# JUnit Testing

## Objective(s)

- Learn how to perform unit testing on Java code.
- Learn how to use the JUnit testing framework to perform unit testing.

## Lab setup

1. Download and import the provided Eclipse project

## Unit testing

Unit testing involves testing the smallest testable parts of an application. For Java applications, the smallest testable unit is usually an entire class or individual method. We focus on testing the behavior of the unit rather than private implementation details.

Unit testing requires that the units are tested in isolation. This means that if the method or class depends on another unit we cannot use that unit in testing. We provide stand-ins during unit testing in the form of stubs/mock objects.

### Stub/mock object

```
public class Receiver {
    private PriorityQueue<Message> messageQueue;
    //...
    public boolean receive(Message message) {
        return messageQueue.add(message);
    }

    public String nextMessage() {
        if (!hasMessage()) {
            return null;
        }
        else {
            return messageQueue.poll().getMessage();
        }
    }
}
```

*Figure 1 Receiver class*

Notice the Receiver class in Figure 1 has a dependency on another class called Message. In order to test Receiver in isolation, we need to provide a stand-in for the Message class. We

can write a mock class for Message that provides only simple implementations of what Receiver uses. This class is shown in Figure 2. Notice we extend the class we are providing the stub for and override any methods that are used. We also redeclare any variables from the original class that are used.

```
public class MessageStub extends Message {

    private String message;
    public MessageStub(Priority priority, String message) {
        /// The parent doesn't define its default constructor, we need to call super()
        super(priority, message);
        this.message = message;
    }

    @Override
    public String getMessage(){
        return message;
    }
}
```

*Figure 2 Message stub class*

## JUnit testing framework

The JUnit testing framework is an open-source testing framework for writing Java unit tests. JUnit has a lot of features that are considered out-of-scope for this course. We may only scratch the surface of JUnit in this lab.

### Annotations

JUnit supports a number of annotations which describe the function of a method. The annotations we focus on are the following:

**`@Test`**
Denotes a method is a test method

**`@ParameterizedTest`**
Denotes a method is a parameterized test.

**`@Disabled`**
Disables the test class or test method.

### Test classes and methods

A test class is a class which contains `@Test` methods. Test classes and methods only require that they are not `private`, so we tend to omit any access modifiers.

### Assertions

We use assertions to state what should be true (or false) when executing our tests. The assertions we will use most often are `assertEquals(expected, actual)` and `assertTrue(expr)`. If any assertions made during a test fails, then the test fails.

## Unit testing with JUnit

We will now try unit testing the provided project. To add a JUnit test to your project, right-click on the project and choose New > JUnit Test Case. A window will pop up to help you configure the test.

At the top, make sure the "New JUnit Jupiter test" option is chosen. We will be testing the Receiver class so in the "Class under test" field Browse for edu.odu.cs.cs251.Receiver. Name the test class "ReceiverTest". At the bottom, choose "Next >".

