TSSI基盤技術研修コース
— プログラミング言語 — ＃2

久野　靖*

2004.4.23

1　はじめに

1.1　前回の皆様のアンケートを拝見して…

□ 興味深かった点は人によってまちまちでしたが、プログラミング言語やそのさまざまな機能について改めて考える機会になった、という意見が多かったです。

□ 実習しながら聞きたいというご意見も頂きましたがそれは時間がとてもかかるので…

- 講義というTSSIの形態からして、皆様のご希望にそえない部分もあるのはしかたないと考えます。部分的にでも有益な内容があればそこを活用して頂くということで。
- もともとJava言語を詳細にやるという趣旨ではないのでそこもご了承を…

1.2　第2回の構成

□ 「モジュールと保護」の話からはじめて改めてオブジェクト指向を導入

□ オブジェクト指向言語の進化と諸概念を並行して見て行く(歴史的な順序におおむね従った方がわかりやすい)

□ オブジェクト指向言語の実装技術→必要な箇所にそのつど挿入(諸概念が出て来たところで説明した方が納得しやすい)

□ オブジェクト指向言語の利用技術→最後にまとめて解説

□ Javaによる例題/演習→それに適した話題のところに適宜挿入

□ Java以外の言語の例も一部出て来る→言語の対比の必要上

2　モジュールと保護

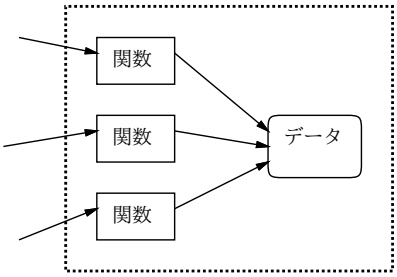
□ 前回スキップした内容を含め、オブジェクト指向の導入になる概念を見て行く

- モジュール
- 抽象データ型
- オブジェクト
- Javaにおける保護(クラス単位、パッケージ単位)

2.1　モジュール

□ モジュール以前の言語→手続きが単位

- 手続きが単位だと「カタマリ」が小さすぎる
- 複数の手続きが1つのデータ構造を共有する→よくあるパターン



□ そのデータ構造は「グローバル変数」じゃいけないの?

□ グローバルだと「よそからいじられる」可能性

- データ構造→通常、「こういう性質を維持」という条件がある(整合性条件)→これが壊れるとそれを前提としているコードも破綻
- グローバルだとそれを分からずにいじられてしまう

□ モジュール機構があれば、データ構造は特定のコードからだけしかいじれなくできる

- 整合性条件の維持→そのデータをいじるコードが限定されていると、そこだけきちんとすれば確実に大丈夫
- さらに、外から直接呼ばれたくない関数も同様にして限定できるとよい。

□ どうやって実装するか?　言語にモジュールがあれば簡単だが、C言語でも「ファイル内のstatic」を使ってある程度できる

*筑波大学大学院経営システム科学専攻

□ 例: 「図形描画できるウィンドウ」

```
---GraphWin.c---
#include <なんとか.h>
static Window *win;
static struct { int x1,y1,x2,y2 } line[1000];
static int linecount;
static int initialized = 0;
static int init() {
  win = ...
  linecount = 0;
}
int addline(int x1, int y1, int x2, int y2) {
  if(!initialized) init();
  line[lincount].x1 = x1; line[lincount].y1 = y1;
  line[lincount].x2 = x2; line[lincount].y2 = y2;
  ++linecount;
}
int update() {
  int i;
  if(!initialized) init();
  for(i = 0; i < linecount; ++i) 線 i を描く;
}
...
```

□ このコードでは「どういう整合条件」があると思うか?

□ こうしておくと他にどういういいことがあるか?

□ 結局、モジュールの機能は「何もしなければアクセスできるものを隠す」こと→カプセル化 (encapsulation)

- 「わざわざ不自由にすること」が実は重要

2.2 抽象データ型

□ モジュールでは隠す対象は「データ一式」

□ では窓を沢山作りたければどうする? →抽象データ型

```
---GraphWin.c---
#include <なんとか.h>
typedef struct {
  Window *win;
  struct { int x1,y1,x2,y2 } line[1000];
  int linecount; } GraphWin;

GrappWin* create_gw() {
  GraphWin *gw = (GraphWin*)malloc(siezof(GraphWin));
  gw->win = ...
  gw->linecount = 0;
  return gw;
}
int gw_addline(GraphWin *gw,
      int x1, int y1, int x2, int y2) {
  gw->line[lincount].x1 = x1;
  gw->line[lincount].y1 = y1;
  gw->line[lincount].x2 = x2;
  gw->line[lincount].y2 = y2;
  ++gw->linecount;
}
int gw_update(GraphWin *gw) {
  int i;
  for(i = 0; i < gw->linecount; ++i) 線 i を描く;
}
...
```

