fightable` `attack` method.
14. We want our player to move, so make it implement `Movable` in addition to `Fightable`. Again, you can have VSCode implement the methods for you, in particular since we're not going to implement them for real. Similar to how we did not make NPCs `fightable`, this shows how you can use interfaces to extend the functionality of certain parts of the class hierarchy, but not others.
15. Let's do an example of how interfaces can be applied to completely unrelated classes. Create a class called "DestructibleObject".
16. `DestructibleObjects`, at least in this game, are not "Characters" – they are walls, rocks, etc., that can be destroyed. So they won't move, don't have a name, etc., and should not inherit from `Character`. However, they can be destroyed, so they *are* `Fightable`. Make `DestructibleObject` implement `Fightable` and create the method stubs.

17. In the “attack” method for DestructibleObject, let’s make it a little more clear that it’s not a normal player or enemy. Instead of SOP’ing “Now attacking.”, SOP “Object destroyed!”.
18. In your main method, create a DestructibleObject object, and add it to your fightables array. Then re-run your program to see the output.
19. Almost done, but let’s differentiate our characters just a little bit. So far, we’ve instantiated objects of Player, Enemy, and NPC types, but never a base “Character” object. This is because Character is designed just to give attributes and methods to these subclasses, not exist itself. So let’s go ahead and make it abstract.
20. Making Character abstract doesn’t give us any errors, because we haven’t added any abstract methods yet. Let’s add a new abstract method that all our classes have to implement. We’ll give our characters some personality by creating a method called “introduce()”. This will be public, abstract, and void. Create this method. Remember, an abstract method is created similarly to how they are done in interfaces.
21. If you complete step 20 correctly and save, you’ll get syntax errors in your three character subclass files. Go to one of these files and mouse over the error. It will look familiar from when we were implementing interface. Have VSCode implement the method in all your files, but once it creates the stub, go into the method yourself and add a SOP that gives some context to the character. For example, the player character could say “I save villages!”, the NPC could be “I sell protective footwear!”, and the enemy could be “I mindlessly follow AI without having clear motives, while letting the player grind experience!”
22. In your SpriteGame class, create one more method, like you did for the character names and the fightables, to output each character’s introduction. Note that you already have an ArrayList with all your characters to use.

That’s all we’ll do with this. If you wanted to complete the game, you’d have to make all the methods that implement the interfaces *actually do things* instead of just being empty. However, this shows how you can use interfaces, inheritance, and abstract classes to set up hierarchies and relationships, and define what classes *are* and what classes *can do*. It also shows how you can use polymorphism to group objects in a way such that you can use functionality that is common to those classes, even though they are of different types.