# オブジェクト指向技術'11 ＃2 — メタプログラミング

久野　靖*

2011.4.21

## 1　メタプログラミングとは?

メタ (meta) とはもともとはギリシャ語で「間に」「変化して」「後退して」などの意味を持つ言葉らしいですが、ともかく今日では「超越した」「高次の」を表す接頭語として使われています。

それで、高次のプログラミングって何でしょう。普通のプログラムでは、プログラムが動いて、データを加工 (読み込み、出力) します。これに対し、メタプログラミングでは**プログラムが動いて、プログラムを加工します。**

それは、どういう利点があるのでしょうか? 1つは、素のプログラミング言語だけでは提供できない (ないし、提供できるけれど弱点がある) 機能をうまく提供できるというものです。たとえば、Lispではプログラムの表現もプログラムが扱うデータもともにS式で表現されるため、古くからメタプログラミングが行われて来ました。それを構造化したものがマクロです。

たとえば、次の式はif-then-else式を表している (言語はCommonLisp)。

```
(if 条件 THEN部 ELSE部)
```

では、「(ifnot 条件 本体)」という新しい構文を入れたいものとします。次のように関数定義すればいいでしょうか?

```
(defun ifnot (c b) (if (not c) b))
```

これはダメです。というのは、関数では引数は本体の実行に入る前に評価 (実行) されてしまうので、条件が成り立っていない時だけ本体を実行するということにならないから。これに対し、マクロを使うと「まずあるべき式を組み立て、続いてそれを実行する」という順番になるのでうまく行くわけです。

```
(defun ifnot (c b) (list 'if (list 'not c) b))
```

マクロがどのように結果を組み立てるかを見るのは macroexpand-1 が使えます。

```
> (macroexpand-1 '(ifnot (> x 100) (incf x)))
(IF (NOT (> X 100)) (INCF X))
```

これを実際動かしてみます。

```
> (setf x 10)
10
> (ifnot (> x 100) (incf x))
11
> (ifnot (> x 10) (incf x))
NIL
> x
11
```

*経営システム科学専攻

マクロはコンパイルすると展開された状態でコンパイルされるので、高速に実行されます。このようなことが可能なのはやはり、Lisp の「プログラムもデータも S 式) という特別な性質が大きく貢献していると言えます。

しかし、メタプログラミングと言った場合、このような「典型的な」もの以外にもいろいろなレベルがあります。今回はそのようなものを、とくにオブジェクト指向の観点から、いくつか見て行くことにします。

## 2　リフレクションと java.lang.reflect

## 3　リフレクションとは

リフレクション (reflection、自己反映計算) とは、実行中のコードが実行系の情報にアクセスしたり実行系の状態・動作を変更したりできるような機能を言います。何のためにそのようなことをするのでしょうか。それは、たとえば次のような用途が考えられます。

- デバッガなどのツールでは、実行系の内部を調べて「どこで止まったか」「どんな例外が出たか」などを調べたいかも知れない。
- システムを拡張するために、たとえば「任意の手続き呼び出しを、それと同等な遠隔サービスを呼ぶような手続き呼び出しに変換して差し替える」などの操作をしたいかもしれない。
- 拡張可能言語 (言語そのものの機能を追加したり修正できる) を構成することで、書きたいコードが容易に書けるようにしたいかも知れない。

ここまではオブジェクト指向に固有の話ではなかったのですが (たとえば初期のリフレクション機能を持つ言語の研究は Lisp 系の言語でした)、オブジェクト指向言語では「オブジェクト」というまとまりがあるため、リフレクションが扱いやすいという側面があります。

具体的には、クラス方式のオブジェクト指向言語では、各オブジェクトの動作はクラスという形で記述します。これまでクラスというのはあくまでもコンパイル時にだけあるものだと考えていたかも知れませんが、実行時にも「クラスというオブジェクト」があるものとして、そのオブジェクトに対してさまざまな操作を施すことは、リフレクションにほかなりません。実は Java では、すべてのクラス `X` に対して、`X.class` という書き方で「そのクラスのオブジェクト」を参照できます。

