CS 320
JUnit

1. JUnit Basics

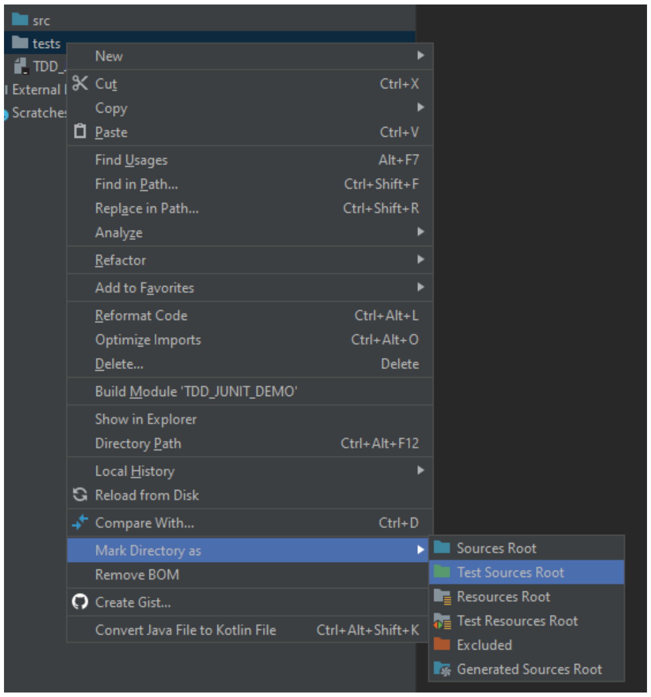
JUnit is a library used for creating Unit tests for Java programs. It has several call features to making automated tests that can be run when validating a program. To include JUnit and it's functions, the class with the calling functions needs to import the methods:

```
// for @Test calls //
import org.junit.Test;

// for assertEqual() calls //
import static org.junit.Assert.*;

// for making a combined test suite //
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

2. TDD_JUNIT_DEMO Getting Started

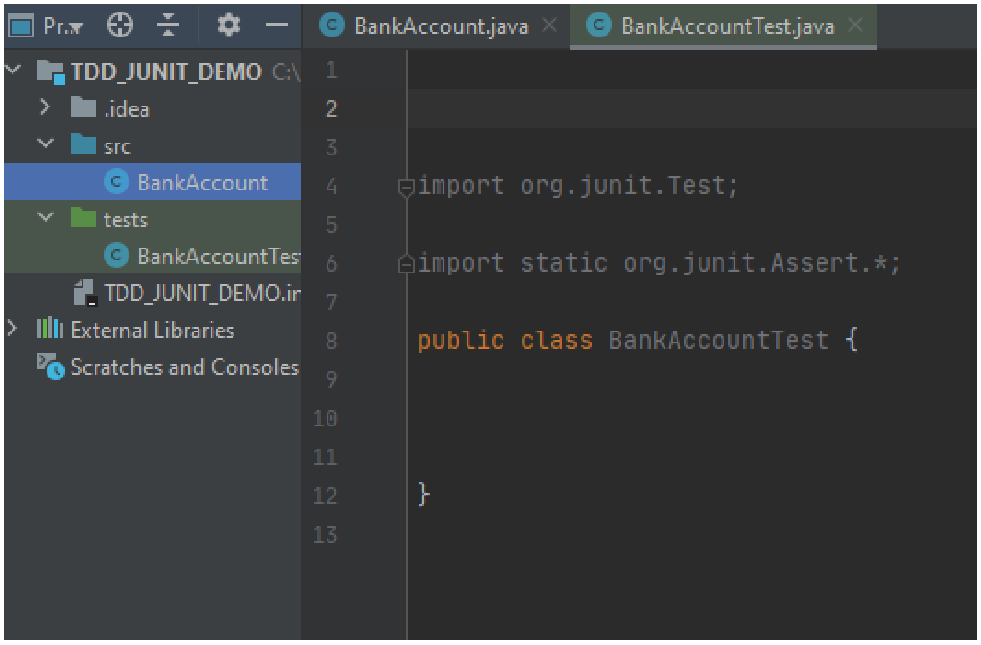
1. The first step is to make a new project with IntelliJ. Make sure a java JDK is available, it will offer to get the newest Oracle version if you don't have one yet. Name the project "TDD_JUNIT_DEMO".

2. Add a new directory called "tests" to the project. Right click on it and say "Mark Directory As" -> "Test Source Root." This should mark the folder as green.



3. Create a new class in the “src” directory called “BankAccount” and another new class in the “tests” directory called “BankAccountTest”

4. In BankAccountTest, import the JUnit Assert and Tests functions. This should cause an error to pop up. Solve this by telling the IDE to add the JUnit library to the classpath.



