プログラミング言語論'10 ＃5

久野　靖*

2010.5.13

1　今回の内容

□ ＋αな話題

- 前回の積み残し (Clojureを使って) --- マクロ、Java連携、マルチメソッドの実際
- スクリプト言語、組み込みスクリプト --- JavaScript
- 総称的プログラミング --- Java Generics、C++ Templates
- Aspect Oriented Programming --- 概論+AspectJ

2　Lisp系およびClojureの続き…

□ 疑問:　なぜこんなヘンな言語があって使われているのか?

- 構文が「(...　(...　))」みたいなのしかない、しかもそれがプログラムが扱うデータでもありプログラムでもある
- プログラムを扱うプログラムが簡単に書ける

```
(eval S式) → S式をClojureプログラムとして実行
(eval '(+ 1 2 3)) → 6
(list '+ 1 2 3)) → (+ 1 2 3)
(eval (list '+ 1 2 3)) → 6
(def x 4)
(eval (list '+ 1 2 x)) → 7
```

- これを系統的にしたのがマクロ→拡張可能言語
- そのほか、「弱い型の言語」「GCのある言語」「記号処理」などの側面が大きかったが、今では他の言語も結構ある
- 記号処理についてはやっぱりまだLispに1日の長
- 例:　数式を微分する

```
(defn diff [exp var]
  (cond
    (nil? exp) nil
    (number? exp) 0
    (symbol? exp) (if (= exp var) 1 0)
    (= (first exp) '+)
      (cons '+ (map (fn [x] (diff x var)) (rest exp)))
    (= (first exp) '*)
      (list '+
        (list '* (diff (nth exp 1) var) (nth exp 2))
        (list '* (nth exp 1) (diff (nth exp 2) var)))))
```

```
(diff '(* x x) 'x)
→ (+ (* 1 x) (* x 1))
(diff '(+ (* x x) (* a x) b) 'x)
→ (+ (+ (* 1 x) (* x 1)) (+ (* 0 x) (* a 1)) 0)
```

□ 以下Clojureの積み残し:　マクロ、Java連携、マルチメソッド

2.1　マクロ

□ Lisp系の言語 --- 構文が単純 (全部S式) →マクロによる拡張が自由

- Lisp系のマクロの動作:　引数をそのままの形でマクロに渡す→マクロは式を組み立てて返す→その返された式を改めてLisp処理系が評価
- よくある例:　unlessを作る --- 関数ではだめ

```
(defn unless [cond body] (if cond nil body))
(def x 0)
(unless (odd? x) (print "Y")) → Yをプリント
(unless (even? x) (print "Y")) → Y をプリント(!!)
```

- 関数は引数を常に評価してしまう→副作用があればそれが起きてしまう (副作用がないとしても計算は起きてしまう)
- マクロで定義すれば「そのまま」取り扱われるのでこの問題がない

```
(defmacro unless [cond body]
  (list 'if cond nil body))
(unless (even? x) (print "Y")) → Yは打ち出されない
(macroexpand-1
  '(unless (even? x) (print "Y")))
→ (if (even? x) nil (print "Y")) …これを実行
```

□ マクロは「Lispプログラムを組み立てる関数」

- →実際には「ほとんどLispプログラムをそのまま書くが、一部だけパラメタとして受け取ったものを埋め込む」形
- →バッククォート記法ないし構文クォート (上記の作業を簡単にこなす仕掛け)

```
`(…そのまま… ~x …そのまま…) ←要素埋め込み
`(…そのまま… ~@l) ←リストの残りを接合
```

- これを使うと先のunlessは次のように書ける

*筑波大学大学院経営システム科学専攻

```
(defmacro unless [cond body]
  (if ~cond nil ~body))
```

□ マクロを使うと「言語の自在な拡張」が可能になる ← Lisp 族ではプログラムがすべて S 式というデータの形になっていることがそれを可能にしている

□ マクロによくある問題→記号の捕捉 (capture)

- マクロの中で識別子を導入したときに起こる

```
(defmacro myor [e1 e2]
  `(let [tmp ~e1] (if tmp tmp ~e2)))
(def tmp 0)
(myor (> tmp 0) tmp) → ???
```

- 普通の Lisp だとこれは次のように展開

```
(let [tmp (> tmp 0)] (if tmp tmp tmp)) → true
```

- 基本的な問題: マクロの中で作業用に使っている tmp とパラメタで渡されて埋め込まれる tmp とか衝突/混同
- よくある対処法: マクロ中では「普通使わなそうな」名前を使う→根本的な解決でない
- Scheme の synatx-case --- 非常にややこしい
- Clojure のアプローチ: ランダム番号つきの名前を自動生成する

```
(defmacro myor [e1 e2]
  `(let [tmp# ~e1] (if tmp# tmp# ~e2)))
(macroexpand-1 '(myor (> tmp 0) tmp))
→(clojure.core/let [tmp__43__auto__ (> tmp 0)]
   (if tmp__43__auto__ tmp__43__auto__ tmp))
```

- 必要なら捕捉が起きるようにすることも可能

2.2 Java との連携

□ Clojure の特徴の 1 つ→ JVM 上の言語→ Java オブジェクトを自在に活用

- (new パッケージ/クラス 引数…) : Java オブジェクトの生成
- (. オブジェクト メソッド 引数…) : Java オブジェクトのメソッド呼び出し
- (パッケージ/クラス/メソッド 引数…) : static メソッドの呼び出し
- 型は適当に変換して渡してくれる

```
(def win (new javax.swing.JFrame))
(. win setSize 600 400)
(. win setVisible true)
(def pane (new javax.swing.JPanel))
(. win add pane)
(. pane setLayout nil)
(def btn1 (new javax.swing.JButton "Button 1"))
(. pane add btn1)
(. btn1 setBounds 50 20 120 40)
(def lab1 (new javax.swing.JLabel "1"))
(. pane add lab1)
(. lab1 setBounds 50 80 120 40)
```

□ Java のクラスを作り出すには→ proxy

- (proxy 名前 [クラス or インタフェース…] [変数…] 関数…)

```
(def btn1action
  (proxy [java.awt.event.ActionListener] []
    (actionPerformed [evt]
      (let [i (Integer/parseInt (. lab1 getText))]
        (. lab1 setText (. (inc i) toString))))))
(. btn1 addActionListener btn1action)
```

- proxy は既存のクラスやインタフェースに従うオブジェクトを作るだけだが、自前のクラスを生成することも可能 (ここでは略)

□ Clojure のデータも実はすべて Java オブジェクト ← JVM 上で動いているのだから、基本型でないものはすべて Java のオブジェクト

2.3 マルチメソッド

□ 通常のメソッド呼び出し (Java も) →レシーバの種別 (クラス) に応じた動的分配

- より柔軟に→ (1) 複数の引数に応じたメソッド選択、(2) メソッド選択をユーザがコーディング
- (defumlti 名前 選択関数) : マルチメソッドであることの宣言
- (defmethod 名前 選択値 [引数…] 本体) : メソッドを 1 つ定義
- 選択関数 : 引数を受け取り、選択値のどれかを返す
- 例: Java の「+」と同様、「引数どちらから文字列なら連結、そうでなければ加算」するメソッド mysum

```
(defn sum-dispatch [x y]
  (cond
    (isa? (. x getClass) String) :string
    (isa? (. y getClass) String) :string
    true :other))
(defmulti mysum sum-dispatch)
(defmethod mysum :string [x y]
  (. (. x toString) concat (. y toString)))
(defmethod mysum :other [x y]
  (+ x y))
```

□ 自由度、柔軟性は非常に高まるが…これほどの柔軟性が必要な場面にはめったに遭遇しない (よくある例としては、ここで挙げたような演算の選択)

3 スクリプト言語、JavaScript

□ って、聞いたことありますか? 何を意味すると思う?

