

Test Driven Development

Traditional Software Development ²

- Waterfall model
- Dependencies in stages
- Cyclic loop for change management.
- Test the application at last stage
- In case of surprises revert back the process
- Tedious to accommodate change
- Islands (isolated) of information

DevOps practices for Development ³

- Test Driven Development
- Iterative and incremental development
- Continuous Testing
- Code Refactoring
- Code analysis
- Code coverage
- Continuous Integration
- Continuous delivery

Development Principles

- Code ownership
- Use code repository
- Testing before committing
- Track the changes
- Track the builds
- Track the bugs and issues
- Collaboration

Test Driven Development

5

- NOT a build and test approach..
- Don't start to write the implementation code initially.
- Follow the test first approach...
- The test first approach
 - The tests are governed by the requirements.
 - The tests govern the implementation code.

Testing phases

- Unit testing : white box testing: during development
- Integration testing : black box testing: during intermediate build
- Acceptance testing : end to end integration testing: post deployment.

Unit Testing

- The steps involved in unit-testing
 - Create the test case method
 - In test case :
 - create the object under test
 - Invoke the method on the object
 - Verify the results
 - If the results are not matching ,modify the code and again run the method.
 - Proceed with more test methods

Apply TDD

- Understand the requirements and design
- Break down the requirement into smaller units/classes/functions to the lowest fine level
- Start evolving the design by first writing the unit test for the lowest component of application.
- Let the test fail, write the code required to make the test succeed.

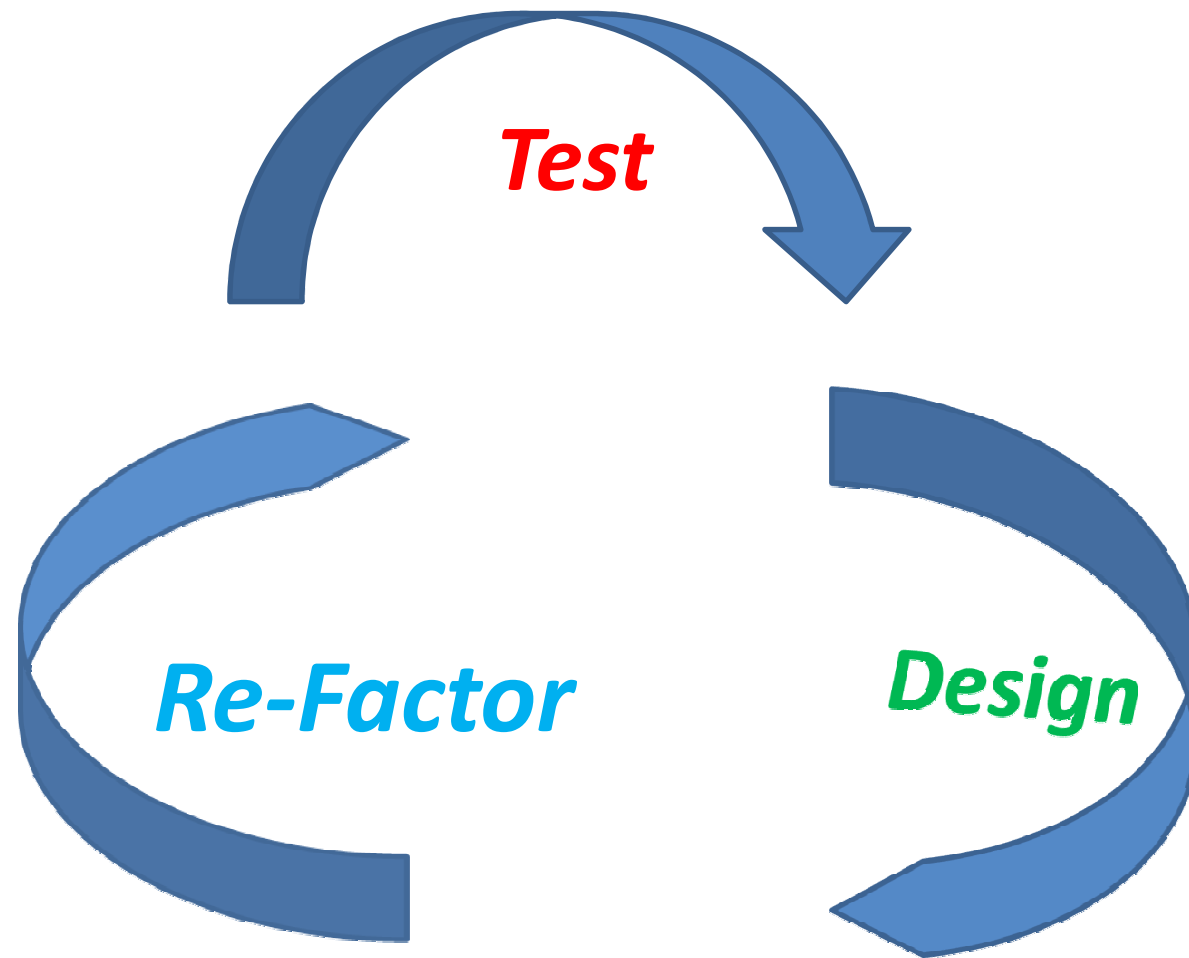
Start with TDD

9

- Write the first basic unit test, that tests the very basic unit of application.
- Run the test and let it fail for the first time, since there is NO implementation code.
- Write the only needed code to make the unit test succeed no more..no less.
- Run the test again and verify that the test succeeds .
- Write more tests one by one and each time ensure the success of all the test cases written earlier.

TDD Cycle

10



RED=>GREEN=>Refactor

User story part1

As a user I should be able to

- To have minimum balance in Saving account
- Deposit amount in SavingAccount
- Verify the amount with balance
- Withdraw amount from the SavingAccount
- Verify the amount with balance

UNIT Details

12

SavingAccount

get the balance

deposit amount

verify the balance

deposit multiple times and verify the balance

withdraw amount

verify the balance

Withdraw and deposit and verify balance

Start with TDD

- Create new Java project 'BankApp' with Eclipse Project Wizard support.
- Add JUnit testing support to the project.
- Create first test case ..
- To verify existence of SavingAccount object.

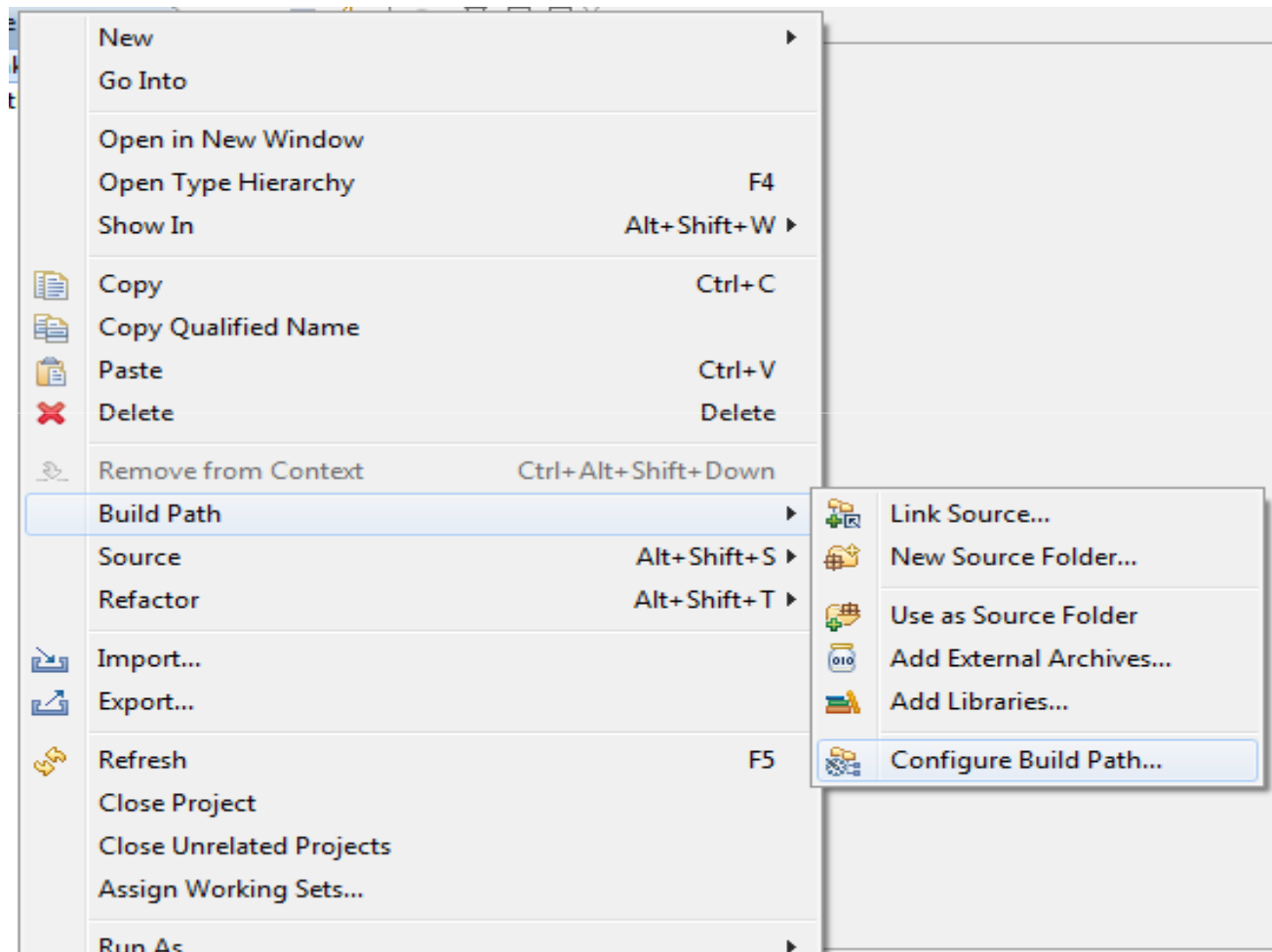
Project Reference –JUnit library

Add to the project the reference of the jUnit ver4.0 library with the project class-path settings.

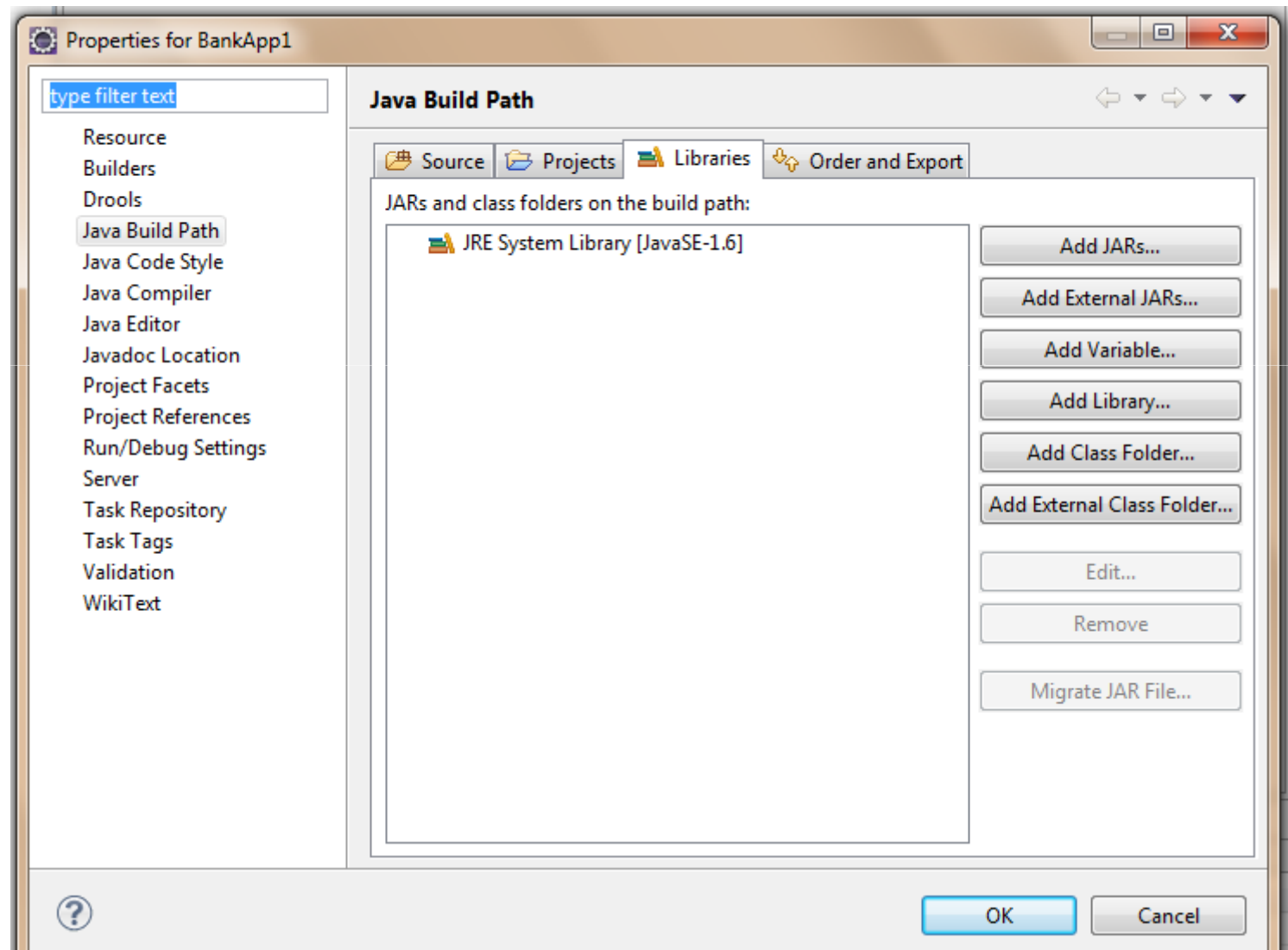
Project Explorer view ,Right click on the project and select

BuildPath ->Configure Build Path and select the JUnit library to add on path

Configure Build Path



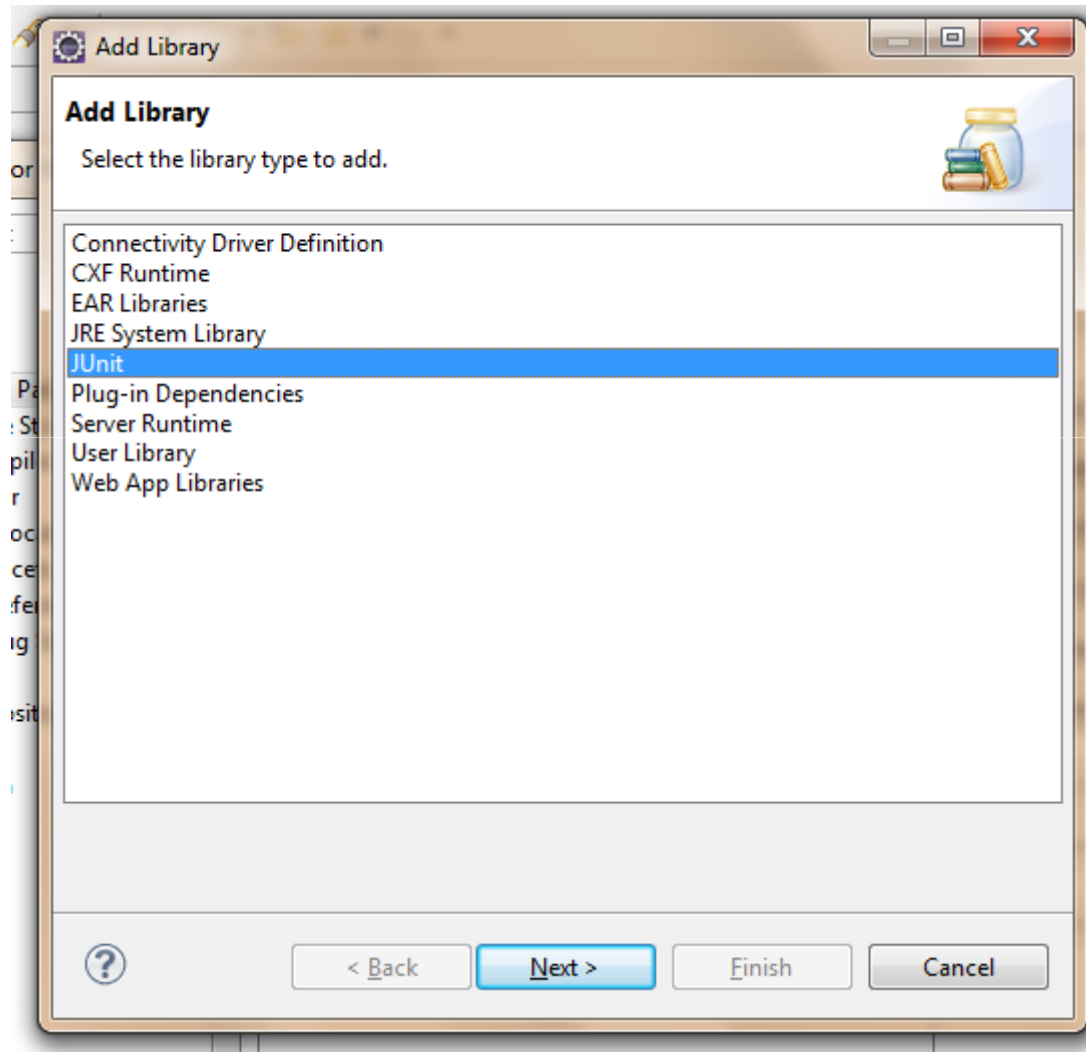
Select the libraries tab



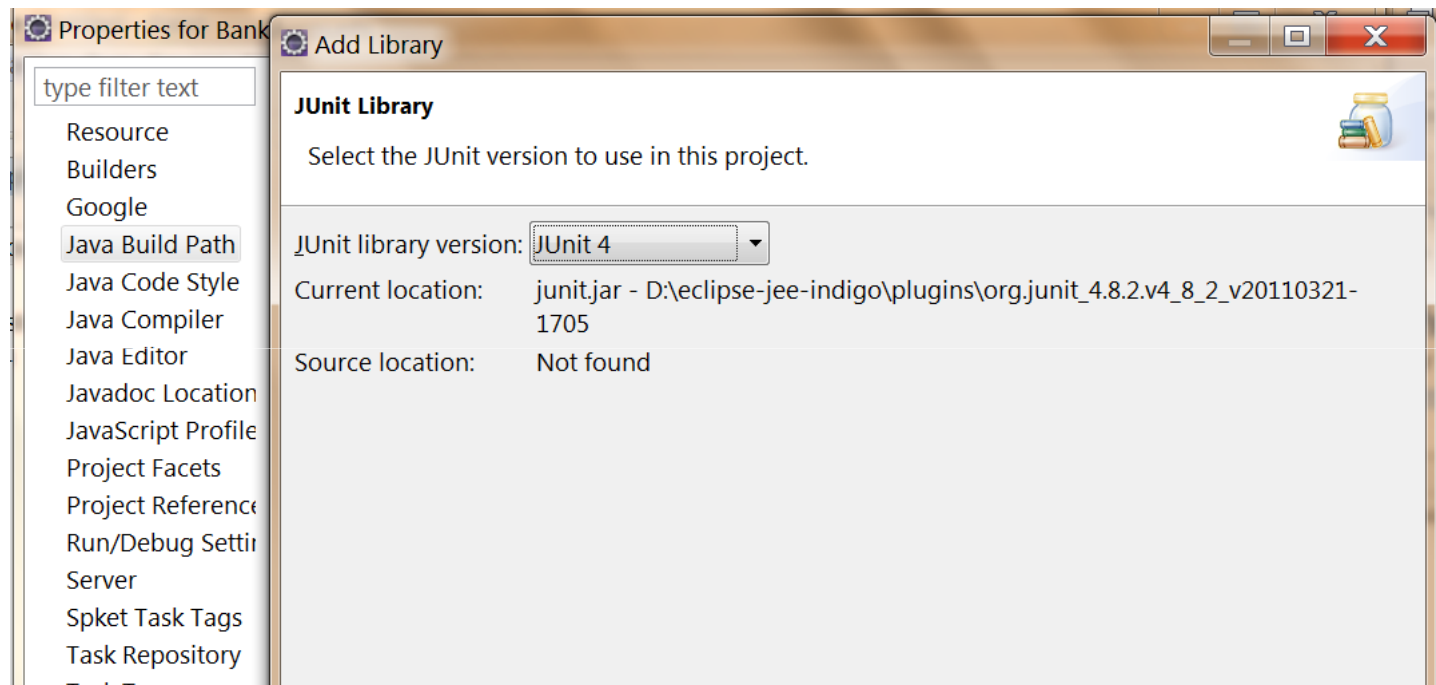
Select the JUnit library path

In the project build path screen select the libraries tab and click on the button 'Add Library' and select the JUnit option and click next button on bottom of screen

