

# オブジェクト指向とデザインパターン 基礎編 ドラフト版



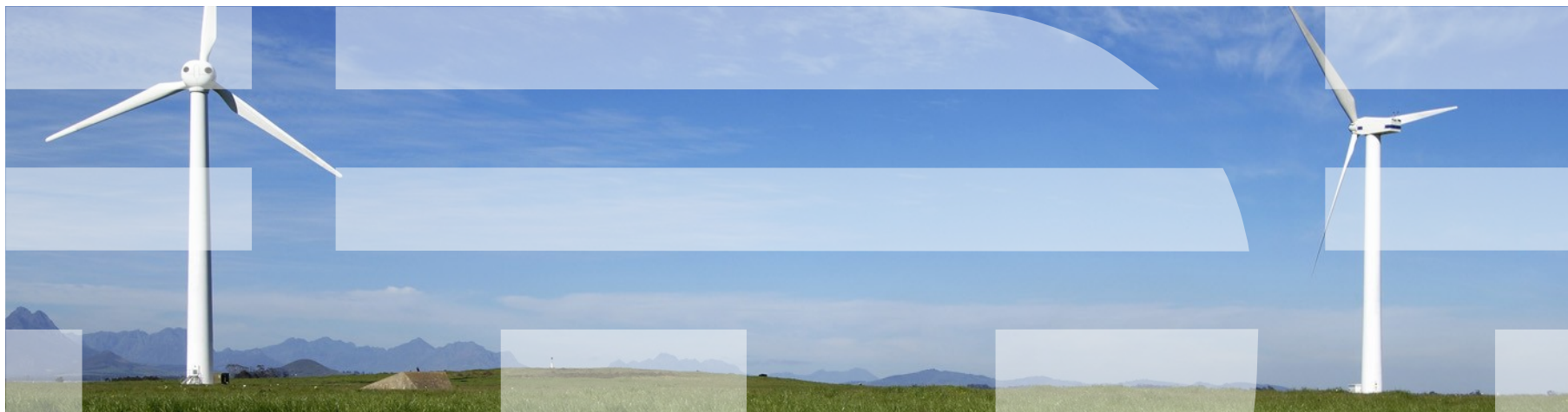
## 目標

- 背景
- 応用編と合わせて、9日コース

---

目次

# 概要



# オブジェクトとデザインパターン

- オブジェクト指向の歴史
- オブジェクト指向の要件
  - カプセル化
  - 継承
  - 多態
- クラス

## オブジェクトとデザインパターン

- デザインパターン

- 『オブジェクト指向における再利用のためのデザインパターン』

- Gang of Four 著

- (エーリヒ・ガンマ、リチャード・ヘルム、ラルフ・ジョンソン、ジョン・ブリシディース)

- 構成

- 生成パターン
  - 構造パターン
  - 振舞パターン

- 上流設計ではなく、コードに密接に関係した実装テクニック。

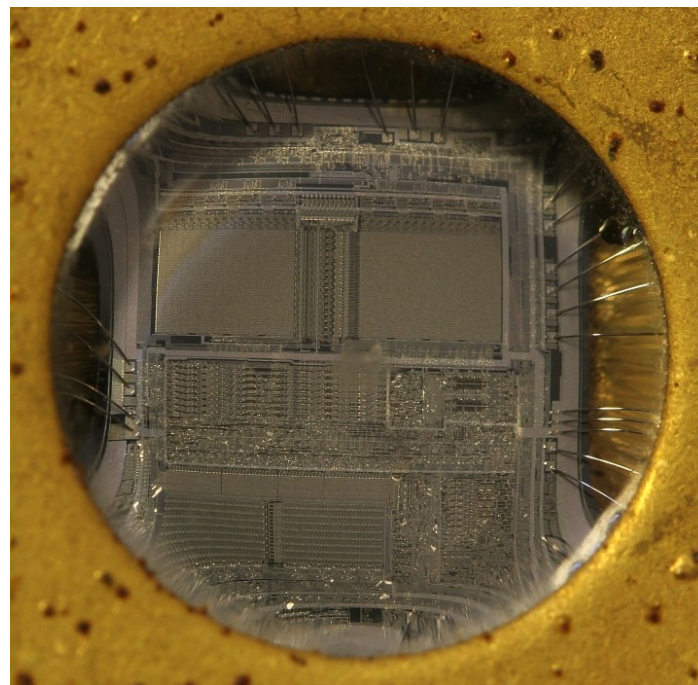
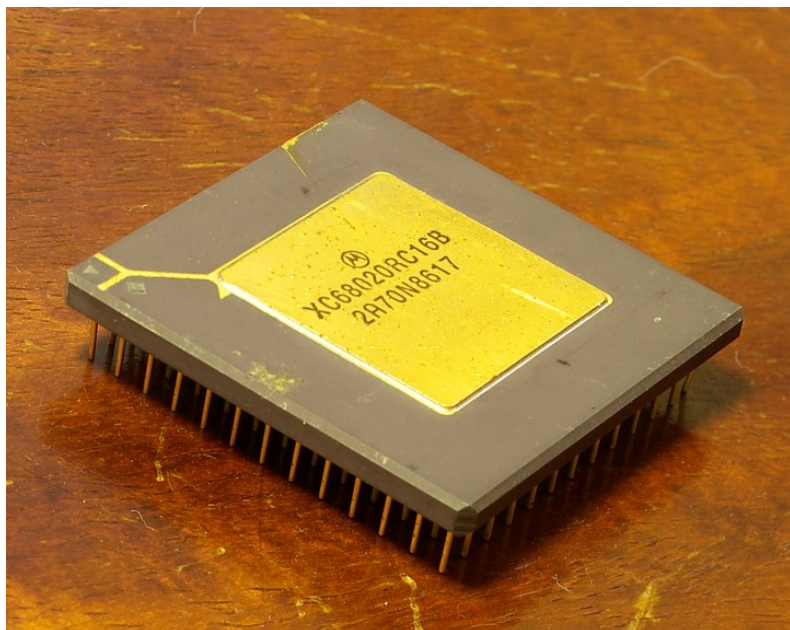
- 共通の語彙を提供する。

# 手続型による回路シミュレータ



## 回路シミュレータ

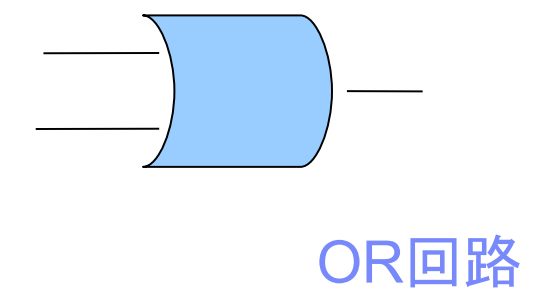
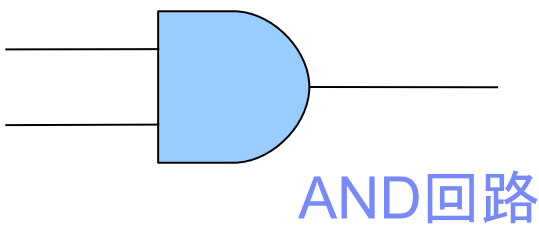
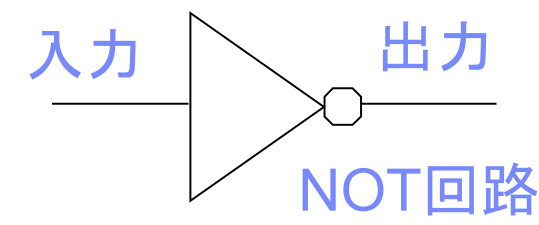
- コンピュータの中には論理演算回路が無数にあり、これらを用いて演算を行っています。
- 個々の論理演算回路は単純なもので、これらを組み合わせると複雑な演算が可能になります。
- オブジェクト指向による設計は、このように単純な部品を組合せて複雑な処理を行う場合に向いています。
- ここでは非常に簡単な回路シミュレータを作成することでオブジェクト指向の基礎を学びましょう。





# 論理演算素子

- 基本的な論理演算素子を以下に示します。



## 真理値表

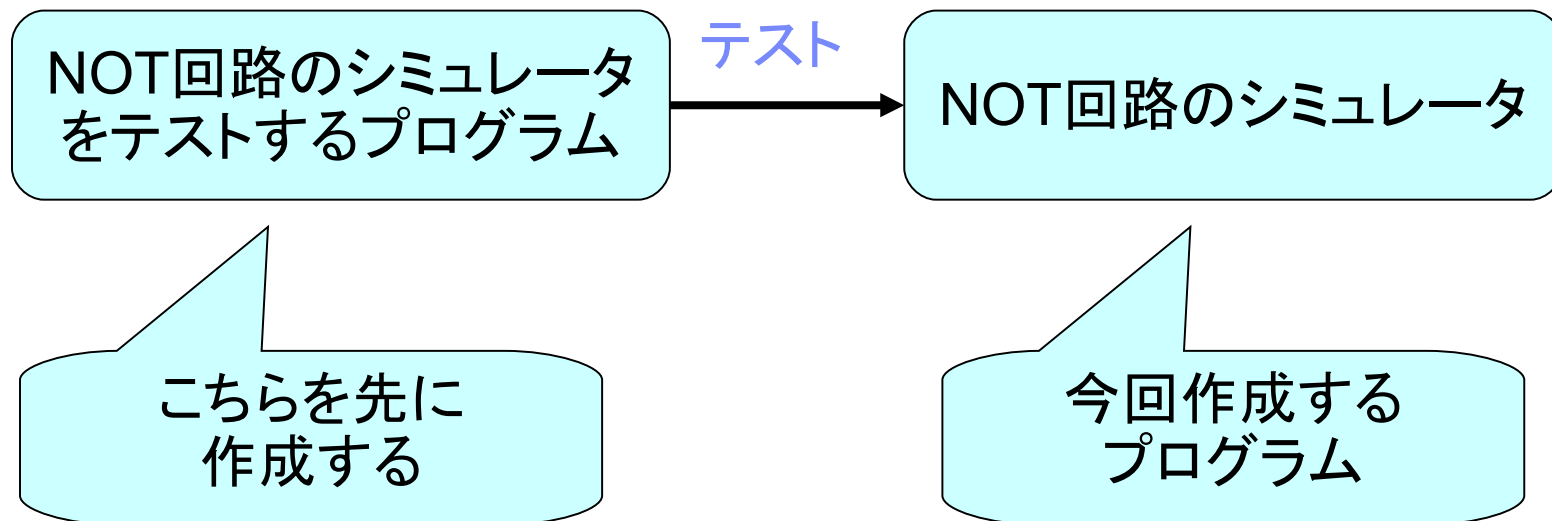
入力		出力
0		1
1		0

入力1	入力2	出力
0	0	0
0	1	0
1	0	0
1	1	1

入力1	入力2	出力
0	0	0
0	1	1
1	0	1
1	1	1

## テスト駆動開発

- このコースでのプログラムは、テスト駆動開発で作成していきます。
- テスト駆動開発では、プログラムを開発する前に、そのプログラムをテストするプログラムを作ります。

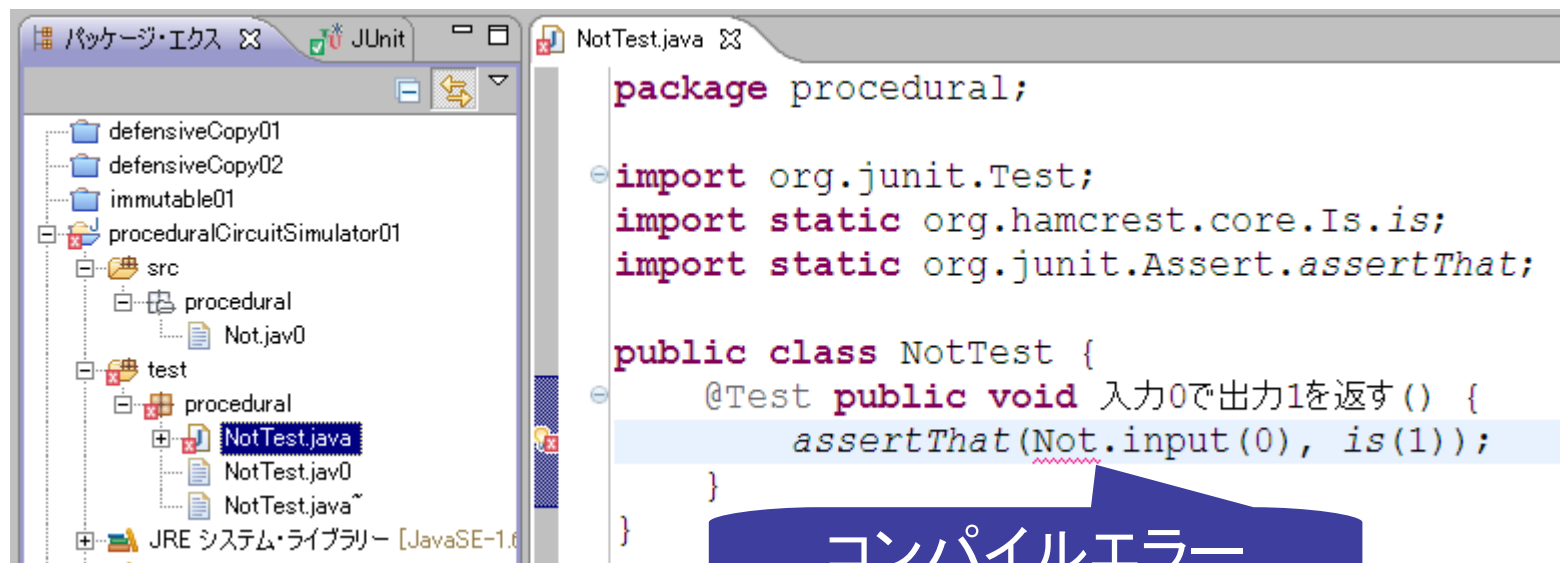


## ペアプログラミング

- このコースでは、ペアを組んでプログラムを作成していきます。
- 1つのPCを2人で囲んで作業をしてください。
- 1人がPCを操作し(ドライバー)、もう1人はそれを見ながら間違いを指摘したり、サジェッションをしたりしてください(ナビゲーター)。
- ドライバーの人は、あまり急がずに。
- ナビゲーターの人は、きちんと作業内容に付いていくように。任せっきりにしない。
- 役割は30分を目安に必ず交代してください。
- 日が変わったら、違うパートナーと組んでください。
- 1時間を目安に休憩をとってください。
- 次のことを意識してください。
  - 仮に相方がいなくなっても自分独力で、今回のコードが修正できること。
  - PCを操作している人は、相方を置いて突っ走らないこと。
  - 操作を見ている人は、少しでも理解できないことがあったら相方に聞いてきちんと理解すること。またなぜ理解しにくかったのか相方に伝えること。自分自身がそういう理解しにくいコードを書いていないか思い返してみること。
  - 聞かれた人は、自分のコードは果たして他人から見て分かりやすいものだったのかどうか、自問自答すること。
  - 自分の知らないPCの操作を相方がやったら、教えてもらって自分の糧にすること。

## 回路シミュレータを手続型で作成する

- まずオブジェクト指向を使わずにNOT回路のシミュレータを書いてみます。
- proceduralCircuitSimulator01プロジェクトを開いてください。
- 既にテストを1つ作成してあります。
- NOT回路のテストは、NotTestという名前になっています。
- メソッド名はテストで検証する内容になっています「入力0で出力1を返す」。
- まだテストしかないので、コンパイルエラーになっています。



## 単体テスト

- JUnitというツールを用いて単体テストを実行します。

```
package procedural;  
  
+ import org.junit.Test;  
  
public class NotTest {  
-     @Test public void 入力0で出力1を返す () {  
        assertThat(Not.input(0), is(1));  
    }  
}
```

クラス名  
は、xxxTestとする

メソッド名に、テスト  
で検証する内容を  
示す名前を付ける。

テストはメソッドの中  
に記述し、メソッドに  
@Testを付ける。

assertThatは、2つ  
の値が等しいことを  
検証する。

## プロジェクトの構成

proceduralCircuitSimulator01

├── bin

│ └── procedural

│ NotTest.class

├── src

│ └── procedural

│ Not.java

└── test

│ └── procedural

│ NotTest.java

コンパイルされたクラスファイル  
が格納されます。

ソースコードが格納されます。

単体テストが格納されます。

- src.org、test.orgには修正前のファイルが入っています。
- src.answer、test.answerには解答例が入っています。

## テスト駆動開発

### ■ 手順1

- 新たな機能が必要になったら、テストを作成する。  
テストの中では、新たな機能の実現のために必要となる、新しいクラスやメソッドを使用するかもしれない。

今回はテストは作成済みです。

### ■ 手順2

- コンパイルエラーがある場合は解決する。

ここから開始します。

## 回路シミュレータを手続型で作成する

- それでは開発を始めましょう。エラーを解決するためエラー行の左の電球をクリックします。
- メニューが現れます。
- 今回はNotクラスを作成したいので「クラス 'Not' を作成します。」をダブルクリックします。

NotTest.java

```
package procedural;

import org.junit.Test;
import static org.hamcrest.core.Is.is;
import static org.junit.Assert.assertThat;

public class NotTest {
    @Test public void 入力0で出力1を返す() {
        assertThat(Not.input(0), is(1));
    }
}
```