なお、上の記述でいうと「普通のオブジェクト」と対比して、「クラス」は「オブジェクトを作り出す」つまり「メタ」なものですから、「クラスオブジェクト」は「メタオブジェクト」であるというふうに言うこともできます。したがって、クラスオブジェクトがどのような操作呼び出しを持っていて利用できるか (プロトコルの情報) のことを、「メタオブジェクトプロトコル」と呼ぶこともあります。

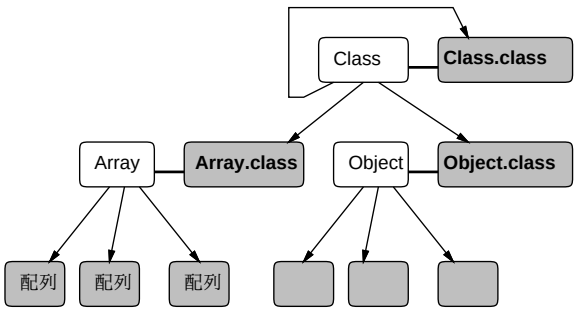


図 1: Java のメタクラスの関係

もう 1 つ、クラス方式の言語であれば、「クラスというオブジェクト」も、何らかのクラスから生成されたものだと考えるのが自然です。そこで、「クラスというオブジェクトを生成するクラス」の

ことを「メタクラス」と呼びます。ということは、メタオブジェクトプロトコルはメタクラスで定義され実装されているようなメソッド群、ということになるわけです。Java ではメタクラスの名前はズバリ「Class」です。そしてこれはすべてのクラスオブジェクトのインスタンスを生成しているわけですから、Class.clas もそうだということになります (図??)。分かりにくいと思いますが、メタを扱っていると最後はこういう循環関係が出て来ることがよくあります。

なお、Java のクラス Class は java.lang パッケージにありますが、リフレクションに関するそのほかのクラスの多くは java.lang.reflect パッケージにまとめられています。API ドキュメントで確認するときは注意してください。

### 3.1 Java のリフレクション機能

Java のリフレクション機能は基本的に、既存のクラスが持っている情報を取り出して来ることですが、メソッドオブジェクトも取り出すことができ、そのメソッドを実際に呼び出すことができるので、結構面白いことができます。具体的には次のような機能があります。

- 「クラス名.class」または「Class.forName(文字列)」でクラスオブジェクトが取得できる。
- クラスオブジェクトの getConstructors()、getMethods()、getFields() でそのクラスに定義されているコンストラクタ、メソッド、フィールドのオブジェクトが取れる。
- コンストラクタは newInstance() で呼び出してオブジェクトを実際に作ることができる。メソッドは invoke() で呼び出して動かすことができる。フィールドは get()、set() で値を取り出したり設定できる。

実例を見てみましょう。

```
import java.util.*;
import java.lang.reflect.*;

public class Sample21 {
  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    while(true) {
      try {
        System.out.print("Class Name? ");
        String cname = sc.nextLine();
        if(cname.equals("")) { break; }
        Class cls = Class.forName(cname);
        Constructor[] cons = cls.getConstructors();
        for(int i = 0; i < cons.length; ++i) { System.out.println(i + ": "+cons[i]);}
        System.out.print("Constructor Number? ");
        int cno = sc.nextInt(); sc.nextLine();
        Object obj = cons[cno].newInstance(new Object[]{});
        Method[] meths = cls.getMethods();
        for(int i = 0; i < meths.length; ++i) { System.out.println(i + ": "+meths[i]); }
        System.out.print("Method Number? ");
        int mno = sc.nextInt(); sc.nextLine();
        Object res = meths[mno].invoke(obj, new Object[]{});
        System.out.println("Result Class:"+(res.getClass()));
        System.out.println("Result: "+res);
```

```
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
}

class Sample21Test {
  int val;
  public Sample21Test() { val = 1; }
  public Sample21Test(int i) { val = i; }
  public Sample21Test add() {
    return new Sample21Test(val+1);
  }
  public Sample21Test sub() {
    return new Sample21Test(val-1);
  }
  public String toString() {
    return "Sample21Test("+val+")";
  }
}
```

