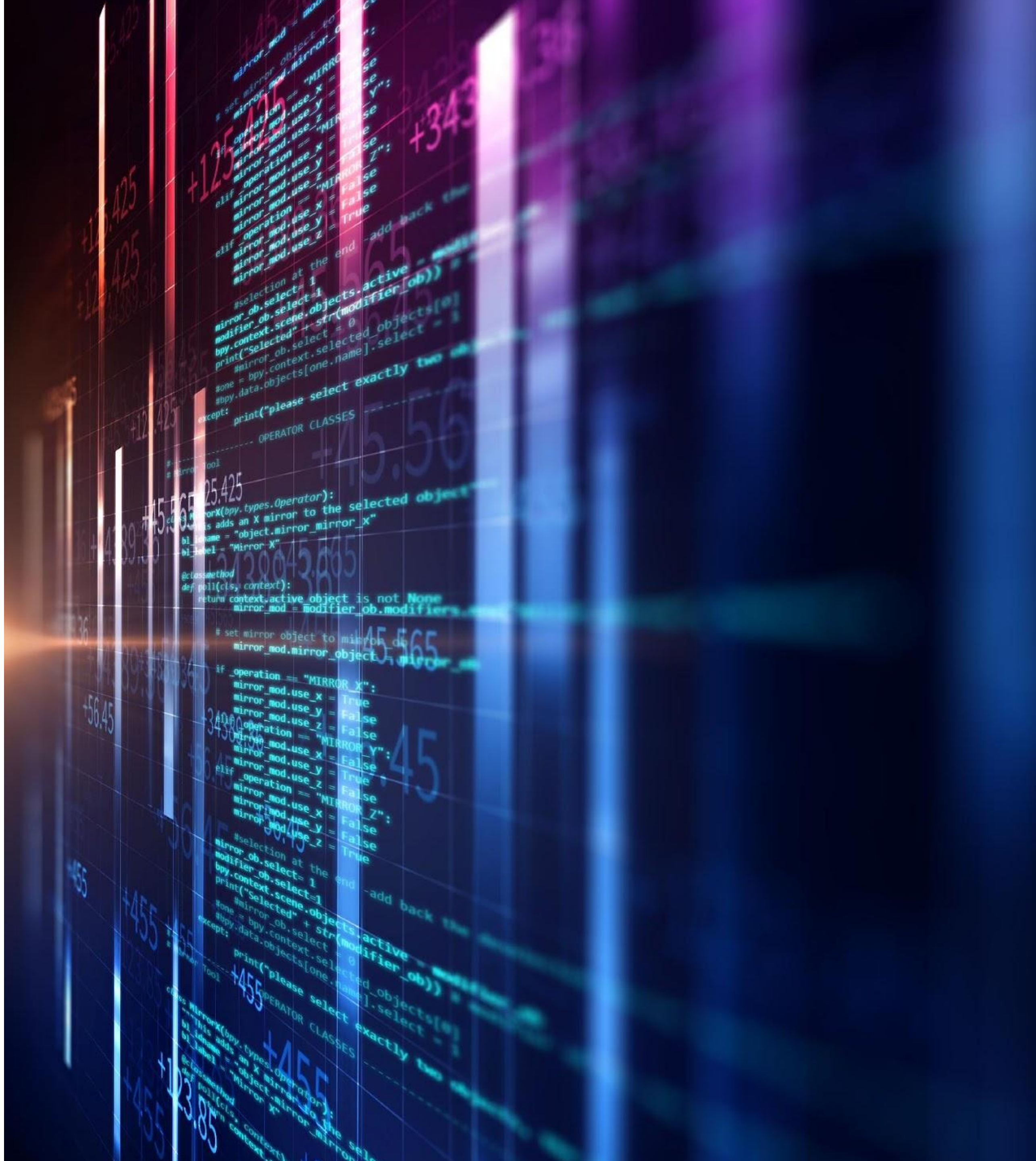
Unit Testing in Java

Brandon Krakowsky

Penn Engineering



Unit Testing

Unit Testing

- Bottom line, 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
  - For example:
    - Calling random methods with different inputs from your *main* method and printing/comparing the results
    - Or running your program and trying different inputs for a *Scanner*
- Or you *can write* a set of unit tests that can be run at any time
  - This will always test your code in the same ways
  - It’s just like the testing class we write to test our main program in Python

Unit Testing

- The *disadvantages* of writing unit tests:
    - It *can* require (a lot of) extra programming
        - But use of a good testing framework can help with the process
    - You don’t have time to do all that extra work
        - But testing reduces debugging time more than the amount of time spent building the actual tests
- The *advantages* of writing unit tests:
    - Guaranteed, your program will have fewer bugs
    - It will be a lot easier to maintain and modify your program
        - This is a huge win for programs that get actual use in production!

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 program 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
- A test case tests the response of a *single* unit (method) to a particular set of inputs
  - You can (and should) have multiple test cases for a single unit test method
- An integration test is a test of how well classes and methods 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
- 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

- 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
- A good approach is to write the program method stubs first, and let Eclipse 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
- Note: Often, you won’t write tests for simple “getter” methods like *getCount*

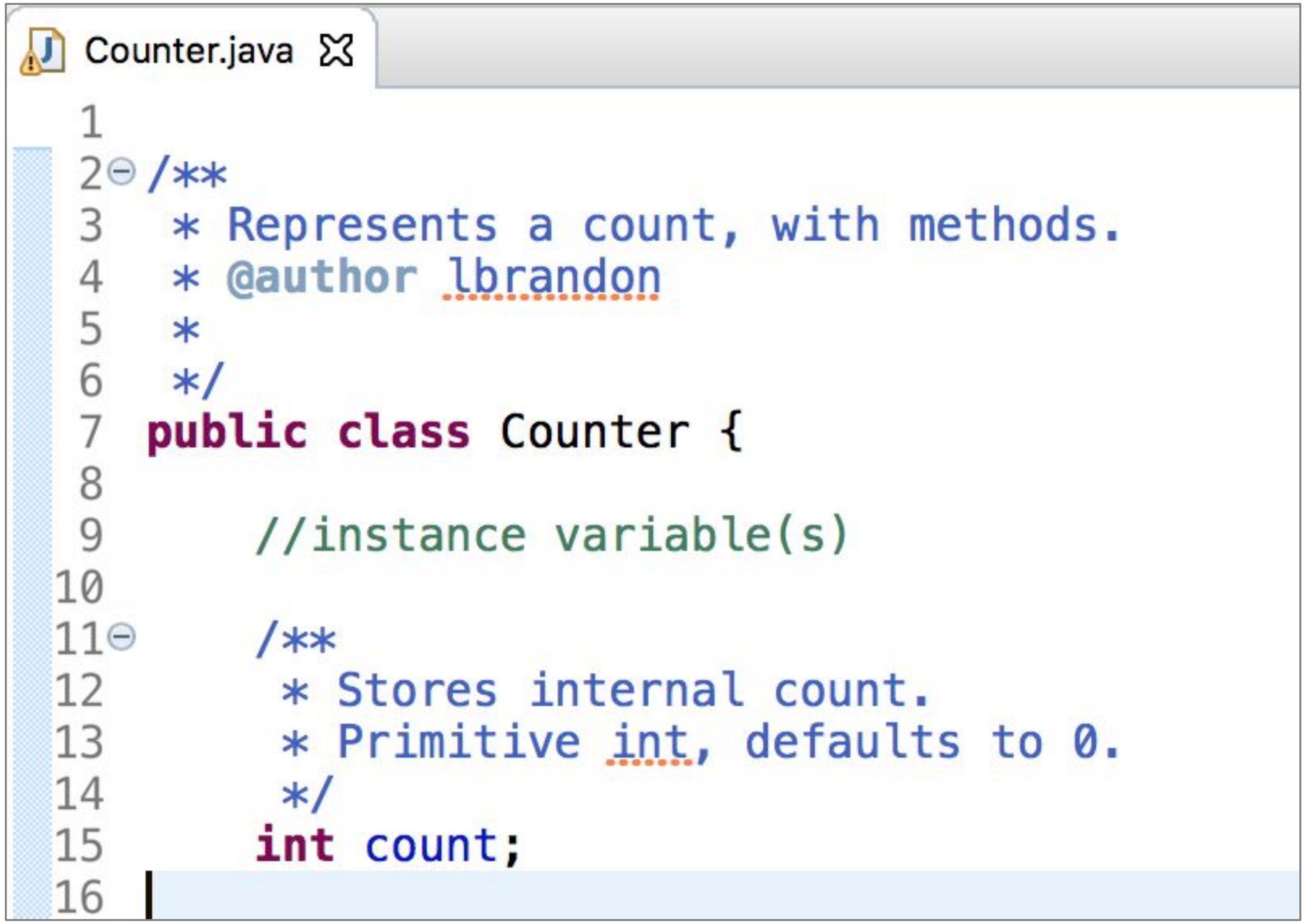
Example – JUnit Tests for Counter Class

```
public class CounterTest {

    Counter counter1; //declare a Counter for testing


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

    @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 each unit test method
    - This is just like the *setUp(self)* testing function in Python
- Each unit test (annotated by @Test) begins with a *brand new* counter
- Note: You can't be concerned with the order in which unit test methods run

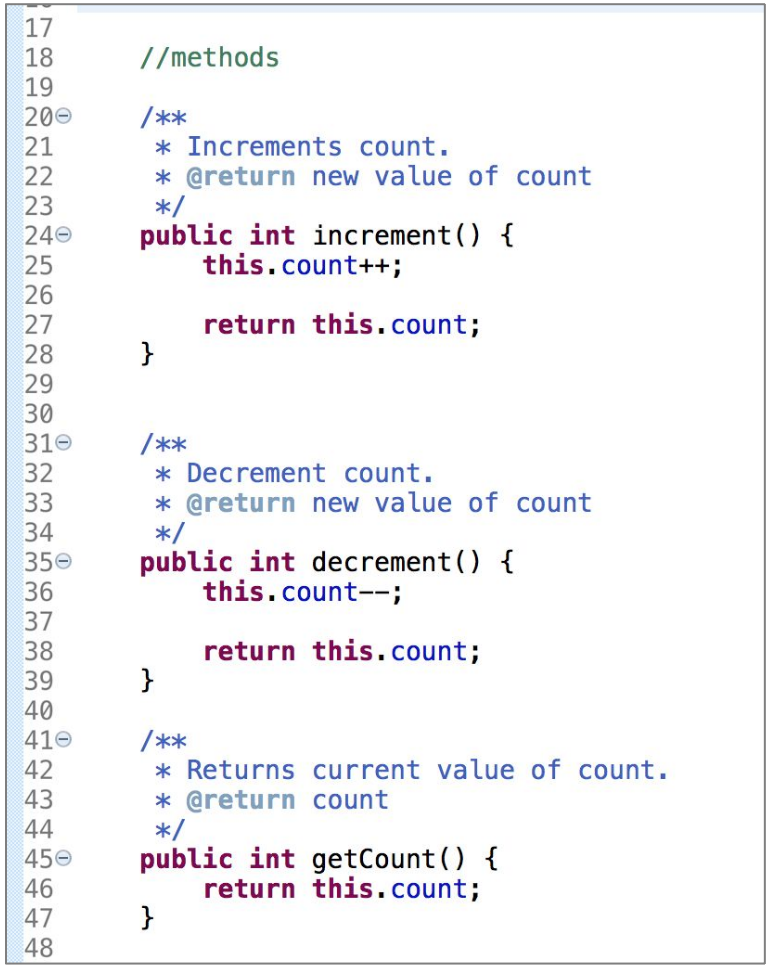
Counter Project

Create Counter Class



```java

/**
 * Represents a count, with methods.
 * @author lbrandon
 *
 */
public class Counter {

    //instance variable(s)

