Unit Testing

CIT 590

Unit Testing

Unit Testing

- **You *have to* test your code to get it working**
  - **You can do *ad hoc* testing by testing whatever occurs to you at the moment**
  - **Or you can write a set of unit tests that can be run at any time**
    - **This is like the testing class we write to test our main program in Python**

Unit Testing

- You *have to* test your code to get it working
  - You can do *ad hoc* testing by testing whatever occurs to you at the moment
  - Or you can write a set of unit tests that can be run at any time
    - This is like the testing class we write to test our main program in Python
- Disadvantages of writing unit tests:
  - It's a lot of extra programming
    - *True* -- but use of a good testing framework can help quite a bit
  - You don't have time to do all that extra work
    - *False* – unit testing reduces debugging time more than the amount spent building the actual tests

Unit Testing

- You *have to* test your code to get it working
  - You can do *ad hoc* testing by testing whatever occurs to you at the moment
  - Or you can write a set of unit tests that can be run at any time
    - This is like the testing class we write to test our main program in Python
- Disadvantages of writing unit tests:
  - It's a lot of extra programming
    - *True* -- but use of a good testing framework can help quite a bit
  - You don't have time to do all that extra work
    - *False* – unit testing reduces debugging time more than the amount spent building the actual tests
- Advantages of writing unit tests:
  - Your program will have fewer bugs
  - More importantly, it will be *a lot* easier to maintain and modify your program
    - This is a *huge* win for programs that, unlike class assignments, get actual use!

JUnit

- **JUnit is a (Java) framework for writing unit tests**
  - **JUnit uses Java's *reflection* capabilities, which allows Java programs to examine their own code**
  - **JUnit helps the programmer:**
    - **Define and execute tests**
    - **Formalize requirements and clarify architecture**
    - **Write and debug code**
    - **Integrate code and always be ready to release a working version**

Terminology

- **A unit test tests the units (methods) in a *single* class**

Terminology

- **A unit test tests the units (methods) in a *single* class**
- **A test case tests the response of a *single* unit (method) to a particular set of inputs**
  - **You can have multiple test cases for a single unit test method**

Terminology

- **A unit test tests the units (methods) in a *single* class**
- **A test case tests the response of a *single* unit (method) to a particular set of inputs**
  - **You can have multiple test cases for a single unit test method**
- **A test runner is software that runs unit tests and reports results**
  - **Eclipse takes care of this**

Terminology

- A unit test tests the units (methods) in a *single* class
- A test case tests the response of a *single* unit (method) to a particular set of inputs
  - You can have multiple test cases for a single unit test method
- A test runner is software that runs unit tests and reports results
  - Eclipse takes care of this
- An integration test is a test of how well classes work together
  - Integration testing (testing that it all works together) is not well supported by Junit and we won't cover this

Assert Methods

- **The unit testing process:**
  - **Call the method being tested in your program and get the actual result**
  - **“Assert” what the correct result should be with one of the assert methods**
  - **Repeat steps as many times as necessary**

Assert Methods

- **The unit testing process:**
  - **Call the method being tested in your program and get the actual result**
  - **“Assert” what the correct result should be with one of the assert methods**
  - **Repeat steps as many times as necessary**
- **An assert method is a JUnit method that performs a test, and throws an AssertionError if the test fails**
  - **JUnit catches these Errors and shows you the result**

Assert Methods

- **The unit testing process:**
  - **Call the method being tested in your program and get the actual result**
  - **"Assert" what the correct result should be with one of the assert methods**
  - **Repeat steps as many times as necessary**
- **An assert method is a JUnit method that performs a test, and throws an AssertionError if the test fails**
  - **JUnit catches these Errors and shows you the result**
- **Some assert methods:**

  void assertTrue(boolean *test*)
  void assertTrue(boolean *test,* String *message*)
  - **Throws an *AssertionError* if the test fails**
  - **The optional *message* is included in the Error**

  void assertFalse(boolean *test*)
  void assertFalse(boolean *test,* String *message*)
  - **Throws an *AssertionError* if the test fails**
  - **The optional *message* is included in the Error**

Example - Counter Class

- **As an example, let's look at a trivial "Counter" class**
  - **The class will declare a counter (int) and initialize it to zero**
  - **The *increment* method will add one to the counter and return the new value**
  - **The *decrement* method will subtract one from the counter and return the new value**

Example - Counter Class

- **As an example, let's look at a trivial "Counter" class**
  - **The class will declare a counter (int) and initialize it to zero**
  - **The *increment* method will add one to the counter and return the new value**
  - **The *decrement* method will subtract one from the counter and return the new value**
- **We usually write the program method stubs first, and let the IDE generate the test method stubs**

Example - Counter Class

- **As an example, let's look at a trivial "Counter" class**
  - **The class will declare a counter (int) and initialize it to zero**
  - **The *increment* method will add one to the counter and return the new value**
  - **The *decrement* method will subtract one from the counter and return the new value**
- **We usually write the program method stubs first, and let the IDE generate the test method stubs**
- **Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself**

Example - Counter Class

```
public class Counter {

    int count = 0;

    public int increment() {
        this.count += 1;
        return this.count;
    }

    public int decrement() {
        this.count -= 1;
        return this.count;
    }

    public int getCount() {
        return this.count;
    }
}
```

