電子工業振興会先端ソフトウェア技術委員会 1999.7.9
— Java 言語に見るプログラミング言語の諸側面 —

久野　靖*

1999.7.9

1　はじめに

□ 二木先生からのリクエスト→「Java 言語のプログラミング言語的な側面」

- Java 言語のプログラミング的でない側面って?
- たぶん、政治力学的 (ビル・ゲイツ的?) 側面はいらないという意味?→○
- 言語を離れた各種応用もいらないという意味?→○

□ 飯泉さんからお伝え頂いた「皆様の要望」

- 言語の歴史 (e.g. CLU、Ada、…) から見た Java の言語としての美学→○
- Java プログラマが Toy Program の世界から脱却するためには→△
- 言語実装上の視点から見た Java の派生言語の目的/課題/今後→○

□ 飯泉さんのお言葉

- 久野の得意とする話題について→◎

1.1　久野の経歴

□ 東京工業大学理学部情報科学科木村研究室

□ 学部のころ…

- 木村研と井上研 (隣の研究室) はプログラミング言語研究のメッカ
- 文字列処理言語
- Pascal
- Lisp
- Ratfor (←ソフトウェア作法)
- 卒業研究→ Kernighan/Plauger 本を見ただけで (!)C のコンパイラを作る

□ 隣の井上研 (←大槻さん) では…

- パーサジェネレータ
- マクロプロセッサ← (佐々先生:助手)
- Algol68
- Ada

□ 「世の中には様々なプログラミング言語がある」

□ 大学院時代 (助手:米澤先生)

- 米澤先生が MIT から CLU の仕様を持ち帰っていた
- 佐渡さん (現群馬大) が CLU のコンパイラを作った
- みんな CLU で色々なシステムを書いていた
- 修論:かな漢字変換システム (←「ソフトウェア作法」本作成)
- 博論:CLU のような言語のための設計技法 (←「オブジェクト指向設計」?)

□ 「抽象データ型の概念は極めて強力かつ重要」

□ 「ソフトウェア工学は自分には向いていない」

□ 東工大助手→筑波大 (現在)

□ オブジェクト指向言語の研究→ Misty

- 強い型のオブジェクト指向言語
- 型パラメタを持つオブジェクト指向言語
- 界面と実装の分離→界面は継承し、実装は委譲による

□ 並列オブジェクト指向言語の研究→ p6/p6i

- 並行制御/同期機構→抽象状態同期
- 抽象状態同期による継承異常の解決

□ 「プログラミング言語研究には無限に課題が存在する」

□ 「新しいプログラミング言語を普及させるのはとても難しい」

- → Javaっていったい…

*筑波大学大学院経営システム科学専攻

1.2 Java 言語の由来

□ Sun Microsystems 社→ Unix WS のイメージだがソフト屋も一流

□ 90 年代前半に、セットトップボックスやディジタル家電用のシステムを開発するプロジェクトを開始

- オブジェクト指向は必須→最初は C++を考えていた→ C++では安全なシステムにならない
- C++をもとに危険な/複雑な機能を削除→ Java
- 「アプレットのための言語」として爆発的に普及→好運な側面
- 言語そのものはまっとうな汎用のオブジェクト指向言語→「まっとう」とは?

1.3 以下の進め方…

□ 系統的/網羅的に話をしようとすると難しい

- 時間も限られているし、「つまらない」「当り前の」話も含まれる
- もっと「面白い」ことを重視

□ →行きあたりばったりに/思い付くままにテーマを挙げて検討

- さらに、あくまで久野の「独断と偏見」(きっと反論多数?)

□ →面白いと思ったところで遠慮無く止めて議論してください (お願い)

2 Java は手続き型言語

□ 手続き型 (命令型) 言語とは…

- 文単位の順次実行、変数、代入
- 本質的に「副作用」ベース (代入=副作用、だから)
- 副作用はあっていいのか? 有害ではないのか?

□ その他「普通の手続き型言語」らしい側面について

- スコープ
- 制御構造

□ 例: Hello World

```
public class Sample1 {
  public static void main(String args[]) {
    System.out.println("Hello, World.");
  }
}
```

□ 例: 数を読み込み 1 たす

```
import java.io.*;

public class Sample2 {
  public static void main(String args[])
      throws Exception {
    BufferedReader rd = new BufferedReader(
      new InputStreamReader(System.in));
    while(true) {
      System.out.print("> "); System.out.flush();
      String s = rd.readLine();
      if(s == null) break;
      int i = (new Integer(s)).intValue();
      System.out.println("" + i + " + 1 = " + (i+1));
    }
  }
}
```

2.1 手続き型 .vs. 関数型とオブジェクト指向

□ 関数型オブジェクト指向言語?

□ 関数型言語→副作用を持たない関数の組合せによるプログラミング

□ 「もの」の同一性を考えない。あくまで「値」

□ 抽象データ型は副作用がなくても作れるが…

□ その抽象データ型を「オブジェクト」として扱うのはやりにくい

- ある時点の人とその人が何かを行った後の人が「別人」になってしまう
- ないし「人」のインスタンスは存在せず、「人」の「痕跡」「スナップショット」の集まりを扱うことになってしまう

□ やっぱり「オブジェクト指向言語」は「副作用」が必要では?

- ただし、immutable なオブジェクトも当然あってよい

□ Mutable なクラスの例

```
import java.io.*;

public class Sample3 {
  public static void main(String args[])
      throws Exception {
    Human h = new Human(20, 55);
    h.addAge(5);
    h.addWeight(10);
    System.out.println("result: "+h);
  }
  static class Human {
    int age, weight;
    public Human(int a, int w) {
      age = a; weight = w;
    }
```

```
    public void addAge(int i) {
      age += i;
    }
    public void addWeight(int i) {
      weight += i;
    }
    public String toString() {
      return "Human("+age+","+weight+")";
    }
  }
}
```

□ Immutable なクラスの例

```
import java.io.*;

public class Sample4 {
  public static void main(String args[])
      throws Exception {
    Human h1 = new Human(20, 55);
    Human h2 = h1.addAge(5);
    Human h3 = h2.addWeight(10);
    System.out.println("result: "+h3);
  }
  static class Human {
    int age, weight;
    public Human(int a, int w) {
      age = a; weight = w;
    }
    public Human addAge(int i) {
      return new Human(age+i, weight);
    }
    public Human addWeight(int i) {
      return new Human(age, weight+i);
    }
    public String toString() {
      return "Human("+age+","+weight+")";
    }
  }
}
```

2.2 ブロックスコープ

□ 内側ブロックで同名の変数を宣言→外側の名前を隠す

- Algol60 以来→「自由に埋め込める」ことから始まった?
- 読みにくさやバグの原因になりやすい
- C ではあまり問題にならないが、オブジェクト指向だとインスタンス変数/クラス変数を隠すことになるので問題が増す

□ CLU では…

- 内側スコープで外側と同名の変数を定義することを許さない
- インスタンス変数は「変数」ではなく「レコードの欄」(後述)

2.3 制御構造

□ Java の制御構造…C とおんなじ! if、while、for、do-while、switch

- ただし goto はない (代わりに例外機構がある)
- とっつき易さのために C とおんなじにしたのはいかがなものか?