このプログラム本体はクラス名を指定するとそのクラスのコンストラクター一覧を表示し、その中から番号を選んでインスタンスを生成した後、メソッド一覧を表示して番号を指定するとそのメソッドが呼ばれる、というふうになっています。Sample21Test は実験用のクラスです (Java の標準クラスでもテストできますが、ごちゃごちゃで見にくいので)。

実際に動かしてみましょう。

```
% javac Sample21.java
% java Sample21
Class Name? java.lang.Object
0: public java.lang.Object()
Constructor Number? 0
0: public final native void java.lang.Object.wait(long) throws ...
1: public final void java.lang.Object.wait(long,int) throws ...
2: public final void java.lang.Object.wait() throws ...
3: public boolean java.lang.Object.equals(java.lang.Object)
4: public java.lang.String java.lang.Object.toString()
5: public native int java.lang.Object.hashCode()
6: public final native java.lang.Class java.lang.Object.getClass()
7: public final native void java.lang.Object.notify()
8: public final native void java.lang.Object.notifyAll()
Method Number? 4
Result Class:class java.lang.String
Result: java.lang.Object@de6ced
Class Name? Sample21Test
0: public Sample21Test()
1: public Sample21Test(int)
Constructor Number? 0
0: public Sample21Test Sample21Test.add()
1: public java.lang.String Sample21Test.toString()
```

```
2: public Sample21Test Sample21Test.sub()
3: public final native void java.lang.Object.wait(long) throws ...
4: public final void java.lang.Object.wait(long,int) throws ...
5: public final void java.lang.Object.wait() throws ...
6: public boolean java.lang.Object.equals(java.lang.Object)
7: public native int java.lang.Object.hashCode()
8: public final native java.lang.Class java.lang.Object.getClass()
9: public final native void java.lang.Object.notify()
10: public final native void java.lang.Object.notifyAll()
Method Number? 0
Result Class:class Sample21Test
Result: Sample21Test(2)
Class Name?
%
```

確かに、オブジェクトを作ったりメソッドを呼んだりができていることが分かります。

### 3.2 例題: GUI ビルダーもどき

先の例はいかにも実験用で実用的な感じがしませんね。もう少し実用ぽいものとして、GUI 生成プログラムのようなものを見てみましょう。

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;

public class Sample22 extends Frame {
  int x1, y1, x2, y2, x0, y0, width, height;
  Component c0 = null;
  Label l0 = new Label("Component:");
  TextField t0 = new TextField();
  Button b0 = new Button("Create");
  Button b1 = new Button("Move");
  Label l1 = new Label();
  Choice c2 = new Choice();
  Label l2 = new Label("String:");
  TextField t2 = new TextField();
  Button b2 = new Button("Call");
  Method[] meths; int[] maps;
  public Sample22() {
    setLayout(null); setSize(600, 600);
    add(l0); l0.setBounds(10, 40, 60, 30);
    add(t0); t0.setBounds(80, 40, 200, 30);
    add(b0); b0.setBounds(290, 40, 60, 30);
    add(b1); b1.setBounds(360, 40, 40, 30);
    add(l1); l1.setBounds(10, 70, 380, 30);
    add(c2); c2.setBounds(10, 110, 200, 30);
    add(l2); l2.setBounds(10, 140, 60, 30);
```

```
add(t2); t2.setBounds(80, 140, 200, 30);
add(b2); b2.setBounds(290, 140, 40, 30);
b0.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    try {
      c0 = (Component)Class.forName(t0.getText()).newInstance();
      Sample22.this.add(c0);
      c0.setBounds(x0, y0, width, height);
      t0.setText("");
      meths = c0.getClass().getMethods();
      maps = new int[meths.length];
      c2.removeAll();
      for(int i=0,k=0; i < meths.length; ++i) {
        Class[] arg = meths[i].getParameterTypes();
        if(arg.length == 1 && arg[0] == String.class) {
          c2.add(meths[i].toString());
          maps[k++] = i;
        }
      }
    } catch(Exception ex) {
      l1.setText(ex.toString());
    }
  }
});
b1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if(c0 != null) {
      c0.setBounds(x0, y0, width, height);
    }
  }
});
b2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    try {
      Method m = meths[maps[c2.getSelectedIndex()]];
      m.invoke(c0, new Object[]{t2.getText()});
      t2.setText("");
    } catch(Exception ex) {
        l1.setText(ex.toString());
    }
  }
});
addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    x1 = x2 = e.getX(); y1 = y2 = e.getY();
    calcbox(); repaint();
  }
```

