JUnit

JUnit

- JUnit is a **framework** for testing Java programs
- Testing in the **small**
- Test the behavior of **single** methods
  - Based on **input-output** check
- By Kent Beck and Eric Gamma
- Open source
- Implementation of the xUnit family
  - Available also for other languages
- http://junit.org/
- Can be exploited inside **Eclipse**

How to design a test case

- Define **what** must be tested
  - E.g., a **method**
- Define **how** to execute the tested method
  - E.g., with which **arguments**
- Define the **expected result** for the defined execution
- **Implement** the test case
  - **Call** the tested method
  - **Check** whether the **actual** result match the **expected** value

What to test

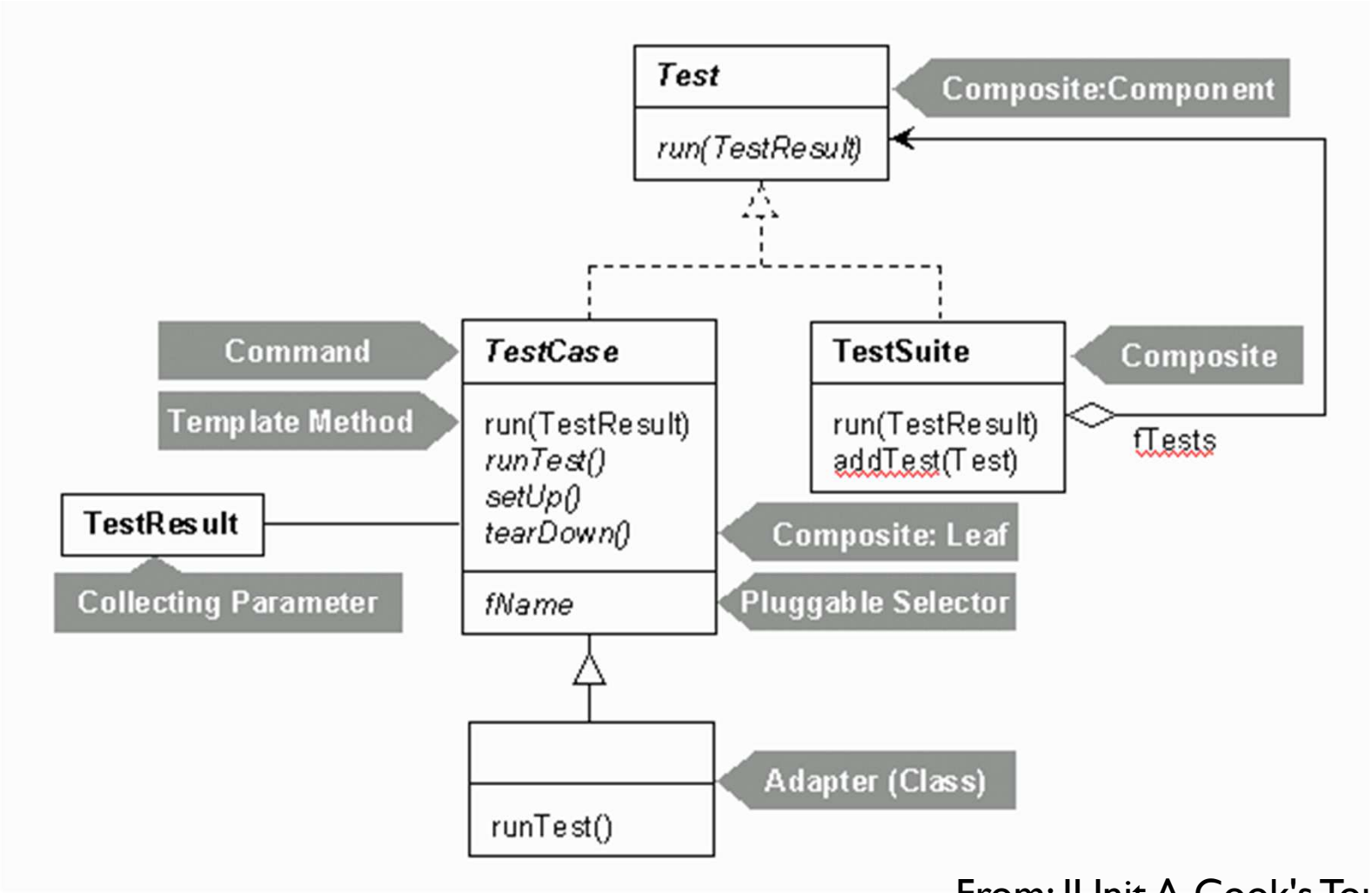
- A function that, given arguments, **returns** a **result**
- A function that **modifies** the **state** of the application

JUnit versions

- On August 16, 2000 the JUnit web site was published (but version 1.0 was released in 1998)
- Version 3
  - Released on ??
  - Can be used with Java **older** than version 5
  - **Still** used
- Version 4
  - Released February 16, 2006
  - Requires Java **5+**
  - Uses **annotations**
- Version 5 (Jupiter)
  - Released on September 10, 2017
  - Requires Java **8**
  - More **granularity**
  - **Simultaneous** execution of different tests

JUnit 3

JUnit class diagram (with patterns)



