Week 3 Review

Day 11: Inheritance

- Allows a class to take on the properties and methods defined in another class.
  - A **subclass** is the derived class that inherits the data and behaviors from another class.
  - A **superclass** is the base class or parent class whose data and behaviors are passed down.
  - All classes are actually subclasses of the `java.lang.Object` class.
  - You may hear superclass referred to as the **parent** class and subclass referred to as the **child** class.
- A class can inherit from another class using the `extends` keyword.
- Subclasses must call at least one superclass if not using the default constructor (use `super` keyword).
- `private` vs. `protected` access modifiers
  - `protected` acts as `private` to all other classes but every class that extends the class will still have access as if defined with the `public` access modifier.

Day 11: Inheritance: Overriding Methods

- A subclass can **override** a method from the superclass by redefining the method.
  - When a subclass method is called, the subclass method will be called if defined, otherwise the superclass method will be.
  - Method **signature must match** the signature being overridden **exactly**.
  - Java provides the `@Override` annotation to make it clear a method overrides the original method.
    - If you use the `@Override` annotation on a method you intend to override, you will get a compiler error if your signature does not match the signature of any signatures in the superclass. This is very useful to ensure your method **WILL** actually override as intended.
- If a subclass overrides a superclass method, that class can always call the superclass method by using the `super.` prefix to access the super version of the method.

Day 11: Inheritance As Polymorphism

- Specialization classes can be referred to by their base class

```
Auction auction = new ReserveAuction();
```

`ReserveAuction` is-an **`Auction`**. We can refer to any subclass of **`Auction`** using **`Auction`** as the variable type.

- This promotes polymorphic code
  - Classes can only inherit from one class

Day 12: Polymorphism in Java

In object-oriented programming, **polymorphism** is the idea that something can be assigned a different meaning or usage based on the context it is referred to as. Put another way, different objects can be treated as the same type of thing within a program.

- Polymorphism using inheritance is the concept that any object which is a subclass can be treated as the superclass type.
  - `Auction buyOutAuction = new BuyOutAuction("Super cool thing", 150);`
- Polymorphism can also be implemented using **interfaces**.

Day 12: Interfaces

- Interfaces define behaviors that objects must implement.
  - You can think of them as contracts that must be followed.
- Keyword: **`implement`**
- Defines what (the method signature) but not how (the actual method body).
- An interface is a contract that defines which methods a user of the interface can expect.
- Cannot be instantiated.
- Objects may implement more than one interface.
- If class A implements interface B, the A "is-a" B, and so are its subclasses.

Day 12: Interfaces

```
public interface Singable {

    String getName();
    String getSound();

}
```

No access modifier specified (defaults to **`public`**).

Ends with semicolon because there is no code. Interface is only the contract.

Day 13: final Methods & Classes

- Making methods **`final`** means that children can't override what the parent has defined
  - Prevents logic that is integral to the application from being overridden by a poorly behaving subclass
  - Just a design decision that should have a good reason for using
- Making classes **`final`** means that another class can't inherit from it
  - Again, just a design decision. Should have a good reason for doing it

Day 13: Constructor Inheritance Review

- In **`FarmAnimal`**, we have a constructor, but have to implement that in the sub classes

- Constructors aren't inherited and must be redefined

- If the default constructor is not defined in the base class, you must call a valid constructor in the subclass constructor

  - Note that you **only have to call <u>one</u> constructor** from you subclass

Day 13: Abstraction & Classes/Methods

- **Abstraction** is the principle of handling complexity by hiding unnecessary details from the user. Its goal is to enable the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.
    - Abstraction is sometimes referenced as a fourth principle of Object-Oriented Programming but it is often not included as one of the principles because it motivates the other three principles in one way or another.
- An **abstract class** is a class that cannot be instantiated. It exists solely for purposes of inheritance and polymorphism.
- An **abstract method/function** is a method/function that does not have an implementation and must be overridden by subclasses.

Day 13: Abstract Class & Methods: Rules

- Abstract classes can not have objects created from them, but they can provide logic and structure to their subclasses.
- Abstract methods are methods with no logic that must be implemented by concrete subclasses.
- If a class has an abstract method, it must be an abstract class.
- If a class does not override an abstract method from its parent, it must also be an abstract class