```
      public void mouseReleased(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY();
        calcbox(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY();
        calcbox(); repaint();
      }
    });
  }
  private void calcbox() {
    x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
    width = Math.abs(x1-x2);
    height = Math.abs(y1-y2);
  }
  public void paint(Graphics g) {
    g.drawRect(x0, y0, width, height);
  }
  public static void main(String[] args) {
    Frame f = new Sample22();
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
  }
}
```

このプログラムは、java.awt.*にある GUI 部品を画面上に「その場で」入れて、そのメソッドを呼び出すことができます (ただし、呼び出せるのは 1 個の文字列を持つメソッドだけ)。部品を置く位置は画面上でマウスでドラグすることで指定するようになってます。

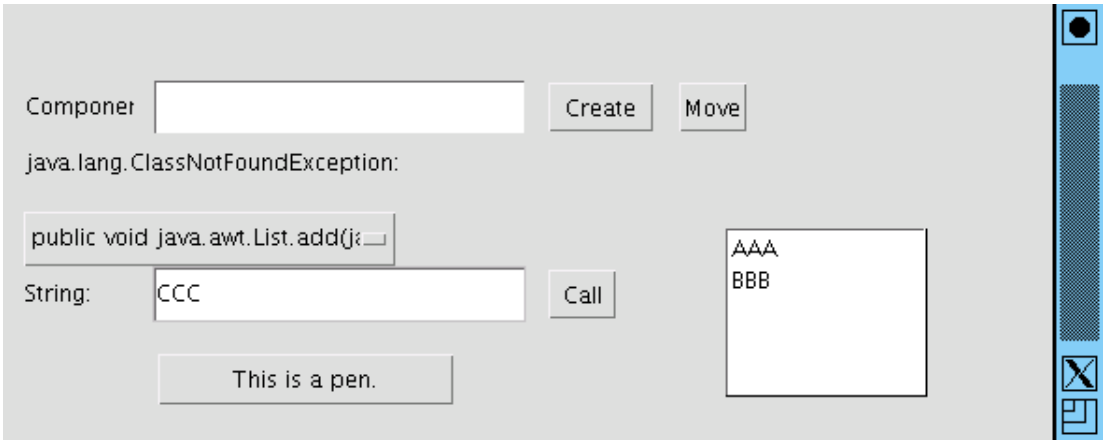


図 2: GUI ビルダもどき

**演習 1** 上記の 2 つのデモのソースファイルをコピーしてきて、動かしてみなさい。GUI 部品については、java.awt.*の API ドキュメントを参照のこと。

## 4 テストと TestNG

プログラムのソースコードを直接いじるのでなくても、「プログラム自体について扱う」ようなコードはやっぱりメタプログラミングだと言えるでしょう。その 1 つの例として、「プログラムの正しさを扱う」というテーマがあります。具体的には、プログラムをテストすることで、テストの記述とは「このプログラムを動かしたら、結果はこうなるはずだけれど、確かにそうなるかどうか、実際に動かして確認する」ということです。どうです、メタっぽいでしょう?

テストは「実際にコードを動かして見る」ことで行うので、そのためのツールないしテスト環境があることが望まれます。とくに回帰テスト (regression test、ソフトを変更するたびに、これまでに作成したテスト群すべてを再度実行して、すべてパスすること、つまりこれまで動いたものを壊していないことの確認をするテスト) を実施するには、毎回大量のテストを実施するので、人手でやるのは現実的でありません。

ここでは、Java 向けのそのようなテストツールである TestNG を試してみましょう。試す題材は「整列」ルーチンとういことにします。さっそく例を見てみましょう。

```
import org.testng.annotations.*;
import static org.testng.Assert.*;

public class TestSample1 {
  // sort a[i]..a[j] portion of given array a.
  public static void bubbleSort1(int[] a, int st, int ed) {
    for(int i = st; i < ed; ++i) {
      if(a[i] > a[i+1]) { swap(a, i, i+1); }
    }
  }
  private static void swap(int[] a, int i, int j) {
    int x = a[i]; a[i] = a[j]; a[j] = x;
  }