From: JUnit A Cook's Tour

Tests

- Tests are performed by methods in the form

```
public void testX()
```

  - No arguments
  - No return value
- A test method calls a few methods to be tested and checks if the result is what is expected
- Example:

```
public void testSum() {
    int sum = calc.sum(2,3);
    assertEquals(5, sum); // is sum == 5?
}
```

expected

result

The **`Assert`** class

- The **`Assert`** class provides a set of methods that perform specific tests
- **`assertEquals(Object expected, Object actual)`**
  - Tests if the two arguments are **equal** (i.e., **`equals`** returns true)
  - Versions with **different types** (String, double, int, …) are available
  - Methods with floats and doubles accept a "**delta**" (a margin of error)
    - The versions without delta are deprecated

The **`Assert`** class (2)

- **`assertTrue(boolean condition)`**
- **`assertFalse(boolean condition)`**
  - Test if the condition is **true** or **false**
- **`assertNull(Object object)`**
- **`assertNotNull(Object object)`**
  - Test if the argument is **null** or not
- **`assertSame(Object expected, Object actual)`**
- **`assertNotSame(Object expected, Object actual)`**
  - Test if two references point to the **same** object or not (i.e., ==)

The **`Assert`** class (3)

- If the test fails, an **`AssertionFailedError`** exception is thrown
- All methods are **static** and return **void**
- All methods have the version that accepts a **message** to be displayed when the test fails
- Example:

```
assertTrue(String message, boolean
condition)
```

The **TestCase** class

- The **TestCase** class defines the following protected and void-implemented methods:

```
protected void setUp()
protected void runTest()
protected void tearDown()
```

- Moreover, there is a method that calls the other method:

```
public void run(TestResult result) {
   setUp();
   runTest();
   tearDown();
}
```

The `TestCase` class (2)

- The `TestCase` class is a **subclass** of `Assert`, from which inherits the test methods
- The `TestCase` class implements the `Test` **interface**
- The `TestCase` class is **abstract** and must be **subclassed** to define a specific test

- The `TestSuite` class is exploited to run different tests
- Test **case**
  - Verifies a **single** code unit
  - Typically one or a few methods
- Test **suit**
  - A **set of test cases** that verifies related functionalities

Setup

- The **setup** of the test can be performed in the **`setUp()`** method
- It is useful to **initialize** objects that will be tested

- Similarly, the **`tearDown()`** method is executed at the **end** of the tests

- Hollywood model
  - Don't call me, I will call you (when I need you)

How to test exceptions

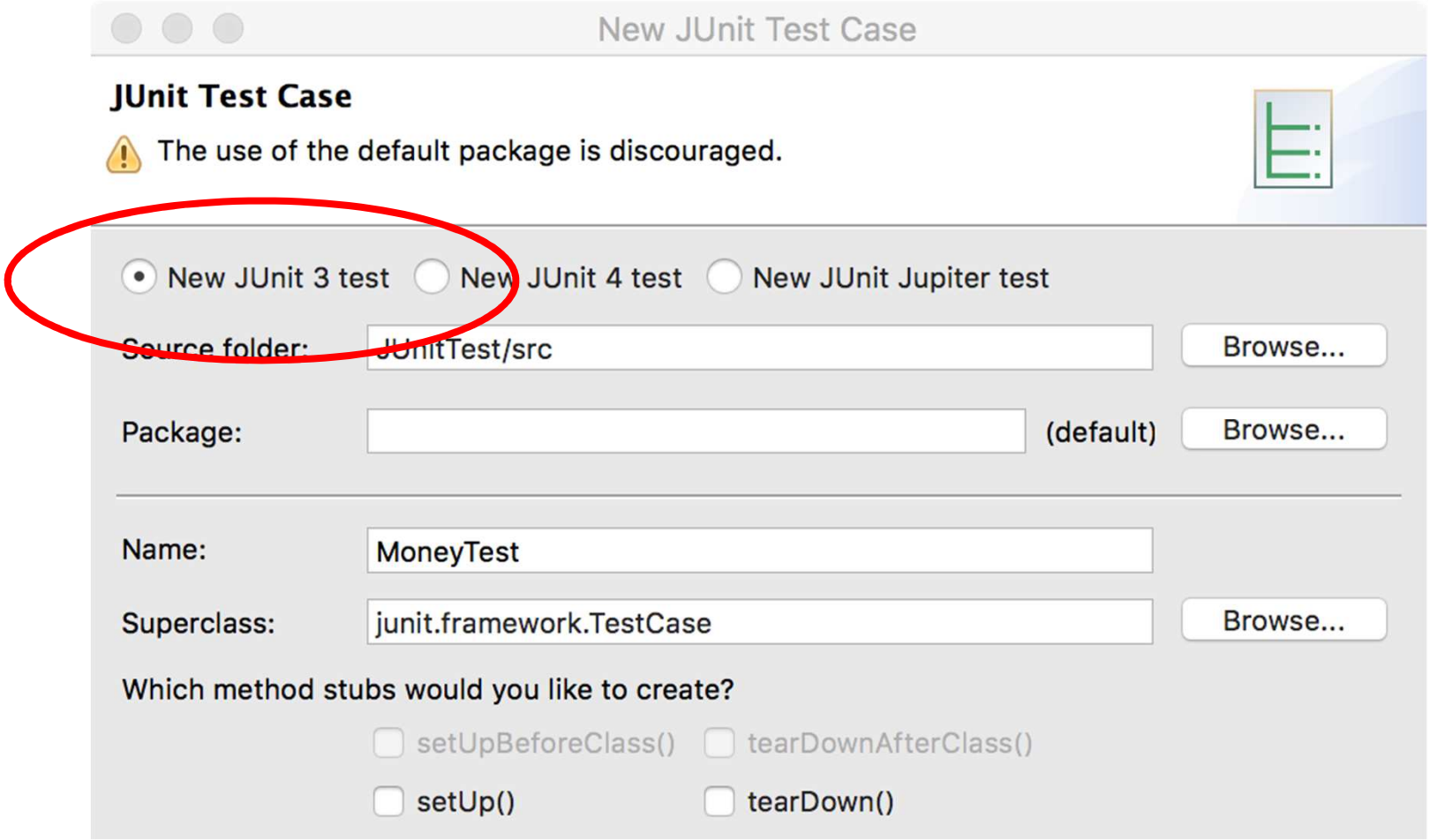
- To test if a method throws an exception given a bad parameter, we can call the method inside a **`try`** followed by **`fail()`**:

```
public void testException() {
    try {
        method_to_test(bad_argument);
        fail("Expected an exception");
    } catch (Exception e) {
       // do nothing }
}
```