□ 現在の言語のトレンド (?) は「begin-end」「{…}」の要らない言語」

- Bourne Shell、CLU、Ada --- if … elseif … else … end
- Perl --- 文が 1 つでも「{…}」で囲むことを強制

2.4 例外機構

□ 「エラー処理のための goto」の代替?

- 「新しい言語には当然の機構」といいつつ…
- Lisp、CLU、Ada とかなり古くからあったが、普及したのは C++への例外処理の追加と Java の普及から?

□ 例外ならではのよさ

- 処理の mainstream をごちゃごちゃさせない
- 同種のエラーをまとめて 1 か所で処理できる

□ Java での構文

```
try {    ←この辺の { } は省略できない!
  文...
} catch(クラス名 変数) {
  ...
} catch(クラス名 変数) {
  ...
} finally {
  ...
}
```

□ Java ならではの特長:

- 例外をクラスで表し、その包含関係で「どの範囲の例外を受け止めるか」を指定する

```
Throwable
  Error
    ...(各種のエラー)
  Exception
    RuntimeException
      ...(実行時に発生し得る各種の例外)
    その他の例外
      ...
```

- 例外もオブジェクトなので任意のデータ構造を持つことができる

- finally ブロック --- 例外で脱出するときも必ず実行される処理を記述できる

□ 例: 自前の例外を定義

```
public class Sample5 {
  public static void main(String args[]) {
    try {
      int i = (new Integer(args[0])).intValue();
      System.out.println("sqrt(i) = "+mysqrt(i));
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  private static double mysqrt(int i)
      throws Negative {
    if(i < 0) throw new Negative(i);
    return Math.sqrt((double)i);
  }
  static class Negative extends Exception {
    int value;
    public Negative(int i) { value = i; }
    public String toString() {
      return "Negative: <"+value+">";
    }
  }
}
```

2.5 例外の受け止めと伝播

□ 言語によってさまざまな流儀

□ 言語サポート→あり/なし (例: C の signal、setjmp/longjmp)

□ 例外の種別→あらかじめ決まったもの/ユーザ定義可能

□ 例外の宣言→しないと投げられない/宣言不要 (例: Lisp)

□ 例外の伝播→必ず受け止める必要(例: CLU)/同じ例外が伝播

□ 例外の受け止めのチェック→あり/なし

□ Java では例外は原則として宣言しないといけない

- ただし Error と RuntimeException はどこでも発生し得る→宣言を強制すると繁雑すぎるため「あらかじめ宣言されている」ものとみなす
- 上記以外の例外を投げるメソッドを呼ぶ場合は→必ずその例外を捕まえるか、または自分自身も同一の例外を投げられる必要がある→ (先の「main(...) throws Exception」はこのため)

□ この辺、「きちんとさせる」思想なのは好感が持てる

- 「エラーフラグ」「戻りコード」なんかだと無限にさぼれる…
- その意味ではもっと「きちんとさせる」手も

3 Java とオブジェクト指向

□ 「オブジェクト指向」「オブジェクト指向言語」の定義…?

- △「レシーバ セレクタ 引数…」のようなメッセージ送信式 (cf. CLOS)
- △「クラス」を持つ (cf. Self)
- △「継承」を持つ (cf. 多くの並列オブジェクト指向言語)

□ 結局、どれか 1 つの性質を持って「オブジェクト指向言語」とは言えない

- ○オブジェクト指向→「もの」と「もの同士の関係」に基づいてシステムを見るという「思想」「考えかた」
- ○オブジェクト指向言語→「オブジェクト指向の考え方をサポートしてくれるような言語」

3.1 オブジェクト指向言語の「純粋さ」

□ Smalltalk-80 →「純粋な」オブジェクト指向言語

- すべての値がオブジェクトである
- すべての操作がメソッド呼び出しである
- 制御構造もオブジェクトのメソッドである (→拡張可能言語???)
- クラスもオブジェクトである (→自己反映計算)

3.2 「すべての値がオブジェクト」とは?

□ Simula、C++、Java →基本型 (整数、文字、実数、…) はオブジェクトでない→効率のため

□ しかしプログラマにとっては極めて不便

- 「Object 型の変数に何でも入れられる」性質に制約が生じる
- 基本型を操作するメソッドの置き場所がない
- Java ではこのために包囲クラス (wrapper class) を設けている
- しかし結局、int と Integer、float と Float などを使い分けるのは繁雑

□ Smalltalk-80 ではすべての値はオブジェクト (Lisp とかもそう)

□ 言語としては、すべての値がオブジェクトである方が絶対よい

□ 例: Vecotor に色々なものを格納する

```
import java.util.*;

public class Sample6 {
  public static void main(String args[]) {
    Vector vec = new Vector();
    vec.add(args);
    vec.add("Hello, World.");
    vec.add(new Integer(10));
    vec.add(new Character('X'));
    for(Enumeration e = vec.elements();
        e.hasMoreElements(); ) {
      Object o = e.nextElement();
      System.out.println(o);
    }
  }
}
```

3.3 「すべての操作がメソッド呼び出し」とは?

□ Smalltalk-80では「1 + 2」は「1」というオブジェクトに「+ 2」というメッセージを送ることに相当。

□ CLUでは「1 + 2」は「int$add(1,2)」の構文糖

- C++では「operator+() によってユーザ定義クラスに演算子をつけられる
- Javaでは「+」は基本型どうしの演算に限定

□ メソッド呼び出しだと何がよいか?

