Javaプログラミング実習

33. プログラムのテスト

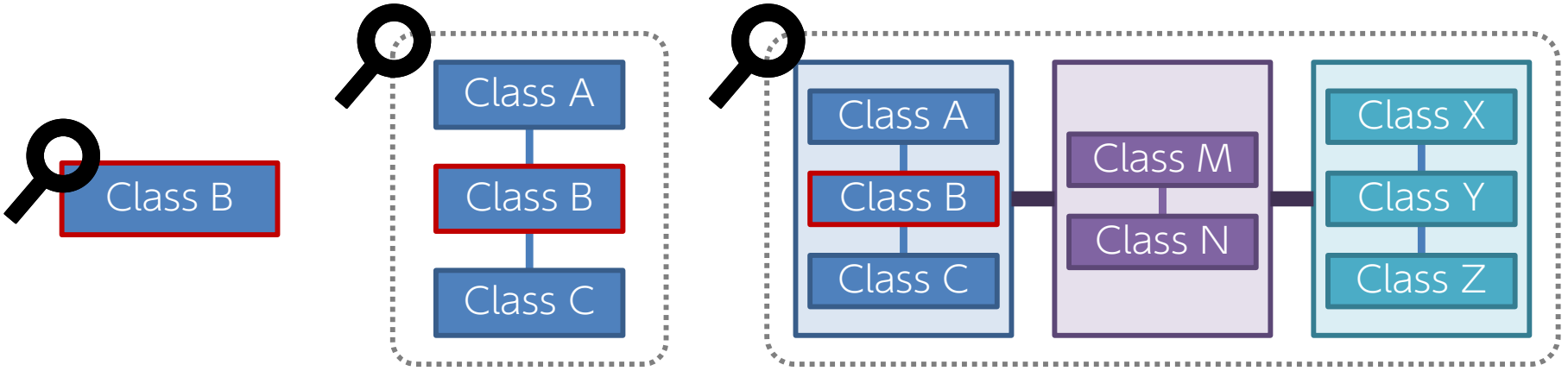
株式会社ジードライブ

今回学ぶこと

- JUnitを利用したプログラムの単体テストの方法

テストについて

- プログラムやシステムが仕様通り動作することを確認する作業
- 単体テスト、結合テスト、総合テストなど、プログラムやシステムの開発段階に応じた様々なレベルのテストが存在する



Unit Test
(単体テスト)

Integration Test
(結合テスト)

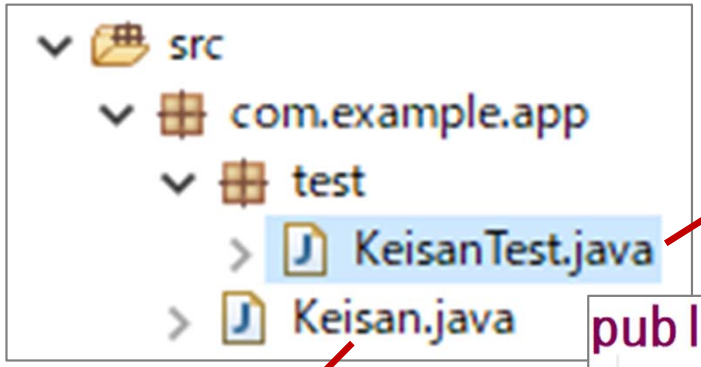
System Test
(総合テスト)

単体テストの方法

- 単体テストでは、クラス内の各メソッドが正しく動作するかの確認を行う
  - エラーが発生しないかを確認する
  - 期待通りの結果を返すか確認する
  - 基本的には、publicなメソッドをテストする
- 単体テストの実施方法としては、以下のようなアプローチが考えられる
  - `main`メソッドを持つテスト用のクラスを作成してテストを行う方法
  - テスト用のフレームワークを利用する方法

mainメソッドによるテスト

- mainメソッドをもつテスト用のクラスを作成し、System.out.printlnで結果を確認する



テスト用クラス

テスト対象のクラス

```
public class KeisanTest {

    public static void main(String[] args) {
        int test1 = Keisan.add(3, 5);
        int test2 = Keisan.multiply(3, 5);
        System.out.println(test1);
        System.out.println(test2);
    }

}
```

テスト対象のメソッドを実行し、
結果をコンソールに出力して確認する

JUnit：テスト用フレームワーク

- Junitは、テストを効率的に行うためのJava用のフレームワーク(ライブラリ)
  - https://junit.org
  - EclipseにはJUnitを使うためのプラグインが組み込まれており、簡単に利用することができる

