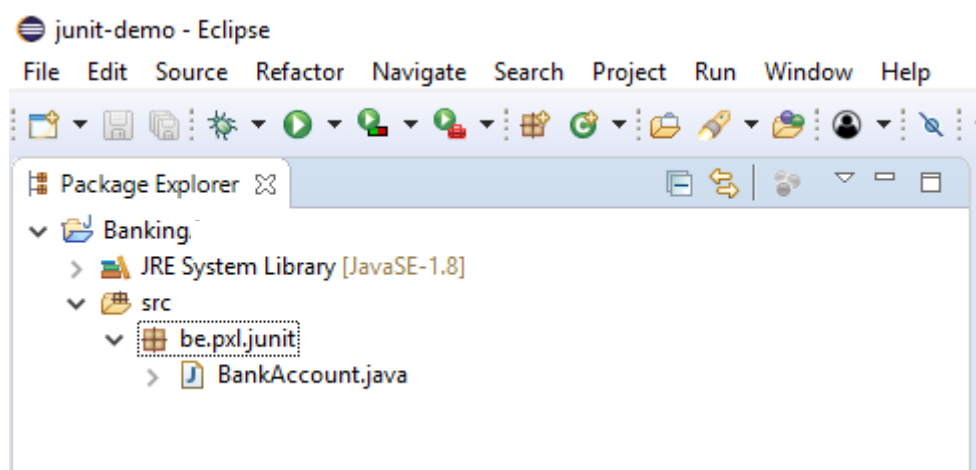


Introductie tot JUnit

Als we software ontwikkelen willen we graag kwaliteitsvolle software afleveren aan de klant. Eén van de technieken om dit te doen is je code voorzien van automatische unit testen. Op die manier kan je de testen, na een succesvolle build van je programma, iedere keer opnieuw uitvoeren en krijg je direct een signaal wanneer er fouten in je programma zijn geslopen.

We spreken over **unittesten** wanneer we de methoden van onze klassen gaan testen. **JUnit** is het framework dat we gebruiken om deze unit testen te schrijven.