□ たとえば…

- (答 1) コマンドを並べたもの←シェルスクリプト?
- (答 2) いい加減な/適当な言語←そんなの目標にするか?

- （答 3） インタプリタ方式の言語← Lisp なんかもそうなの?
- （答 4） ささっと書いて動かせる言語←コンパクト、インタプリタ
- （答 5） くっつける (glue) 言語←組み込み型

□ スクリプト言語世代説

- 第 1 世代→シェルスクリプト、Awk →コンパクト、適当に書ける
- 第 2 世代→ Perl →それ 1 つで結構何でも書ける (汎用)、自立した言語。but キタナイ
- 第 3 世代→ Python、Ruby →かなり美しい、オブジェクト指向
- JavaScript も第 3 世代でいいと思うが…

3.1 JavaScript の由来

□ Netscape Navigator 2 →当時の「最先端の」ブラウザ

- Netscape 社はブラウザにスクリプト言語を組み込んで様々なことができるようにしようとした
- 言語は当初「LiveScript」という名前で開発→おりからの Java ブーム→ Sun にお金を払って「JavaScript」にした
- 文法はできるだけ Java そっくりにしたが、言語としてはまったく別の言語。よく JavaScript と Java を混同している人がいるので迷惑
- ブラウザ組み込みのスクリプト言語として「事実上の標準」
- MSIE もこれと互換の「JScript」を搭載

□ 標準化の必要性→ ECMA(欧州情報通信システム標準化機構) による標準化→ ECMA-262 規格として成立

- その後、ブラウザ組み込みだけでなく Flash のスクリプト言語 (AcitonScript)、MS ASP(Active Server Page) のスクリプト言語としても採用
- ECMA によるコア部分は共通だが組み込まれている追加機能はそれぞれの環境で異なる

□ 以下では (当然) ブラウザ組み込みの JavaScript を扱う (どこにでもあって簡単に試せる)

3.2 JavaScript 入門

□ HTML の説明はしませんので…

□ JavaScript コードを書ける場所…HTML ファイル中の次の 3 箇所

□ その 1: script 要素 (埋め込み、別ファイル)

```
<script type="text/javascript">
コード...
</script>

<script type="text/javascript" src="ファイル">
</script>
```

- この場合、コード中で「document.write(...)」を実行して出力したものは HTML 中のその場所 (script 要素のある場所) に埋め込まれる

```
<pre><script type="text/javascript">
document.writeln(' 乱数: ' + Math.random());
</script></pre>
```

□ その 2: HTML タグの「onmouseover="..."」(マウスが上に乗ったら～) 「onclick="..."」(クリックしたら) などの指定 (イベントハンドラ) の中

- 長いものは書きづらいのでよそに関数を定義して呼ぶことが多い

```
<p onmouseover="window.alert('   乗った ね')">ここ</p>
```

□ まとめると…

- 常に実行するもの (関数定義、ページ中への埋め込み) → script 要素
- ユーザの操作に対応するもの→イベントハンドラ

```
<html>
<head>
<title>文書のタイトル…</title>
<script type="text/javascript">
  常に実行するコード (関数定義等)
  ...
</script>
</head>
<body>
ページ内容…
<script type="text/javascript">
  document.write() を使うようなコード…
</script>
... onclick="..." ...
</html>
```

3.3 JavaScript の特徴

□ 制御構造の構文は Java にソックリ

- while、for、do-while、if、try-catch
- 文字列との連結の「+」も同じ

```
var x1 = 1, x2 = 1;
while(x1 < 50) {
  document.writeln(x1);
  var z = x1+x2; x1 = x2; x2 = z;
}
```

□ オブジェクト指向言語

```
o = new Object();
document.writeln("...");
```

☐ 弱い型→型宣言はない、変数に型がない

```
var x = "ABC", y = 1;
```

- 文字列/文字の区別はなく「'x'」でも「"x"」でも同じこと

☐ 「値」と「オブジェクト」の 2 種類がある (Java と同じ)

☐ 「値」は数値型 (1 つだけ)、論理値型 (Java と同じ)、文字列型 (文字型も兼ねる。Java では文字型は値、文字列はオブジェクト)

☐ 3 種類の「値」それぞれに対応するオブジェクト種別もある (Java と同じ)

☐ 値に対してメソッドを呼ぼうとすると自動的にオブジェクトに

```
var n = 3.141592;
document.writeln(n.toExponential(4));
```

☐ 関数 (function) がある、関数もオブジェクト

☐ 関数リテラルがある。下の 2 つは同じこと

```
function add(x, y) { return x+y; }
var sub = function(x, y) { return x-y; }
document.writeln(add(3, 5) + ', ' + sub(3, 5));
```

☐ 関数オブジェクトはクロージャ(まわりの環境を一緒に持つ)

```
function test(n) {
  return function() { return ++n; };
}
var f = test(5), g = test(10);
document.writeln(f() + ' : ' + g());
document.writeln(f() + ' : ' + g());
document.writeln(f() + ' : ' + g());
```

☐ オブジェクトはすなわち連想配列 (任意の型の値を添字として取れる配列)

☐ オブジェクトのプロパティ(フィールド) は連想配列の要素と同じ

☐ 一般に「x.y」と「x['y']」は同じものを指す

```
var x = new Object();
x['a'] = 10; x['bc'] = 11; x[100] = 12;
x.b = 13; x.de = 14;
for(var i in x) document.writeln('x['+i+'] == '+x[i]);
```

3.4 JavaScript のオブジェクト指向機能

☐ オブジェクトのプロパティとして関数を入れる→メソッド

☐ メソッドの中ではオブジェクトを「this」で参照できる

```
var o = new Object();
o.count = 100;
o.show = function(msg) {
  document.writeln(msg + this.count);
}
o.show('My number is: ');
```

☐ コンストラクタも実はただの関数だが、「new 関数名(...)」の形で呼び出すと中で this が使える

```
function Counter(n) {
  this.count = n;
  this.add = function(n) { this.count += n; }
  this.getCount = function() { return this.count; }
}
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

☐ これで Java と遜色なくオブジェクト指向…だと思う?

☐ ちゃんとあるもの

- オブジェクト、インスタンス変数 (プロパティ)、メソッド
- 同種のオブジェクト、コンストラクタ←かなり ad hoc
- メッセージ送信記法
- 動的分配←型がないので簡単

☐ 足りないと思うもの…

- クラス→クラスは必要なのか? (単なる構文単位?)
- カプセル化→確かに保護は「全然」ない→スクリプトだから? (ブラウザと組み合わせた場合、ブラウザ側での保護はアリ)
- 継承 (のようなもの) →以下で説明

3.5 プロトタイプ方式オブジェクト指向言語

☐ オブジェクトの種類 (形) を表す (定義する) やり方→おもに 2 通り

- 「こういう種類のオブジェクトはこういう形」という定義 (==クラス) を書く→クラス方式
- 必要なプロパティ、メソッドを持つオブジェクトをまず作り、あとはそれをコピーする (概念的には) →プロトタイプ方式
- 実際には全部コピーすると領域が無駄なので、プロトタイプ (親) へのポインタを持っておき、「自分が持っていないものは親が持っているものを使う」ようにする

☐ JavaScript ではどうなっているか…

- 「new 関数 (...)」によってオブジェクトが作成されるときに、
- その「関数」に prototype というプロパティが定義されていると、
- その値が作成されたオブジェクトの「親」としてセットされる

- (実際には関数を作ると prototype プロパティには「new Object()」の結果が自動的にセットされる→そこに色々設定すればよい)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

□ プロトタイプ方式の特徴…

- いつでも親にメソッドを追加したりしていじることができる (単なるオブジェクトだから)
- 場合によっては親を指しているポインタをとり替えることで、オブジェクトをまったく別物に「変身」させられる (ECMA-262 では不許可、ただし一部の実装ではこれを許している)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3); c1.add(4);
Counter.prototype.sub =
  function(n) { this.count -= n; }
c1.sub(2);
document.writeln(c1.getCount());
```

□ 継承みたいなのはどうするの?