- If the exception **IS** thrown, the fail is **NOT** executed and the test **succeeds**
- If the exception is **NOT** thrown, the fail **IS** executed and the test **fails**

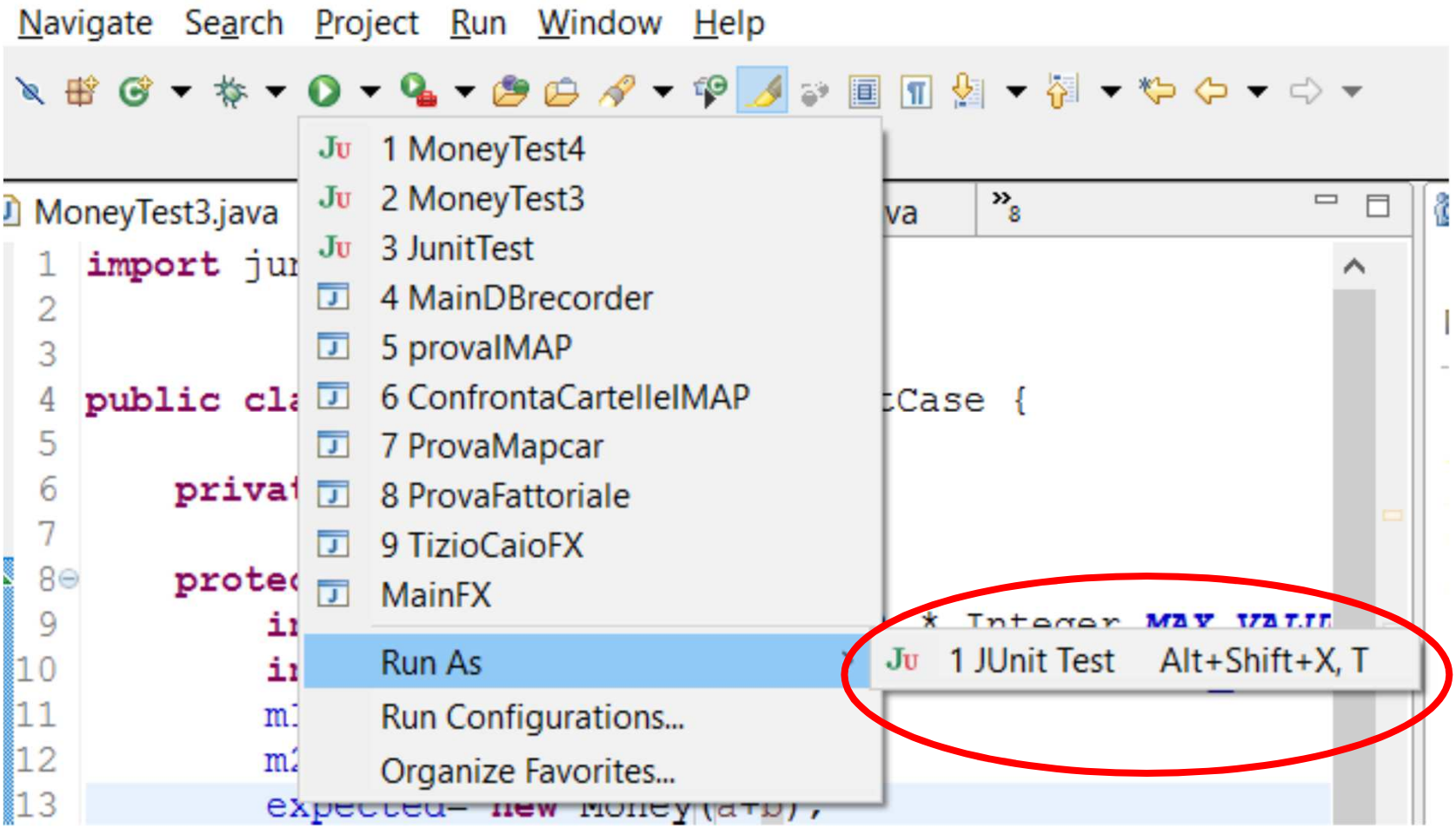
How to create tests

- In Eclipse, New → Java → Junit



How to run tests

- In Eclipse, a specific menu item is available



How to run tests (2)

- By **command line**:
- **`java junit.textui.TestRunner <test class name>`**
  - **Name**, not file!
- Be sure to have the needed directories in the **CLASSPATH** variable
  - Directory of the **test** classes
  - Directory of the classes **to be tested**
- Or specify the directories by the **`-cp`** command line option

How to run tests (3)

- A **GUI interface** is available by command line:
- **`java junit.swingui.TestRunner <test class name>`**