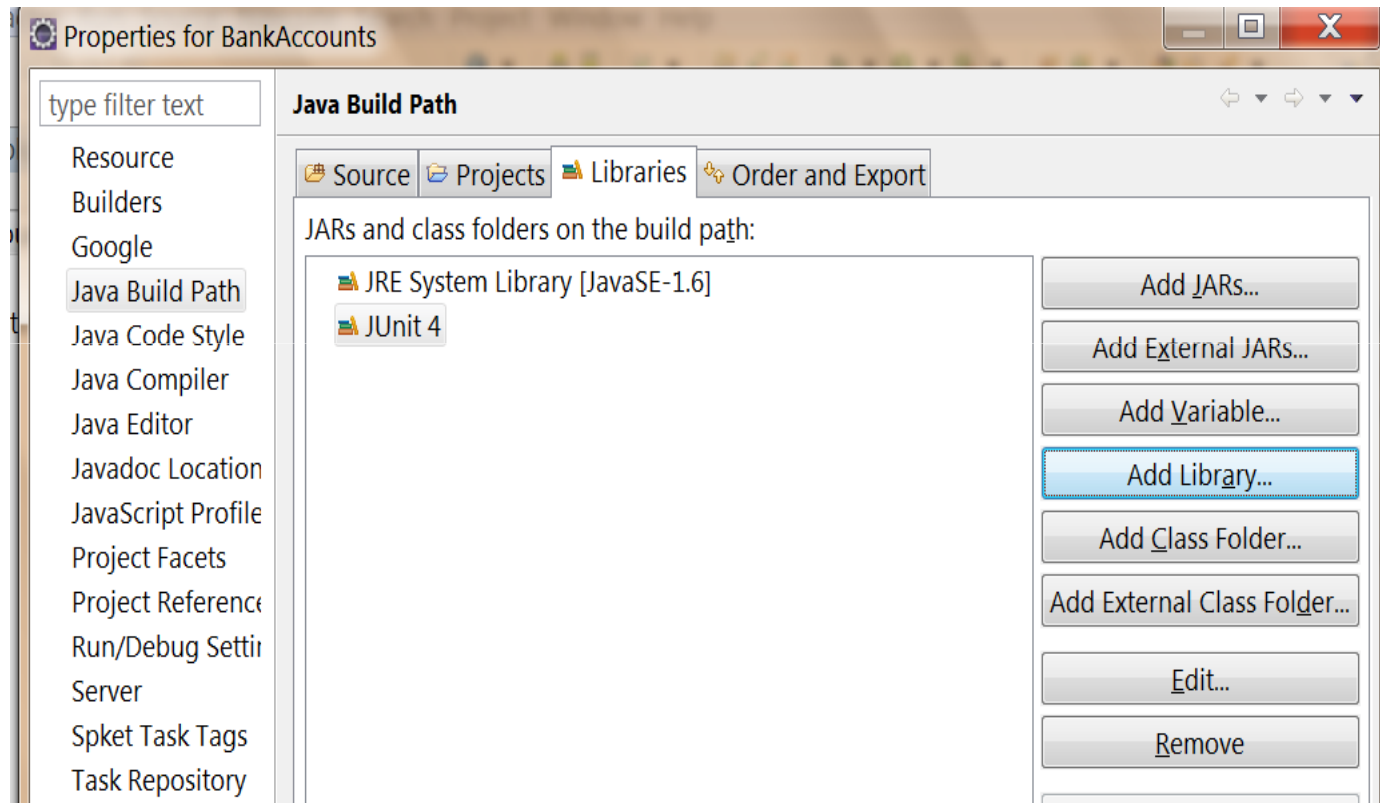
JUnit path



Select the JUnit library version 4

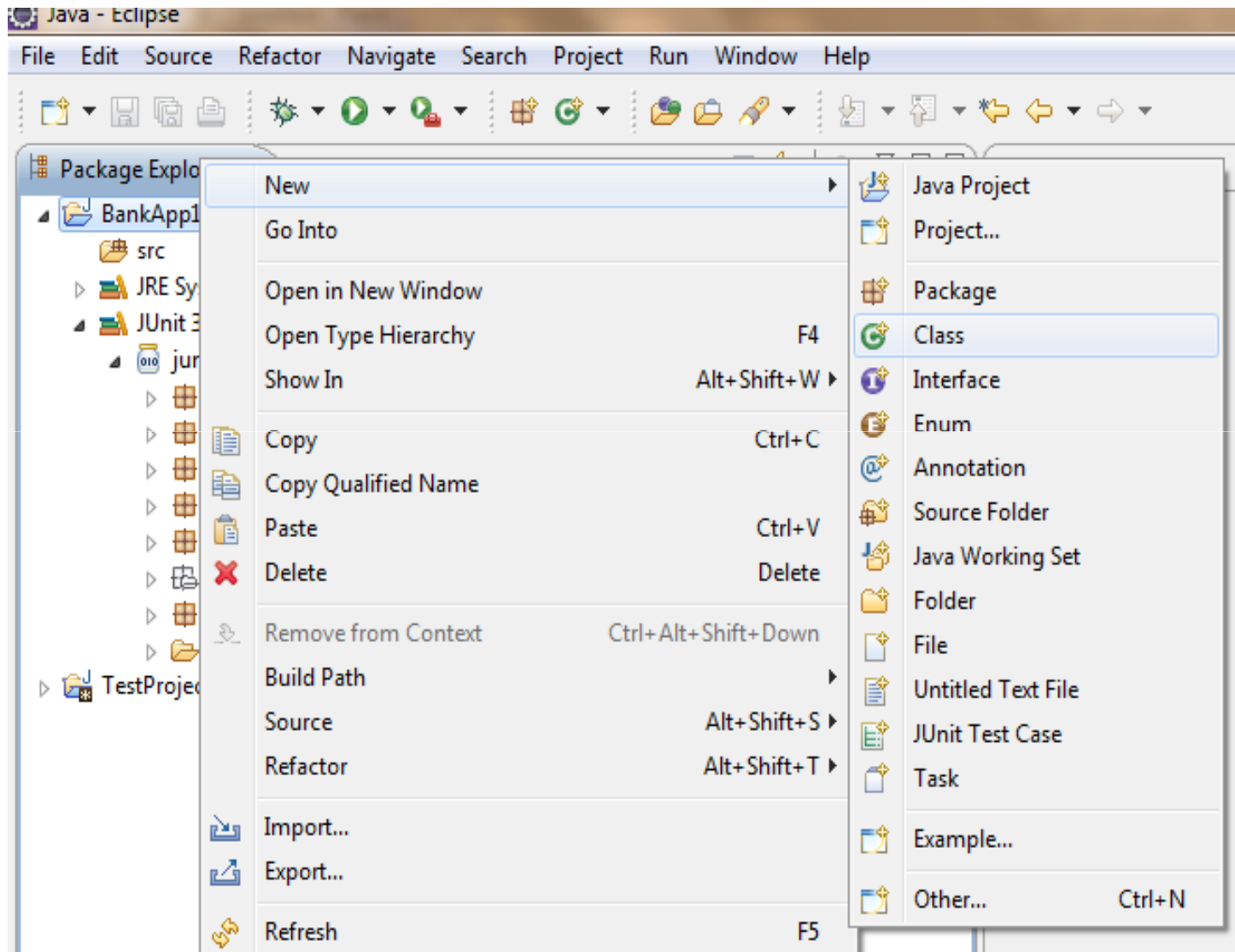


JUnit 4 library added on the classpath



Add a new Java Test Class

21



Java Test class 'TestSavingAccount'

22

New Java Class

Java Class
Create a new Java class.

Source folder: BankAccounts/src Browse...

Package: com.bank.test Browse...

☐ Enclosing type: Browse...

Name: TestSavingAccount

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add...
Remove

Which method stubs would you like to create?

The Test case class : The Test Fixture

23

```
package com.server.test;  
  
public class TestSavingAccount {  
  
}
```

Add the first Test method to Test class 24

```
package com.server.test;
```

```
import org.junit.Test;
```

```
public class TestSavingAccount {
```

```
    @Test
```

```
    public void testSavingAccountObject()
```

```
    {
```

```
    }
```

```
}
```



test method

Add variable to test class

```
package com.server.test;

import org.junit.Test;

public class TestSavingAccount {

    SavingAccount obj; //Compiler Error! Fix IT..

    @Test
    public void testSavingAccountObject()
    {

    }

}
```

Add the domain class SavingAccount

26

```
package com.server.bank;
```

```
public class SavingAccount
```

```
{
```

```
}
```

Add the test code


```
package com.server.test;

import junit.framework.Assert;
import org.junit.Test;
import com.server.bank.SavingAccount;

public class TestSavingAccount {

    SavingAccount obj;

    public void testSavingAccountObject()
    {
        Assert.assertNotNull(obj);
    }
}
```

 *Verify that the obj is not NULL..*

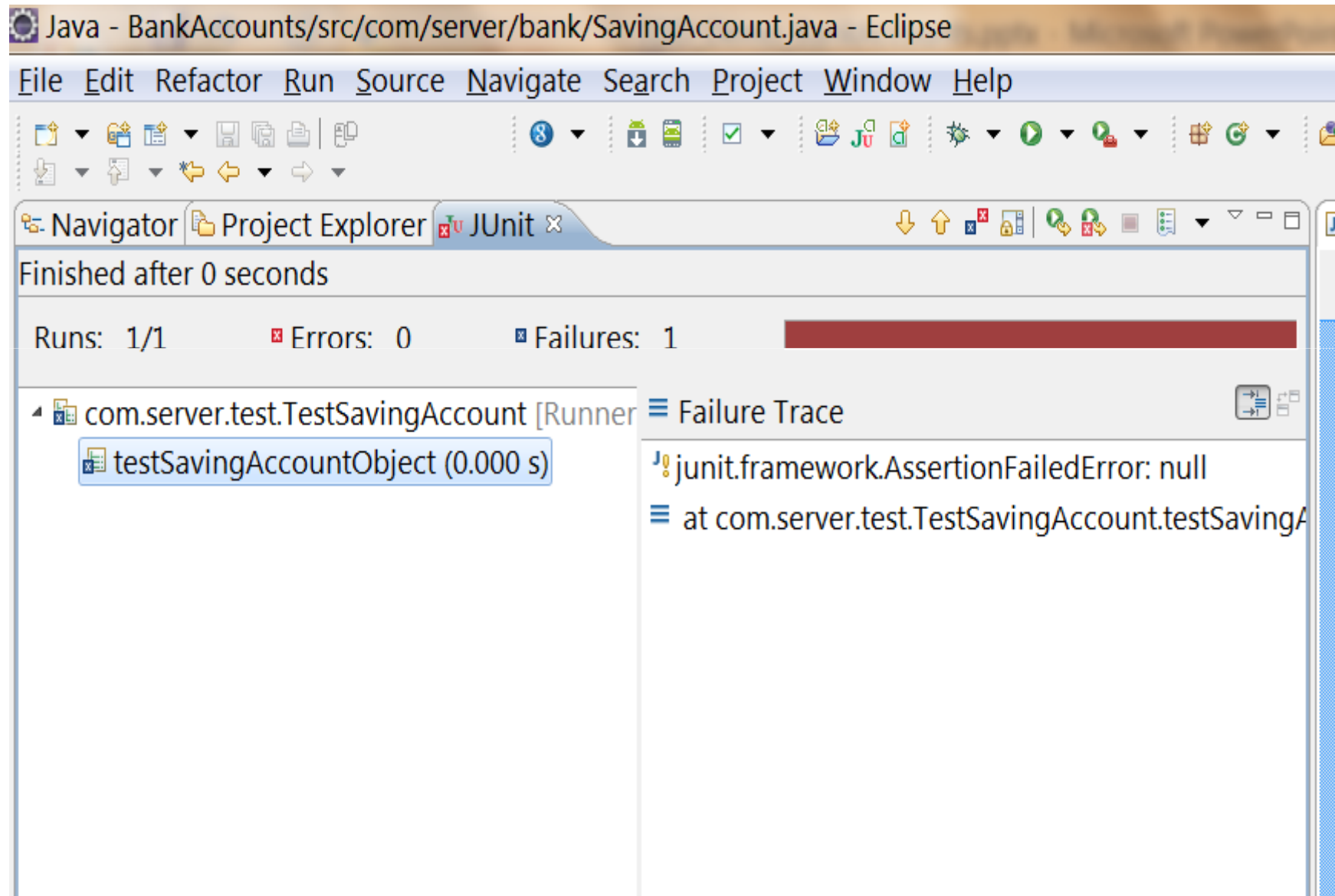
Run the Test Class as JUnit Test

28

- In Project Explorer view, Right click on the 'TestSavingAccount.java' and select ..
- Run As->JUnit Test option.

Congrats! Your First Test FAILED!

29



Test Failure description

30

junit.framework.AssertionFailedError: null

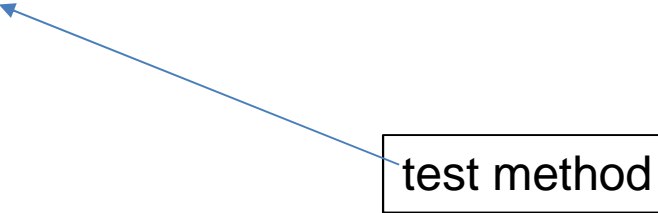
com.server.test.TestSavingAccount.testSavingAccountObject
(TestSavingAccount.java:35)

Why the failure as first test?

- The failure is most easy to implement!
 - Do Nothing!!
- We know the cause of failure.
- Incase if the test fails in a way not as we expected, something is wrong..fix it and correct it..helps to correct the errors easily..
- Helps to isolate the bugs at actual failures.

Add the next Test method to Test class ³²

```
package com.server.test;  
  
import org.junit.Test;  
  
public class TestSavingAccount {  
  
    @Test  
    public void checkSavingAccountDeposit()  
    {  
  
    }  
  
}
```




test method

Add the test code

33

```
package com.server.test;

import junit.framework.Assert;
import org.junit.Test;
public class TestSavingAccount {
    @Test
    public void checkSavingAccountDeposit()
    {
        //Compiler errors..fix it
        SavingAccount obj= new SavingAccount();
        obj.deposit(1000);
        int amount = obj.balance;
        Assert.assertEquals(1000, amount);
    }
}
```

 *Verify that the obj balance is equal to amount*

Add deposit method in SavingAccount ³⁴

Write the minimum required code ...

```
package com.server.bank;
```

```
public class SavingAccount {
```

```
public void deposit(int amount) {
```

```
}
```

```
}
```

Add the balance variable in SavingAccount

35

```
package com.server.bank;  
  
public class SavingAccount {  
  
    public int balance ;  
  
    public void deposit(int amount) {  
  
    }  
}
```

Run the Test

Congrats! Your Test FAILED!

37

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Run, Source, Navigate, Search, Project, Refactor, Window, and Help. Below the menu is a toolbar with various icons. The Package Explorer on the left shows the project structure, with the JUnit tab selected. The JUnit output window displays the test results: "Finished after 0 seconds", "Runs: 1/1", "Errors: 0", and "Failures: 1". A red bar indicates a failure. The test name is "checkSavingAccountDeposit [Runner: JUnit 4] (0.000)". The main editor shows the source code for "TestSavingAccount.java". The code includes imports for JUnit and the SavingAccount class. The test method "checkSavingAccountDeposit()" is highlighted in blue. The code is as follows:

```
package com.bank.test;

import org.junit.Assert;
import org.junit.Test;

import com.bank.server.SavingAccount;

public class TestSavingAccount {

    @Test
    public void checkSavingAccountDeposit()
    {
        SavingAccount obj= new SavingAccount();
        obj.deposit(1000);
        int amount = obj.balance;
        Assert.assertEquals(1000, amount);
    }
}
```

Test Failure description

38

```
java.lang.AssertionError: expected:<1000> but was:<0>
```

The first test as failure and NOT Success!

39

- The failure is most easy to implement!
 - Do Nothing!!
- We know the cause of failure.
- Incase if the test fails in a way not as we expected, something is wrong..fix it and correct it..helps to correct the errors easily..
- Helps to isolate the problems at actual failures
.

An Unstable System

The Symptom of an Unstable System is that there is no single obvious place to look at when there is failure in the system.

To make the Test succeed..

- In SavingAccount class initialize the balance variable value to 1000 .
- Run the Test..

Modified SavingAccount

42

```
package com.bank.server;
```

```
public class SavingAccount {
```

```
public int balance =1000;
```

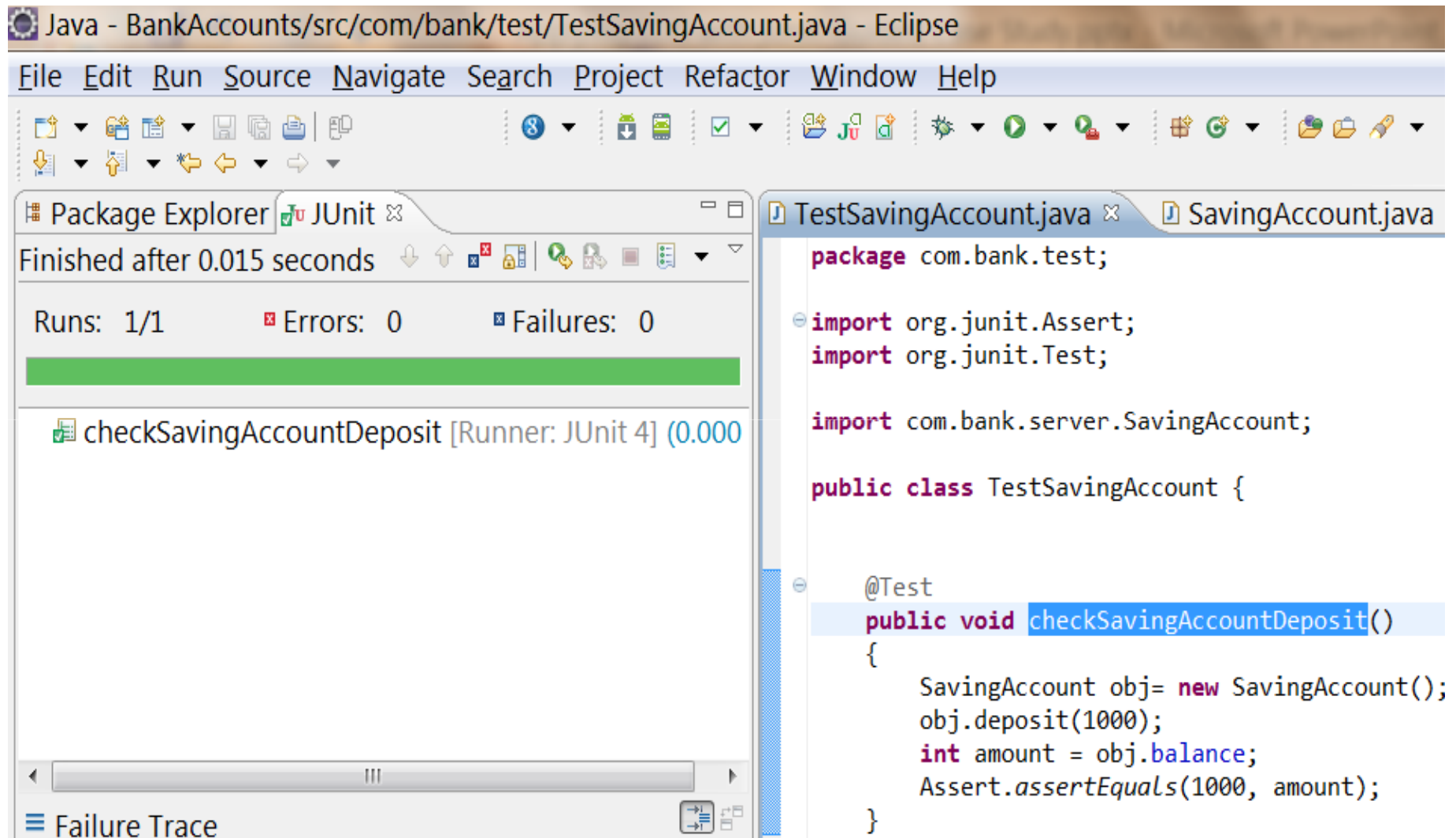
```
public void deposit(int i) {
```

```
}
```

```
}
```

Your First GREEN Bar of SUCCESS!