1 クリック

ダブルクリック

2

新規クラス・ウィザードを開いて型を作成します。

パッケージ: procedural  
public class Not {  
}

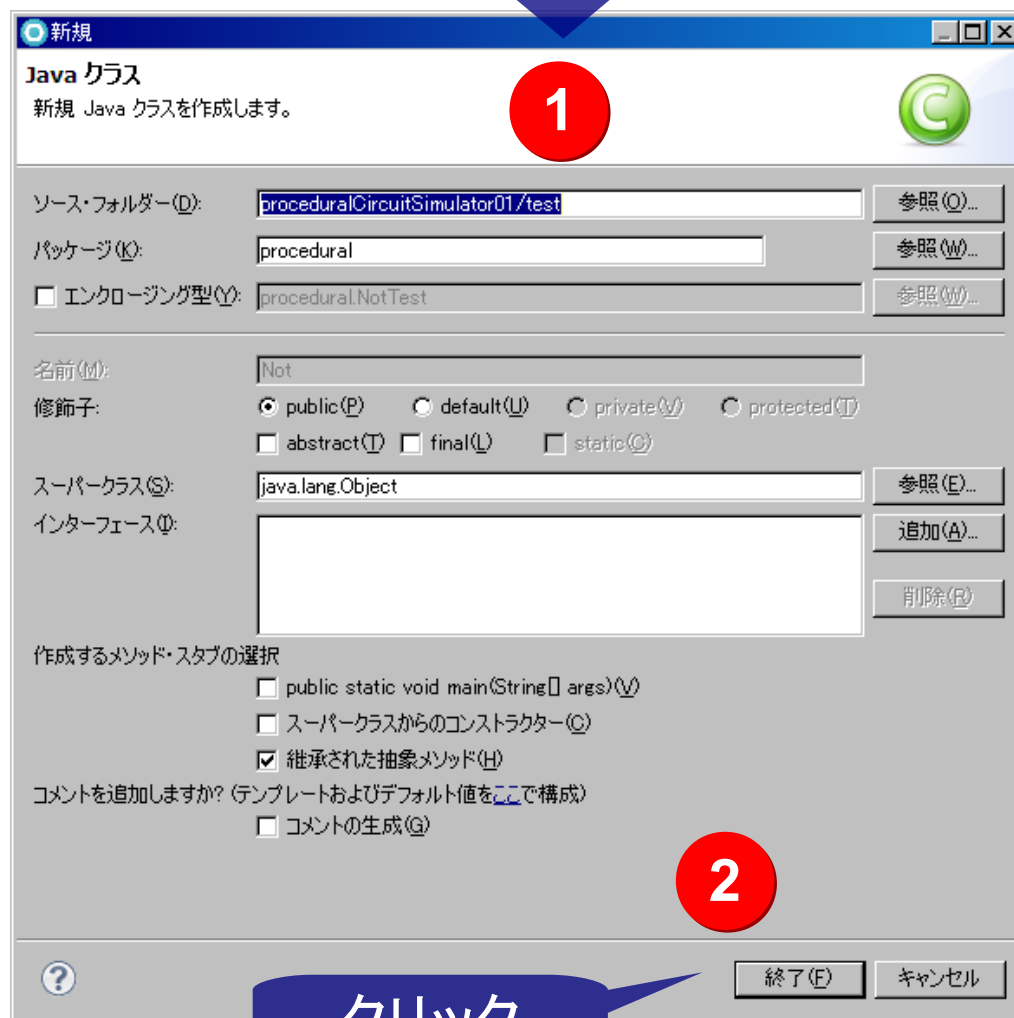
プロポーザル・テーブルから「タブ」を押すか、クリックしてフォー



## 回路シミュレータを手続型で作成する

- ダイアログが表示されます。
- デフォルトだとソースの場所がtestの下になっているので、これをsrcの下に変更します。
- 終了を押します。

testをsrcに変更

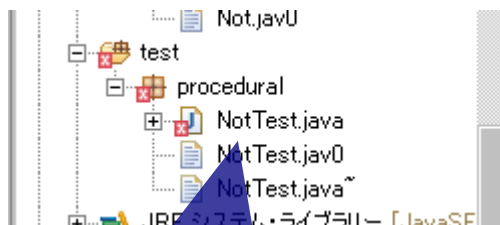


クリック

## 回路シミュレータを手続型で作成する

- Finishボタンを押すとNotクラスが作成されます。
- まだエラーがあるので、NotTestに戻ります。

### 戻り方1



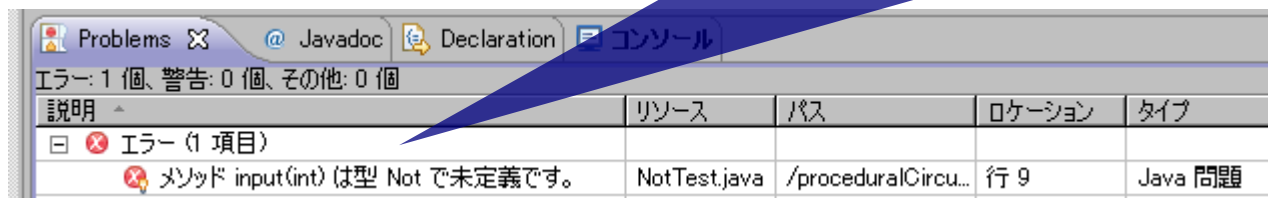
NotTest.javaをダブルクリック

### 戻り方2

Alt + ←を押す

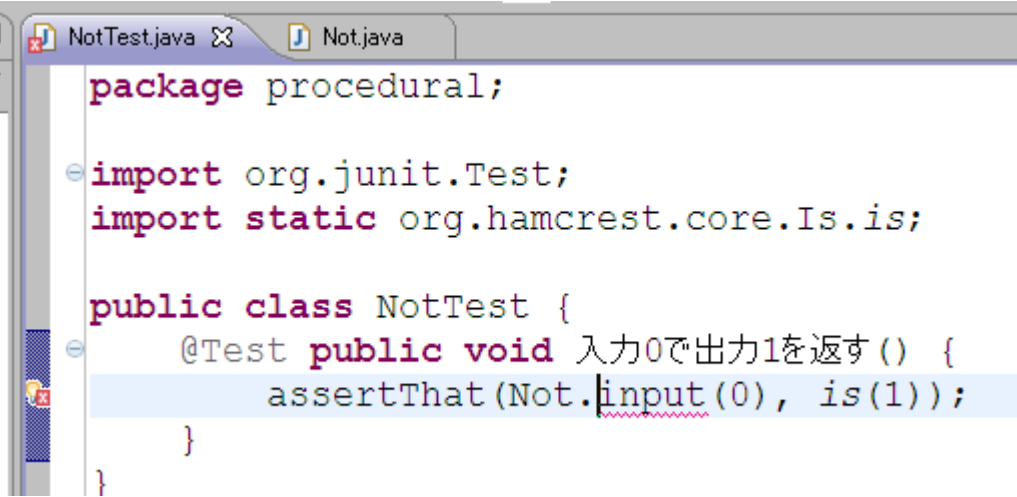
### 戻り方3

Problemsビューで項目をダブルクリック



## 回路シミュレータを手続型で作成する

- Input()メソッドをまだ作成していないので、エラーになっています。



The screenshot shows a Java IDE with two tabs: 'NotTest.java' and 'Not.java'. The 'NotTest.java' tab is active, displaying the following code:

```
package procedural;

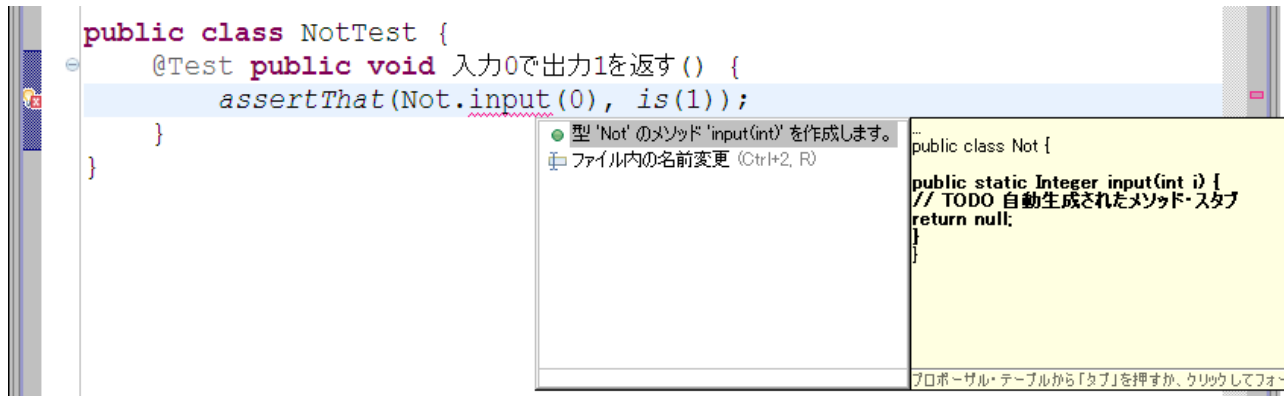
import org.junit.Test;
import static org.hamcrest.core.Is.is;

public class NotTest {
    @Test public void 入力0で出力1を返す() {
        assertThat(Not.input(0), is(1));
    }
}
```

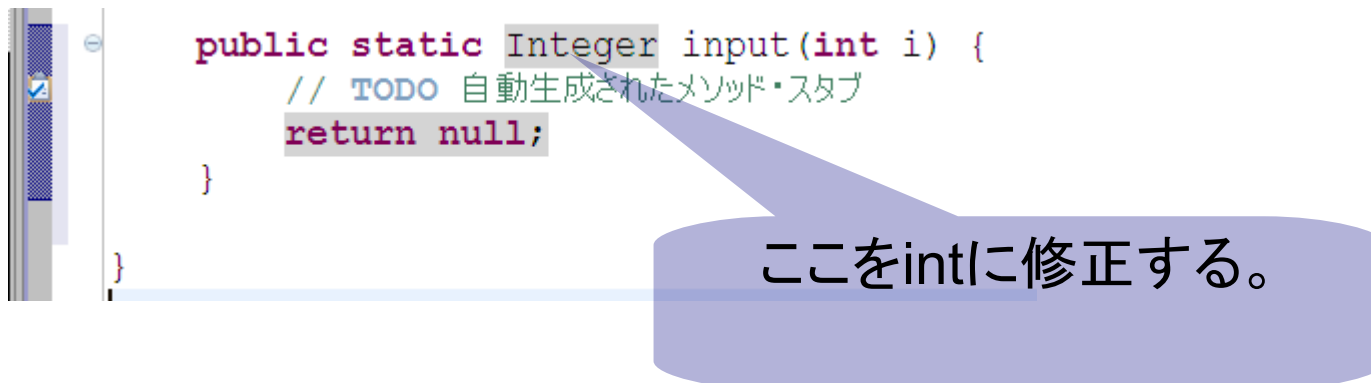
The code is syntactically correct, but the IDE highlights the `input` method call with a red squiggly line, indicating a compilation error because the `input` method has not been implemented in the `Not` class.

## 回路シミュレータを手続型で作成する

- 同じように電球をクリックして、input()メソッドを作成します。




- ただし自動生成されたメソッドは戻り値がIntegerになっているので、intに修正します。



## 回路シミュレータを手続型で作成する


- intに修正したので、return null;がエラーになります。

A screenshot of an IDE's code editor. On the left, a vertical toolbar contains icons for a file explorer, a search icon, and a bug icon. The code editor shows a Java method: 

```
public static int input(int i) {  
    // TODO 自動生成されたメソッド・スタブ  
    return null;  
}
```

 The line `return null;` is highlighted in grey, and the word `null` is underlined with a red wavy line, indicating a compilation error.

- return 0;に変更し、TODOコメントも削除します。

A screenshot of an IDE's code editor. On the left, a vertical toolbar contains icons for a file explorer, a search icon, and a bug icon. The code editor shows the following Java code: 

```
public class Not {  
  
    public static int input(int i) {  
        return 0;  
    }  
}
```

 The line `return 0;` is highlighted in light blue, and the `// TODO` comment has been removed.

これでコンパイルエラーがなくなりました。

## もう一度手順を振り返ってみましょう。

### ■ 手順1

- 新たな機能が必要になったら、テストを作成する。  
テストの中では、新たな機能の実現のために必要となる、新しいクラスやメソッドを使用するかもしれない。

今回はテストは作成済みです。

### ■ 手順2

- コンパイルエラーがある場合は解決する。

ここまでが完了しました。

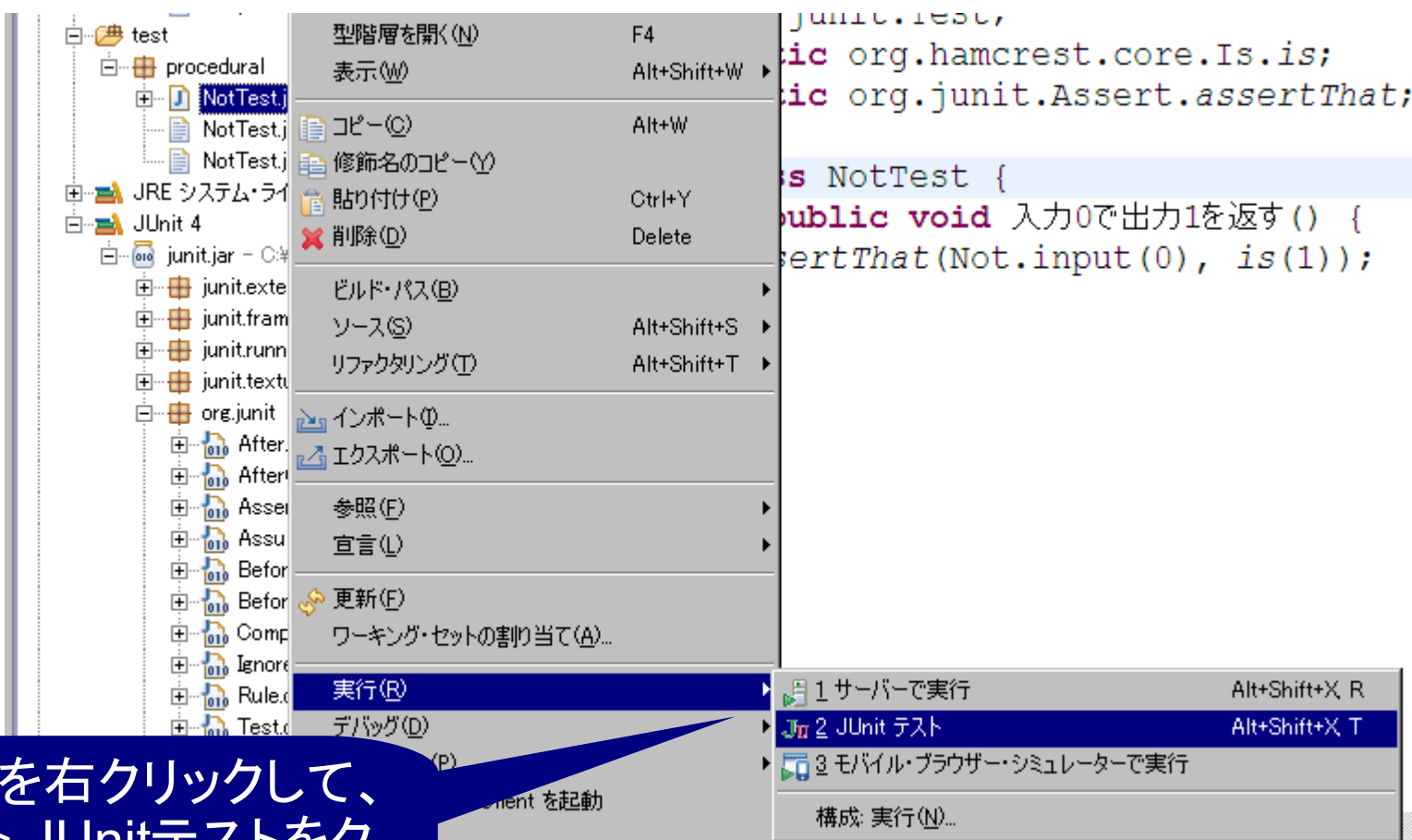
### ■ 手順3

- テストの失敗を確認します。

次の手順。

# 回路シミュレータを手続型で作成する

- それではテストを実行してみましょう。



```
import org.junit.Test;
import org.hamcrest.core.Is.is;
import org.junit.Assert.assertThat;

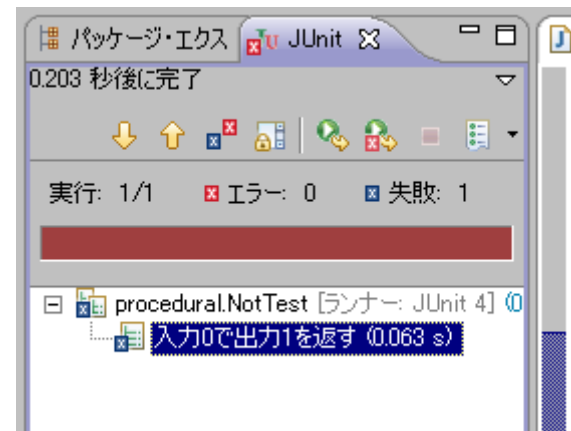
public class NotTest {

    public void 入力0で出力1を返す() {
        assertThat(Not.input(0), is(1));
    }
}
```

NotTestを右クリックして、  
実行 => JUnitテストをク  
リックします。

## 回路シミュレータを手続型で作成する

- テストが失敗すれば成功です(!?)
- テストが失敗すると右のように赤いバーが表示されます。



テストが失敗した原因が分かりますか？

```
public class Not {  
    public static int input(int i) {  
        return 0;  
    }  
}
```

Notクラス

```
public class NotTest {  
    @Test public void 入力0で出力1を返す() {  
        assertThat(Not.input(0), is(1));  
    }  
}
```

テストコード



もう一度手順を振り返ってみましょう。

■ 手順3

－テストの失敗を確認します。

ここまで完了しました。

■ 手順4

－テストが成功するように本体コードを修正します。

次の手順。

## 回路シミュレータを手続型で作成する

- コードを修正します。

```
public class Not {  
    public static int input(int i) {  
        return 1;  
    }  
}
```

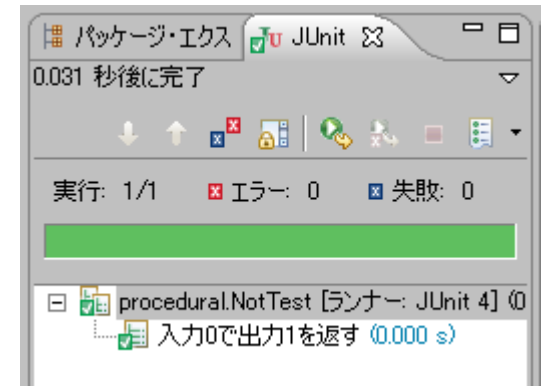
Notクラスを作成する時に、初めからこのように作成してはダメなのでしょうか？

コンパイルエラーを修正するためにクラスやメソッドの雛形を作る場合、なるべくテストが失敗するように作成して、まずテストの失敗を確認します。

これによりテストが、きちんと対象コードをテストできているのかを確認することができます。

## 回路シミュレータを手続型で作成する

- 緑のバーが表示されればテストは成功です。
- 赤のバーになる時は、どこかが間違っています。見直して修正してください。



## 回路シミュレータを手続型で作成する

- ところで今回作成したinput()メソッドは正しいのでしょうか？
- これでは入力にかかわらず、1が返りますね。
- しかしここではまだ、内容を作り込まずに、とにかく目先のテストが成功することを目標にします。

```
public static int input(int i) {  
    return 1;  
}
```