- 親 (プロトタイプ) を探して見つからない場合はさらにその親を探しに行く。
- 例: Counter オブジェクトを継承してメソッド sub を追加するには…

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
function ExCounter(n) { this.count = n; }
ExCounter.prototype = new Counter(0);
ExCounter.prototype.sub =
  function(n) { this.count -= n; }
var c1 = new ExCounter(7);
c1.add(4); c1.sub(2);
document.writeln(c1.getCount());
```

□ プロトタイプ方式の特徴をまとめると…

- 実行系のデザインはわりとシンプルにできる
- コンパイラよりインタプリタ向き。動的
- その分、何をやっているのか分かりにくい
- (きちっと構文を作って行くとクラス方式みたいに…)

3.6 組み込み型スクリプト

□ スクリプト言語の 1 つの目的→組み込み型処理系

- 大きなアプリケーションがあったとして、それをカスタマイズすることを考える→どうしますか?

□ 沢山パラメタがあって、それを設定して調整すればいいか?

□ パラメタは静的

- 「こういう場合はこう」というのが書きにくい
- 「動き」はつけにくい

□ →パラメタを設定する代わりに「プログラム」を設定する

- →そのプログラムが動いてさまざまな動作をすればよい
- →組み込み型スクリプト
- (例は既に出て来た…「マウスが乗ったら〜をする」)

□ 組み込み型スクリプトに使われる言語…

- tcl →もともとこのような用途のために作られた。しかし言語仕様があまり美しくないので普及はいまいち (tcl/tk としてだけ有名)
- scheme →もともとは Lisp 系の汎用言語だが、コンパクトな処理系が作れるので組み込みスクリプト用としても使われるように。gwm(ウィンドウマネージャ)、gimp(画像加工ソフト)
- ECMA-262(JavaScript) →元はブラウザ用の組み込みスクリプト言語だが、他の用途にも進出 (Flash ActionScript など)

□ JavaScript の場合、オブジェクト指向ならではの利点→具体的には?

□ アプリケーション自体および、アプリケーションが扱うデータ→オブジェクトであると考えることができる

- オブジェクトを自由に操作するには、オブジェクト指向言語がよい (アタリマエ)
- これまでと違うところ→オブジェクトは既に大量に (アプリやそのデータとして) 存在している→既存のオブジェクトを操るのが主目的
- 従来のプログラムでは自前でオブジェクトを設計したり生成したりしてからそれを使っていた→大きな変化 (どんな?)

□ できあいのオブジェクトを操作する場合の考慮点

- 自分が設計したデータ構造でないのでよく分からない
- 向こう側はチェックして例外を投げる側、こちらはだましだまし使う側
- きちんと型を書くのが煩雑 (しかし型検査はあった方がよいと思うが…)

□ 具体例は次の DOM で

3.7 Document Object Model (DOM)

☐ DOM →ブラウザ内のページをオブジェクトの組み合わせで表現

- W3C による標準化→ http://www.w3.org/DOM/
- 現在のブラウザでは DOM level 2(DOM2) が中心
- DOM2 の標準は Core、Views、Events、Style、Traversal、HTML と分かれている
- 以下では例題とともに簡単に説明

☐ DOM2 では文書全体は Node オブジェクトの木構造として表される

- Node オブジェクトのプロパティ/メソッドを用いて木構造を自由に変更できる

```
<script type="text/javascript">
function rot() {
  var n = document.body.lastChild;
  document.body.removeChild(n);
  document.body.insertBefore(n,
    document.body.firstChild);
}
function dup() {
  var n = document.body.firstChild;
  var m = n.cloneNode(true);
  document.body.appendChild(m);
}
function del() {
  var n = document.body.firstChild;
  document.body.removeChild(n);
}
```

☐ HTML 要素に対応するノードのスタイル→その Node オブジェクトの style プロパティに値をつけることで色々操作できる

- 位置の変更→ CSS の位置指定機能で実現できる

```
<script type="text/javascript">
var xpos = 500, ypos = 400;
function change(dx, dy, col) {
  var n = document.getElementById('x1');
  n.style.backgroundColor = col;
  n.style.left = (xpos += dx) + 'px';
  n.style.top = (ypos += dy) + 'px';
}
</script>
<button onclick="change(-10,0,'yellow')">B1</button>
<button onclick="change(0,-10,'purple')">B2</button>
```

☐ その他できること…

- テキストノードの中の文字列を操作できる (挿入、削除、…)
- マウスイベント、キーイベントを受け止めて処理できる
- フォーム部品の内容 (値など) を操作できる (わりと昔からある機能)

☐ 結局、DOM で何ができるかというと…

- ブラウザが表示している内容→内部のデータ構造が対応
- データ構造 (オブジェクト) を任意に操作→自由に内容が変更できる
- →究極のカスタマイズ (?)

3.8 本節のまとめ

☐ スクリプト言語→ささっと書ける言語、glue 言語

☐ JavaScript →ブラウザ内蔵むけスクリプト言語が発端

- 弱い型の言語
- プロトタイプ方式のオブジェクト指向言語

☐ 組み込み型スクリプト

- ブラウザ組み込み→ブラウザの制御、DOM によるページ内容の制御

4 総称的プログラミング

☐ 配列型→「int の配列」「char の配列」など型をパラメタに持てる

- 見かたを変えれば…1 つの「もの」が複数の型に対し利用できる←「総称的」(generic) という
- 組み込み演算子なども総称的。5 / 2 == 2, 5.0 / 2.0 = 2.5。
- 総称関数 … max(int,int) → int, max(double,double) → double 等。← オーバローディングである程度はできるが、有限個数
- array[int]、array[double] …型がパラメタになっている (型パラメタ) →無限の場合に対応。

☐ そういうものを「自前で」(ユーザ定義で) 作れるようにするには?

- 以下で説明する Java Generics を使うには JDK 1.5 以降必要。C++ テンプレートは最近の C++コンパイラはだいたい実装ずみ。

4.1 コンテナクラス

☐ C++でも Java でも配列は「個数を最初に指定」→途中で大きさが伸び縮みできるものが欲しい→ライブラリで用意。

☐ Java (JKD1.4 まで) では → 例: ArrayList クラス。任意のオブジェクトを格納できる配列。