    /**
     * Stores internal count.
     * Primitive int, defaults to 0.
     */
    int count;

```

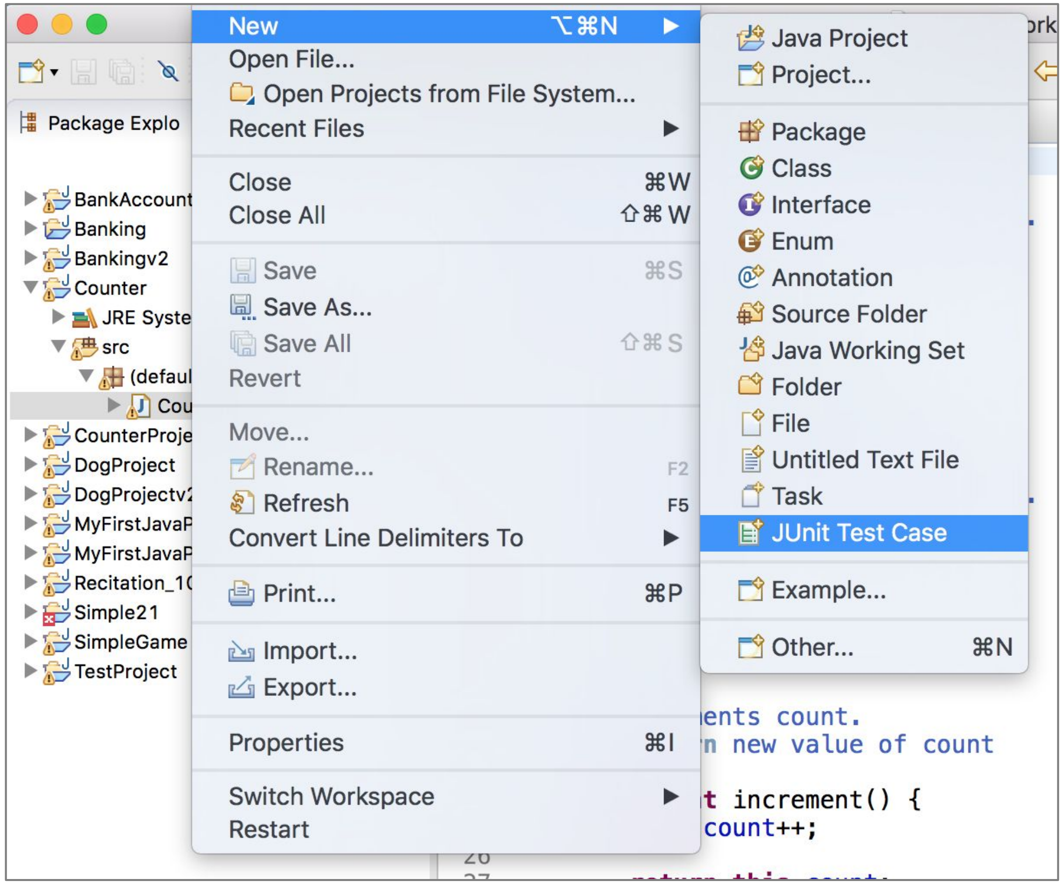
Create Counter Class



```
    //methods

    /**
     * Increments count.
     * @return new value of count
     */
    public int increment() {
        this.count++;

        return this.count;
    }


    /**
     * Decrement count.
     * @return new value of count
     */
    public int decrement() {
        this.count--;

        return this.count;
    }

    /**
     * Returns current value of count.
     * @return count
     */
    public int getCount() {
        return this.count;
    }

```

Create JUnit Tests

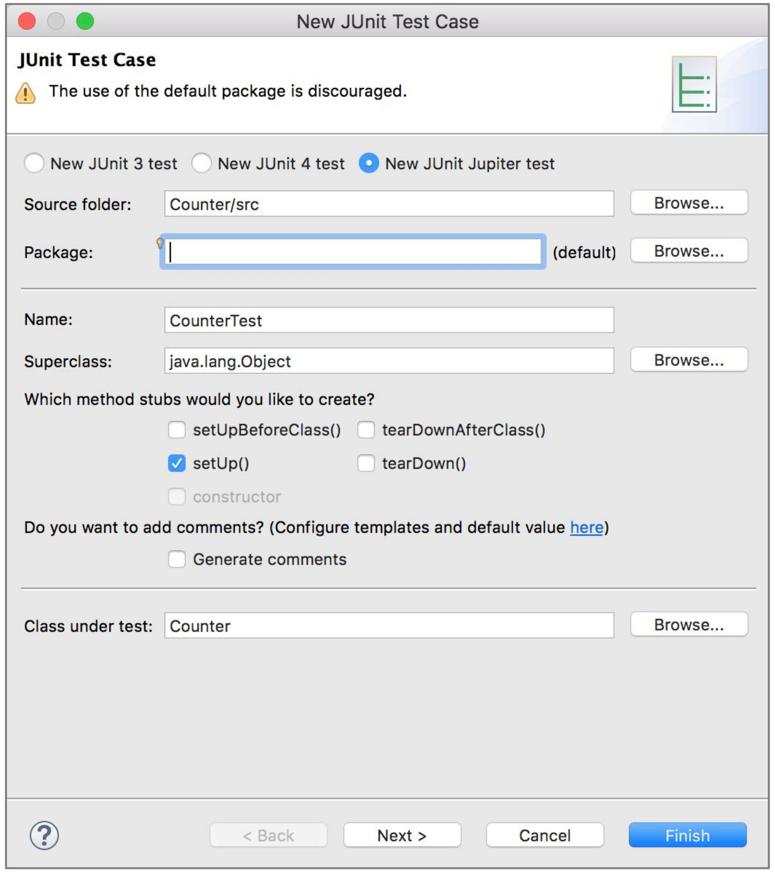
- Select the class file in the Package Explorer, and go to “New” □ “JUnit Test Case”



```
ents count.
n new value of count

t increment() {
count++;
```

Create JUnit Tests

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



New JUnit Test Case

JUnit Test Case

⚠ The use of the default package is discouraged.

○ New JUnit 3 test ○ New JUnit 4 test ● New JUnit Jupiter test

Source folder: Counter/src

Package: (default)

Name: CounterTest

Superclass: java.lang.Object

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()

☑ setUp() □ tearDown()

□ constructor

Do you want to add comments? (Configure templates and default value here)

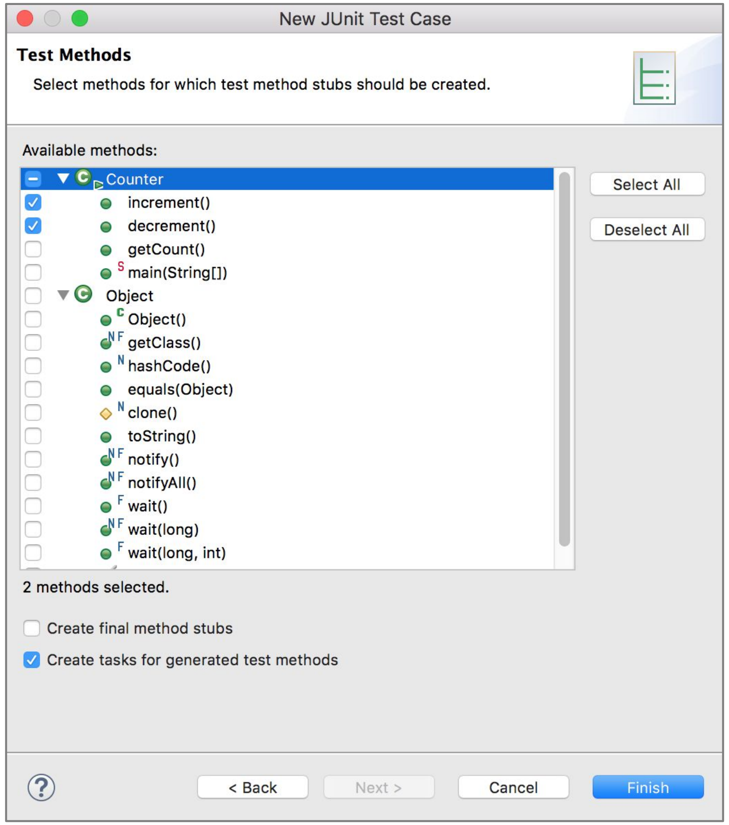
□ Generate comments

Class under test: Counter

- 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.



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- ⊟ Counter
  - ☑ increment()
  - ☑ decrement()
  - ☐ getCount()
  - ☐ S main(String[])
- ☐ Object
  - ☐ C Object()
  - ☐ N F getClass()
  - ☐ N hashCode()
  - ☐ equals(Object)
  - ☐ N clone()
  - ☐ toString()
  - ☐ N F notify()
  - ☐ N F notifyAll()
  - ☐ F wait()
  - ☐ N F wait(long)
  - ☐ F wait(long, int)

2 methods selected.

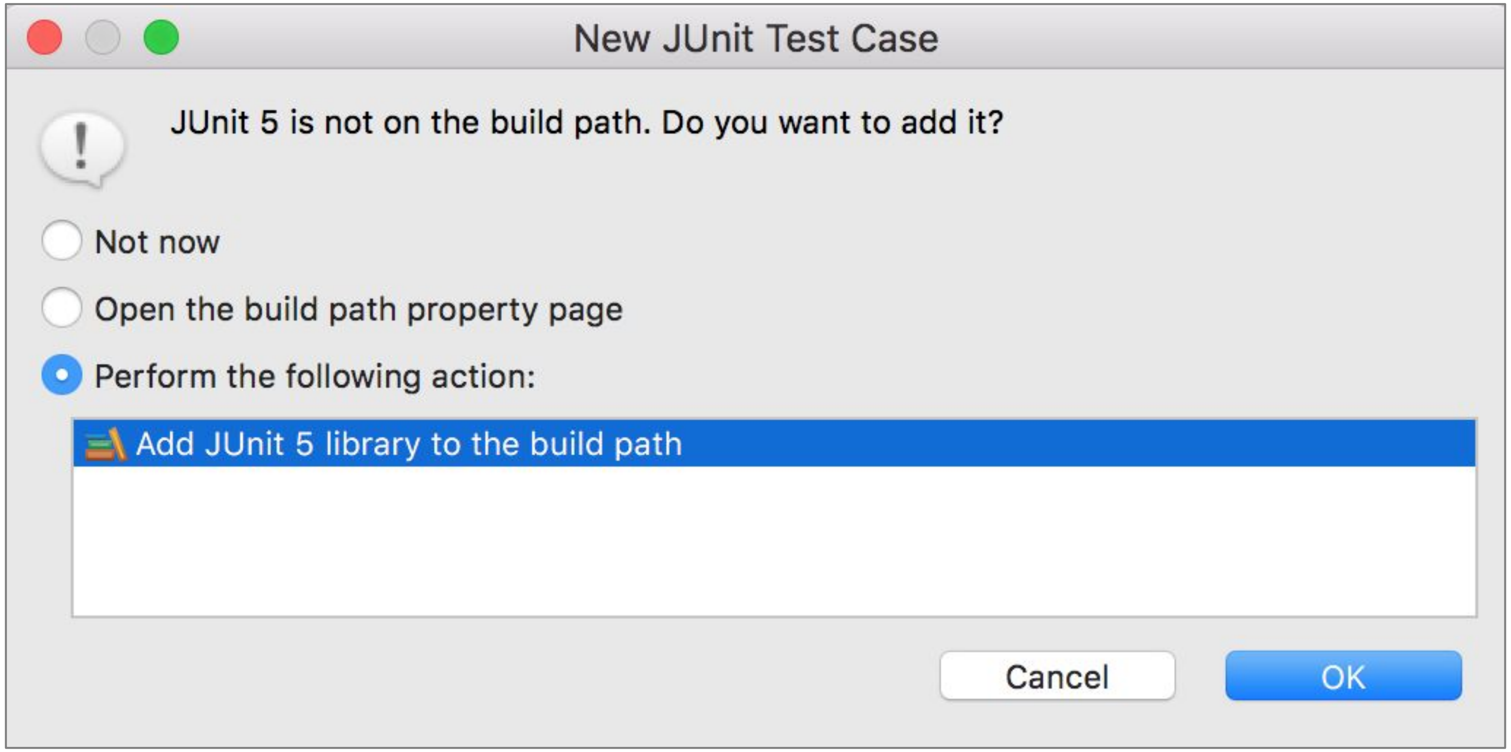
☐ Create final method stubs

☑ Create tasks for generated test methods

- 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



New JUnit Test Case

JUnit 5 is not on the build path. Do you want to add it?

○ Not now

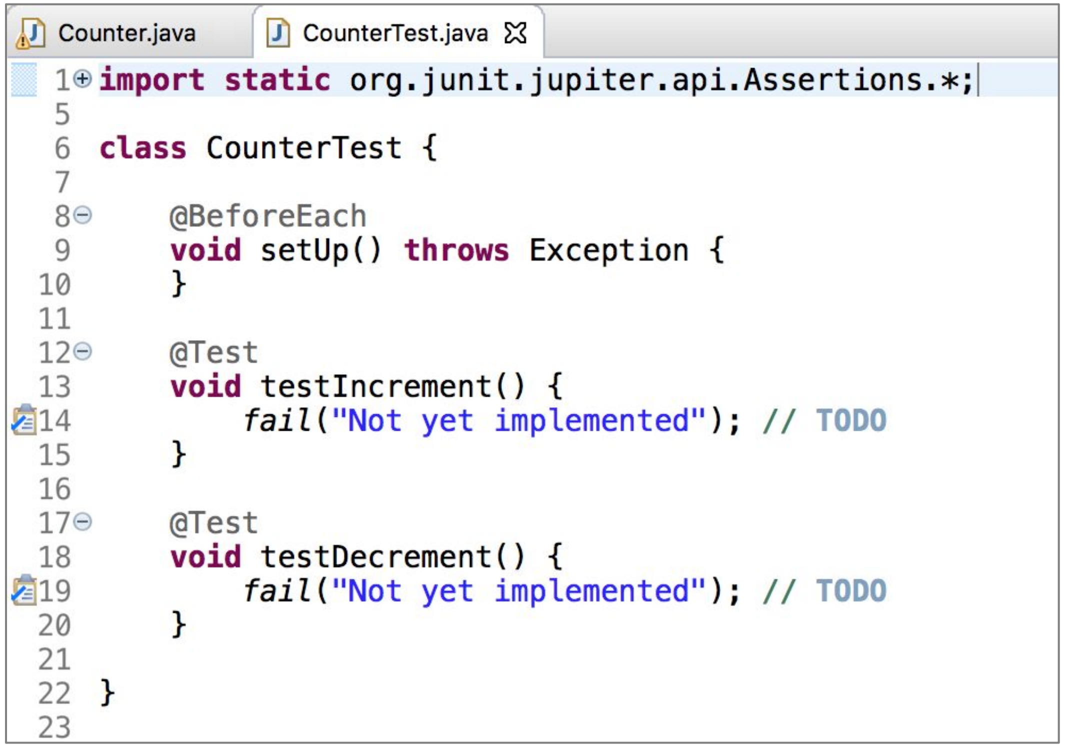
○ Open the build path property page

● Perform the following action:

Add JUnit 5 library to the build path

Create JUnit Tests

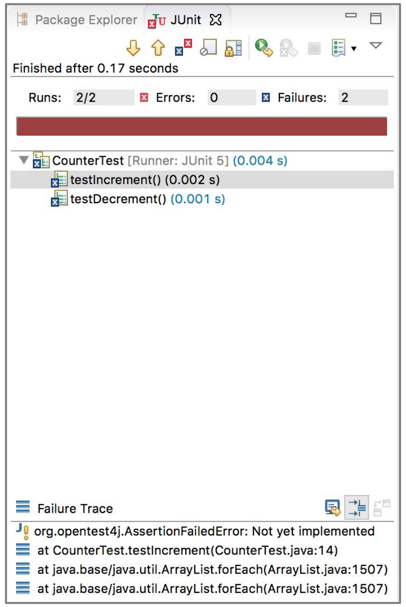
- Eclipse will add a new JUnit Test class in the same package (or default)
  - You'll see test method stubs to be implemented
  - The code in each test method is calling *fail* (with a message), to force the test methods to initially fail