```java


import org.junit.Test;

import static org.junit.Assert.*;

public class BankAccountTest {



}

```

5. We should be set to start now.

3. TDD_JUNIT_DEMO First Tests

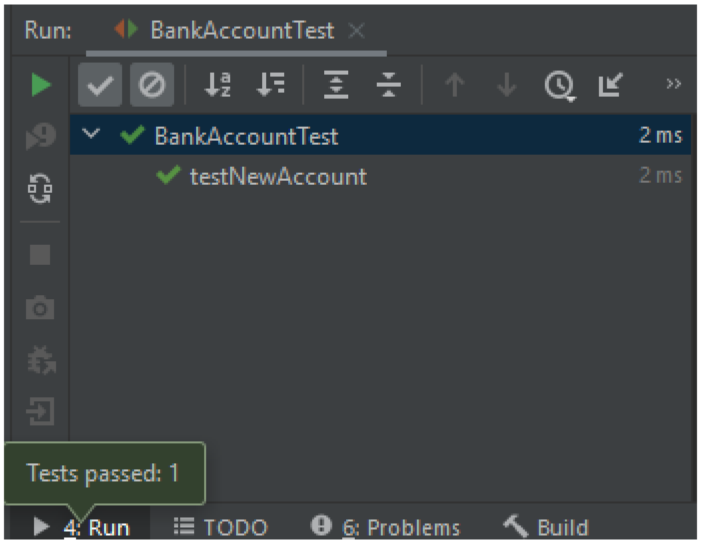
1. We can add a new test using the “@Test” syntax. For the first

```
     @Test
     public void testNewAccount() {
          BankAccount ba = new BankAccount();
          assertEquals(0, ba.settlement());
     }
```

2. When you run this, it will fail as the settlement method doesn’t exist yet. So, next step is to write this code in the BankAccount class.

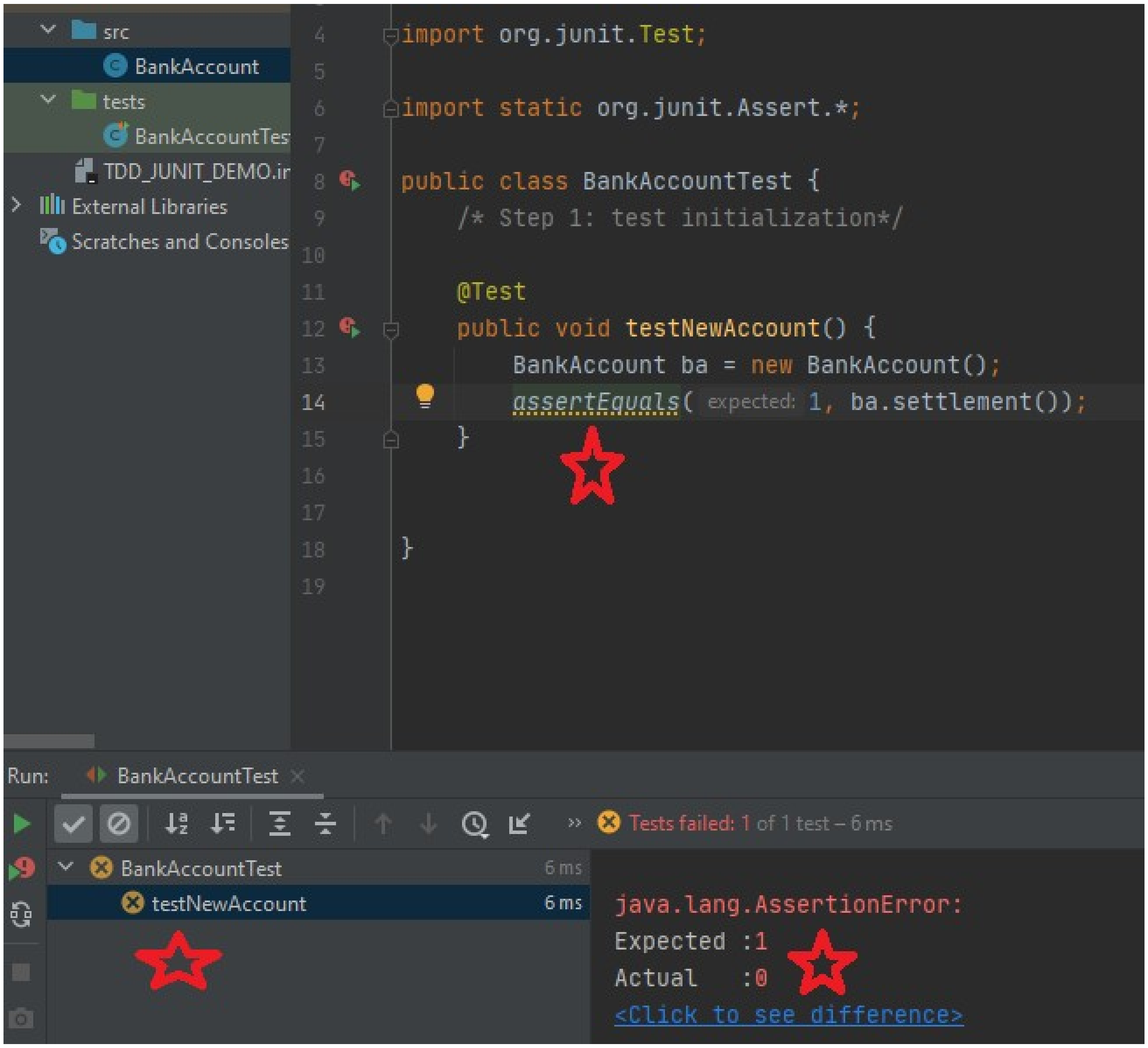
```
    int currentBalance = 0;