- JUnit tests can be run also by ANT

Example 1: class Money

```
public class Money {
  private int value;

  public Money(int v) { value = v; }
  public Money(){ this(0); }

  public Money add(Money second){ return
new Money(value+second.value);  }
```

Example 1: class Money (2)

```
  public boolean equals(Object o){
    if (o instanceof Money) return
((Money)o).value == value;
    else return false;
  }

  public String toString(){ return
""+value; }

  public int getValue(){ return value; }
}
```

Example 1: class MoneyTest3

```
import junit.framework.TestCase;
public class MoneyTest3 extends TestCase {
  private Money m1, m2, expected;

  protected void setUp(){
    int a = (int) (Math.random() *
Integer.MAX_VALUE);
    int b = (int) (Math.random() *
Integer.MAX_VALUE);
    m1= new Money(a);
    m2= new Money(b);
    expected = new Money(a+b);
 }
```

Example 1: class MoneyTest3 (2)

```
  public void testAdd() {
    Money result = m1.add(m2);
    assertEquals(expected, result);
  }
}
```

Example 1: results (Eclipse)

Package Explorer | JUnit

Finished after 0,005 seconds

Runs: 1/1 Errors: 0 Failures: 0

MoneyTest3 [Runner: JUnit 3] (0,000 s)

Failure Trace

MoneyTest3.java | MoneyTest4.java | Money.java

```
1  import junit.framework.TestCase;
2  public class MoneyTest3 extends TestCase {
3      private Money m1, m2, expected;
4
5      protected void setUp(){
6          int a = (int) (Math.random() * Integer.MAX_VALU
7          int b = (int) (Math.random() * Integer.MAX_VALU
8          m1= new Money(a);
9          m2= new Money(b);
10         expected= new Money(a+b);
11     }
12
13     public void testAdd() {
14             Money result= m1.add(m2);
15             assertEquals(expected, result);
16             //assertTrue(expected.equals(result));
17     }
18 }
19
```

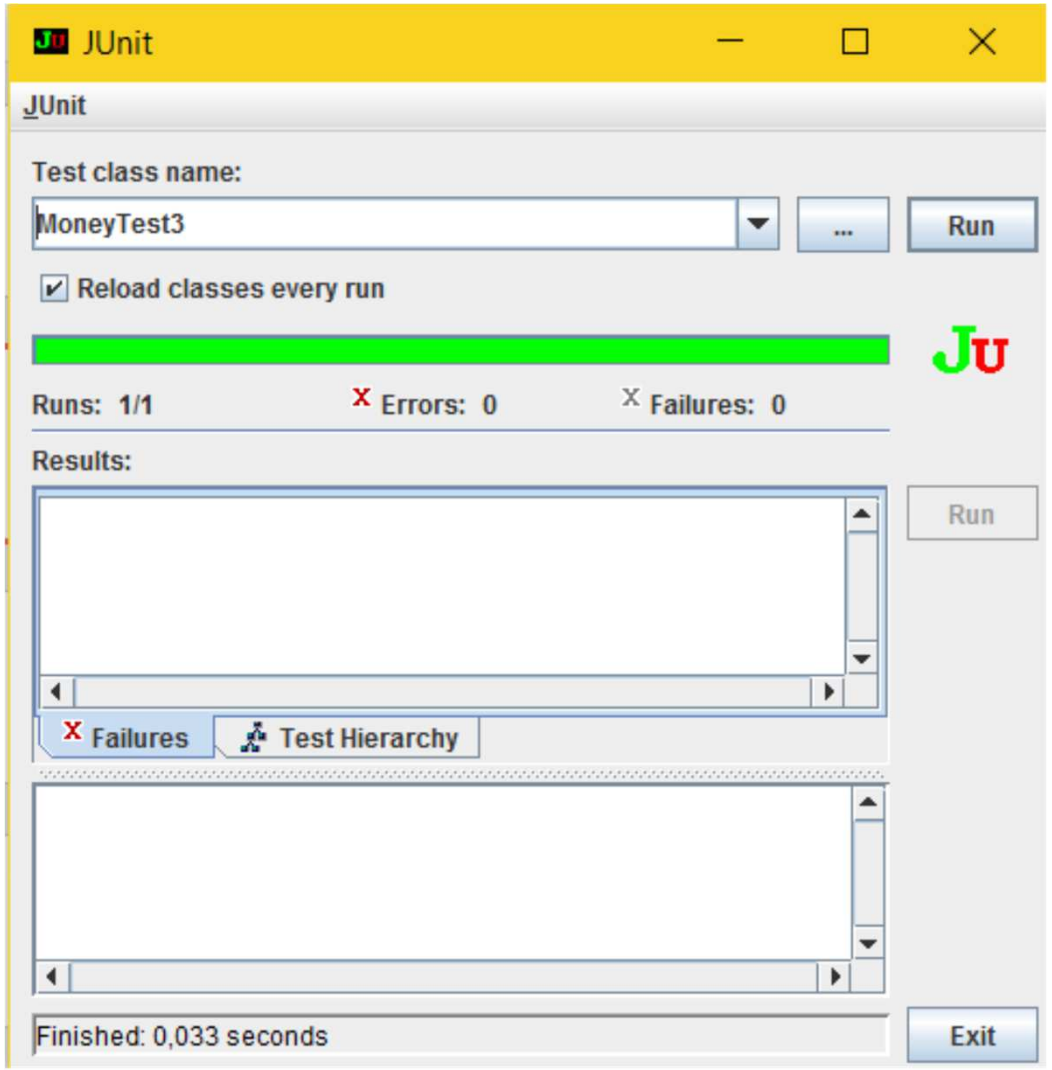
Example 1: results (text)

```
[C:\MyProg\java\JunitTest\bin]java -cp
.;..\..\junit.jar
junit.textui.TestRunner MoneyTest3
.
Time: 0,003

OK (1 test)


[C:\MyProg\java\JunitTest\bin]
```