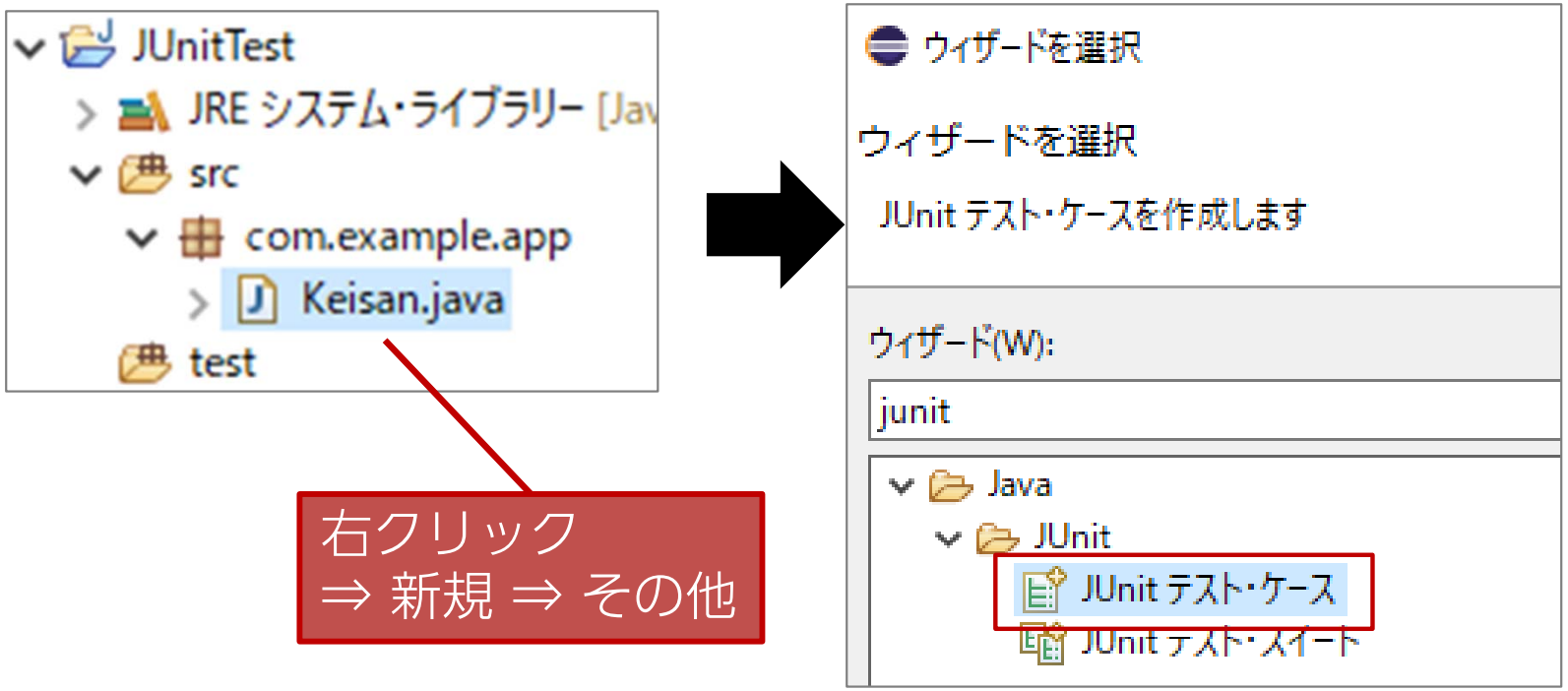
Junitの導入

1. テスト用ソースフォルダの作成

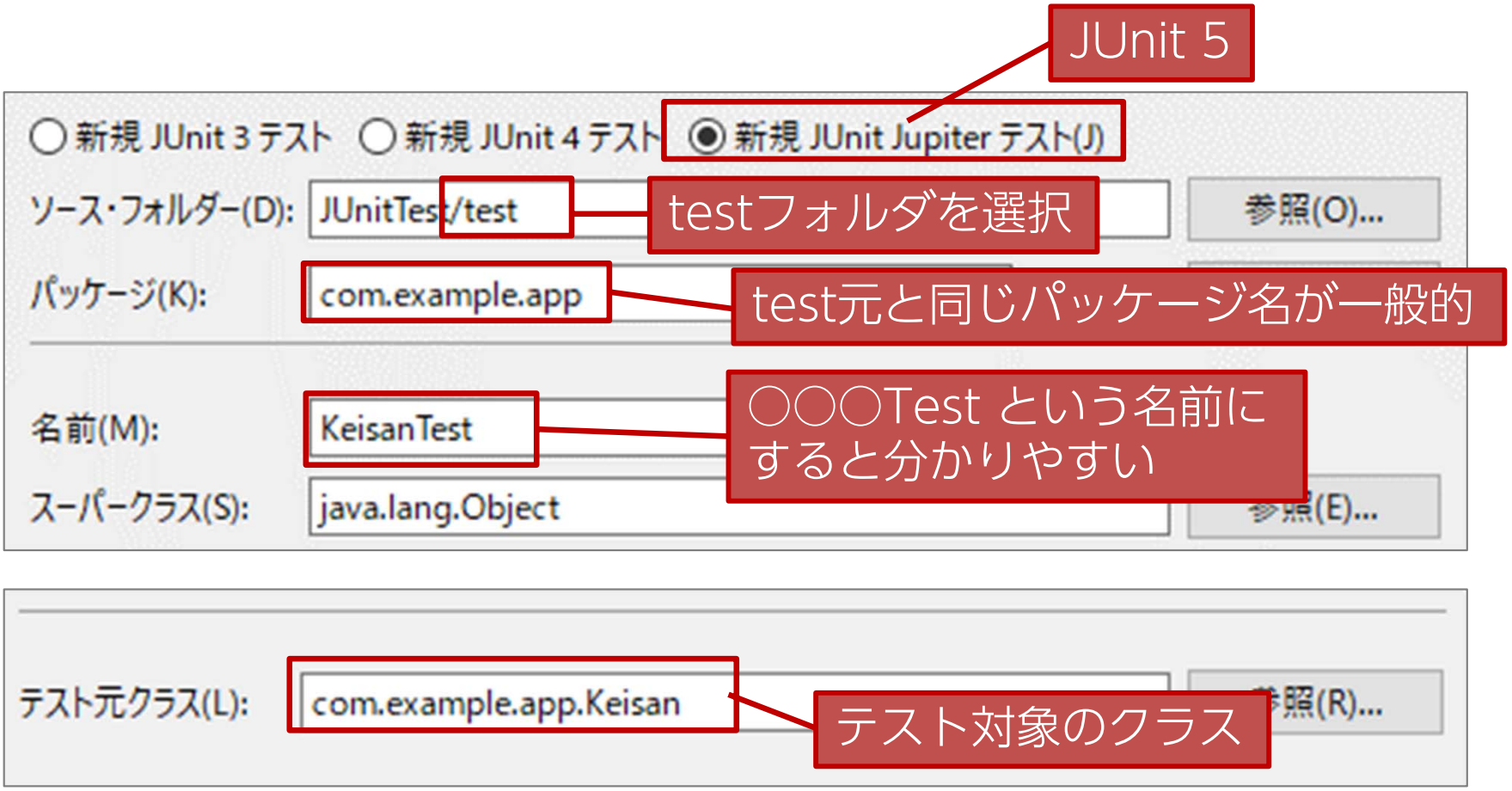
- テストケース格納用に「test」という名前の新規ソースフォルダを作成する
  - ソースフォルダ名は任意

2. テストケースの作成

- テストの対象となるクラス上で右クリックし、
  新規 ⇒ その他
- Java ⇒ JUnit ⇒ Junitテストケース を選択する
  - テストケースは、テスト内容を記述したクラス