```
import java.util.*;

public class Sample31 {
  public static void main(String[] args) {
    ArrayList a = new ArrayList(); //サイズ0
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(int i = 0; i < a.size(); ++i) {
      String s = (String)a.get(i); // 取出し
      System.out.println(s);
    }
  }
}
```

- 何か疑問な点はありますか?
- この方法の弱点は何だと思う?

□ ダウンキャスト … 「元の型にキャストし戻す」こと。

- a.add("abc") --- Object 型のパラメタに String を渡す→ OK
- String s = a.get(i) --- Object 型は String 型に入れられない。×。
- String s = (Object)a.get(i) --- ダウンキャスト:親クラスの型から子クラスの型へのキャスト。このとき「元々なに型だったか」がチェックされる。String でなかったものは String にはキャストできない (ClassCastException が投げられる)。

□ 弱点 (まずいこと) のまとめ

- a. ダウンキャストは繁雑。
- b. 実行時にチェックが必要→オーバヘッドになる (Java はそれを気にしない言語だということはあるが)。
- c. オブジェクトでないと入れられない。たとえば int は Integer クラスのインスタンスにして入れなければならない (これもオーバヘッド)。
- d. 配列だから「均一なものの並び」にしたいのに、何でも入ってしまう。
- e. 間違ってヘンなものを入れてもコンパイラは怒ってくれない。
- f. そのヘンなものが取り出されて来たところで実行時にエラー。

□ 実行してみないとエラーが出ないというのはコンパイルする言語にとっての敗北。

□ 根本的な問題…本来「型パラメタ」によって複数の型 (クラス) を使用するべきところを 1 個の ArrayList クラスで済ませていること。

□ JDK 1.5 から「型パラメタ」が使えるようになった (Java Generics)。ArrayList<T>のようにパラメタは「<>」の中に指定する。T のところには任意の型 (ただしクラス) を入れてよい。

□ 動かすときは…当然、JDK 1.5 の javac を使う。逆に 1.5 の javac で 1.4 のソースを動かすときはオプション指定必要

- javac -source 1.4 Sample31.java ← JDK1.4 ソース
- javac Sample31b.java ← JDK1.5 ソース (Generics)

```
import java.util.*;

public class Sample31b {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(int i = 0; i < a.size(); ++i) {
      String s = a.get(i);
      System.out.println(s);
    }
  }
}
```

- これで先の弱点のうち a、b、e、f が解消。残る c はダメ。
- 実はこのコードを実行するとき内部的には先と同様のコードが使われている。つまり ArrayList の実装は 1 つ (均一な実装、homogeneous implementation)。だから c の制約が残っている。
- c の緩和策として、JDK 1.5 で int と Integer、double と Double 等の自動変換 (AutoBoxing/Unboxing) を追加。

```
ArrayList<Integer> a = ...
a.add(3); // JDK1.5 で○
...
int i = a.get(0); // 〃
```

□ C++では→言語設計上のポリシーとして、c は受け入れられない。また、C++では配列は「直接そのオブジェクトが埋め込まれた」ものになるので、コンテナクラスも同様でなければ受け入れられない。←フルスピードで動く C に負けないことが原則。

```
//sample31
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(int i = 0; i < a.size(); ++i)
    cout << a[i] << '\n'; // operator[]
}
```

- C++では型パラメタをソースのその位置に埋め込んで全体をコンパイルするという実装 (非均一な実装、heterogenous implementation)。だから型パラメタごとに別のコードができる。その代わり、その場展開ができる→極めて高速。

4.2 型パラメタと継承関係

- □ X ⊃ Y で継承関係を表すものとする。たとえば Object ⊃ String。
- □ パラメタつきクラス C<T>を考える。
- □ クイズ: X ⊃ Y ならば、C<X>⊃ C<Y> だといえるか?
  - たとえば、ArrayList<Object> ⊃ ArrayList<String> か? YES/NO?
- □ 次のように考えるべき
  - X ⊃ Y とは、X 型オブジェクトの代わりに Y 型オブジェクトを使っても OK であること (互換性があること)
  - 例: 「自動車」クラスの変数に「乗用車」を格納しても OK(互換性がある)
- □ ArrayList<Object> a = new ArrayList<String>; は OK?
  - a.get(0) ... String が取り出されるから OK。
  - a.set(0, Z) ... Object が入れられないと困るが String しか駄目!!!
- □ よって先の質問の答えは「NO」であるべきなんだけど…
- □ 実は配列でも同じことが起こる。
- □ しかし! Java では X ⊃ Y ならば X[] ⊃ Y[] になっている
  - 「X[] a = new Y[10];」が可能。そのあとで「a[0] = new X();」→ ArrayStoreException 発生
- □ 一般にこういうのを contravariance/covariance problem というらしい

4.3 Java Generics の制約

- □ Java Generics では実行時には型パラメタ情報がなくなっている→そのため、型検査の抜け道が起きる可能性→それを防ぐための制約
  - たとえば、「パラメタ型の配列は作れない」「パラメタつき型の配列は作れない」

```
ArrayList<String>[] lsa = new ArrayList<String>[5];
// ↑これは本来は許されない
Object[] oa = (Object[])lsa; // Java では OK
oa[0] = new ArrayList<Integer>(); // !!!
...
String s = lsa[0].get(0); // 実行時例外!!!
```

4.4 イテレータ

- □ コンテナの種類→可変長配列 (ArrayList<T>、vector<T>)、連結リスト、B 木、ハッシュ表、などなど。
  - どれでも「順番に要素を処理する」などの操作は必要→それをどれでも同等に扱えるようにしたい (後でコンテナを取り替えたりしても利用コードは変えないで済むように)。
  - 「イテレータ (iterator、反復子)」→次々に要素にアクセスしていくためのオブジェクト。
- □ Java では共通の Iterator インタフェースとして規定。JDK 1.5 では Iterator<T>のように各要素の型が型パラメタになる。

```
  bool hasNext() --- 終わりかどうか調べる
  T next() --- 次の値 (T:パラメタ型) を取り出す
  void remove() --- 現在値を削除 (今回は使わない)
```

```
import java.util.*;

public class Sample31c {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(Iterator<String>
        i = a.iterator(); i.hasNext(); ) {
      String s = i.next();
      System.out.println(s);
    }
  }
}
```

- □ さらに! Iterable<E>を実装するオブジェクト X については

```
  for(E v: X) { ...
```

という for ループ (for-in ループ) を書くと以下と同等に動作

```
for(Iterator<E> i = X.iterator(); i.hasNext(); ) {
  E v = i.next(); ...
```

- □ たとえば上の例も ArrayList<E> が Iterable<E> を実装しているので次のように書ける。

```
import java.util.*;

public class Sample31d {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(String s: a) System.out.println(s);
  }
}
```

- □ C++では共通のインタフェースはない (各コンテナごとに違う) が形としては次のもの。

```
  *p --- 現在の要素を参照
  ++p、p++ --- 次の要素に進める
  p == q --- 同じ要素を指すかどうか判定
```

```
//sample31b
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(vector<int>::iterator
      i = a.begin(); i != a.end(); )
    cout << *i++ << '\n';
}
```

- 「vectior<int>::iterator」→ vector<int>クラスにおいて typedef で定義されている iterator という型名。
- 普通の「配列要素を指すポインタ」もイテレータとして使える
- コンテナにもよるが単なるポインタではないイテレータが大半

4.5 型パラメタつきクラスを作る

□ 例: 「最後の 2 つの値を記憶するバッファ」(前回やったもの) ただし、記憶される値の型は「任意」

□ C++版 (クラス定義の中に直接コード本体も書いている。こうするとインライン展開が可能に。または従来通り分けて書いてもいいが、その場合は inline と指定。)

```
//sample32
#include <iostream>
using namespace std;

template<typename T> class last2 {
  T v1, v2;
public:
  last2(T x, T y) { v1 = x; v2 = y; }
  void put(T v) {
    v2 = v1; v1 = v;
  }
  T get() {
    T x = v1; v1 = v2; return x;
  }
};

int main() {
  last2<int> a(0, 0);
  a.put(1); a.put(2); a.put(3);
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  last2<char*> b("", "");
  b.put("ab"); b.put("cd"), b.put("ef");
  cout << b.get() << '\n';
  cout << b.get() << '\n';
  cout << b.get() << '\n';
}
```

□ Java 版。