Example 1: results (GUI)



JUnit 4

JUnit 4

- Same **concepts**
- Different **implementation**
  - Exploits Java **annotations**
- Requires Java 5+
- Now `Test` is an *annotation*, not an interface

- Java annotation: **metadata** about the code, not code
  - Information for the compiler
  - Compile-time and deployment-time processing
  - Runtime processing

Use of Java annotations

- @Test (import org.junit.Test)
  - Specifies that the method is a **test**
  - **Replace `TestCase`** class
- @Before (import org.junit.Before)
  - The method is executed **before every** test
- @After (import org.junit.After)
  - The method is executed **after every** test
- @BeforeClass (import org.junit.BeforeClass)
  - The **static** method is executed **before** executing the **first** test
- @AfterClass (import org.junit.AfterClass)
  - The **static** method is executed **after** having executed the **last** test

Use of Java annotations

- @Ignore (import org.junit.Ignore)
  - Specifies to **ignore** a test
  - Can be applied to a test **method** or to a test **class**
- @Test(timeout=500) (import org.junit.Test)
  - Specifies a **timeout** after which the test fails
- @Test(expected=*Exception.class*)
  - Specifies to test the **throwing** of a given exception

The `Assert` class

- The `Assert` class is still available, with the **same test methods**
  - `assertEquals(Object expected, Object actual)`
  - `assertTrue(boolean condition)`
  - `assertFalse(boolean condition)`
  - `assertNull(Object object)`
  - `assertNotNull(Object object)`
  - `assertSame(Object expected, Object actual)`
  - `assertNotSame(Object expected, Object actual)`
- They are static, must be invoked
  - By `Assert.assertX()`
  - By **statically importing** the methods (see later)

Setup

- In version 4, **any** method can be labelled as "setup" method by means of the **@Before** and **@BeforeClass** annotation
- So, there is **no need** to define a specific **`setUp()`** method

Test suites

- A set of tests can be run
- The **`Suite`** class is available
  - Exploited by an annotation
  - **`@RunWith(Suite.class)`**
- By another annotation the test classes are listed
- **`@Suite.SuiteClasses`**

Example of Test suites

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

@RunWith(Suite.class)

@Suite.SuiteClasses({
    TestJunit1.class,
    TestJunit2.class
})

public class JunitTestSuite {
}
```

How to test exceptions

- To test if a method throws an exception given a bad parameter, we can exploit @Test(expected=*Exception.class*):

```
@Test(expected=Exception.class)
public void testException() {
  method_to_test(bad_argument);
}
```

- If the exception is **NOT** thrown, the test **fails**

How to test exceptions (2)

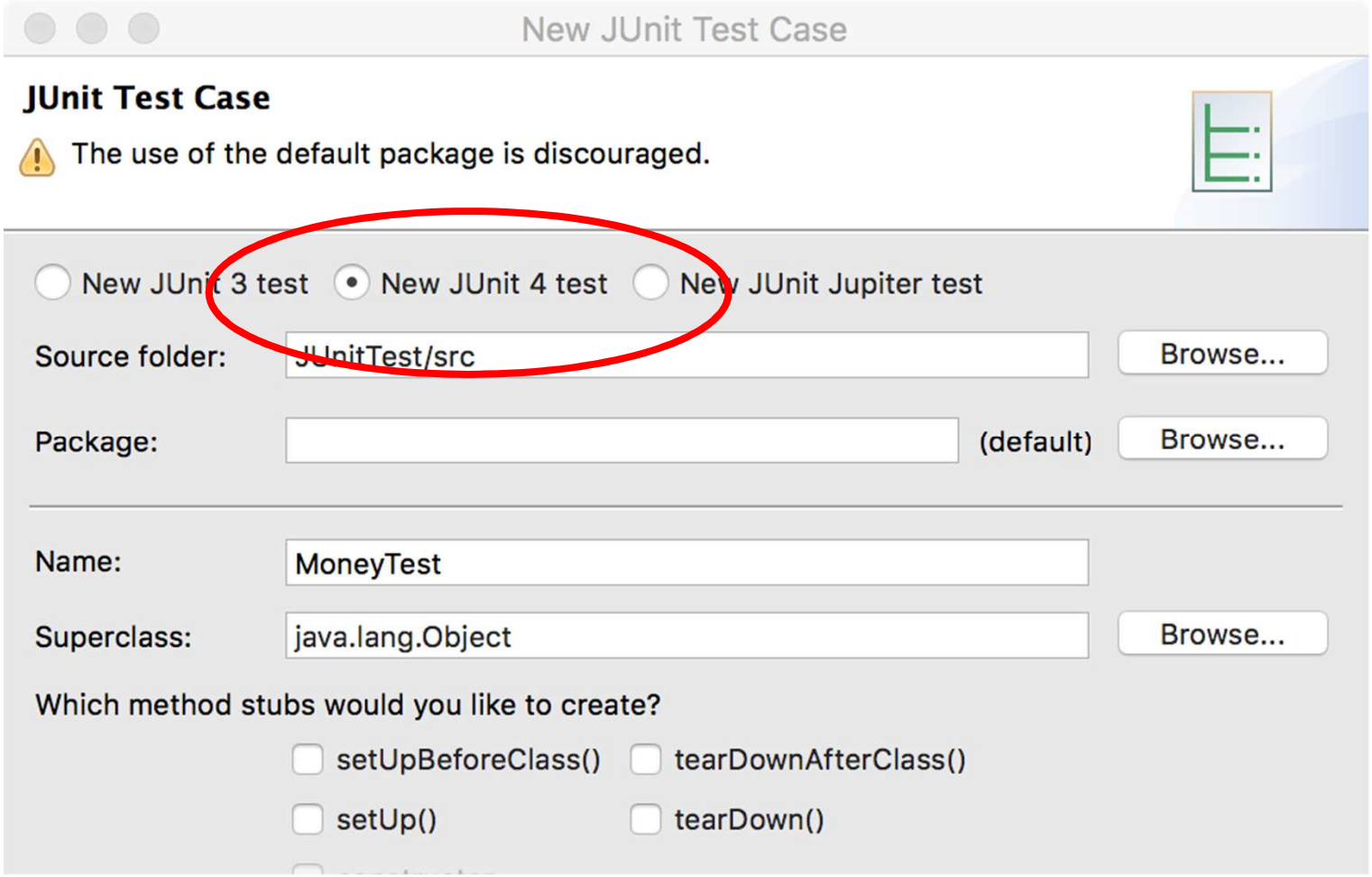
- However, if we need **more control** on the test, we can exploit the `fail()` method as in JUnit 3:

```
@Test
public void testException() {
    try {
        method_to_test(bad_argument);
        fail("Expected an exception");
    } catch (Exception e) {
       // do nothing }
}
```

- If the exception **IS** thrown, the `fail` is **NOT** executed and the test **succeeds**
- If the exception is **NOT** thrown, the `fail` **IS** executed and the test **fails**

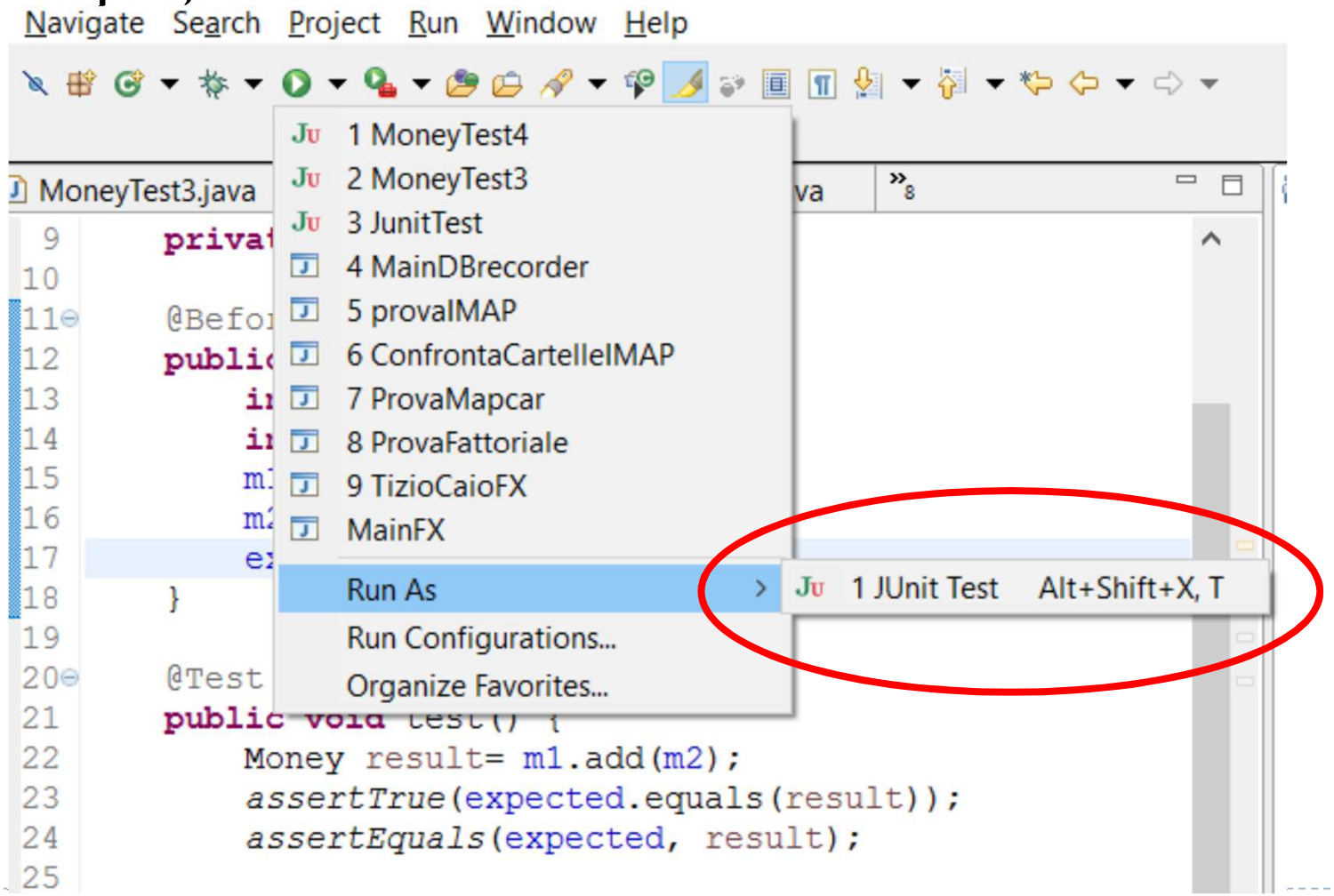
How to create tests

- In Eclipse, New → Java → Junit



How to run tests

- In Eclipse, the same menu item runs version 4



How to run tests (2)

- By **command line**:
- **`java org.junit.runner.JUnitCore <test class name>`**
  - **Name**, not file!
- Be sure to have the needed directories in the **CLASSPATH** variable
  - Directory of the test classes
  - Directory of the classes to be tested
- Or specify the directories by the **`-cp`** option

- JUnit tests can be run also by ANT

Example 2: class MoneyTest4