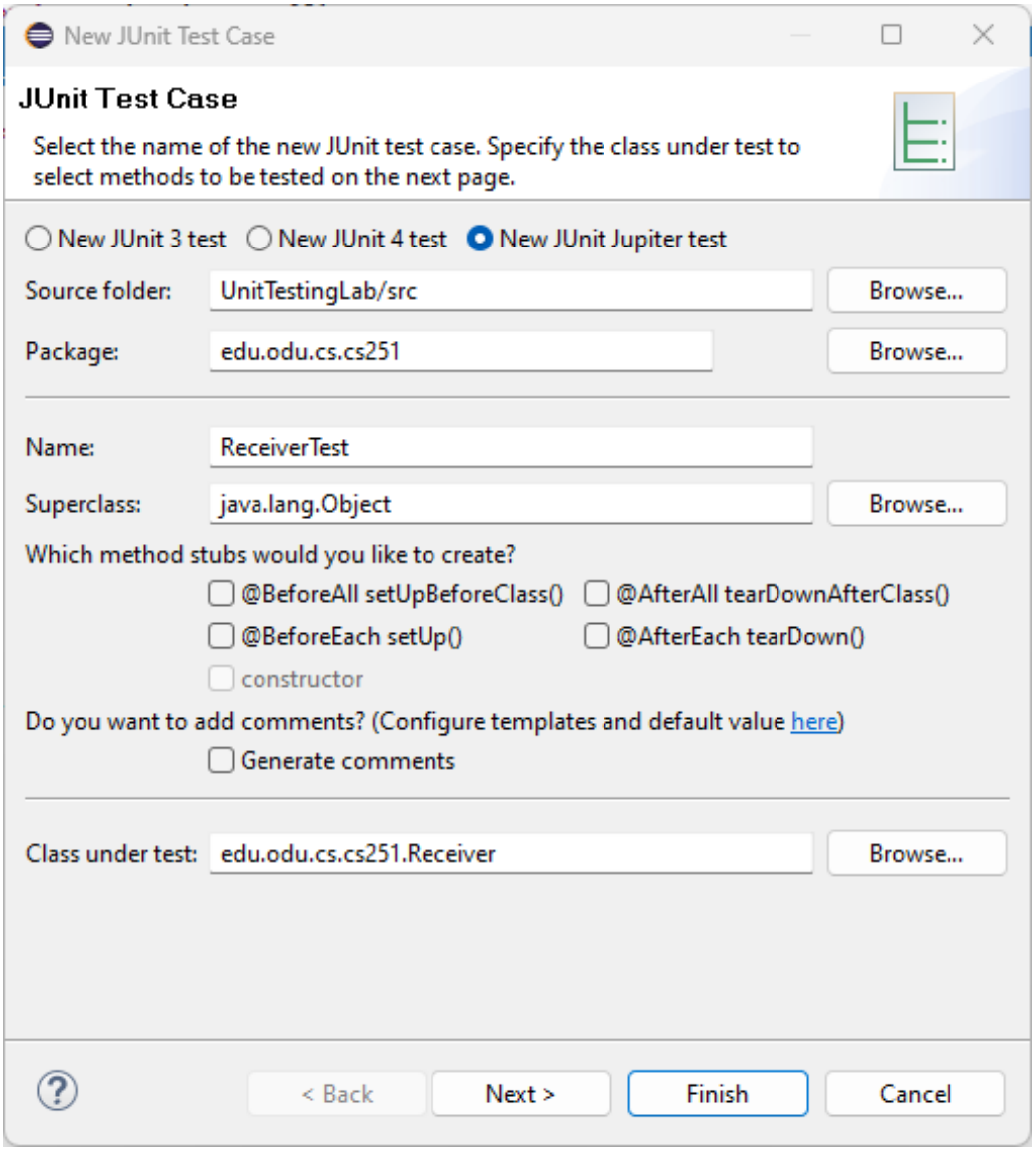


*Figure 3 Creating a new JUnit Test for the Receiver class.*

This will show you the methods that you can have Eclipse automatically generate stubs for. We want to test the constructor and any methods which change the state of the object. Check the methods `receive(Message)` and `nextMessage()`. Also check the constructor `Receiver(String, int, int)`.