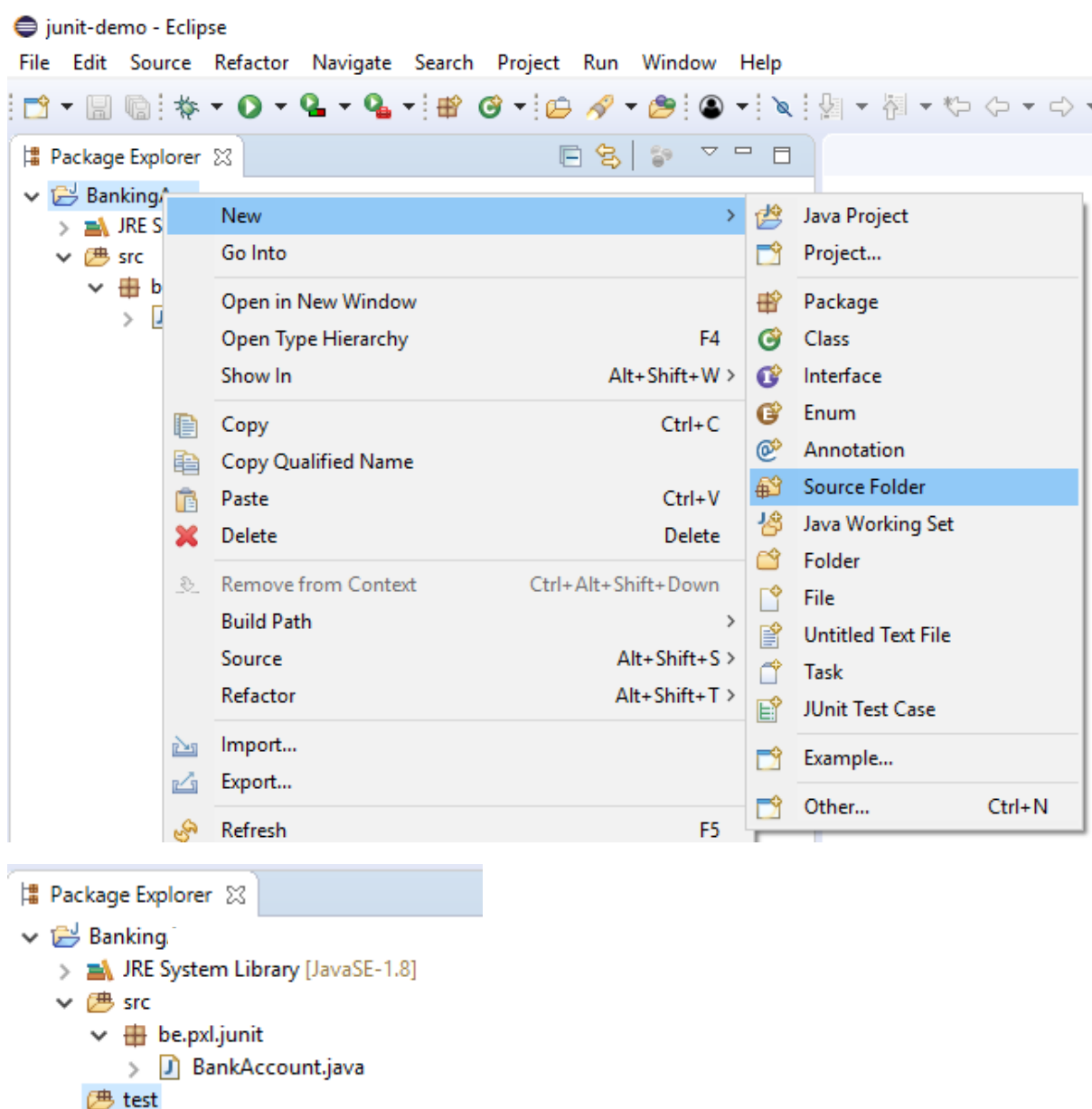
We starten met een nieuw project **Banking**. Maak hierin package `be.pxl.junit`. Voorzie hierin een klasse **BankAccount**. De klasse **BankAccount** heeft 3 member variabelen: `firstName (String)`, `lastName (String)` en `balance (double)`. Maak een constructor aan waar de 3 kenmerken worden geïnitialiseerd.



Vervolgens implementeren we 2 methoden: *deposit()* en *withdraw()*. Beide methoden hebben 1 parameter: `amount (double)`. Voor *deposit()* voeg je het gegeven bedrag toe aan de balance, bij *withdraw()* verminder je de balance met het gegeven bedrag.