- Is JUnit testing overkill for this little class?
- Doesn't matter, writing JUnit tests for trivial classes is no big deal
- Often, you won't write tests for simple "getter" methods like *getCount*

Example – JUnit Tests for Counter Class

```
public class CounterTest {

    Counter counter1; //declare a Counter for testing here


    @BeforeEach
    void setUp() throws Exception {
        this.counter1 = new Counter(); //initialize the Counter here
    }

    @Test
    void testIncrement() {
        assertTrue(this.counter1.increment() == 1);
        assertTrue(this.counter1.increment() == 2);
        assertEquals(3, this.counter1.increment());
    }

    @Test
    void testDecrement() {
        assertEquals(-1, this.counter1.decrement());
        assertTrue(this.counter1.decrement() == -2);
    }
}
```

- The *setUp* method (annotated by @BeforeEach) runs before every unit test method
- Each unit test (annotated by @Test) begins with a *brand new* counter
- You shouldn't be concerned with the order in which unit test methods run

Counter Project

Create Counter Class

Counter.java

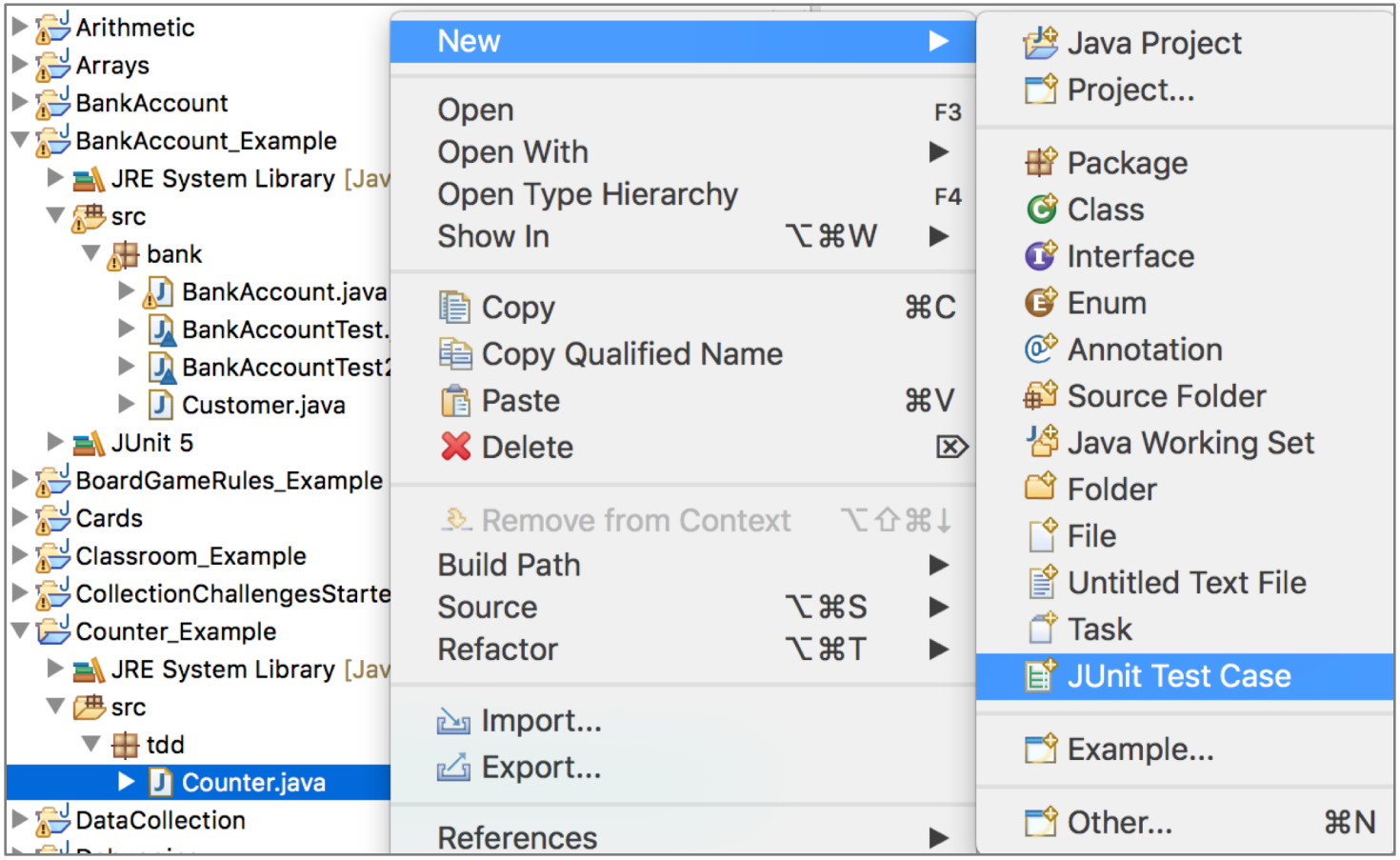
```
1  package tdd;
2
3  /**
4   * Represents a simple counter.
5   * @author brandonkrakowsky
6   *
7   */
8  public class Counter {
9
10     //instance variables
11     int count;
12
13     //constructor
14
15     //methods
16
```