43



The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Run, Source, Navigate, Search, Project, Refactor, Window, and Help. Below the menu bar is a toolbar with various icons. The Package Explorer on the left shows the project structure, with the JUnit tab selected. The JUnit output window displays the results of a test run: "Finished after 0.015 seconds", "Runs: 1/1", "Errors: 0", and "Failures: 0". A green progress bar indicates a successful run. The test method "checkSavingAccountDeposit" is listed with a duration of 0.000 seconds. The main editor window shows the source code for "TestSavingAccount.java". The code includes imports for JUnit and the SavingAccount class, and a test method "checkSavingAccountDeposit" that creates a SavingAccount object, deposits 1000, and asserts that the balance is 1000.

```
package com.bank.test;

import org.junit.Assert;
import org.junit.Test;

import com.bank.server.SavingAccount;

public class TestSavingAccount {

    @Test
    public void checkSavingAccountDeposit()
    {
        SavingAccount obj= new SavingAccount();
        obj.deposit(1000);
        int amount = obj.balance;
        Assert.assertEquals(1000, amount);
    }
}
```

Although the result was Faked..

Test for the amount deposited..

44

- How to verify the amount that is deposited number of times in SavingAccount ?
- Add another test function 'VerifyDepositsOnSavingAccount' in Test class to verify the amount deposited.. .
- Inside the test function create the SavingAccount object , call the deposit function two times with different amount values and verify the balance on the SavingAccount object with assertion.

The test

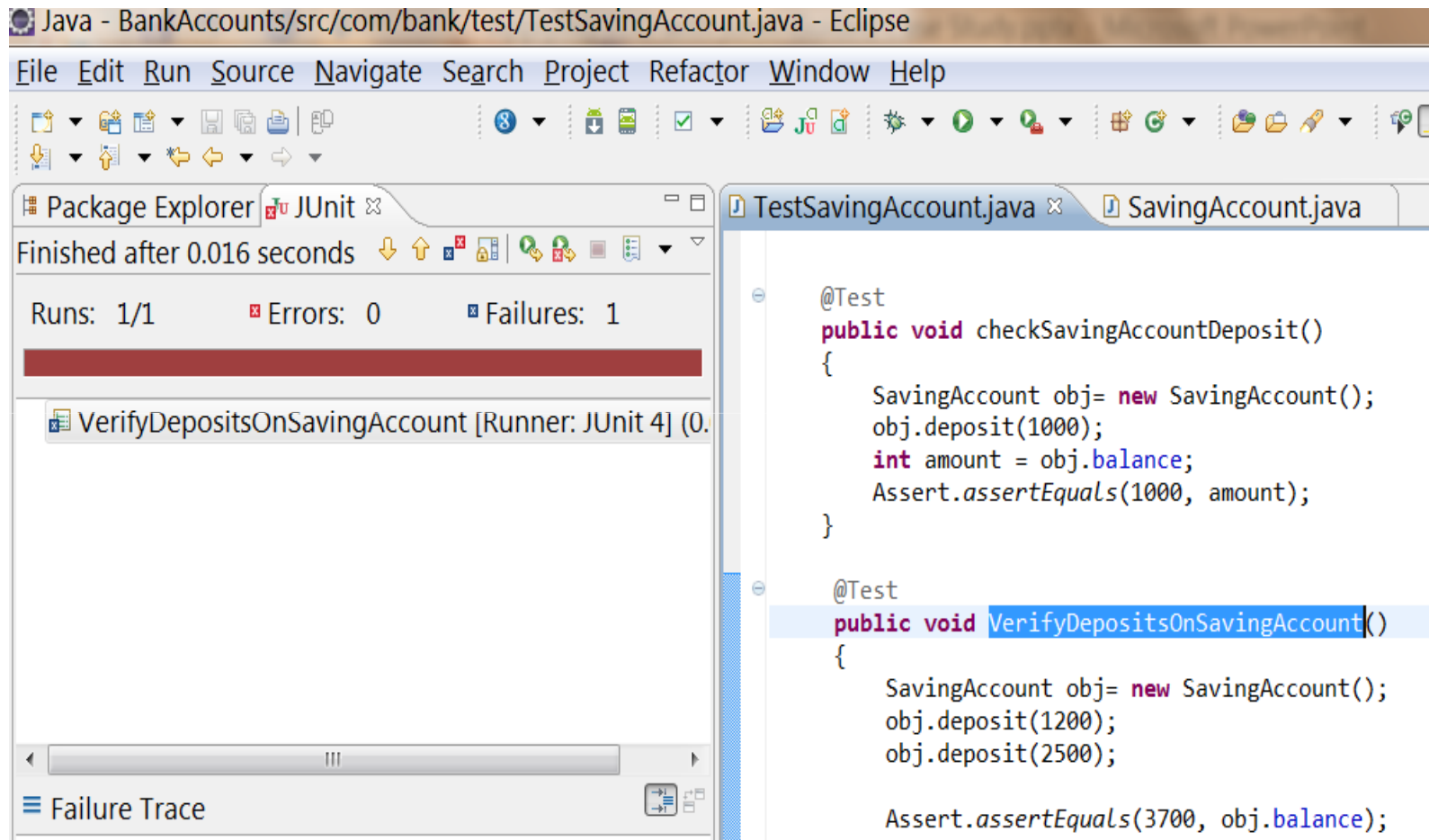
45

```
@Test
public void VerifyDepositsOnSavingAccount()
{
    SavingAccount obj = new SavingAccount();
    obj.deposit(1000);
    obj.deposit(2400);

    //verify the final balance value
    Assert.assertEquals(3400, obj.balance);
}
```

Run the test..

46



junit.framework.AssertionFailedError: expected:<3700> but was:<1000>

Why the test failed ?

- We have earlier faked the balance value with fixed value..
- Now we will have modify the SavingAccount with minimum code required to update the balance added by the deposit method.

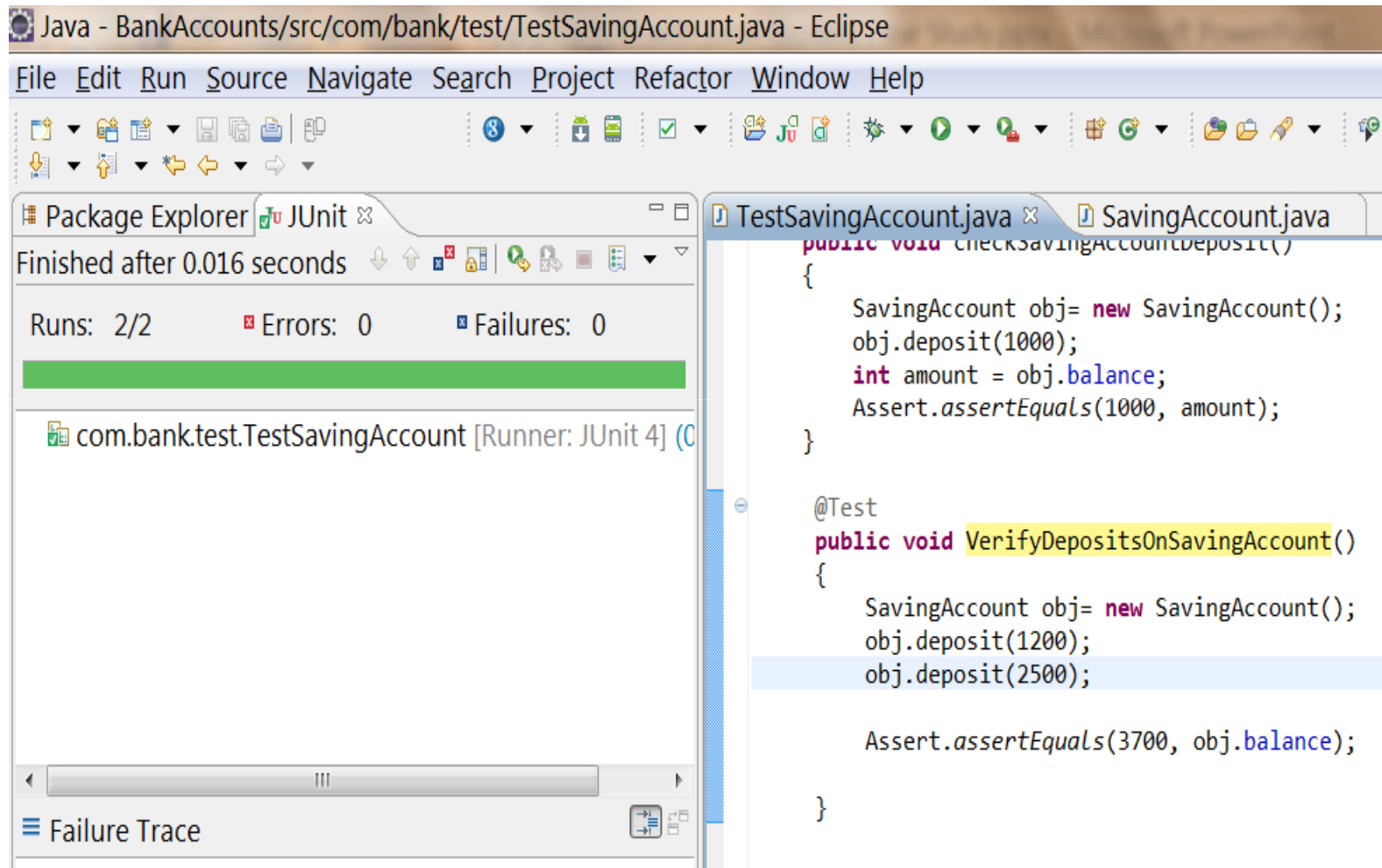
SavingAccount Deposit

48

```
public class SavingAccount {  
  
    public int balance;  
  
    public void deposit(int amount) {  
  
        balance+= amount;  
  
    }  
}
```

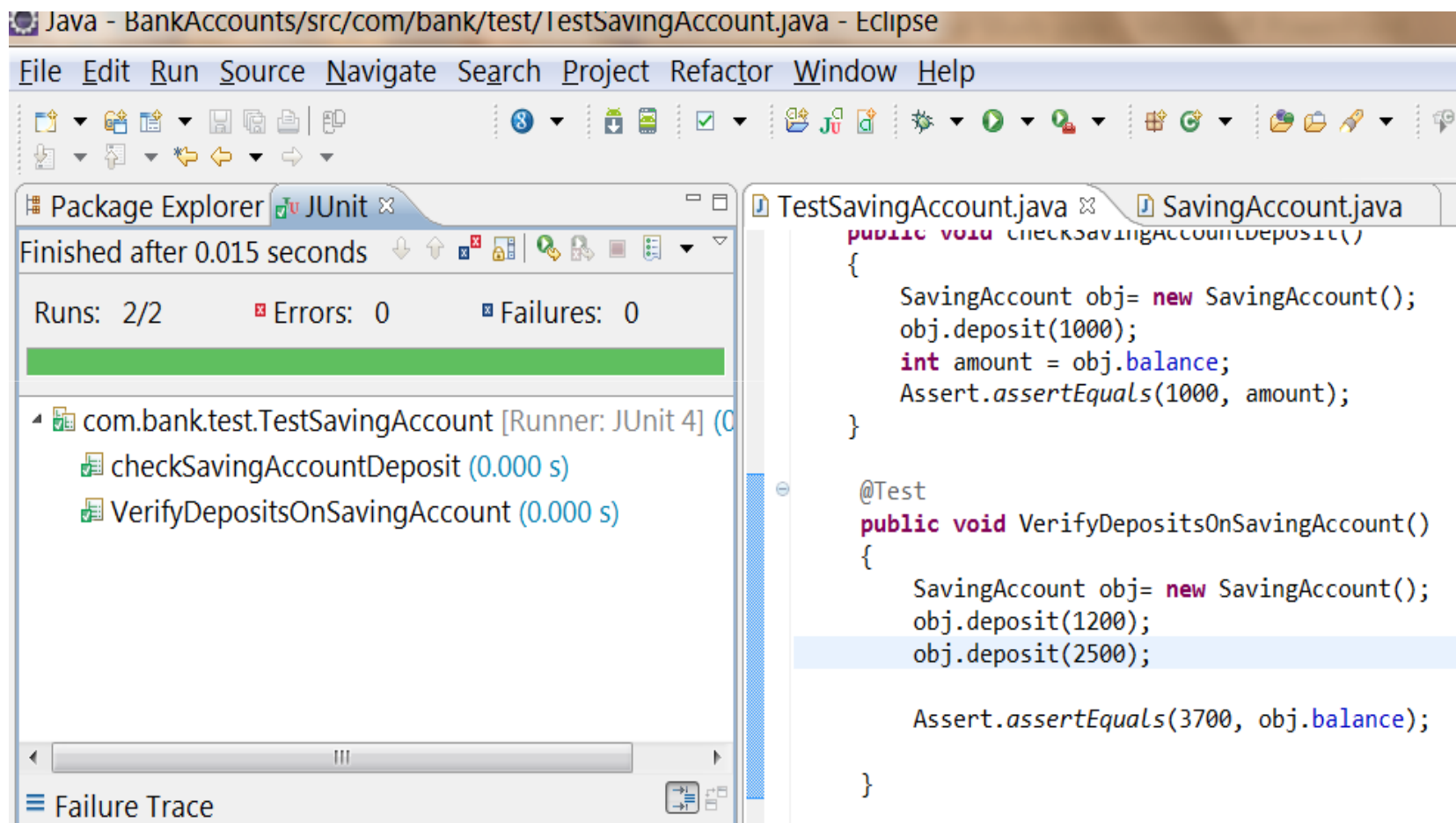

Run the Test

49



All other tests ?

50



All the tests are successful!!!

What is done ?

- Defined balance variable in SavingAccount class and exposed it to the user.
- Update the balance of SavingAccount by deposit method.

Correct it..

- Limit the access of balance in SavingAccount to outside world by making it private.
- But to share the balance add a public method **getBalance** in SavingAccount class.
- In the test class instead of using the balance directly change it to call getBalance on SavingAccount object.

SavingAccount Revised

53

```
public class SavingAccount {  
  
    private int balance;  
  
    public void deposit(int amount) { }  
    public int getBalance()  
    {  
        return balance;  
    }  
}
```

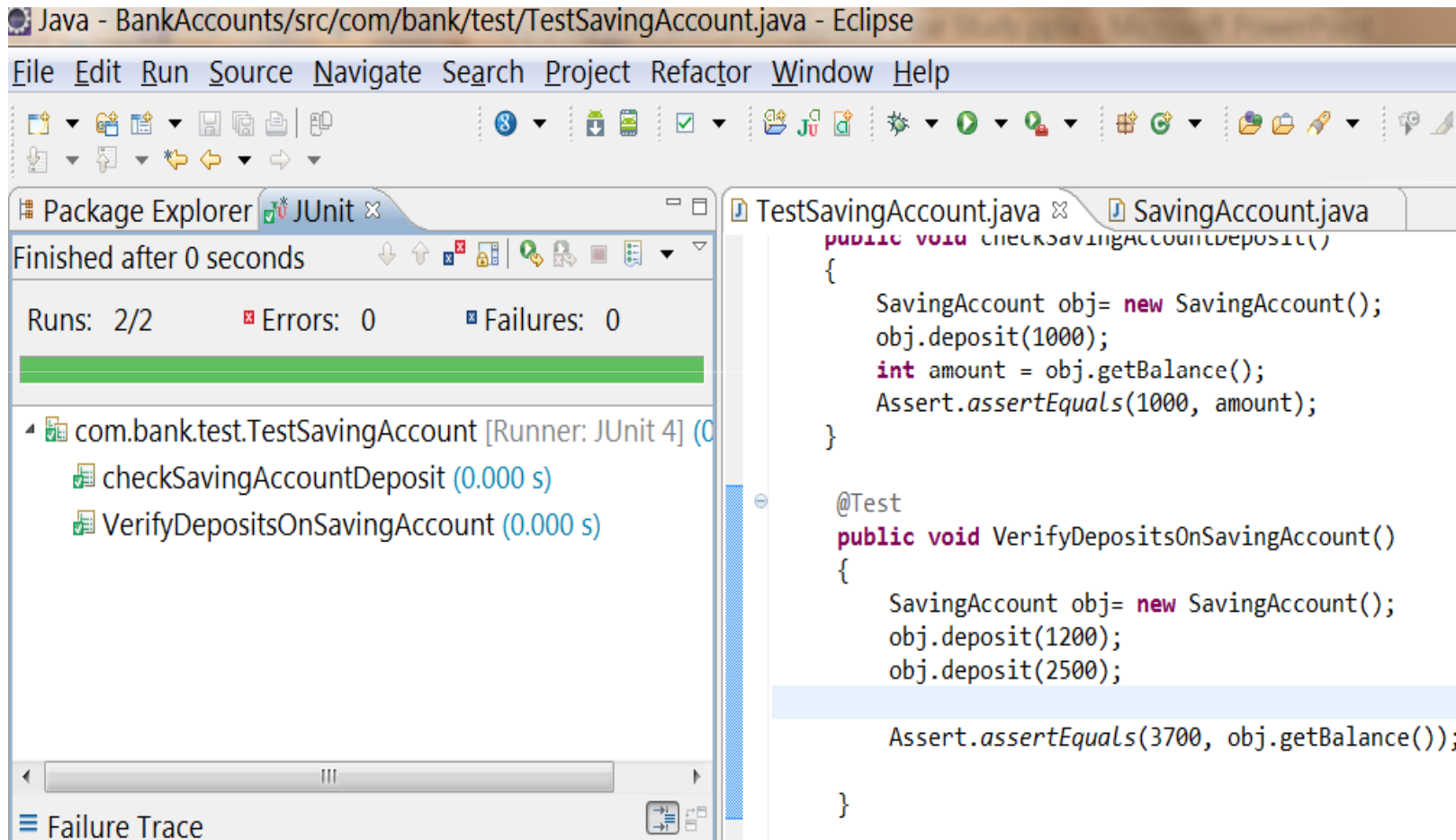
Call getBalance in Test Class

```
@Test
public void VerifySavingAccountBalance() {
    SavingAccount obj = new SavingAccount();
    obj.deposit(1200);
    Assert.assertEquals(1200, obj.getBalance());
}

@Test
public void VerifySavingAccountMultipleDeposits(){
    SavingAccount obj = new SavingAccount();
    obj.deposit(1000);
    obj.deposit(2400);
    Assert.assertEquals(3400, obj.getBalance());//verify the balance
}
```

Run all the tests

55



All the tests have succeeded..