```
import static org.junit.Assert.*;
import org.junit.*;

public class MoneyTest4 {
      private Money m1, m2, expected;

      @Before
      public void createMoney(){
            int a = 1;
            int b = 2;
            m1 = new Money(a);
            m2 = new Money(b);
            expected = new Money(a+b);
      }
```

Needed because this is not a subclass of Assert

Example 2: class MoneyTest4 (2)

```
    @Test
    public void test() {
        Money result= m1.add(m2);
        assertEquals(expected, result);
    }
}
```

Example 2: results (Eclipse)

Package Explorer | JUnit

Finished after 0,013 seconds

Runs: 1/1 Errors: 0 Failures: 0

MoneyTest4 [Runner: JUnit 4] (0,000 s)

Failure Trace

MoneyTest4.java | Money.java

```java
import static org.junit.Assert.*;
import org.junit.*;

public class MoneyTest4 {
    private Money m1, m2, expected;

    @Before
    public void createMoney(){
        int a = 1;
        int b = 2;
        m1 = new Money(a);
        m2 = new Money(b);
        expected= new Money(a+b);
    }

    @Test
    public void test() {
        Money result= m1.add(m2);
        assertTrue(expected.equals(result));
        assertEquals(expected, result);

```

Example 2: results (text)

```
[C:\MyProg\java\JunitTest\bin]java -cp
.;..\..\* org.junit.runner.JUnitCore
MoneyTest4
JUnit version 4.12
.
Time: 0,015

OK (1 test)


[C:\MyProg\java\JunitTest\bin]
```

What happens in case of error?

- Let us suppose that the **add** method is wrong:
- `public Money add(Money second){`
- `  return new Money(value-second.value);`
- `}`

absent-minded programmer

Finished after 0,027 seconds

Runs: 1/1 Errors: 0 Failures: 1

MoneyTest4 [Runner: JUnit 4] (0,000 s)

test (0,000 s)

Failure Trace

java.lang.AssertionError: expected:<3> but was:<-1>

at MoneyTest4.test(MoneyTest4.java:20)

What happens in case of error? (2)

- From command line

```
[C:\MyProg\java\JunitTest\bin]java -cp .;..\..\*
org.junit.runner.JUnitCore MoneyTest4
JUnit version 4.12
.E
Time: 0,017
There was 1 failure:
1) test(MoneyTest4)
java.lang.AssertionError: expected:<3> but was:<-1>
        at org.junit.Assert.fail(Assert.java:88)
        ...

FAILURES!!!
Tests run: 1,  Failures: 1
```

JUnit 5

Differences vs version 4

- Assertions class instead of Assert class
- Relies on Java 8 features
  - Lambda expressions

Differences vs version 4 (exception)

- Parameter of the @Test annotation are not allowed
- Version 4:

```
@Test(expected = Exception.class)
public void shouldRaiseAnException() throws
Exception {
    // ...
}
```

- Version 5:

```
public void shouldRaiseAnException() throws
Exception {
    Assertions.assertThrows(Exception.class,
() -> {    //...    });
}
```

Differences vs version 4 (timeout)

- Version 4:

```
@Test(timeout = 1)
public void shouldFailBecauseTimeout()
throws InterruptedException {
    Thread.sleep(10);
}
```

- Version 5:

```
@Test
public void shouldFailBecauseTimeout()
throws InterruptedException {

Assertions.assertTimeout(Duration.ofMillis(1
), () -> Thread.sleep(10));
}
```

Other modifications

- *@Before* annotation is renamed to *@BeforeEach*
- *@After* annotation is renamed to *@AfterEach*
- *@BeforeClass* annotation is renamed to *@BeforeAll*
- *@AfterClass* annotation is renamed to *@AfterAll*
- *@Ignore* annotation is renamed to *@Disabled*

- *@Order(n) specifies the order of the tests (from JUnit 5.4)*
  - *Otherwise, the order of the execution is not granted*

Example 3: class MoneyTest5

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

public class MoneyTest5 {
      private Money m1, m2, expected;

      @BeforeEach
      public void createMoney(){
            int a = 1;
            int b = 2;
            m1 = new Money(a);
            m2 = new Money(b);
           expected = new Money(a+b);
      }
```