Create Counter Class

```
13      //constructor
14
15      //methods
16
17      /**
18       * Increments the count by 1.
19       * @return new count
20       */
21      public int increment() {
22          this.count++;
23
24          return this.count;
25      }
26
27      /**
28       * Decrements the count by 1.
29       * @return new count
30       */
31      public int decrement() {
32          this.count--;
33
34          return this.count;
35      }
36
37      /**
38       * Gets the current count.
39       * @return int value of count
40       */
41      public int getCount() {
42          return this.count;
43      }
44
45  }
```

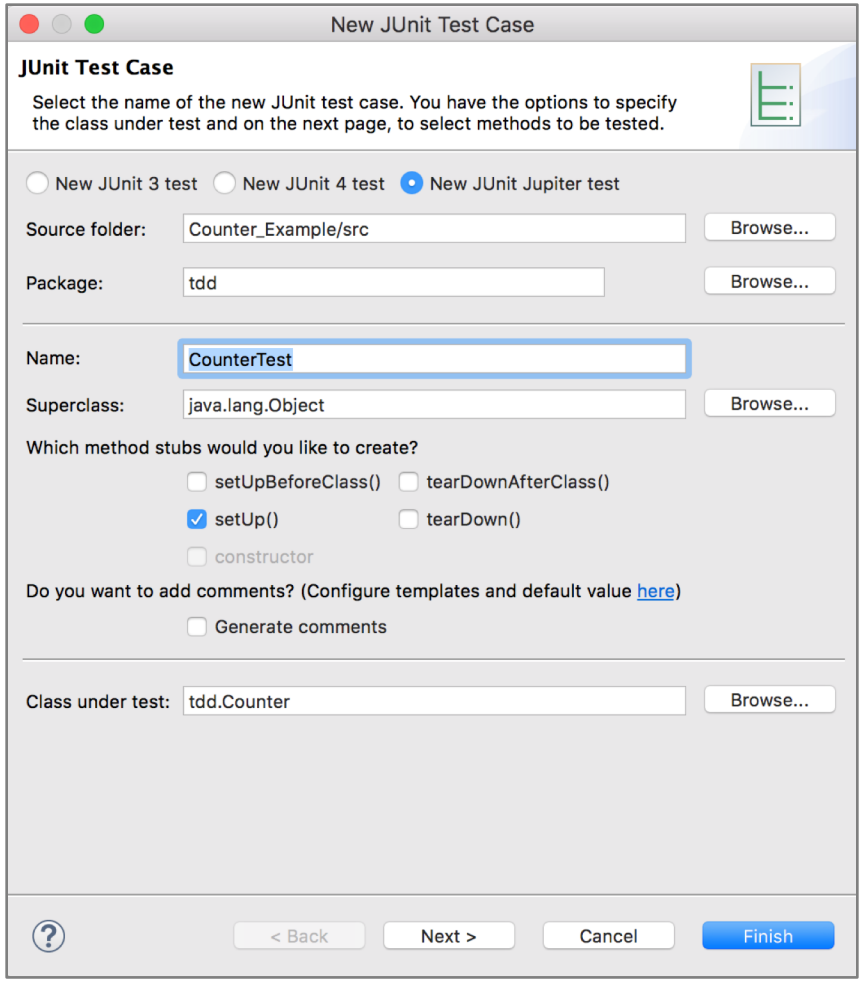
Create JUnit Tests

- **Select the class file, and go to "New" → "JUnit Test Case"**



Create JUnit Tests

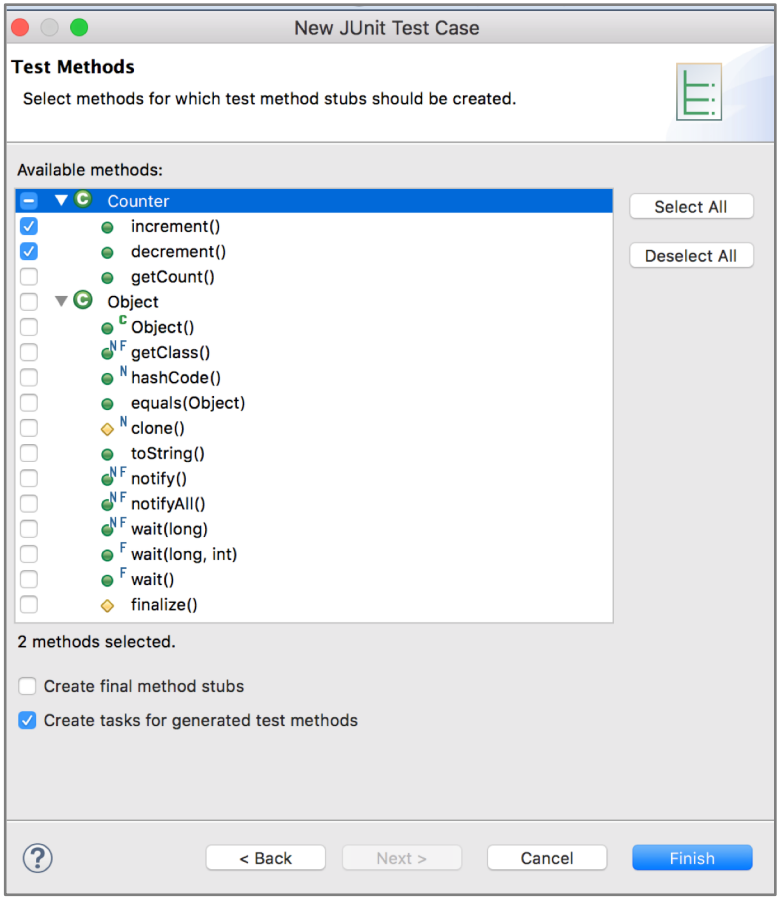
- **Use the default name provided for your JUnit Test Case class**