How much to test in one step ?

56

- How much to test in one test case ?
- TDD advises implementing smaller short features one by one.
- That makes the design incremental and iterative.

Steps in TDD

- How much code should we write at one step ?
 - Just the minimum to make the test success.
- How many tests should we write at a time..?
 - Ideally one at a time. But if you are sure about the design upfront keep all the test cases defined and leave it to the dev team to implement it.
- How big jump should we make in code ?
 - Just minimum to make the test success.

Baby steps or High Jumps ?

58

- Writing baby steps is always safe.
- But if you are sure about the next logical implementation, go on adding it.
- But make sure that the added code is completely tested for success.

Additional requirements

- No negative amount to deposit
- No negative amount to withdraw
- Keep minimum balance after withdraw
- Throw error in case NO compliance with above

What if

- The SavingAccount implementation described here is for Indian banking environment where Rupee is the default currency..
- What if an NRI person wants to remit the amount in US Dollars to the corresponding Indian SavingAccount ?
- How this US Dollars amount will be accommodated here ?
- Obviously the US Dollars amount need to be first converted to Indian Rupees and then transacted...

The deposit Dollar test method

61

@Test

```
public void testSavingAccountDollarTransfer()  
{
```

```
    obj = new SavingAccount();  
    obj.deposit(1000);  
    obj.deposit(100);
```

//Compiler Errors Fix it

Dollar usCurrency = new Dollar(100);

obj.deposit(usCurrency);

*double amount = Dollar.getRupeesConvresionRate() *
usCurrency.getCurrencyValue();*

//verify the current balance

Assert.assertEquals(1000+amount, obj.getBalance());

```
}
```

Overloaded Deposit methods

62

```
public void deposit(int amount) {
```

```
    balance+=amount;
```

```
}
```

```
public void deposit(Dollar dollarAmount) {
```

```
    int dollarValue = dollarAmount.getCurrencyValue();
```

```
    int rupeeRate =
```

```
        dollarAmount.getRupeesConvresionRate();
```

```
    int amount = dollarValue * rupeeRate;
```

```
    balance+=amount;
```

```
}
```

The Dollar Class

63

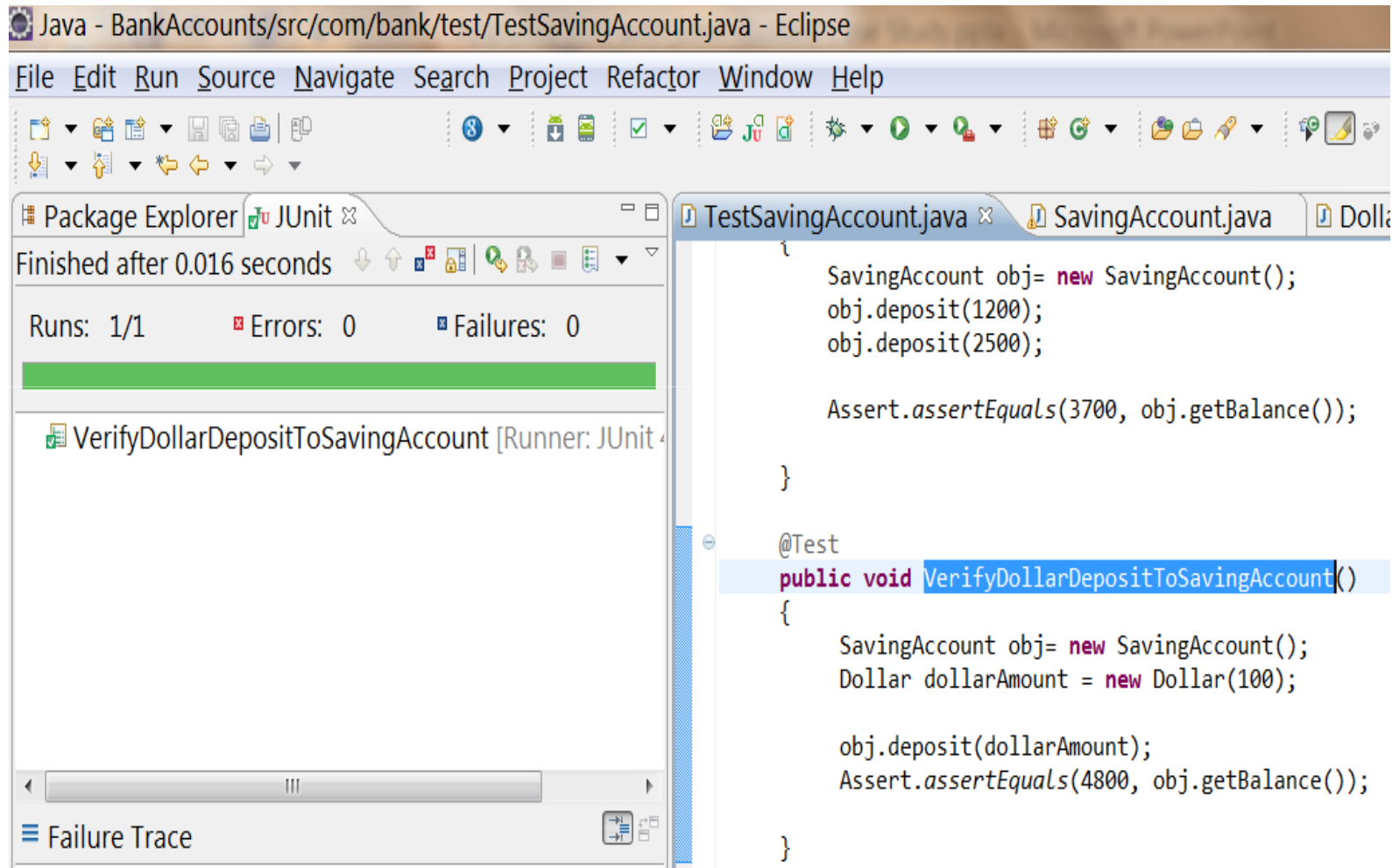
```
package com.bank.server;

public class Dollar {
    private int currencyValue;

    public Dollar(int value){
        this.currencyValue = value;
    }
    int getCurrencyValue() {
        return currencyValue;
    }
    private static int rupeesConvresionRate = 48;
    static int getRupeesConvresionRate() {
        return rupeesConvresionRate;
    }
}
```

Run test

64



The taste of success!

Run all the Tests

65

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Run, Source, Navigate, Search, Project, Refactor, Window, and Help. Below the menu is a toolbar with various icons. The Package Explorer on the left shows the project structure, with the JUnit tab selected. The JUnit results view shows that the tests finished after 0.016 seconds, with 3 runs, 0 errors, and 0 failures. The test results list includes:

- com.bank.test.TestSavingAccount [Runner: JUnit 4] (0.000 s)
- checkSavingAccountDeposit (0.000 s)
- VerifyDepositsOnSavingAccount (0.000 s)
- VerifyDollarDepositToSavingAccount (0.000 s)

The main editor area displays the source code for TestSavingAccount.java. The code is as follows:

```
1 SavingAccount obj= new SavingAccount();
  obj.deposit(1200);
  obj.deposit(2500);

  Assert.assertEquals(3700, obj.getBalance());
}

@Test
public void VerifyDollarDepositToSavingAccount()
{
    SavingAccount obj= new SavingAccount();
    Dollar dollarAmount = new Dollar(100);

    obj.deposit(dollarAmount);
    Assert.assertEquals(4800, obj.getBalance());
}
```

One more Currency..

66

What if the user wants to transfer the Euro currency to this SavingAccount ?

add a test method

```
@Test
public void VerifyEuroDepositToSavingAccount()
{
    SavingAccount obj= new SavingAccount();
    Euro euroAmount = new Euro(60);

    obj.deposit(euroAmount);
    Assert.assertEquals(1320, obj.getBalance());

}
```

Evolve the Euro class

- Define the Euro class to be used as a ValueObject to make the transactions.
- The attributes required for Dollar
 - currencyValue
 - rupeesConvresionRate for converting to Indian Rupees.

Euro class

69

```
package com.bank.server;

public class Euro {
    private int currencyValue;

    public Euro(int value){
        this.currencyValue = value;
    }

    int getCurrencyValue() {
        return currencyValue;
    }

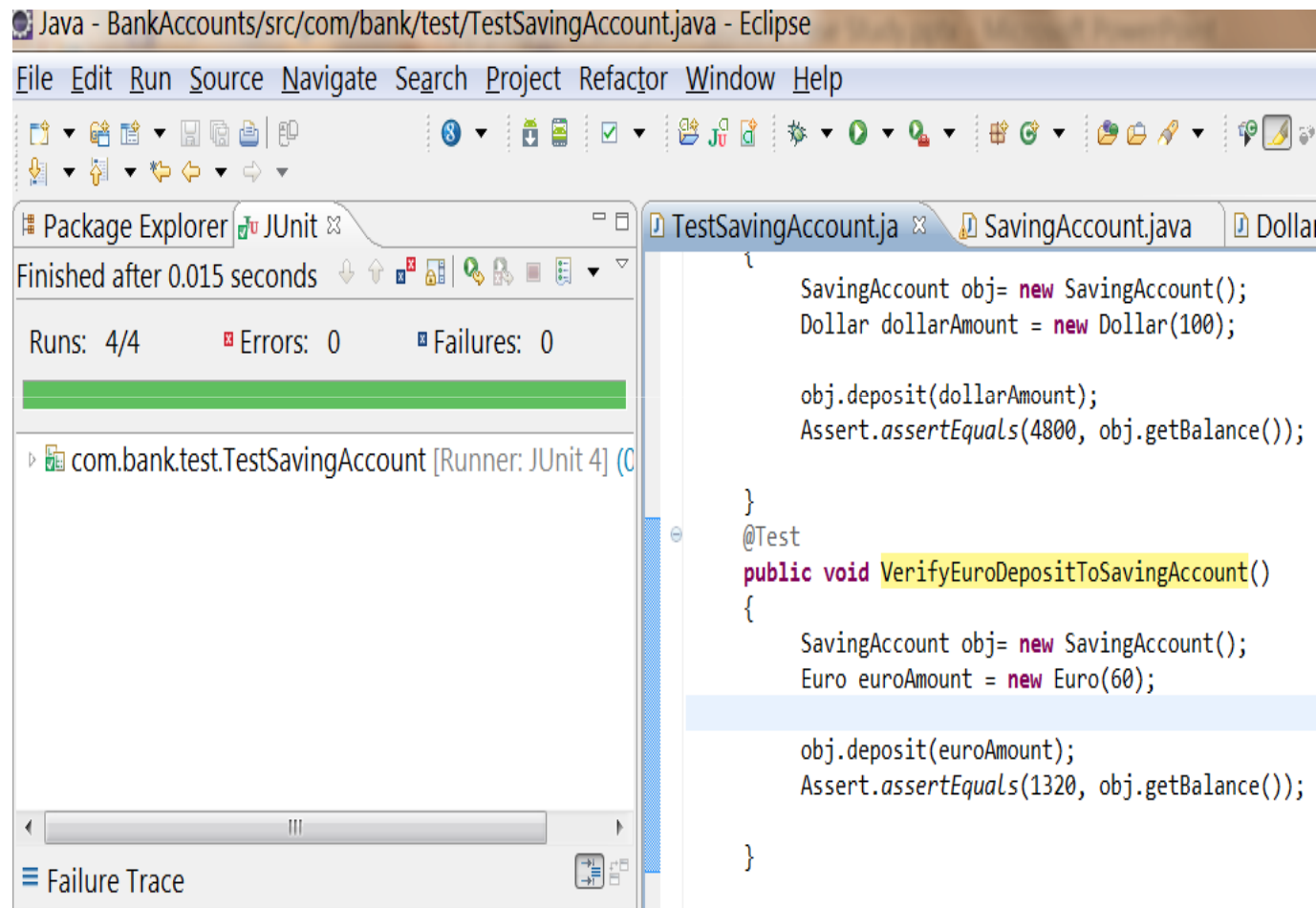
    private static int rupeesConvresionRate = 22;
    static int getRupeesConvresionRate() {
        return rupeesConvresionRate;
    }
}
```

Overloaded deposits

```
public void deposit(Dollar dollarAmount) {  
    int dollarValue = dollarAmount.getCurrencyValue();  
    int rupeeRate = dollarAmount.getRupeesConvresionRate();  
    int amount = dollarValue * rupeeRate;  
    balance+=amount;  
}  
  
public void deposit(Euro euroAmount) {  
    int euroValue = euroAmount.getCurrencyValue();  
    int euroRate = euroAmount.getRupeesConvresionRate();  
    int amount = euroValue * euroRate;  
    balance+=amount;  
}
```

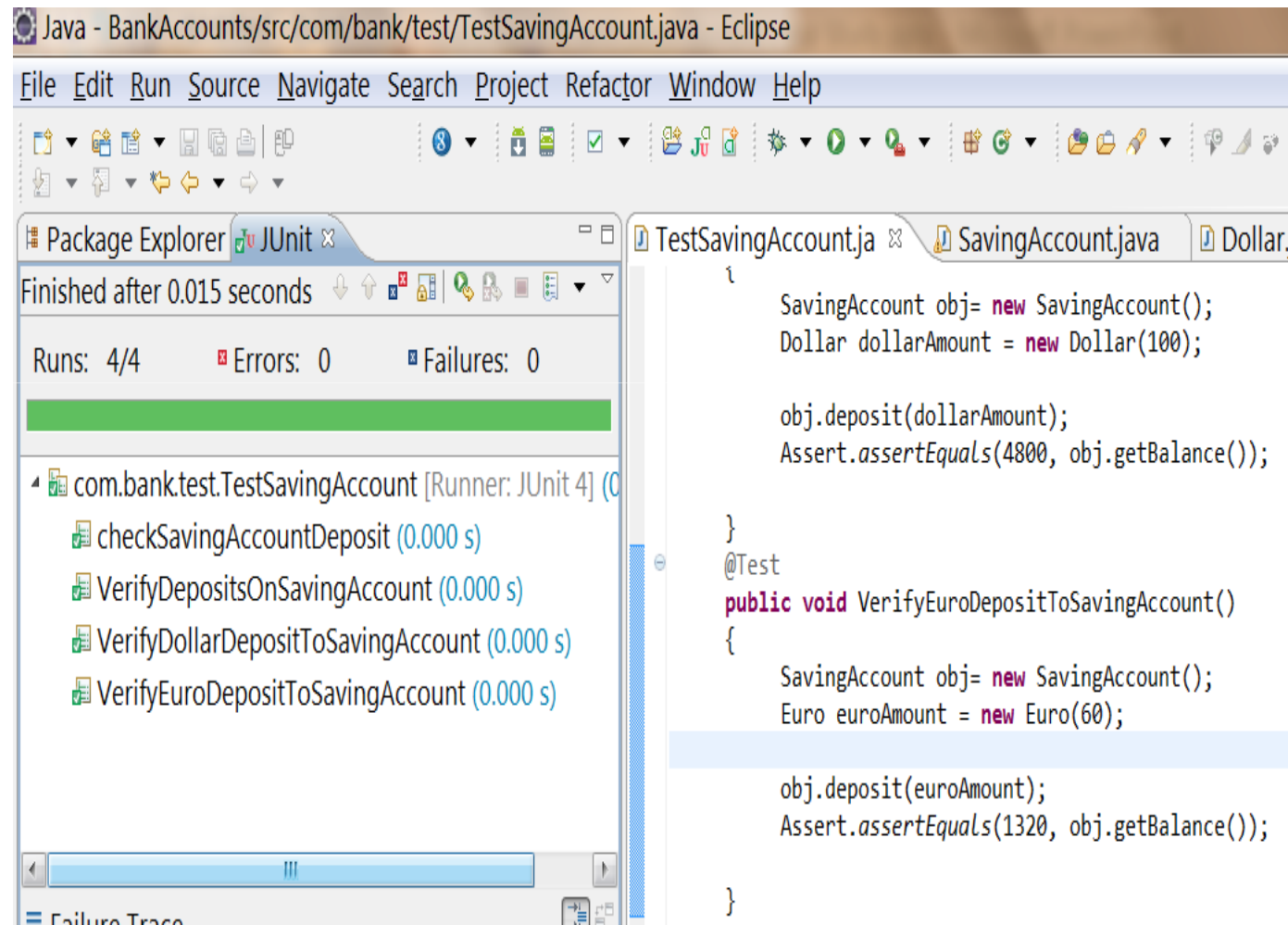
Run the Test

71



Run All the Tests

72



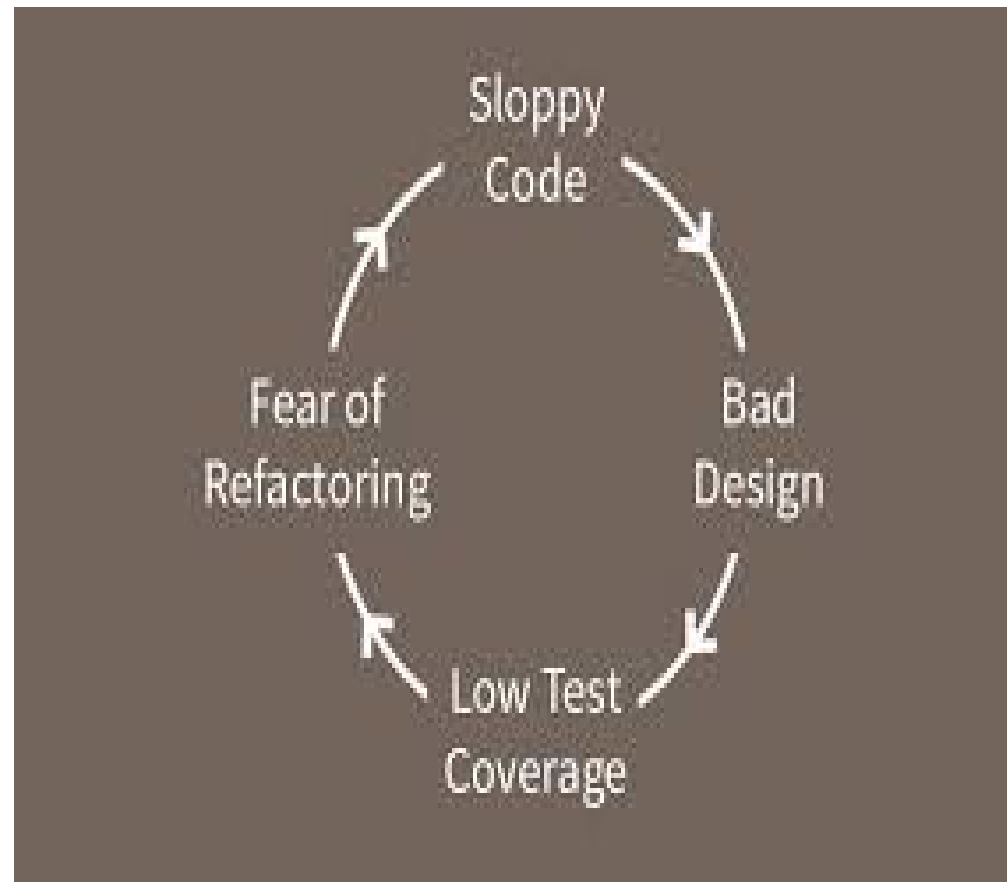
Refactoring

What is Refactoring?

