# Pro Dev Session

# Pro Dev Session

You may want to write this down.

Collaboration is an essential part of being a great teammate and developer. To be a good collaborator, we must master our collaboration tools, namely git and GitHub.

Although we've done this exercise before, working with pull requests and merge conflicts is pivotal enough to warrant more practice. Next class, we'll introduce some intermediate git concepts.

# Git Practice

## INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

**Goals:**

- Working with a partner, create a new repository.
- Each partner should clone the repository.
- Partner A should create a branch and on this branch create a new IntelliJ project with a Main class that contains the main method.
- Partner A should add, commit, and push their changes
- Partner B should review the pull request and merge it into master.
- Both partners should use git pull to locally update their master.
- Now both partners should add code to the main method. Any code is fine.
- Partner A should add, commit, and push and merge the changes into master.
- Partner B should pull. What happens? You should see a merge conflict. This is an opportunity to decide which lines of code to keep and which to disregard.
- Watch this video, then work together to resolve the merge conflict.

**20 minutes!**

## Stand Up

## Testing

## Testing

### ENSURING QUALITY

Notebooks Ready? It’s time for a mini lecture.

## Testing

ENSURING QUALITY

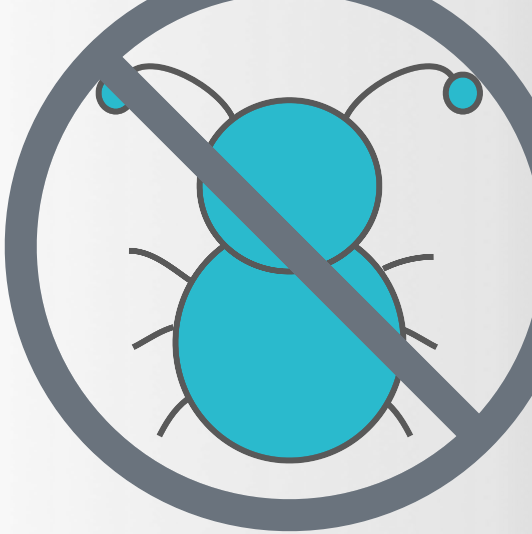
Testing is arguably one of the most important concepts that we will cover in this course.

This module is just the introduction. We will continue to learn about testing for the remainder of the course.

If you fall behind here, please seek help early. A solid understanding of testing will be vital to your success in the course.

# Testing

ENSURING QUALITY



**What is Software Testing?**

Testing is the process of ensuring that an application meets the specifications and is free of bugs.

## **Critical** Thinking

- Why is testing so important?

# Testing

ENSURING QUALITY

**What does this mean practically?**

This means that we need to write additional code that runs parts of our application and tests that they do what we expect.

## Testing

### TYPES OF TESTS

There are several different types of tests. In this course we’ll focus primarily on 4 types:

Unit Tests - *testing individual units of code (usually a method)*
Integration Tests - *testing the interaction between 2 or more units of code*
System Tests - *testing that the entire system meets requirements*
Acceptance Tests - *testing that the software meets business and user needs*

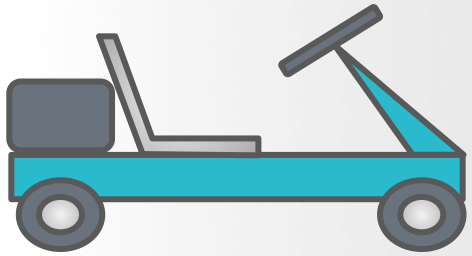
## Testing

### TYPES OF TESTS

As an analogy, suppose you manufacture small go-karts for children.

You begin by manufacturing an axle. Axles need to support any weight up to 200lbs.

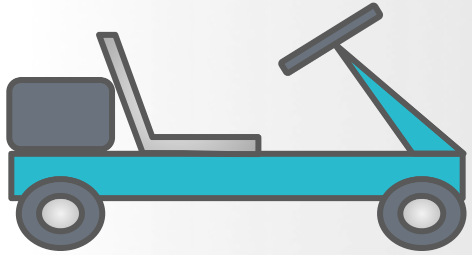
You begin by testing the axle. This is a **UNIT TEST**.



## Testing
### TYPES OF TESTS

Next you build the wheels. The wheels need to also hold weight up to 200lbs and maintain integrity for 200 miles of travel on rough gravel.