Make sure setUp() is checked

Create JUnit Tests

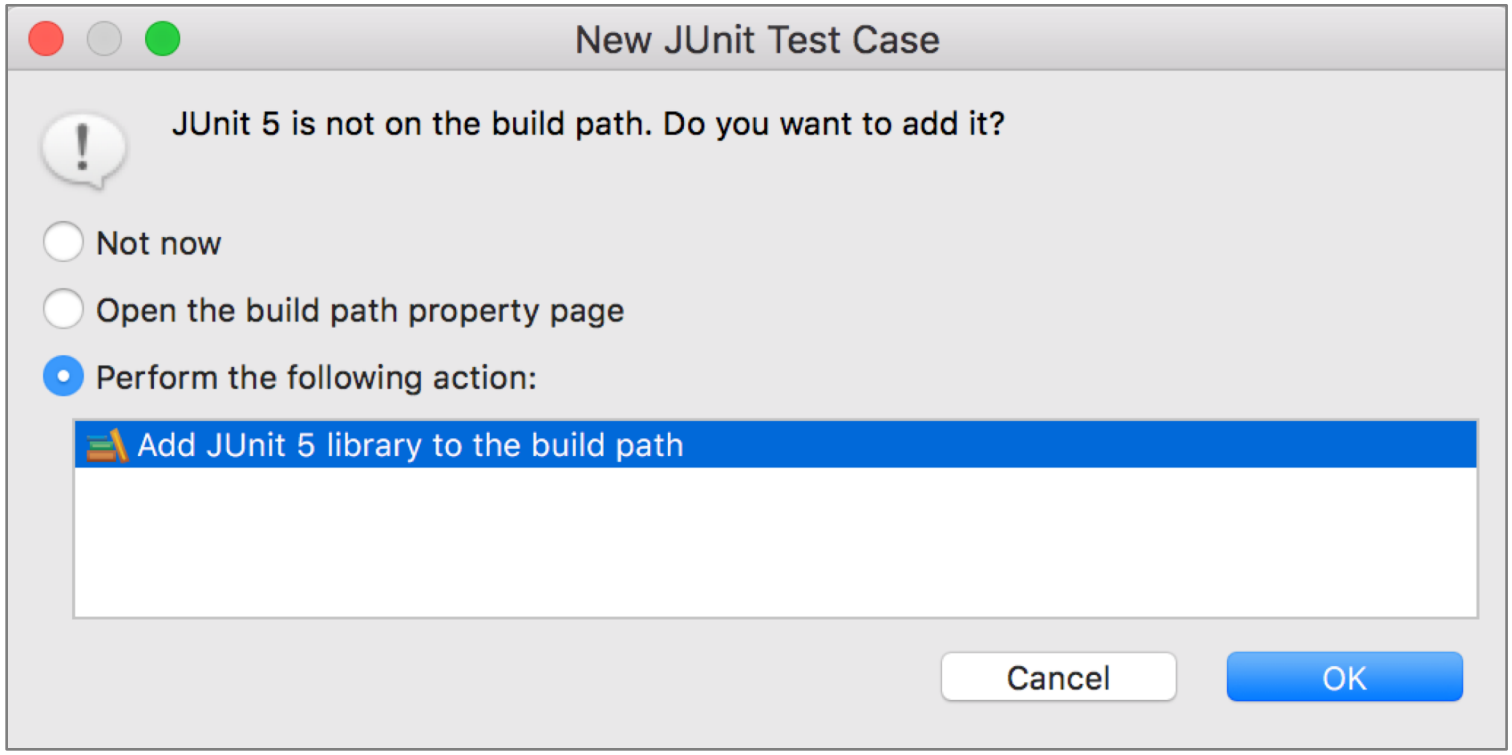
- **To have Eclipse generate test method stubs for you, use the checkboxes to decide which methods you want test cases for. Don't select Object or anything under it.**



Check create tasks for generated test methods

Create JUnit Tests

- **Add the JUnit 5 library to the project build path**
  - **This includes the necessary JUnit framework in your project**