```java
import static org.junit.jupiter.api.Assertions.*;

class CounterTest {

    @BeforeEach
    void setUp() throws Exception {
    }

    @Test
    void testIncrement() {
        fail("Not yet implemented"); // TODO
    }

    @Test
    void testDecrement() {
        fail("Not yet implemented"); // TODO
    }

}
```

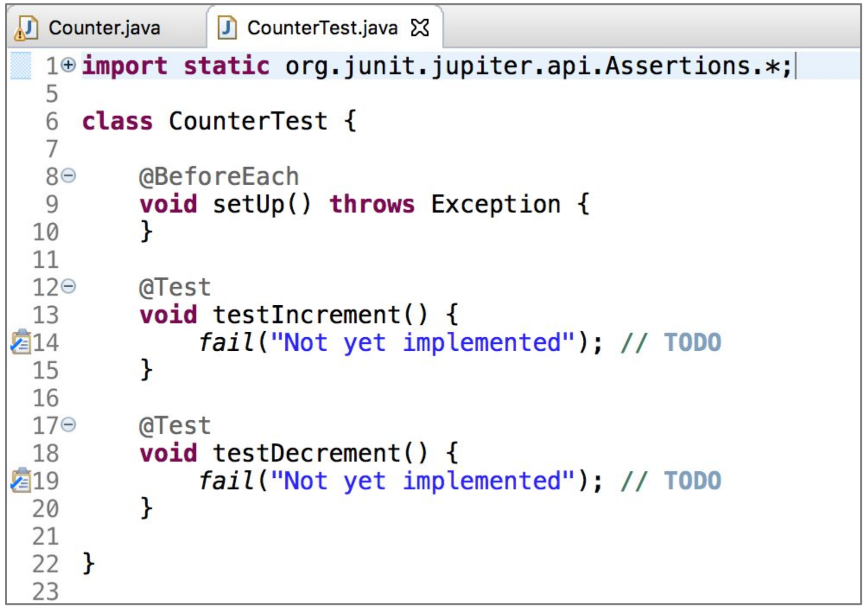
Create JUnit Tests

- 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



```
Finished after 0.17 seconds

Runs: 2/2    Errors: 0    Failures: 2

CounterTest [Runner: JUnit 5] (0.004 s)
    testIncrement() (0.002 s)
    testDecrement() (0.001 s)
```

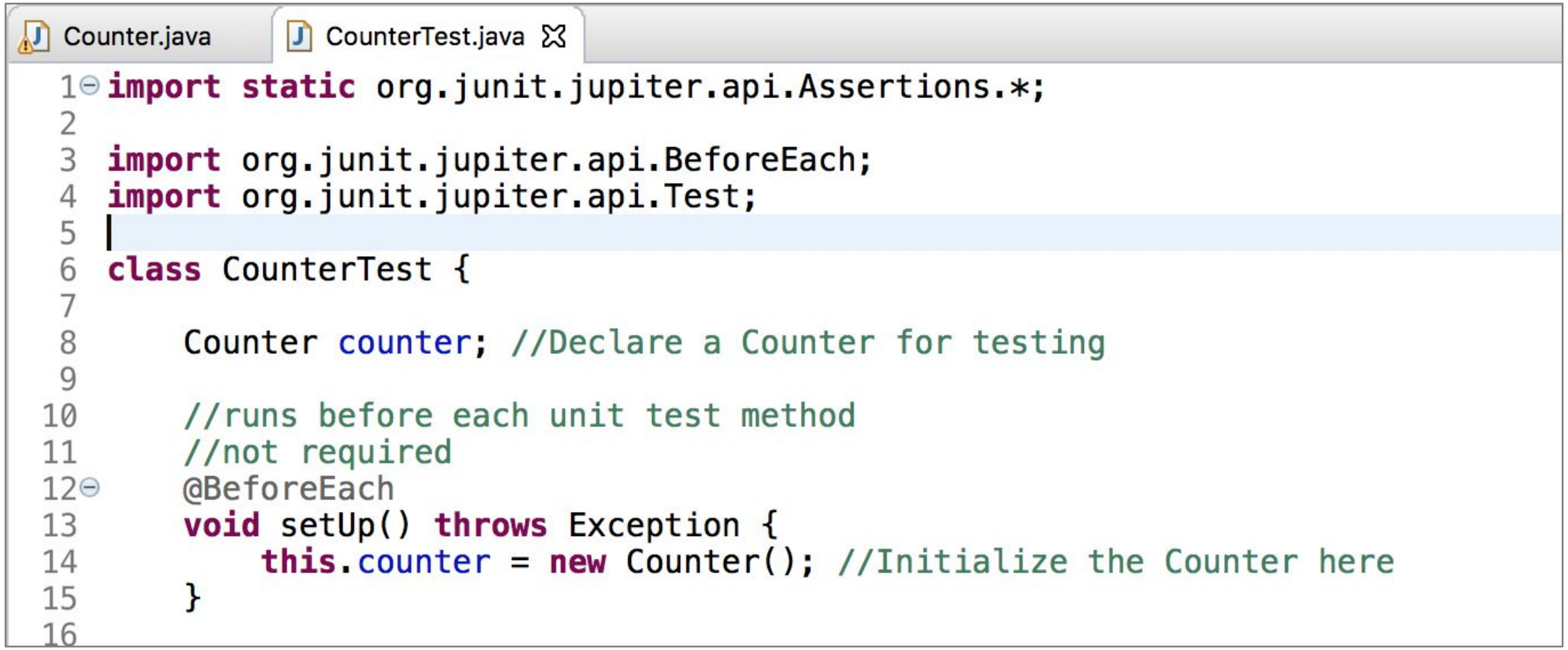
```
org.opentest4j.AssertionFailedError: Not yet implemented
at CounterTest.testIncrement(CounterTest.java:14)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507)
```



```java
import static org.junit.jupiter.api.Assertions.*;

class CounterTest {

    @BeforeEach
    void setUp() throws Exception {
    }

    @Test
    void testIncrement() {
        fail("Not yet implemented"); // TODO
    }

    @Test
    void testDecrement() {
        fail("Not yet implemented"); // TODO
    }

}
```

Create JUnit Tests

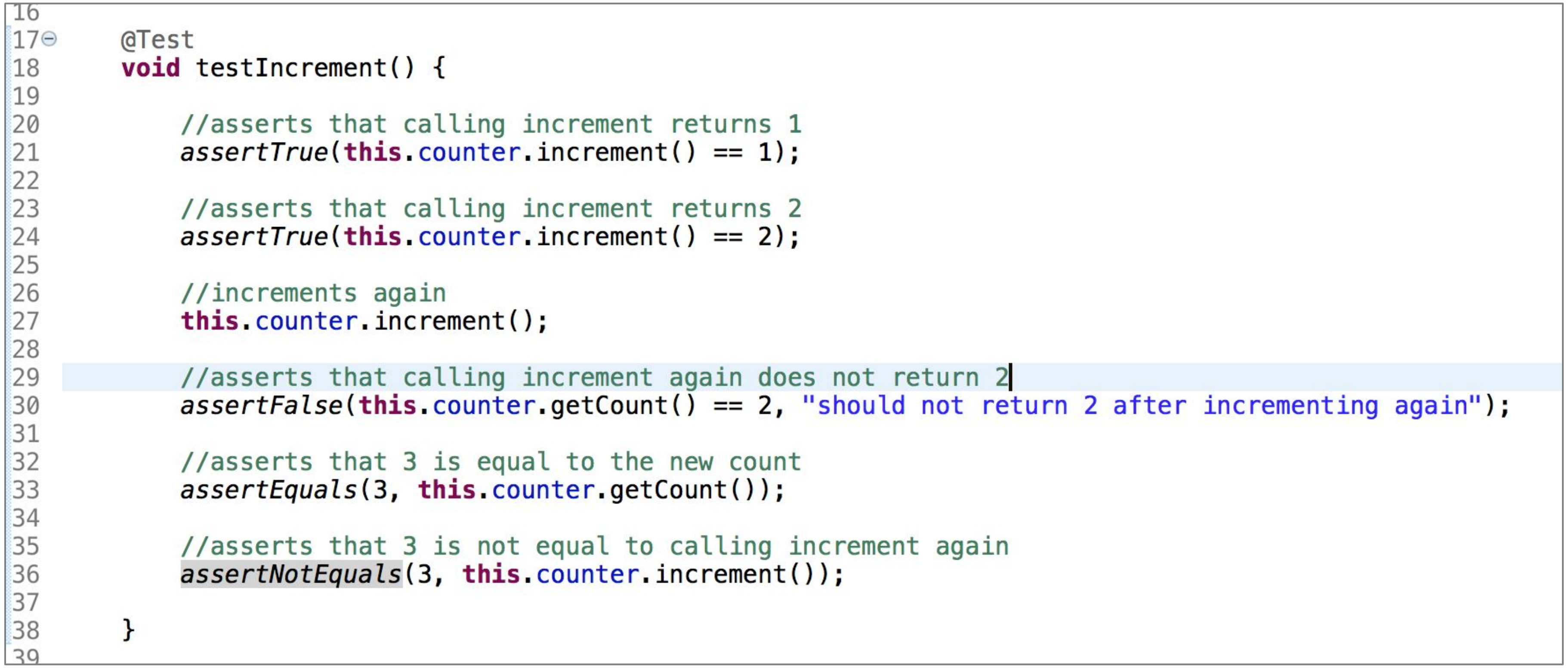
- Declare and initialize a Counter object for testing