  @Test
  public void smokeTestBubble1() {
    int[] arr = {90, 28, 1, 73, 5, 44};
    bubbleSort1(arr, 0, arr.length-1);
    assertTrue(arr[0] == 1); // 先頭は 1
    assertTrue(arr[arr.length-1] == 90); // 最後は 90
  }
}
```

ここで最初のメソッドが私が適当に書いたバブルソートです。整数の配列と、その整列される範囲の先頭/終端の添字を指定すると、その範囲を昇順に整列します。2 番目のメソッドは下請けで「配列と 2 つの添字を指定するとその 2 要素を交換」します。ここまでが本体コードです。

その後にある`@Test`とついたメソッドがテスト用メソッドになります。テスト用メソッドはいくつ作っても構いません。ここでは通電テストとして、5 要素の配列を作った後、先頭と末尾に最小と最大が来ているかどうかチェックしています。`asserTrue`は TestNG が提供しているメソッドで、論理式が真であればその項目は合格という意味になります。

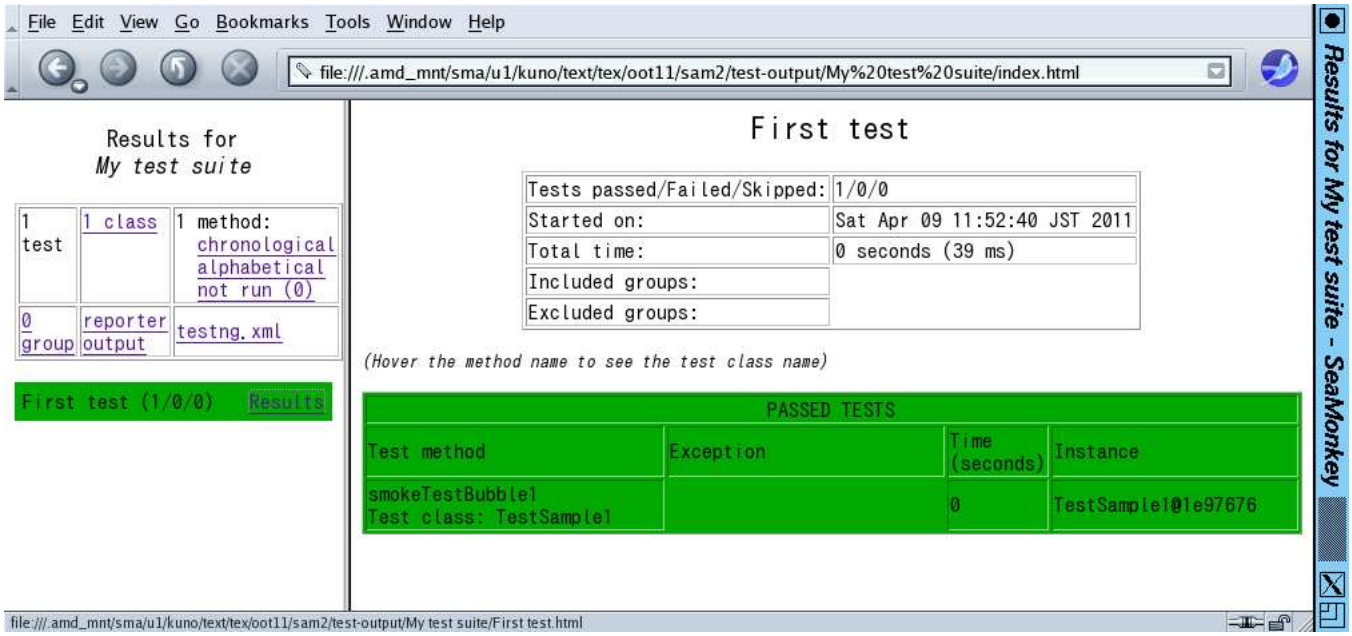


図 3: TestNG の出力例

ではこれをコンパイルします。CLASSPATH の設定は最初に 1 回だけやればよいです。

