Baby steps:
Basic Unit Testing

What is JUnit?

In order to work effectively with unit testing, we need:

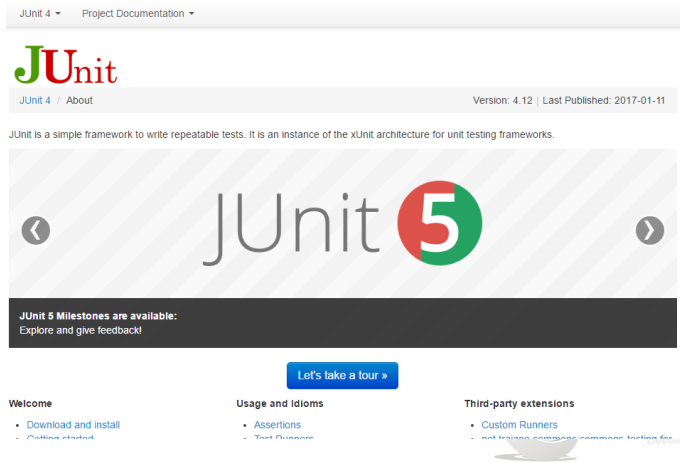
- A way to name test cases
- A way to exercise the code we wish to test, see if it works as expected and report the results
- Something to help us understand the results (pass/fail)

While we could build all of this ourselves, it's far better to use a testing framework.

JUnit is the most widely known and widely used Java unit testing framework, and we will use it during this course.

Getting JUnit

Junit 4 has a webpage at **www.junit.org**. We don't need to download it because both **IntelliJ** and **Eclipse** already ships with JUnit.



Test fixtures and tests

Test cases are written as methods, marked with the **@Test** annotation.

Such methods are written inside a class. The class can optionally take annotations too, but we don't need them right now.

```
public class StatusCalculatorTests {

    @Test
    public void startWithZeroPointsAndBasicStatus() {
        // Test code comes here
    }
}
```

Note that both the class and the test methods must be **public**.

An immediate design issue

We'd like our tests not to break as our program evolves, and to control the data they are working with. Therefore, we will pass the status levels into the **StatusCalculator**'s constructor.

```
private Map<String, Integer> statusLevels
        = new HashMap<String, Integer>(); {
    statusLevels.put("Basic", 0);
    statusLevels.put("Silver", 50000);
    statusLevels.put("Gold", 100000);
}

@Test
public void startWithZeroPointsAndBasicStatus() {
    StatusCalculator sut = new StatusCalculator(statusLevels);
    // More to do here...
}
```

Class does not exist yet

Assertions

Asserts are used to check that some property or method of the object we are testing produces the expected value. Typically an assertion has:

- The expected value
- Something that obtains the value from the object
- A description

```
@Test
public void startWithZeroPointsAndBasicStatus() {
    StatusCalculator sut = new StatusCalculator(statusLevels);

    Assert.assertEquals("Zero points", 0, sut.getPoints());
    Assert.assertEquals("Basic status", "Basic", sut.getStatus());
}
```

Method does not exist yet

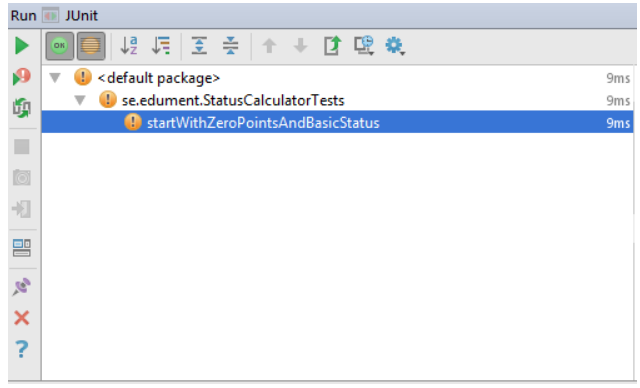
Stub the class

So far, our test won't even compile - we have no **StatusCalculator** class! Let's create one, stubbing our getter methods by returning the default value of each type for now.

```
public class StatusCalculator {

    private Map<String, Integer> statusLevels;

    public StatusCalculator(Map<String, Integer> statusLevels) {

        this.statusLevels = statusLevels;
    }

    public int getPoints() {
        return 0;
    }

    public String getStatus() {
        return null;
    }
}
```

See the test fail

We can now compile our test and run it under JUnit. Naturally, it fails.