一気に機能を作り込もうとせずに、  
少しずつ漸進的に進めることが大切です。

- 上記のような問題に気付いたらテストを追加します。
- 失敗するテストを作成して、そのテストが成功するようにコードを追加、修正していきます。

テストと実装、これを小刻みに繰り返してください。  
リズムが重要です。

## もう一度手順を振り返ってみましょう。

- 手順1
  - 新たな機能が必要になったら、テストを作成する。
  - テストの中では、新たな機能の実現のために必要となる、新しいクラスやメソッドを使用するかもしれない。
- 手順2
  - コンパイルエラーがある場合は解決する。
- 手順3
  - テストの失敗を確認します。
- 手順4
  - テストが成功するように本体コードを修正します。
- 手順5
  - 必要に応じてリファクタリングします。



小刻みに  
繰り返します



後述します。

## 回路シミュレータを手続型で作成する

- NotTestにテストを追加します。

```
public class NotTest {  
    @Test public void 入力0で出力1を返す () {  
        assertThat(Not.input(0), is(1));  
    }  
  
    @Test public void 入力1で出力0を返す () {  
        assertThat(Not.input(1), is(0));  
    }  
}
```

- テストをもう一度実行しましょう。  
(Ctrl+F10で前回実行したテストを再実行できます)。
- テストはちゃんと失敗したでしょうか？ 常に1しか返さない  
input()メソッドでは、今回のテストは当然通りませんね。



## 回路シミュレータを手続型で作成する

- Notクラスのinput()メソッドを修正します。
- テストを再実行して成功することを確認してください。

```
public static int input(int i) {  
    if (i == 1) return 0;  
    else return 1;  
}
```

## リファクタリングする

- 引数のiというのは通常、ループの変数などに用いる名前では適切ではありません。もう少し意味のある名前に変更しましょう。
- これをリファクタリングと呼びます。
- iの上にカーソルを置いてShift+Alt+Rを押します。

```
public class Not {  
    public static int input(int i) {  
        if (i == 1) return 0;  
        else return 1;  
    }  
}
```

新しい名前を入力し、Enter を押してリファクタリングします ▼

- iを、inputに変更してEnterを押してください。  
iが使用されている場所全てが自動的にinputに変わります。
- 終わったらテストを実行して成功することを確認します。

名前は重要です、常に見直して修正してください。変数名だけでなく、メソッド名、クラス名、どれも同じやり方で修正できます。  
終わったら毎回テストで確認することを忘れずに。



## 回路シミュレータを手続型で作成する

- これで完成でしょうか？
- 引数のintには0、1以外の数値が指定可能です。
- 現在のNotクラスでは0以外は1と見なしています。
- 引数をチェックして0、1以外はエラーにしましょう。
- JavaにはIllegalArgumentExceptionという引数エラーを通知する例外があるので、これを使用します。引数にエラーがある時は、この例外をスローします。
- これまで通りテストを先に作成します。

```
@Test(expected = IllegalArgumentException.class)
public void 入力が0未満はエラー() {
    Not.input(-1);
}
```

- このように、expected = 例外クラスという指定をすると、テスト実行中に指定された例外が発生しないとテスト失敗になります。

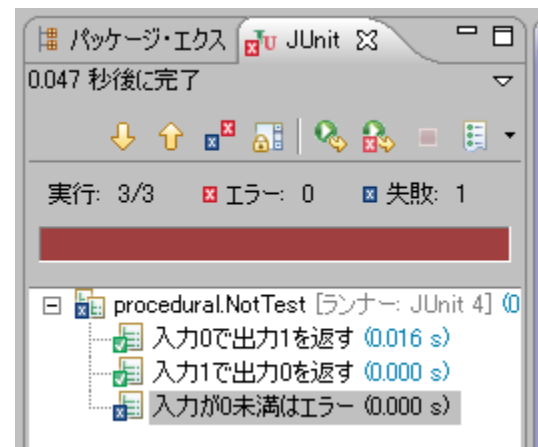
## 回路シミュレータを手続型で作成する

- テストを実行して失敗することを確認します。

- それでは、Notクラスを修正してください。

```
public static int input(int input) {  
    if (input == 1) return 0;  
    else if (input == 0) return 1;  
    else throw new IllegalArgumentException  
        ("値(=" + input + ")は、0か1でなければなりません。");  
}
```

- テストを実行して成功することを確認します。



## 回路シミュレータを手続型で作成する

- 引数エラーで例外を投げる時の注意。  
例外を投げる時には例外クラスにメッセージを渡しますが、その際には必ず受け取る側のことを良く考えてください。以下のことが分かるようにしてください。
  - どんな引数が実際には渡されたのか。
  - どんな引数を渡さなければならなかったのか。
- 以下は実際に表示されるエラー:

Exception in thread "main" java.lang.IllegalArgumentException: 値  
(=2)は、0か1でなければなりません。  
at procedural.Not.input(Not.java:8)  
at procedural.Not.main(Not.java:13)

実際に渡された値

本来渡さなければ  
ならない値

## 回路シミュレータを手続型で作成する

- テストはまだ不十分です。
  - 入力が-1の場合にエラーになることは確認しましたが、0、1が正常値の場合、境界値として2をテストしなければなりません。
- テストを追加してください。

```
@Test(expected = IllegalArgumentException.class)
public void 入力が2以上はエラー() {
    Not.input(2);
}
```

- コードを1文字でも修正したら、もちろんテストです。そろそろ慣れてきましたか？
- テストが成功したら完成です。お疲れ様でした。

## 演習: 手続き型のAND、OR回路シミュレータを作成する。

- これまでの内容を参考に、AND、OR回路のシミュレータを作成してください。
- プロジェクトは、proceduralCircuitSimulator02です。
- 中にはAND用に既にテストが1つだけ作ってあります。

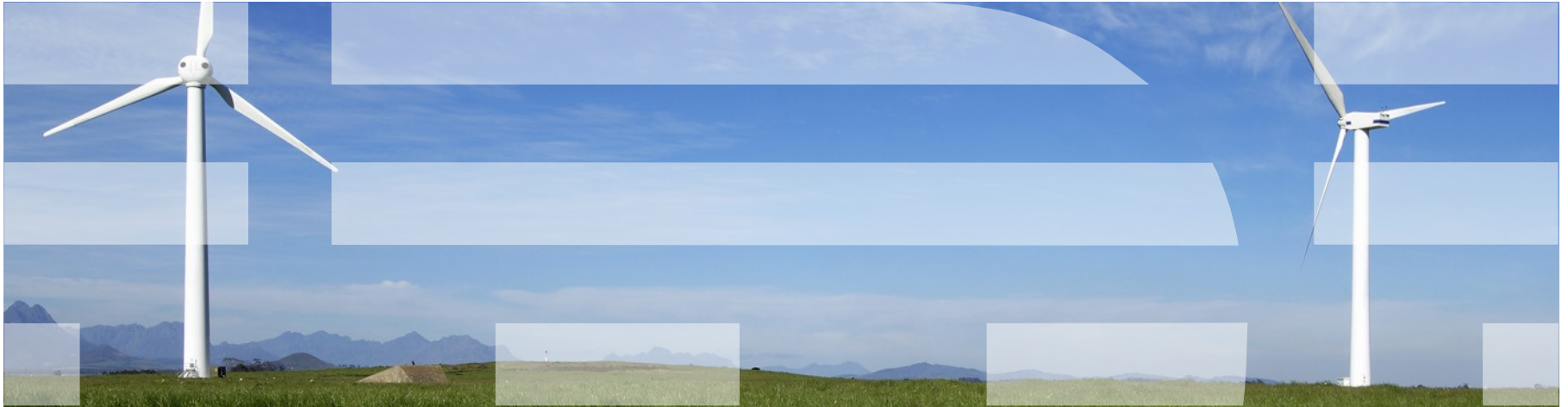
```
@Test public void 入力0と1で出力0を返す() {  
    assertThat(And.input(0, 1), is(0));  
}  
  
// TODO: AND回路の真理値表を参照してテストを追加してください。
```

- これをもとに、AndTestにテストを追加しながら、Andクラスを作成します。
- 同様にOrクラスも作成します。

## まとめ

- TDDとペアプログラミングによって、手続き型で回路シミュレータを作成しました。
- 先にテストを作成し、テストが成功するように本体プログラムを作成していく開発手法をTDD(テスト駆動開発)と呼びます。
- テスト駆動開発では、テストとプログラミングとが不可分です。テストを少しずつ追加しつつ、本体コードも少しずつ開発していきます。
- 開発中は、いきなり完成を目指すのではなく、とりあえず目先のテストを成功させることに集中します。
- 2人が1組となり、1つの端末で開発するやり方をペアプログラミングと呼びます。
- IDE(統合開発環境)を使って、プログラミングする方法を学習しました。
  - コンパイルエラーを頼りにウィザード(電球マーク)を用いて、コードを自動生成することができます。
  - 簡単に単体テストを実行でき、また前回のテストを簡単に再実行できます。
  - リファクタリングによって、簡単に名前を変更できます。
  - コードを修正したらテストを実行することで、いつでもプログラムが正しいことを確認できます。

# TDDとペアプログラミング



## TDD

---

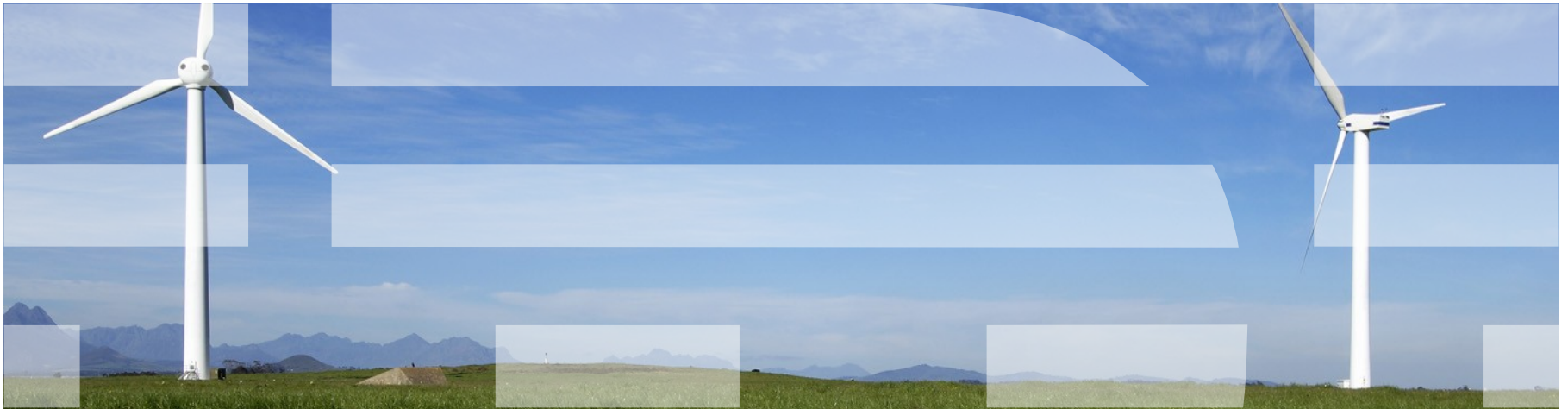
- TODO: TDDの生い立ち、一般的な効用



## ペアプログラミング

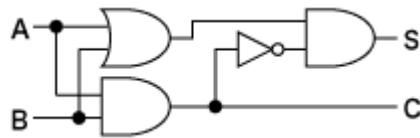
- TODO: ペアプログラミングの生い立ち、一般的な効用

## 手続き型による加算器の作成



## 半加算器

- NOT、AND、OR回路を作成しましたが、これだけだと何の役に立つのか疑問ですね。
- しかし、これらを結線すると色々な演算ができるようになります。
- 今回は、これらを組み合わせて足し算を行う回路を作成します。

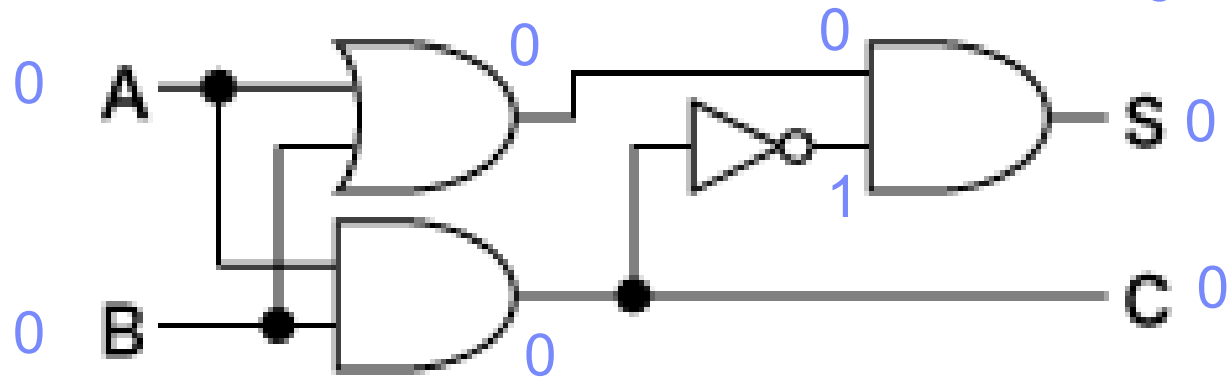


TODO: 図はWikipediaから  
コピペなので描き直す。

加算器

A = 0、B = 0だと？

TODO: 図はWikipediaから  
コピーなので描き直す。



真理値表

入力A	入力B	出力C	出力S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

2進数での、1bit + 1bitの  
結果が得られている。

## 演習: 手続き型の半加算器シミュレータを作成する。

- プロジェクトは、proceduralHalfAdder01です。
- テストは作成済ですが、HalfAdderの一部が虫食いになっています。
- HalfAdderは、出力を複数返すのでintの配列を返すようになっていることに注意してください。
- コメントを見ながら完成させてください。

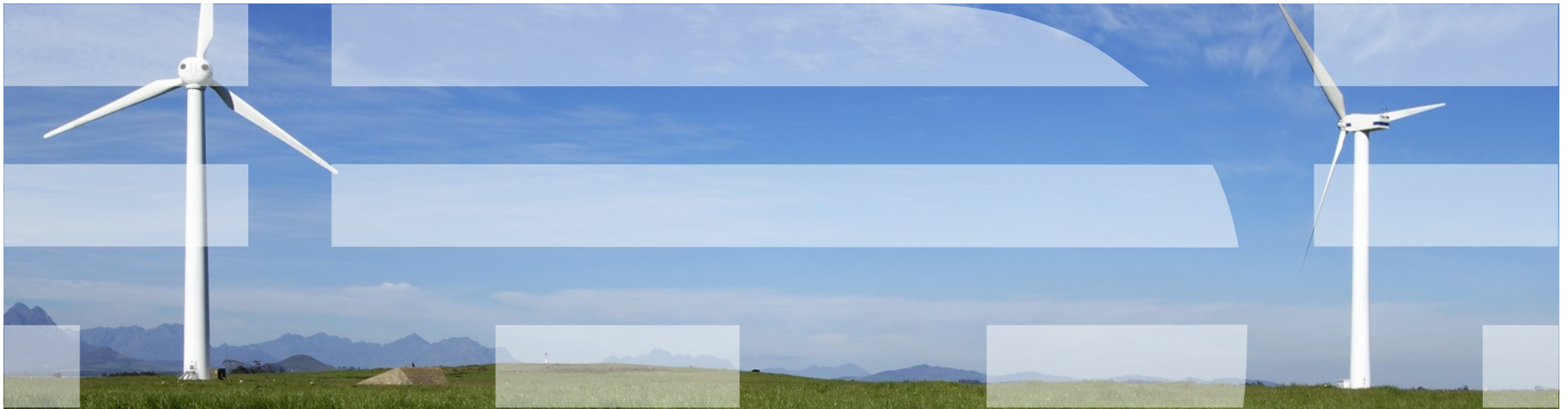
TDDだと、難しいロジックでも安心して実装できることを体感してください。

## これまで作成した手続き型プログラムの課題点

- コード重複
  - 入力チェックが全てのクラスに書かれており、同じコードが何度も繰り返し書かれている。
  - テストコードでも全てのクラスに対して入力チェックのテストを何度も書かなければならない。
- コード重複が多いと
  - コード量が多いので単純に生産性が悪い。
  - もしも引数チェックの方法が変更になった場合、あらゆるところに繰り返し書かれた引数チェックのコード、テストを書き直さなければならない。これは手がかかるだけでなく、修正漏れが発生する危険がある。テストの側で修正漏れが起きると、本体コード側に修正漏れがあっても検出できない。
- 半加算器の実装と、実際の回路図との対応が直感的でない
  - 実際の結線を見ながら、プログラマがプログラミングしていかないといけない。
- intで返したり、intの配列で返したりと演算回路によって型が変わるので、統一的に扱うことができない。

これらの問題をオブジェクト指向によって解決していきましょう。

# オブジェクト指向による回路シミュレータ



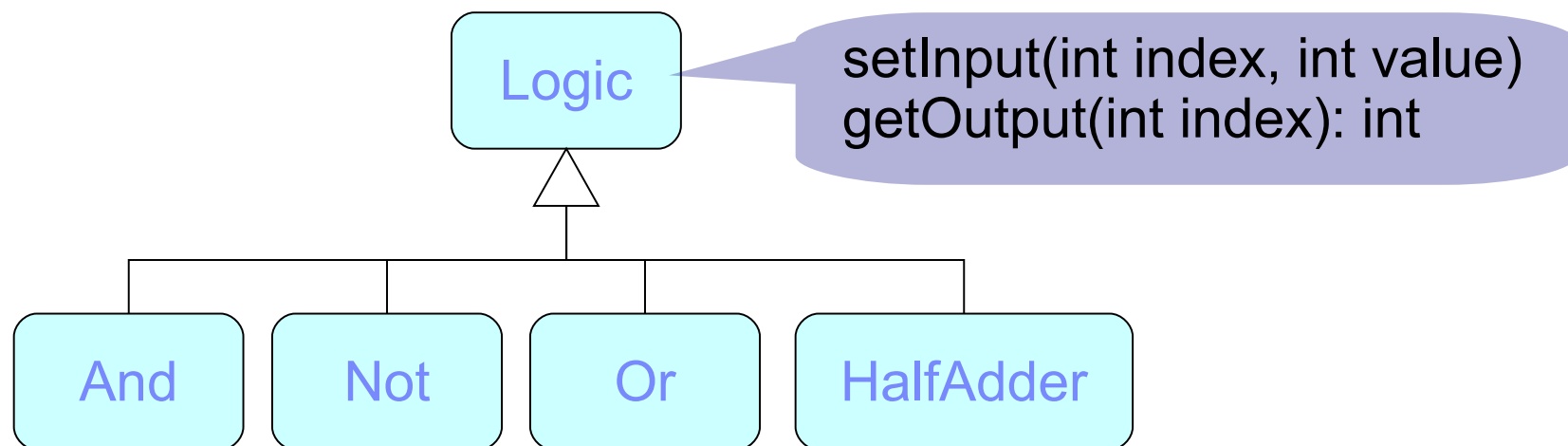
## オブジェクト指向設計

- オブジェクト指向設計ではオブジェクトの発見が、要(かなめ)となります。
- 良くビジネスドメインにおける「名詞」に注目せよというガイドがあります。もちろんこれは大枠では間違っていないのですが、これだけでは、まだまだ粒度が粗すぎます(実際の例をこの後の章で見えます)。
- 何をオブジェクトとして認識するかは、人それぞれで絶対的な解はありません。
- これまでの回路シミュレータを見た時、どんなオブジェクト指向設計ができるでしょうか？