    public int settlement() {
        return currentBalance;
    }
```

Running this causes the test to pass. Notice that when running these tests, the results are reported with check marks in the bottom left.



```
✓ BankAccountTest        2 ms
    ✓ testNewAccount     2 ms
Tests passed: 1
```

Change the assert value to a 1 instead of a 0 and notice how it's reported as failing, the expected and actual values are reported, and the offending test is marked.



```java
import org.junit.Test;

import static org.junit.Assert.*;

public class BankAccountTest {
    /* Step 1: test initialization*/

    @Test
    public void testNewAccount() {
        BankAccount ba = new BankAccount();
        assertEquals( expected: 1, ba.settlement());
    }


}
```

```
java.lang.AssertionError:
Expected :1
Actual   :0
<Click to see difference>
```

3. With this functionality working, we'll add a new test for depositing to the account:

```java
@Test
public void testDeposit() {
      BankAccount ba = new BankAccount();
      ba.deposit(10);
      assertEquals(10, ba.settlement());
}
```

4. The test fails when run so we add the next implementation:

```
	public void deposit(int i) {
		currentBalance += i;
	}
```

5. The next test will be for withdrawing:

```
	@Test
	public void testWithdraw() {
		BankAccount ba = new BankAccount();
		ba.deposit(10);
		ba.withdraw(5);
		assertEquals(5, ba.settlement());
	}
```

6. Next, we implement the code to make this new test pass:

```
	public void withdraw(int i){
		currentBalance -= i;
	}
```

7. The final steps are more complex. Lets prevent an over draw of the account from occurring. Start by making a new file in “src” called “InsufficientBalanceException” and we’ll have this class extend Throwable:

```
public class InsufficientBalanceException extends Throwable {}
```

We'll also make a new test file called "BankAccountTest2" in the tests directory. Here, since we want our new exception is expected to be thrown, we'll add the "expected" condition to the new test:

```
import org.junit.Test;

public class BankAccountTest2 {
    @Test(expected = InsufficientBalanceException.class)
    public void testInvalid() throws InsufficientBalanceException{
        BankAccount ba = new BankAccount();
        ba.deposit(10);
        ba.withdraw(20);
    }
}
```

Running this, since the expected exception wasn't thrown, it fails. So lets modify our withdraw method

8. Change the withdraw method in the BankAccount class to throw an exception, and add the thrown exception possibility to the previous withdraw test in the first BankAccountTest class as an exception can now be thrown.

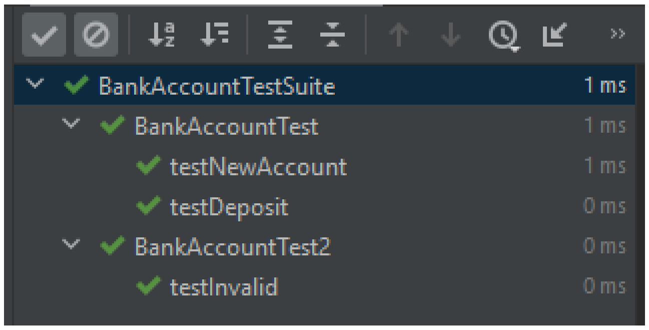
BankAccount:
```
     public void withdraw(int i) throws InsufficientBalanceException{
          if (currentBalance < i){
               throw new InsufficientBalanceException();
          }
          else {
               currentBalance -= i;
          }
     }
```
BankAccountTest:
```
     @Test
     public void testWithdraw() throws InsufficientBalanceException{
          BankAccount ba = new BankAccount();
          ba.deposit(10);
          ba.withdraw(5);
          assertEquals(5, ba.settlement());
     }
```

Running the new test should now pass since the exception is properly thrown.

9. The final step is combining our separate tests into a combined test-suite. First, create a new class in the “tests” directory called “BankAccountTestSuite” and we’ll use @RunWith to make our test suite.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
      BankAccountTest.class,
      BankAccountTest2.class
})

public class BankAccountTestSuite {}
```

When we run this new class, it will execute both individual test classes.



```
✓ BankAccountTestSuite      1 ms
  ✓ BankAccountTest         1 ms
    ✓ testNewAccount        1 ms
    ✓ testDeposit           0 ms
  ✓ BankAccountTest2        0 ms
    ✓ testInvalid           0 ms
```