Why might we want to see our test fail before we work on the feature it exercises?

Do as little as needed to make the test pass

The only thing we need to do to make the test pass is return sensible values from the **Points** and **Status** getters.

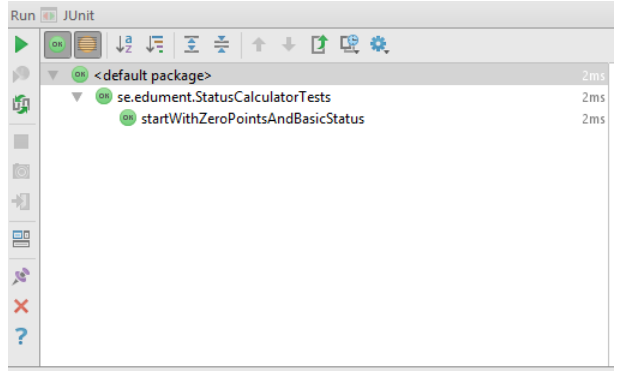
```
public int getPoints() {
    return 0;
}

public String getStatus() {
    return "Basic";
}
```

When starting out with TDD, it's best to try and avoid thinking too far ahead. We could add many more things at this point - but try to resist the temptation. Sometimes, it can be surprising to learn what you don't need!

"Always implement things when you **actually** need them,
never when you just **foresee** that you need them."
http://c2.com/xp/YouArentGonnaNeedIt.html

Now we have green

With the changes on the previous slide, we now have a passing test.



At this point, our code is extremely simple, and there is no interesting refactor that we could do. Thus, we move on.

The "System Under Test" convention

A unit test should exercise a single unit (typically, an object). At the moment there is only one object involved in our tests, but in the future there may be others (for example, mock of stub objects).

Therefore, it is useful to adopt a naming convention for the object that is the **S**ystem **U**nder **T**est, so it can be easily identified. In this course, we will use the name **sut** consistently for this.

```
StatusCalculator sut = new StatusCalculator(statusLevels);

Assert.assertEquals("Zero points", 0, sut.getPoints());
Assert.assertEquals("Basic status", "Basic", sut.getStatus());
```


Testing point counting

We know from our requirements that we need a way to record when points are earned. We therefore write a test to exercise a (to be implemented) **AddPoints** method.

```
@Test
public void recordsExtraPointsAndKeepsCorrectStatus() {

    StatusCalculator sut = new StatusCalculator(statusLevels);

    sut.addPoints(2000);
    sut.addPoints(5000);

    Assert.assertEquals("Correct number of points",
                        7000, sut.getPoints());
    Assert.assertEquals("Still have basic status",
                        "Basic", sut.getStatus());
}
```

Method Not implemented yet

The "Arrange, Act, Assert" practice

Most tests can be broken up into three distinct phases.

Phase	Description
Arrange	Get the System Under Test into a starting state for the test.
Act	Exercise the System Under Test, typically by calling a method.
Assert	Check that the consequences of the action were as expected.

Arranging your tests into these 3 distinct phases will help to make them more understandable, but also help you to get your tests to be a sensible size.

The "**A**rrange, **A**ct, **A**ssert" subdivision is considered good practice. Follow it unless there is a really good reason not to.

The "Arrange, Act, Assert" practice in our test

Most tests can be broken up into three distinct phases.

Arrange:

```
StatusCalculator sut = new StatusCalculator(statusLevels);
```

Act:

```
sut.addPoints(2000);
sut.addPoints(5000);
```

Assert:

```
Assert.assertEquals("Correct number of points",
                    7000, sut.getPoints());
Assert.assertEquals("Still have basic status",
                    "Basic", sut.getStatus());
```

Using this **AAA** convention will make your tests easy to read, maintain and share with other developers.