Create JUnit Tests

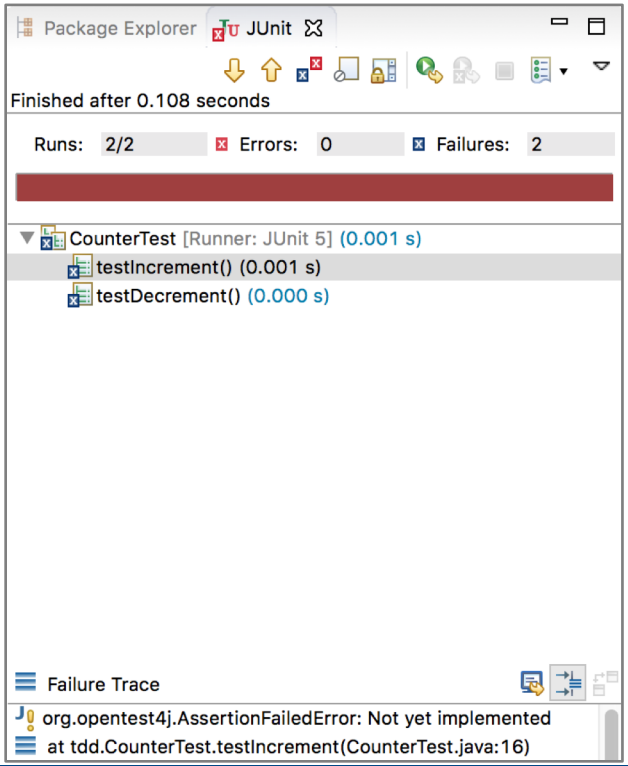
- **Eclipse will add a new JUnit Test class in the same package**
  - **You'll see test method stubs to be implemented**
  - **The code in each test method is calling *fail* (with a message), to force the test methods to initially fail**

Counter.java | CounterTest.java

```
1  package tdd;
2
3  import static org.junit.jupiter.api.Assertions.*;
7
8  class CounterTest {
9
10     @BeforeEach
11     void setUp() throws Exception {
12     }
13
14     @Test
15     void testIncrement() {
16         fail("Not yet implemented"); // TODO
17     }
18
19     @Test
20     void testDecrement() {
21         fail("Not yet implemented"); // TODO
22     }
23
24 }
25
```

Create JUnit Tests

- If you run the tests, they should ALL fail
  - Eclipse will open the JUnit panel (on the left)
    - The top bar will show red
    - The number next to “Failures” will show 2
    - The message at the bottom will explain why the tests failed



Create JUnit Tests

- **Implement the test methods, adding test cases with assert methods**

Counter.java | CounterTest.java

```
1  package tdd;
2
3  import static org.junit.jupiter.api.Assertions.*;
9
10 class CounterTest {
11
12     Counter counter1; //declare a Counter for testing here
13
14     @BeforeEach
15     void setUp() {
16         this.counter1 = new Counter(); //initialize the Counter here
17     }
18
19     @Test
20     public void testIncrement() {
21         //asserts that calling increment returns 1
22         assertTrue(this.counter1.increment() == 1);
23
24         //asserts that calling increment returns 2
25         assertTrue(this.counter1.increment() == 2);
26
27         //asserts that calling increment again does not return 2
28         this.counter1.increment();
29         assertFalse(this.counter1.count == 2, "should not return 2 after incrementing again");
30
31         //asserts that 3 is equal to the new count
32         assertEquals(3, this.counter1.count);
33
34         //asserts that 3 is not equal to calling increment again
35         assertNotEquals(3, this.counter1.increment());
36     }
```