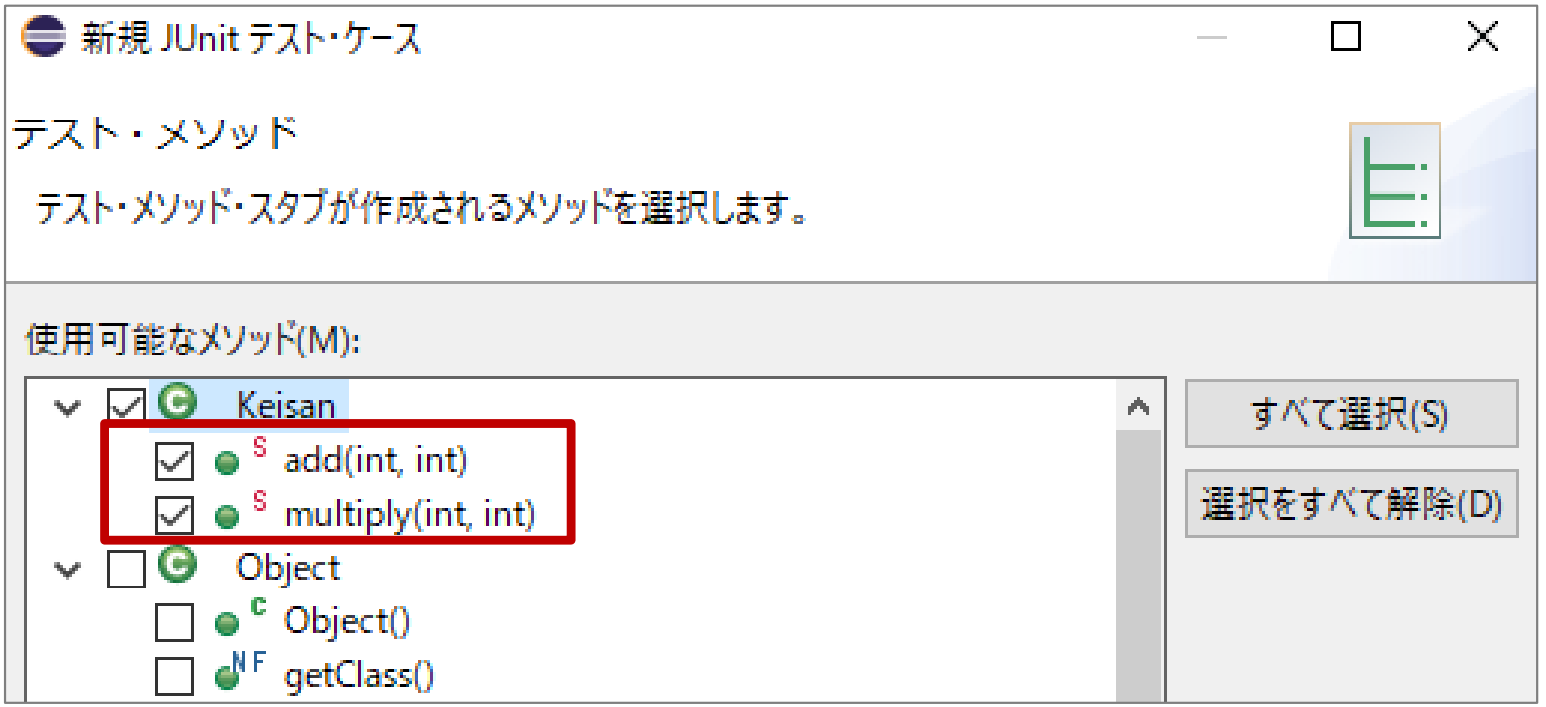
3. テストケースの設定



テスト対象のメソッドを選択する場合は「次へ」を押下

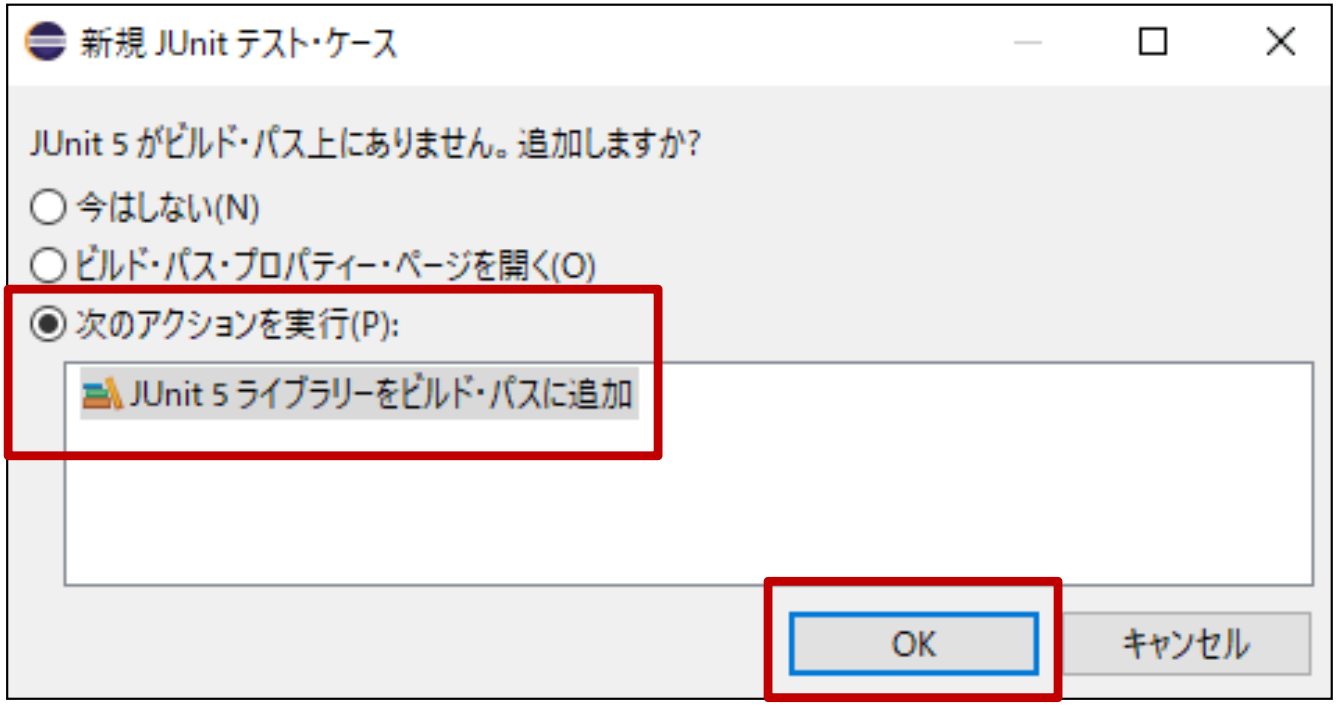
4. テスト対象のメソッド

- テスト対象となるメソッドを選ぶ



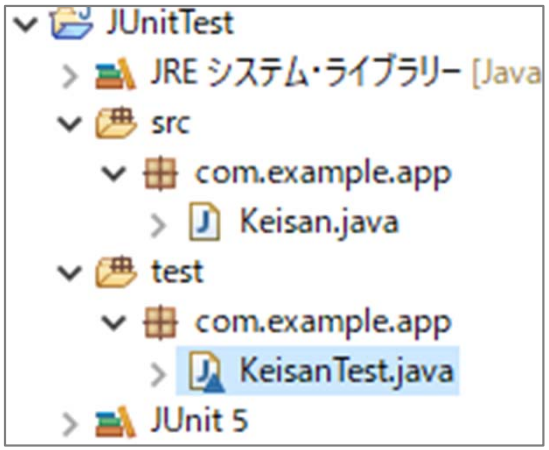
5. Junitの読み込み

- 初回のテストケース作成では、JUnitライブラリがまだ読み込まれていないため、以下のダイアログが表示される



6. テストケースの生成

- テストケース(テスト用クラス)が生成され、**@Test**というアノテーションが付いたテスト用メソッドが挿入される




```
1  package com.example.app;
2
3  import static org.junit.jupiter.api.Assertions.*;
6
7  class KeisanTest {
8
9      @Test
10     void testAdd() {
11         fail("まだ実装されていません");
12     }
13