```
import java.util.*;
public class Sample32 {
  public static void main(String[] args) {
    Last2<Integer> a = new Last2<Integer>(0, 0);
    a.put(1); a.put(2); a.put(3);
    System.out.println(a.get());
    System.out.println(a.get());
    System.out.println(a.get());
    Last2<String> b = new Last2<String>("", "");
    b.put("ab"); b.put("cd"); b.put("ef");
    System.out.println(b.get());
    System.out.println(b.get());
    System.out.println(b.get());
  }
}

class Last2<T> {
  T v1, v2;
  public Last2(T x, T y) {
    v1 = x; v2 = y;
  }
  public void put(T v) {
    v2 = v1; v1 = v;
  }
  public T get() {
    T x = v1; v1 = v2; return x;
  }
}
```

4.6 Standard Template Library (STL)

□ C++の標準ライブラリの土台となった、テンプレートを駆使したライブラリ群→現在では C++標準ライブラリのコンテナクラス部分のことをそう呼ぶ。重要なアイデア:

- テンプレートを使った効率のよいコンテナクラス群
- 要素アクセスはコンテナクラスのメソッドではなくイテレータ経由 (その方が高速化しやすい)
- イテレータに対して働くアルゴリズム群。find (線形探索)、sort (整列)、count(計数)、などなど。
- アルゴリズム群はテンプレート関数 (型パラメタつき関数) →高速。

□ 例: 普通の swap(C 版)

```
void swap(int *x, int *y) {
  int z = *x; *x = *y; *y = z;
}
int a[100]; ... swap(&a[i], &a[j]) ...
```

- 関数呼び出しのオーバヘッド、ポインタ経由のアクセス→遅い。しかも型ごとに用意する必要。

□ 関数テンプレート版の swap(C++版)

```
template<typename T> inline void swap(T& x, T& y) {
  T z = x; x = y; y = z;
}
float a[100]; ... swap(a[i], a[j]) ...
```

- 関数テンプレートの型パラメタは推定される (a[i] 等が float だから T は float) →繁雑な指定が不要

- 「float z = a[i]; a[i] = a[j]; a[j] = z;」がこの場所に埋められているのと同様→高速

□ アルゴリズム関数を利用すれば制御構造をこちら側で書かなくてもよくなる。たとえば for_each を使うとループが不要になる。

```
//sample33
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

template<typename T> void f(T x) {
  cout << x << '\n';
}

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i);
  for_each(a.begin(), a.end(), &f<int>);
}
```

- これで for_each の中で「f(値)」が繰り返し呼ばれる→「ベクタの各要素を出力」ができる。ループは不要。
- しかし「合計を取る」だとどうするか?

□ 関数オブジェクト… operator() を持つようなオブジェクト。先の「f(値)」というのは関数でなく operator() の呼び出しでもよいので。

```
//sample33b
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template<typename T> class sum {
  T res;
public:
  sum(T init) { res = init; }
  void operator()(T x) { res += x; cout<<res<<"\n"; }
  T result() const { return res; }
};

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i);
  sum<int> s(0);
  s = for_each(a.begin(), a.end(), s);
  cout << s.result() << '\n';
}
```

- ループ周回を通じて保存されて欲しいものは関数オブジェクトのインスタンス変数にすればよい。→ ループ構造と処理の分離

4.7 テンプレートの特殊化

□ テンプレートで一般向けの実装のほかに「特殊ケース」を用意したいことがある。

- たとえば bool(0/1) のバッファだったら 1 ビットずつ入れれば 32 個の値が 1 語で入るとか…
- コンテナクラスの場合、「ポインタ用」はそれなりのポインタ用のものを使う方がよさそう… (「ポインタ用」のクラスを 1 つ用意して残りはすべてそれを呼ぶ→コードを何回も展開しなくて済む→コード量の爆発を防ぐ上で重要なテクニック)

□ そのような「特殊ケース」を別途用意できる→特殊化

```
template <typename T> class last32 {
  //一般
}
template <typename T> class last32<T*> {
  // ポインタ用
}
template <> class last32<bool> {
  // bool 用
}
```

4.8 型パラメタに対する制約

□ 例: maxbuf<T>というコンテナを作る。

- 次々に put(T x) で値を入れていく。
- T get() でその時点での最大値を取得できる。

```
//sample34
#include <iostream>

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  double a2[] = { 3.0, -2.7, 6e2, 4e1, 0.5 };
  maxbuf<double> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

□ 型 T には演算子「<」が必要。これを満たさないと…

```
//sample34b
#include <iostream>

template<typename T> class maxbuf {
  T max;
public:
```

```
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

class point {
  double x, y;
public:
  point(double a, double b)
    : x(a), y(b) { }
};

int main() {
  point a3[] = { point(0,0), point(1,1),
    point(2,2), point(3,3), point(4,4) };
  maxbuf<point> m3(a3[0]);
  for(int i = 1; i < 5; ++i) m3.put(a3[i]);
}
```

□ 上記のをコンパイルすると…

```
In method ‘void maxbuf<point>::put(point)’:
22:   instantiated from here
7: no match for ‘point & < point &’
```

□ 確かに「<」がない、と言われる… 次のを入れればよくなるが。

```
bool operator<(point &p) {
  return x*x+y*y < p.x*p.x+p.y*p.y;
}
```

- しかし「コンパイルしてみたら無かった」では実装の中に立ち入っていてエラーメッセージとしてあまり良くないのでは…
- 実際、極めて分かりづらいエラーメッセージに遭遇することがよくある
- 本来だったら「この型パラメタにはこういう型しかあてはめてはいけませんよ」という指定ができた方がいいのでは? → 賛否両論。否定派: 指定が繁雑になり自由度が減る。ともあれ、Java Genericsではこちらの立場。

□ 上記の Java Generics 版。整数は Integer クラスを使う必要。

```
import java.util.*;

public class Sample33 {
  public static void main(String[] args) {
    int[] a1 = { 5, 3, 8, 2, 4 };
    Maxbuf<Integer> m1 = new Maxbuf<Integer>(a1[0]);
    for(int i = 0; i < 5; ++i) m1.put(a1[i]);
    System.out.println(m1.get());
    String[] a2 = { "this", "by", "foo", "z", "x" };
    Maxbuf<String> m2 = new Maxbuf<String>(a2[0]);
    for(int i = 0; i < 5; ++i) m2.put(a2[i]);
    System.out.println(m2.get());
  }
}

class Maxbuf<T extends Comparable<T>> {
  T max;
  public Maxbuf(T x) { max = x; }
  public void put(T x) {
    if(max.compareTo(x) < 0) max = x;
  }
  public T get() { return max; }
}
```

□ Comparable<T>というのは x.compareTo(T y) というメソッドを持ち、このメソッドが x と y の大小関係に応じて正/負/零を返す。

□ この方が確かにあらかじめチェックはできるが、窮屈かも…

□ あらかじめ Comparable<T>を implements しているクラスにしか使えない→自分がソースを持っていればいいが、そうでないと…

- 実は C++版にも同様の問題がある。たとえば char* の maxbuf を取ろうとすると「アドレスが最大の」ものが取れてしまう…求めていることと違う。

□ C++の場合は、単独関数テンプレートと特殊化を使えば OK。

```
//sample34c
#include <iostream>
#include <cstring>
using namespace std;

template<typename T> bool lt(T x, T y) {
  return x < y;
}
template<> bool lt(char* x, char* y) {
  return strcmp(x, y) < 0;
}

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(lt(max, x)) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  char* a2[] = { "this", "by", "foo", "z", "x" };
  maxbuf<char*> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

□ 特殊化の応用

- 一般の T →「<」が使われる
- char* → strcmp が使われる
- その他特殊ケースはいくらでも対応可能 … C++の実用性に 1 日の長があるかも?

4.9 テンプレートによる mixin クラス

□ 前回やったように、mixin とは「他のクラスに混ぜるためのクラス」

- それ自体単独でインスタンスを生成しないことから「抽象サブクラス」と呼ぶことも
- 前回の話では flavors などの多重継承を使って混ぜていた
- しかし C++のような静的検査な言語では多重継承ではうまく行かない

□ たとえば、次のようなクラスを考える。