□ なぜこれが「抽象データ型」かというと…GraphWin 型というデータ型を新しく作っていると考えられる。たとえば利用側は:

```
---GraphWin.h---
typedef void *GraphWin;
GraphWin create_gw();
int gw_addline(GraphWin gw, int x1, int y1, int x2, int y2)
int gw_update(GraphpWin gw);
...

---アプリケーション.c---
include <GraphWin.h>
int main() {
  GraphWin gw = create_gw();
  ...
  gw_addline(gw, ...);
  ...
  gw_update(gw);
  ...
}
```

□ この「2 回ずつ言う」あたりを何とかしたのがオブジェクト指向の「メッセージ送信記法」と考えてもらえればよい。

2.3 オブジェクト指向

□ いちいち上のようにヘッダファイルを使い分けたり、第 1 引数でデータ構造のポインタを渡したりすると煩雑→間違い→破綻。

□ これを言語の方で自動的にやってくれると嬉しい

```
class GraphWin {
  Window win;
  Line[] line = new Line[100];
  int linecount;
  public GraphWin {
    win = ...;
    linecount = 0;
  }
  public addLine(int x1, int y1, int x2, int y2) {
    line[linecount++] = new Line(x1,y1,x2,y2);
  }
  public upate() {
    for(int i = 0; i < linecount; ++i) 線を描く…
  }
  ...
  class Line {
    int x1, y1, x2, y2;
    public Line(int px1, py1, px2, py2) {
      x1 = px1; y1 = py1; x2 = px2; y2 = py2;
    }
    public int getX1() { return x1; }
    public int getY1() { return y1; }
    public int getX2() { return x2; }
    public int getY2() { return y2; }
  }
}
```

□ 使う側では次のようになる

```
import ...
public class Test {
  public static void main(String[] args) {
    GraphWin gw = new GraphWin();
    ...
    gw.addLine(...);
    ...
    gw.update();
  }
}
```

- 初期設定とかポインタとかに煩わされなくなった
- メッセージ送信記法→「馬から落馬」がなくなった

2.4 Java の保護機構

□ このように、「個別のデータを」「クラス内外で保護する」まではできた

□ しかし今度はオブジェクト指向言語になると「クラスが山のようにできる」状態に

□ このため、「このクラスはどこからでも使う」「このクラスはここだけの作業用」という区別をしたくなった

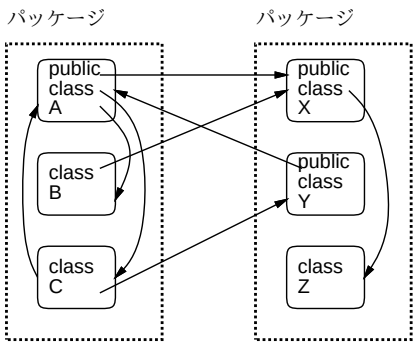
- 入れ子クラス (クラスの中にクラスを入れられる) で十分?

□ 入れ子クラスは「ある特定のクラス内だけで使う下請け」にはベスト

- 「複数のクラスで共通に使う下請け」にはちょっと使いづらい
- 「ブロック構造」と「モジュール」の違いのようなもの

□ このため「パッケージ」という単位をさらに設けた

- Java ではすべてのクラスは「パッケージ内のみで参照可能」か「public(パッケージ外から参照可能)」かのどちらかになる
- ついでに…Java では「public 指定のクラスはそのクラス名と同じ名前のファイルに入れないとコンパイルできない」という規則。
- 逆に言えば、public でないクラスなら一緒に入れても構わない (class ファイルはどのみちクラスの数だけ生成される)。



□ さらに! あるクラスの中に含まれているもの (変数、メソッド) についても次の 4 つの「保護レベル」がある (4 つの p)

- public → パッケージ外からでも参照できる
- 無指定 → パッケージ内からだけ参照できる
- private → そのクラス内からだけ参照できる
- protected → 無指定プラス、そのクラスのサブクラスから参照できる (これについては少し後で)

□ おすすめの保護設定は…

- 変数は原則として private にするのがよい (カプセル化)。final 指定 (変更不可) なら public でもよい
- メソッドは必要に応じて private/無指定/public のどれかに