```
% export CLASSPATH = .:/u1/kuno/work/testng.jar
% javac TestSample1.java
```

次に、テストを記述する XML ファイルが必要です。今回は決め打ちで最大限簡単にした次のものを使ってください。

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="My test suite">
  <test name="First test">
    <classes>
       <class name="TestSample1" />
    </classes>
  </test>
</suite>
```

では、実行します。

```
% java org.testng.TestNG test1.xml
[TestNG] Running:
  /u1/kuno/text/tex/oot11/sam2/test1.xml
===============================================
My test suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
%
```

正しく実行されました。テスト結果は test-output というディレクトリの下の「My test suite」にできていますから、そこをブラウザで開いて確認します (図**??**)。

ちなみに、このようなテストツールを使って小刻みにテスト (むしろ実行例と呼んだ方が適切という説もある) を追加しつつ、テスト追加→コード追加→リファクタリングというサイクルを繰り返す

ことでソフトウェアを開発して行く手法 (Test Driven Development、TDD) が最近注目されてます。そのような手法には、上で見たようなこまめにテストを実行できる仕組みが不可欠なことはお分かり頂けると思います。

**演習 2** 上記のテストをファイル (Java ソース、XML) をそっくりコピーしてそのまま動かしてみなさい。動いたら、次のテストを追加してみなさい。それぞれ、新しいテスト用メソッドを作って、先頭に@Test とつけること。

a. メソッド `swap` の通電テスト
b. その他好きな種別のテスト
c. 別の整列アルゴリズムで整列を書いて、それをテスト

# 5 Java のアノテーション機能

## 5.1 アノテーションとは

ところで、先の例題に出て来た「`@Test`」というのは何でしょう? Java 言語は JDK 1.5.0 から、クラスやメソッドなどにこのような「付加情報」を追加する機能が入りました。これをアノテーション (annotation、注記) と呼びます。

アノテーションと一般的に言った場合、文章やコードなどに注記を付加することを言い、それを利用する方法としては「人間が読む」「ツールで別途取り出して集めたり加工して利用する」などが普通です (たとえば javadoc 形式のコメントが典型例)。その場合、プログラムのソースコードにアノテーションを付加したとしても、その情報はコンパイラは取り扱わないので、プログラム上で何らかの作用を及ぼすようにするのなら、プリプロセサ (ソースを加工してコンパイル前のプログラムに変更を及ぼすようなプログラム) を使うのが普通でした。しかしその場合、そのプリプロセサ用の記法はプログラムごとに違うのが普通なので、さまざまな書き方が現れて不統一だったり、プリプロセスしていないコードではその機能は使えないなどの問題がありました。

Java のアノテーションは上のような問題を解決すべく、次のような機能を提供しています。

- クラス、メソッドなどの前に「@名前…」のような形で書くことにより、それらに対する言語本体では表現されていない「追加」の情報を記入できる。
- 予め標準で定義されているもののほかに、ユーザが定義することもできるので、さまざまな用途ごとにそれ向けのアノテーションを作って使える。さらにそのとき、どのようなパラメタを持つかも指定できる。
- アノテーション定義時に「コンパイル時、クラスファイル、実行時のどの段階まで情報を保持するか」を指定できる。

## 5.2 標準定義されているアノテーション

アノテーションの中には標準 API で定義され、コンパイラ等に処理が組み込み済みのものもあります。

たとえば、`@Override` というアノテーションはメソッドに付加でき、このメソッドが親クラスのメソッドをオーバライドしていることを明示します。つまり、コンパイラはこのメソッド定義のところで、親クラスから同名のメソッドが継承されているかどうかを調べ、継承されていなければ (つまりオーバライドになっていなければ) エラーを出します。実は Java などのクラス方式の言語で非常によくある間違いが、「メソッド定義時にタイプミスによって上書きすべきメソッドを上書きし損なう」というものであり、そのような間違いを防止するためにこのアノテーションをつけておくことはとても訳に立つわけです。

```
% cat AnnotTest1.java
public class AnnotTest1 {
  @Override                                    ←@Override
  public String ToString() { return "???"; } ← T は小文字が正解
  public static void main(String[] args) {
    System.out.println(new AnnotTest1().toString());
  }
}
% javac AnnotTest1.java
AnnotTest1.java:2: method does not override or implement a
    method from a supertype
  @Override
  ^
1 error
% vi AnnotTest1.java
% cat AnnotTest1.java
public class AnnotTest1 {
  @Override
  public String toString() { return "???"; }
  public static void main(String[] args) {
    System.out.println(new AnnotTest1().toString());
  }
}
% javac AnnotTest1.java
% java AnnotTest1
???
%
```