```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CounterTest {

    Counter counter; //Declare a Counter for testing

    //runs before each unit test method
    //not required
    @BeforeEach
    void setUp() throws Exception {
        this.counter = new Counter(); //Initialize the Counter here
    }

```

Create JUnit Tests

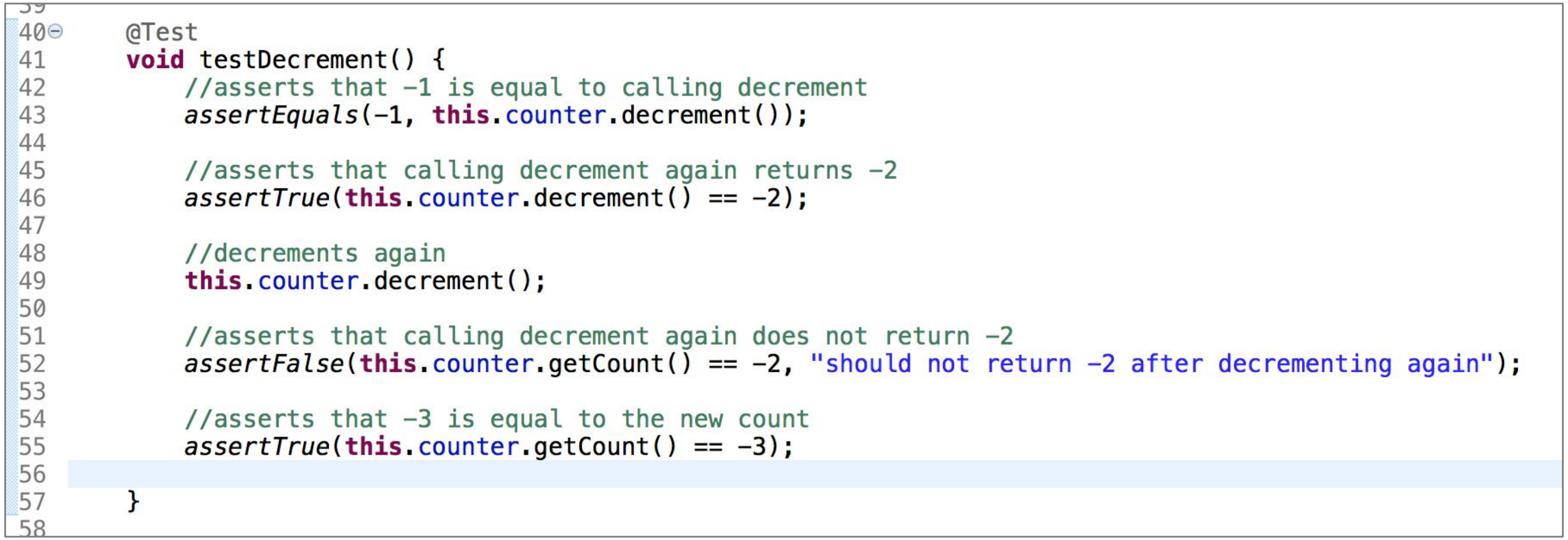
- Implement the test methods, adding test cases with assert methods



```
    @Test
    void testIncrement() {

        //asserts that calling increment returns 1
        assertTrue(this.counter.increment() == 1);

        //asserts that calling increment returns 2
        assertTrue(this.counter.increment() == 2);

        //increments again
        this.counter.increment();

        //asserts that calling increment again does not return 2
        assertFalse(this.counter.getCount() == 2, "should not return 2 after incrementing again");

        //asserts that 3 is equal to the new count
        assertEquals(3, this.counter.getCount());

        //asserts that 3 is not equal to calling increment again
        assertNotEquals(3, this.counter.increment());

    }
```

Create JUnit Tests

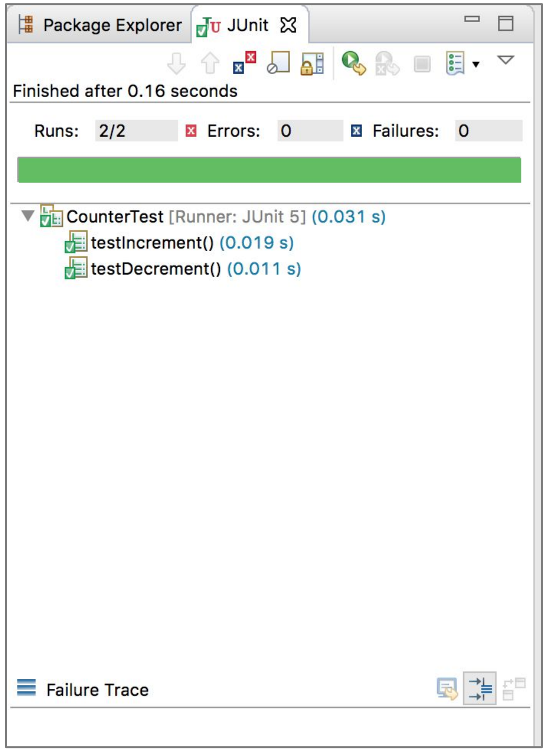
- Implement the test methods, adding test cases with assert methods



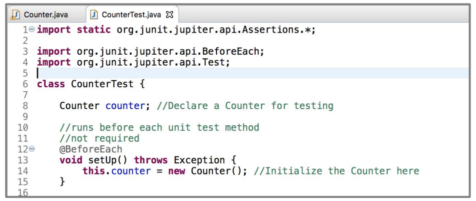
```
    @Test
    void testDecrement() {
        //asserts that -1 is equal to calling decrement
        assertEquals(-1, this.counter.decrement());

        //asserts that calling decrement again returns -2
        assertTrue(this.counter.decrement() == -2);

        //decrements again
        this.counter.decrement();

        //asserts that calling decrement again does not return -2
        assertFalse(this.counter.getCount() == -2, "should not return -2 after decrementing again");

        //asserts that -3 is equal to the new count
        assertTrue(this.counter.getCount() == -3);

    }
```

Create JUnit Tests

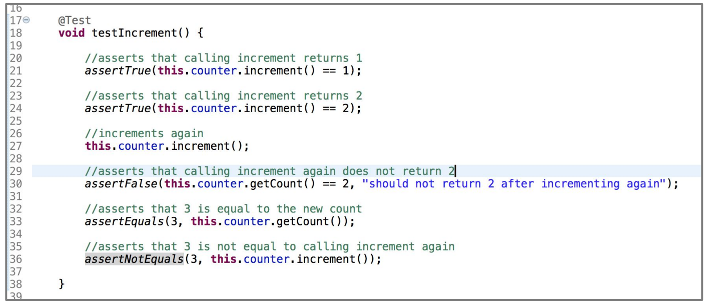
- If you run the tests, they should ALL pass
  - In the JUnit panel (on the left) – the top bar will show green
  - The number next to “Failures” will show 0



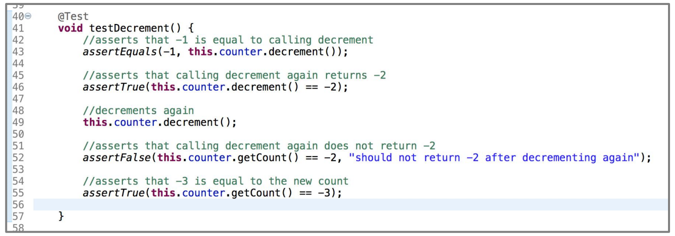
```
Finished after 0.16 seconds

Runs: 2/2    Errors: 0    Failures: 0

CounterTest [Runner: JUnit 5] (0.031 s)
    testIncrement() (0.019 s)
    testDecrement() (0.011 s)
```



```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CounterTest {

    Counter counter; //Declare a Counter for testing

    //runs before each unit test method
    //not required
    @BeforeEach
    void setUp() throws Exception {
        this.counter = new Counter(); //Initialize the Counter here
    }

```



```java

    @Test
    void testIncrement() {

        //asserts that calling increment returns 1
        assertTrue(this.counter.increment() == 1);

        //asserts that calling increment returns 2
        assertTrue(this.counter.increment() == 2);

        //increments again
        this.counter.increment();

        //asserts that calling increment again does not return 2
        assertFalse(this.counter.getCount() == 2, "should not return 2 after incrementing again");

        //asserts that 3 is equal to the new count
        assertEquals(3, this.counter.getCount());

        //asserts that 3 is not equal to calling increment again
        assertNotEquals(3, this.counter.increment());

    }

```



```java

    @Test
    void testDecrement() {
        //asserts that -1 is equal to calling decrement
        assertEquals(-1, this.counter.decrement());

        //asserts that calling decrement again returns -2
        assertTrue(this.counter.decrement() == -2);

        //decrements again
        this.counter.decrement();

        //asserts that calling decrement again does not return -2
        assertFalse(this.counter.getCount() == -2, "should not return -2 after decrementing again");

        //asserts that -3 is equal to the new count
        assertTrue(this.counter.getCount() == -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);
```

- 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

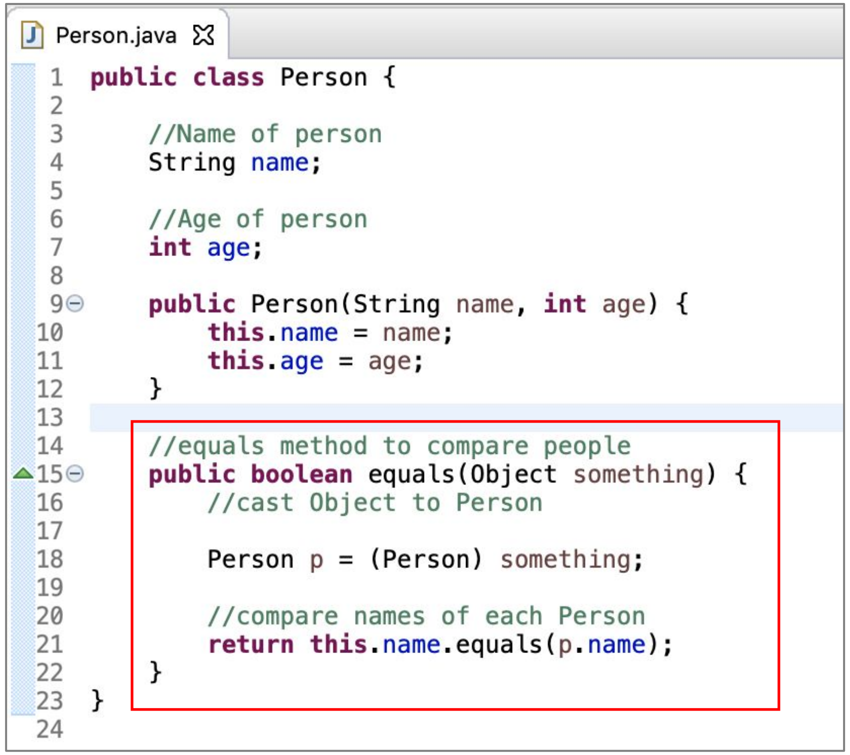
- Why is all of this important?
  - The JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *Objects*
- To define *equals* for your own objects, you'll have to define *exactly* this method in your class:

```
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

- Here’s a full (sample) implementation of *equals* inside a Person class


Person.java

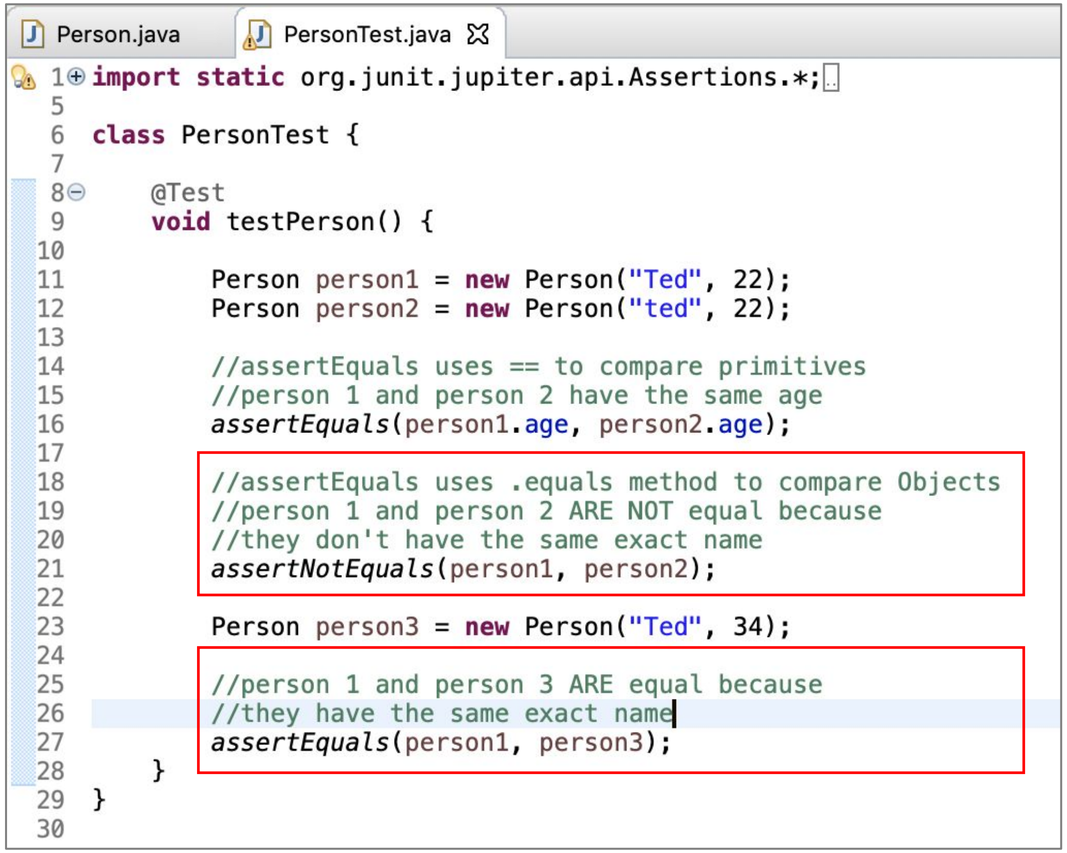
```java
public class Person {

    //Name of person
    String name;

    //Age of person
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    //equals method to compare people
    public boolean equals(Object something) {
        //cast Object to Person

        Person p = (Person) something;

        //compare names of each Person
        return this.name.equals(p.name);
    }
}

```

- Two people are “equal” if they have the same exact name

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

Testing for Equality in Java

- Here's how we compare people in our unit testing class



```java
import static org.junit.jupiter.api.Assertions.*;

class PersonTest {

    @Test
    void testPerson() {

        Person person1 = new Person("Ted", 22);
        Person person2 = new Person("ted", 22);

        //assertEquals uses == to compare primitives
        //person 1 and person 2 have the same age
        assertEquals(person1.age, person2.age);

        //assertEquals uses .equals method to compare Objects
        //person 1 and person 2 ARE NOT equal because
        //they don't have the same exact name
        assertNotEquals(person1, person2);

        Person person3 = new Person("Ted", 34);

        //person 1 and person 3 ARE equal because
        //they have the same exact name
        assertEquals(person1, person3);
    }
}
```

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 in *"About_Equality"* lecture)

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

- Asserts that two int arrays are equal

Assert Methods with Floating Points Types

- Note: When you want to compare floating point types (e.g. double or float) with a high amount of precision
  - 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 the *expected* and *actual* are equal, within the given *delta*
  - *delta* is typically a very small double (e.g. 0.000001) used for comparison

- For example:

```
void assertEquals(aDoubleValue, anotherDoubleValue, 0.000001)
```

  - This evaluates to: Math.abs(aDoubleValue – anotherDoubleValue) <= 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

```
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)

```
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

```
void assertDoesNotThrow(() -> {
    //code that does not throw an exception
});
```

- Asserts that the enclosed code does not throw an Exception

- 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()*

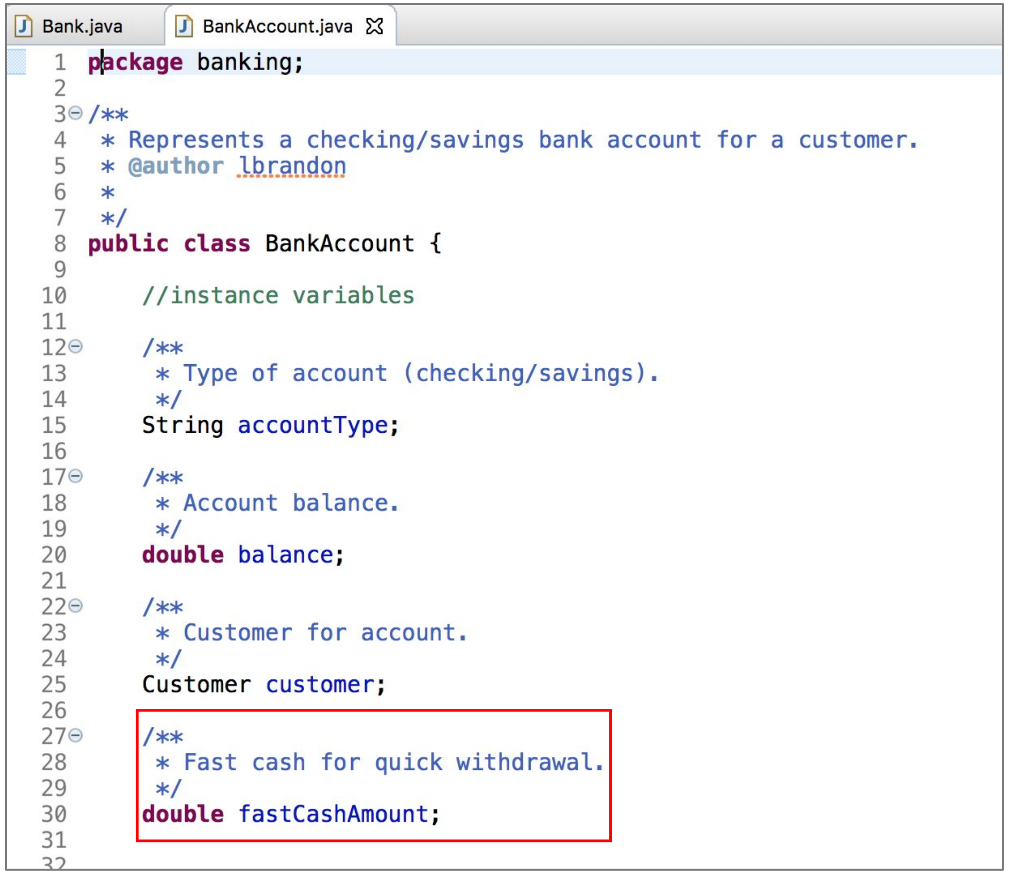
Banking Project w/ Unit Testing

Banking Project w/ Unit Testing

- We’ll unit test our previous “Banking” project, which had 3 classes
  - Bank
    - Includes the public static void main(String[] args) method
    - No updates needed
  - Customer
    - No updates needed
  - BankAccount
    - Updates needed!
- Create new unit testing classes
  - CustomerTest
    - For testing the Customer class
  - BankAccountTest
    - For testing the BankAccount class

Updated BankAccount Class

- Add a fastCashAmount instance variable

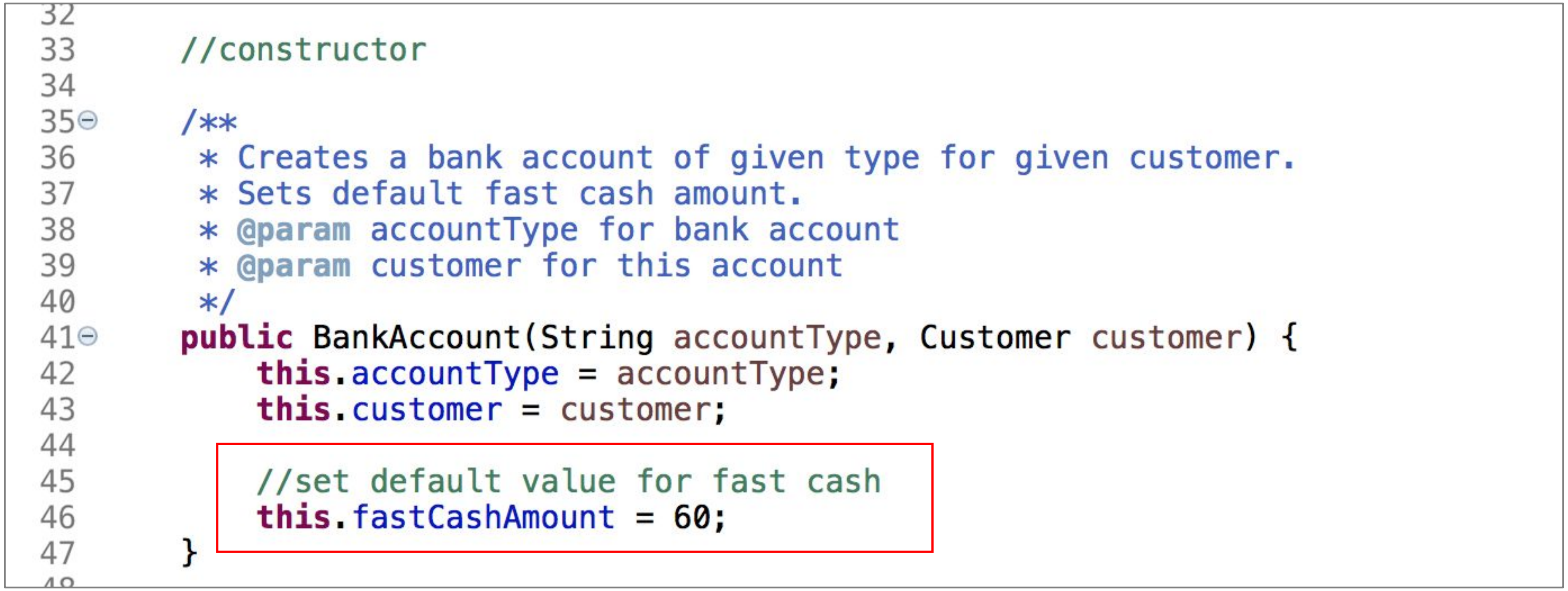


```java
package banking;

/**
 * Represents a checking/savings bank account for a customer.
 * @author lbrandon
 *
 */
public class BankAccount {

    //instance variables

    /**
     * Type of account (checking/savings).
     */
    String accountType;

    /**
     * Account balance.
     */
    double balance;

    /**
     * Customer for account.
     */
    Customer customer;

    /**
     * Fast cash for quick withdrawal.
     */
    double fastCashAmount;