- **Improving the design of existing code without changing its external behavior.**
- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Its heart is a series of small behavior preserving transformations.

Need of refactoring

75



Why Do we refactor ?

- Improves the design of the software.
- Easier to maintain and understand.
- Easier to facilitate change.
- More flexibility.
- Increased reusability.
- To help find and locate the bugs easily

When do we refactor ?

- Duplicate code.
- Improper naming
- Large class
- Class with multiple different unrelated behavior
- Large switch-case blocks makes it difficult to add one more case block
- Long methods.
- Long parameter list for methods.
- Large no of local variables
- Dead code(un used)
- Hard coupling between classes
- Improper naming of variables/classes/methods.

What to refactor ?

- Programs that are hard to read (hard to modify).
- Programs that have duplicate logic(hard to modify)
- Programs that require additional behavior that requires you to change running code(hard to modify.)
- Programs with complex conditional logic(hard to modify.)

Rules for Refactoring

- Each transformation (called a 'Refactoring') does little, but a sequence of transformations can produce a significant restructuring.
- The system should be kept fully working after each small Refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Steps in Refactoring

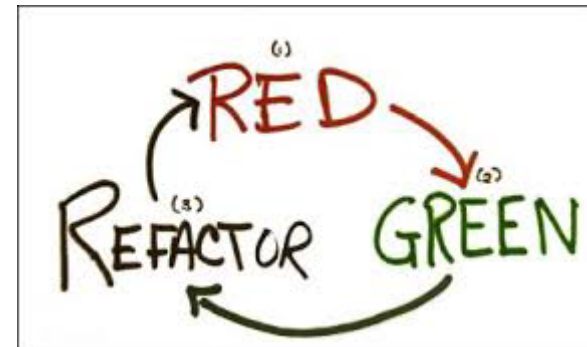
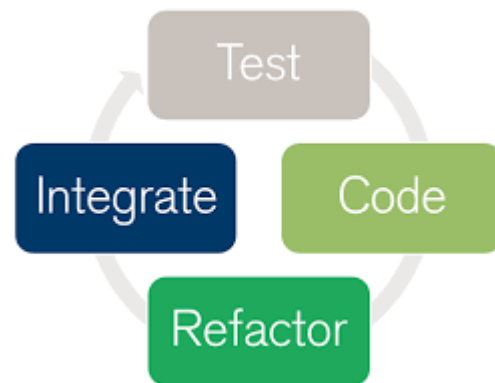
- Ensure the test exists for testing current code behavior.
- Make sure all the tests succeed before starting the refactoring.
- Make one refactoring change.
- Execute the test for that change done and ensure success.
- Execute all the test cases and ensure the success for all the tests.

Refactoring the Code..

81

Each transformation (called a 'Refactoring') does little, but a sequence of transformations can produce a significant restructuring.

The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.



Refactor and go GREEN!..

82

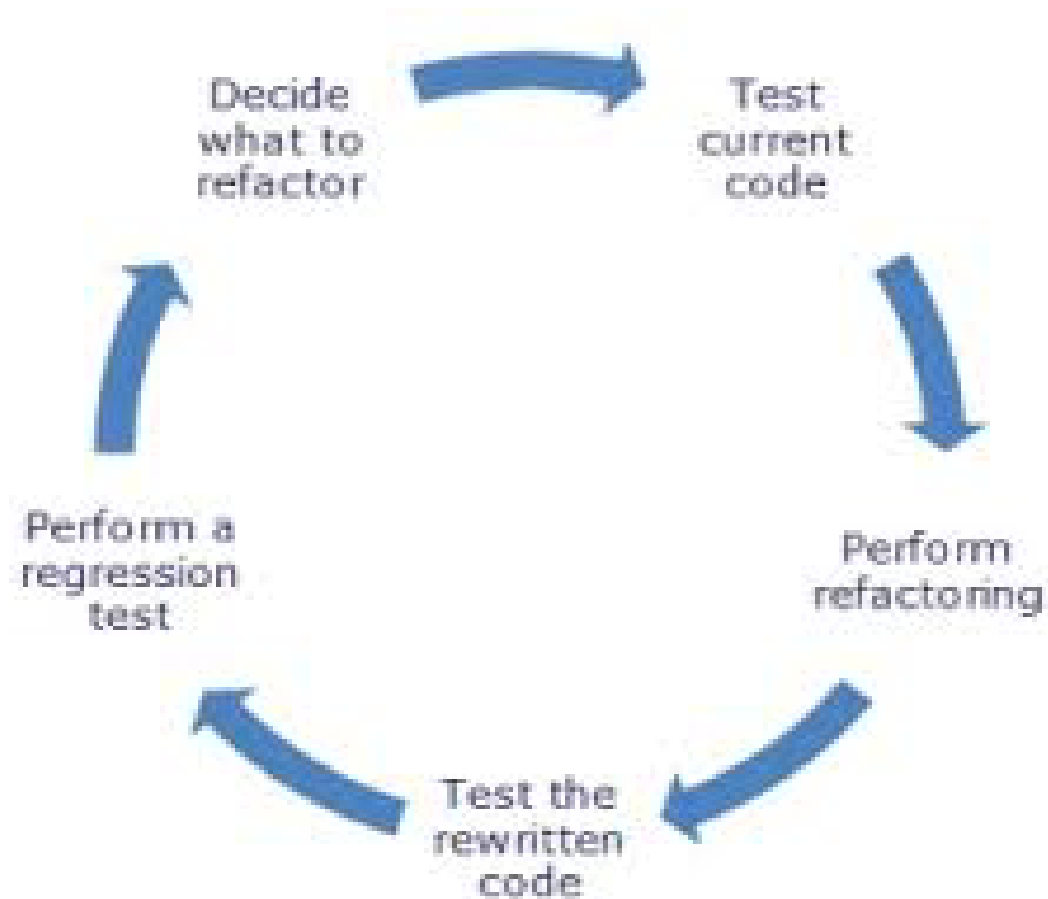
- Extract the functions out of existing duplicate code.
- Remove duplications without changing the external behaviors of functions.. All the tests must succeed.
- Run all the tests again..
- It must be GREEN bar always..

Refactoring advantage

- The development tasks, maintenance and enhancement, often conflict since new features, especially those that do not fit cleanly within the original design, result in an increased maintenance effort.
- The refactoring process aims to reduce this conflict, by aiding non destructive changes to the structure of the source code, in order to increase code clarity and maintainability.
- When a system's source code is easily understandable, the system is more maintainable, leading to reduced costs and allowing precious development resources to be used elsewhere.
- For the well structured code, new requirements can be introduced more efficiently and with less problems.
- However many developers and managers are hesitant to use refactoring. The most obvious reasons for this is the amount of effort required for even a minor change, and a fear of introducing bugs. These problems can be solved by using an automated refactoring tool and well defined set of test cases to ensure re-testing of refactored code again and again.

Refactoring in TDD Cycle

84



Refactoring principles

- Each refactoring should be a simple process which makes one logical change to the structure of the code.
- When changing a lot of the code at one time it is possible that bugs get introduced. But when and where these bug were created is no longer reproducible.
- If however, a change is implemented in small steps with tests running after each step, the bug likely surfaces in the test run immediately after introducing it into the system. Then the step could be examined or, after undoing the step, it could be split in even smaller steps which can be applied afterwards.

Tests for Refactored code

- The benefit of comprehensive unit tests in a system, is that the system behavior remains intact even after refactoring.
- These tests, give the developers and management confidence that the refactoring has not broken the system, the code behaves the same way as it behaved before.

Refactoring techniques

- Extract small methods from large method
- Rename long named method
- Reduce number of parameters
- Make the code inline from another method to save the call.
- Extract duplicate variable in super class
- Extract super class from two or more classes
- Extract class from big class and make it composite
- Extract interface from multiple common behaved classes
- Rename classes and variables
- Move class in related packages
- Replace conditional logic with Polymorphism

Refactoring with tools

- When doing refactoring the externally observable behavior of the code must be guaranteed to stay the same.
- If refactoring are carried out manually, one needs to frequently rebuild the system and run tests.
- Manual refactoring is applicable the following conditions hold:
- The system, of which the refactored code is a part, can be rebuilt *quickly*.
- There are automated "regression" tests that can be frequently run to verify the impact of refactoring.

Refactoring to patterns

- Replace state altering conditionals with state pattern e.g. door close/open
- Move creation knowledge to Factory
- Extract Composite
- And on and on.....

Currency Abstraction

- Extract the top level abstract class as Currency from Dollar and Euro classes and override the required methods
- Currency as top level abstract class with required abstract methods.
- Define Dollar and Euro classes extending and overriding the currency conversion and value methods.

The Currency Parent

```
package com.server.bank;
```

```
abstract class Currency {
```

```
    abstract double getRupeesConvresionRate() ;
```

```
    abstract int getCurrencyValue() ;
```

```
}
```

The Dollar revised

```
package com.server.bank;

public class Dollar extends Currency {
    private int currencyValue;
    int getCurrencyValue() {
        return currencyValue;
    }
    private static int rupeesConvresionRate = 48.6 ;
    Int getRupeesConvresionRate() {
        return rupeesConvresionRate;
    }
    public Dollar(int currentValue) {
        this.currencyValue = currentValue;
    }
}
```

Revised Euro Class

```
package com.server.bank;
public class Euro extends Currency {
    private int currencyValue;
    private static int rupeesConvresionRate = 25 ;
    Euro(int currentValue) {
        this.currencyValue = currentValue;
    }
    Int getRupeesConvresionRate() {
        return rupeesConvresionRate;
    }
    public int getCurrencyValue() {
        return currencyValue;
    }
}
```

Modify the SavingAccount

```
public class SavingAccount {
```

```
    private int balance;
```

```
    public void deposit(int amount) {  
        balance+= amount;  
    }
```

```
        public void deposit(Currency currencyType) {  
            double amount = currencyType.getRupeesConvresionRate() *  
            currencyType.getCurrencyValue();  
            balance+= amount;  
        }
```

Update the test method code

```
@Test
public void testSavingAccountDollarTransfer()
{
    obj = new SavingAccount();
    obj.deposit(1000);
    Currency usCurrency = new Dollar(100);
    obj.deposit(usCurrency);

    int amount = (int) (usCurrency.getRupeesConvresionRate() *
usCurrency.getCurrencyValue());

    //verify the current balance
    Assert.assertEquals( 1000+amount, obj.getBalance());
}
```

Run the test

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Refactor, Run, Source, Navigate, Search, Project, Window, and Help. Below the menu is a toolbar with various icons. The left sidebar contains the Navigator, Project Explorer, and JUnit views. The JUnit view shows a successful test run for 'testSavingAccountDollarTransfer' [Runner: JUnit 4] (0.0 seconds), with 'Runs: 1/1', 'Errors: 0', and 'Failures: 0'. The main editor displays the source code of 'TestSavingAccount.java'. The code includes a package declaration, imports for 'Assert' and 'JUnit4', a class declaration, and a test method 'testSavingAccountDollarTransfer'. The test method creates a 'SavingAccount' object, deposits 1000, creates a 'Dollar' object, and deposits it. A tooltip for 'junit.framework.Assert' is visible, stating 'A set of assert methods. Messages are only displayed if the assertion fails.' The code snippet is as follows:

```
package com.server.test;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import com.server.account.SavingAccount;
import com.server.currency.Dollar;

@RunWith(JUnit4.class)
public class TestSavingAccount {

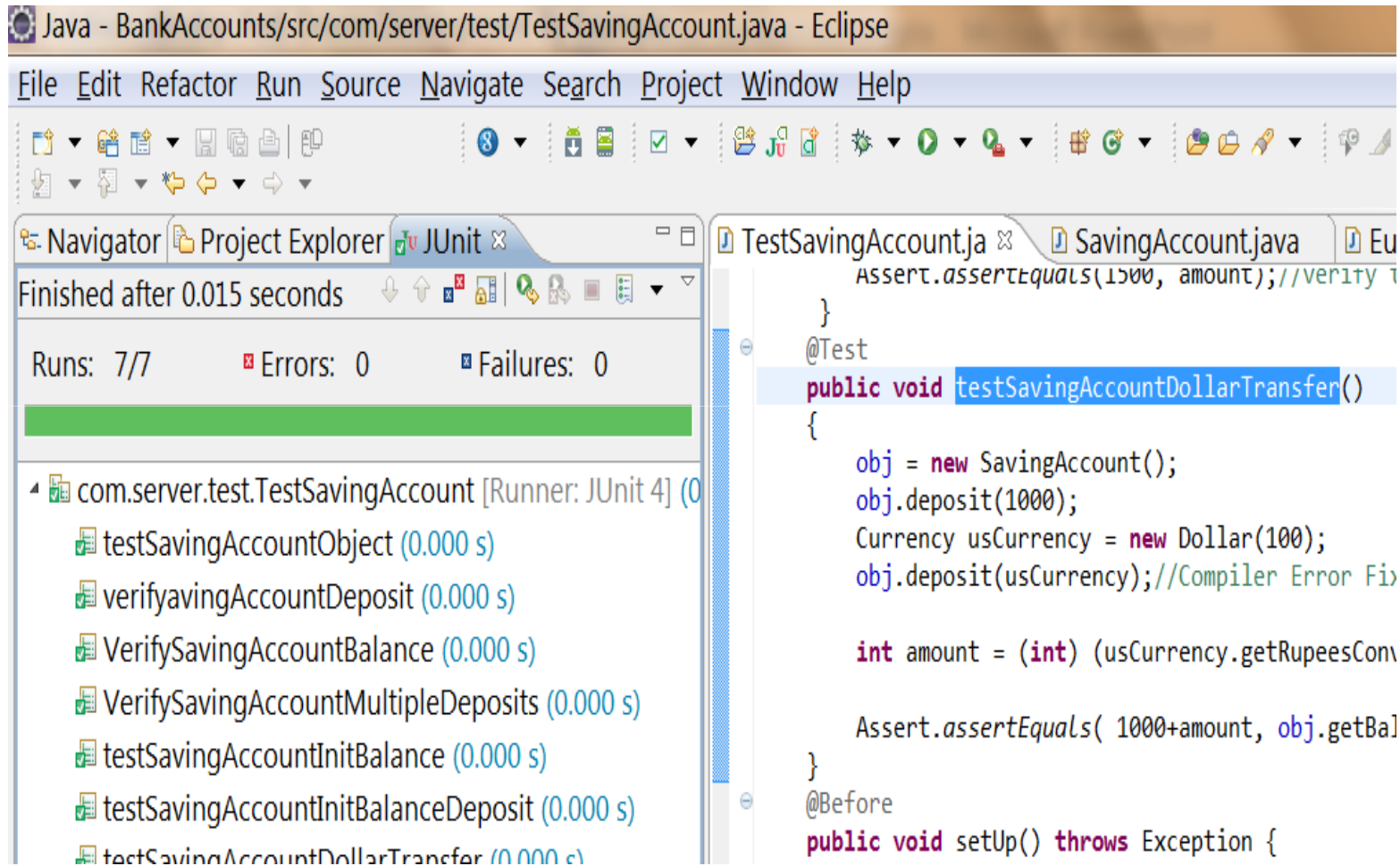
    @Test
    public void testSavingAccountDollarTransfer() {
        SavingAccount obj = new SavingAccount();
        obj.deposit(1000);
        Currency usCurrency = new Dollar(100);
        obj.deposit(usCurrency); //Compiler Error Fix

        int amount = (int) (usCurrency.getRupeesCorrespondingValue());

        Assert.assertEquals(1000+amount, obj.getBalance());
    }

    @Before
    public void setUp() {
        // ...
    }
}
```


Run all the Tests



Next User story

SavingAccount – withdraw amount from account balance and verify with getBalance.

TestCase for the withdraw

In the test class add another test function
TestSavingAccountWithdraw

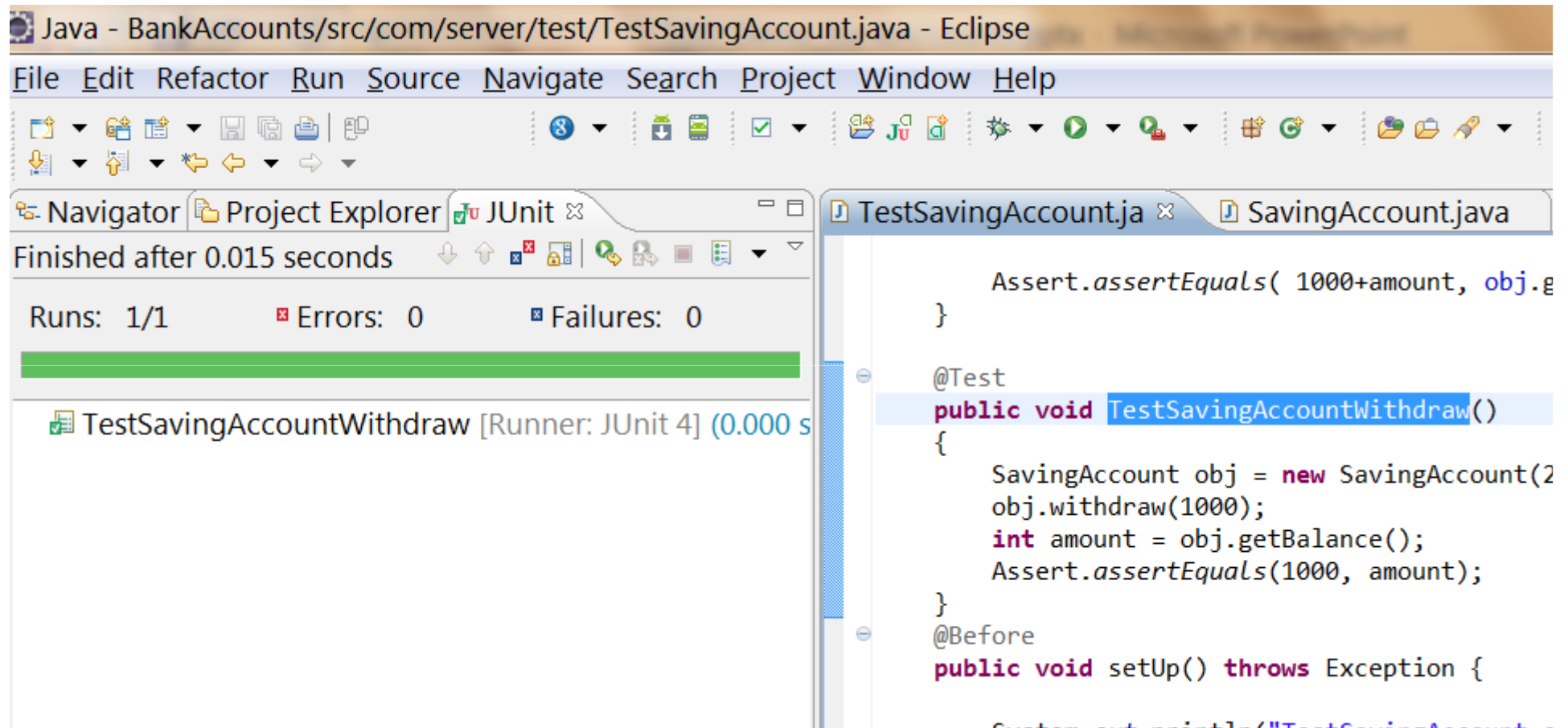
TestSavingAccountWithdraw

```
@Test
public void TestSavingAccountWithdraw()
{
    SavingAccount obj = new SavingAccount(2000);
    obj.withdraw(1000); //Compiler Error..Fix It!
    int amount = obj.getBalance();
    Assert.assertEquals(1000, amount);
}
```