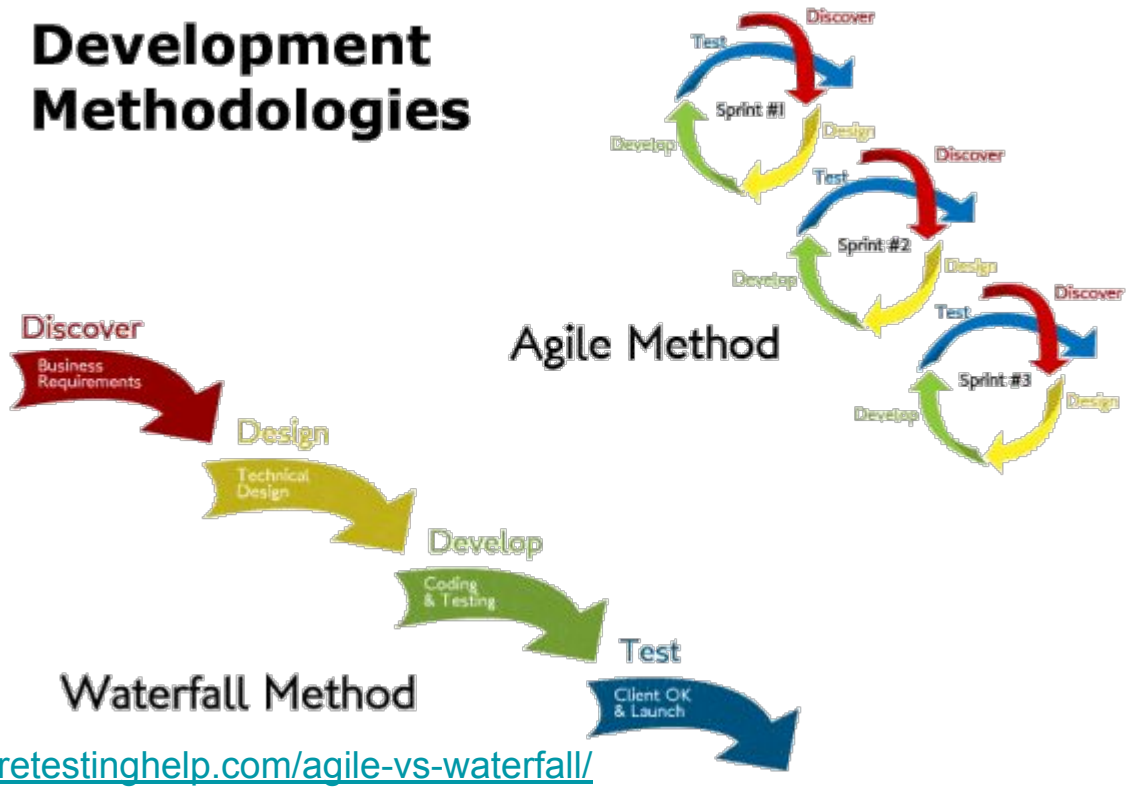
Day 13: How to Choose Access Modifiers

- **`public`** is for any set in stone methods that you want other programmers to rely on to use your object.

- **`protected`** is for building connections between inherited classes. It lets you have methods in a parent that are accessible by its children, and vice versa, but not anyone else.

- **`private`** is for unstable methods that may change and only have use inside the class itself.

Day 14: SDLC - Software Development Life Cycle



https://www.softwaretestinghelp.com/agile-vs-waterfall/

Day 14: Types of Testing

- **Unit Testing**: Tests the smallest units possible (i.e. methods of a class).
- **Integration Testing**: Tests how various units or parts of the program interact with each other.
  - It can also be used to validate some external dependencies like database systems or API's.
- **User Acceptance Testing**: Tests the functionality from the end user's perspective. It can be conducted by a non-technical user.

Day 14: Other Types of Testing

- **Security Testing**: Is our data safe from unauthorized users?
- **Performance Testing**: it works with 1 user, what about a million?
- **Platform Testing**: Works great on my laptop, what if I pull up the app from my phone?
- **Test-Driven Development (TDD)**: Code is written by creating tests that initially fail and writing all the needed code to make them pass.

Day 14: Unit Testing in Java: Introduction

The most commonly used testing framework in Java is **JUnit**.

- **JUnit** is written in Java and will leverage all the concepts you've learned so far: declaring variables, calling methods, instantiating objects.
- All related tests can be written in a single test class containing several methods, each method could be a test.
- Each method should contain an assertion, which compares the result of your code against an expected value.

Day 14: Testing Using Assertions

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {

      public boolean isEven(int num) {
            return num % 2 == 0;
      }

}
```

```
@Test
public void isEven_withEvenValue_shouldReturnTrue() {
      Example example = new Example();
      boolean expected = true;
      boolean result = example.isEven(6);
      assertEquals(expected, result);
}
```

Create object.

This is the result we expect with an even number.

Call method with even number and store result.

Compares expected to result, fails they are not equal.

Day 14: More About the Test Class

```
public class ExampleTest {

    @Test
    public void isEven_withEvenValue_shouldReturnTrue() {
        Example example = new Example();

        boolean expected = true;
        boolean result = example.isEven(6);

        assertEquals(expected, result);
    }

    @Test
    public void isEven_withOddValue_shouldReturnFalse() {
        Example example = new Example();

        boolean expected = false;
        boolean result = example.isEven(9);

        assertEquals(expected, result);
    }

}
```

`@Test` is an annotation indicating this is a test.

Evaluates result and fails if it doesn't match expected result.

There are two methods: one to test even case and one to test odd case. Tests are usually declared with `void` return type.

Day 14: @Before and @After Methods

A method annotated with **@Before** will run before **each** test.

A method annotated with **@After** will run after **each** test.

```
public class ExampleTest {

        @Before
        public void setUp() {
                // do test setup
        }

        @After
        public void tearDown() {
                // do test cleanup
        }

        @Test
        public void isEven_withEvenValue_shouldReturnTrue() {
                // test code
        }

        @Test
        public void isEven_withOddValue_shouldReturnFalse() {
                // test code
        }

}
```

Test flow:

1. **setup()**
2. first test
3. **tearDown()**
4. **setup()**
5. second test
6. **tearDown()**

Day 14: Unit Testing Structure

- **Arrange**: begin by arranging the conditions of the test, such as setting up test data

- **Act**: perform the action of interest, i.e. the thing we're testing

- **Assert**: validate that the expected outcome occurred by means of an assertion (e.g. a certain value was returned, a file exists, etc.

Unit Testing Best Practices

- No external dependencies
- One **logical** assertion per test (i.e. each test should only contain one "concept")
- Test code should be of the same quality as production code

How to Unit Test

Find boundary cases in the code

- Is there an if statement?
  - Test around the condition that the if statement tests
- Is there a loop?
  - Test arrays in the loop that are empty, only one element, lots of element
- Is an object passed in?
  - Pass in null, an empty object, an object missing values that the method expects