```

Penn Engineering

Updated BankAccount Class

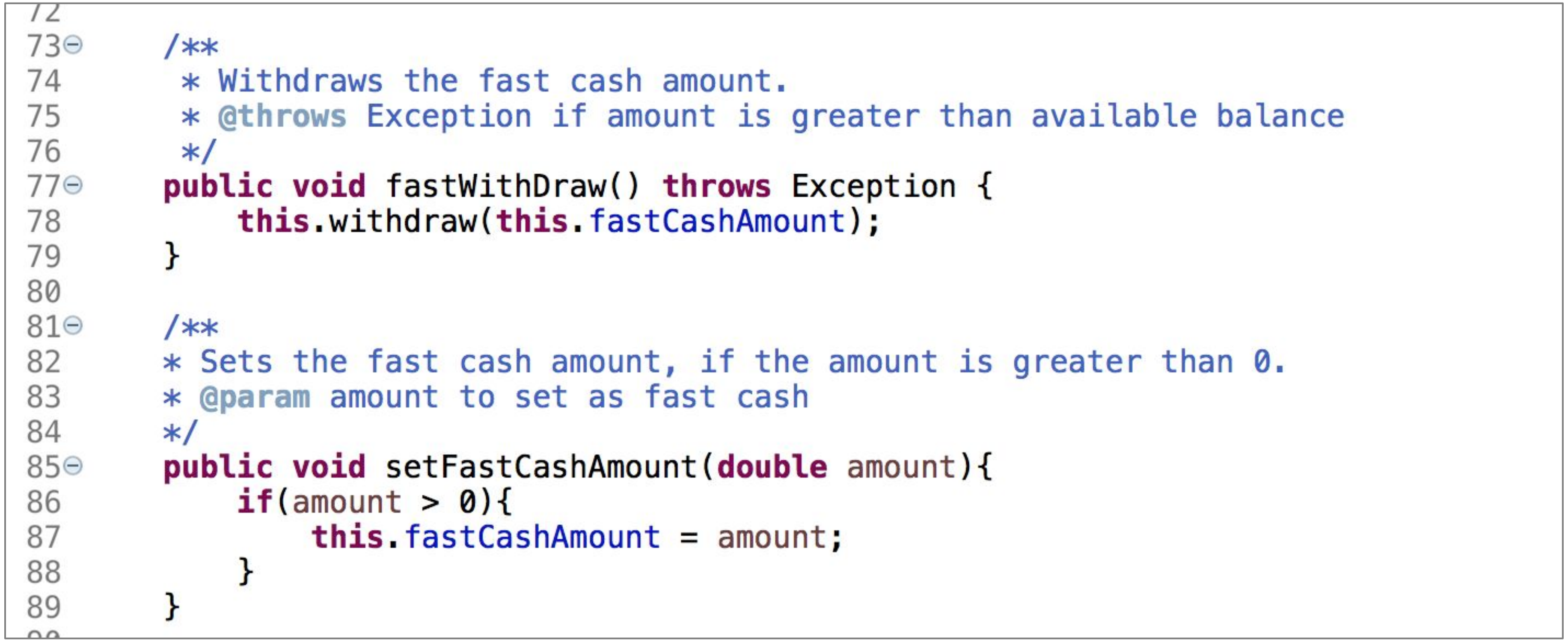
- Update the constructor to set the initial value for fastCashAmount



```java
    //constructor

    /**
     * Creates a bank account of given type for given customer.
     * Sets default fast cash amount.
     * @param accountType for bank account
     * @param customer for this account
     */
    public BankAccount(String accountType, Customer customer) {
        this.accountType = accountType;
        this.customer = customer;

        //set default value for fast cash
        this.fastCashAmount = 60;
    }
```

Updated BankAccount Class

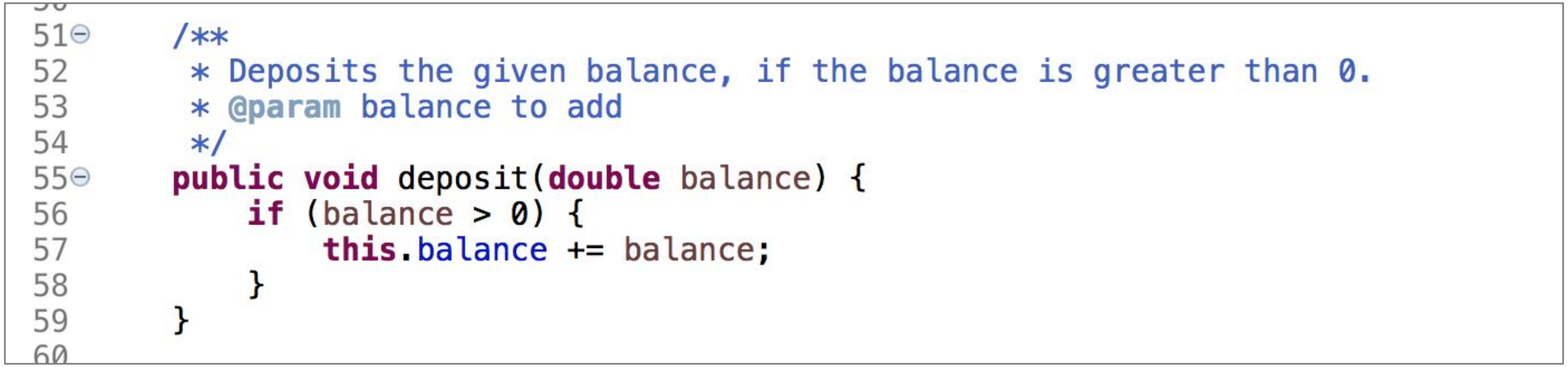
- Add the fastWithDraw and setFastCashAmount methods



```
    /**
     * Withdraws the fast cash amount.
     * @throws Exception if amount is greater than available balance
     */
    public void fastWithDraw() throws Exception {
        this.withdraw(this.fastCashAmount);
    }

    /**
    * Sets the fast cash amount, if the amount is greater than 0.
    * @param amount to set as fast cash
    */
    public void setFastCashAmount(double amount){
        if(amount > 0){
            this.fastCashAmount = amount;
        }
    }
```

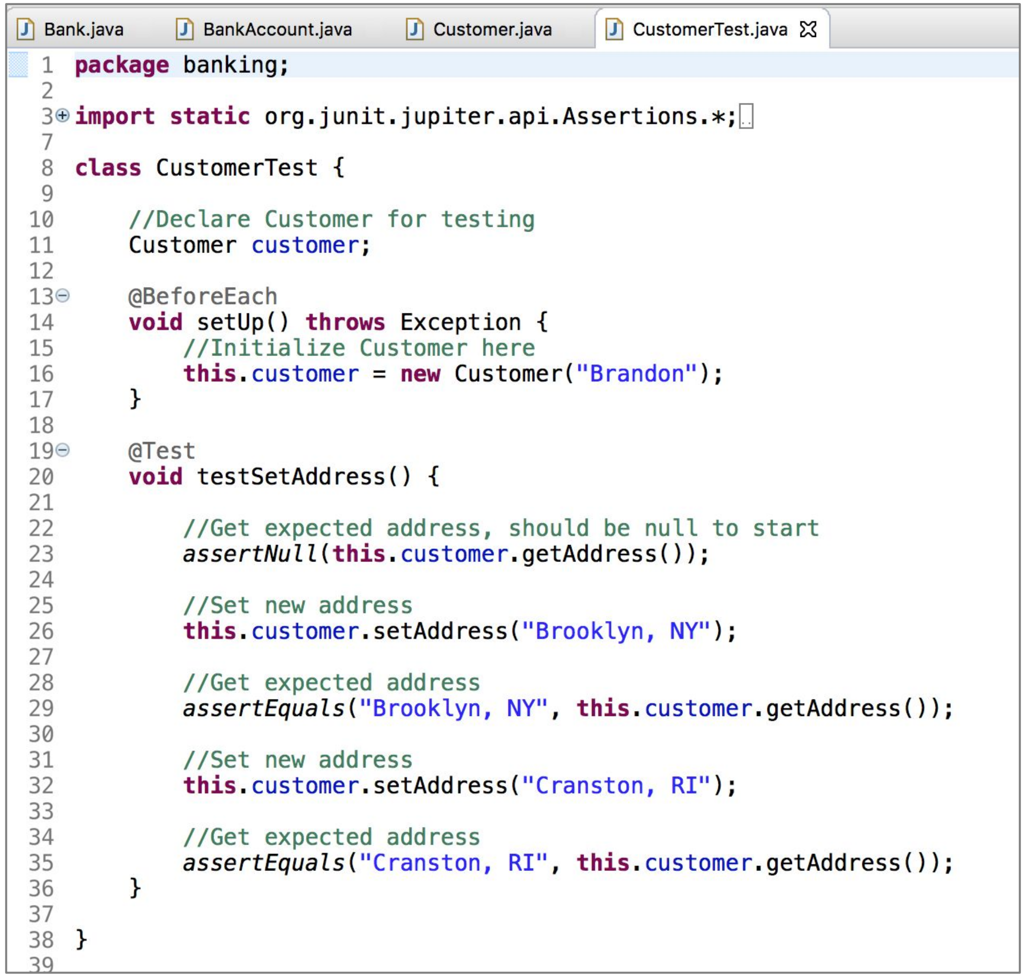
Updated BankAccount Class

- Update the deposit method



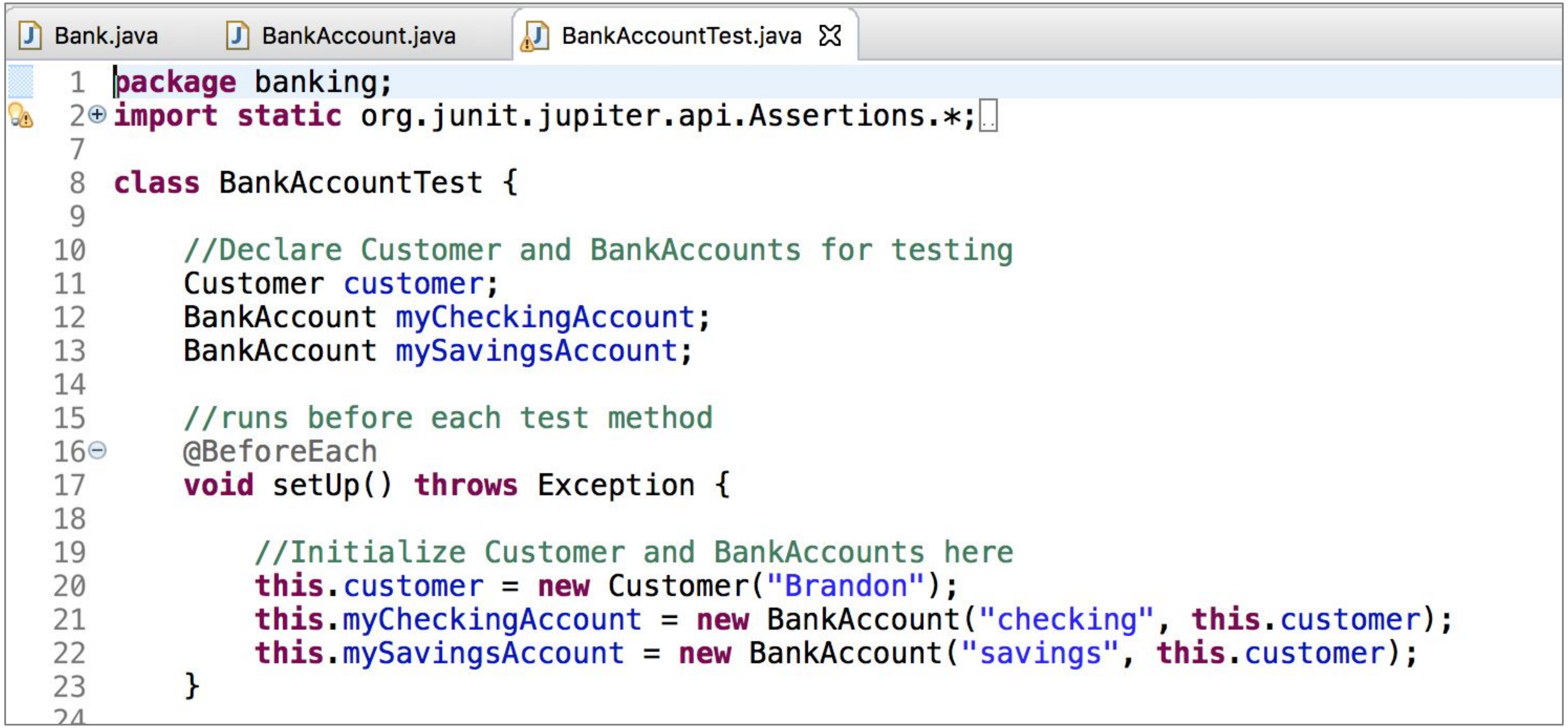
```java
    /**
     * Deposits the given balance, if the balance is greater than 0.
     * @param balance to add
     */
    public void deposit(double balance) {
        if (balance > 0) {
            this.balance += balance;
        }
    }
```