14     @Test
15     void testMultiply() {
16         fail("まだ実装されていません");
17     }
18
19 }
```

テスト用メソッドに付与するアノテーション

明示的にテストを失敗させるメソッド

7. テスト用メソッドを実装

- 各メソッドをテストするためのメソッドを実装する

add() をテストするメソッド

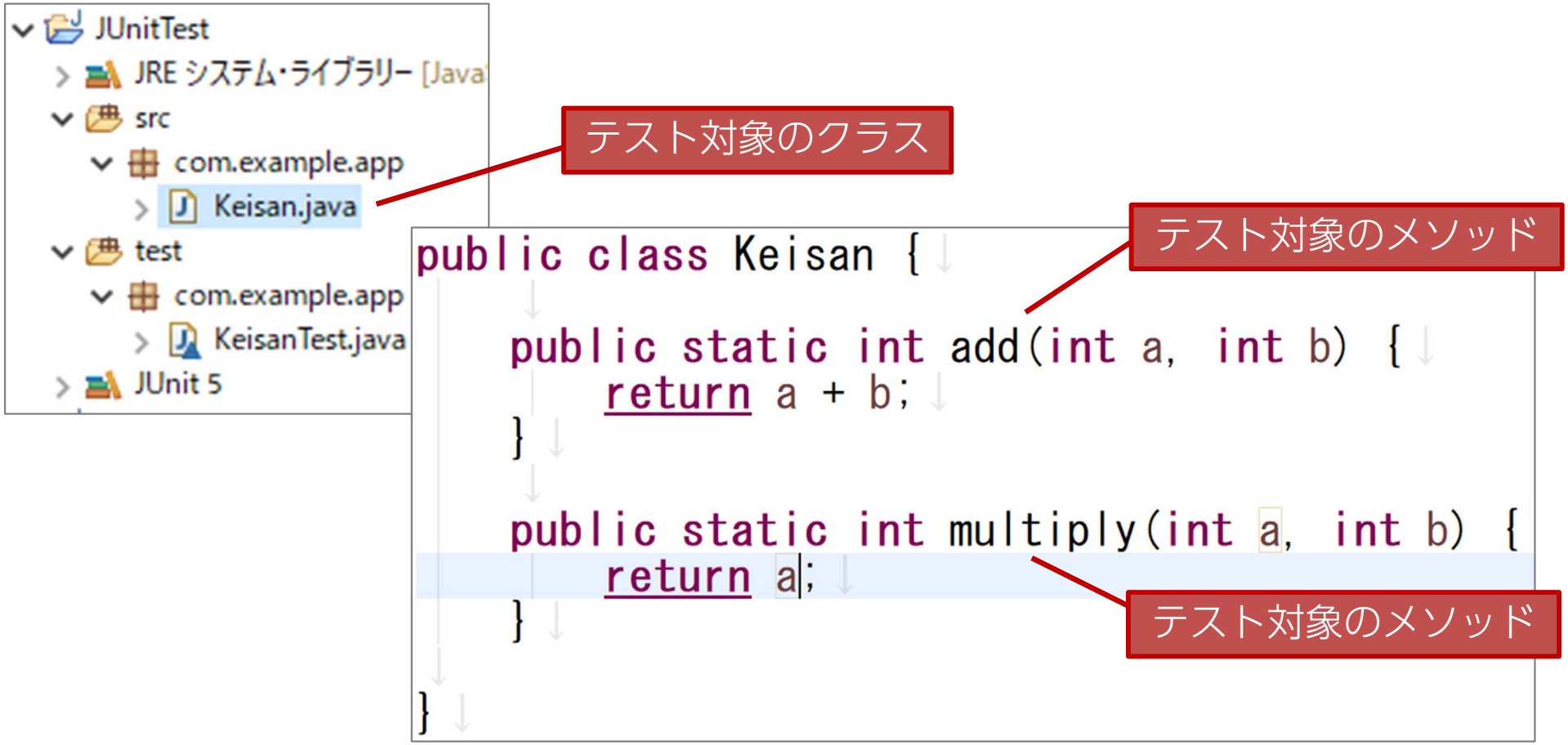
```
@Test
void testAdd() {
    int actual = Keisan.add(3, 5); // 実際の結果
    int expected = 8; // 予期される結果
    assertEquals(expected, actual); // 予想と実際の比較
}

@Test
void testMultiply() {
    int actual = Keisan.multiply(3, 5); // 実際の結果
    int expected = 15; // 予期される結果
    assertEquals(expected, actual); // 予想と実際の比較
}
```