SavingAccount - withdraw

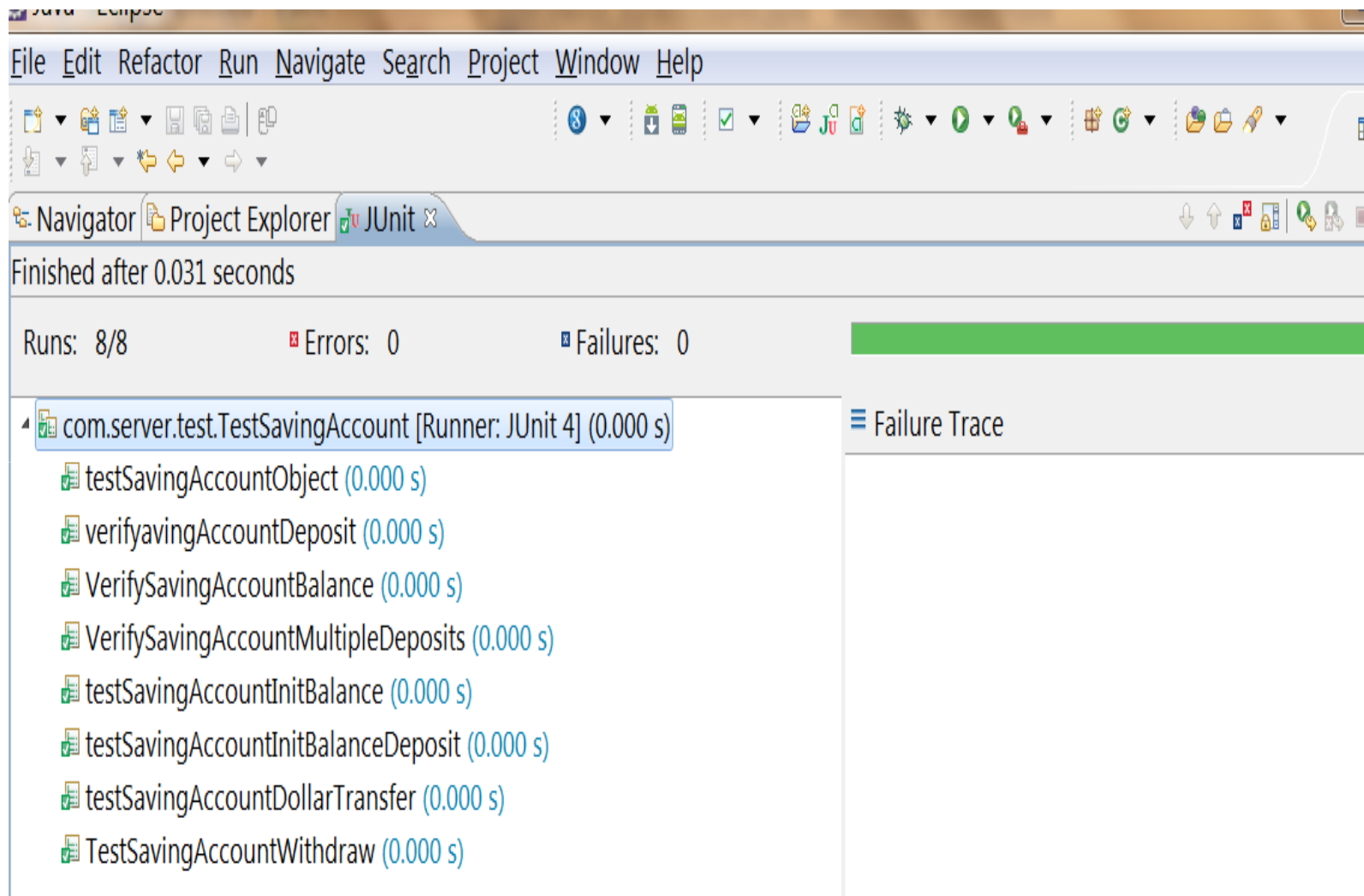
```
public void withdraw(int amount)
{
    //obvious implementation
    balance-= amount;
}
```

Run the Test



Cheers! We implemented successful function right at first..

Run all the tests ...



Huryo!!! All the tests have been successful..

What if

- If I invoke withdraw after deposit, I must get the cumulative balance added by deposit and subtracted by withdraw..
- Add another test function
TestSavingAccountDepositAndWithdraw in
Test class.

TestSavingAccountDepositAndWithdraw

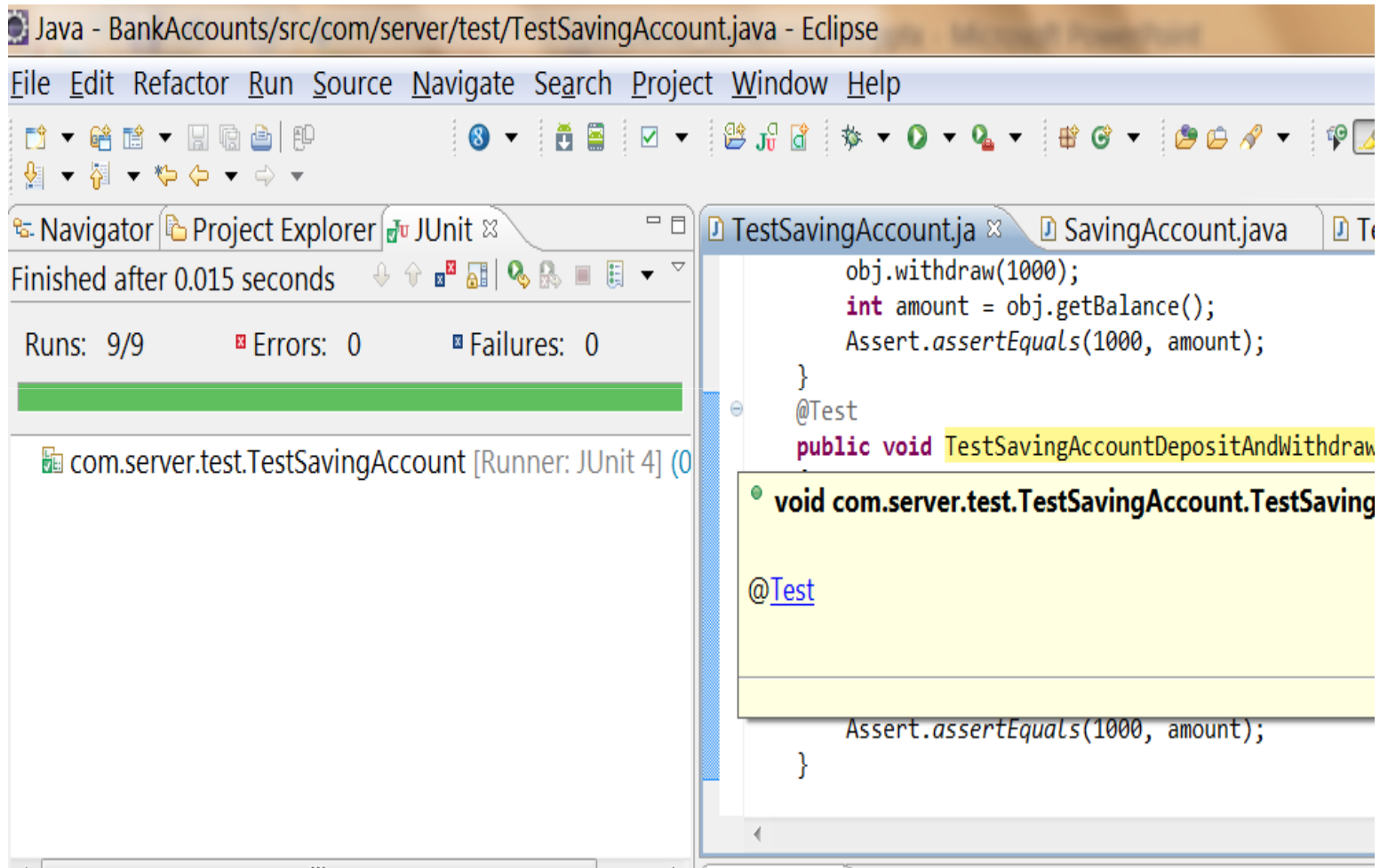
```
@Test
public void TestSavingAccountDepositAndWithdraw()
{
    obj.deposit(2000);

    obj.withdraw(2000);

    int amount = obj.getBalance();

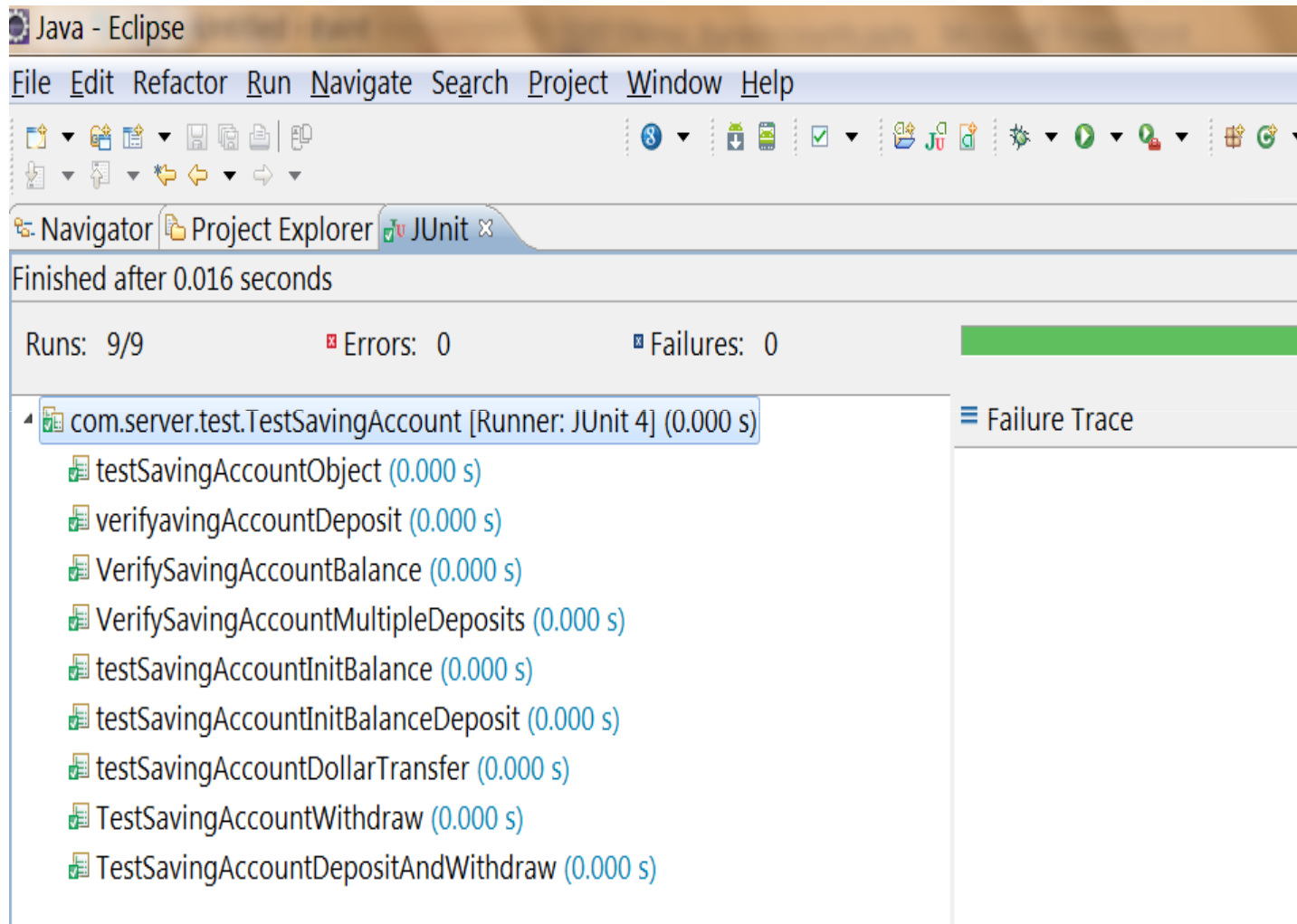
    Assert.AreEqual(1000, amount);
}
```

Run the Test



Cheers! This time also we were successful..

Run all the tests..



What If

- If I try to withdraw more amount than the balance amount, I should get error `WithdrawAmountMoreThanBalance` and balance intact..
- Add another test function `TestSavingAccountWithdrawMoreAmount` in `TestSavingAccount` class.
- Define new class `WithdrawAmountMoreThanBalance`.

TestSavingAccountWithdrawMore_ Amount

```
@Test(expected=com.server.bank.WithDrawAmountMoreThanBalance.class)
public void TestSavingAccountWithdrawMoreAmount()
{
    SavingAccount obj = new SavingAccount();

    obj.deposit(1000);

    obj.withdraw(3000);

    int amount = obj.getBalance();

    Assert.assertEquals(2000, amount);
}
```

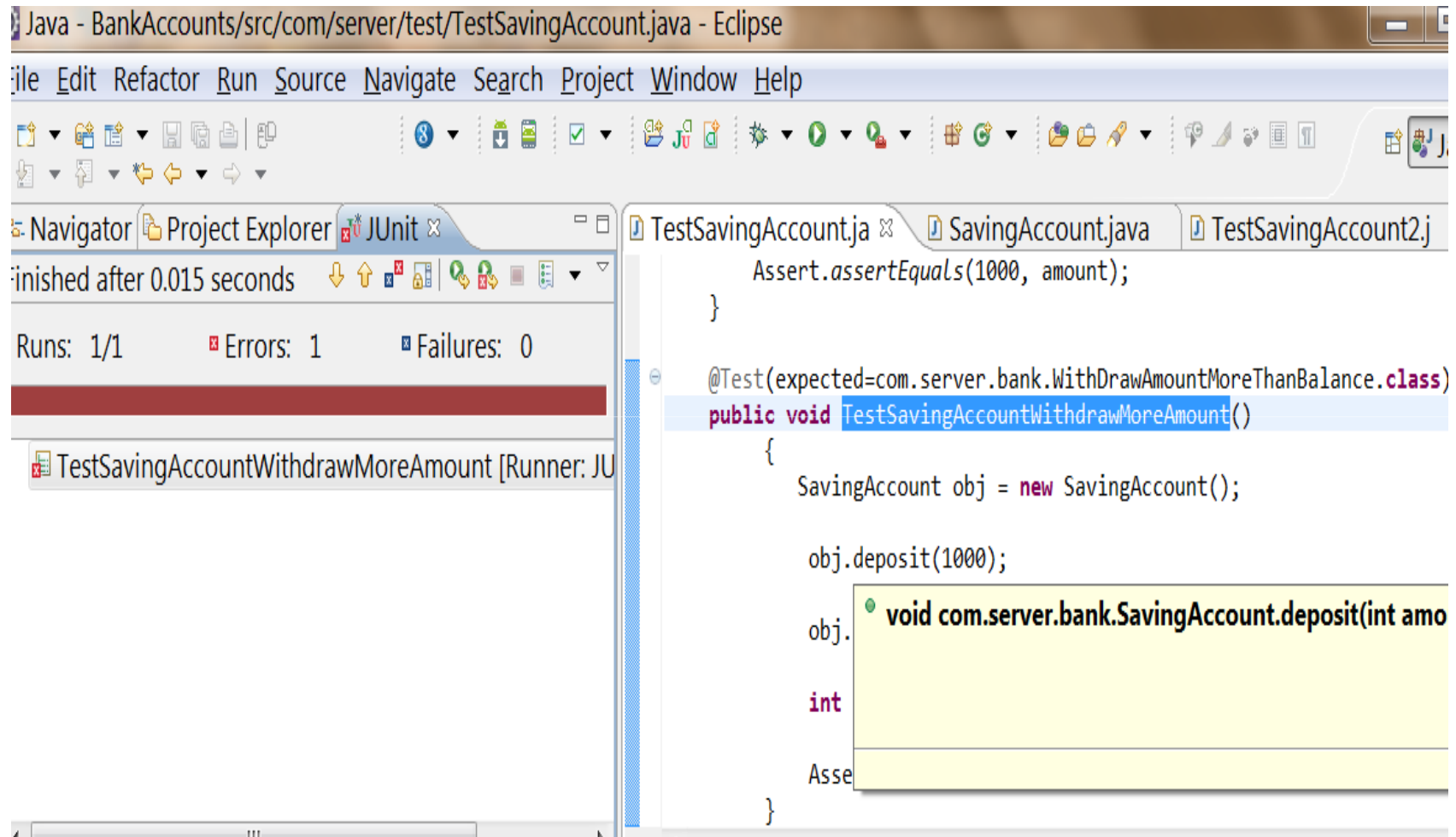
The Exception class

```
package com.server.bank;
```

```
public class WithDrawAmountMoreThanBalance  
    extends RuntimeException {
```

```
    public WithDrawAmountMoreThanBalance(String error)  
    {  
        super(error);  
    }  
  
}
```

Run the Test..Let it fail!



The test failed..expected exception not thrown..

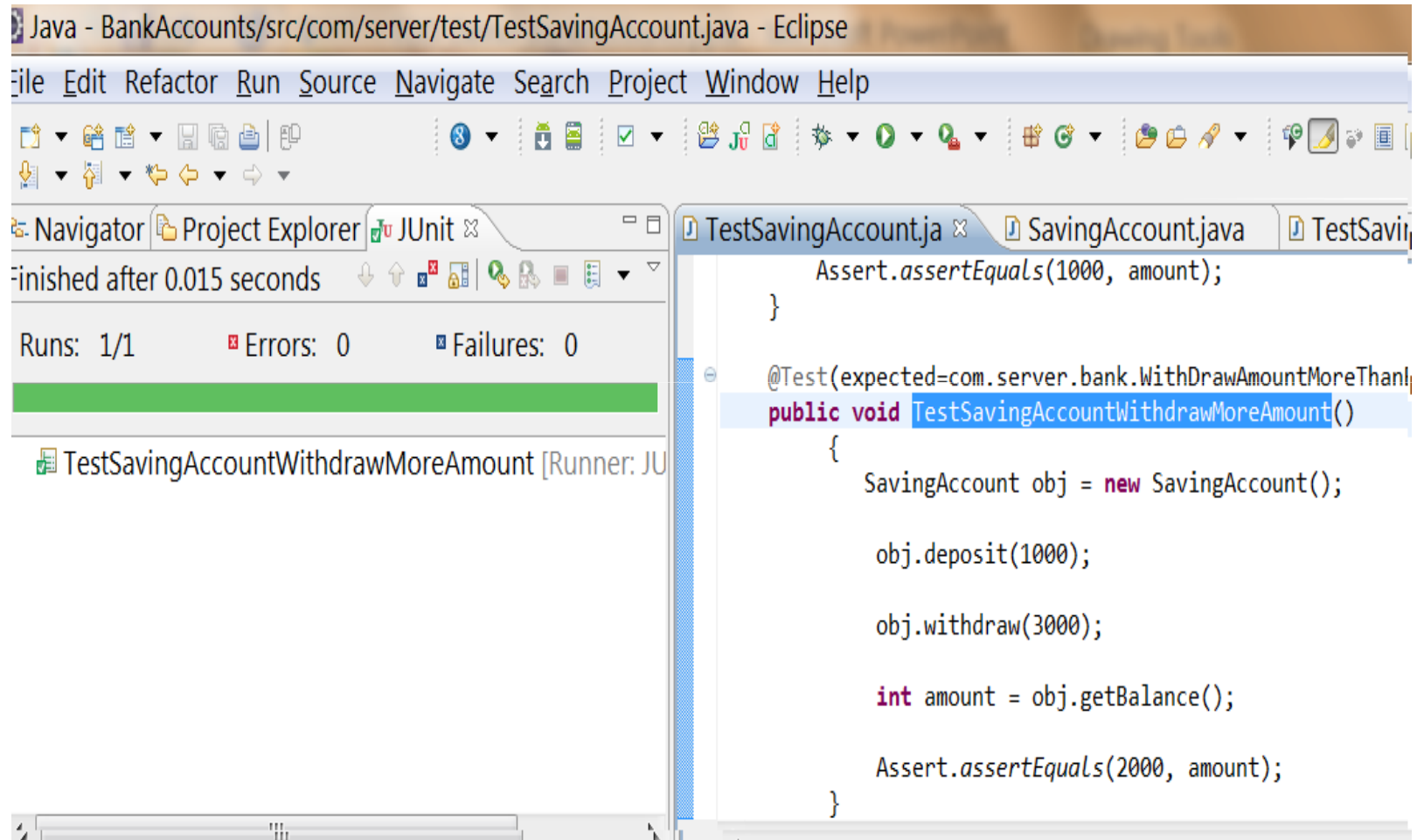
Modify withdraw

```
public void withdraw(int amount)
{

    if(amount > balance )
    {
        throw new WithDrawAmountMoreThanBalance("More amount
withdrawn");
    }

    //obvious implementation
    balance-= amount;
}
```


Run The Test



Run All the Tests

The screenshot shows an IDE's JUnit test runner interface. At the top, there are tabs for 'Navigator', 'Project Explorer', and 'JUnit'. The 'JUnit' tab is active, displaying the text 'finished after 0.016 seconds'. Below this, a summary bar shows 'Runs: 10/10', 'Errors: 0', and 'Failures: 0', accompanied by a green progress bar. The main area lists the test results for 'com.server.test.TestSavingAccount [Runner: JUnit 4] (0.000 s)'. Each test method is preceded by a green checkmark icon and followed by its duration in seconds. To the right of the test list is a 'Failure Trace' panel, which is currently empty.