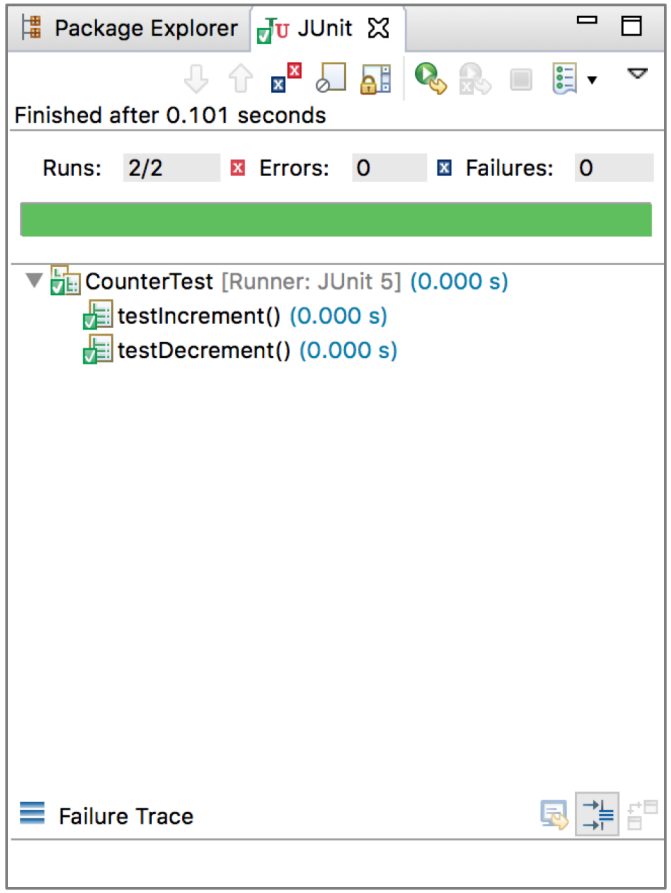
Create JUnit Tests

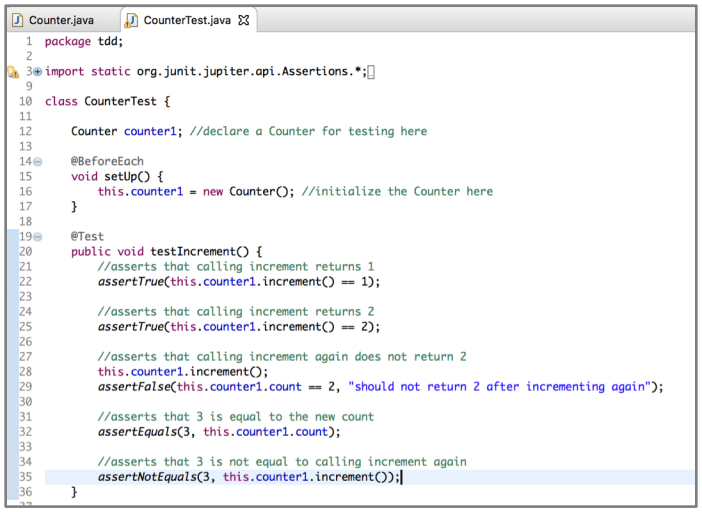
- **Implement the test methods, adding test cases with assert methods**

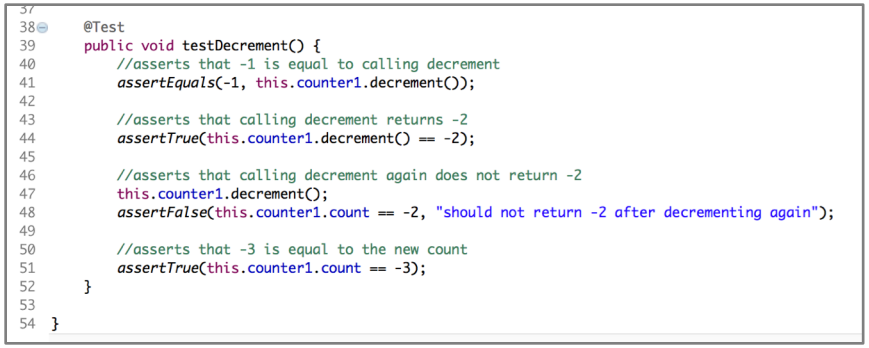
```
37
38      @Test
39      public void testDecrement() {
40          //asserts that -1 is equal to calling decrement
41          assertEquals(-1, this.counter1.decrement());
42
43          //asserts that calling decrement returns -2
44          assertTrue(this.counter1.decrement() == -2);
45
46          //asserts that calling decrement again does not return -2
47          this.counter1.decrement();
48          assertFalse(this.counter1.count == -2, "should not return -2 after decrementing again");
49
50          //asserts that -3 is equal to the new count
51          assertTrue(this.counter1.count == -3);
52      }
53
54  }
```

Create JUnit Tests

- **If you run the tests, they should ALL pass**
  - **In the JUnit panel (on the left) – the top bar will show green**
  - **The number next to “Failures” will show 0**







Testing for Equality in Java

- **In Java, you use == to compare *primitives***
- **For example:**

```
//e will be set to true if 2 is equal to 3
boolean e = (2 == 3);
```

Testing for Equality in Java

- **In Java, you use == to compare *primitives***
- **For example:**

```
//e will be set to true if 2 is equal to 3
boolean e = (2 == 3);
```

- **And you use the method *x.equals(y)* to compare *objects***
- **For example:**

```
//e is set to true if “thisString” is equal to “thatString”
boolean e = “thisString”.equals(“thatString”);
```

Testing for Equality in Java

- **Rule: When comparing a literal String value (known String value) to an unknown String value, use the *equals* method of the known value**
- **For example:**

```
//e is set to true if "thisString" is equal to String value stored in someUnknownString
boolean e = "thisString".equals(someUnknownString);
```

Testing for Equality in Java

- **Rule: When comparing a literal String value (known String value) to an unknown String value, use the *equals* method of the known value**
- **For example:**

```
//e is set to true if "thisString" is equal to String value stored in someUnknownString
boolean e = "thisString".equals(someUnknownString);
```

- **Why?**
  - **Because you know, at least, that "thisString" exists and is not null, so it MUST have an *equals* method**
  - ***someUnknownString*, on the other hand could be null, in which case calling it's *equals* method will return an error**

Testing for Equality in Java

- **Why is all of this important?**
  - **The JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *objects***

Testing for Equality in Java

- **Why is all of this important?**
  - **The JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *objects***
- **To define *equals* for your own objects, you'll have to define *exactly* this method in your class:**

  public boolean equals(Object *obj*) { ... }
  - **The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (e.g. Person):**

Testing for Equality in Java

- **Why is all of this important?**
  - **The JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *objects***
- **To define *equals* for your own objects, you'll have to define *exactly* this method in your class:**

  public boolean equals(Object *obj*) { ... }
  - **The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (e.g. Person):**
- **Here's a full (sample) implementation of *equals* inside a class**

```
public boolean equals(Object something) {
    Person p = (Person)something;
    return this.name == p.name; //test whatever you like here
}
```

Testing for Equality in Java

- **Why is all of this important?**
  - **The JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *objects***
- **To define *equals* for your own objects, you'll have to define *exactly* this method in your class:**

  public boolean equals(Object *obj*) { ... }
  - **The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (e.g. Person):**
- **Here's a full (sample) implementation of *equals* inside a class**

```
public boolean equals(Object something) {
    Person p = (Person)something;
    return this.name == p.name; //test whatever you like here
}
```