Create CustomerTest Class



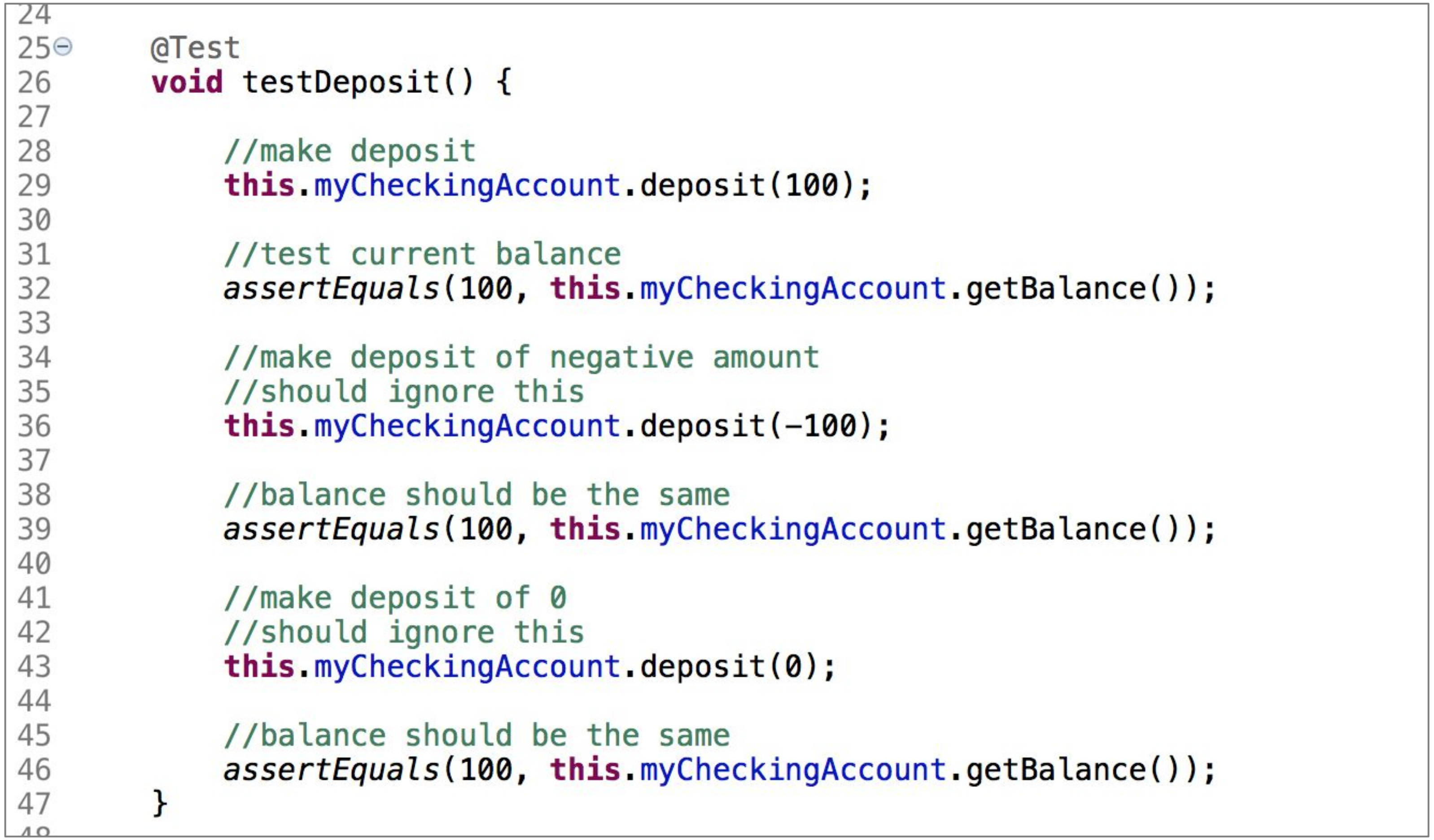
```java
package banking;

import static org.junit.jupiter.api.Assertions.*;

class CustomerTest {

    //Declare Customer for testing
    Customer customer;

    @BeforeEach
    void setUp() throws Exception {
        //Initialize Customer here
        this.customer = new Customer("Brandon");
    }

    @Test
    void testSetAddress() {

        //Get expected address, should be null to start
        assertNull(this.customer.getAddress());

        //Set new address
        this.customer.setAddress("Brooklyn, NY");

        //Get expected address
        assertEquals("Brooklyn, NY", this.customer.getAddress());

        //Set new address
        this.customer.setAddress("Cranston, RI");

        //Get expected address
        assertEquals("Cranston, RI", this.customer.getAddress());
    }

}
```

Create BankAccountTest Class



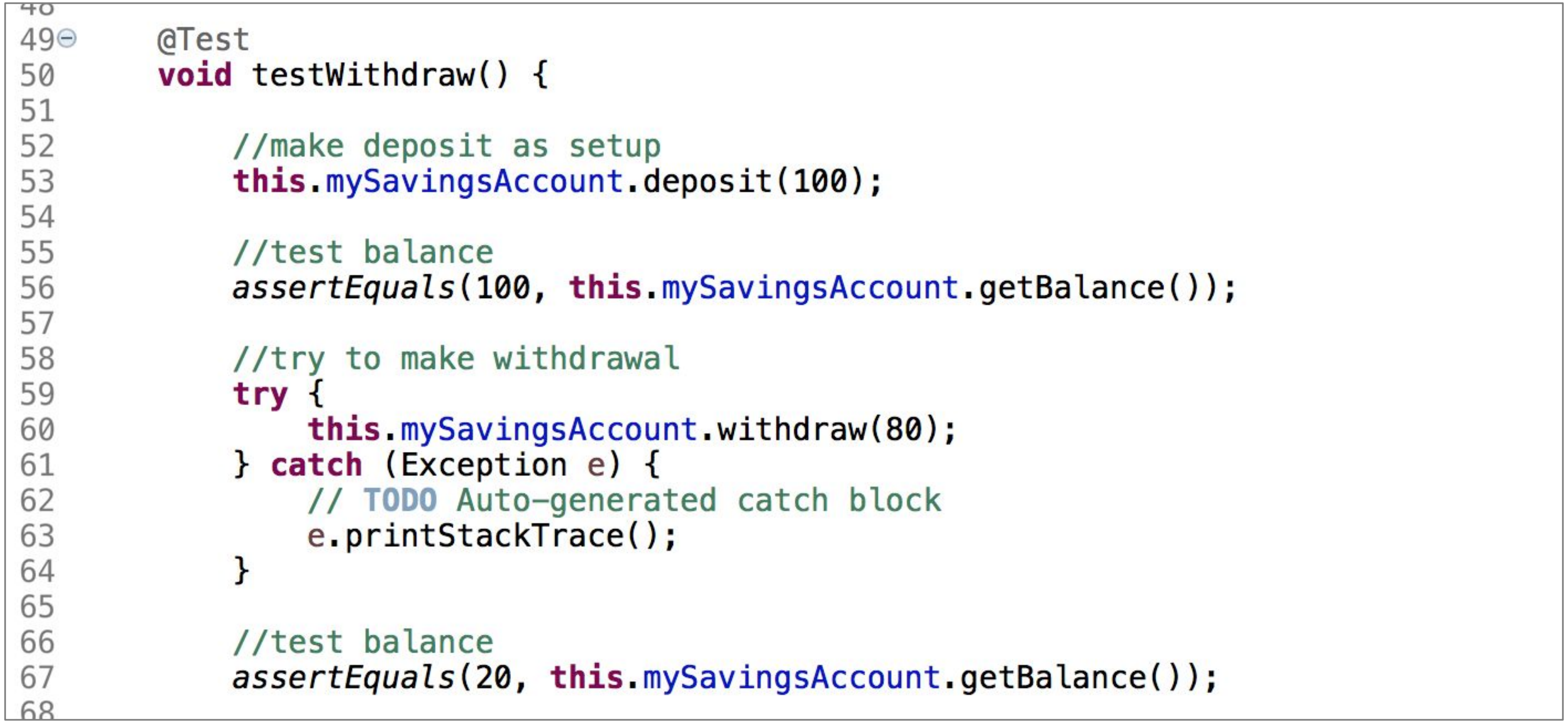
```
package banking;
import static org.junit.jupiter.api.Assertions.*;

class BankAccountTest {

    //Declare Customer and BankAccounts for testing
    Customer customer;
    BankAccount myCheckingAccount;
    BankAccount mySavingsAccount;

    //runs before each test method
    @BeforeEach
    void setUp() throws Exception {

        //Initialize Customer and BankAccounts here
        this.customer = new Customer("Brandon");
        this.myCheckingAccount = new BankAccount("checking", this.customer);
        this.mySavingsAccount = new BankAccount("savings", this.customer);
    }
```