```
class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};
```

□ 「update の回数を数える」

```
class count {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; update(v); }
  int getCount() { return num; }
};
```

□ 「更新履歴を記録する」

```
class history {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = getValue(); update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
};
```

□ これを次のようにして使いたい。

```
class MyClass :
  public balance, count, history {
};
```

□ flavors ではこれでできる (どっちの update が呼ばれるかの規則があって OK)。しかし C++ではそもそも getValue() や update() の宣言がないと型検査できなくてアウト。

□ ではどうするか…

- count や history で親クラスを指定すればもちろん動くようになる。
- しかしさまざまなものにこれらの機能を組み込もうと思って作ったのに、決め打ちで親クラスを書いてしまうのでは 1 つにしか使えない (または繰り返しコピーになる) のでよろしくな。
- ではどうすればいい?

□ 答: その親クラスをテンプレートパラメタで指定する!!

```
//sample35
#include <iostream>

class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};

template<class T> class count : public T {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; T::update(v); }
  int getCount() { return num; }
};

template<class T> class history : public T {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = T::getValue(); T::update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
};

int main() {
  int a[] = { 5, 9, 2, 7, 6 };
  history< count< balance > > c1;
  count < history< maxbuf > > c2;
  for(int i = 0; i < 5; ++i) {
```

```
    c1.update(a[i]); c2.update(a[i]);
  }
  cout << "c1: " << c1.getCount(); c1.showHistory();
  cout << "c2: " << c2.getCount(); c2.showHistory();
}
```

□ mixin クラスを使うことで、1 つのクラスに盛り込まれている複数の側面を分離して記述し、使う時に組み立てることができる。

□ しかし、実際には 1 つの「側面」が複数のクラスにまたがって存在していることが普通→このような「一群のクラス」を 1 つの「層」として、それを積み重ねてシステムが構成されるというイメージ。

□ そのようなものを実現するには→ C++でもクラスの中にクラスが書けることを利用し、外側クラスをパラメタつきとする。

```
template<class T> class ThisLayer : public T {
public:
  class First : public T::First { ... };
  class Second : public T::Second { ... };
  ...
};
```

□ こうすれば、一群のクラスで Fist は First どうし、Second は Second どうし、…で縦に合成が起きる。

- ThisLayer の中にはこの層が実現する機能のためのデータやコードをまとめて入れることができる。
- このような構成を「mixin layers」と呼ぶ。
- このような「構成法」の代替案→言語自体の拡張 (AOP 言語) →すぐ後で

4.10 テンプレートメタプログラミング (簡単に)

□ テンプレートの展開→関数の評価のようなもの→計算に利用できる

```
//sample36
#include <iostream>

template<int n> class Fact {
public:
  enum { RET = Fact<n-1>::RET * n };
};
template<> class Fact<0> {
public:
  enum { RET = 1 };
};