- オーバライドの可能性→基本型そのもにに対してはなさそうだが…
- ユーザ定義クラスを使う際の「見ためのよさ」→結構重要 (STLなど)

□ Javaでも「x + y」を「x.add(y)」に機械的に書き換えるようにコンパイラで手直しすることはごく簡単→あってもよい機能では???

3.4 「制御構造もメソッド」とは?

□ Smalltalk-80の場合→「ブロックオブジェクト」が重要な役割

- ブロックとはメソッドのコードの断片だがそれ自身オブジェクト

```
z ← [x ← x + 1. ↑ x] value.
z ← [:n | x ← x + n. ↑ x] value: 100.
```

- Booleanクラスや Blockクラスのメソッドで制御構造を実現

```
(x < 10) ifTrue: [x ← x - 1] ifFalse: [ ... ].
[x < 10] whileTrue: [ ... ].
```

- そのほかにも map関数、整列の比較関数、探索のマッチ関数などもブロックとして渡すスタイル (Lispではあたりまえ)

□ 何がよいか?

- 制御構造が拡張できる…本当に拡張するんだろうか?
- 動作の断片を引数等で渡せる→柔軟性

□ Javaでは「内部クラス」「無名内部クラス」でかなり対応

□ 無名内部クラスを用いた whileTrue:

```
public class Sample7 {
  static int x = 0;
  public static void main(String args[]) {
    (new Block() {
       public boolean doit() { return getX() < 10; }
     }).whileTrue(new Block() {
       public boolean doit() {
         System.out.println("> "+getX());
         setX(getX()+1);
         return true;
       }
     });
  }
  public static void setX(int i) { x = i; }
  public static int getX() { return x; }
  static abstract class Block {
    public abstract boolean doit();
    public void whileTrue(Block b) {
      while(this.doit()) b.doit();
    }
  }
}
```

4 型/値/データ構造

□ 型とは→値の種類

- 値の種類とは→「その値にどういう操作が施せるかの集合」(抽象データ型的みかた)

□ オブジェクト指向言語では…「値 (オブジェクト) の種類」==「クラス」→クラスを型と対応させるのは自然な帰結

4.1 さまざまな型

□ Pascalでは…基本型 (整数、実数、文字、列挙型)+構造型 (レコード、配列、集合、ポインタ)

- Cでもだいたいこれに相当 (unionが加わっている)

□ C++では…クラスは「レコードの特別なもの」

- クラスは必ずしも参照 (ポインタ) でない→クラス値の埋め込み
- 記憶領域管理の面で多少いいこともあるかも知れないが…
- プログラミングの上ではすごく繁雑になるだけ

4.2 Java における型

□ 前述のように、基本型は特別扱い→不満な点

□ それ以外の値→すべてオブジェクト型、すべてのオブジェクト値はポインタ→簡潔で好ましい点

- GC が前提となる→まっとうなオブジェクト指向言語では当然のこと

□ よくある話題: 「列挙型が欲しい」

- Java では int 型の final(書き換え不能) 変数に定数値を入れることで代用
- これでは C の#define と同じという批判も
- その代わり、クラスをまたがって違う値を設定できるという話も

□ 例: 列挙型の代用

```
public class Sample8 {
  public static void main(String args[]) {
    System.out.println(X.x1);
    System.out.println(X.x2);
    System.out.println(Y.y1);
    System.out.println(Y.y2);
  }
  public static class X {
    public static final int x1 = 0;
    public static final int x2 = x1 + 1;
  }
  public static class Y {
    public static final int y1 = X.x2 + 1;
    public static final int y2 = y1 + 1;
  }
}
```

4.3 データ構造とオブジェクト指向

□ データ構造→そのデータ構造を表すような値の構成→型→クラス

□ 静的なデータ構造→配列、レコード、union、その組合せ

- 配列はある (後述) が、レコードはクラスに対応? union は?

□ 動的なデータ構造→ポインタでつなぎあわせて構成

- オブジェクト指向にはポインタはない???
- C、C++では…

```
struct node {
  xxx info;
  struct node *left, *right; } ...
```

- Java では…

```
class Node {
  xxx info;
  Node left, right; ←「ポインタだから」と見る?
  ...                 「再帰的な定義」と見る?
}
```

□ 例: スタックによる辿りと再帰的辿り

```
public class Sample9 {
  public static void main(String args[]) {
    Node n = new Node('+',
               new Node('*', new Node('1'),
                             new Node('2')),
               new Node('3'));
    print2(n); System.out.println();
    n.print1(); System.out.println();
  }
  public static void print2(Node n) {
    int i = 0;
    Node[] a = new Node[100];
    a[i++] = n;
    while(i > 0) {
      Node n1 = a[--i];
      System.out.print(n1.ch);
      if(n1.right != null) a[i++] = n1.right;
      if(n1.left != null) a[i++] = n1.left;
    }
  }
  static class Node {
    public char ch;
    public Node left, right;
    public Node(char c, Node l, Node r) {
      ch = c; left = l; right = r;
    }
    public Node(char c) {
      ch = c; left = null; right = null;
    }
    public void print1() {
      System.out.print(ch);
      if(left != null) left.print1();
      if(right != null) right.print1();
    }
  }
}
```

4.4 型検査

□ 弱い型の言語→変数に型がない。データには通常、実行時に定まる型がある (例外:Bliss、BCPL、…)

□ 強い型の言語→変数に型がある。コンパイル時に型検査を行う

- 厳密な型の言語→変数の型と実行時の型は常に一致→実行時の検査は不要→実行時の効率はよい/柔軟性はない

  int x = 1; ←左辺と右辺は常に型が一致

- 厳密でない強い型の言語→変数の型と実行時の型が同じとは限らない→実行時の検査が必要→実行時の情報も必要/柔軟性がある

□ オブジェクト指向言語では→「親クラスの変数に子クラスが格納できる」(Simula 以来の規則)

```
Object obj = new X();
...
X x1 = (X)obj;  ←実行時検査
```

- (ある程度まで)union 型の機能を提供している
- ただし「あるクラス以下」という範囲なので「これとこれ」とは言えない。基本型も wrapper に入れないと使えない。