Navigator | Project Explorer | JUnit

finished after 0.016 seconds

Runs: 10/10 Errors: 0 Failures: 0

com.server.test.TestSavingAccount [Runner: JUnit 4] (0.000 s)

- testSavingAccountObject (0.000 s)
- verifyavingAccountDeposit (0.000 s)
- VerifySavingAccountBalance (0.000 s)
- VerifySavingAccountMultipleDeposits (0.000 s)
- testSavingAccountInitBalance (0.000 s)
- testSavingAccountInitBalanceDeposit (0.000 s)
- testSavingAccountDollarTransfer (0.000 s)
- TestSavingAccountWithdraw (0.000 s)
- TestSavingAccountDepositAndWithdraw (0.000 s)
- TestSavingAccountWithdrawMoreAmount (0.000 s)

Failure Trace

How much we wrote ?

- Total 10 test functions to test on SavingAccount class.
- Evolved 5 functions on SavingAccount class.
 - SavingAccount default constructor
 - SavingAccount int parameterized constructor
 - deposit(int amount);
 - withdraw(int amount);
 - getBalance(void);

What if..

- What if I want to have initial balance for SavingAccount?
- Define another test function `testSavingAccountInitBalance` in the `TestSavingAccount` class.

The Test code

```
@Test
public void testSavingAccountInitBalance()
{
    SavingAccount obj = new SavingAccount(1000);//Compiler
Error..Fix it

    int amount = obj.getBalance();

    Assert .assertEquals(1000, amount);//verify the current balance
}
```

- Add the parameterized constructor in SavingAccount class
- You will be also required to add the definition of default constructor for the SavingAccount class !

SavingAccount with Initial Balance

```
package com.server.bank;

public class SavingAccount
{
    private int balance;
    //parameterized constructor
    public SavingAccount(int amount)
    {

    }

    //default constructor
    public SavingAccount()
    {

    }
    .....
}
```

Run the Test

The screenshot displays an IDE interface with two main panels. The left panel shows the test execution results, and the right panel shows the source code of the test class.

Test Execution Results (Left Panel):

- Finished after 0.015 seconds
- Runs: 1/1
- Errors: 0
- Failures: 1
- testSavingAccountInitBalance [Runner: JUnit 4] (0.000 s)
- Failure Trace: `AssertionFailedError: expected:<1000> but was:<0>`
- Test Name: `est.TestSavingAccount.testSavingAccountInitBalance(TestSavir`

Source Code (Right Panel):

```
SavingAccount obj = new SavingAccount();
obj.deposit(1000);
obj.deposit(2400);
Assert.assertEquals(3400, obj.getBalance());
}

@Test
public void testSavingAccountInitBalance()
{
    SavingAccount obj = new SavingAccount(1000);

    int amount = obj.getBalance();

    Assert.assertEquals(1000, amount); //verify the
}

@Test
public void testSavingAccountInitBalanceDeposit()
{
    SavingAccount obj = new SavingAccount(1000);
    obj.deposit(500);
    int amount = obj.getBalance();
}
```

Problems Panel (Bottom):

- 0 errors, 3 warnings, 0 others
- Description

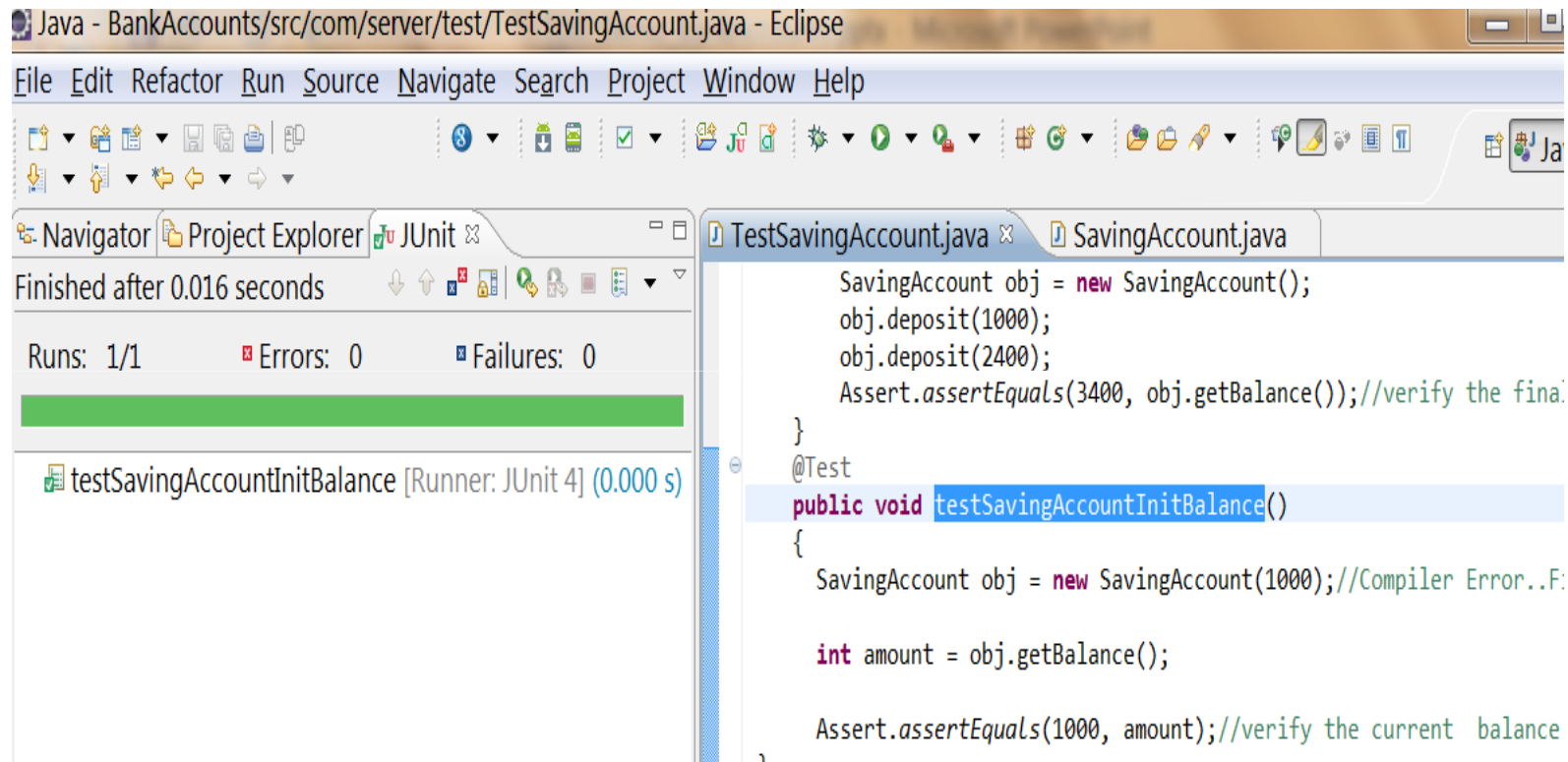
Fake or Actual ?

- What is the obvious implementation to make this test success ?

```
public SavingAccount(int amount)
{
    this.balance = amount;
}
```

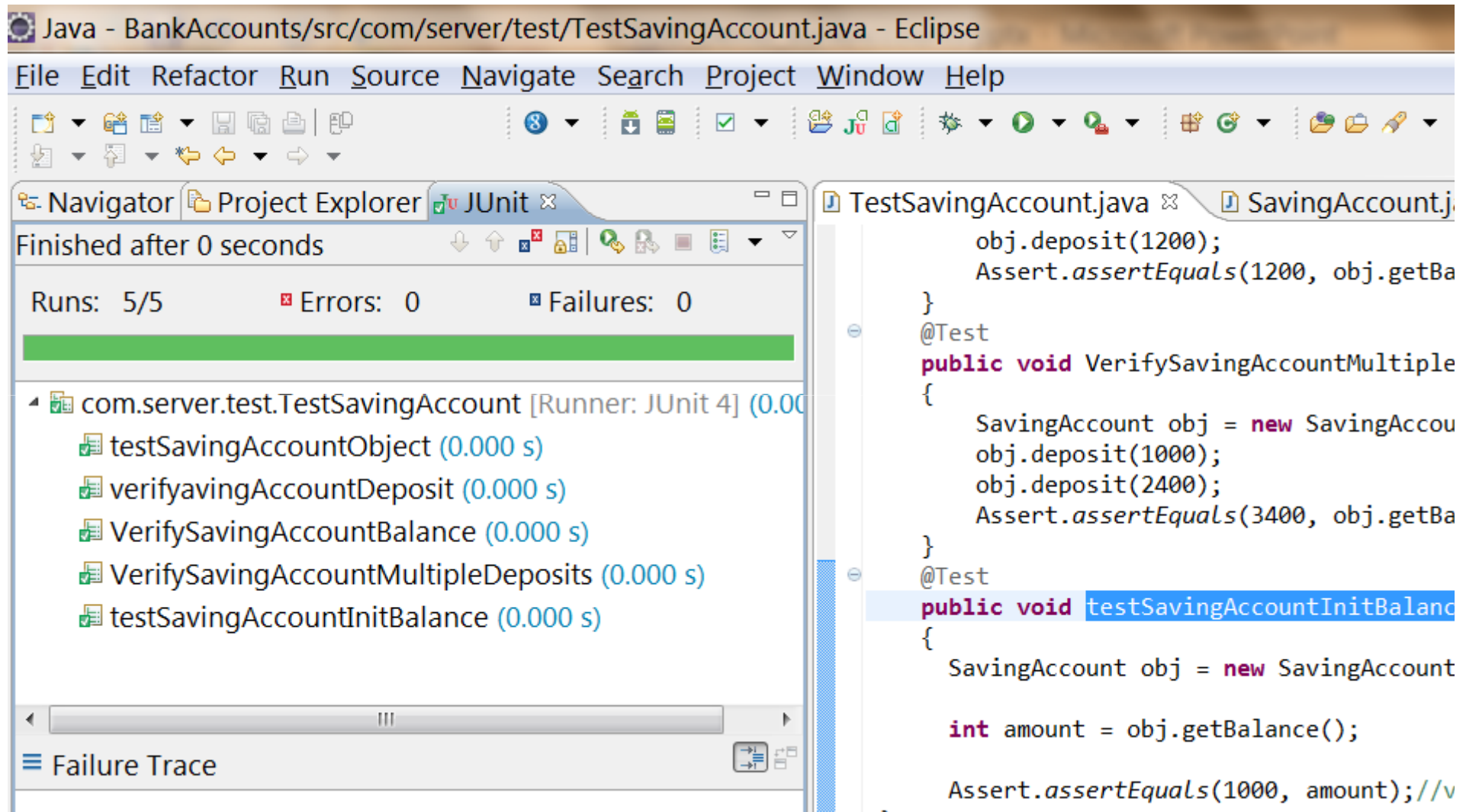
We know the obvious code..Not necessary to fake it here..

Run the Test



The obvious way to Success!

Run all the Tests..



The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for File, Edit, Refactor, Run, Source, Navigate, Search, Project, Window, and Help. The left sidebar contains the Navigator, Project Explorer, and JUnit views. The JUnit view shows a successful test run for 'com.server.test.TestSavingAccount' with 5 runs, 0 errors, and 0 failures. The test methods listed are: testSavingAccountObject, verifyavingAccountDeposit, VerifySavingAccountBalance, VerifySavingAccountMultipleDeposits, and testSavingAccountInitBalance. The right pane displays the source code for 'TestSavingAccount.java' and 'SavingAccount.java'. The code for 'TestSavingAccount.java' includes a test method 'testSavingAccountInitBalance' which is currently selected and highlighted in blue. The code for 'SavingAccount.java' shows the 'deposit' and 'getBalance' methods.

Java - BankAccounts/src/com/server/test/TestSavingAccount.java - Eclipse

File Edit Refactor Run Source Navigate Search Project Window Help

Navigator Project Explorer JUnit

Finished after 0 seconds

Runs: 5/5 Errors: 0 Failures: 0

com.server.test.TestSavingAccount [Runner: JUnit 4] (0.00 s)

- testSavingAccountObject (0.000 s)
- verifyavingAccountDeposit (0.000 s)
- VerifySavingAccountBalance (0.000 s)
- VerifySavingAccountMultipleDeposits (0.000 s)
- testSavingAccountInitBalance (0.000 s)

Failure Trace

TestSavingAccount.java SavingAccount.java

```
obj.deposit(1200);
Assert.assertEquals(1200, obj.getBa
}
@Test
public void VerifySavingAccountMultiple
{
    SavingAccount obj = new SavingAccou
    obj.deposit(1000);
    obj.deposit(2400);
    Assert.assertEquals(3400, obj.getBa
}
@Test
public void testSavingAccountInitBalanc
{
    SavingAccount obj = new SavingAccount
    int amount = obj.getBalance();

    Assert.assertEquals(1000, amount);//v
```

The savingAccount object with initialBalance succeeded..

How much to fake ?

- When you are not sure about the actual code.
- The code that depends on is NOT available. Be E.g.. Database or service to be defined by another team.
- Faking helps to test in isolation, without having the actual implementation.
- **Remove the fake implementation in production code.**

Mock for dependents

- If dependent classes/service are not available during testing, they can be mocked to present the required behavior.
- Mocking frameworks jmock, Mockito can be used to represent the objects.
- The behavior can be recorded or played back.
- This is called as stub or proxy depending on which way you are simulating.

Test with Initial balance and Deposit

- Add another test function in TestSavingAccount class 'testSavingAccountInitBalanceDeposit' and verify result with getBalance on SavingAccount object..
- **@Test**
- **public void testSavingAccountInitBalanceDeposit()**
 {
 SavingAccount obj = **new SavingAccount(1000);**
 obj.deposit(500);
 int amount = obj.getBalance();
 Assert *.assertEquals(1500, amount);*//verify the current balance
 }
- Rebuild the test and run the tests..

Test result..

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Refactor, Run, Source, Navigate, Search, Project, Window, and Help. Below the menu is a toolbar with various icons. The left sidebar contains the Navigator, Project Explorer, and JUnit views. The JUnit view shows a successful test run for 'testSavingAccountInitBalanceDeposit' with the following statistics: Runs: 1/1, Errors: 0, Failures: 0. The test was finished after 0.016 seconds. The right pane displays the source code for 'TestSavingAccount.java' and 'SavingAccount.java'. The code for 'TestSavingAccount.java' includes a test method 'testSavingAccountInitBalanceDeposit' that creates a 'SavingAccount' object, deposits 500, and asserts that the balance is 1500. The code for 'SavingAccount.java' includes a 'getBalance()' method.

Java - BankAccounts/src/com/server/test/TestSavingAccount.java - Eclipse

File Edit Refactor Run Source Navigate Search Project Window Help

Navigator Project Explorer JUnit

Finished after 0.016 seconds

Runs: 1/1 Errors: 0 Failures: 0

testSavingAccountInitBalanceDeposit [Runner: JUnit 4] (C)

Failure Trace

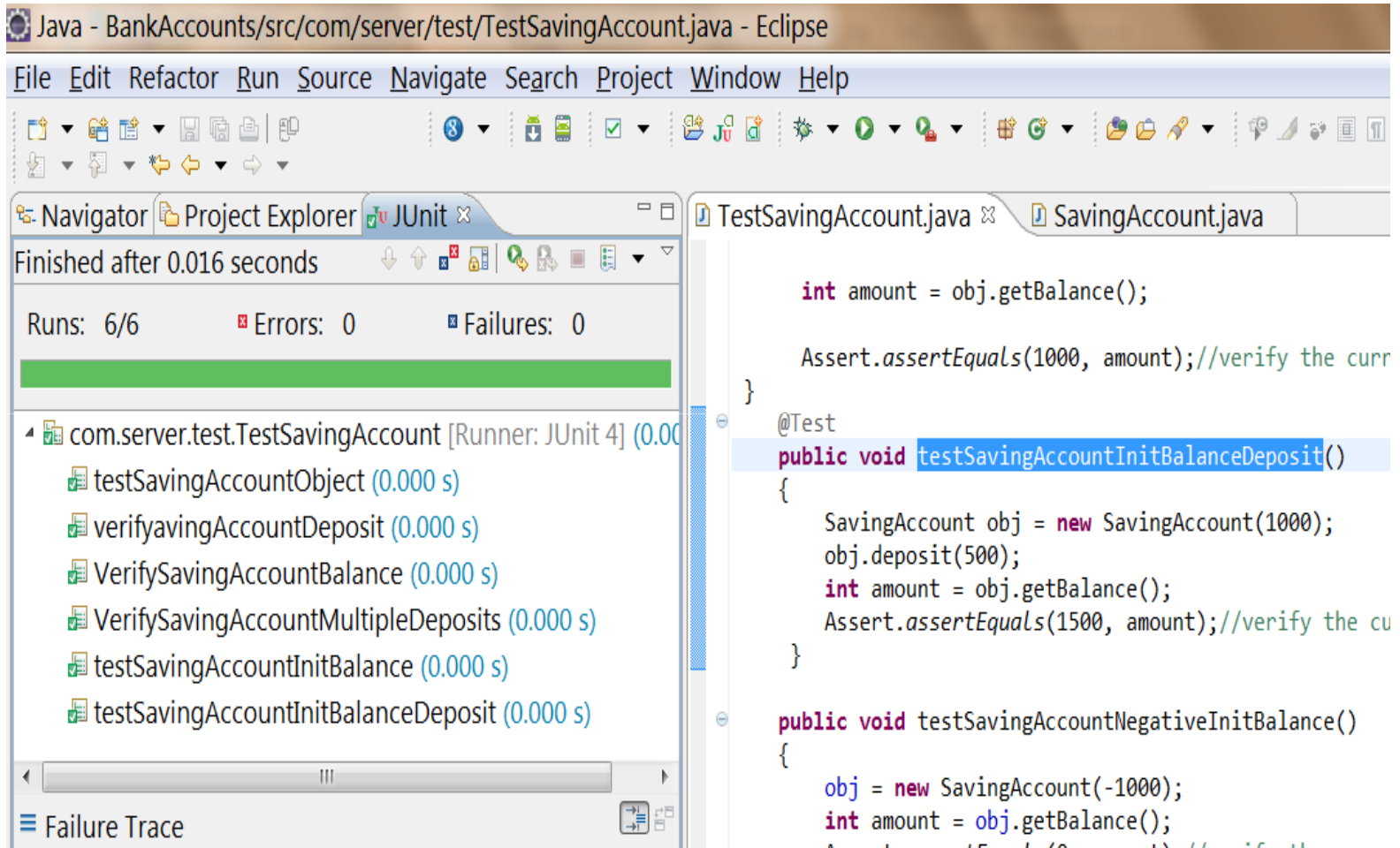
```
int amount = obj.getBalance();

Assert.assertEquals(1000, amount);//verify the
}

@Test
public void testSavingAccountInitBalanceDeposit()
{
    SavingAccount obj = new SavingAccount(1000);
    obj.deposit(500);
    int amount = obj.getBalance();
    Assert.assertEquals(1500, amount);//verify th
}

public void testSavingAccountNegativeInitBalance(
{
    obj = new SavingAccount(-1000);
    int amount = obj.getBalance();
    Assert.assertEquals(1500, amount);//verify th
}
```

Run all the tests again..