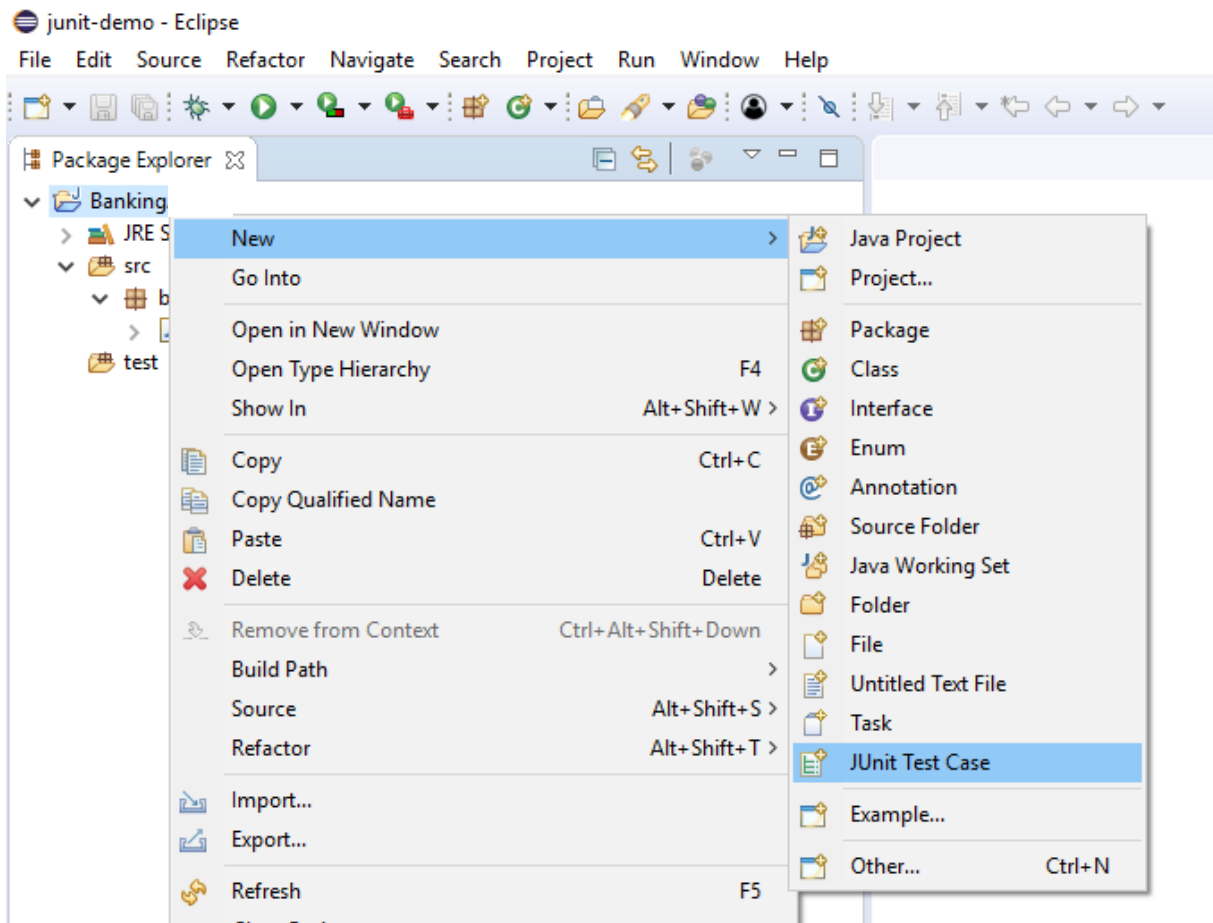
Voorzie de nodige getters en setters.

Nu gebruiken we JUnit om testen te schrijven voor onze klasse **BankAccount**. Om de JUnit library te kunnen gebruiken, moet je deze nog toevoegen aan het project. Maak nu eerst een source-folder “test” aan in je programma-structuur. Dit is om je project-code (in de “src” broncode folder) en je testen gescheiden te houden.



Voeg vervolgens een nieuwe **JUnit Test Case** toe aan je project. Kies voor de “test”-folder om je testklasse in op te slaan. De testklasse plaatsen we in dezelfde package als de klasse die we gaan testen (maar de testklasse zit in de “test” folder en de klasse die getest wordt zit in de “src” folder).

Wanneer eclipse merkt dat de JUnit library nog niet is toegevoegd, krijg je hiervoor ook nog een extra vraag om JUnit4 aan je buildpath toe te voegen.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

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:

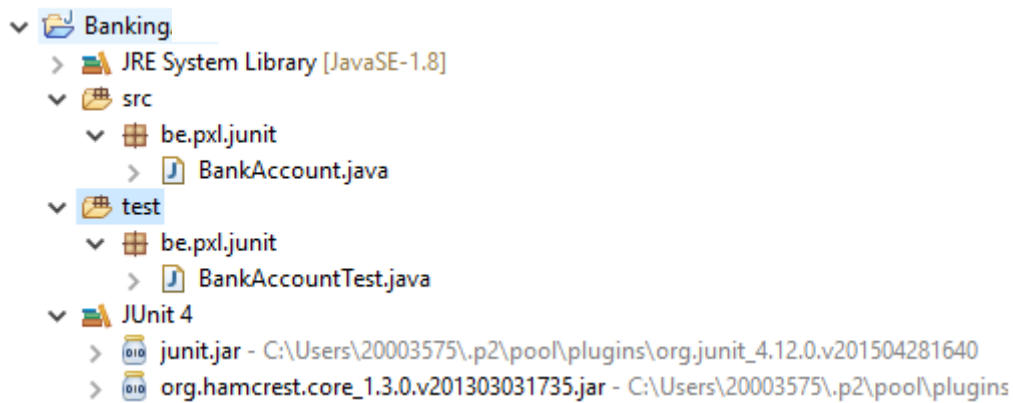
New JUnit Test Case

 JUnit 4 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 4 library to the build path