□ 例: Object 型にいろいろなものを入れて取り出す

```
import java.awt.*;

public class Sample10 {
  public static void main(String args[]) {
    print(new Point(10,100));
    print(new Integer(5));
    print("ABC");
  }
  public static void print(Object o) {
    if(o instanceof Point) {
      Point p = (Point)o;
      System.out.println("Point: "+p.getX()+
                         ","+p.getY());
    } else if(o instanceof Integer) {
      int i = ((Integer)o).intValue();
      System.out.println("Int: " + i);
    } else {
      System.out.println("???: " + o);
    }
  }
}
```

4.5 実行時の型と動的分配

□ 動的分配 (dynamic dispatch) →オブジェクト指向の強力さの源の 1 つ

- 基本的には「インスタンスのクラスに応じたメソッドの選択」

```
Vechile v = ... (Car、Bike、Track)
...
v.turnRight();  ← v に格納された実態によって
                  どのメソッドを呼ぶか自動選択
```

- これが CLU などだと次のようになってしまう

```
...
if v が自動車 then Car$turnRight(v)
elseif v が自転車 then Bike$turnRight(v)
...
```

□ なぜ強力なのか?

- △コードが短くて済む←それなりに重要
- ◎コーディング時に想定していなかった選択肢が実行時に自由に追加可能 (乗物はいくつでも増やせ、その際に際コンパイルが不要)

□ 例: 先の例を動的分配に書き換え

```
import java.awt.*;

public class Sample11 {
  public static void main(String args[]) {
    print(new UnionPoint(new Point(10,100)));
    print(new UnionInt(5));
  }
  public static void print(Union u) {
    u.print();
  }
  static abstract class Union {
    public abstract void print();
  }
  static class UnionPoint extends Union {
    Point pt;
    public UnionPoint(Point p) { pt = p; }
    public void print() {
      System.out.println("Point: "+pt.getX()+
                         ","+pt.getY());
    }
  }
  static class UnionInt extends Union {
    int it;
    public UnionInt(int i) { it = i; }
    public void print() {
      System.out.println("Int: "+it);
    }
  }
}
```

5 クラス/インタフェースと継承

□ クラス方式のオブジェクト指向言語→オブジェクト指向言語の主流

- cf. CLOS 方式、Self 方式、…
- 言語屋にとっては「遊べる材料」が増えたことに…

5.1 クラスの役割

□ 「データの種類」(型) を定義し、名前をつけること→抽象データ型に他ならない

- クラスを「雛型」としてオブジェクト (インスタンス、実体) を生成

□ クラス定義に含まれるもの…(Simula、Smalltalk-80 以来の伝統)

- クラス変数/インスタンス変数の定義
- クラスメソッド/インスタンスメソッドの定義

□ ということは、クラスは「実装を定め」ている。

- 実装を規定することを通じてインタフェースを定めている、ともいえる

5.2 継承

□ 継承の古典的解釈…(Simula、Smalltalk-80 以来の伝統) →インスタンス変数定義、メソッド定義を親クラスから子クラスへ引き継ぐ

- その結果として実体の互換性も生まれる
- 変数やメソッド定義の追加→機能/ふるまいの追加 (上位互換)
- オーバライド (メソッド定義の差し替え) →互換性についてはプログラマ次第

5.3 継承の用途

□ 継承にはさまざまな用途が「相乗り」している

- △親クラスを下敷にして少ない労力 (再利用) で新しいクラスを作る (差分プログラミング)
- ○型どうしの親子関係を定義する
- ○動的分配の基礎となる

□ 実装を利用したいだけなのに型としての関係ができるのは嬉しくない

5.4 多重継承

□ 複数の親から継承すること。継承の目的は上と同じ

□ 言語仕様上も実装上も難しい

- オブジェクトの形 (単一継承のように美しく実装できない)
- 複数の親が共通の祖先を持っていたら? (repeated inheritance) → C++ではそれらを「重複して持つ」「重ねる」という 2 つの選択肢

□ 実装を多重継承するのは面倒で複雑

□ Java も (クラスの/実装の) 多重継承は行わない

5.5 継承と委譲

□ 委譲とは→「自分で実装しないメソッドを他のオブジェクトにたらいまわす」こと

□ メソッド定義を継承する代わりに、そのメソッドを「呼び出しても」よい→委譲は継承の代わりに使うことができる

- 継承で生じるいろいろな問題がない
- 実行時に動的に委譲先を切替えることができる
- デザインパターンで多用

□ 言語として委譲を前面に出したもの→ Self

□ 例: 簡単なスレッド/継承版

```
public class Sample12 {
  public static void main(String args[]) {
    Thread tr = new Thread1();
    tr.start();
    try {
      tr.join();
    } catch(InterruptedException e) { }
  }
  static class Thread1 extends Thread {
    public void run() {
      for(int i = 1; i < 10; ++i) {
        try {
          Thread.sleep(500);
        } catch(InterruptedException e) { }
        System.out.println(i);
      }
    }
  }
}
```

□ 例: 簡単なスレッド/委譲版

```
public class Sample13 {
  public static void main(String args[]) {
    Thread tr = new Thread(new Runnable1());
    tr.start();
    try {
      tr.join();
    } catch(InterruptedException e) { }
  }
  static class Runnable1 implements Runnable {
    public void run() {
      for(int i = 1; i < 10; ++i) {
        try {
          Thread.sleep(500);
        } catch(InterruptedException e) { }
        System.out.println(i);
      }
    }
  }
}
```

5.6 界面と実装の分離

□ 結局「型どうしの関係」はインタフェース (界面) の問題→界面についてのみ、継承ができればよい

- 界面は多重に継承しても問題がない (矛盾があれば検出でき、実装と違って動かしてみないとうまく行くかどうか分からないということはない)
- 界面は継承、実装は委譲によればよい→ Misty の考え方
- Java では実装は単一継承、インタフェースは多重継承

5.7 Javaのインタフェース機能

□ 構文的にはクラスと同様だが、変数定義、メソッド定義を含まない(定数は定義できる)

- インタフェースもクラスと同様に型として扱われる
- クラスはインタフェースを実装(implements)できる

□ クラスの継承「も」残っているのはどんなものか?