You test the wheels ability to meet these requirements. This is a **UNIT TEST**.

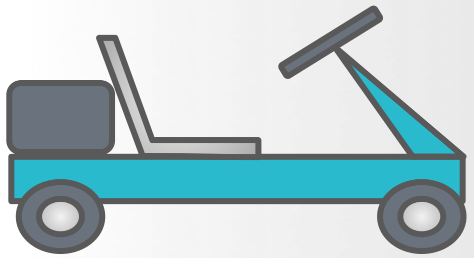


## Testing

### TYPES OF TESTS

Then you put the wheels on the axle. They need to fit perfectly with zero slip and stay connected during turns with an angular frequency of 1.8 radians/second*.

You test that these requirements are met when the two units are put together.
This is an **INTEGRATION TEST**.



*This is just physics mumbo jumbo that you can ignore.

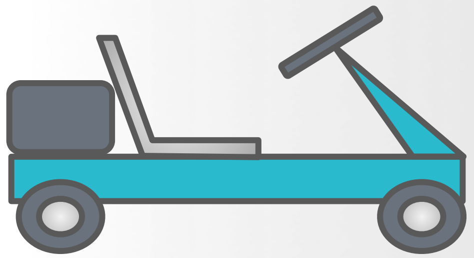
# Testing

## TYPES OF TESTS

You continue to unit test each piece (seat, steering wheel, motor, chassis, etc ) as you build them and then test their integration with every component that they directly interface with.

Finally, you have a complete Go-Kart! You put a crash dummy on board and take that sucker out for a spin. Ensuring it runs, turns, steers, and brakes as expected.

Congrats! You just completed your **SYSTEM TEST**.



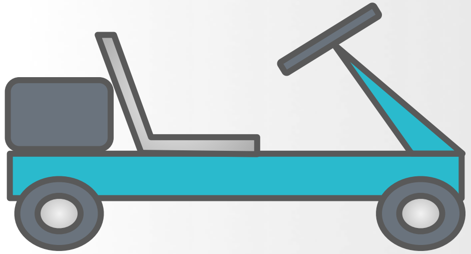
## Testing

### TYPES OF TESTS

Now you get a bunch of kids together and make sure they love it!

Can they reach the pedals? Do they understand how to steer, accelerate, and brake? Do they like the color? Is the seat comfortable?

Woohoo **ACCEPTANCE TEST** complete.



# Testing

## TYPES OF TESTS

Today we will focus on Unit Tests:

**Unit Tests - *testing individual units of code (usually a method)***
Integration Tests - *testing the interaction between 2 or more units of code*
System Tests - *testing that the entire system meets requirements*
Acceptance Tests - *testing that the software meets business and user needs*

# Testing

```
public class Calculator{
    public int add(int a, int b){
                    return 0;
          }
}
```

---

```
public class CalculatorTest{
    @Test
    public void shouldAddTwoPositiveIntegers() {
        Calculator calc = new Calculator();
        assertEquals(3,calc.add(1,2)));
    }

}
```

REQUIREMENTS: The add method should add two integers.

# Testing

```
public class Calculator{
    public int add(int a, int b){
                return a+b;
          }
}
```

---

```
public class CalculatorTest{
    @Test
    public void shouldAddTwoPositiveIntegers() {
        Calculator calc = new Calculator();
        assertEquals(3,calc.add(1,2)));
    }

    @Test
    public void shouldAddTwoNegativeIntegers() {
        Calculator calc = new Calculator();
        assertEquals(-5,calc.add(-1,-4)));
    }

}
```

# Testing

```
public class CalculatorTest{

    Calculator calc;

    @Before
    public void setUp() {
        calc = new Calculator();
    }

    @Test
    public void shouldAddTwoPositiveIntegers() {
        assertEquals(3,calc.add(1,2)));
    }

    @Test
    public void shouldAddTwoNegativeIntegers() {
        assertEquals(-5,calc.add(-1,-4)));
    }

}
```

## Testing

### UNIT TESTS

When unit testing we can’t test every possible input. We need to group inputs together into equivalence classes.

If our add method can add 1+2 and 100+15 and 43+72, it stands to reason that it can handle all positive integers with some upper bound based on memory limitations.

## **Critical** Thinking

- What equivalency classes might we have for an add method?

## **Critical** Thinking

Did you come up with the following? Fewer? More?