Na de voorgaande stappen krijg je dus het volgende resultaat:

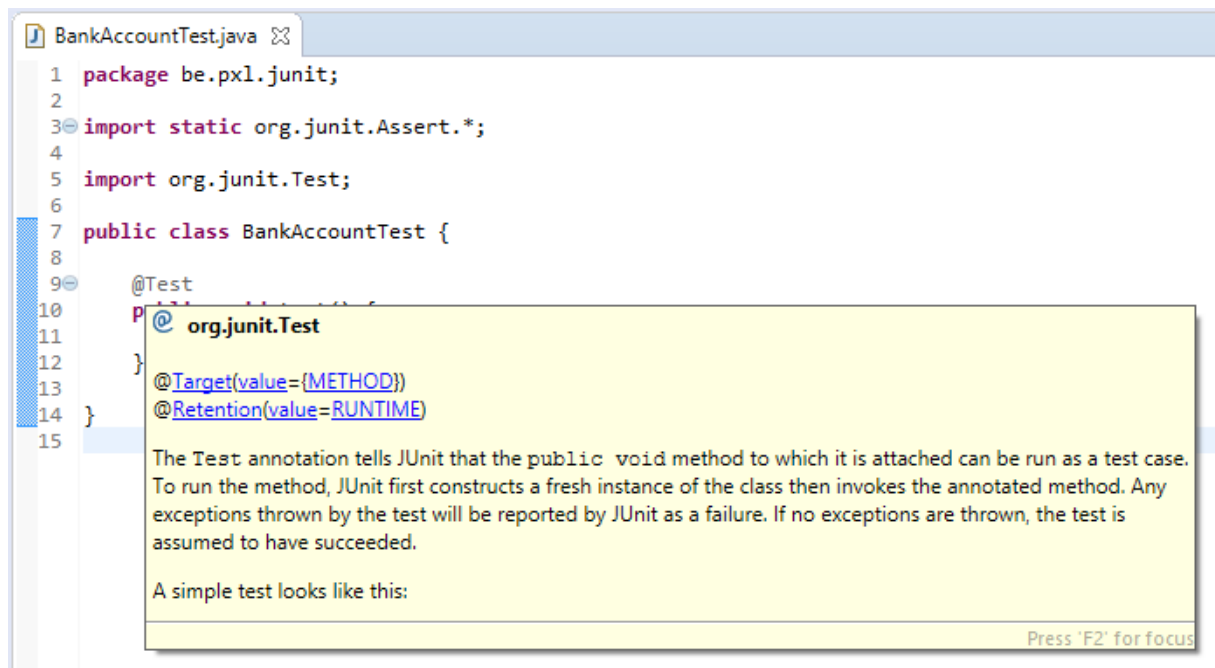


Open nu de klasse **BankAccountTest**.

The screenshot shows the code for `BankAccountTest.java` in an IDE. The code is as follows:

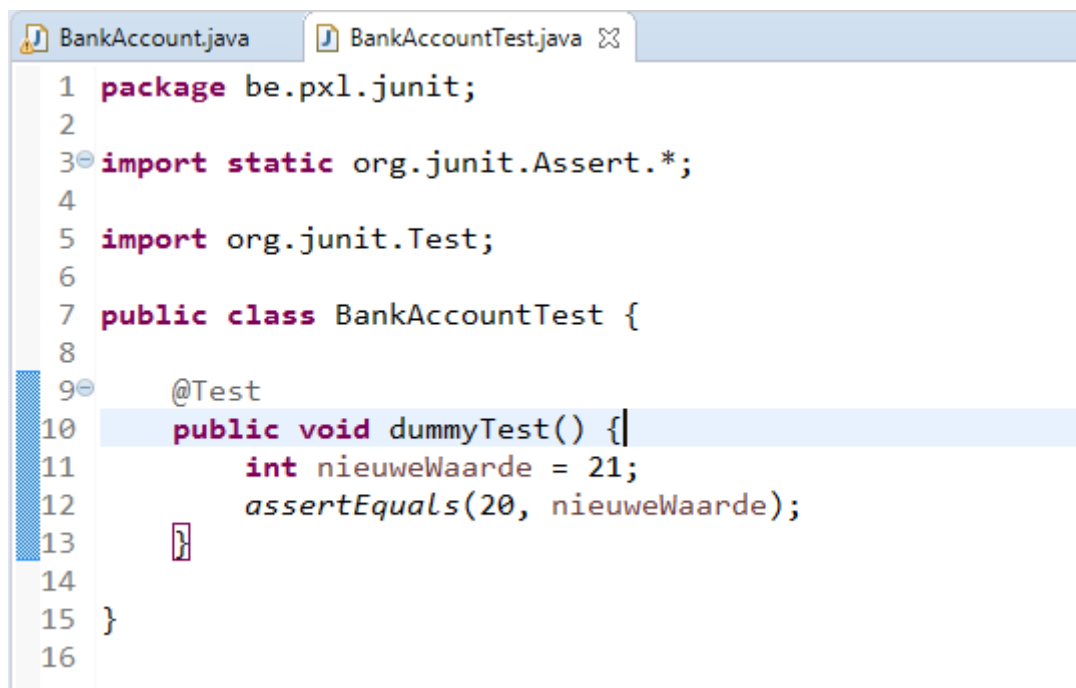
```
1 package be.pxl.junit;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class BankAccountTest {
8
9     @Test
10     public void test() {
11         fail("Not yet implemented");
12     }
13
14 }
```

Hierin vinden we reeds een `test()` methode terug. Iedere test die in de klasse wordt toegevoegd, wordt geannoteerd met **@Test**. Om deze annotatie te kunnen gebruiken wordt de klasse **org.junit.Test** geïmporteerd.

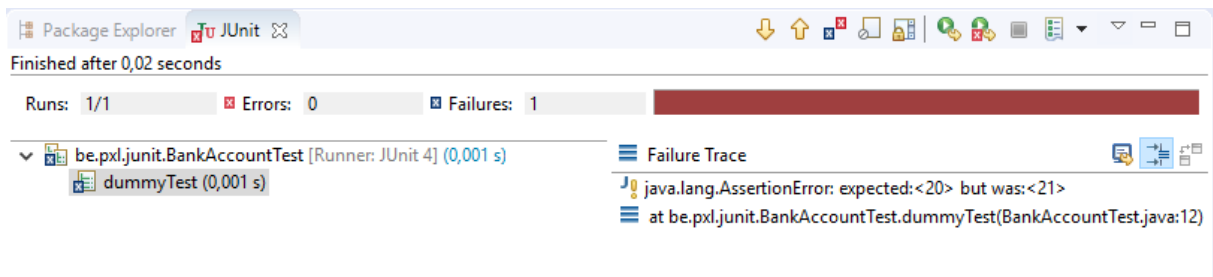


De klasse **org.junit.Assert** bevat een aantal statische methoden waarmee je beweringen kan schrijven. Als de bewering faalt, faalt je test. Bijvoorbeeld met *Assert.assertEquals(...)* kan je controleren of 2 waarden gelijk zijn. Meestal gebruik je de static methoden uit de Assert klasse via een static import, dit bevordert de leesbaarheid van je testen.

Pas nu de bestaande test aan zoals weergegeven in onderstaande afbeelding.



We testen de bewering dat *nieuweWaarde* gelijk is aan 20. Voer de test eens uit (via de rechtermuisknop krijg je een menu waarmee je de JUnit test kan uitvoeren: Run as > JUnit test).