- **We'll talk more about implementing methods, like *equals*, later in the course**

More Assert Methods

void assertEquals(*expected*, *actual*)
void assertEquals(*expected*, *actual,* String *message*)

- ***expected* and *actual* must both be objects *or* the same primitive type**
- **For primitives, this method compares using ==**
- **For objects, this method compares using the *equals* method**
  - **For your own objects, you'll need to define the *equals* method properly (as described on previous slide)**

More Assert Methods

void assertEquals(*expected*, *actual*)
void assertEquals(*expected*, *actual,* String *message*)

- ***expected* and *actual* must both be objects *or* the same primitive type**
- **For primitives, this method compares using ==**
- **For objects, this method compares using the *equals* method**
  - **For your own objects, you'll need to define the *equals* method properly (as described on previous slide)**

void assertArrayEquals(int[] *expected*, int[] *actual*)
void assertArrayEquals(int[] *expected*, int[] *actual,* String *message*)

- **Asserts that two int arrays are equal**

Assert Methods with *doubles*

- **Note: When you want to compare floating point types (e.g. double or float), you should use *assertEquals* with the additional parameter *delta* to avoid problems with round-off errors while doing floating point comparisons**
- **The assert method syntax to use is:**

  void assertEquals(double *expected*, double *actual*, double *delta*)

  - **This asserts that *expected* and *actual* are equal, within the given *delta***
  - ***delta* is a very small double (e.g. 0.000001) used for comparison**

Assert Methods with *doubles*

- **Note: When you want to compare floating point types (e.g. double or float), you should use *assertEquals* with the additional parameter *delta* to avoid problems with round-off errors while doing floating point comparisons**
- **The assert method syntax to use is:**

  void assertEquals(double *expected*, double *actual*, double *delta*)

  - **This asserts that *expected* and *actual* are equal, within the given *delta***
  - ***delta* is a very small double (e.g. 0.000001) used for comparison**

- **For example:**

  void assertEquals(aDoubleValue, anotherDoubleValue, 0.000001)

  - **This evaluates to: Math.abs(expected – actual) <= delta**

More Assert Methods

void assertSame(Object *expected*, Object *actual*)
void assertSame(Object *expected*, Object *actual,* String *message*)

- **Asserts that two arguments refer to the *same* object**

More Assert Methods

void assertSame(Object *expected*, Object *actual*)
void assertSame(Object *expected*, Object *actual,* String *message*)

- **Asserts that two arguments refer to the *same* object**

void assertNotSame(Object *expected*, Object *actual*)
void assertNotSame(Object *expected*, Object *actual,* String *message*)

- **Asserts that two objects do not refer to the same object**

More Assert Methods

void assertNull(Object *object*)
void assertNull(Object *object,* String *message*)

- **Asserts that the object is null (undefined)**

More Assert Methods

void assertNull(Object *object*)
void assertNull(Object *object,* String *message*)

- **Asserts that the object is null (undefined)**

void assertNotNull(Object *object*)
void assertNotNull(Object *object,* String *message*)

- **Asserts that the object is not null**

More Assert Methods

void assertNull(Object *object*)
void assertNull(Object *object,* String *message*)

- **Asserts that the object is null (undefined)**

void assertNotNull(Object *object*)
void assertNotNull(Object *object,* String *message*)

- **Asserts that the object is not null**

fail()
fail(String *message*)

- **Causes the test to fail and throw an *AssertionFailedError***

More Assert Methods

```
void assertThrows(Exception.class, () -> {
    //code that throws an exception
});
```

- **Asserts that the enclosed code throws an Exception of a particular type**

More Assert Methods

```
void assertThrows(Exception.class, () -> {
   //code that throws an exception
});
```

- Asserts that the enclosed code throws an Exception of a particular type

For example:

```
String test = null;
assertThrows(NullPointerException.class, () -> {
   test.length();
});
```

- Asserts that *test.length()* throws a NullPointerException
- Why? *test* is null, so there is no method *length()*

Bank Account Project

Bank Account Project

- In Eclipse, create a new “Bank_Account” project
- Create 2 classes:
  - BankAccount
    - Provide the package name “bank”
    - Make sure public static void main(String[] args) IS checked
  - Customer
    - Provide the package name “bank”
    - Make sure public static void main(String[] args) IS NOT checked

BankAccount Class – Instance Variables

BankAccount.java | Customer.java

```
1  package bank;
2
3  /**
4   * Class representing a bank account.
5   * @author brandonkrakowsky
6   *
7   */
8  public class BankAccount {
9
10     //Instance variables
11     String accountType;
12     double balance;
13     Customer customer;
```

Customer Class – Instance Variables

BankAccount.java | Customer.java

```
1  package bank;
2
3  /**
4   * Class representing a customer for a bank account.
5   * @author brandonkrakowsky
6   *
7   */
8  public class Customer {
9
10     //instance variables
11     String name;
12     BankAccount checkingBankAccount;
13     BankAccount savingsBankAccount;
14
```