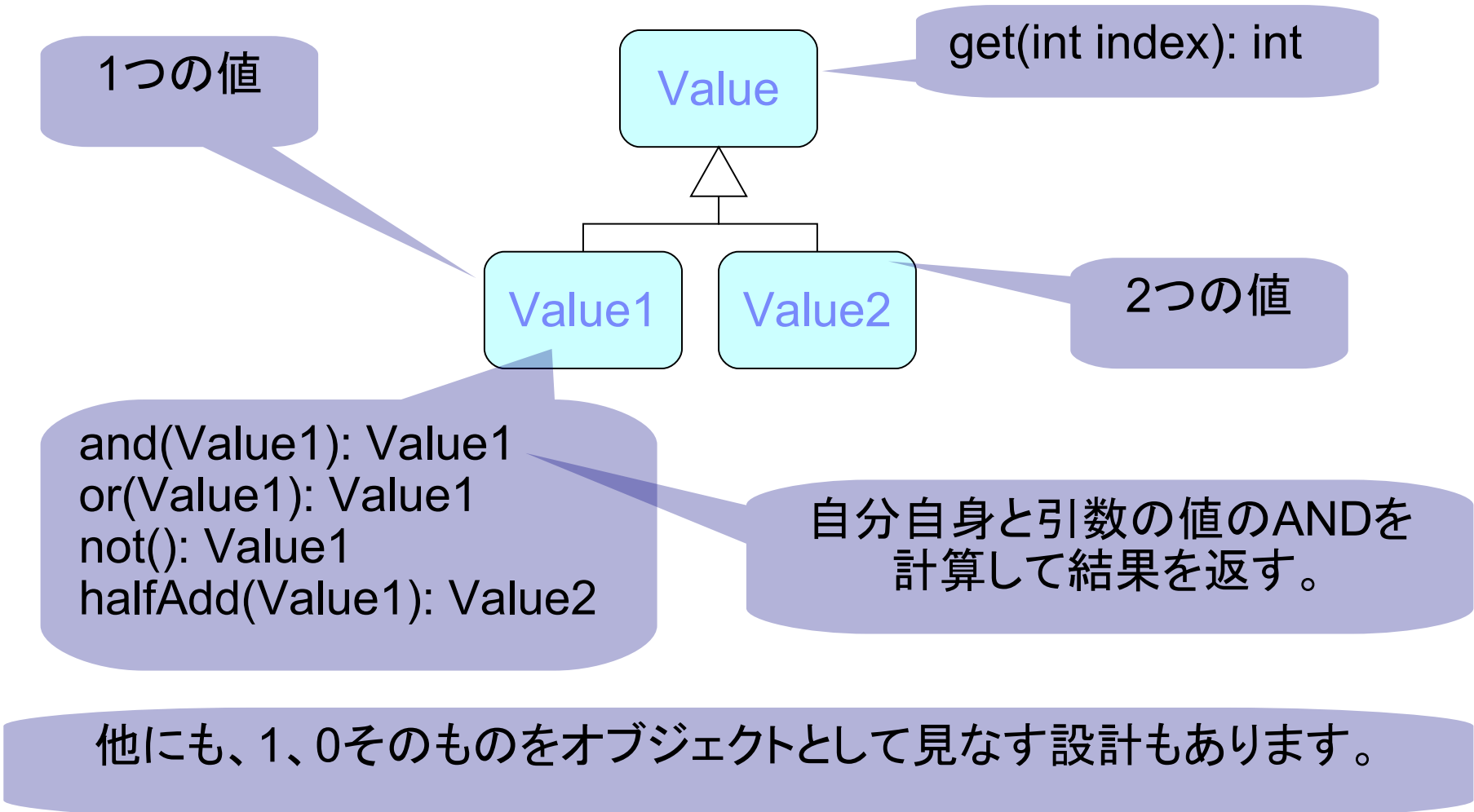
## 設計1

- おそらく最もオーソドックスな設計は、論理素子をオブジェクトと見なすものです。
- この例では、Logicというベースクラスに、入力設定用のsetInput()と、出力取得用のgetOutput()メソッドを用意しているので、どんな論理素子でも、これらの共通のメソッドで入力の設定、出力の取得が可能です。



# 設計2

- 以下の設計では、0とか1の値をオブジェクトとした場合です。



## オブジェクト指向設計

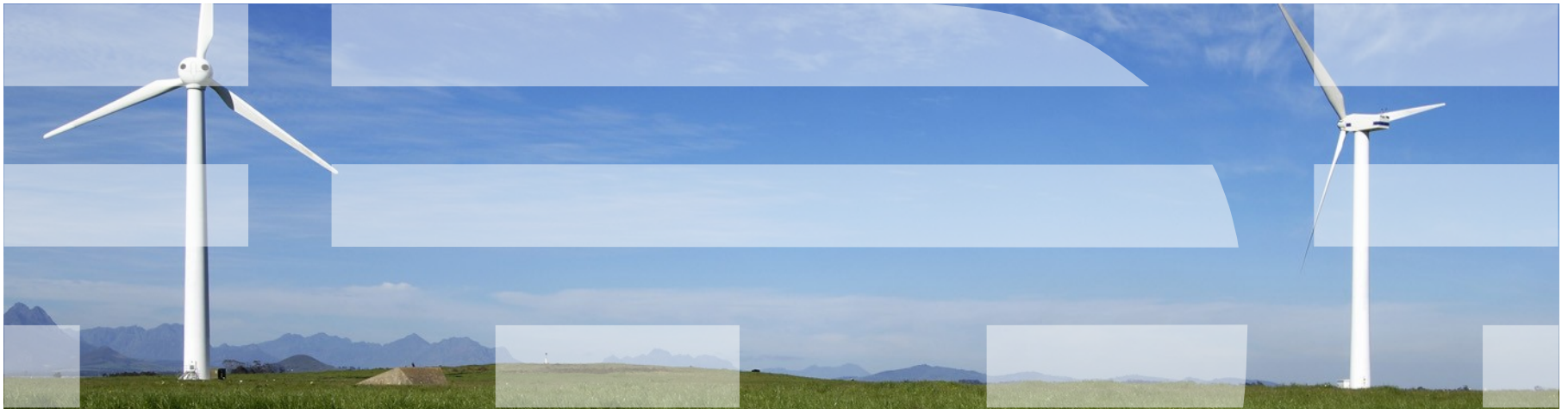
- どちらの設計がいいのでしょうか？
- これはかなり難しい問題で絶対的な正解はありません。
- 最初良いと判断した設計が後で間違いだったことに気付くことも多いのです。  
次のような観点で考えます。
  - 現在の仕様はどちらが実装し易いか。
  - どちらがテストし易いか。
  - どちらが高速か。
  - どちらがメモリ消費量が少ないか。
  - 知らない人が見た時にどちらが理解し易いか。
- 将来の変更についても予想します。
  - 現在の4つの素子以外の素子が追加される可能性は？  
もしも素子が追加されると設計2では多くのクラスにメソッドを追加しなければなりません。
  - 0と1以外の状態が追加される可能性は？  
もしも状態が追加されると設計1では多くのクラスの処理に影響が及びます。

今回は設計1を進めます。

## まとめ

- オブジェクト指向設計には絶対的な解は存在しない。
- 最初に良いと判断した設計が、後で良く無かったと判明することがある。  
自分自身で「9割がた開発が終わった」と思っても最後にどんでん返しがあるかもしれない!
- オブジェクト指向設計の良し悪しを判定するには、次の観点を考慮する。
  - 現在の仕様はどちらが実装し易いか。
  - どちらがテストし易いか。
  - どちらが高速か。
  - どちらがメモリ消費量が少ないか。
  - 知らない人が見た時に理解し易いか。
  - 将来の変更としてどんなものが想定されるか、それらの変更による設計変更が最小で済むか。

# カプセル化



## カプセル化

- 前章の課題の1つは、入力チェックが重複していることでした。オブジェクト指向では重複している部分をオブジェクトに抜き出して解決します。入力チェックは値に対して行われるので、今回は値を表すValueというクラスを作成します。
  - Valueは値を持つ。
  - Valueの値は、0か1のみである。
- 単に値を保持できれば良いのであれば、以下のような実装が考えられます。

```
public class Value {  
    public int value;  
}
```

- しかし、これは一般には良いとは言えません。
  - valueにどんな値でも設定できてしまうので「Valueの値は、0か1のみである」が実現できていない。
  - これまでのように、単にintで保持するのと比べて何も価値を生み出していない。
- オブジェクト内部の状態は外から直接は見えないようにして、状態へのアクセスをアクセッサを通してのみ行うようにするのがカプセル化です。

## カプセル化

- 以下は良く見かけるクラスです。
- valueはprivateになっていて外から見えませんし、アクセスには、setかgetといったアクセッサを使用しています。しかしあまり良い設計とは言えないValueクラスです。

```
public class Value {  
    private int value;  
  
    public void set(int value) {  
        this.value = value;  
    }  
  
    public int get() {  
        return value;  
    }  
}
```

このコードの問題は？

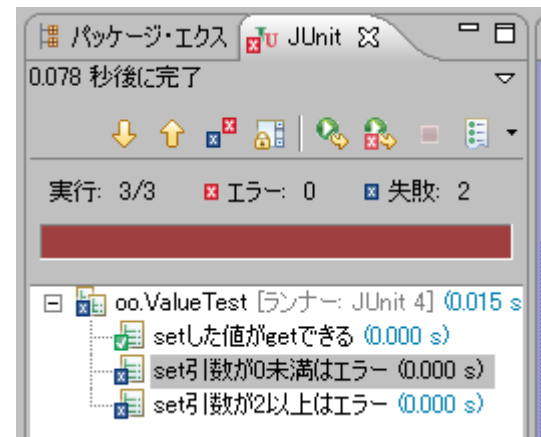
## 不変式

- カプセル化されているようで、カプセル化されていないクラス。
  - 前ページのクラスは、getメソッドとsetメソッドが用意されており、オブジェクト内部のprivateフィールドには、これらのメソッドを経由しなければアクセスできません。
  - しかし、これらのメソッドが「素通し」なので実質的にはカプセル化されているとは言えません。
  - このため「Valueの値は、0か1のみである」が満たされていないValueオブジェクトを生成できてしまいます。
- オブジェクトの状態に特定の条件が付与されている場合、それを不変式(invariant)と呼びます。今回の例であれば「Valueの値は、0か1のみである」が不変式です。
- 異常な状態になったオブジェクトは、やっかいなバグの原因になります。なぜなら通常、異常な状態になったオブジェクトを使用する場面で異常動作が起きますが、オブジェクトがそのような異常な状態になるのは、その前に起きているため犯人の特定が困難だからです。
- オブジェクトに対して、どのような操作を行っても不変式が破られないように設計することが、真のカプセル化です。
- 不変式を破られないようにするには、オブジェクトの状態を変更するメソッド(これをミューテータと呼びます)で、不変式を破るような変更をエラーではじくようにします。



## 演習: 不変式

- Valueクラスの不変式が破られないように変更してください。
- プロジェクト名は、value01です。
- 既に単体テストが作成済みで、そのままではエラーになります。
- Valueクラス内のコメントを頼りにテストが成功するように修正してください。



## 演習: 防御的コピー

- Valueを使って、Not回路を作成してみましょう。
- Notの入力と出力にintの代わりにValueを用いるようにします。
- もちろんまずテストから作成します。

```
public class NotTest {  
    @Test public void 入力0で出力1を返す() {  
        // 入力0を作成  
        Value value = new Value();  
        value.set(0);  
  
        // Not回路を作成して、入力0に値を設定  
        Not not = new Not();  
        not.setInput(0, value);  
  
        assertThat(not.getOutput(0).get(), is(1));  
    }  
}
```

## 演習: 防御的コピー

- プロジェクトは、defensiveCopy01です。
- 既にテストが1つ作成してあり、失敗するようになっています。
- Notクラス内のコメントを参考に修正してください。
- 2つ目のテストも完成させて、テストが全て成功することを確認してください。

## 防御的コピー

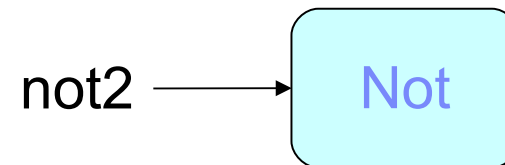
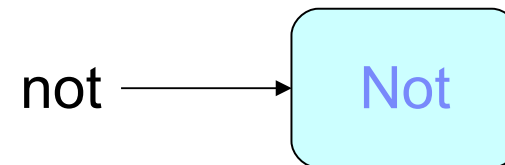
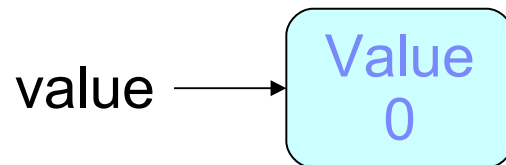
- 以下は2つのNot回路を作って、値を設定して結果を確認しています。
- しかし、このテストは失敗します。
- 何が起きているのでしょうか？

```
@Test public void not回路2つでテスト() {  
    Value value = new Value();  
    Not not = new Not();  
    Not not2 = new Not();  
  
    value.set(1);  
    not.setInput(0, value);  
  
    value.set(0);  
    not2.setInput(0, value);  
  
    assertThat(not.getOutput(0).get(), is(0));  
    assertThat(not2.getOutput(0).get(), is(1));  
}
```

## 防御的コピー

```
Value value = new Value();  
Not not = new Not();  
Not not2 = new Not();
```

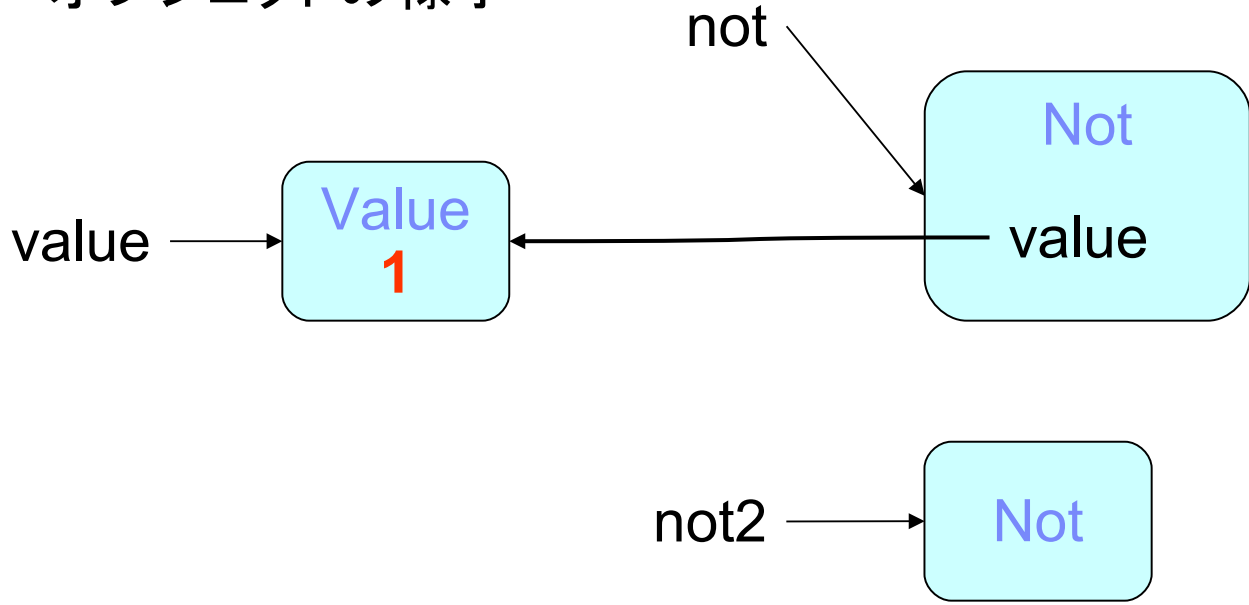
### オブジェクトの様子



防御的コピー

```
value.set(1);  
not.setInput(0, value);
```

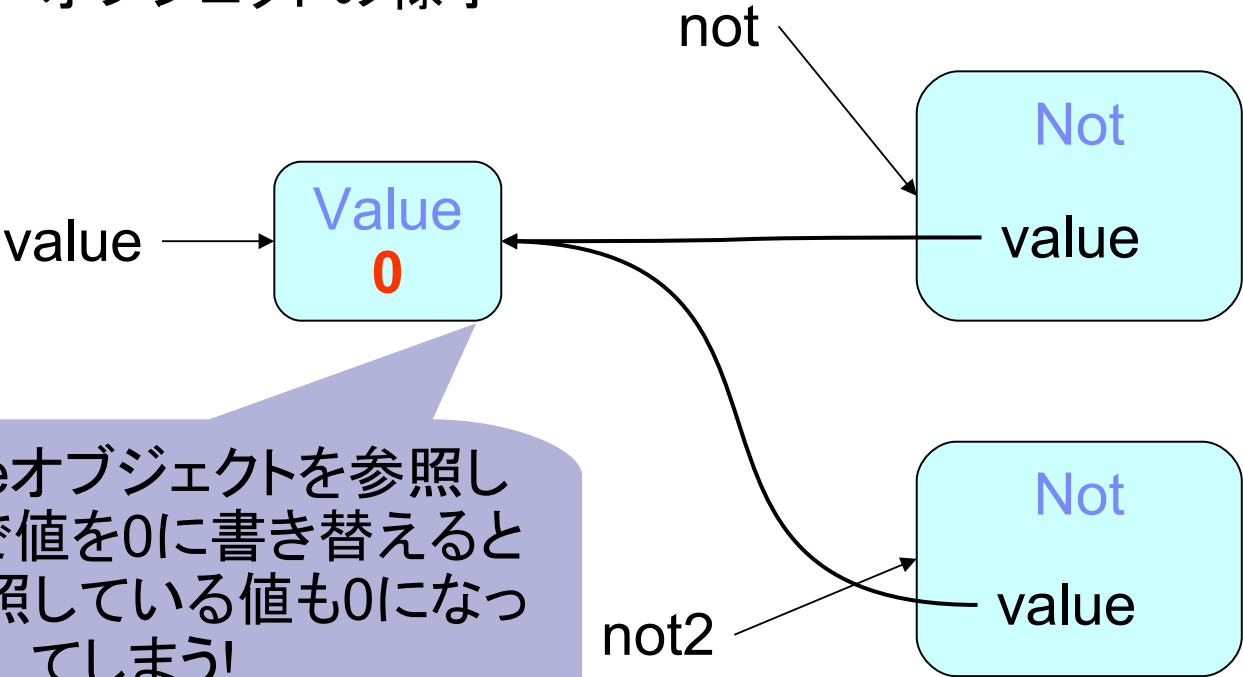
オブジェクトの様子



# 防御的コピー

```
value.set(0);  
not2.setInput(0, value);
```

オブジェクトの様子



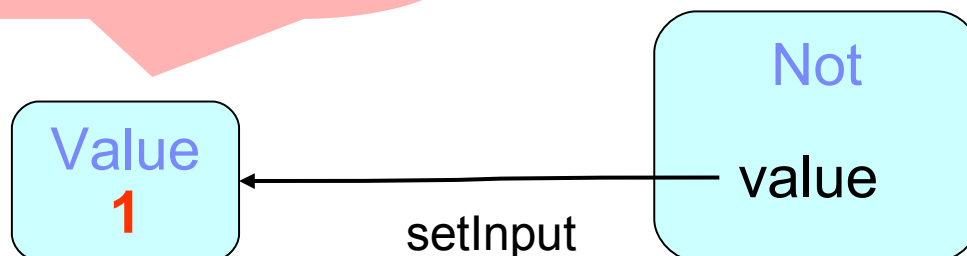
同じValueオブジェクトを参照しているなので値を0に書き替えると、notが参照している値も0になってしまう!