The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Refactor, Run, Source, Navigate, Search, Project, Window, and Help. Below the menu is a toolbar with various icons. The left sidebar contains the Navigator, Project Explorer, and JUnit views. The JUnit view shows the results of a test run: "Finished after 0.016 seconds", "Runs: 6/6", "Errors: 0", and "Failures: 0". A list of test methods is displayed, all with green checkmarks indicating success:

- testSavingAccountObject (0.000 s)
- verifyavingAccountDeposit (0.000 s)
- VerifySavingAccountBalance (0.000 s)
- VerifySavingAccountMultipleDeposits (0.000 s)
- testSavingAccountInitBalance (0.000 s)
- testSavingAccountInitBalanceDeposit (0.000 s)

The right sidebar shows the code editor with the `TestSavingAccount.java` file open. The code includes the following methods:

```
int amount = obj.getBalance();

Assert.assertEquals(1000, amount);//verify the curr
}

@Test
public void testSavingAccountInitBalanceDeposit()
{
    SavingAccount obj = new SavingAccount(1000);
    obj.deposit(500);
    int amount = obj.getBalance();
    Assert.assertEquals(1500, amount);//verify the cu
}

public void testSavingAccountNegativeInitBalance()
{
    obj = new SavingAccount(-1000);
    int amount = obj.getBalance();
    Assert.assertEquals(0, amount);//verify the curr
```

The savingAccount object with all tests succeeded..

Relax!

- Now we have successfully implemented **deposit** and **getBalance** functionality in SavingAccount and also tested it..
- We have written couple of test methods and accordingly added required minimum code into the domain class SavingAccount..It is the *TDD Approach!*

Sharing common object across the tests..

- Can we share the single object of SavingAccount to all the test methods instead of creating a new one every time ?
- Define a private Saving Account object variable in Test class ,initialize in the TestClass constructor and use the same object across all the test methods..
- Rebuild the Test and run all the tests again..

Run all the tests again....

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Refactor, Run, Source, Navigate, Search, Project, Window, and Help. Below the menu is a toolbar with various icons. The left sidebar contains the Navigator, Project Explorer, and JUnit views. The JUnit view shows the test results for 'com.server.test.TestSavingAccount' [Runner: JUnit 4] (0.000 s). The results are as follows:

Test Name	Duration	Status
testSavingAccountObject	0.000 s	Passed
verifyavingAccountDeposit	0.000 s	Passed
VerifySavingAccountBalance	0.000 s	Passed
VerifySavingAccountMultipleDeposits	0.000 s	Passed
testSavingAccountInitBalance	0.000 s	Passed
testSavingAccountInitBalanceDeposit	0.000 s	Passed

The right sidebar shows the source code for 'TestSavingAccount.java' and 'SavingAccount.java'. The 'TestSavingAccount.java' file is open, showing the following code:

```
Assert.assertEquals(3400, obj.getBalance())
}
@Test
public void testSavingAccountInitBalance()
{
    SavingAccount obj = new SavingAccount(1000);

    int amount = obj.getBalance();

    Assert.assertEquals(1000, amount); //verify th
}
@Test
public void testSavingAccountInitBalanceDeposit
{
    SavingAccount obj = new SavingAccount(1000)
    obj.deposit(500);
```

TDD Mantra for Final Success

- Fake the result in code->Test the code->Implement the obvious code ->Test it again->Fail ->Modify->Test->Success->Modify->Test->Failure->Update->Test Final->Success...
- Learn from our own Success and Failures!!!

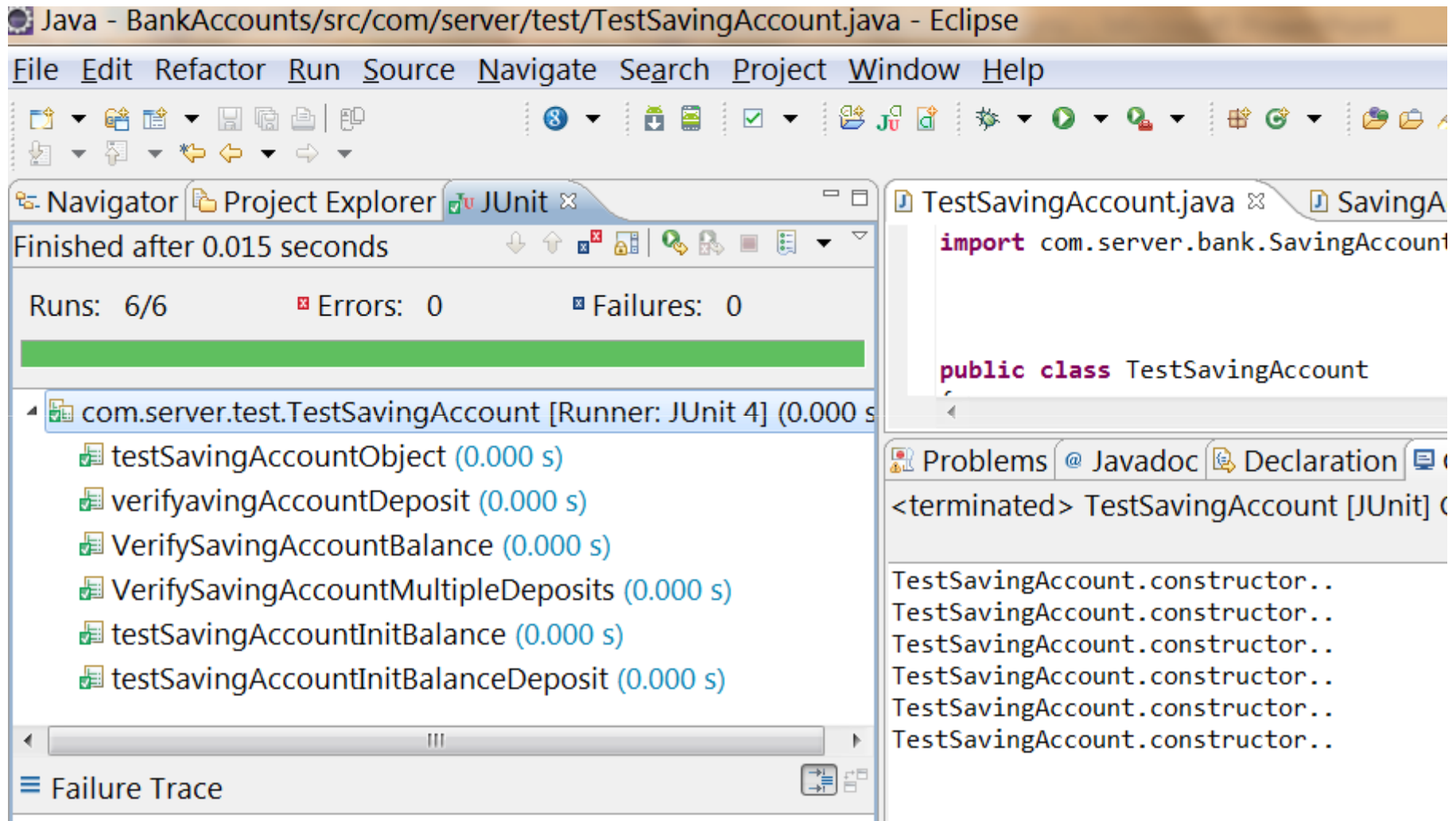
Observation

- When we run all the tests, the deposit amount passed in every test method was different..
- But still our all the tests have verified successfully their own balances even though there was a common shared SavingAccount object across all the test methods..
- How this is possible ?

Analyze..

- In TestSavingAccount class add the default constructor with console print statement and run all the tests again and watch the console.
- **public TestSavingAccount()**
 {
 System.out.println("TestSavingAccount.constructor..");
 }

Console view



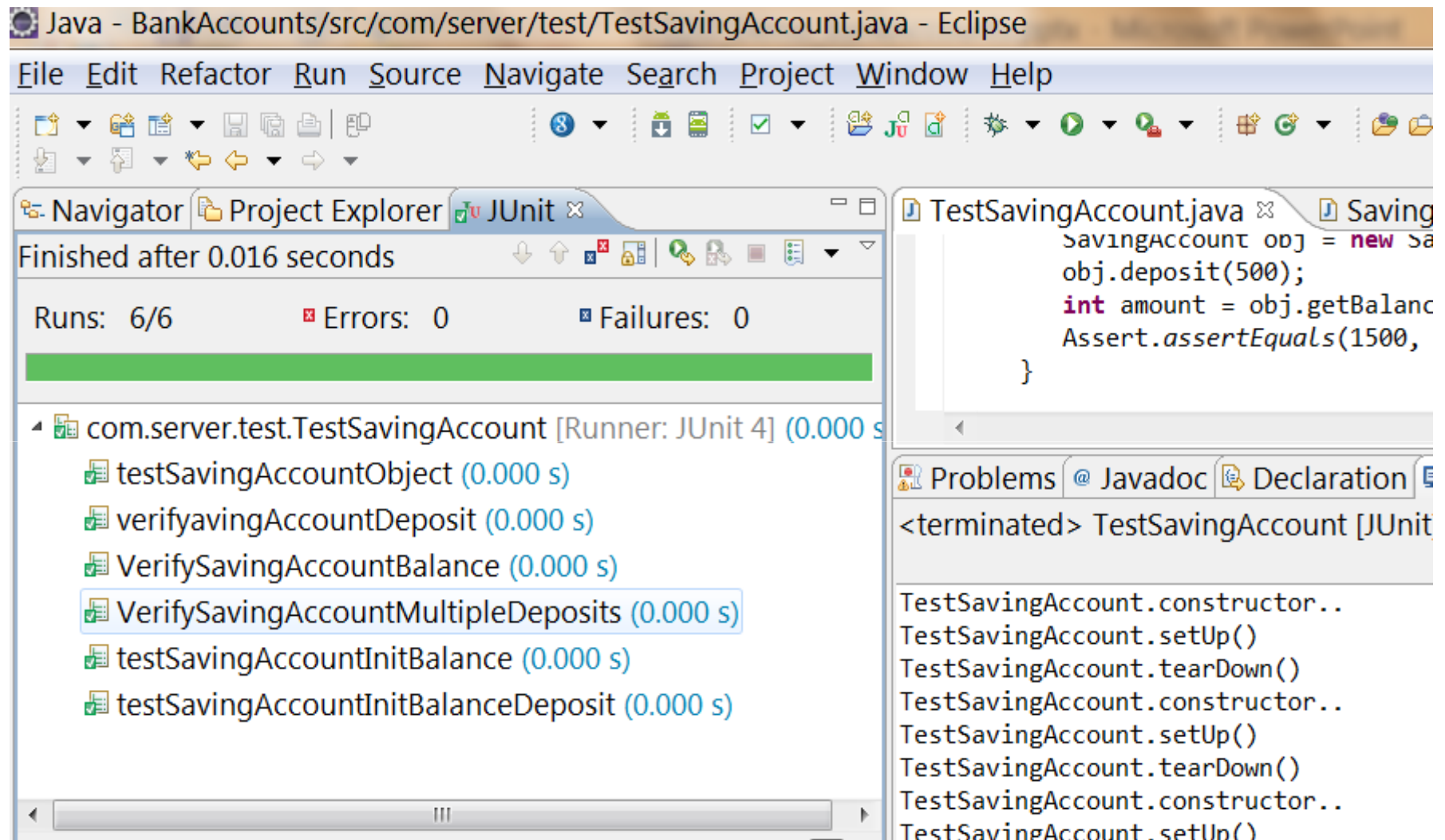
Test objects in JUnit Framework

- For every test case methods the framework has created a new instance of the test class..
- This is the default behavior..

Common shared methods for every test ?

```
public class TestSavingAccount {  
    private SavingAccount obj;  
  
    @Before  
    public void InitObjects() throws Exception {  
        System.out.println("TestAccount.setUp()");  
        obj = new SavingAccount();  
    }  
  
    @After  
    public void closeObjects() {  
        System.out.println("TestAccount.tearDown()");  
        obj = null;  
    }  
}
```


Run All the tests..



All the tests in testSavingAccount succeeded..

Add multi-currency support to withdraw

What if the user wants to withdraw US Dollars
or Euro amounts from this Indian
SavingAccount ?

Retrospect..

- Duplicate code across the functions of SavingAccount class.
- What if we change the strategy to deal with negative values?
- Are we going to modify all the three functions again in the SavingAccount class?

Refactor and go GREEN!..

- Extract the functions out of existing duplicate code.
- Remove duplications without changing the external behaviors of functions.. All the tests must succeed..
- Run all the tests again..
- It must be GREEN bar always..

Next User Story

- As a user I should be able to execute following operations on the LoanAccount
- deposit- the amount deposited should get subtracted from current balance.
- getBalance – the amount returned reflects the current LoanAccount balance.
- Withdraw - the amount to withdraw should get subtracted from current balance.

Evolve the logic for LoanAccount

- Define another Test class TestLoanAccount with test functions one by one and get evolved the functions for LoanAccount to succeed the TestLoanAccount test functions same way as earlier done for SavingAccount...
 - Write the test...
 - First time fake the functionality and run the test and observe GREEN Bar!
 - Write the obvious implementation code to replace fake code and run the test and observe GREEN Bar!

New Project_Refactored

- Create new project BankAccounts_Refactored to define new versions of Loan and Saving Account functionalities.
- Add the SavingAccount and LoanAccount classes along with their corresponding test classes here in new project.

Account restructuring

Now We have two Account classes

- SavingAccount
- LoanAccount
- We have duplicate code scattered across both the above classes.
- To minimize the this code duplication we need to define a class hierarchy through which common code can be shared.

Evolve Account

- Extract/Design the class Account as base class for both
- To standardize the common functions we push the deposit and withdraw functions as abstract in base class.
- Push the getBalance implementation in base class Account.
- Push the balance variable as protected in base class Account.
- Push the ValidateAccountBalance function as protected to base class Account.

Class Account

```
public abstract class Account
{
    protected int balance;
    public abstract void deposit(int amount);
    public abstract void withdraw(int amount);

    public int getBalance()
    {
        ....
    }
    protected boolean ValidateAccountBalance()
    {
        .....
    }
}
```

Account Functions

```
public int getBalance()
{
    //if (balance < 0)
    //    balance = 0;
    if (!ValidateAccountBalance())
        return 0;
    return balance;
}
protected boolean ValidateAccountBalance()
{
    if (balance < 0)
        return false;
    else return true;
}
```

Restructured SavingAccount

```
public class SavingAccount extends Account
{ //parameterized constructor
    public SavingAccount(int amount)
    { .... }
    //default constructor
    public SavingAccount()
    { .... }
    public override void deposit(int amount)
    { ..... }
    public override void withdraw(int amount)
    { ..... }
}
}
```

LoanAccount-deposit

```
public override void deposit(int amount)
{
    if (!ValidateAccountBalance())
        return;

    if (amount < 0)
        throw new NegativeDepositException();

    balance -= amount;
}
```

New Behavior

- In this application we are allowing the users to create the objects of `SavingAccount` and `LoanAccount` classes and calling their individual functions.(e.g. in `TestSavingAccount` and `TestLoanAccount` classes).
- We want to isolate the object creation and usage behaviors so that objects can be managed independent of usage and functions used independent of creation process.

New Project

- Define Java project Factory_BankApp as windows Forms application
- Add LoanAccount and SavingAccount classes from previous project with their dependencies except the test classes to this new project.

Factory to manage objects

- To isolate the creation and usage of objects we decide to use a factory class to manage the Account objects.
- To specify the type of object to be created by the factory we define 'enum AccountType ' to specify possible types.
- The factory will be responsible for creating the specific type of Account and sharing it to clients.

Test the Factory

- Add test class TestAccountFactory to test the and build/evolve the functionality of AccountFactory class.
- Add test function TestGetLoanAccountObject to test the LoanAccount type object returned from AccountFactory.

TestGetLoanAccountObject

```
public class TestAccountFactory
{
    @Test
    public void TestGetLoanAccountObject()
    {
        Account loanAccountObject=
            AccountFactory.getAccountObject(Loan);
        //Compiler error ..Fix it
        Assert.IsNotNull(loanAccountObject);
    }
}
```

AccountFactory class

- Define class AccountFactory with static function as 'getAccountObject(AccountType mode)'
- Fake the implementation right now with returning the NULL.

The factory and the enum

AccountType

```
public enum AccountType
{
    Saving,
    Loan
}

public class TestAccountFactory
{
}
}
```

The factory ::getAccountObject

```
Public Account  getAccountObject(AccountType mode)
{
    return NULL;
}
```

AccountFactory getObject

```
public static Account getObject(AccountType acType)
{
    Account obj = null;

    switch (acType)
    {
    case AccountType.Loan: obj = new LoanAccount(1000);
        break;
    case AccountType.Saving: obj = new SavingAccount();
        break;
    }
    return obj;
}
```

Added code to return corresponding Account type object

Test for SavingAccount object

- Add test function TestGetSavingAccountObject to test the SavingAccount type object returned from AccountFactory.

TestGetSavingAccountObject

```
@Test
public void TestGetSavingAccountObject()
{
    Account savingAccountObject =
        AccountFactory.getAccountObject(AccountType.Saving);

    Assert.IsNotNull(savingAccountObject);

    System.out.println(savingAccountObject.GetType().FullName);

}
```


LoanAccount deposit Test

```
@Test
public void TestLoanAccountDeposit()
{
    Account loanAccountObject =
    AccountFactory.getAccountObject(AccountType.Loan);

    loanAccountObject.deposit(500);
    int balAmount = loanAccountObject.getBalance();
    Assert.AreEqual(500, balAmount);
}
```

SavingAccount deposit test

```
@Test
public void TestSavingAccountDeposit()
{
    Account savingAccountObject =
    AccountFactory.getAccountObject(AccountType.Saving);

    savingAccountObject.deposit(500);
    int balAmount = savingAccountObject.getBalance();
    Assert.AreEqual(500, balAmount);
}
```

Kudos to TDD and refactoring !!