In het JUnit tabblad zie je dat de test faalt en in de Failure Trace zie je dat de waarde 20 werd verwacht, maar dat *nieuweWaarde* 21 is.

Ken aan *nieuweWaarde* 20 toe en voer de test opnieuw uit!

Een overzicht van de statische methoden die je terugvindt in de klasse **org.junit.Assert**:

assertEquals()	Evalueert de gelijkheid van 2 waarden. De test slaagt als beide waarden gelijk (equal) zijn.
assertFalse()	Evaluatie van een booleaanse uitdrukking. De test slaagt indien de uitdrukking false is.
assertTrue()	Evaluatie van een booleaanse uitdrukking. De test slaagt indien de uitdrukking true is.
assertNotNull()	Vergelijkt een object referentie met null. De test slaagt indien de object referentie niet null is.
assertNull()	Vergelijkt een object referentie met null. De test slaagt indien de object referentie null is.
assertSame()	Vergelijkt het geheugenadres van 2 object referenties (door gebruik te maken van de ==-operator). De test slaagt indien beide object referenties naar hetzelfde object verwijzen.
fail()	Zorgt ervoor dat de huidige test zal falen. Wordt regelmatig gebruikt bij het testen van exception handling.

Nu gaan we de *deposit* methode van de **BankAccount** klasse testen. In onze test maken we eerst een **BankAccount**-object aan met een initiële balans van 1000. Daarna storten we 200.5 op de rekening. Nu vergelijken we *balance* van onze *bankAccount* met het resultaat dat we verwachten.

Merk op dat we nog een derde parameter meegeven aan de methode *assertEquals()* wanneer we doubles vergelijken. Dit is een delta-waarde, of een indicatie hoeveel beide getallen mogen verschillen. In ons geval willen we exact 1200.5 als resultaat krijgen, vandaar de 0 als derde parameter. Wanneer we 2 gehele getallen met elkaar vergelijken, hoeft de 3^{de} parameter er niet te staan.

```
BankAccountTest.java ✕
1 package be.pxl.junit;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class BankAccountTest {
8
9     @Test
10    public void testDeposit() {
11        BankAccount bankAccount = new BankAccount("Guust", "Flater", 1000.0);
12        bankAccount.deposit(200.5);
13        assertEquals(1200.5, bankAccount.getBalance(), 0);
14    }
15
16 }
17
```

We voegen nu een extra test toe, voor de methode *withdraw()*.

Testen mogen elkaar niet beïnvloeden. Iedere test moet onafhankelijk van de andere testen uitgevoerd kunnen worden en een test is steeds maar verantwoordelijk om 1 scenario van 1 methode te evalueren.

```
BankAccountTest.java ✕
1 package be.pxl.junit;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class BankAccountTest {
8
9     @Test
10    public void testDeposit() {
11        BankAccount bankAccount = new BankAccount("Guust", "Flater", 1000.0);
12        bankAccount.deposit(200.5);
13        assertEquals(1200.5, bankAccount.getBalance(), 0);
14    }
15
16    @Test
17    public void testWithdraw() {
18        BankAccount bankAccount = new BankAccount("Guust", "Flater", 1000.0);
19        bankAccount.withdraw(50.6);
20        assertEquals(949.4, bankAccount.getBalance(), 0);
21    }
22
23 }
```


Voorzie nu de methode *isNegative()* voor de klasse **BankAccount**. Indien member variabele *balance* kleiner is dan 0, is de returnwaarde *true*, anders is de returnwaarde *false*.

Met *assertTrue(...)* en *assertFalse(...)* kan je beweringen schrijven over het resultaat van *isNegative()*.

Schrijf nu 3 testen voor de methode *isNegative()*: een eerste waarbij je het exacte bedrag van de bankrekening opneemt, één waarbij je een groter bedrag opneemt dan er op de bankrekening staat en een derde test waarbij je maar een gedeelte van het bedrag op de bankrekening opneemt.