## 防御的コピー

- Notクラスは、カプセル化がきちんとなされていません。
  - setInput()メソッドは受け取ったValueオブジェクトを素通ししているので、Notオブジェクトの中身が外に漏れ出してしまっています。

外から受け取ったオブジェクトが、状態変更可能なものならば、そのままオブジェクト内部に格納してはならない。  
格納すると、オブジェクトの内部が外に漏れ出してしまい、オブジェクトの知らないところで、オブジェクトの状態が変更されてしまう。

valueをそのまま格納すると、Notオブジェクトを通さずに、valueが変更できてしまう。



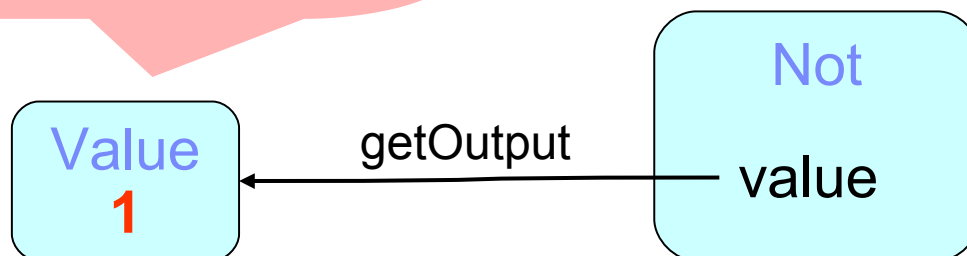


## 防御的コピー

- この注意はセッターだけでなく、ゲッターでも必要です。
  - 今回は、getOutput()を呼ぶごとにValueオブジェクトを生成していますが、もしもフィールドのvalueをそのまま返すようなゲッターがあると、戻ってきたvalueの中身を変更することでオブジェクトの中身が変化してしまいます。

オブジェクトが保持するフィールドが状態変更可能なオブジェクトを参照しているならば、それをそのまま返してはならない。返すとオブジェクトの内部が外に漏れ出してしまう、オブジェクトの知らないところで、オブジェクトの状態が変更されてしまう。

valueをそのまま返すと、Notオブジェクトを通さずに、valueが変更できてしまう。



## 演習: 防御的コピー

- プロジェクトは、defensiveCopy02です。
- テストが失敗するようになっていますので、成功するようにNotクラスを修正してください。

## まとめ

- カプセル化とはオブジェクトの内部の状態を隠蔽して、アクセッサを通してのみ状態を変更できるようにする手法を指します。
- オブジェクトには、オブジェクトが正しく機能するための前提となる条件があり、これを不変式と呼ぶ。
- 真のカプセル化のためにはオブジェクトの不変式が常に破られないようにする必要がある。
- セッターで状態変更が可能なオブジェクトを、フィールドにそのまま格納するとオブジェクトの内部が外部に筒抜けになってカプセル化が破壊される。
- 状態変更が可能なオブジェクトを、フィールドに保持している場合、それをそのままゲッターで返すと、オブジェクトの内部が外部に筒抜けになってカプセル化が破壊される。
- 状態変更が可能なオブジェクトを外部とやりとりする場合、防御的コピーが必要となる。

# 値オブジェクトと不変オブジェクト



## 値オブジェクトと不変オブジェクト

- Valueのように、もっぱら値を保持することが主目的で他にふるまいをほとんど持たないオブジェクトを値オブジェクトと呼びます。
- 値オブジェクトは、intなどのようなプリミティブ型と同じように使用できるべきです。
- ところが前章で見た通り、実際にはintと比べると防御的コピーというわずらわしいプログラミングが必要になってしまいます。
- しかし、これを解決できる方法があります。それはオブジェクトを不変にするという設計方法です。
- 不変なオブジェクトのことをイミュータブル(Immutable)なオブジェクトとも呼びます。逆に状態を変更できるオブジェクトのことをミュータブル(Mutable)なオブジェクトと呼びます。
- 値オブジェクトは不変オブジェクトにすべきです。これにより防御的コピーで見たような、やっかいな問題を避けることができます。
- 不変オブジェクトは値が変更できないので、最初に不変式を守りさえすればオブジェクトの不変式が途中で破られる心配はいりません。

# 値オブジェクトと不変オブジェクト

- 標準で用意されている値オブジェクトの例(全て不変オブジェクト)

クラス	役割
String	文字列を表現する。
Integer	intのラッパークラス
Long	longのラッパークラス

- 標準で用意されている値オブジェクトの例(ミュータブル)

クラス	役割
Point	座標を表す。
Date	日付を表す。

ミュータブルな値オブジェクトは設計ミスと考えられています。Dateクラスについては、Java8でイミュータブル版が追加される予定です。  
自分で設計する時は不変オブジェクトとしてください。

## 不変オブジェクト

- 不変オブジェクトを作成するには次のようにします。
  - オブジェクトを変更するメソッドを用意してはいけません。
  - フィールドはfinal宣言して、コンストラクタで値を設定します。
  - 必要に応じてビルダークラスを用意します(この内容は高度になるので、応用編で解説します)。
- Valueクラスを、イミュータブルにしましょう。

## 演習: 値オブジェクト

- プロジェクトは、immutable01です。
- テストが既に用意されています。
- テストが成功するように、Valueクラスを修正してください。
- Valueクラスには、equals()、hashCode()というメソッドが用意されていますが、これはValueを比較するためのメソッドです。内容が高度になるので入門編では解説しません。これ以外にも値オブジェクト、不変オブジェクトには様々なトピックがあります。興味がある方は、応用編を受講ください。



## 値オブジェクト

- Notクラスは、なぜValueオブジェクトのget()を呼んでいるのでしょうか？

```
public Value getOutput(int index) {  
    if (value.get() == 0) return new Value(1);  
    else return new Value(0);  
}
```

- Valueオブジェクト自体が値を表現しているのですから、Valueオブジェクト自体を比較するべきでしょう。つまり以下のように記述した方が直接的ですね。

```
public Value getOutput(int index) {  
    if (value.equals(new Value(0))) return new Value(1);  
    else return new Value(0);  
}
```

- ここで、Valueオブジェクトはイミュータブルです。つまり状態を変更できません。ということはValue(0)と、Value(1)の2つのオブジェクトを用意して必要な場面で同じオブジェクトを共有しても安全です。これをフライウエイパターンと呼びます。
- また上記のようにValueオブジェクト自体が比較できるのであれば、もうintの値には意味がありません。Valueクラスを変更しましょう。

## 値オブジェクト

- 以下は改良したValueオブジェクトです。  
これまでとは大きく変化していることが分かります。

```
public class Value {  
    public static final Value ZERO = new Value();  
    public static final Value ONE = new Value();  
  
    Value() {}  
}
```

不変なので、ZEROとONE  
を使い回せば良い。

Value自体を比較すれば良  
いので、int値は削除。

ZERO, ONE以外のインスタンスが勝手に作られないよ  
うにコンストラクタをパッケージプライベートに変更。

- Java5からは、このような値オブジェクトを簡単に作成するための機構が用意されています。  
タイプセーフenumです。これを使うと以下のように簡単に書くことができます。

```
public enum Value {  
    ZERO, ONE;  
}
```

## nullチェック

- ValueのインスタンスはZEROとONEの2つのみになりました。もう入力チェックは不要でしょうか？
- 残念ながら1つのみチェックが必要です。それはnullチェックです。

```
public class Not {  
    private Value value;  
  
    public void setInput(int index, Value value) {  
        if (value == null) throw new NullPointerException();  
        this.value = value;  
    }  
  
    public Value getOutput(int index) {  
        if (value.equals(Value.ZERO)) return Value.ONE;  
        else return Value.ZERO;  
    }  
}
```

ここでvalueがnullにならないように不正な値をはじきます。

valueがnullだと、ここでNullPointerExceptionが発生します。つまりvalue != nullがNotクラスの不変式です。

残念ながらJavaではnullを許容しない型を作成できません。  
不変式を検討する際には、nullに十分注意すること。

## 演習：タイプセーフenum

- Notクラスをタイプセーフenumに合わせて修正してください。
- プロジェクト名は、typeSafeEnum01です。
- コンパイルエラーがあるので修正してください。
- Notクラスのテストが一部、未完成です。コメントに従って完成させてください。

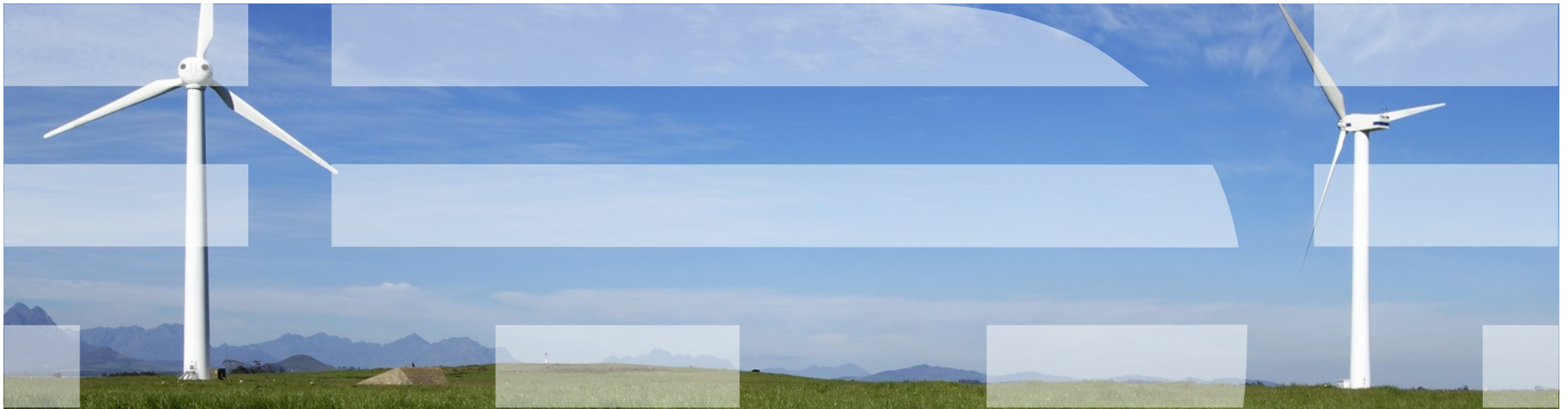
## 何が改善されたのか？

- それでは手続き型の際に問題となった項目が、ここまででどのくらい改善されているのか、振り返ってみましょう。
- 手続き型の場合、入力チェックに伴うコード重複が問題となっていました。今回はどうでしょうか？ 入力チェックは、nullチェックだけになりました。これに伴ってコードの量がテストも含めて劇的に減っていることに注意してください。
- 今回は1か0を表現するためにintが「流用」されていた点に着目して、オブジェクトを導入しました。このように「流用」に着目してより適切な抽象化を導入するのがオブジェクト導入の1つの方法です。特に文字列型は良く流用されるので安易に使用しないように注意が必要です。
- 残りの問題は、この後の章で解決していきます。

## まとめ

- もっぱら値を保持することが主目的で他にふるまいをほとんど持たないオブジェクトを値オブジェクトと呼ぶ。
- 値オブジェクトがミュータブルだと、オブジェクト間でのやりとりに防御的コピーが必要となってしまう、プリミティブ型のように手軽に扱うことができない。
- 値オブジェクトをイミュータブルにすることで、プリミティブ型のようにオブジェクト間でそのまま受け渡しすることが可能となる。
- 原則、値オブジェクトはイミュータブルとすべきである。
- プリミティブ型や文字列型が「流用」されている部分に着目してオブジェクトが導入できないか検討する。
- もしも限られた個数の値オブジェクトのインスタンスさえあれば良いのであれば、タイプセーフenumにすることを検討する。
- 不変式を検討する際には、nullに十分注意し、必要に応じてnullチェックを実施すること。

# Nullオブジェクト



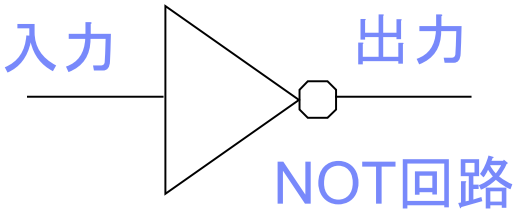
## Nullオブジェクト

- 現在のNotクラスは、あまり良い設計とは言えません。  
Notクラスのインスタンスを生成した後、setInput()せずにいきなりgetOutput()するとNullPointerExceptionが発生します。これはgetOutputでnullチェックをしていないことが直接の原因です。  
しかし根本的な原因は、別にあります。前章の最後で「value != nullがNotクラスの不変式です。」と述べましたが、**setInput()する前はこの不変式が満たされていません**。
- もちろんgetOutputの中にnullチェックを入れて、入力がnullだったら出力もnullにするという設計も可能です。
- しかしnullという値は、扱いにくい存在です。
  - 注意していないとNullPointerExceptionのスローを招きます。このためnullチェックのコード重複を招きます。
  - 非対称です。null.equals(new Value(0))はエラーですが、new Value(0).equals(null)は可能です(falseが返る)。
  - オブジェクトには自分でふるまいを定義できますが、nullにはふるまいを定義できません。ValueクラスのnullとNotクラスのnullの間に区別はありません。
- このため一般にはなるべくnullを扱わないで済むように、nullのかわりに特殊な値、ふるまいを持ったオブジェクトを用意します。これをNullオブジェクトパターンと呼びます。

Valueに「不定」という値を追加しましょう。



# Nullオブジェクトパターン



真理値表

入力	出力
1	0
0	1
不定	不定

- Not回路の場合、入力が不定ならば(1か0か分からない)ならば、出力も不定です。
- Notクラスにおける、valueの初期値をこの「不定」にしましょう。
- これでNotクラスの不変式をあらゆる場面で満たすことができます。

## 演習: nullオブジェクトパターン

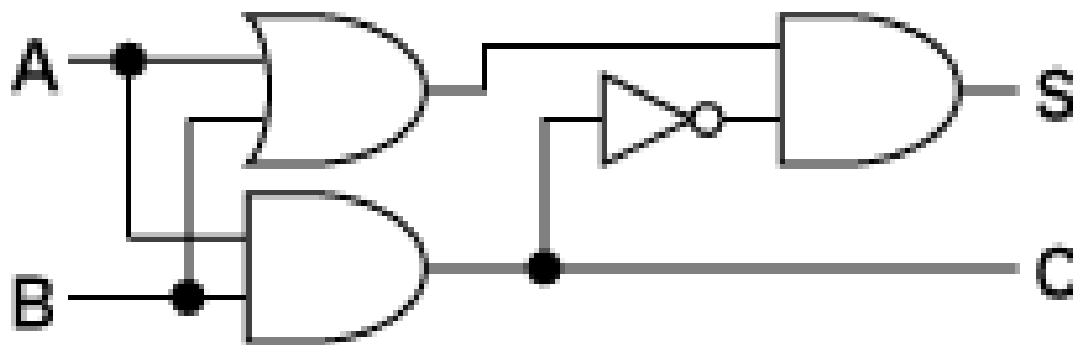
- プロジェクト名は、nullObject01です。
- テストが失敗するので、それをもとに修正してください。

# オブザーバパターン



## 結線を、もっと直感的に表現するには？

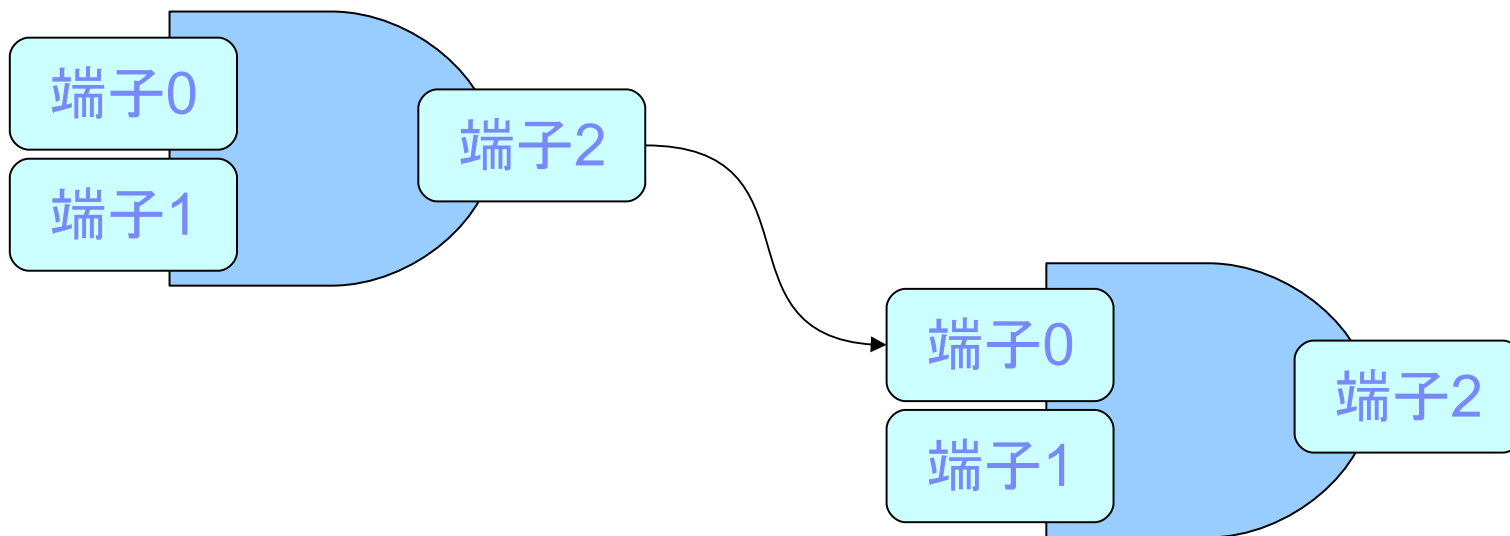
- 手続き型の最後の問題にとりかかりましょう。  
半加算器の実装では、半加算器の結線と半加算器の実装との間に乖離があることが問題でした。
- このように実装とドメインとの間に乖離があるケースも、オブジェクト導入の候補となります。