multiply() をテストするメソッド

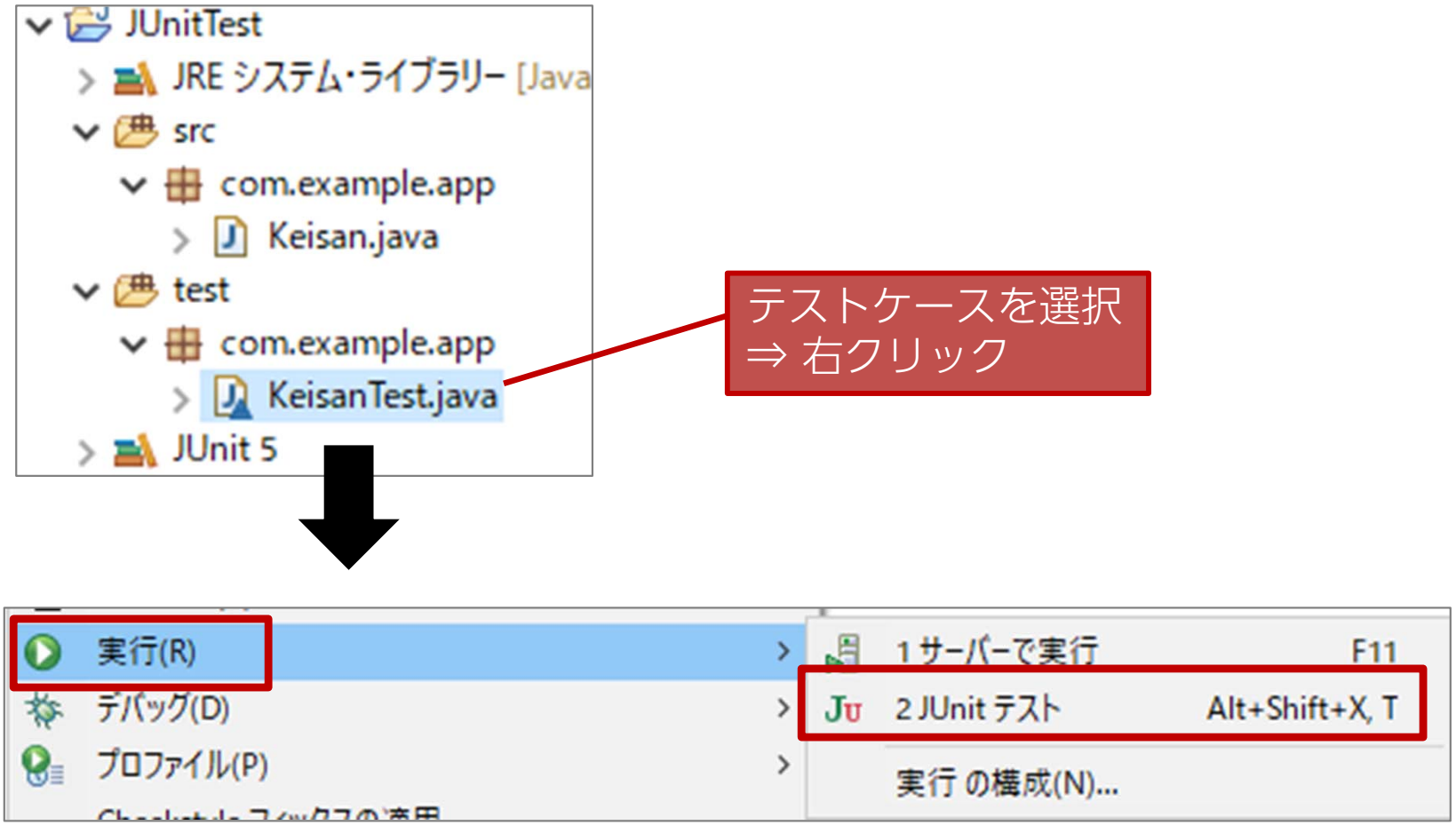
8. テスト対象の実装

- テスト対象のメソッドを実装する



9. テストの実行

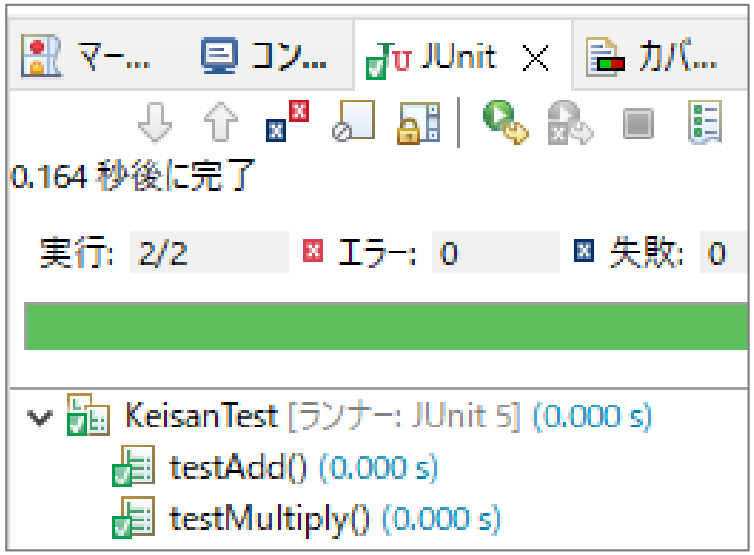
- JUnitテストを実行する



10. テストの実行結果

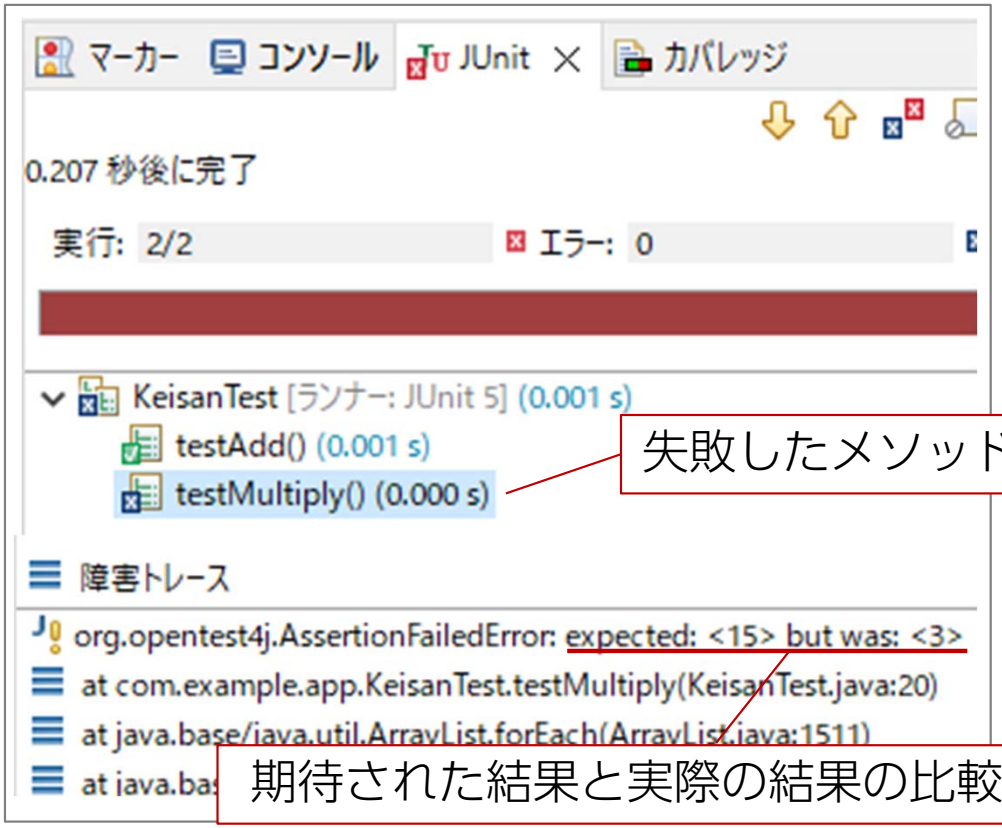
- 緑色のバーが表示されれば成功、赤色のバーが表示されれば失敗となる

成功した場合の表示例



テストが成功するように
テスト対象を実装する

失敗した場合の表示例



練習

- 練習33-1

テスト用アノテーションとメソッド

テスト用アノテーション

アノテーション	説明
`@Test`	「実行⇒JUnitテスト」で実行したいメソッドに付与する
`@DisplayName`	クラスやメソッドに付与することで、テスト結果の表示を変えることができる 
`@TestMethodOrder`	クラスに付与することで、テストメソッドの実行順を制御することができる
`@Order`	`@TestMethodOrder`と併せて利用する。各テストメソッドに付与することで、実行順を制御することができる

記述例：テスト用アノテーション