Merk nu op dat de eerste regel van iedere test identiek is. In die regel maken we onze testgegevens klaar. Voor het initialiseren van testgegevens kan je ook een methode gebruiken die je annoteert met **@Before**. Hiermee duid je aan dat deze methode bij de start van iedere test eerst uitgevoerd moet worden.

```

1 package be.px1.junit;
2
3 import static org.junit.Assert.*;
4
5
6
7
8 public class BankAccountTest {
9
10     private BankAccount bankAccount;
11     private static final double INITIAL_AMOUNT = 1000.0;
12
13     @Before
14     public void init() {
15         bankAccount = new BankAccount("Guust", "Flater", INITIAL_AMOUNT);
16     }
17
18     @Test
19     public void testDeposit() {
20         bankAccount.deposit(200.5);
21         assertEquals(INITIAL_AMOUNT + 200.5, bankAccount.getBalance(), 0);
22     }
23
24     @Test
25     public void testWithdraw() {
26         bankAccount.withdraw(50.6);
27         assertEquals(INITIAL_AMOUNT - 50.6, bankAccount.getBalance(), 0);
28     }
29
30
31     @Test
32     public void testIsNegativeReturnsTrueWhenMoreMoneyWithdrawn() {
33         bankAccount.withdraw(INITIAL_AMOUNT + 0.01);
34         assertTrue(bankAccount.isNegative());
35     }
36
37     @Test
38     public void testIsNegativeReturnsFalseWhenBankAccountBalanceZero() {
39         bankAccount.withdraw(INITIAL_AMOUNT);
40         assertFalse(bankAccount.isNegative());
41     }
42

```

We gaan nu in de klasse **BankAccount** de methode *withdraw()* aanpassen. Bedragen boven 500.00 mogen enkel afgehaald worden in het bankkantoor en niet aan een bankautomaat. Hiervoor voeg je een extra boolean parameter *branch* toe (*branch* is *true* bij afhaling in het bankfiliaal en *false* indien afhaling bij bankautomaat). De methode *withdraw()* zal ook een boolean waarde teruggeven als resultaat. Deze resultaatwaarde is *true* indien de geldafhaling gelukt is, en *false* indien de geldafhaling niet lukt. Merk op dat je een aantal testen zal moeten aanpassen zodat ze terug correct kunnen werken. Voor de functionaliteit *withdraw()* zal je nieuwe testen moeten toevoegen (en de oude testen eventueel verwijderen of herschrijven) om de nieuwe vereisten van de methode te testen.

Indien het concept van **test driven development (TDD)** wordt toegepast, zullen de programmeurs eerst de testen voorzien. Pas hiervoor je signatuur van de methode *withdraw()* al aan, maar wijzig de implementatie nog niet. Schrijf vervolgens de testen, je krijgt dan al een goed idee hoe je code moet werken. Implementeer daarna de methode en kijk of alle testen slagen. Indien er nog testen falen, is je code nog niet compleet. Indien je testen slagen, bekijk je implementatie dan eens kritisch om te kijken of je de code kan vereenvoudigen.

```
@Test
public void testWithdrawMoreThan500AtBranch() {
    assertTrue(bankAccount.withdraw(500.01, true));
}

@Test
public void testWithdrawMoreThan500AtATMReturnsFalse() {
    assertFalse(bankAccount.withdraw(500.01, false));
}

@Test
public void testWithdraw500AtATMReturnsTrue() {
    assertTrue(bankAccount.withdraw(500, false));
}

@Test
public void testBalanceNotChangedWhenWithdrawingMoreThan500AtATM() {
    double amountBefore = bankAccount.getBalance();
    bankAccount.withdraw(500.01, false);
    assertEquals(amountBefore, bankAccount.getBalance(), 0);
}
```

Nog interessante informatie en artikelen:

<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

<http://tutorials.jenkov.com/java-unit-testing/asserts.html>

<https://objectpartners.com/2013/09/18/the-benefits-of-using-assertthat-over-other-assert-methods-in-unit-tests/>