- 上の回路で、Aに入力された状態は、ORとANDの入力へと伝えられます。つまり結線とは1つの出力を複数の入力へと伝える仕組みです(\*1)。
- あるオブジェクトの状態を監視する場合に適したデザインパターンがあります。オブザーバパターンです。

(\*1) 実際の回路では、複数の出力が1つに結線されることもありますが無視します。

## オブザーバパターン



- 上の図は、2つのAND回路を結線した状態を表しています。
- AND回路には3つの端子があり、それぞれに端子オブジェクトを用意しています。
- 左の端子オブジェクトを、右の端子オブジェクトに「接続」することで、端子2の変化が端子0に伝わるようにします。
- それでは端子オブジェクトを作成していきましょう。

## オブザーバパターン

### ■ 端子オブジェクトのユースケース

- 論理回路は、1つ以上(通常は2つ以上)の端子オブジェクトを持つ。
- ある端子オブジェクトは別の端子オブジェクトを結線することで監視できる。監視対象オブジェクトの状態が変化すると、監視しているオブジェクトに伝えられる。
- 端子オブジェクトに対し現在の状態を問い合わせることができる。

それではテストを作成していきましょう。

observer00を開いて一緒に作業してみてください

Notは、getInput()、getOutput()で、Terminalオブジェクトを返します。

```
@Test public void 入力が0だと出力は1() {  
    Not not = new Not();  
    Terminal inputTerminal = not.getInput(0);  
    inputTerminal.setValue(Value.ZERO);  
  
    Terminal outputTerminal = not.getOutput(0);  
    assertThat(outputTerminal.getValue(), is(Value.ONE));  
}
```

TerminalのgetValue()状態を得ることができます。

## オブザーバパターン

- テストを作成するとコンパイルエラーになるので、これまでのようにRADを使ってコンパイルエラーを修正していきます。

```
-----  
Terminal inputTerminal = not.getInput(0);  
inputTerminal.setValue(Value.ZERO);
```

Terminalクラスが無いというエラーなので、電球をクリックして作成。

```
Terminal inputTerminal = not.getInput(0);  
inputTerminal.setValue(Value.ZERO);
```

setValue()が無いというエラーなので、電球をクリックして作成。

```
Terminal outputTerminal = not.getOutput(0);  
assertThat(outputTerminal.getValue(), is(Value.ONE));
```

getValue()が無いというエラーなので、電球をクリックして作成。

# オブザーバパターン

- テストを作成するとコンパイルエラーになるので、これまでのようにRADを使ってコンパイルエラーを修正していきます。

```
Terminal inputTerminal = not.getInput(0);
inputTerminal.setValue(Value.0)
```

getInput()が無いというエラーなので、電球をクリックして作成。

```
Terminal outputTerminal = not.getOutput(0);
assertThat(outputTerminal, equalTo(Value.ONE));
```

```
public Terminal getOutput(int index) {
    if (value.equals(Value.ZERO)) return Value.ONE;
    ...
}
```

- 'outputTerminal' の型を 'Value' に変更します。
- getOutput() の戻りの型を 'Terminal' に変更
- ファイル内の名前変更 (Ctrl+2, R)
- ワークスペース内の名前変更 (Alt+Shift+R)

型が違うというエラーなので、Not.getOutput()の戻り型をTerminalに変更。

Notを実装しましょう。



## オブザーバパターン

```
public Terminal getOutput(int index) {  
    if (value.equals(Value.ZERO)) return Value.ONE;  
    else return Value.ZERO;  
}
```

これまでValueを返していましたが、今回からTerminalを返すようになったので、書き直しが必要。

## オブザーバパターン

### 変更後のNotクラス

```
public class Not {  
    final Terminal inputTerminal = new Terminal();  
    final Terminal outputTerminal = new Terminal();  
  
    public Terminal getOutput(int index) {  
        return outputTerminal;  
    }  
  
    public Terminal getInput(int index) {  
        return inputTerminal;  
    }  
}
```

ユースケースの残り、監視機能を追加します。  
これまで通りテストを追加します。

## オブザーバパターン

```
@Test public void connectされた端子に情報が伝わる() {  
    Terminal from = new Terminal();  
    Terminal to = new Terminal();  
    from.addListener(to);  
  
    assertThat(to.getValue(), is(Value.UNKNOWN));  
  
    from.setValue(Value.ZERO);  
    assertThat(to.getValue(), is(Value.ZERO));  
  
    from.setValue(Value.ONE);  
    assertThat(to.getValue(), is(Value.ONE));  
}
```

fromに変化があったら、toに伝える。

これまで通り、RADの機能で、addListener()メソッドを作成します。  
TerminalTestを実行して失敗することを確認しましょう。

## 演習:オブザーバパターン

- プロジェクト名は、observer01です。
- テストの失敗を手掛りに、コードを修正してください。
- NotTestは失敗したままで構いません。TerminalTestが通るように変更してください。

## オブザーバパターン

- Terminalクラスはできましたが、Notクラスが未完成です。

```
public class Not {  
    final Terminal inputTerminal = new Terminal();  
    final Terminal outputTerminal = new Terminal();  
}
```

inputTerminalに変化があったら、Notの結果を、outputTerminalに設定したい。

しかし現在のTerminalは、単純に同じ値を伝えることしかできず柔軟性が無い。Notでは反転した値を伝えたい。

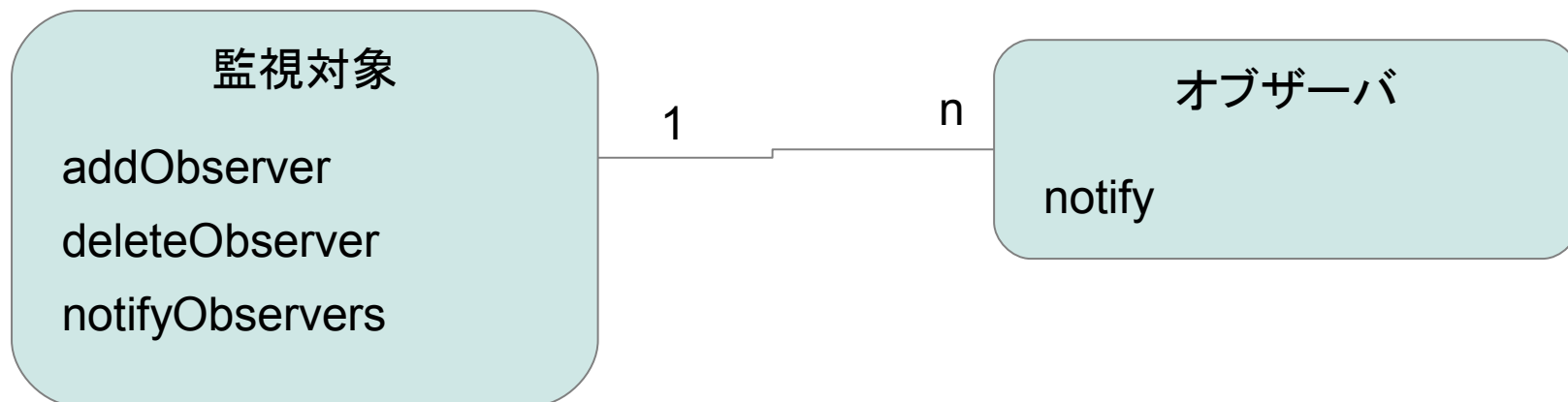
```
final List<Terminal> listeners = new ArrayList<Terminal>();  
  
public void setValue(Value v) {  
    if (v == null) throw new NullPointerException();  
    value = v;  
    for (Terminal t: listeners) {  
        t.setValue(v);  
    }  
}
```

そこでTerminal自体をaddするのではなく、処理をaddできるようにする。

## オブザーバパターン

### ■ オブザーバパターン

- 監視対象にaddObserverでオブザーバを登録しておく。
- 監視対象側は、自分の状態が変化したら登録されているオブザーバ全てに対して、notifyを呼んで状態が変化したことを知らせる。



### ■ Javaの場合、慣例的に次のような命名が用いられることが多い。

- オブザーバ => XXXListener
- addObserver => addXXXListener
- notifyObservers => fireXXX
- notify => onXXX

## 演習: オブザーバパターン

- テストを作成します。

```
@Test public void 端子を監視する() {  
    final Value[] latestValue = {Value.UNKNOWN};
```

オブザーバ

```
    Terminal terminal = new Terminal();  
    terminal.addListener(new TerminalListener() {  
        public void onChange(Value newValue) {  
            latestValue[0] = newValue;  
        }  
    });
```

端子の状態が変化すると、ここが呼び出される。

```
    assertThat(latestValue[0], is(Value.UNKNOWN));
```

```
    terminal.setValue(Value.ZERO);  
    assertThat(latestValue[0], is(Value.ZERO));
```

```
    terminal.setValue(Value.ONE);  
    assertThat(latestValue[0], is(Value.ONE));
```

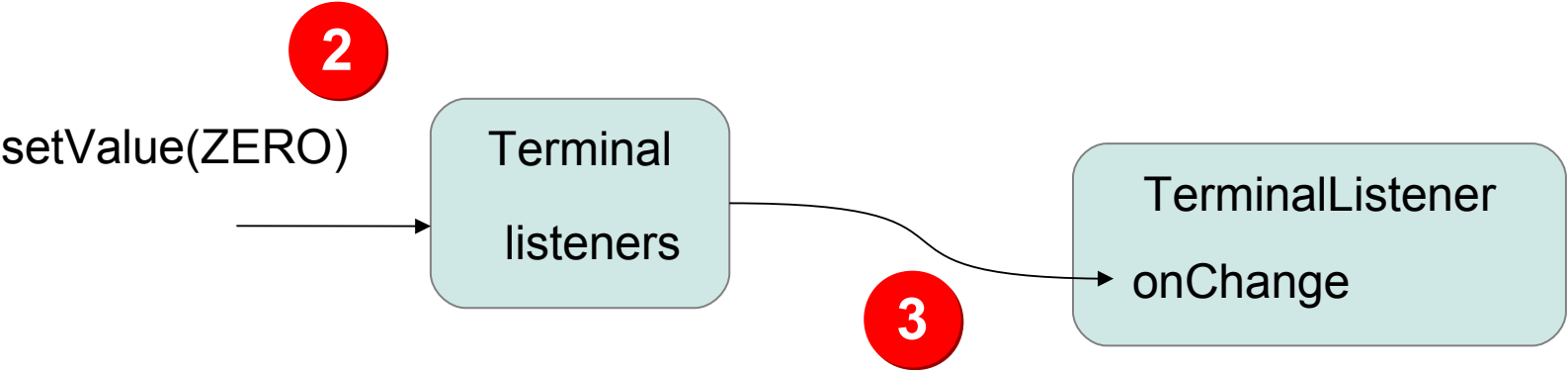
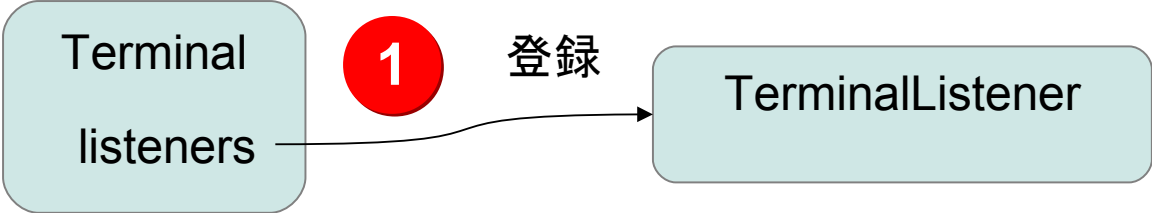
```
}
```

## 演習: オブザーバパターン

- プロジェクト名は、observer02です。
- コンパイルエラー、テスト失敗を手掛りに完成させてください。
- TerminalListenerはインターフェースとして作成してください。
- addListener(Terminal)は、addListener(TerminalListener)に変更されています。
- これまでの、addListener(Terminal)は、connectTo(Terminal)という名前に変更されています。ここが虫食いになっているのでテストを手掛りに完成させてください。
- 動きが良く分からなかったら、次ページの図も参考にしてください。
- NotTestは失敗したままで構いません。



# オブザーバパターン



listenersに保持された全てのリスナに対し、onChangeメソッドで新しい値が通知される。

## オブザーバパターン

- 最後の演習はちょっと難しかったかもしれませんが、これでようやくお膳立てが揃いました。
- Notクラスのテストが失敗したままになっています。これを修正しましょう。

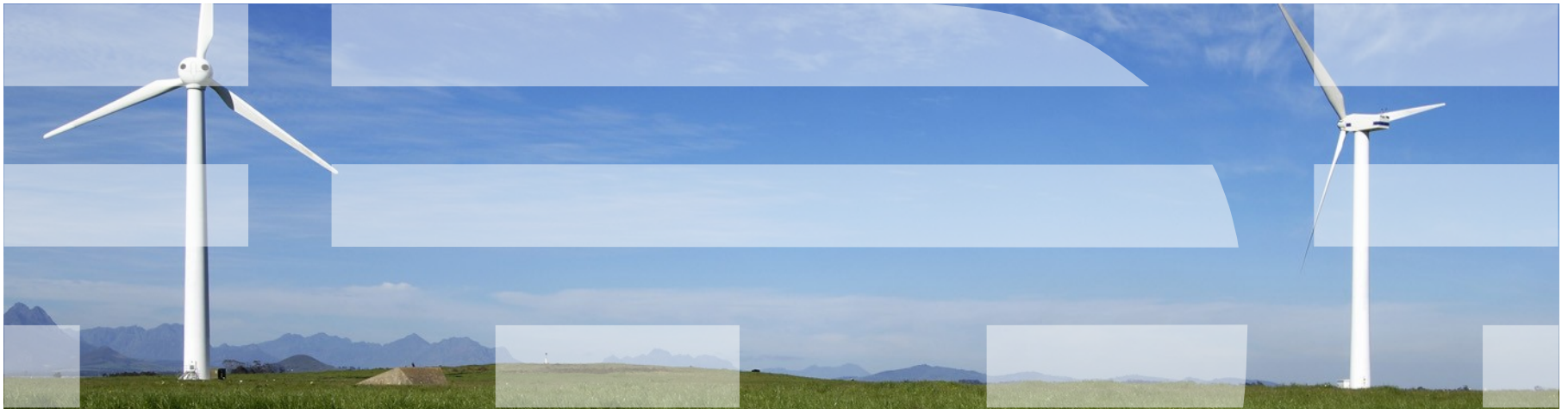
## 演習:オブザーバパターン

- プロジェクト名は、observer03です。
- コンパイルエラー、テスト失敗を手掛りに完成させてください。
- Notクラスが虫食いになっているので修正してください。

## まとめ

- オブザーバパターンは、オブジェクトの状態を監視するためのデザインパターンです。
- オブザーバパターンを用いると、監視する側と監視される側との間の依存を最小限にすることができます。

# 多態



## 多態

- オブジェクト指向設計では、外部インターフェースが同一で、ふるまいのみが異なるオブジェクトを作成することができます。これによって、呼び出し側で処理内容を決めるのではなく、呼び出された側で処理内容を決定することができます。つまりオブジェクトに対して全く同一のメソッドを同一の引数で呼び出しても、オブジェクトによって、その処理内容は変化します。
- というのが一般的な説明ですが...