- Positive integers
- Negative integers
- One positive and one negative
- Zeros
- One zero along with a positive
- One zero along with a negative
- Two integers whose sum exceeds the maximum or minimum allowed integer

## Testing

INDEPENDENT PRACTICE

It’s time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

**Goals:**

- Create a class called Calculator.
- In the class create a method called divide that takes two integers.
- Divide should return the integer that results from dividing the 2 integers.
- It should return 0 if the numerator or denominator is 0*
- Write unit tests to test your divide method

* Yes, this is bad math, but clients ask for weird things.

**15 minutes!**

Breathe

## **Stay Seated** &Take 3 Deep Breaths.

**RELAX.**

Now take a short walk. Clear your head. After a few minutes break, quickly review your notes. We’ll start back a few minutes.

## TDD

## TDD

### TEST DRIVEN DEVELOPMENT

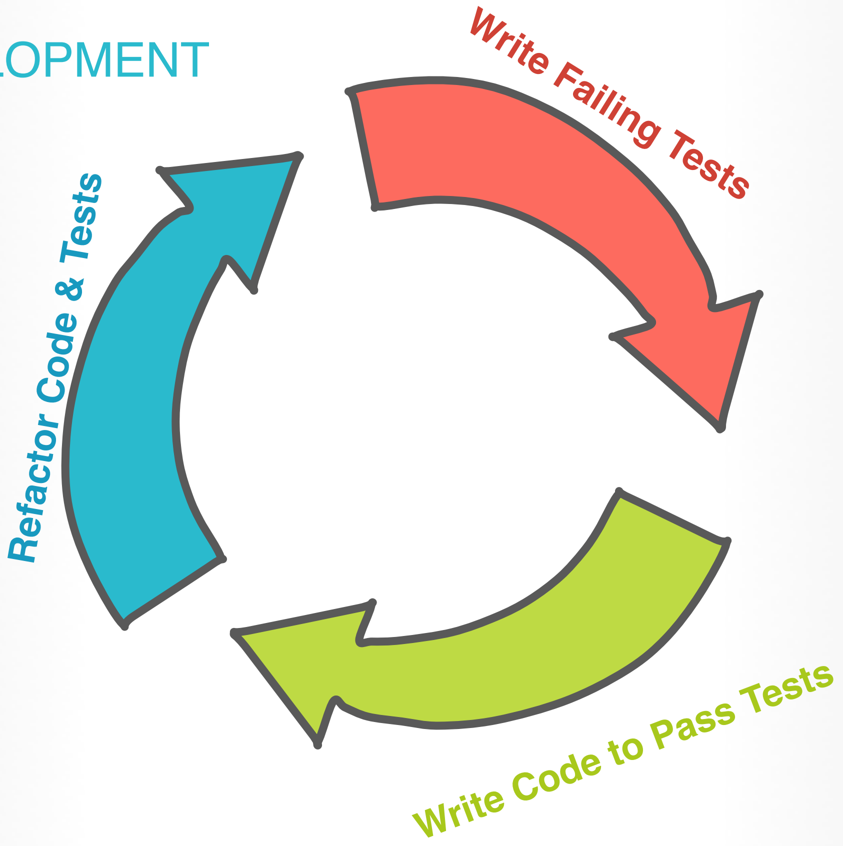
Notebooks Ready? It’s time for a mini lecture.

## **Key**Point

In this class, we will be utilizing a style of development called Test Driven Development (TDD). From this point on, you MUST write your tests first and then write the minimum amount of code to pass the tests.

# TDD

TEST DRIVEN DEVELOPMENT



## TDD

### TEST DRIVEN DEVELOPMENT

We begin by writing failing tests, thinking carefully through equivalence classes as we design our tests.

Then write the code to pass all tests.

Then we refactor our code and tests BUT NOT AT THE SAME TIME.

Then we rinse and repeat as needed.