- 継承が「一切ない」方がすっきりするけれど過激?
- 単一継承のクラス階層はそれなりにあった方がよいという説も

□ より過激な考え方: クラスは変数の型としては使わない(使えないような言語にする)→各コードをよりよく実装から分離できる

```
IIII var = new CCCC(...);
↑界面          ↑実装
```

- さらにこれを進めると「特定のクラスを決め打ちで書くことは避けたい」ことになる→デザインパターンのFactoryMethodパターン

5.8 インタフェースの別の用途

□ 処理系に「このクラスはこういうクラス」と教えるために使う→メソッドが1個も定義されていないインタフェースを使う

- 例: implements Clonable → clone()してよいことを表す
- 例: implements Comparable → 線形順序を持つことを表す
- 例: implements Serializable → 直列化可能であることを表す

6 Javaとメソッド

□ メソッドの実行モデル→ごく標準的な手続き実行

- Pascalのように囲むブロックの変数をアクセスしないでいい→実現が容易
- いっぽう、クラス変数/インスタンス変数はメソッドの外側にある変数として利用できる→それなりに便利

□ メソッド呼び出しでなく「メッセージ送信」という言い方も(Smalltalk-80)→並列ぽい感じ。実際にはSmalltalk-80でも手続き呼び出し。

- 並列オブジェクト指向言語だとこの辺は違う(Javaは並列オブジェクト指向言語ではない)

6.1 引数/返値機構

□ 教科書的な→「名前呼び」「値呼び」「参照呼び」

□ Javaみたいにオブジェクトの参照を渡すのは?→「call by sharing」(CLUでの用語)、「オブジェクト呼び」

- 実質的には値呼び
- 副作用を返したいなら渡されたオブジェクトを変更

□ オブジェクトには変更不能なものもある

- →返値を複数持てるとよい(CLU、CommonLisp)。これがJavaに入れられなかったのは不満(JVMでは実装は何ら問題ないはず)。

6.2 オブジェクトアクセス

□ インスタンスメソッドは暗黙のうちにカレントオブジェクト(this)を渡されている→標準的なやりかた

□ メソッド内側ではそのクラスのクラス変数、インスタンス変数へのアクセスが許される→「中が開いている」モデル。C++も同様。

□ c.f. CLUでは→「抽象データ型」と「内部実現型」の間の視点の切替えがある

□ このため、CLUではオブジェクトの実現が整数型などであってもよい。

- Javaなどでは必ずimplicit recordになってしまう(そうでない実装があってもよいかも???)

6.3 コンストラクタ

□ C++で始まった、オブジェクトを生成するとき自動的に呼び出される「初期設定メソッド」

- コンストラクタの実行中はオブジェクトが「半完成状態」にあるので注意が必要
- コンストラクタは「クラス名と同名」のため継承されない→何か間違っている感じがする(親のコンストラクタを呼ぶ機能はある)

6.4 オーバローディング

□ オーバローディングとは→ソースコード上では同名のメソッドをコンパイル時に識別して呼び分けること

- 演算子はもともと「オーバロード」されている(+→整数の+、実数の+)

- 言語として大々的にはじめたのは Ada、引数と返値による識別
- Java では引数の個数と型だけが識別に使用される

□ オーバローディング (静的に解決) と動的分配 (実行時に解決) がまざっているとやっぱりごちゃごちゃしないか? 統一した方が美しいのでは?

- 些細な問題だが、コンストラクタは名前が決まっているので複数定義するとオーバロードにならざるを得ない

7 Java と自己反映機能

□ 自己反映機能とは→処理系自身に関する情報をプログラム側で取得/変更するような機能

- CLOS などでは「変更する」方もあり→たとえば動的分配のふるまいなどを修正することもできる
- Java では「取得」だけサポート→それでも昔の C++ よりずっと使いやすい (現在の C++ではある程度サポートがある)

7.1 型に関する情報

□ オブジェクトの実行時の情報で一番必要なのは「このオブジェクトは具体的には何の型か?」

- Java では instanceof 演算子で「ある型のサブクラスかどうか」を直接判定可能
- しかしそもそも、「クラスオブジェクト」が存在し、すべてのオブジェクトは Object 型から継承している getClass() でそのオブジェクトが取れる→そのメソッドとかにしておいてもよかった???
- オブジェクトをダウンキャストする際、指定したクラス以下でなければ例外が投げられる→instanceof と同じことだからこれでよい?

7.2 クラスオブジェクト

□ クラスオブジェクト→ 1 つの型に対応 (インタフェースにも)