このほかに、メソッドが非推奨であることを表す@Deprecated、クラスやメソッドなどの範囲についてコンパイル時の警告を種別を指定して抑制する@SuppressWarnings などのアノテーションが標準で用意されています。

### 5.3 アノテーションを定義する

では、自前でアノテーションを定義する方法についても説明しておく。アノテーションは構文としてはJavaのインタフェースと同じ形で定義するが、ただしキーワードとして@interface を使います。

```
public @interface Test { }
```

この場合、@Test というアノテーションは「ついているかどうか」だけが問題となるようなアノテーション (マーカアノテーション) になります。

```
@Test
アノテーションが付加されるものの定義…
```

しかし、用途によってはさらにアノテーションに対してさまざまな値を指定したいことがあります (フルアノテーション)。この場合、アノテーションはインタフェースの構文で定義するので、引数なしのメソッドを使って値とその型を指定します。

```
public @interface Test1 {
  int id();
  String name();
  String comment() default "no comment";
}
```

なお、上記のように default を指定することで、その値に標準値を設定できます。

```
@Test1(id = 100, name = "abc")
アノテーションが付加されるものの定義…
```

アノテーションの値として使える型は、基本型、文字列、クラス型、列挙型、アノテーション型、および前記の型の配列のみです。

3 番目の場合として、1 つだけ値を持たせたい場合は、その名前を value にしておくことで、いちいち「名前=値」の形で指定しなくても済むようになります (単一値アノテーション)。

```
public @interface Test2 {
  String value comment();
}
```

これを使うときは直接値だけをかっこ内に書けば済みます。

```
@Test2("this is a comment...")
アノテーションが付加されるものの定義…
```

先に、アノテーション情報が残される段階を指定できると書きましたが、このような情報は「アノテーションに対するアノテーション」(メタアノテーション) で指定します。

```
@Retention(RetentionPolicy.SOURCE) --- コンパイル時まで利用可
@Retention(RetentionPolicy.CLASS) --- クラスファイルまで利用可
@Retention(RetentionPolicy.RUNTIME) --- 実行時に利用可
```

このほかのメタアノテーションとして、次のものがあります。

```
@Documented --- 当該アノテーションはドキュメントに現れる
@Inherited ---  当該アノテーションは継承される
@Target(ElementType.METHOD) --- メソッドにつくことを指定
```

@Target に指定する値は列挙型 ElementType の値 (またはその配列…複数種類につけられるようにする場合) です。

では、さっきやった TestNG のように@Test というアノテーションを作ってそれがついているメソッドだけを実行させてみます (もちろん、TestNG にはもっと沢山機能がありますが)。

```
import java.lang.annotation.*;
import java.lang.reflect.*;

public class AnnotTest2 {
  public static void main(String[] args) throws Exception {
    int passed = 0, failed = 0;
    for(Method m: Class.forName(args[0]).getMethods()) {
      if(!m.isAnnotationPresent(Test.class)) { continue; }
      System.out.println("Testing: " + m.getName());
      System.out.println("comm: " + m.getAnnotation(Test.class).value());
```

```
      try {
        m.invoke(null); ++passed;
      } catch(Throwable ex) {
        System.out.println("fail: " + ex.getCause()); ++failed;
      }
    }
    System.out.printf("passed = %d, failed = %d\n", passed, failed);
  }
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Test {
  String value() default "no comments...";
}

class AnnotTest2Test {
  public static void m1() { }

  @Test
  public static void m2() { }

  @Test("throws exception")
  public static void m3() {
    throw new RuntimeException("???");
  }
}
```