int main() {
  std::cout << Fact<5>::RET << '\n';
}
```

- これを動かすと当然「120」が表示されるが、それは実行時に計算しているのではなく、コンパイル時に「5*4*3*2*1*1」という式が生成され、それが計算されている→高速
- もっと複雑なデータを処理したり、複雑な手順を持つ関数を生成したりすることもできる → テンプレートメタプログラミング
- 使いこなすのはとっても難しいと言われている (あまりやりたくない)

4.11 本節のまとめ

□ 総称的プログラミング (型パラメタ、パラメタつき型) → C++ テンプレート、Java Genrics で実現 (内容はかなり違っている)。主な用途: コンテナクラス

□ STL(C++) → イテレータによるアクセス、汎用アルゴリズム

□ 型パラメタの制約 → 賛否分かれるところ。Java ではインタフェース/サブクラスによる制約指定可能 (だが不自由でもある) → C++に対する制約機構 (concepts) は設計途上 → 後で少し紹介

□ mixin クラス (C++) → 親クラスを拡張するための抽象サブクラス

□ 全体として C++テンプレートは強力、しかし難しい。より詳しく知りたい人は:

- アンドレイ·アレキサンドレスク著, 村上訳, Modern C++ Design, ピアソン, 2001. (むずかしい)
- ε π ι σ τ η μ η, 高橋晶, C++テンプレートテクニック, ソフトバンク, 2009. (少しやさしい。筆頭著者の読みは「えぴすてーめー」)

5 AOP — Aspect-Oriented Programming

□ オブジェクト指向 (OO、Object-Orientation) は確かに問題を「人間に分かりやすく構造化」する手段を提供してくれた (と思う)。

□ しかし、どのようにうまく「構造化」しても、その構造化から「洩れている」側面が必ず存在する→横断的側面 (Croscutting Concern、 XC)

□ OO が定着した現在では、この XC をうまく取り扱う方法が言語研究者の関心時の 1 つとなっている→ Aspect-Oriented Programming(AOP)

- AOP によって OO だけでは扱えない部分をうまくカバーした言語が作れる (はずだし、実際そういう言語を提案している) → AOP 派の主張

□ 以下で紹介するのは AOP 派の主張であり、それに「説得」される必要は必ずしもありません。

- ある言語 X が 99%のニーズをうまく満たしているなら残り 1%をカバーするため言語を 2 倍複雑にすることは引き合わない。
- しかしこれが「80%満たしている」のと「1.2 倍複雑」だったら引き合うのかも知れない。
- なお、いつまでたっても「100%カバー」はあり得ない。
- 実は OO が従来の手続き型に対してやったことも似たようなもの→ということは、OO による進歩のあとで「残っていること」がどれくらい大きいのかは疑問な気もする。
- また現状の AOP 言語による「カバーのしかた」がいいのかどうかも分かっていない (研究段階)

5.1 なぜ OO だけでは不足なのか?

□ OO がもたらしたもの→プログラムの扱う世界を「もの」の集まりとして位置付け、「もの」ごとに「性質」(インスタンス変数など) と「動作」(メソッドなど) を記述するようにした

- なおかつ「もの (の種類)」どうしの関係についても取り扱えるようにした (インタフェース/動的分配による総称性、継承による階層化、実装の引き継ぎ)。

□ じゃあ「それで十分じゃん」と思いますか? 何か「こういうことは欠けているのではないか」というのはありますか?

□ 例: 図形を扱うアプリケーションで「図形が変化したことを記録する」

- いろいろな図形がいろいろな風に変化する
- これまでの方法→変化させるメソッドに記録のコードを挿入
- ×本体の動作と記録の動作がごちゃまぜになる
- ×記録コードそのものは類似したものをそこら中にばらまくことに

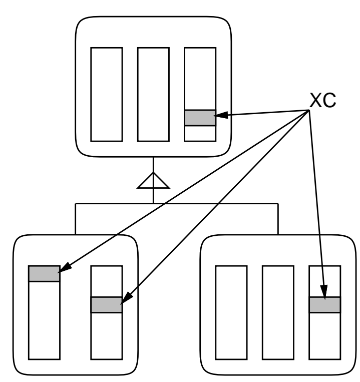
□ この他にもさまざまな「あちこちに同じもの挿入」の例がある

- データの永続化
- 表示の更新
- トレース、アンドゥ
- などなど…

5.2 横断的側面

□ 結局、これらの問題は「オブジェクトの階層構造とは直交した横断的な機能や動作」をプログラムに組み込もうとするところに起因する。

- このようなものを「横断的側面 (Croscutting Concern)」と呼ぶ



□ 横断的側面はオブジェクト指向による構造化と直交しているので、無理矢理オブジェクト指向で扱うよりも、新しい言語機構を導入して扱う方がよい。

- …というのが AOP の主張ですが、OK と思う? 疑問?

5.3 Aspect Oriented Programming (AOP)

□ AOP の基本的な考え方

- プログラムはいくつもの「側面」を持っている→それらをそれぞれ「分けて」取り扱うのがよい (separation of concerns)
- そのような「側面」のうちには、さまざまな部分にバラバラに現れて来るものがある→横断的側面 (croscutting concern)
- それらを分けて扱うため、メソッドやクラス階層とは「別の」メカニズム (=アスペクト) を導入して、これを通じて横断的側面を記述可能にする→Aspect Oriented Programming
- AOP 言語→そのような機構を採り入れたプログラミング言語
- 用語: join point → 複数の側面を「接合させる」点のこと

□ 代表的な AOP 言語の流派 (?)

- Adaptive Methods --- メソッドを状況に応じて変化 (適応) させることでさまざまな場合に使えるように → Demeter/Java (DJ) ライブラリ
- MultiDimensional Separation Of Concerns (MSDOC) アプローチ--- 複数の「側面」をそれぞれ記述し合成して 1 つのシステムに →Hyper/J 言語
- Composition Filters --- 複数の側面を組み合わせるフィルタを用意してこれらを集めてシステムを組み立てる

- Aspect/J --- Aspectと呼ばれる記述を別に用意して既存のクラス構造を修正/拡張 ← AOP言語の「代表」の地位を占めつつある

□ 以下ではAspect/Jを例にどんな感じか見ていただく

5.4 Aspect/J

□ Gregor Kiczalesらが作成したJavaを土台としたAOP言語

- http://www.eclipse.org/aspectj/ でフリー配布

□ 特徴…

- Java の上位互換 (ふつーの Java プログラムは AspectJのプログラムでもある)
- 既存のJavaプログラムにアスペクトを追加して行ける(既存のクラスを後づけで変化)

□ 以下では最低限感じが分かる程度の例題のみ説明

- 網羅的な解説とかは上記サイトで

5.5 Pointcut

□ Pointcut --- プログラム実行時の「ここ」という点。

- call(返値の型 クラス名.メソッド名(引数の型…)) --- これこれのクラスのこれこれのメソッドが呼ばれるところ
- call(クラス名.new(引数の型…)) --- これこれのクラスのコンストラクタが呼ばれるところ
- initialization(クラス名.new(引数の型…)) --- 同様だが、初期設定が行われるところ
- cflow(call(...)) --- 指定した呼び出し以下で行われる実行
- cflowbelow(call(...)) --- 同様だが、その呼び出し自体は含まない

□ 上記の基本的なpointcutを組み合わせて名前をつけることができる

- pointcut 名前(...) : 定義すみpointcutの組合せ ;

5.6 Advice

□ Advice --- 「既存のプログラムのここをこういう風に直せ」という指定。

- before() : *PC* { コード } --- pointcut *PC* 実行直前にこれこれを行え。
- after() : *PC* { コード } --- pointcut *PC* 実行直後にこれこれを行え。
- after() returning : *PC* { コード } --- 同上ただし正常リターン時のみ。
- after() throwing : *PC* { コード } --- 同上ただし例外で戻った場合のみ。
- 返値の型 around() : *PC* { コード } --- pointcut *PC* の代わりに実行するコードを指定。コード中で proceed(...) というものを書いておくとそこで本来の実行コードが呼ばれる。

5.7 下敷きとなるJavaの例題

□ 前回の例題の類似品

- アイコンをマウスでドラグして移動できるだけ

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

// 窓を開くアプリケーション
public class Sample41 extends Frame {
  DrawSet set = new DrawSet(); // アイコンの集合
  public Sample41() {
    set.addObj(new DraggableIcon("A", 100, 100));
    set.addObj(new DraggableIcon("B", 120, 120));
    set.addObj(new DraggableIcon("C", 140, 140));
    setSize(400, 400);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        set.mousePressed(evt.getX(), evt.getY());
        repaint(); // マウスダウン時の処理
      }
      public void mouseReleased(MouseEvent evt) {
        set.mouseReleased(evt.getX(), evt.getY());
        repaint(); // マウスアップ時の処理
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        set.mouseDragged(evt.getX(), evt.getY());
        repaint(); // ドラグ時の処理
      }
    });
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) {
        System.exit(0); // 窓が閉じる→終了
      }
    });
  }
  public void paint(Graphics g) { set.draw(g); }
  public static void main(String[] a) {
    (new Sample41()).setVisible(true);
  } // ここで窓を開く
}

// 画面上に現れるものを表すインタフェース
interface DrawObj {
  public void draw(Graphics g);
  public boolean hit(double x, double y);
  public void mouseDown();
```

```
  public void mouseUp();
  public void mouseDrag(double x, double y);
}

// DrawObj の集まりを保持しマウスで選択可能に
class DrawSet {
  DrawObj hit = null; // 選択中のもの
  DrawObj[] a = new DrawObj[200];
  int count = 0;
  public void addObj(DrawObj o) {
    if(count < a.length) a[count++] = o;
  }
  public void draw(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
  public void mousePressed(double x, double y) {
    hit = null;
    for(int i = count-1; i >= 0; --i) {
      if(a[i].hit(x, y)) { hit = a[i]; break; }
    } // マウスの座標に当たった最初のものを選択
    if(hit != null) hit.mouseDown();
  }
  public void mouseReleased(double x, double y) {
    if(hit != null) hit.mouseUp();
    hit = null;
  }
  public void mouseDragged(double x, double y) {
    if(hit != null) hit.mouseDrag(x, y);
  }
}

// ドラグできるアイコン
class DraggableIcon implements DrawObj {
  Font fn = new Font("Serif", Font.BOLD, 24);
  String label;
  double xpos, ypos, rad = 25.0;
  Color col = Color.yellow;
  public DraggableIcon(String s, double x, double y) {
    label = s; xpos = x; ypos = y;
  }
  public void draw(Graphics g) {
    int r2 = (int)(rad*2);
    g.setColor(col);
    g.fillOval((int)(xpos-rad),(int)(ypos-rad),r2,r2);
    g.setColor(Color.black); g.setFont(fn);
    g.drawString(label, (int)(xpos-12), (int)(ypos+8));
  }
  public boolean hit(double x, double y) {
    return (xpos-x)*(xpos-x)+(ypos-y)*(ypos-y)<rad*rad;
  }
  public void moveTo(double x,double y){xpos=x;ypos=y;}
  public void setColor(Color c) { col = c; }
  public double getX() { return xpos; }
  public double getY() { return ypos; }
  public void mouseDrag(double x,double y){moveTo(x,y);}
  public void mouseDown() { setColor(Color.pink); }
  public void mouseUp() { setColor(Color.yellow); }
}
```