# Testing

```
public class TVTest {

        TV tele;

        @Before
        public void setUp() {
            tele = new TV(15);
        }

        @Test
        public void shouldTurnOn() {
            tele.on();
            assertTrue(tele.getIsOn());
        }

        @Test
        public void shouldTurnOff() {
            tele.off();
            assertFalse(tele.getIsOn());
        }

        @Test
        public void shouldChangeChannel() {
            tele.increaseChannel();
            tele.increaseChannel();
            assertEquals(17, tele.getChannel());
        }
}
```

## Testing

INDEPENDENT PRACTICE

It’s time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

**Goals:**

In your Calculator project:

- Add failing tests for a sumArrays method. sumArrays should take two arrays and sum all the numbers in both arrays.
- Add a failing test for arrayify. arrayify should take in two integers. It should create an array of the length of the first integer that contains consecutive integers starting at the second integer. For example, arrayify(3,5) returns [5,6,7].
- Once you have failing tests with a sufficient number of equivalence Classes, write code to pass the tests.

*You'll need to use* `assertArrayEquals` *to compare arrays.*

**20 minutes!**

## A Quick Aside

You don’t need to write this down.

Note that tests serve as a debugging tool, a means to ensure that your code meets the clients requirements, documentation, and design tool.

It is entirely normal to spend more time writing tests than coding.

It is entirely normal to have more lines of test code than production code.

The money spent on developer time writing code, pays for itself ten fold in saved developer time on maintenance and debugging.

lunch.

## Lab Time

# TDD

## CODE-A-LONG

Open your laptop. Code with me. Don’t jump ahead.

Time to test something a little bigger. Let’s work together complete all of the goals below.

**Goals:**

- Let’s whiteboard out the classes and methods needed for an online wine store.
- Wine can be searched by name (complete or partial), vintage range(inclusive), region or varietal (exact match), country (exact match),  producer (partial match).

## TDD

INDEPENDENT PRACTICE

It’s time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

**Goals:**

- Write all the tests you think are necessary to have adequate test coverage for this small application.
- Write the code for the application ONLY AFTER you have 100% test coverage. Write the minimal amount of code needed to pass tests.
- Once all tests are passing, refactor.

We are not using Spring or JPA for this. This application is not currently runnable except through tests. This will essentially become our service layer and part of our data layer.

**Rest of Class**

## **Stay Seated** & Take 3 Deep Breaths.

**RELAX.**

Now take a short walk. Clear your head. After a few minutes break, quickly review your notes. We’ll start back a few minutes.

## Sneak Peek

Note that next class will start with the sneak peek topic. You are NOT expected to master this today.

## Mocks

Mockito to the Rescue

Notebooks Ready? It’s time for a wee little lecture.

# Mocks

## Mockito to the Rescue

Unit tests are fairly easy to write when your methods are all stand alone. But that’s not always the case. The web layer relies on the service layer. The service layer relies on the data layer.

Testing a method that calls another method generally requires a mock.

## Mocks

### Mockito to the Rescue

**What’s a mock?**

Mocking is creating a dummy object that simulates the behavior of another service that the method being tested relies on.

For example, our service layer relies on our DAO which relies on our database. But in a unit test, we just want to know if this one part of the service layer works. We don’t want our unit test of a service layer method to fail because our database connection failed. So we mock the DAO.

Here I’ll show you…

## Mocks

```
private void setUpDogDaoMock() {
    dogDao = mock(DogDao.class);
    Dog dog = new Dog();
    dog.setId(1);
    dog.setName("Carl");
    dog.setKennelNum(10);
    List<Dog> dogList = new ArrayList<>();
    dogList.add(dog);

    doReturn(dog).when(dogDao).getDog(1);
    doReturn(dogList).when(dogDao).getAllDogs();
}
```

Now we can test service layer methods that rely on the data layer methods getDog and getAllDogs without having to know whether those methods are functional or even built yet.

This allows us to independently test unit methods that depend on other methods.

# Wrap Up

## **Module 5** Lesson 2

HOMEWORK

You don’t have to submit your nightly homework, but you are expected to complete it.

Complete the Wine Application. If you have already finished, add additional functionality that allows wine to be searched based on characteristics (exact matches only). The characteristics are as follows:

Body -> can be light, medium, or full
Alcohol -> can be low, medium, or high
Intensity -> can be low, medium, or high
Sweetness -> can be dry, semi-sweet, or sweet
Tannin -> can be low, medium, or high
Acidity -> can be low, medium, or high

## **Daily** Assessment

You may leave after a staff member approves your assessment.

## **Daily** Assessment

Work INDIVIDUALLY to complete all of the goals below.

**Goals:**

- Write a complete series of tests to test a method that takes a string that is lowercase “hello world” and returns the string camel-cased “helloWorld”.