Customer Class - Constructor

```
14
15      //constructor(s)
16      /**
17       * Creates a new customer with the given name.
18       * @param name for customer
19       */
20      public Customer(String name) {
21          this.name = name;
22      }
23
```

Customer Class - Methods

```
24      //methods
25      /**
26       * Sets this customer's checking account.
27       * @param checkingBankAccount for customer
28       */
29      public void setCheckingBankAccount(BankAccount checkingBankAccount) {
30          this.checkingBankAccount = checkingBankAccount;
31      }
32
33      /**
34       * Sets this customer's savings account.
35       * @param savingsBankAccount for customer
36       */
37      public void setSavingsBankAccount(BankAccount savingsBankAccount) {
38          this.savingsBankAccount = savingsBankAccount;
39      }
40
```

BankAccount Class - Constructor

```
14
15      //Constructors
16      /**
17       * Creates a new bank account for the given customer.
18       * @param accountType Type of account ("checking"/"savings")
19       * @param customer for account
20       */
21      public BankAccount(String accountType, Customer customer) {
22          //set type of account
23          this.accountType = accountType;
24
25          //set the customer
26          this.customer = customer;
27
28          //set the customer's checking/savings account to this bank account
29          //"this" means this instance of BankAccount
30          if ("checking".equals(accountType)) {
31              this.customer.setCheckingBankAccount(this);
32          } else if ("savings".equals(accountType)) {
33              this.customer.setSavingsBankAccount(this);
34          }
35
36      }
37
```

BankAccount Class - Methods

```
36      //Methods
37      /**
38       * Deposit given amount into this account.
39       * @param amount to deposit
40       */
41      public void deposit(double amount) {
42          this.balance += amount;
43      }
44
45      /**
46       * Withdraw given amount from this account.
47       * @param amount to withdraw
48       */
49      public void withdraw(double amount) {
50          this.balance -= amount;
51      }
52
```

BankAccount Class - Methods

```
53      /**
54       * Transfer given amount from this account to otherAccount.
55       * @param otherAccount to deposit
56       * @param amount to transfer
57       */
58      public void transfer(BankAccount otherAccount, double amount) {
59          this.withdraw(amount);
60          otherAccount.deposit(amount);
61      }
62
```

BankAccount Class – *main* Method

```
63      public static void main(String[] args) {
64
65          //Create a customer
66          Customer customer = new Customer("Brandon");
67
68          //Create some accounts
69          BankAccount checkingAccount = new BankAccount("checking", customer);
70          BankAccount savingsAccount = new BankAccount("savings", customer);
71
72          //Initially check customer's accounts
73          System.out.println(customer.name + " checking account: " + checkingAccount.balance);
74          System.out.println(customer.name + " savings account: " + savingsAccount.balance);
75
```

BankAccount Class – *main* Method

```
74
75          //Make some deposits
76          checkingAccount.deposit(123.45);
77          savingsAccount.deposit(10000.00);
78
79          //Check balances
80          System.out.println("----------");
81          System.out.println("Balances:");
82          System.out.println(checkingAccount.balance);
83          System.out.println(savingsAccount.balance);
84
```

BankAccount Class – *main* Method

```
84
85          //Make some withdrawals
86          checkingAccount.withdraw(23.02);
87          savingsAccount.withdraw(200);
88
89          //Check balances
90          System.out.println("----------");
91          System.out.println("Balances:");
92          System.out.println(checkingAccount.balance);
93          System.out.println(savingsAccount.balance);
94
```

Create JUnit Tests for the Bank Account Program

Customer.java | BankAccount.java | BankAccountTest.java

```
1  package bank;
2
3  import static org.junit.jupiter.api.Assertions.*;..
7
8  class BankAccountTest {
9
10     //declare this.customer and accounts for testing here
11     Customer customer;
12     BankAccount myCheckingAccount;
13     BankAccount mySavingsAccount;
14
15     @BeforeEach
16     void setUp() throws Exception {
17         //initialize this.customer and accounts for testing here
18         this.customer = new Customer("Brandon");
19         this.myCheckingAccount = new BankAccount("checking", this.customer);
20         this.mySavingsAccount = new BankAccount("savings", this.customer);
21     }
22
```

Create JUnit Tests for the Bank Account Program

```
23      @Test
24      void testDeposit() {
25          //make a deposit
26          this.myCheckingAccount.deposit(100);
27
28          //check balance of checking
29          assertEquals(100, this.myCheckingAccount.balance);
30      }
31
32      @Test
33      void testWithdraw() {
34          //make a deposit
35          this.myCheckingAccount.deposit(100);
36
37          //make a withdrawal
38          this.myCheckingAccount.withdraw(50);
39
40          //check balance of checking
41          assertEquals(50, this.myCheckingAccount.balance);
42      }
43
```

Create JUnit Tests for the Bank Account Program

```
44      @Test
45      void testTransfer() {
46          //make a deposit
47          this.mySavingsAccount.deposit(100);
48
49          //transfer money from savings to checking
50          this.mySavingsAccount.transfer(this.myCheckingAccount, 50);
51
52          //check balance of checking
53          assertEquals(50, this.myCheckingAccount.balance);
54
55          //check balance of savings
56          assertEquals(50, this.mySavingsAccount.balance);
57      }
58
```