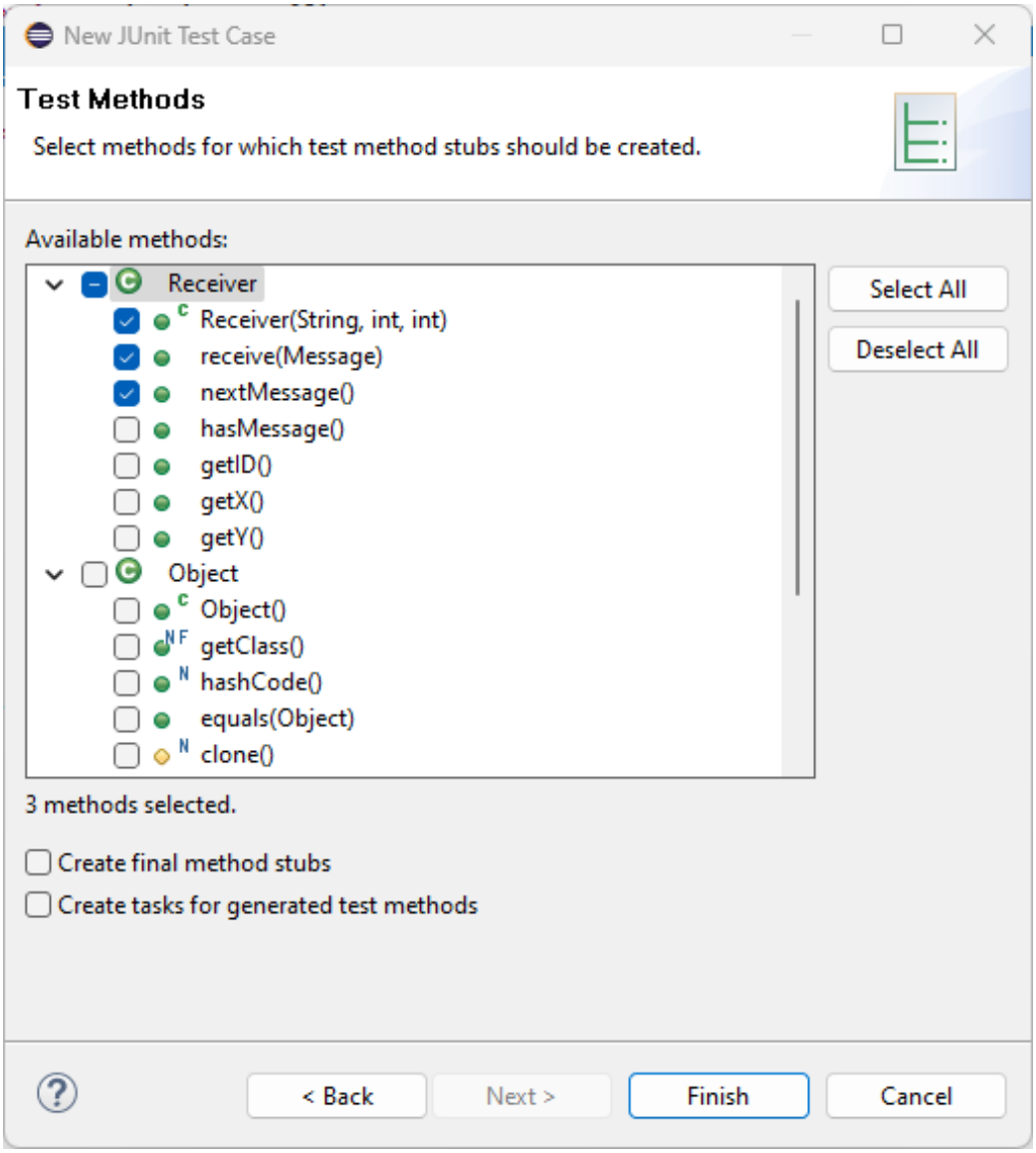


*Figure 4 Choosing methods to create method stubs.*

Click "Finish". Eclipse will automatically open the test class in the editor. You will also see that there are three method stubs in the test class each with only a call to a `fail()` method. You should leave these in place until you are finished implementing a test.

## Testing

We will go through the code for testing the three methods we chose in the previous section.

Testing Receiver()

```
@Test
void testReceiver() {
    Receiver r = new Receiver("abc", 0, 1);

    assertEquals("abc", r.getID());
    assertEquals(0, r.getX());
    assertEquals(1, r.getY());
}
```

*Figure 5 testReceiver() method*

Testing the constructor is simple. Instantiate an instance of Receiver then check the values of the fields using assertions. We should test all the constructors and we could use one test method for each constructor.

Testing receive(Message)

```
@Test
void testReceive() {
    Receiver r = new Receiver("abc", 0, 1);

    Message m = new MessageStub(Priority.NORMAL, "A normal message");
    r.receive(m);

    assertTrue(r.hasMessage());
}
```

*Figure 6 testReceive() method*

When testing this method, we need to consider what this method is specified to do. When receive() is called, the message should be added to the message queue. The state of the queue before a receive() does not necessarily matter, but we can test it for posterity. After a receive(), the queue should have a message.

Can we tell if the message added to the queue is actually the message we gave to receive()? If you look at the Receiver class, you will notice we don't have access to the queue except through hasMessage() and nextMessage().

Testing nextMessage()

```java
@Test
void testNextMessage() {
    Receiver r = new Receiver("abc", 0, 1);

    // A message hasn't been received
    assertEquals(null, r.nextMessage());

    // After a message is received
    Message m = new MessageStub(Priority.NORMAL, "A normal message");
    r.receive(m);
    assertEquals("A normal message", r.nextMessage());

    // The message queue should be empty now
    assertEquals(null, r.nextMessage());
}
```

nextMessage() returns the next message in the message queue or null if the queue is empty. We need to consider the states the queue can be in: its initial empty state, the state after it has received a message, the state after that message has been retrieved, and the state after a message has been retrieved from the queue, while it still has another message.