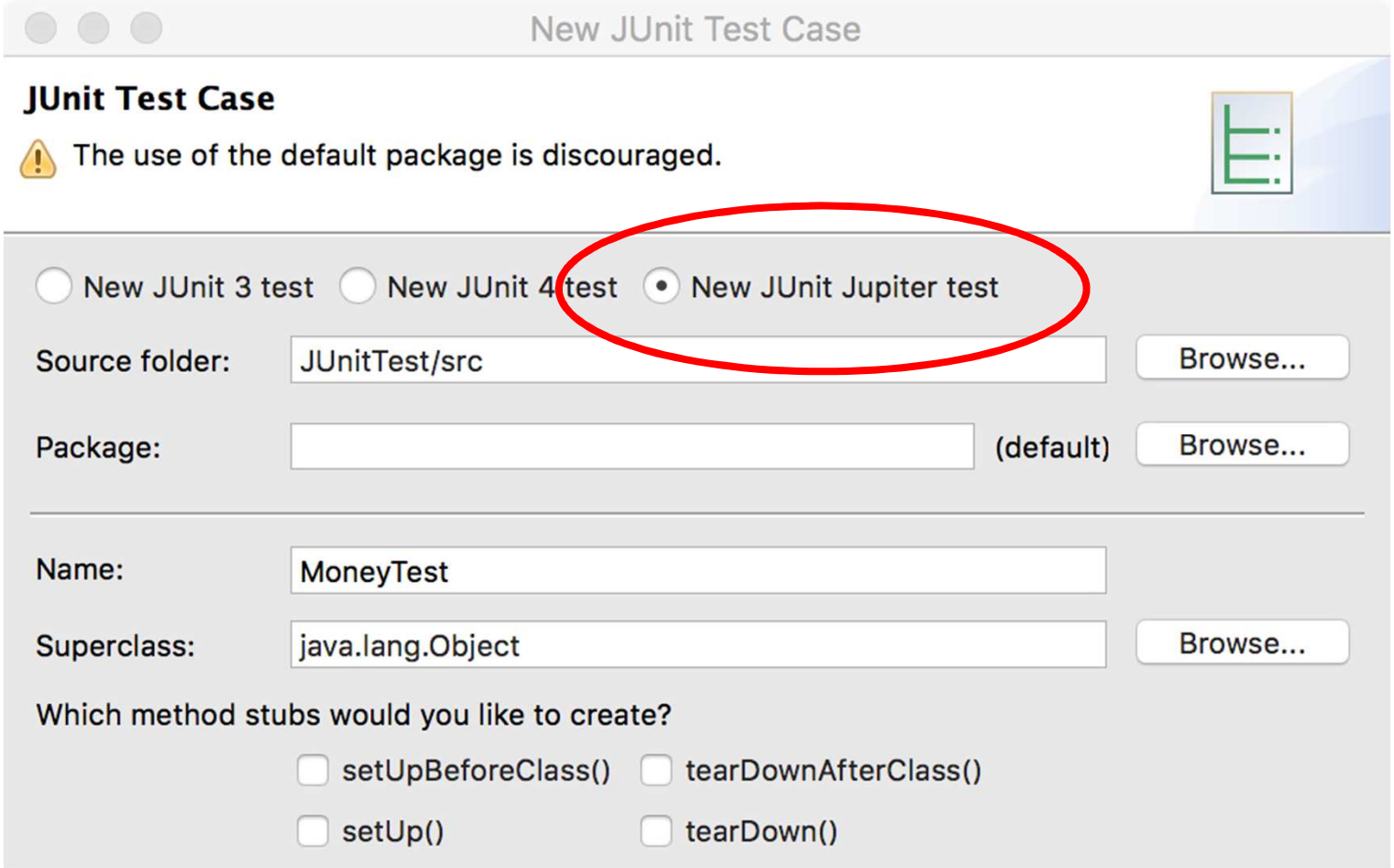
Needed because this is not a subclass of Assertions

Example 3: class MoneyTest5 (2)

```
    @Test
    public void test() {
        Money result= m1.add(m2);
        assertEquals(expected, result);
    }
}
```

How to create tests

- In Eclipse, New → Java → Junit



How to run tests

- Download the **`junit-platform-console-standalone-<version>.jar`** file
- By **command line**:
- **`java -jar junit-platform-console-standalone-1.1.0-M2.jar --cp <classes' path> -c <test class name>`**
- **Name**, not file!

How to run tests (2)

- Example
- `$ java -jar junit-platform-console-standalone-1.1.0-M2.jar --cp . -c MoneyTest5`
- `|`
- `├─ JUnit Jupiter ✓`
- `└─ JUnit Vintage ✓`
-

```
Test run finished after 28 ms
[         2 containers found      ]
[         0 containers skipped    ]
[         2 containers started    ]
[         0 containers aborted    ]
[         2 containers successful ]
[         0 containers failed     ]
[         0 tests found           ]
[         0 tests skipped         ]
[         0 tests started         ]
[         0 tests aborted         ]
[         0 tests successful      ]
[         0 tests failed          ]
```

Summing up

Best practices

- Test should be written either **before** the code or by **another** person
  - Who writes the test is **not** influenced by the code
  - Test Driven Development
- Perform the test with "**edge**" values
  - E.g., the **first** or the **last** element of an array, the **limit** values of a set, …
- Re-run the tests **often**
  - **Regression** tests

To test or not to test?

- The most effective debugging tool is still careful **thought**, coupled with judiciously placed **print** statements [Brian Kernighan, "Unix for Beginners“,1979]
- Whenever you are tempted to type something into a print statement or a debugger expression, write it as a **test** instead [Martin Fowler]

Pros and cons of JUnit

- Pros
  - Enables **automatic** tests
  - **Standard** de facto
    - Supported by different **IDEs** (e.g., Eclipse)
  - **Regression** tests
  - Focus on **what** the software is expected to do, not how
  - Many **extensions** available
- Cons
  - A **lot of code** (and time) is required
  - **Heavy** infrastructure for simple tests
  - Risk of spending **more time** in finding the test than solving the problem
  - Risk on **relying too much** on tests for correctness
    - Remember the halt problem that cannot be decided