乱暴に言えば、多態はプログラムから条件分岐を無くすための仕組みです。

もしも自分のプログラム内に、ifやswitchが多いなら、きっとそれはオブジェクト指向的というより手続的です。

## 多態

- 現在のプログラムを見てみましょう。

```
public Not() {  
    inputTerminal.addListener(new TerminalListener() {  
        @Override  
        public void onChange(Value newValue) {  
            switch (newValue) {  
                case UNKNOWN:  
                    outputTerminal.setValue(Value.UNKNOWN);  
                    break;  
  
                case ZERO:  
                    outputTerminal.setValue(Value.ONE);  
                    break;  
  
                case ONE:  
                    outputTerminal.setValue(Value.ZERO);  
                    break;  
            }  
        }  
    });  
}
```

この部分は、オブジェクトの値を調べて、switchで分岐しておりオブジェクト指向的とは言えません。多態を用いて解決しましょう。

## 多態

- 現在は、Valueの値を調べて分岐しています。
- 逆に言えば、ValueにはUNKNOWN、ZERO、ONEという値があり、この値によって、Notの結果が決まっています。
- それなら、Value側にnot結果を取得するメソッドnot()を用意して、Notクラスはnot()の結果をそのまま返してしまえば分岐が不要になります。
- テストを書いてみましょう。

```
assertThat(Value.UNKNOWN.not(), is(Value.UNKNOWN));  
assertThat(Value.ZERO.not(), is(Value.ONE));  
assertThat(Value.ONE.not(), is(Value.ZERO));
```



## 演習: 多態

- Valueクラスをコンパイルエラーとテストを手掛りに完成させてください。
- Notクラスを、コメントに従い、Value.not()を使用するように変更してください。
- プロジェクトはpolymorphism01です。

## タイプセーフenumとメソッド

- タイプセーフenumで定義される定数は、オブジェクトです。
- このためメソッドを宣言することができます。
- 今回は、これを利用してnot()メソッドを用意しています。

```
public enum Value {  
    UNKNOWN {  
        @Override public Value not() {  
            return Value.UNKNOWN;  
        }  
    },  
    ZERO {  
        @Override public Value not() {  
            return Value.ONE;  
        }  
    },  
    ONE {  
        @Override public Value not() {  
            // TODO: 実装してください。  
            return Value.???;  
        }  
    }  
};  
  
abstract public Value not();  
}
```

## 多態

- 何が改善されたのでしょうか？
- これまでのように値に応じてswitchによる分岐処理を書いていると、もしも状態がUNKNOWN、ZERO、ONE以外に追加されたら、全てのswitch文を修正しなければなりません。
- 一方、Valueにnot()を用意しているなら、Valueを修正するだけで済みます。
- 逆の視点で見てみましょう。NotクラスとValue.not()が併存することで、notに関連するロジックが複数に存在することになり、分かりにくくなっているとも言えます。
- Notクラスは単に端子を持っているだけで一見無駄に見えます。コンストラクタで入力端子と出力端子との間に接続を形成しているのみなので、これはクラスでなくてもメソッドでも構いません。
- しかし、Not回路という視点とNot演算という視点では、やはり別概念と考えられます。例えば回路中にNot回路が2つ以上あり、それらを区別したい(例えば名前を付けるとか)場合、やはりNotクラスというグルーピングする概念のものが良かった方が良いでしょう。
- そろそろ設計が固まってきました。NotだけでなくAndとOrも作成しましょう。

# 繼承



## 継承

- 前章でタイプセーフenumであるValueにnot()を定義しました。
- 同じようにして、and()とor()を用意しましょう。andとorは2つの入力がありますから、Valueの場合1引数のメソッドとして用意します。

```
@Test public void and() {  
    assertThat(Value.UNKNOWN.and(Value.UNKNOWN), is(Value.UNKNOWN));  
    assertThat(Value.UNKNOWN.and(Value.ZERO), is(Value.ZERO));  
    assertThat(Value.UNKNOWN.and(Value.ONE), is(Value.UNKNOWN));  
  
    assertThat(Value.ZERO.and(Value.UNKNOWN), is(Value.ZERO));  
    assertThat(Value.ZERO.and(Value.ZERO), is(Value.ZERO));  
    assertThat(Value.ZERO.and(Value.ONE), is(Value.ZERO));  
  
    assertThat(Value.ONE.and(Value.UNKNOWN), is(Value.UNKNOWN));  
    assertThat(Value.ONE.and(Value.ZERO), is(Value.ZERO));  
    assertThat(Value.ONE.and(Value.ONE), is(Value.ONE));  
}
```

Value.and()は自分自身と引数のValueとのand演算結果を返す。AND演算では片方がZEROなら、もう片方の値によらずZEROになることに注意。

## 演習: 継承

- プロジェクト名は、inheritance01です。
- 既にand()が完成しています。or()のテストと本体を完成させてください。
- OR演算では、片方がONEなら、もう片方の状態によらず結果がONEになることに注意してください。

## 継承

- `and()`と`or()`ができたので、Andクラス、Orクラスを作成しましょう。

## 演習: 継承

- プロジェクト名は、inheritance02です。
- 既にAndのテストが完成しています。まずコンパイルエラーとテストを手掛りに、Andクラスを完成させてください。
- Orクラスの単体テストを作成してください。
- コンパイルエラーとテストを手掛りに、Orクラスを完成させてください。



## 継承

- Not、And、Orクラスには重複があります。
  - getInput()、getOutput()は同じロジックです。
  - input, outputフィールドも共通です。
- もしもクラス間に共通のふるまい、状態がある場合、それらをくり出して共通のクラスを導出することができます。
- 導出されたクラスを親クラス、スーパークラス、基底クラスと呼びます。
- 共通クラスを導出した元のクラス側を、子クラス、サブクラス、継承クラスと呼びます。
- これを「子クラス側が、親クラス側を継承している」と呼びます。
- Not、And、Orクラスの共通部分を括り出したLogicクラスを作成しましょう。

## 継承

- リファクタリングを行ってベースクラスを導出します。
- 手順を示しながら進めていきます。実際に手元でも操作してみてください。
- 開始プロジェクトはinheritance03です。
- 現在、Notクラスは入力が1つなのでinputが配列になっていませんが、AndとOrは入力が2つなので配列になっています。このままだと共通化できないので、Notのinputも配列にしましょう。

```
public class Not {  
    final Terminal[] input = {new Terminal()};  
}
```

まずinputを配列にします。

```
public Not() {  
    input.addListener(new TerminalListener() {  
        @Override  
        public void onChange(Value newValue) {  
            output.setValue(newValue.not());  
        }  
    });  
}
```

すると、inputを使用しているところが全てコンパイルエラーになるので、それをてがかりに修正します。

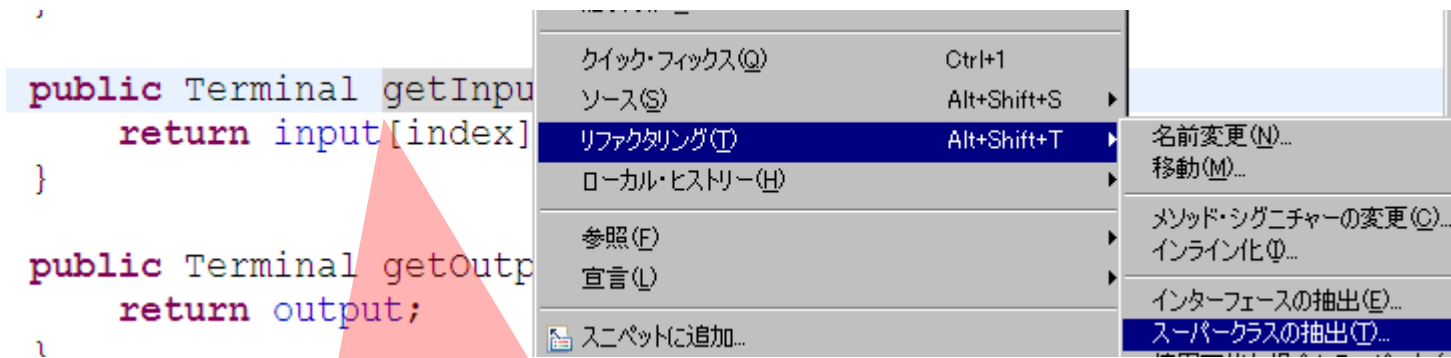
## 継承

```
public Not() {  
    input[0].addListener(new TerminalListener() {  
        @Override  
        public void onChange(Value newValue) {  
            output.setValue(newValue.not());  
        }  
    });  
}  
  
public Terminal getOutput(int index) {  
    return output;  
}  
  
public Terminal getInput(int index) {  
    return input[index];  
}
```

テストを実行してください。コード1文字でも変更したらテストを実行します。もしも失敗するようなら、どこか間違っています。

# 継承

- それではベースクラスを導出します。
- といってもRADで自動的にできますから、身構える必要はありません。



AndクラスのgetInput()メソッドの上にカーソルを置いて、右クリックしリファクタリング => スーパークラスの抽出を選びます。

継承

リファクタリング

スーパークラスの抽出  
新規の型へ抽出するメンバーの選択。

スーパークラス名(S):

☒ 使用可能な場合は、抽出クラスを使用(E)  
☐ 'instanceof' 式で抽出クラスを使用(U)  
☒ 必要なメソッド・スタブを抽出型の非抽象サブタイプ(作成(C))

スーパークラスの抽出元の型(T):  

And - oo

追加(D)...

削除(R)

メンバーのアクションを指定(P):

メンバー	アクション
<input checked="" type="checkbox"/> input	
<input type="checkbox"/> output	
<input type="checkbox"/> getInput(int)	
<input type="checkbox"/> getOutput(int)	

すべて選択(A)

選択をすべて解除(U)

アクションの設定(O)...

必須を追加(R)

0 個のメンバーが選択されました。

?

< 戻る(B)

次へ(N) >

終了(F)

キャンセル

スーパークラス名に、Logicと入力

OrとNotを含めるため、追加ボタンをクリック。

## 継承



OrとNotを含めてOKボタンをクリック  
(Ctrlキーを押しながら選択します)。

# 継承

リファクタリング

スーパークラスの抽出

新規の型へ抽出するメンバーの選択。

スーパークラス名(S): Logic

☒ 使用可能な場合は、抽出クラスを使用(E)

☐ 'instanceof' 式で抽出クラスを使用(U)

☒ 必要なメソッド・スタブを抽出型の非抽象サブタイプに作成(O)

スーパークラスの抽出元の型(T):

And - oo

Not - oo

Or - oo

追加(D)...  
削除(R)

メンバーのアクションを指定(P):

メンバー	アクション
<input checked="" type="checkbox"/> input	抽出
<input checked="" type="checkbox"/> output	
<input checked="" type="checkbox"/> getInput(int)	
<input checked="" type="checkbox"/> getOutput(int)	

すべて選択(S)

4 個のメンバーが選択されました。

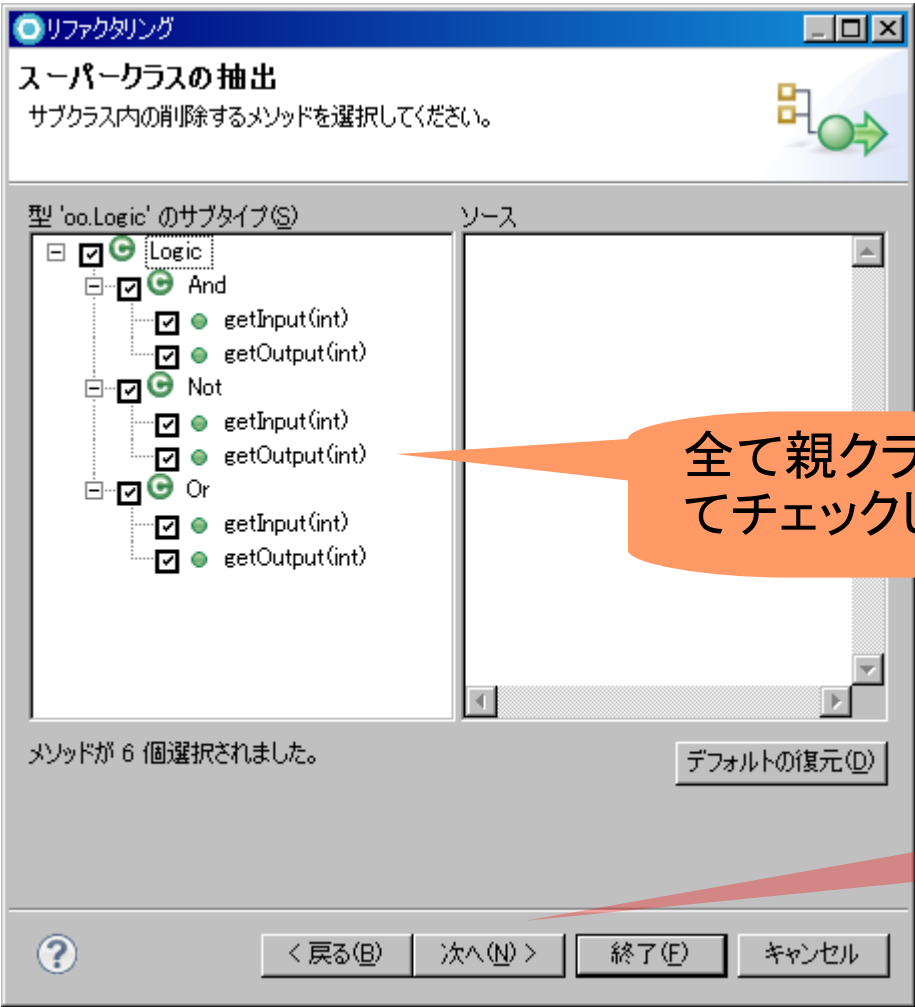
?

< 戻る(B) 次へ(N) > 終了(E) キャンセル

全て親クラスに持って行きたいので全てチェックします。

次へをクリック。

継承





# 継承

リファクタリング

スーパークラスの抽出

下のリストに示されている情報を確認してください。'次へ >' をクリックして次の項目を表示するか、'終了' をクリックします。

検出された問題

作成時またはコンストラクターで初期化されていない場合、final フィールドを移動するとコンパイル・エラーになります

作成時またはコンストラクターで初期化されていない場合、final フィールドを移動するとコンパイル・エラーになります

And.java

```
package 〇〇;

public class And extends Logic {
    public And() {
        for (Terminal in: input) {
            in.addListener(new TerminalListener() {
                @Override
                public void onChange(Value newValue) {
                    output.setValue(input[0].getValue().and(in
                });
            });
        }
    }
}
```

?

< 戻る(B)

次へ(N) >

終了(F)

キャンセル

finalフィールドを移動しようとしているので警告が出ますが、今回は問題ないので終了をクリックします。

## 継承

- これで、Not、And、Orから共通部分が削除され、Logicに移動したはずですが。確認してください。
- テストを流してください。
- テストは成功するかもしれませんが、実は1箇所うまくいっていないところがあります。
- Logicクラスを見ると以下のようになっています。

```
public class Logic {  
  
    protected final Terminal[] input = {new Terminal(), new Terminal()};  
    protected final Terminal output = new Terminal();  
}
```

- Notには入力が1つしかありませんから、inputの配列サイズが2になってしまうのは不適切です。
- 修正しましょう。ただし、いきなり本体クラスを修正しないでください。常にテストが先です。今回の問題が検出できるようにテストを修正しましょう。

もしもバグが見つかったら本体コードを直す前に、そのバグがきちんと捕捉できるようにテストを修正します。

# 継承

- Not回路のテストを追加します。

```
@Test(expected = IndexOutOfBoundsException.class)
public void getInputの添字が1以上はエラー () {
    Not not = new Not();
    not.getInput(1);
}
```

- テストが失敗することを確認します。
- それでは本体コードを修正しましょう。今回はLogicクラスのコンストラクタで入力端子の個数を受け取り、その個数に応じた入力端子を生成するようにします。

```
public class Logic {
    protected final Terminal[] input;
    protected final Terminal output = new Terminal();

    public Logic(int inputTerminalCount) {
        this.input = new Terminal[inputTerminalCount];
        for (int i = 0; i < input.length; ++i)
            input[i] = new Terminal();
    }
}
```

入力端子の個数を受け取る。

指定された個数の入力端子を生成する。

## 継承

- これによってコンパイルエラーが起きるので、それを手掛りにコードを修正します。

```
public class And extends Logic {  
    public And() {  
        super (2);  
        for (Terminal in: input) {
```

AndとOrは、端子数2を渡す。

```
public class Or extends Logic {  
    public Or() {  
        super (2);  
        for (Terminal in: input)
```

Notは、端子数1を渡す。

```
public class Not extends Logic {  
    public Not() {  
        super (1);  
        input[0].addListener(new
```

テストが通ることを確認してください。

## @Beforeアノテーション

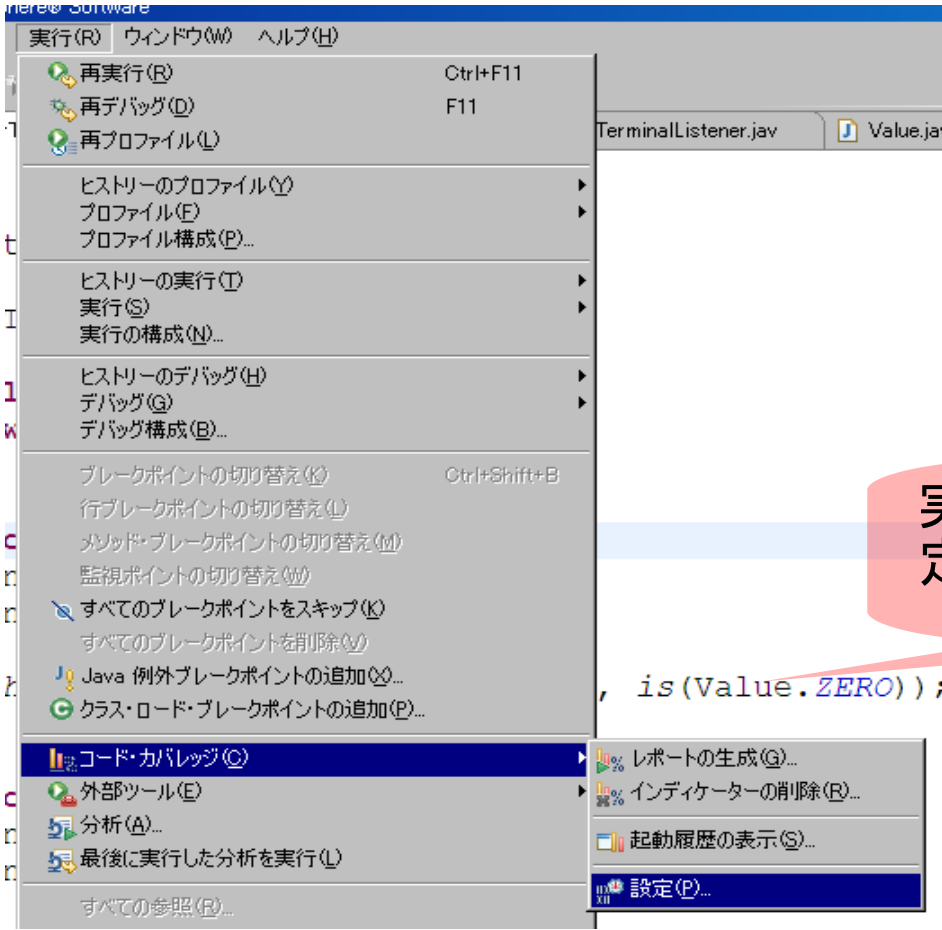
- OrTestを見てみてください。これまでは見かけなかった@Beforeというアノテーションが使用されています。

```
public class OrTest {  
    Or or;  
    @Before public void setup() {  
        or = new Or();  
    }  
  
    @Test public void 入力が0と0だと出力は0() {  
        or.getInput(0).setValue(Value.ZERO);  
        or.getInput(1).setValue(Value.ZERO);  
  
        assertThat(or.getOutput(0).getValue(), i  
    }  
}
```

- @Beforeアノテーションが付いたメソッドは、各テストメソッドが実行される前に都度実行されます。これによりOrのインスタンスを自動的に生成しておくことが可能になります。