5.8 例: 画面の変化を記録するとしたら…

□ 案 1: DraggableIcon クラスのメソッドに記録用のコードを挿入してまわる

- もっといろいろなものが画面に現れるようになると大変そう
- 「記録する」のはどこか 1 箇所で何とかしたい

□ 案 2: アスペクトとして別途分離して記述

□ AspectJ で記述した変化表示アスペクト

```
import java.awt.*;

aspect DisplayChange {
  int count = 0;

  pointcut displayChange() :
    call(void DraggableIcon.moveTo(double, double)) ||
    call(void DraggableIcon.setColor(Color));

  after() : displayChange() {
    System.out.println("changed: " + (++count));
  }
}
```

□ 読み方

- DraggableIcon のメソッド moveTo() または setColor() が呼ばれたところ→「表示変化」という pointcut にする。
- 「表示変化」という pointcut の直後に「いくつ目の変更」というメッセージ表示を挿入する。

□ 動かし方

```
  % ajc Sample41.java DisplayChange.java
  % export CLASSPATH=.:/compat/linux/user/local/
Java/aspectj1.5/lib/aspectjrt.jar
  % java Sample41
```

- ajc --- AspectJ Compiler
- 実行はこれまでと同じ ← 生成されるコードは普通の Java バイトコードだからそのまま客先に配布できる
- ただし! 1 つだけ追加のライブラリファイル (.jar ファイル) がある→それを CLASSPATH に追加 (上記 AspectJ 配布キットにドキュメント)

□ 驚くべきこと

- もとの Sample41.java には何ら手を加えていない
- 「ここをこう直したい」ということはアスペクト側だけで指定
- これによって複数のアスペクトを混合しても対処可能に
- またアスペクトの書き方を工夫すれば再利用可能に

5.9 例: 画面の変化を記録し後戻り可能にする

□ たとえば上のプログラムをさらに次のようにする

- とりあえず「動き」だけについて記録

- 記録の各レコードを表すクラスを用意する
- 位置変化ごとにそのインスタンスを配列に追加する
- 後戻りボタンを画面に貼り付けておく
- ボタンが押されたら配列から最後の位置記録を取り出してそこへ戻す

□ こういうのを実現するとしたらどういう機能がさらに必要?

□ さっきのと何が違うか? → 挿入したコードの中からアイコン等のオブジェクトを参照しなければできない

- 実は pointcut の冒頭で「変数宣言」できる。
- さらにその宣言した変数が「カレントオブジェクトである」「メソッド呼び出し対象オブジェクトである」等の指定を書くことであてはめが行われ、変数に値が取り出せる。
- その pointcut を利用する側でこのパラメタを参照できる。

□ さっきより少し長いが全部まとめて示すと…

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

aspect RecordChange {
  History[] record = new History[5000];
  int count = 0;
  Button b1 = new Button("Back");

  pointcut createApp(Sample41 app) :
    this(app) &&
    initialization(Sample41.new(..));
  pointcut positionChange(DraggableIcon i) :
    target(i) &&
    call(void DraggableIcon.moveTo(double, double))&&
    !cflowbelow(call(void History.apply()));
  pointcut colorChange(DraggableIcon i) :
    target(i) &&
    call(void DraggableIcon.setColor(Color));

  after(DraggableIcon i) : positionChange(i) {
    System.out.println("moved: " + (++count) + " : " +
      i.getX() + "," + i.getY());
  }
  before(DraggableIcon i) : positionChange(i) {
    record[count] = new History(i, i.getX(), i.getY());
  }
  after(Sample41 a) : createApp(a) {
    final Sample41 app = a;
    app.setLayout(null); app.add(b1);
    b1.setBounds(20, 20, 80, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        if(count > 0) {
          record[--count].apply(); app.repaint();
        }
      }
    });
  }
  static class History {
    DraggableIcon icon; double xpos, ypos;
    public History(DraggableIcon i, double x, double y) {
     icon = i; xpos = x; ypos = y;
    }
    public void apply() { icon.moveTo(xpos, ypos); }
  }
}
```

□ まず最後に記録用の入れ子クラスがつけてある。これのインスタンスにアイコンとその位置を記録する

□ 冒頭の配列 record にインスタンスを順次入れていく

□ pointcut は 3 つに増えている。いずれもパラメタを取るようにしている

- Sample41 のインスタンスを作るところ
- アイコンが動くところ、ただし History からの呼び出しは除く
- アイコンの色が変わるところ

□ 位置変化の「後」の動作はさっきと同じ

□ 位置変化の「前」に配列にに位置を記録する

□ Sample41 が初期化された「後」にボタンを貼り付ける

- ボタンの動作の中で後戻りを実現
- ボタンの動作中から app を参照するには変数 app は final である必要がある

5.10 AspectJ と AOP のまとめ

□ AspectJ にはほかにもいろいろ機能があるけど…

- 要するに「外から」既存のクラスの動作前後に介入できることがポイント
- SOC (Separation of Concern) とはこういう感じ
- 他の AOP 言語もそれぞれ違った側面からこのような「分離」に挑戦している

□ それで…

- ○このような言語はすばらしい! と思いましたか? または
- ×こんなに複雑にしてもしょうがないじゃん! と思いましたか?

6 演習問題

□ (9) JavaScript を用いて、自分の Web ページに動きや機能をつけるような「ライブラリ」(つまり自分だけにしか使えないのではなく、他人にあげたら使ってもらえるような形のもの) を作ってみよ。その経験に基づいて、JavaScript によるプログラミングと他の言語 (Java 等) によるプログラミングの違いについて考察せよ。「動

きや機能」の例としてはたとえば次のものが考えられる(ただしこれらに限定せず自由にアイデアを出してよい)。

- HTML 要素を動かしたり現われさせたり消したりさせる
- ポップアップ/プルダウンメニュー機能 (サブメニューも出せるとなおよい)
- ページ内でクリックすることでクリックした個所を強調するなどページ内容が加工できるようにする

□ (10) C++テンプレートまたは Java Generics (JDK 1.5 以降必要) を用いて、自分なりの「汎用的な役に立つクラス」を開発してみよ (イメージとしては max バッファや last2 バッファのような感じでよい)。実際に複数の型をあてはめて動作させること。また、それを使った場合と使わずに「普通に」書いた場合の性能比較も行うこと。(さらに、C++テンプレートと Java Generics の両方作って両者の比較もするとなおよい。)

□ (11) AspectJ の処理系を入手し (入手先は本文中に記載)、本文の例題を動かしてみよ。納得したら次のような拡張を行ってみよ。

- アイコンの色変化も記録/再現できるようにしてみよ。
- もっと他の図形や他の動きも入れたプログラムにして、それを記録/再生できるようにしてみよ。
- その他、AspectJ の機能を活用した例題を新たに考えて作成してみよ。

□ (12) C++テンプレートメタプログラミングで階乗以外の何らかの値の計算をさせてみよ (組合せの数とかフィボナッチ数とか?)。その上で、普通のコードで計算した場合と性能比較を行ってみよ。もしコンパイルが異常に遅いならその原因も究明せよ。

7　さいごに

□ この講座では、さまざまなプログラミング言語に現れる概念を、その設計思想、用途、実装、特徴、問題点などの観点から整理して見てきたつもりです。

□ 構成としては、1 回目→プログラミング言語一般〜抽象データ型、2 回目→オブジェクト指向の導入、3 回目→オブジェクト指向の主要部分＋ドリトル言語、4 回目→オブジェクト指向言語のさまざまな機能＋ Clojure、5 回目→組み込みスクリプト＋総称的プログラミング＋ AOP、のように構成しました。

□ プログラミング言語にはさまざまな機能や側面があり、それらを取捨選択して 1 つの言語としてまとめるのはなかなか簡単ではないことがお分かり頂けたかと思います。