見て分かるように、`@Test` は文字列つきの単一値アノテーションとして、メソッドのみにつけられ、実行時に利用可能にします。そして `main()` では、コマンド引数で指定された名前を持つクラスのメソッドを順次取り出して来て、`@Test` がついているものがあればその文字列を表示してから実行します。実行の様子も見てみましょう。

```
% javac AnnotTest2.java
% java AnnotTest2 AnnotTest2Test
Testing: m2
comm: no comments...
Testing: m3
comm: throws exception
fail: java.lang.RuntimeException: ???
passed = 1, failed = 1
%
```

機能の豊富さに違いはあっても、TestNG でやっていることもこれと原理的に同じだということは納得頂けるかと思います。

**演習 3** 上の例をコピーしてきて動かせ。動いたら、アノテーションの種類や持たせる情報を増やして試してみよ。

### 5.4 フルアノテーション、コンパイル時処理

前項のように、リフレクションを使って実行時にアノテーションを処理することができるが、JDK 1.6 からはコンパイル時にコンパイラの中でアノテーション処理を動かすことができるようになった。ただし、これを行うのは結構面倒くさいので予め予告しておきます。

今回の例ではコンパイラからアノテーション処理を呼ぶため、アノテーション処理と処理対象アノテーションとテストクラスを分離しました。まず処理対象アノテーションは次のものとします。

```
// TodoList.java
import java.lang.annotation.*;

@Target({ElementType.METHOD,ElementType.TYPE})
@interface TodoList {
  Todo[] value();
}
@interface Todo {
  String value();
}
```

つまり、TodoList は型とメソッドにつけられ、その値として Todo の配列を持ち、そして Todo の値は文字列、とういこと。

では、これを処理する方法に進む。そのためには、クラス AnnotationProcessor のサブクラスを作成し、それに対して「どのアノテーションを処理するか」を指定するとともに、メソッド process() をオーバライドする。これはアノテーションのかたまりが見つかるごとに、アノテーション群とそれを含む環境をパラメタとして外側から呼び出される。

```
// AnnotProc.java
import java.util.*;
import java.lang.annotation.*;
import java.lang.reflect.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
import javax.tools.*;

@SupportedSourceVersion(SourceVersion.RELEASE_6)
@SupportedAnnotationTypes("TodoList")
public class AnnotProc extends  AbstractProcessor {
  @Override
  public boolean process(Set<? extends TypeElement> annots,
                         RoundEnvironment env) {
    for(TypeElement annot: annots) {
      for(Element elt: env.getElementsAnnotatedWith(annot)) {
        TodoList lis = elt.getAnnotation(TodoList.class);
        Messager msg = processingEnv.getMessager();
        for(Todo t: lis.value()) {
          msg.printMessage(Diagnostic.Kind.NOTE, t.value(), elt);
        }
      }
```

```
      }
      return true;
    }
  }
```

呼び出された時にやることは、まず各アノテーション種別ごとに、その種別のアノテーションがついている要素を取り出し、そこから TodoList アノテーションを取り出す。次に、コンパイラの出力にメッセージを出すために親クラスの変数 processingEnv に入っている ProcessingEnv オブジェクトからメッセージャを取り出し、ここに対して各 Todo 項目の文字列を出力する。非常に面倒ですよね…
では、最後にデモ用にアノテーションをつけたクラスを示す。

```
// AnnotTest3.java
import java.lang.annotation.*;

@TodoList({@Todo("more realisic one needed!")})
public class AnnotTest3 {
  @TodoList({
    @Todo("write this method!"),
    @Todo("think carefully!"),
    @Todo("work hard!")})
  void m1() { }
}
```

これを動かしたようすは次の通り。

```
% javac TodoList.java
% javac AnnotProc.java
% javac -processor AnnotProc AnnotTest3.java
AnnotTest3.java:4: Note: more realisic one needed!
public class AnnotTest3 {
       ^
AnnotTest3.java:9: Note: write this method!
  void m1() { }
       ^
AnnotTest3.java:9: Note: think carefully!
  void m1() { }
       ^
AnnotTest3.java:9: Note: work hard!
  void m1() { }
       ^
%
```

**演習 4** アノテーションを使って何か面白いことをしてみなさい。