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("計算クラステスト")
class KeisanTest {

        @Test
        @DisplayName("足し算テスト")
        @Order(2)
        void testAdd() { ... }

        @Test
        @DisplayName("掛け算テスト")
        @Order(1)
        void testMultiply() { ... }

}
```

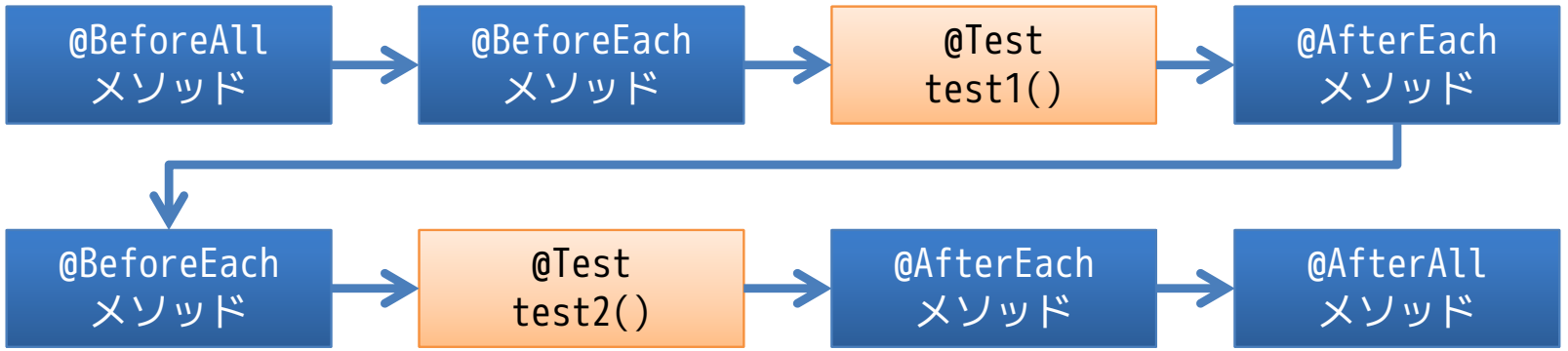
以下を指定することもできる
- **DisplayName**
- **MethodName**
- **Random**

番号の昇順で
実行される

テスト用アノテーション

アノテーション	説明
@BeforeAll	テストケースの最初に呼び出したいstaticメソッドに付与する ファイル等のリソースの読み込みに利用する
@AfterAll	テストケースの最後に呼び出したいstaticメソッドに付与する リソースのクローズに利用する
@BeforeEach	各テスト用メソッドの実行前に呼び出すメソッドに付与する
@AfterEach	各テスト用メソッドの実行後に呼び出すメソッドに付与する

呼び出しの順番



記述例：テスト用アノテーション