Create BankAccountTest Class



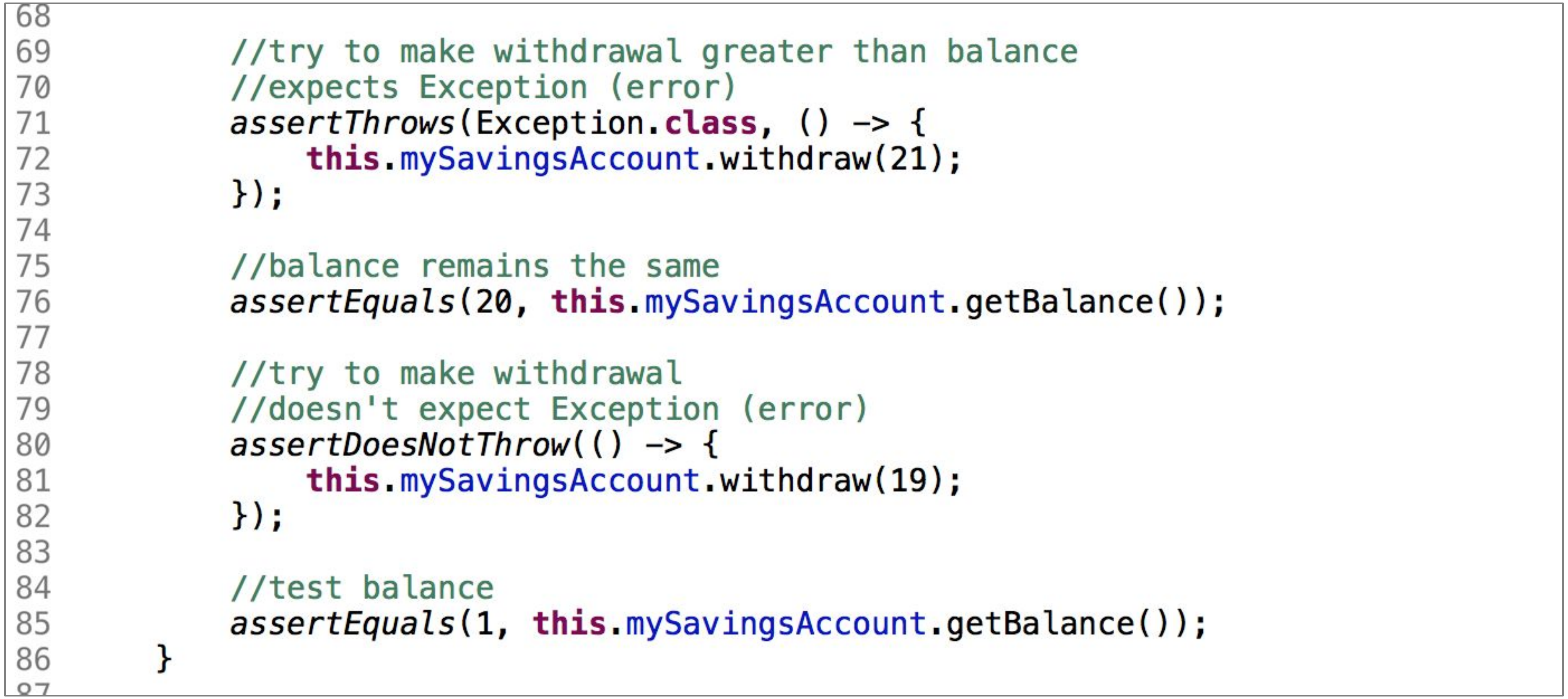
```
    @Test
    void testDeposit() {

        //make deposit
        this.myCheckingAccount.deposit(100);

        //test current balance
        assertEquals(100, this.myCheckingAccount.getBalance());

        //make deposit of negative amount
        //should ignore this
        this.myCheckingAccount.deposit(-100);

        //balance should be the same
        assertEquals(100, this.myCheckingAccount.getBalance());

        //make deposit of 0
        //should ignore this
        this.myCheckingAccount.deposit(0);

        //balance should be the same
        assertEquals(100, this.myCheckingAccount.getBalance());
    }
```

Create BankAccountTest Class



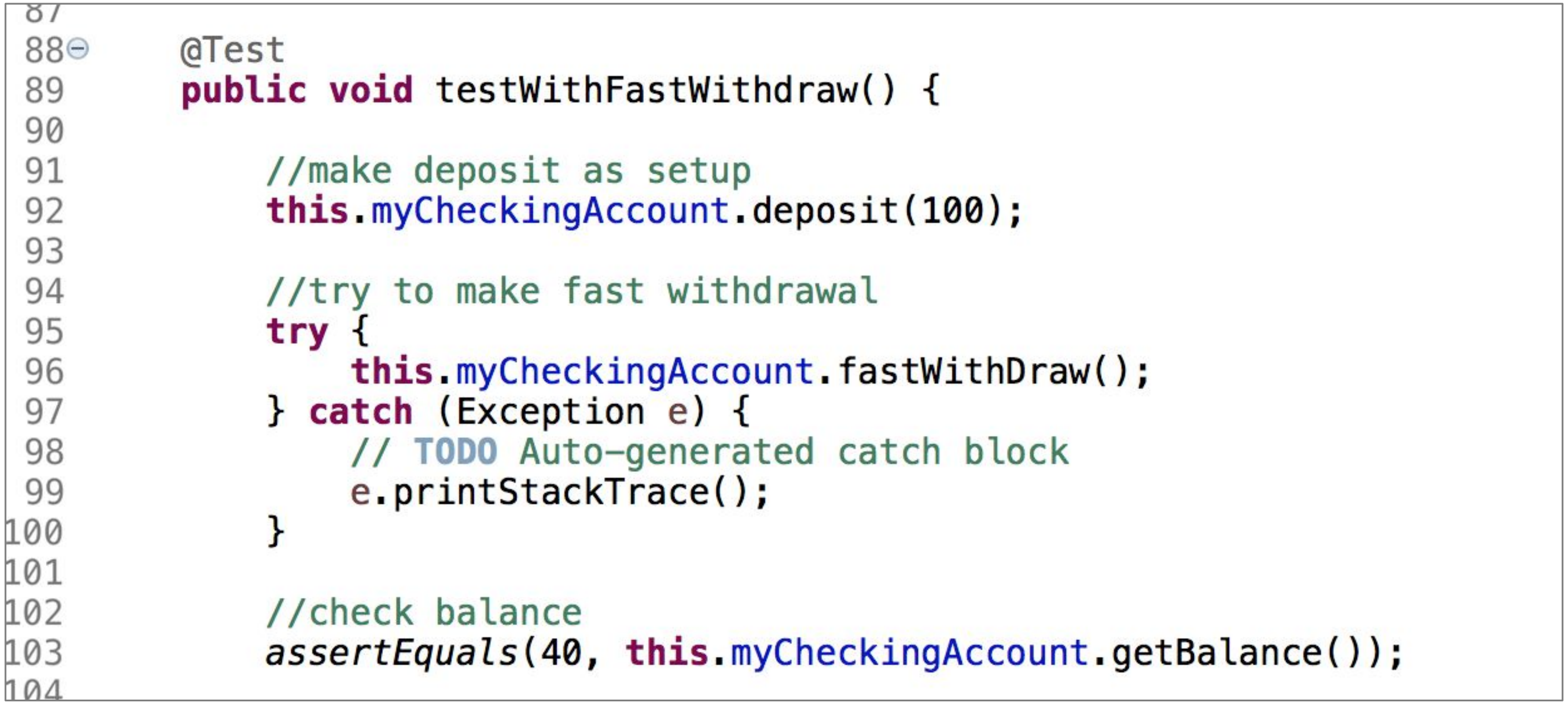
```
    @Test
    void testWithdraw() {

        //make deposit as setup
        this.mySavingsAccount.deposit(100);

        //test balance
        assertEquals(100, this.mySavingsAccount.getBalance());

        //try to make withdrawal
        try {
            this.mySavingsAccount.withdraw(80);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        //test balance
        assertEquals(20, this.mySavingsAccount.getBalance());
```

Create BankAccountTest Class



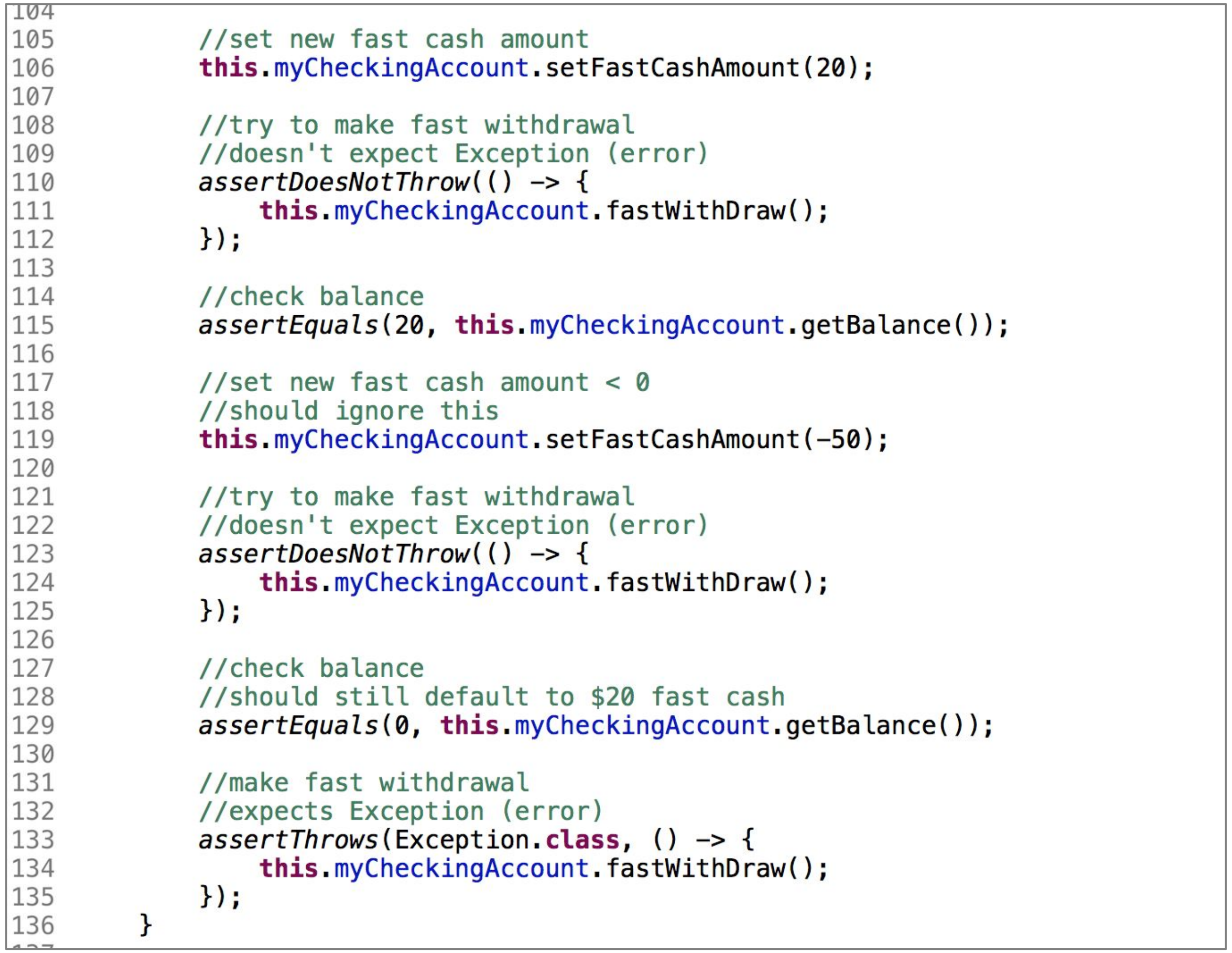
```java

        //try to make withdrawal greater than balance
        //expects Exception (error)
        assertThrows(Exception.class, () -> {
            this.mySavingsAccount.withdraw(21);
        });

        //balance remains the same
        assertEquals(20, this.mySavingsAccount.getBalance());

        //try to make withdrawal
        //doesn't expect Exception (error)
        assertDoesNotThrow(() -> {
            this.mySavingsAccount.withdraw(19);
        });

        //test balance
        assertEquals(1, this.mySavingsAccount.getBalance());
    }
```

Create BankAccountTest Class



```
    @Test
    public void testWithFastWithdraw() {

        //make deposit as setup
        this.myCheckingAccount.deposit(100);

        //try to make fast withdrawal
        try {
            this.myCheckingAccount.fastWithDraw();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        //check balance
        assertEquals(40, this.myCheckingAccount.getBalance());
```

Create BankAccountTest Class



```java
        //set new fast cash amount
        this.myCheckingAccount.setFastCashAmount(20);

        //try to make fast withdrawal
        //doesn't expect Exception (error)
        assertDoesNotThrow(() -> {
            this.myCheckingAccount.fastWithDraw();
        });

        //check balance
        assertEquals(20, this.myCheckingAccount.getBalance());

        //set new fast cash amount < 0
        //should ignore this
        this.myCheckingAccount.setFastCashAmount(-50);

        //try to make fast withdrawal
        //doesn't expect Exception (error)
        assertDoesNotThrow(() -> {
            this.myCheckingAccount.fastWithDraw();
        });

        //check balance
        //should still default to $20 fast cash
        assertEquals(0, this.myCheckingAccount.getBalance());

        //make fast withdrawal
        //expects Exception (error)
        assertThrows(Exception.class, () -> {
            this.myCheckingAccount.fastWithDraw();
        });
    }
```