□ 基本型はオブジェクトではないが、なぜかクラスオブジェクトはある→統一的に扱うため。かなり気持ちわるい

□ クラスオブジェクトの各メソッドを使って、そのクラスのインスタンスがどんな変数やメソッドを持っているか、またそのシグニチャは何かなどがすべて調べられる

- さらに、メソッドオブジェクトを経由してそのメソッドを呼び出すことも可能
- 値の受け渡しには Object[] を多用。さらに基本型なら包囲クラスを使用

□ 例: 任意のクラスを生成してメソッドを呼ぶ

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

public class Sample14 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    while(true) {
      try {
        System.out.print("Class Name? ");
        System.out.flush();
        String cname = br.readLine();
        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? ");
        System.out.flush();
        int cno = Integer.parseInt(br.readLine());
        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? ");
        System.out.flush();
        int mno = Integer.parseInt(br.readLine());
        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(); }
    }
  }
  public static class Test {
    int i;
    public Test() { i = 0; }
    public Test(int n) { i = n; }
    public Test next() { return new Test(i+1); }
    public void print() {
      System.out.println("Test value: "+i);
    }
  }
}
```

7.3 JavaBeans と自己反映機能

□ JavaBeans → COM/DCOM のようにインスタンスベースでの組み立てを可能とするアーキテクチャ

- インスタンスベースのカスタマイズが可能である必要

- ツールが「このインスタンスはどんな操作があるか」を見て、その情報に基づいてカスタマイズを実行
- 具体的には「getXXX()」というメソッドがあれば「XXX」というプロパティがあるものと判断する。さらに「setXXX()」というメソッドがあればそのプロパティは書き換え可能だと判断する
- もっと高度なカスタマイズ→ ZZZ という Bean クラスと一緒に ZZZBeanInfo というクラスがパッケージされていて、なおかつそれが implements BeanInfo であれば、このクラスを用いてさまざまな情報を受け渡す

8 Java の内部クラス機能

□ JDK 1.1 から導入された機能

□ 単に「クラスの中にクラスが書ける」というだけ…

- だけど、結構奥が深く、使い手がある
- 前提として、JVM を変更せず、コンパイラだけで実装←既に動いている多量の JVM に対する配慮← HotJava α 3 から JDK 1.0 への移行時の混乱

8.1 内部クラスとは

□ クラスの中にクラス定義が書ける

- たとえば Adaptor パターンや Command パターンを使うと「クラスだらけ」になるが、内部クラスにしておけばソースがごちゃごちゃにならずに済む

8.2 無名内部クラス

□ 多くの場合、内部クラスは 1 箇所でインスタンスを生成してそれでおしまい→いちいち名前をつけたり考えるのは無駄な感じ

- そこで、「このクラスのサブクラス」「このインタフェースを実装してるクラス」という形で、名前は無名のままで済ますことができるようにした。
- 名前がないのでコンストラクタが作れない→代わりに「インスタンスイニシャライザ」を用意した←やっぱりコンストラクタは…

8.3 内部クラスから囲む環境へのアクセス

□ 内部クラスのインスタンスからそれを生成したインスタンスメソッドの this を参照できる (何段階でも) →当然、自分の this もあるから、this が複数ある。それらを参照するには「クラス名.this」(気持わるい!)

- 当然、それらのメソッドも (private でなければ) 呼べる
- このほか、ローカル変数や引数も final 指定 (つまり読み出しのみの変数や引数) なら参照できる

8.4 GUI と内部クラス

□ JDK 1.0.x までは、GUI 部品は「サブクラスを作ってそこで動作をオーバライドする」方式だった (Thread のサブクラス版と同様)

□ JDK 1.1 からは「部品はサブクラス化せず、アダプタクラスを設定する」方式に

- 部品の抽象データ型を壊さずに動作を独立してつけられる
- GUI 生成ツールなどにより適した方式 (特に Beans などに)
- 多数のアダプタクラス→内部クラスの使用が (ほぼ) 前提に

□ 例: 内部クラスによる GUI 部品のアダプタ

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Sample15 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  String str = "";
  Label lab = new Label(str);
  Button b1 = new Button("A");
  Button b2 = new Button("B");
  public void init() {
    setLayout(null);
    add(lab); lab.setFont(fn);
    lab.setLocation(20, 20); lab.setSize(200, 40);
    lab.setBackground(Color.white);
    add(b1); b1.setFont(fn);
    b1.setLocation(20, 80); b1.setSize(60, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        addStr("A"); repaint();
      }
    });
    add(b2); b2.setFont(fn);
    b2.setLocation(120, 80); b2.setSize(60, 40);
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        addStr("B"); repaint();
      }
```

```
    });
  }
  public void paint(Graphics g) {
    lab.setText(str);
  }
  public void addStr(String s) {
    str = str + s;
  }
}
```

8.5 内部クラスの実装

□ JVM を変更しないという前提→ソースレベル変換で可能な設計

- ただし名前は衝突しないようにシステムに予約された「$」を使う。名前があれば「OuterClass$Inner」、なければ「OuterClass$0」のような名前になる
- 外側の this、ローカル変数、引数も「$」つきのインスタンス変数に格納 (final だと変更はされないからコピーを持てばよい)

□ 内部クラスのインスタンスから直接変数を書き換えない、というのは言語設計としてもそれなりによいと思うし実装も楽ができる

8.6 内部クラスの使いで

□ 上述の Adapter などにはもちろん重宝

□ 普通のオブジェクト指向言語なのに、Smalltalk-80 のブロックがあるみたいな→なんか変な気分

- もちろん、アイデア次第でいろいろに使える
- C の「関数ポインタを渡す」代わりに使うこともあり
- × Microsoft は独自拡張で Java に関数ポインタを入れようとしたが…

9 パッケージと名前空間

□ ブラウザ上の複数アプレット、JavaOS などで分かるように、JVM 上には多数の独立した Java プログラムが共存→名前の衝突とかは?

- パッケージ機能→多数のライブラリを階層構造で整理
- しかし名前空間の分離はパッケージと別に必要

9.1 パッケージ機能

□ クラスを「グループ化し」「一意的な名前をつける」

- ファイル先頭の「package a.b.c.d;」といった宣言で指定
- パッケージの階層構造とディレクトリの階層構造が対応する
- JDK 1.0.x では大袈裟だと思ったが JDK 1.2 ではこれくらいないと標準クラスでも大変。まして他のベンダーのクラスまで含めると…

□ 標準以外のパッケージ名→ドメイン名をもとにつける

- 例: xxx.co.jp → package jp.co.xxx;
- ネットワーク上で一意的になる→なかなか興味深い
- java.*、javax.*については標準/標準拡張クラス用に SUN が予約

□ クラス名の代わりに常にフルクラス名 (パッケージまで含めた名前) を指定することができる

9.2 名前空間

□ パッケージを指定しない場合は「無名パッケージ」に属するものとして扱われる

□ パッケージ名の規定は強制するメカニズムが (今のところ) ない

□ ということは「名前の衝突」は考える必要がある←単一言語系における昔からの問題

□ Java での解決策→「クラス名単独ではなく、(クラス名、クラスローダのインスタンス)」の対でクラスの一意性を識別する」

- システムクラスローダ→システムクラス (CLASSPATH 環境変数で指定されているディレクトリにあるクラス) をロード→システムクラスはすべてのプログラムで共通
- AppletClassLoader →アプレットをロード/強制再ロードするごとにインスタンスができる→それぞれはたとえ同じコードであっても別の名前空間になる
- その他特別なクラスローダを作った場合も同様

10 Java と並列

□ Java は並列言語か? (YES/NO)

- Java の並列実行モデル→スレッド

- Java の並列実行に対するサポート→オブジェクトにロックが組み込み
- なぜ並列言語とは言えないか? 何が問題か?