```
class KeisanTest {

        int num1;
        int num2;

        @BeforeEach ----------------------
        void setNumbers() {
          num1 = 3;
          num2 = 5;
        }

        @Test
        void testAdd() { ... }

        @Test
        void testMultiply() { ... }

}
```

以下の順で実行されるようになる
1. setNumbers()
2. testAdd()
3. setNumbers()
4. testMultiply()

テスト用メソッド

- テスト用のメソッドには、`@Test`というアノテーションを付与する
- アクセス修飾子は付けなくてよい
  - 以前のバージョンのJUnitではpublicが必要
- メソッド名は任意(日本語が含まれていてもよい)
  - どのようなことをテストしているのか、メソッド名からわかるようにするのが望ましい
- メソッド内では、テスト対象のメソッドが想定通りの結果を返すか確認をする
  - **アサーションメソッド**を使い、確認をする

アサーションメソッド

- テスト対象のメソッドが期待通りに動いているかを検証するためのメソッド
- JUnitでは、以下のようなアサーションメソッドが用意されている

メソッド	役割
`assertEquals(A, B)` `assertNotEquals(A, B)`	予期される結果「A」と実際の結果「B」が等しいこと、または等しくないことを検証する
`assertTrue(A)` `assertFalse(A)`	引数が`true`であること、もしくは`false`であることを検証する
`assertNull(A)` `assertNotNull(A)`	引数が`null`であること、または`null`でないことを検証する
`assertThrows(A, B)` `assertDoesNotThrow(B)`	Bで定義された処理が実行されると、Aという種類の例が投げられること、または例外が投げられないことを検証する

記述例：アサーションメソッド

```
@Test
void testMultiply() {
    int result = Keisan.multiply(5, 0);
    assertEquals(0, result);
    assertNotEquals(5, result);
}

@Test
void testIsNegativeNumber() {
    assertTrue(Keisan.isNegativeNumber(-10));
    assertFalse(Keisan.isNegativeNumber(10));
}
```

Keisan.multiply(a, b)
は掛け算 a x b を行うメソッドとする

Keisan.isNegativeNumber(x)
はxが負の数の場合trueを返すメソッドとする

記述例：アサーションメソッド

```
@Test
void testDivideByZero() {
    Exception e = assertThrows(Exception.class, () -> {
            Keisan.divide(10, 0);
    });

    assertEquals("0で割ることはできません", e.getMessage());
}
```

Keisan.divide(a, b)は
bが0の場合、例外を投げるメソッドとする

- assertThrowsの第二引数 **() -> { }** は**ラムダ式**と呼ばれるもので、
  { }内に複数のステートメントを記述できる

練習

- 練習33-2

@ParameterizedTest

- 異なるテスト用パラメータのセットで、同一のテストを複数回実行したい場合に役立つアノテーション
  - @Testの代わりに記述する
- パラメータのソースを指定するためのアノテーションと併せて利用する

以下のパラメータセットでtestAddを実行する
１回目: 3, 5, 8
２回目: 4, -3, 1