**Graded submission Task 1:**

Write the remaining tests for Message and Transmitter. Provide any stubs or mocks as necessary. Submit all related code.

Lab wrap-up

In this lab we discussed unit testing and the JUnit testing framework. Unit testing involves testing the smallest testable unit of a program in isolation. If a class we wish to test depends on another class, we must provide a mock or stub class as a stand-in.

JUnit test classes are made up of methods marked with the @Test annotation. We perform tests using assertions. Assertions state what should be true (or false) after a statement is executed in our tests. A test fails if any of the assertions are false.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder "Lab24_firstName_lastName".
- Submit the zipped file to the respective Canvas link.

| Rubrics |
| --- |
| Task 1 (points allotted)<br>- Compilation (2)<br>- Message stub class (2)<br>- Receiver stub class (2)<br>- All tests written (4) |

# Appendix

This section describes testing specifications and more about stubbing and mocking.

## Testing according to specifications

What it means for your code to be correct ultimately comes down to what you (or your team) decide. Typically, you would have decided on a specification or design before you start real coding work. Even then, you will most likely implement something necessary for the specification, but still not specified. An example we would like to cover are the constructors.

In this lab, we tested the constructors by asserting the initial values are what is expected. We did this even for the default constructor, whose default values might not be specified by our specification. If the default values are not specified by the specification, should we be asserting the initial values are those default values?

We can still test the default constructor, but it may feel like bending the rules a bit. Instead of saying "the default constructor initializes the fields to these particular values", we can instead just say it "the default constructor initializes the fields". In code, this means that instead of asserting the fields are initialized to some default values we can assert that the fields are simply not null.

## Stubbing and mocking tightly coupled classes

Tight coupling is when a group of classes are highly dependent on one another. Ideally, we would avoid designing tightly coupled classes because it affects testability. Okay, but we have a couple so how do we test them?

In this example, we are testing the `hashCode()` method of the class Player. Player has a String field and another class called Statistics as a field. A Player's hash code is calculated by adding the hash values of the String field and the Statistics field. The Statistics field overrides its `hashCode()` method. This is shown in code below.

```
public class Player {
    private String name;
    private Statistics stats;

    @Override
    public int hashCode() {
        return name.hashCode() + stats.hashCode();
    }
}
```

We cannot ignore this dependency on Statistics for the purposes of testing. First thing we will need to do is modify Player to be more testable. We will call the `getStatistics()` method instead of directly referring to the field in `hashCode()`. Next, we need to stub Statistics' `hashCode()` method. We will extend the Statistics class with a class called

StubStatisticsHashCode and override `hashCode()`. The overriding implementation does not need to be complex, we will simply return 1. These are shown in code below.

```
public class Player {
    private String name;
    private Statistics stats;

    public Statistics getStats() {
        return stats;
    }

    @Override
    public int hashCode() {
        return name.hashCode() + getStats().hashCode();
    }
}

public class StubStatisticsHashCode extends Statistics {
    @Override
    public int hashCode() {
        return 1;
    }
}
```

The last thing we need to do before we go back to the test method is to mock the Player class. But, isn't Player the class we are testing? Yes. We will be taking advantage of polymorphism. We can extend the Player class and override `getStats()` to return an instance of StubStatisticsHashCode. This is shown below.

```
public class MockPlayerStatistics extends Player {

    public MockPlayerStatistics() {
        super();
    }

    public MockPlayerStatistics(String name) {
        super(name);
    }

    @Override
    public Statistics getStatistics() {
        return new StubStatisticsHashCode();
    }
}
```

Now, with everything in place let's go to the test method. It is shown below followed by commentary on the code.

```
void testHashCode() {
    String testName = "Gatsby";
    StubStatisticsHashCode s = new StubStatisticsHashCode();

    Player p = new MockPlayerStatistics(testName);

    int expectedHash = testName.hashCode() + s.hashCode();

    assertEquals(expectedHash, p.hashCode());
}
```

We declare a test String and StubStatisticsHashCode variable which we will use to calculate the expected hash value. Next, we declare a Player variable and use it to construct a MockPlayerStatistics object. We calculate the expected hash value then assert it is equal to the return value of `p.hashCode()`. MockPlayerStatistics uses its parent's (Player) `hashCode()` method definition. The call to `getStatistics()` in `hashCode()` uses the definition in MockPlayerStatistics thus returning a StubStatisticsHashCode object instead of the field.