10.1 Java の並列実行モデル

□ スレッドモデル→ Thread ないしそのサブクラスのインスタンスに対して start() というと、指定したコードが独立したスレッドとして実行

- ということは、オブジェクトとスレッドが直交している
- c.f. 並列オブジェクト指向言語→オブジェクトがスレッドと密接に関連している

□ 何がよくないか→共有メモリモデルの相互排斥をしながらのプログラミングを強いられる。Ada などの方がずっとまし

- Par Brinch Hansen の批判

10.2 Java の並列実行に対するサポート

□ オブジェクトにロックが付随している

- synchronized(...) { ... } で排他領域を作成できる。ただし advisory
- 条件同期→ wait()、notify()、notifyAll() による

□ 例: 有限バッファ

```
public class Sample16 {
  public static void main(String args[]) {
    BoundedBuf b = new BoundedBuf();
    Thread t1 = new Putter(b, 100, 20);
    Thread t2 = new Putter(b, 200, 30);
    Thread t3 = new Getter(b, 50);
    t1.start(); t2.start(); t3.start();
    try {
      t1.join(); t2.join(); t3.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
  static class Putter extends Thread {
    BoundedBuf buf;
    int start, count;
    public Putter(BoundedBuf b, int s, int c) {
      buf = b; start = s; count = c;
    }
    public void run() {
      try {
        for(int i = 0; i < count; ++i) {
          System.out.println("put "+(start+i));
          buf.put(start+i);
        }
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
  static class Getter extends Thread {
    BoundedBuf buf;
    int count;
    public Getter(BoundedBuf b, int c) {
      buf = b; count = c;
    }
    public void run() {
      try {
        for(int i = 0; i < count; ++i) {
          System.out.println("got "+buf.get());
        }
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
  static class BoundedBuf {
    int[] a = new int[10];
    int ipt = 0, opt = 0;
    public synchronized void put(int v)
        throws InterruptedException {
      while((ipt+1)%10 == opt) wait();
      a[ipt] = v; ipt = (ipt+1)%10; notifyAll();
    }
    public synchronized int get()
        throws InterruptedException {
      while(opt == ipt) wait();
      int x = a[opt]; opt = (opt+1)%10;
      notifyAll(); return x;
    }
  }
}
```

10.3 何が問題か?

□ 並列実行単位間の競合が野放しで「手当したい人は synchronized を使ってね」になっている→過去の経験の放棄

□ 条件同期のためのキューがモニタ 1 つにつき 1 つだけ→複数の条件で寝る場合に困る

- 定石として「全員起こして、条件が OK の人以外はまた寝る」ことを繰り返す→ものすごく無駄

11 Java と分散

□ Java は分散言語か? (YES/NO)

□ 「分散オブジェクト技術 (distributed object technology)」→分散言語でないオブジェクト指向言語で分散プログラミングをするための技術

□ 分散言語の方がいいのかどうか? いいとすればどうして?

11.1 Javaによる古典的な分散プログラミング

□ 要するに「ソケットプログラミング」→ネットワーク上のStreamを使えばわりあい簡単

□ しかしJavaだからどうということはない

11.2 分散オブジェクト技術

□ CORBA → JavaIDL + CORBA製品

- ○他言語との相互運用性がある
- ○インタフェースからCORBAインタフェースができる→有利な点
- △汎用なだけあって遅そう

□ HORB →国産!!!

- 電総研の平野　聡さんが作ったJava用ORB+コンパイラ+ランタイム
- ○ CORBAやRMIがまだなかったころから動いていた→先見の明
- ？効率は「よい」という主張と「よくない」という主張がある
- △必ずしも標準でない、制約が強いなどの弱点

□ RMI (Remote Method Invocation)

- 普通のRPC。クライアントスタブ+サーバプロキシ
- Javaならではの利点→スタブを予めインストールしなくてもネットワーク経由でロードして来られる

□ RPCベースでいいのか?　それってオブジェクト指向なの???

11.3 JavaSpaces/Jini

□ Linda →古くからあるもう1つの(というのはRPC、メッセージ送信と対比して言っている)分散言語モデル

- TupleSpaceと呼ばれる「エーテル」に「タプル」を送出→マッチングにより取り出す
- 相手を特定しない/時間を限定しない通信が可能

□ Javaでこの通信モデルを採用することの意味…

- 家電製品、ホームバスなどでは「どういうモノがつながっているかの発見」「それらの間でのプロトコルの交渉」などが必要
- そのためには、「マッチングによる情報の取得」「時間に束縛されない」「誰がいるか分からなくても情報を送出できる」といった性質が極めて有用

□ 言語そのものの話じゃなかったですね

12 Javaとセキュリティ

□ Javaの普及の原動力→アプレット→「ネットワーク経由でやってくるコード」→安全でなければ誰も使わない→セキュリティのための設計が最初からある

- 言語処理系/実行系に基づくセキュリティ→古くからあるアイデアだがこれほど本格的に普及したのはJavaが最初

□ いくつかの変遷

- ～JDK 1.0 →セキュリテイサンドボックス
- JDK 1.1 →署名アプレットによるサンドボックスからの逸脱
- JDK 1.2 →より一般的なセキュリテイモデル

12.1 セキュリティサンドボックス (～JDK 1.0)

□ アプレットは「セキュリティサンドボックス」で実行

- その中ではファイル読み書き禁止、ネットワーク接続はダウンロード元サーバに限定
- 実装→セキュリティマネージャクラスのcheckAccess()等を呼んでチェック
- この中では、「何レベル以内にユーザクラスのコードがあるか」を場面ごとにチェック→正しく実装するのが難しい

12.2 署名アプレット

□ サンドボックスの制約を逃れたいという強い要求

- アプレットクラス群をJARファイルと呼ばれるアーカーブ形式に格納し、全体に署名を施す
- 実行時に署名を検証できたら、サンドボックスを外す

□ All or nothingというのは問題がある

12.3 JDK 1.2のセキュリティモデル

□ より柔軟なセキュリティモデル

- すべてのコードおよび実行のインスタンスには暗黙のうちに「権限」が存在している
- あるコードから別のコードを呼ぶ際、権限は両者のうち弱い方に合わせられる
- それだけだと困るので、「ここは特にチェックしたから本来の権限に強める」という「おまじない」を用意する

```
Object val = AccessController.doPrivileged(
  new PrivilegedAction() {
    public Object run() {
      ... // この中が権限が強い
    }});
```

- なお、昔は次のようになっていたがこれでは問題 (なぜか?)

```
AccessController.beginPrivileged();
... // この中が権限が強い
AccessController.endPrivileged();
```

13 モジュールパラメタ/型パラメタ

□ モジュールパラメタ→「コンパイル時に束縛する」パラメタ

- 例: 配列の大きさみたいなもの

□ モジュールが提供している機能が「型をつくり出す」場合→パラメタによって違う型ができることも

□ さらに、パラメタ自体も型であってよい→型パラメタ→これだけあれば他のパラメタはあまり必要とされない

- もともとは CLU、Ada など抽象データ型言語で始まる
- しかし実は配列型なども型パラメタを持つと考えられる

13.1 Java と配列型

□ 配列→「何を格納する配列か?」→パラメタのある型 (ないし型生成子)

□ 言語によりさまざまな流儀

- C、C++では配列型はあるが変数はみなポインタ→さまざまな問題
- Pascal では配列は「要素型、添字型」をパラメタに持つ
- Java では配列は要素型のみをパラメタに持つ

□ 型パラメタ機構があれば、それを用いて統一的に扱えるはず?