```
@ParameterizedTest
@CsvSource({
        "3, 5, 8",
        "4, -3, 1",
})
void testAdd(int num1, int num2, int expected) {
        int actual = Keisan.add(num1, num2);
        assertEquals(expected, actual);
}
```

パラメータのソース指定

- 以下のようなアノテーションを使い、パラメータのソースを指定することができる

アノテーション	説明
@CsvSource	指定したCSVデータをパラメータとして読み込む
@CsvFileSource	別途作成したCSVファイルから、パラメータを読み込む
@ValueSource	指定した数値や文字列の配列をパラメータとして読み込む
@MethodSource	別途定義したメソッドを使い、パラメータを読み込む
@EnumSource	別途定義したEnumから、パラメータを読み込む

@CsvSource

- パラメータをカンマ区切りの文字列データで指定することができる
  - `@CsvSource("山田太郎, 25, true")`
- 複数のCSVデータを指定する場合は、配列にする
  - `@CsvSource({"山田太郎, 25, true", "鈴木次郎, 30, false"})`
- テスト用メソッドに引数を設定することで、メソッド内でパラメータを利用できるようになる

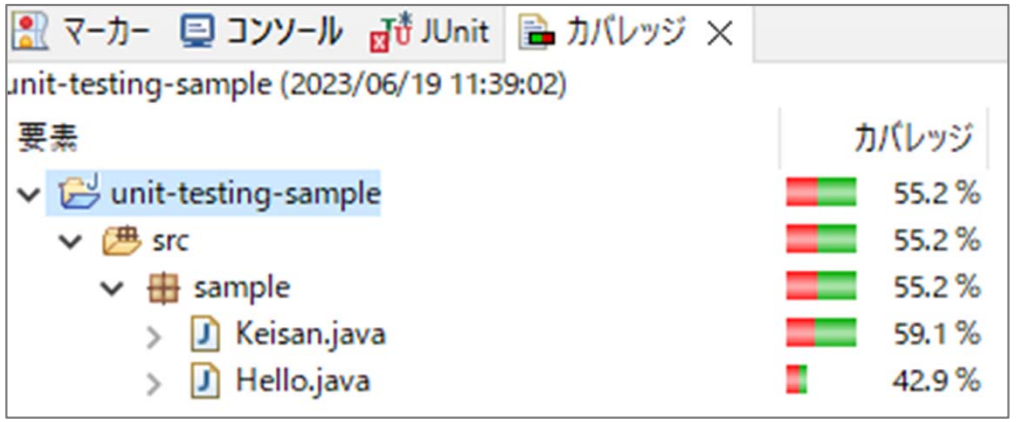
```
@ParameterizedTest
@CsvSource({
        "山田太郎, 25, true",
        "鈴木次郎, 30, false"
})
void testMethod(String name, int age, boolean isStudent) { ... }
```

練習

- 練習33-3

コードカバレッジ

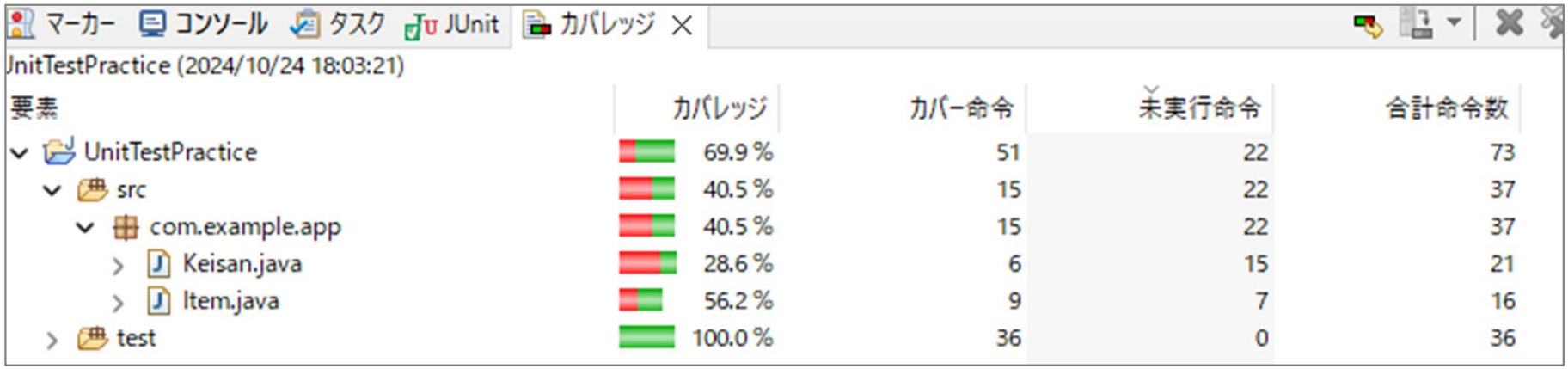
- ソースコードに対するテストの割合を示すもの
- 100%の達成は現実的ではない
  → 70%～80%を目指す
- プログラムの品質を測る指標の一つとなる
  - 質の良くないテストコードであっても、コードカバレッジを高める要因になり得るので、あくまでも指標の一部として捉える



コードカバレッジの確認

- プロジェクト上で右クリックし、「カバレッジ→Junit テスト」

- テストが実行され、カバレッジが表示される



要素	カバレッジ	カバー命令	未実行命令	合計命令数
UnitTestPractice	69.9 %	51	22	73
src	40.5 %	15	22	37
com.example.app	40.5 %	15	22	37
Keisan.java	28.6 %	6	15	21
Item.java	56.2 %	9	7	16
test	100.0 %	36	0	36

コードカバレッジの確認

- ソースコードを見ると、テストされている部分を確認することができる

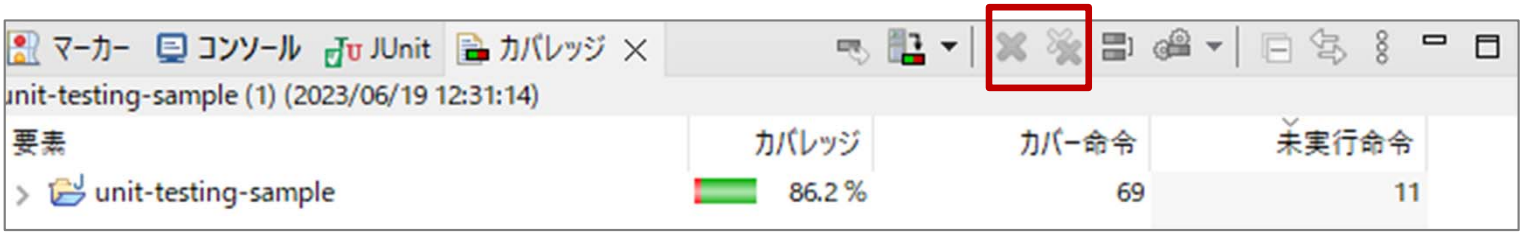
```
 5  public class Keisan {
 6
 7      public int add(int ...numbers) {
 8          if(numbers.length == 0) return 0;
 9          return Arrays.stream(numbers)
10                  .reduce((accum, num) -> accum + num)
11                  .getAsInt();
12      }
13
14      public double getAverage(int ...numbers) {
15          return ((double) add(numbers)) / numbers.length;
16      }
```

この分岐はテストされていない

テストされている

テストされていない

  - ソースコードについた背景色はカバレッジパネルで消すことができる



テスト駆動開発

- 以下のステップを繰り返しながら開発する手法をテスト駆動開発 (TDD：Test Driven Development)と呼ぶ

1. 新しく追加する機能に対するテストを作成する
2. そのテストが成功するような最低限の実装を行う (レッド ⇒ グリーンになるようにする)
3. 必要に応じてテストが通る状態のままで内部構造の改善を行う（リファクタリング)

テストの分類

- テストはブラックボックステストとホワイトボックステストの２種類に分類できる
- ブラックボックステスト
  - テストする対象の内部構造が分からない状態で行う
  - 業務知識 (出力されるものが予期する結果か判定できる知識) をもつ人間が行う
- ホワイトボックステスト
  - テストする対象の内部構造が分かっている状態で行う
  - コードを理解しているプログラマが行う

テスト用入力値の選択

- 全ての入力値の組み合わせでテストを行うのは非現実的
- **同値分割、境界値分析**などを行ってテストを実施する入力値の組み合わせ（テストパターン）を決める

- 同値分割
  - 出力が同じになる入力の組み合わせを１つのグループとみなし、各グループから代表値を選んでテストデータとする手法
- 境界値分析
  - 同値分割で分けたグループ同士の境界やその付近の値を入力データとする手法

テスト用入力値選択の例

例：鉄道料金算出プログラムのテストを行う
⇒ 12歳未満では子供料金、 12歳以上では普通料金が適用される
0歳～130歳までに対応している

0歳～130歳まで、すべての値をテストするのは時間がかかり過ぎる

⬇

同値分割：
12歳未満の値から少なくとも１つ、12歳以上の値から少なくとも１つの値をテストデータとする

⬇

境界値分析：
12歳というのが分割の境界となっているので、「11, 12」をテストデータとする

練習

- 練習33-4