Lesson 5

* So far, we’ve been doing Procedural Programming. This works for small things, but when you build a large application it becomes difficult to maintain.
  + The most used methodology around how to write and shape code is called Object Oriented Programming (OOP).
  + It may seem difficult at first, but it is actually how humans already interact with objects in real life.
    - Objects have attributes that define them, and things they can do.
      * A DVD player has attributes such as
        + Height
        + Weight
        + Width
        + Color
      * And things it can do such as
        + Play
        + Fast forward
        + Rewind
        + Pause
      * A DVD player interacts with other objects as well that each have their own attributes and things they can do. For example, the main object a DVD player interacts with is a DVD which has attributes such as:
        + Movie length
        + Image
        + Size
  + When we think of the things we are programming as individual objects that can interact with other objects, we can start to shape our code in a more maintainable way using OOP.
  + OOP offers a lot, the four main pillars of OOP are Abstraction, Encapsulation, Inheritance, and Polymorphism
    - **Abstraction**: exposing essential features of an entity while hiding details that don’t matter as much or change from implementation to implementation.
      * An example of abstraction we are used to in real life is a triangle. Triangle is an abstract concept. Each triangle is different, but the things they share in common are the abstraction of a triangle. Those things include 3 points, 3 lines, and 3 angles adding up to 180 degrees, among other trigonometric properties.
    - **Encapsulation**: hiding the inner workings of your code so that certain methods are only accessible in certain areas of the code.
      * An example is when you drive a car, you don’t have to know exactly what the engine is doing to know how to drive. That functionality has been encapsulated so that you, as a user, don’t have to worry about it.
    - **Inheritance**: This one is probably the most familiar concept already. Just as we inherit traits from our parents and ancestors, OOP allows objects to inherit from parent objects, or classes to be more precise.
    - **Polymorphism**: Means that something can take different forms.
      * For example, an Animals communicate differently based on which animal they are. A dog barks, a cat meows, and a bear growls. All are instances of an animal communicating, but the communication method is polymorphic based on which animal is performing the action.
  + Aside from these four pillars of OOP, we need to learn how to structure our code. The most common Java construct for creating OOP code is called a Class.
  + You have heard this term used hand in hand with the term Object up to this point, but there is a very specific difference between a Class and an Object.
    - A Class is like a blueprint of an Object, not the actual Object itself.
    - An Object is an instance of a Class.
    - Just like you can create many different houses from the same blueprint, you can create many different Objects from the same Class.
* We have used Classes before that come with Java such as String, StringBuilder, Scanner, Array, ArrayList, and others. Each of these classes holds some specific data and performs some specific operations or methods. But sometimes, there isn’t already a premade class for the way we want to structure our data and methods.
  + We create a class with the class key word followed by an identifier or name for the class. In Eclipse, it automatically does this for you when you create a new class.
  + Use Pascal Case naming when naming classes.
  + Up to this point, all of the classes we have created contained a main method to be the entry point of the code. However, in large projects there is only one entry class, the rest do not have this method.
  + Inside the class we can create methods just like we have, and have class level properties or attributes.
  + Let’s create a class that represents a Student.
    - firstName, lastName, phoneNumber, grade
    - introduce() method
    - Remember that the class is the blueprint, so we wouldn’t actually assign values to these fields yet, each instance of the class will have it’s own specific values for these fields. This allows us use this class to create as many student objects, or instances of Student as we want.
    - If, however, you need a variable to be shared by all instances of a class, in other words to have a class level property, you would add the static keyword to the field.
    - Now let’s create a few instances of the class in our application class main method.
    - The type of the variable will be Student, matching the class name, and the value will be a new Student();
    - Then we can assign values to the student’s properties using dot notation.
    - This can be very tedious to add all the values one line at a time, to mitigate this we can add a special method to the Student class called a **constructor.**
      * This method is what get’s called when we create an instance of a student.
      * We use the `this` keyword to specify the field that belongs to the instance. So, if we have three different students, this.firstName would represent the first name of the specific student in question. Just like we all have our own first name.
    - Some students may not have all the information needed to use the constructor. We can overload the constructor.
    - We can then call the introduce method on each of the students using dot notation.
* Encapsulation - In the previous video, we were able to access object properties using dot notation. While this works, it is considered very bad practice in OOP. We need to encapsulate an objects data and only expose what needs to be used. We will use access modifiers to do this.
  + There are four access modifiers that Java provides us
    - Public: accessible everywhere
    - Protected: accessible within the class, other classes in the same package, and in all subclasses
    - No modifier: same as protected except not accessible in subclasses in a different package
    - Private: only accessible within the class itself
  + The two most common are Public and Private.
  + We want to encapsulate our properties so that they cannot be changed outside of the proper channels to change them.
    - An example where not doing this would cause a problem is if we had a Square class, with length, width, and volume, and someone changed the volume without changing the length and width.
  + We can do this by adding public getters and setters and making the properties private. **Add getters and setters**
    - Getters and setters, also known as accessors and mutators, are methods used for accessing and changing data in a class.
  + We can then add validation to getters and setters where needed, but even if no validation is added and the getter and setter just return and set the value, it is still best practice to have them and make the properties private.
