

■ 演習プリント 1

JUnit の操作演習

1. テスト用プロジェクト作成

最初にテスト用のプロジェクトを作成します。**Eclipse** 画面のファイル・メニューから、新規→Java プロジェクトを選択します。そこで、次の画面のようにプロジェクト名を入力して「完了」をクリックします。

新規 Java プロジェクト

Java プロジェクトの作成

Java プロジェクトをワークスペースまたは外部ロケーションに作成します。

プロジェクト名 (P):

NyujoRyokin

☐ デフォルト・ロケーションを使用 (D)

ロケーション (L):

c:\User¥xxxx¥Desktop¥workspace¥NyujoRyokin

参照 (R)

JRE

☒ 実行環境 JRE の使用 (V):

JavaSE-1.8

☐ プロジェクト固有の JRE を使用 (S):

jre

☐ デフォルト JRE の使用 (A) (現在は 'jre')

ire を構成

プロジェクト・レイアウト

☐ プロジェクト・フォルダーをソースおよびクラス・ファイルのルートとして使用 (U)

☒ ソースおよびクラス・ファイルのフォルダーを個別に作成 (C)

デフォルトを構成

ワーキング・セット

☐ ワーキングセットにプロジェクトを追加 (T)

新規 (W)

ワーキングセット (O):

選択 (E)

☐

< 戻る (B)

次へ (N) >

完了 (F)

キャンセル

そうすると、最初に次のようなコードが生成されます。(実際には、// のコメントはありません)

```
package junit.tutorial;           // チュートリアルを使用した例
public class Ryokin {             // クラスの名前の例
}
```


この原型に、必要なコードを追加してプロダクション用となる「テスト対象クラス」を作り上げます。

コーディングの順序として、先に「テストクラス」をコーディングし、そのテストに合格するように「テスト対象クラス」をコーディングしていく手法があります。この手法を「テスト駆動型開発」と呼びます。

2. テストクラスの作成

「テスト対象クラス」とは別のソース・フォルダーに「テストクラス」を作成します。

- Eclipse から「テスト対象クラス」を開いておいて「ファイル」⇒「新規」⇒「JUnit テストケース」を選択します。
- ダイアログが開かれるので、「パッケージ」にはクラスを格納する場所の名前を入力し、「名前」の箇所には、これから作成する「テストクラス」の名称（ここでは NyujoRyokin）を記入します。
- 「テスト元クラス」に「テスト対象クラス」を指定して「完了」をクリックします。

新規 JUnit テスト・ケース		<input type="checkbox"/>	X
JUnit テスト・ケース			
 デフォルト・パッケージの使用は推奨されません。			
<hr/>			
○ 新規JUnit3テスト(3) ● 新規JUnit4テスト(4)			
ソース・フォルダー(D):	<input type="text" value="JUnitテスト/src"/>	参照(O)	
パッケージ(K):	<input type="text"/>	(デフォルト)	参照(W)
<hr/>			
名前(M):	<input type="text" value="RyokinTest"/>		
スーパークラス(S):	<input type="text" value="junit.framework.TestCase"/>	参照(E)	
どのメソッド・スタブを作成しますか？			
<input type="checkbox"/> SetUpBeforeClass(P)		<input type="checkbox"/> tearDownAfterClass(A)	
<input type="checkbox"/> SetUp(U)		<input type="checkbox"/> tearDown(T)	
<input type="checkbox"/> コンストラクター(C)			
コメントを追加しますか？(テンプレートの構成およびデフォルト値については ここ を参照)			
<input type="checkbox"/> コメントの生成(G)			
<hr/>			
テスト元クラス(L):	<input type="text" value="NyujoRyokin"/>	参照(R)	
<hr/>			
<input type="checkbox"/>	<input type="button" value="戻る(B)"/>	<input type="button" value="次へ(N)>"/>	<input type="button" value="完了(F)"/>
<input type="button" value="キャンセル"/>			

そうすると、次のようなテストクラスの原型が自動的に作成されます。(コメントの部分は自動生成ではありません)

```

package junit.tutorial;           // チュートリアルを使用した例
import static org.junit.Assert.*;
import org.junit.Test;
public class NyujoRyokin { // クラスの名前の例
    @Test                        // Test アノテーション
    public void test() {
        fail("Not yet implemented");
    }
}

```

この原型に、必要なコードを追加してテストコードを作り上げます。

このまま「Run（実行）」させると、コードに記述されている”Not yet implemented（まだ実装されていません）”のメッセージが出されてエラーとなり、「赤色バー」の上部に「Runs（実行）:1/1 Error（エラー）:0 Fail（失敗）:1」が表示されます。この時点では、これは正常です。英語版には（）内の日本語はありません。

3. テストメソッド

次にテスト仕様書に従って「テストクラス」のソフトウェアに「テストメソッド」を定義します。これがテストコードであり、「テスト対象クラス」の振る舞いを検証する機能となります。テストコードは、機能単位であったり条件単位であったりします。例えば、ひとつのクラスにおいて、次のように二つの乗算の結果をテストすることができます。これは非常に簡単な例ですが、境界値テストなどによく使われます。

```

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

省略

@Test
Public void 乗算 12 テスト() {
    Calculator calc = new Calculator();
    int expected =12           // 期待値
    int actual = calc.multiply(3,4); // 3 x 4
    assertThat(actual, is(expected)); // 計算結果と期待値の比較
}

```

省略

@Test

```
Public void 乗算 35 テスト() {
```

```
    Calculator calc = new Calculator();
```

```
    int expected = 35                // 期待値
```

```
    int actual = calc.multiply(5,7); // 5 x 7
```

```
    assertThat(actual, is(expected)); // 計算結果と期待値の比較
```

```
}
```

以下省略

4. アノテーション

「@Test」は Java で使われるアノテーションのひとつです。アノテーションは Java5 から導入された言語仕様ですが、クラス、メソッド、変数などについて注釈を加える役割を担っています。実際に JUnit で使われる主なアノテーションを表にしました。

アノテーション	意味
@Test	テストメソッドであることを宣言する。二つの属性をもつ expected; 例外の検証テストにおいて期待される例外クラスを指定する timeout; 実行時間がかかりすぎる場合に失敗させる
@Ignore	テストの実行から除外する。一時的に停止させる場合に使用する
@Before	テスト実行前に処理する
@After	テスト実行後に処理する
@BeforeClass	テスト実行前に一度だけ処理する
@AfterClass	テスト実行後に一度だけ処理する

さらに、JUnit では値を比較する手法として Assert クラスに assertThat メソッドや Matcher API を活用します。JUnit の以前のバージョンには assertEquals や assertTrue などもありましたが、現在は assertThat と fail メソッドのみで十分です。assertThat は二つの引数をもっています。ひとつ目は実測値で、ふたつ目は期待値です。JUnit では基本的には static インポートを使うことを想定しているのですが、static インポートを使わないで記述すると次のようになります。

```
Assert.assertThat(1 + 2, CoreMatchers.is(3));
```

これは、 $1 + 2 = 3$ であることを検証するためのコードです。

CoreMatchers API を活用して **static** インポートを使えば、次のように簡単に自然言語に近くなります。

```
assertThat(1 + 2, is(3));
```

fail メソッドは、無条件にテストを失敗させるために使われます。

```
@Test
public void 失敗のテスト() {
    fail("失敗しました");
}
```

このように、**fail** メソッドからは直ちに **AssertionError**（この例では“失敗しました”）が送出されて失敗させます。