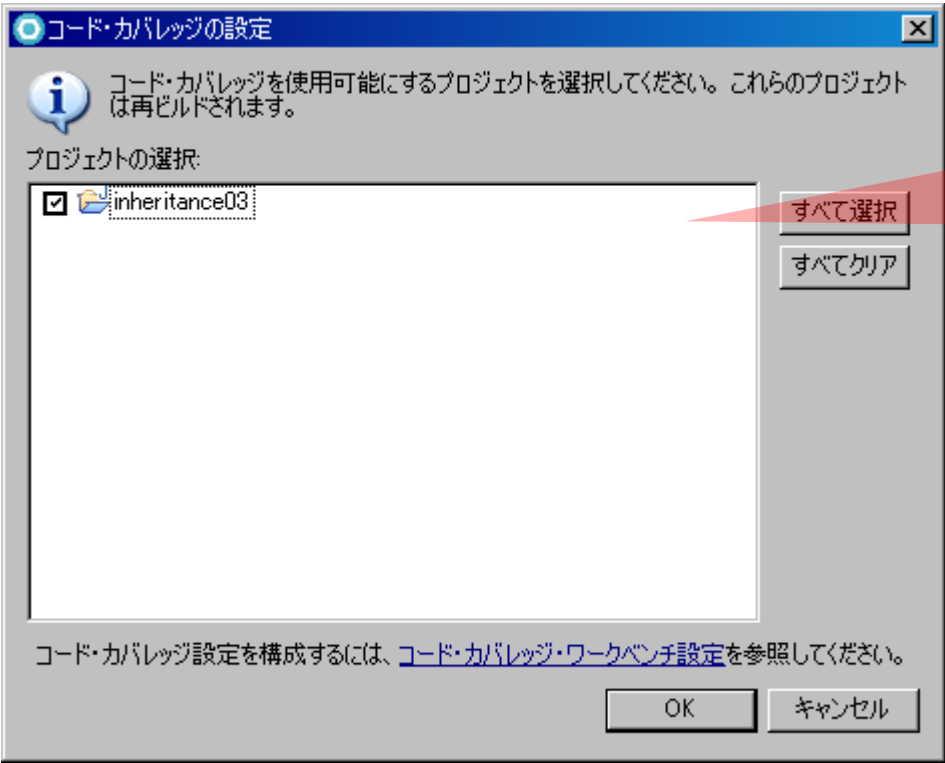
# カバレッジ

- だいぶクラスの数が増えてきました。テストが漏れている場所はないでしょうか？
- こういう場合はカバレッジが参考値になります。



実行 => コード・カバレッジ => 設定をクリックします。

# カバレッジ



カバレッジを取りたいプロジェクトがチェックされていることを確認します。

# カバレッジ

defensiveCopy01  
defensiveC  
immutable  
immutable  
inheritanc  
inheritanc  
inheritanc  
src  
oo  
コピー(C) Alt+W  
修飾名のコピー(Y)  
貼り付け(P) Ctrl+Y  
削除(D) Delete  
ビルド・パス(B)  
ソース(S) Alt+Shift+S  
リファクタリング(T) Alt+Shift+T  
インポート(I)...  
エクスポート(E)...  
更新(F)  
プロジェクトを開じる(S)  
ワーキング・セットの割り当て(A)...  
**実行(R)**  
デバッグ(D)  
プロファイル(P)

```
org.junit.Test;

class
efore
or =

est public void 入力が0と0だと出力は0
or.getInput(0).setValue(Value
or.getInput(1).setValue(Value

assertThat(or.getOutput(0).ge

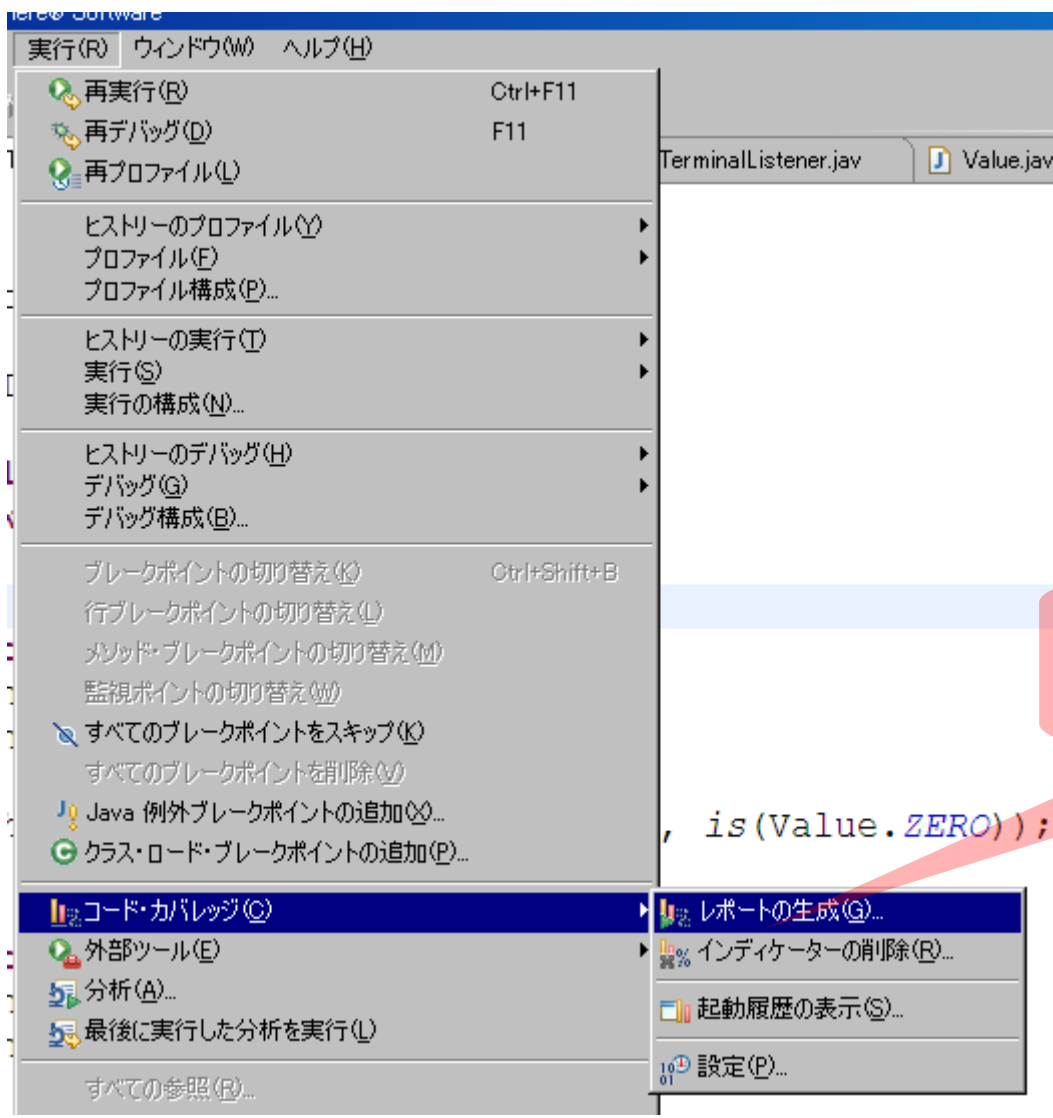
est public void 入力が0と1だと出力は1
or.getInput(0).setValue(Value
or.getInput(1).setValue(Value
```

1 Java アプリケーション Alt+Shift+X J  
2 Java アプレット Alt+Shift+X A  
3 JUnit テスト Alt+Shift+X T

プロジェクトを右クリックし、実行  
=> JUnitテストをクリックします。  
これで、プロジェクト内の全テスト  
が実行されます。



# カバレッジ



レポートの生成をクリック  
します。

# カバレッジ

コード・カバレッジ・レポート

コード・カバレッジ・レポート

起動によって生成されたコード・カバレッジ統計のレポートを作成します。

起動の選択

名前	日付
inheritance03 (アクティブ)	2012/01/31 18:12:10
proceduralCircuitSimulator01	2012/01/26 11:42:40

レポートのフォーマットおよびロケーション

☒ カバレッジ・レポート   ☐ 比較レポート

☒ ワークベンチ・レポート   ☐ HTML レポート

☒ レポートの表示 (V)

☐ レポートの表示と保存 (S):

参照 (R)...

?

実行 (R)

キャンセル

対象プロジェクトが反転していることを確認して、実行ボタンを押します。

# カバレッジ

コード・カバレッジ・レポート (2012/01/31 18:12:11) NotTest.java Value.java

コード・カバレッジ・レポート

コード・カバレッジの要約

'inheritance03' のコード・カバレッジ・レポート (生成日時: 2012/01/31 18:12:11)

要素	カバレッジ	カバーされた行数	合計行数
[inheritance03/test] oo	99%	152	153
ValueTest.java	100%	25	25
TerminalTest.java	100%	23	23
OrTest.java	100%	39	39
NotTest.java	95%	19	20
NotTest	95%	19	20
入力を設定していないと出力は不定0 : void	100%	4	4
入力が1だと出力は00 : void	100%	6	6
入力が0だと出力は10 : void	100%	6	6
NotTest0	100%	1	1
getInputの添字が1以上はエラー0 : void	67%	2	3
AndTest.java	100%	46	46
[inheritance03/src] oo	97%	77	79
Value.java	93%	27	29
Value\$3	100%	7	7
Value\$2	100%	7	7
Value\$1	100%	9	9
Value	67%	4	6
Terminal.java	100%	19	19
Or.java	100%	8	8
NotTest.java	100%	1	1

ドリルダウンして、カバレッジの低いところが調査できます。ダブルクリックするとソースに飛びます。

## カバレッジ

```
public enum Value {  
    UNKNOWN {  
        @Override public Value not() {  
            return Value.UNKNOWN;  
        }  
  
        @Override public Value and(Value that) {  
            if (that == null) throw new NullPointerException();  
            if (that == Value.ZERO) return Value.ZERO; // ZEROと0  
            else return Value.UNKNOWN;  
        }  
    }  
}
```

黄色のところは、特定条件でしか実行されていないことを意味します。確かにAndで入力にnullを与えた場合のテストが不足しているようです。

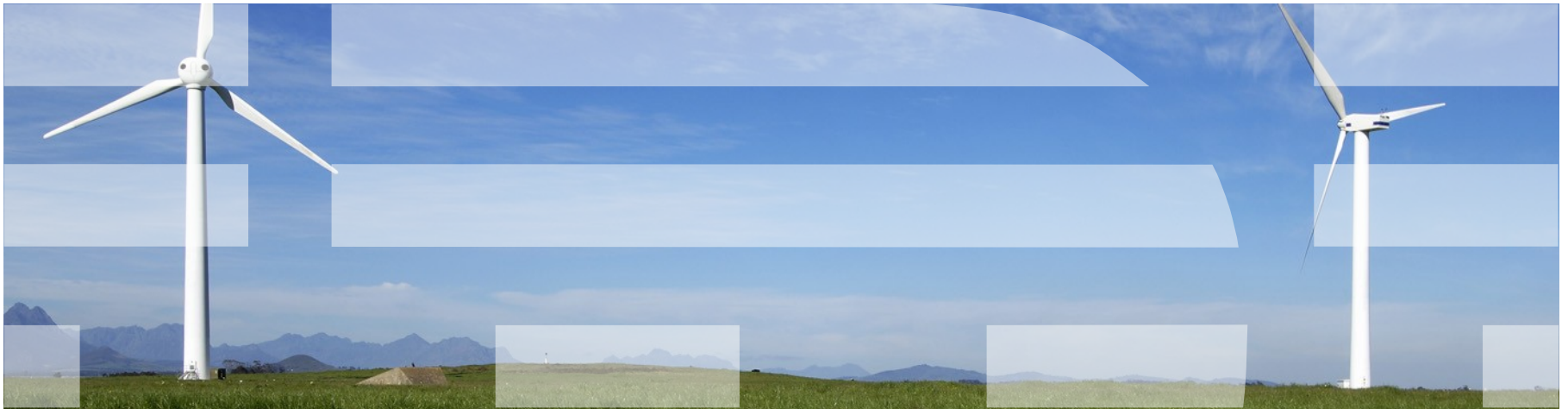
## カバレッジ

```
@Test(expected = IndexOutOfBoundsException.class)
public void getInputの添字が1以上はエラー() {
    Not not = new Not();
    not.getInput(1);
}
```

別の例です。この場合は例外が投げられることをテストしているので問題ありません。

このようにカバレッジは必ずしも100%にはなりません。著しく低いのは問題ですが、100%になることを目標とするべきではありません。あくまでテスト漏れを発見するための参考として使用します。

## 半加算器を完成させる



## 演習: 半加算器を完成させる(テストの作成)

- いよいよオブジェクト指向版の半加算器を完成させましょう。
- まずテストからです。
- プロジェクトはooHalfAdderです。
- HalfAdderTestを完成させてください。

## 演習: 半加算器を完成させる(コンパイルエラーの解消)

- テストを参考にしてコンパイルエラーを解消します。
- プロジェクトはooHalfAdder02です。



## 演習: 半加算器を完成させる(Logicクラスの変更)

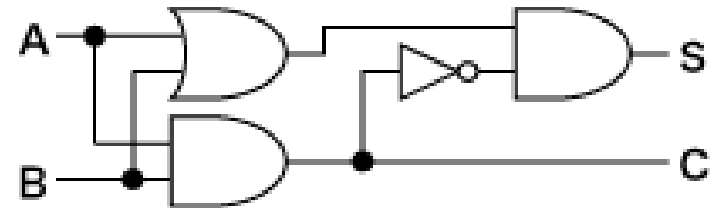
- 現在のLogicクラスは出力端子を1つしか持てません。Logicクラスを変更して、出力端子を複数持てるようにしてください。
- プロジェクトは引き続きooHalfAdder02を使用します。
- Logicクラスのコンストラクタ引数に、出力端子数を追加して、出力端子が複数持てるように修正します。

```
public Logic(int inputTerminalCount, int outputTerminalCount) {  
    this.input = new Terminal(inputTerminalCount);
```

- Logicのコンストラクタ引数が増えたことで他のクラスにコンパイルエラーが起きるようになります。コンパイルエラーを手掛りに修正してください。
- テストを実行してください。HalfAdderTest以外は成功するはずです。

## 演習: 半加算器を完成させる(HalfAdderの実装)

- HalfAdderを実装してください。
- プロジェクトは引き続きooHalfAdder02を使用します。
- ヒント: 半加算器の回路を見ながら結線します。
  - Sをoutput[0]、Cをoutput[1]と見なします。



```
public HalfAdder() {  
    super(2, 2);
```

```
    And and0 = new And();  
    And and1 = new And();  
    Or or = new Or();  
    Not not = new Not();
```

```
    // 結線する。
```

```
    input[0].connectTo(and0.getInput(0));  
    input[0].connectTo(or.getInput(0));
```

```
    ...
```

```
    and0.getOutput(0).connectTo(output[1]);  
    not.getOutput(0).connectTo(and1.getInput(1));  
    and1.getOutput(0).connectTo(output[0]);
```

## まとめ

- 出来上がったアプリケーションの設計は、当初決めた設計からはちょっと異なっています。論理回路ごとにクラスが用意されているという点は変わりませんが、値に対してもクラスが用意されており、設計2も混ざった状態になっていることが分かります。
- 今回のように単純なアプリケーションでも、このように設計は作っていく中で変化していきます。ですから次のようなことは避けるべきです。
  - 最初に長い時間をかけて設計を検討する  
どうせ途中でどんどん変わるので、あまり時間をかけても無駄です。
  - 最初の設計をもとに多量のドキュメントを作成する  
どうせ途中でどんどん変わるので、ドキュメントは無駄になります。必要最低限に留めるべきです。
- 逆に次のようにする必要があります。
  - 狭いスコープで一通り作成してみる  
今回の例ならNot回路だけに絞って、プロトタイピングを進め、その中でValueやTerminalといったクラスを導入して設計を確定させ、最後にAnd、Orを追加しました。このようにすることで設計の見直しがあっても手戻りを最小化できます。
  - 変更が容易にできるようにしておくべき。  
単体テストを用意し、コード重複を最低限に抑えて設計の変更に柔軟に対応できるようにします。

## まとめ

- 半加算器を手続き型とオブジェクト指向の両方で実装しました。
- オブジェクト指向では粒度の細かなオブジェクトを利用することで、コードの重複をうまく取り除くことが可能になります。
- 今回は端末オブジェクトを導入することで、素子の間の結線をそのままコード上に表現することができるようになりました。これにより例えば結線状態を外部ファイルに保存したり読み込んだりということも簡単にできるようになるでしょう。
- オブジェクト指向の半加算器では、入力を与えるとそれが端末オブジェクトの間で次々と配信されていき勝手に結果が得られます。まるで個々のオブジェクトに命が吹き込まれたかのように自律的に動作します。
- これに対し手続き型では、全体を見通せる神のような存在が操り人形を操作するようです。ある素子で得られた結果を別の素子に1つ1つ伝えていかなければなりません。これは規模の小さなプログラムであれば容易ですが、規模が大きくなると困難な仕事になっていくでしょう。
- 反面、オブジェクト指向のプログラムでは粒度の細かなオブジェクトが複雑に連携して動作するため、デバッグの難易度は上がります。
- オブジェクト指向と手続き型は、どちらかが絶対的に優れているというものではありません。ユースケースや環境(人員なども含む)に応じて最適な方法を選択する必要があります。

## まとめ

- not()、and()、or()をValueに用意しましたが、halfAdder()はValueに用意しませんでした。これは、NOT、AND、ORが基本となる論理回路なのに対し、半加算器は基本となる論理回路を配線するだけで実現できるためです。not()を導入したのは、Notクラスからswitch文を取り除くためでした。halfAdder()を用意しなくても、HalfAdderクラスにはswitch文が必要無いことに注意してください。
- LogicはgetInput()、getOutput()で、自分が持っているTerminalオブジェクトを返しています。Terminalオブジェクトはミュータブルなオブジェクトなので、防御的コピーが必要なのではないのでしょうか？  
これは理想的には、その通りです。今の設計だと、例えばNotからgetOutput()で出力端子を取得しsetValue()を呼ぶことで、Notクラスの外で勝手に状態を設定できてしまいます。  
**間違った使い方を未然に防ぐことができない、あるいは使用するのに余計な知識や前提が必要な場合、そのクラス設計は優れているとは言えません。**  
これ以外にも出力端子だけでなく、入力端子に対してもconnectTo()が呼び出せたり、出力端子同士をconnectTo()できたりなど、まだまだ荒削りな部分があります。これらの問題を防ぐには、例えばsetValue()を非公開にしたり、入力と出力とで端子のクラスを違うものにしたりする必要があります。  
ただしValueのような値オブジェクトと異なり、Terminalのような「機能」を抽象化したオブジェクトや、副作用を利用する(例えば画面やファイルに出力するためのオブジェクト)の場合、防御的コピーを行わないケースがあります。これはケースバイケースで明確な基準は残念ながらありません。

## 自由研究

- 今回作成した半加算器を組み合わせると、3つの入力から2つの出力を生成する全加算器を作成することができます。全加算器を実装してみてください。
- 全加算器と半加算器があると、複数ビットの加算器が実現できるようになります。複数ビットの加算器を実装してみてください。