    - Let’s add some validation on firstName, lastName, and phoneNumber
      * All cannot be blank
      * phoneNumber can only have ten digits
    - Let’s add validation to grade, only 1 through 12 alloud.
    - Let’s make some of the methods private that don’t need to be exposed, validation methods are a perfect example of this.
  + Notice how private properties and methods are not accessible via autocomplete and if we still try to use them we get a compile error.
  + Now we have encapsulated our data and methods so that the user of the class doesn’t need to be concerned with the inner workings and cannot break our code by messing with things they shouldn’t.
  + Let’s create a Rectangle class
    - Width, length, area(no setter)
    - Private calculateArea() called on setWidth and setLength
* Inheritance, Abstraction, and Polymorphism – Let’s build some classes SalariedEmployee, HourlyEmployee, Manager
  + These each could have names, pay, phoneNumber, address, reportsTo and more. However, they will handle some things differently, like the difference in how salaried and hourly employees are paid, and the manager could have an extra property called bonusPercentage.
  + If we started programming these classes, we would see a bunch of duplicate code, since they are similar, and all have the fields we discussed.
  + To avoid such duplication, we can take everything that the classes have in common and create a parent, or super class for them to inherit from. Employee.
    - This class will be an abstract class, meaning that it can’t be instantiated because it has some abstract methods which must be implemented by inheriting classes like our SalariedEmployee, HourlyEmployee, and Manager.
    - Let’s put firstName, lastName, phoneNumber, address, pay, and reportsTo on this class with getters and setters.
    - Let’s create a public method called getInformation and add an implementation that returns information about the employee.
    - Let’s add an abstract method called getPaycheckAmount().
      * Since it’s abstract it has no implementation and inheriting classes must provide their own implementation, which is an example of polymorphism.
    - Make each of our employee sub classes inherit from the Employee class using the extends keyword. Each of the sub classes now have access to the parent class’s properties and methods, they have inherited them.
      * You can see this by using the Eclipse autocomplete to view the methods on the sub classes. Notice how it shows they are part of the parent class.
      * Also notice that there are other methods that come from the Object class. This is because all classes implicitly inherit from the Object class.
  + We have now used inheritance, abstraction, and polymorphism to make code reusable and more manageable, which is what OOP is all about.
* Exceptions – Sometimes, even though code compiles, it crashes when something goes wrong. When this happens, an exception is thrown. An exception is like an error, it is literally an exception to what was supposed to happen.
  + A common exception you have seen by now is the ArrayIndexOutOfBounds exception that occurs when you try to access an index of an array that doesn’t exist, or is out of bounds.
  + Unless an exception is handled, it will cause the program to crash and terminate.
  + There are two types of exceptions: Checked Exceptions and Unchecked Exceptions
  + Unchecked exceptions are exceptions we don’t check for in our code. We can avoid these by writing clean code and following best practices. In other words, these exceptions are only thrown because of our mistakes, we have complete control over them.
    - Two main examples of these are IndexOutOfBounds and NullPointer
    - Both of these can be mitigated by coding properly
    - Check the length of an array or collection before accessing an element
    - Ensure that a variable’s value is not null before trying to access a property or method on it
  + Checked exceptions are exceptions we don’t have control over, such as an IO (input/output) exception. If we are reading from a database and the database shuts down and we lose connection, that will throw an exception.
    - We may not be able to stop these types of exceptions with code because they are out of our hands, but we can check for them and handle them if and when they occur.
    - We use try catch finally blocks to catch and handle exceptions.
      * A try block encompasses code that could throw an exception and tries to execute it
      * If an exception is thrown, the proceeding catch block will catch the exception and then execute the code placed in the block
      * A finally block is not required, but if used it will always run after the try and/or catch block.
    - Example: <https://www.caveofprogramming.com/java/java-file-reading-and-writing-files-in-java.html> - code this and show examples of the exception being thrown and handled
    - If you write a method that calls code that throws an exception but doesn’t handle it, you must add a throws clause to your method signature so that when the method is called the exception can be handled then.
* Menu driven app with Team and Player classes
* Research Homework
  + Read 5 online articles about Object Oriented Programming with Java
  + Summarize what you learned in each article
  + Include the link to the article
  + Research and write about the differences between checked and unchecked exceptions
* Coding homework – create three classes: Card, Deck, and Player.
  + Card
    - Fields
      * Value (e.g. 2 – 10 are their respective values. Jack – Ace is 11 – 14)
      * Name (e.g. Ace of Diamonds, or Two of Hearts)
    - Methods
      * Getters and setters for the fields
      * A method that prints out information about the card
  + Deck
    - Fields
      * A List of Card
    - Methods
      * Shuffle (should randomize the order of the cards)
      * Draw (should remove the top card from the list of cards and return it)
      * In the constructor, when a new Deck is instantiated, it should have the standard 52 cards in a Deck.
  + Player
    - Fields
      * Hand (List of Card)
      * Score
      * Name
    - Methods
      * Describe (should print out info about the Player)
      * Discard (should remove a specific card from the Hand field)
      * Draw (should take a Deck as an argument and call the draw() method on the Deck an add the returned Card to the hand field)