Passing the test

To make the test pass we add a **private field** to record the points and the **addPoints** method.

```
public class StatusCalculator {

    private Map<String, Integer> statusLevels;
    private int points;

    public StatusCalculator(Map<String, Integer> statusLevels) {

        this.statusLevels = statusLevels;
    }

    public int getPoints() {

        return points;
    }

    public String getStatus() {
        return "Basic";
    }

    public void addPoints(int i) {
        points=points + i;
    }
}
```

A small refactor

We may not have much code yet, but that doesn't mean there's nothing wrong with it. In fact, this method...

```
public void addPoints(int i) {
        points=points + i;
    }
```

...is a pretty good hint that we should consider some variable renaming.

We rename the field to **currentPoints** and the parameter to **newPoints** to make the code clearer.

```
public void addPoints(int newPoints) {
    currentPoints = currentPoints + newPoints;
}
```

Testing that upgrades are given as expected

The next test adds enough points to reach silver status, then asserts that the status is correctly reported as silver.

```
@Test
public void enoughPointsEarnSilverStatus() {
    StatusCalculator sut = new StatusCalculator(statusLevels);

    sut.addPoints(20000);
    sut.addPoints(35000);

    Assert.assertEquals("Correct number of points",
                        55000, sut.getPoints());
    Assert.assertEquals("Upgraded to silver status",
                        "Silver", sut.getStatus());
}
```

Now we need to implement this failing test.

Isolated changes

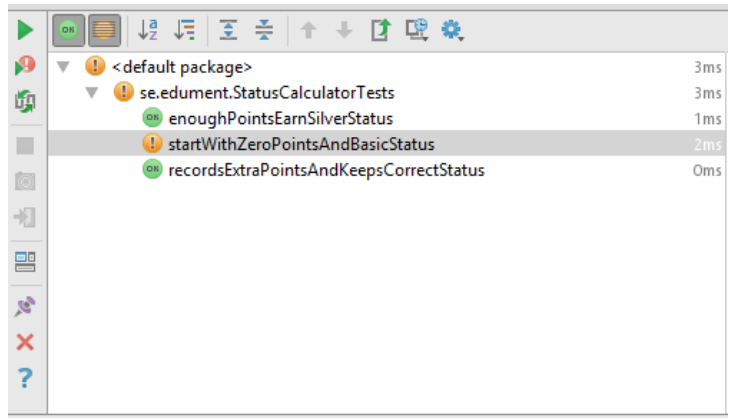
We may be tempted to add a field for the current status and update it as points are added.

However, there's a simpler way that needs an update to just one field: calculate it on demand.

```
public String getStatus() {
    return levelFor(currentPoints);
}

private String levelFor(int limit) {
    String key = null;
    int highestValue = Integer.MIN_VALUE;
    for (Map.Entry<String, Integer> e : statusLevels.entrySet()) {
        if (highestValue < e.getValue() && e.getValue() < limit) {
            highestValue = e.getValue();
            key = e.getKey();
        }
    }
    return key;
}
```

Oops, regression!

The code seemed reasonable enough. And it does, in fact, pass the test that we just wrote.



However, we broke a previously passing test! Our tests have caught a regression: having zero points no longer indicates basic status.

Fixing the regression

The fix comes from realizing that you are eligible for a level as soon as you have reached the required number of points. Thus, the **<** should have been **<=**.

```
for (Map.Entry<String, Integer> e : statusLevels.entrySet()) {
    if (highestValue < e.getValue() && e.getValue() <= limit) {
        highestValue = e.getValue();
        key = e.getKey();
    }
}
```

Now, all of our tests pass!

(Stop and ponder all the ways this regression could be discovered later if the tests hadn't caught it. Which way do you prefer?)

Summing up what we've done

We've already started to see how tests influence the design process, and once they are passing they serve as regression tests.

Some key things to remember:

- A unit test should be given a **descriptive, but concise name**.
- It is wise to check that your tests **fail first**, before working on making them pass.
- The **Arrange Act Assert** practice helps divide a test into easily recognizable parts.
- The **S**ystem **U**nder **T**est naming convention helps to pinpoint the unit we're currently testing.

Exercise #1

Let's do exercise #1