- Java では型パラメタ機構はなく、配列型だけが特別扱いで、唯一のパラメタを持つ型 (ないし型生成子) になっている

13.2 Contravariance/Covariance 問題

□ Java では B が A のサブクラスなら、B[] は A[] のサブクラス

- 実はこれは問題!!!

□ サブクラス互換性→ B が A のサブクラス⇒ A が持つメソッドはすべて B でも利用可能 (型検査を通る)

- ということは、B が A と同名のメソッドを持っている場合、その引数はより広くなければいけない※

```
A#add(A x) を
B#add(B x) でオーバライドしてはいけない!
```

- ただし Java では上の例はオーバライドにならない (引数が違うので区別可能)
- また、B が A と同名のメソッドを持っている場合、その返値はより*狭く*なければいけない

□ それが配列型とどういう関係にあるかというと???

- Java では B が A のサブクラスなら、B[] は A[] のサブクラス
- A[] に入れることができていたものを、B[] には入れられない→これが※に抵触する

```
A[] a = new B[100];  ←サブクラスだから OK
...
a[10] = new A(...);  ←実行時に例外!
```

- あるべき解決策…(1) 配列の親子関係を設けない (2) 読み出しのみの配列に限って親子関係を設ける

13.3 型パラメタの難しさ

□ 配列みたいに「入れておくだけ」のもの (コンテナクラス) は型パラメタを持たせてもすべてポインタとして一様に扱える→さほど問題がない

□ 「常に昇順に並んでいる配列」→型パラメタを持つモジュール (クラス) の中から、パラメタ型の操作を呼び出す必要がある→難しい (特に型検査をきちんとしなければならないので)

- CLU、Ada など→「この型パラメタはどれどれの操作を持っている」という情報を明示的に指定する→大変繁雑
- オブジェクト指向では→「このクラスまたはそのサブクラス」という情報があればどのような操作があるか分かる→これを bounded polymorphism と呼ぶ (上界となる型を bound と呼ぶ)。さらに、上界が型パラメタを持つ→ F-bounded polymorphism

13.4 オブジェクト指向には型パラメタ不要?

□ 型パラメタが要らないわけ…あるクラスが「X」というクラスを利用していたとすれば、いつでもその値を X のサブクラスで置き換えることができる→すべてのオブジェクト利用は暗黙のうちに型パラメタである

□ 型パラメタが必要なわけ…???

- ○型パラメタであればコンパイル時にチェックできる
- ○型パラメタであれば実行時にチェックできる
- ○型パラメタであれば効率よく翻訳可能
- ◎型パラメタによって明示的に「切り替わる」箇所を表現できるべき

13.5 Java 言語への型パラメタの導入

□ 現在非常にホットな話題

□ 主要な論点→言語仕様、実装方法、JVM の変更の有無、既存のコードの移行性

□ 代表的な実装の流れ: Pizza → GJ → NextGen

- http://wwwipd.ira.uka.de/~pizza/
- その他にも研究としては沢山あるが、上の流れが「Sun と仲良く」してるので将来の Java 言語に入るのかも

13.6 実装方法の選択

□ 均質な (homogenous) 実装 .vs. 不均質な (heterogenous) 実装

□ 均質とは→コード 1 つで様々な型パラメタに対応

- コードが少なくて済む
- さまざまな制約がある (例: コンストラクタが使えない)

□ 不均質とは→型パラメタごとに別々のコードになる

- C++のテンプレートなど→マクロ展開が前提のような言語仕様→不均質な実装になる→コードが増える
- 混合した形として→共通部分を親クラスとしてくくりだした上で、それを継承する形でパラメタごとのクラスを作るものも

13.7 GJ における均質な実装の手法

□ 前提として、JVM は変更しない。コンパイラのみで対処

- バイトコードには JVM が直接参照しない部分があり、そこにパラメタ情報を記録する

□ 上に述べたように、型パラメタの部分を消去して上界のクラスとしてコンパイルする

- 基本型は包囲型につつんで渡す (いまいち!!)

□ 制約: パラメタ型のオブジェクトの生成ができない←コンストラクタはそれぞれ違うはずなのにコードは 1 つだから

□ キャストや instanceof が駄目←クラスが 1 つになってしまうから当然

- 一般に均質な実装ではリフレクション関係は全滅になる

□ GJ のもう 1 つの特徴: 既にコンパイル済みの (型パラメタを使っていない) コードをシグニチャだけから型パラメタを使ったコードに持ち上げる

13.8 NextGen における均質/不均質の混合実装

□ これも前提として JVM は変更しない

- 共通部分を GJ と同様にパラメタ消去して汎用クラスとして生成
- 各パラメタごとにそれに対応するインタフェースとクラスとを対で生成 (パラメタ情報を name mangling により付随させる) →クラスの方は汎用クラスのサブクラスかつインタフェースを実装
- パラメタ型のインスタンスを生成するなどの部分は、汎用クラス内では snippet と呼ばれる抽象メソッドの呼び出しにしておく
- パラメタごとのクラスでこの snippet をオーバライドしてそこに適切なコードを入れておく

□ JVM を変更すれば何とでもなるだけに、変更しないという前提に基づくチャレンジとはいえ、相当苦しい感じがする

14 さいごに

□ こうしてまとめて見ると、Java は新しい言語だけにこれまでの (伝統的な手続き型系列の) プログラミング言語研究の成果がぎっしり詰まっていて、題材として興味が尽きない。

□ というわけで、つい欲張ってしまって準備が修羅場になり、大変でした。

□ しかも reference をきちんと書く暇がありませんでした。どうも申し訳ありません。だいたいは WWW で検索すれば見つかると思います。