□ 実際は状況に応じてさらに考える…

2.5 本節のまとめ

□ 「本来見えるものを隠すこと」はプログラム言語では重要

□ 歴史: モジュール (1 個) →抽象データ型 (複数) →オブジェクト指向 (多数) →クラス群ないしパッケージ

□ 多数のクラスの構造化についてはまだ先があるかも…

3 オブジェクト指向言語とその諸概念

□ オブジェクト指向の基本的なアイデアは簡単

□ 実際に言語として設計すると多くの考慮点

□ 歴史的な推移に従って見ていくと分かりやすい

3.1 オブジェクト指向の定義

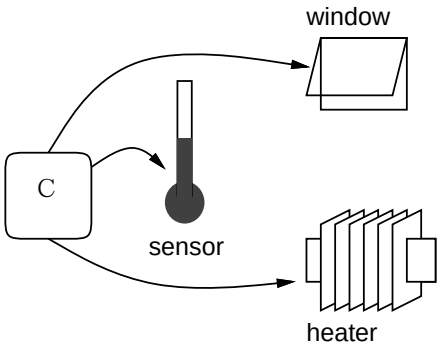
□ オブジェクト指向とは、プログラムが扱う対象のそれぞれを自立した「もの」として扱う「考え方」

□ オブジェクト指向が取り入れられている分野はいろいろある

- オブジェクト指向プログラミング
- オブジェクト指向ソフトウェア工学 (分析、設計、開発、UML…)
- オブジェクト指向ソフトウェア技術 (コンポーネント、分散オブジェクト、…)
- オブジェクト指向データベース
- ここでは「言語」を中心に扱う

3.2 オブジェクト指向的な考え方

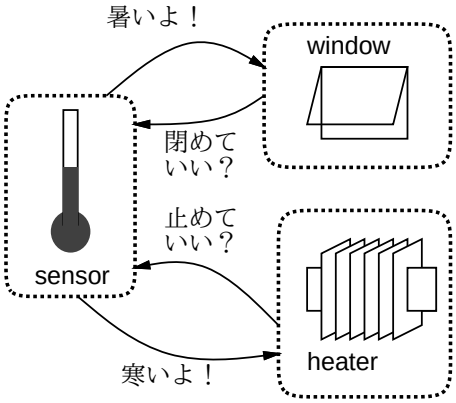
□ たとえば、「温室の温度調節システム」を考える。



□ 旧来の考え方→手続き+受動的なデータ

- 「センサーを見て、気温が下がってきたらヒーターを通電するが、温度が上がりすぎたらヒーターを切る」「気温が上がってきたら窓を開くが、下がってきたら閉める」など機能中心に考える
- 制御する要素や条件が複雑になるとごちゃごちゃになりやすい。

□ オブジェクト指向だと…



- オブジェクト指向→「気温センサ」「ヒーター」「窓開閉装置」などの「もの」を考える→「気温センサ」は温度が低いと「ヒーター」、高いと「窓」に注意を喚起→「ヒーター」は注意を喚起されると、定期的に「センサ」に温度を尋ね、一定以下だと通電、十分暖かいならヒーターを止めて仕事を終る→人間にとって考えやすく、適度な大きさに分けて考えられる。

□ …で。

- 「結局、従来と同じことをやってる」と思いますか?
- 「それは大変よさそうだ」と思いますか?

□ 「オブジェクト指向が」よいかどうか…

- プログラムの動作そのものは同じことをさまざまに書ける。当然。
- 人間にとって考えやすくさせてくれるならそれは「よいこと」。
- 問題は「その分だけ余計な概念が増えて負担になる」ことを差し引いてトータルで儲かっているかどうか。

□ 現在大量に書かれている C++や Java のコード→ある意味では「儲かっている」ことの傍証かも (盲目的にそう信じることはできないが)。

3.3 本節のまとめ

□ 「オブジェクト指向」とは「考え方」

□ 「もの」と「動作」→我々の日常に近い→考えやすい→同じ労力でより複雑なものまで考えられる

□ オブジェクト指向言語→オブジェクト指向のための機能をさまざまに追加 (次節で見て行く)

- 色々追加したことによる複雑さと見合うかどうかが問題

4 オブジェクト指向言語の基本部分

□ 「最初のオブジェクト指向言語」って知っていますか?

□ Simula →最初のオブジェクト指向言語

- 開発されたのは 1960 年代半ば。最終版は Simula67
- もともと「シミュレーションのための言語」→「もの」を「まねする」しくみとしてオブジェクト指向を導入
- しかし現在のオブジェクト指向言語に見られる基本的な仕組みはすべて持っている→極めて先進的
- Simula が持っていた機能 → クラス、インスタンス、カプセル化、メソッド、メッセージ送信記法、継承、包含型、動的分配
- その後追加された基本概念→ インタフェース、抽象クラス、入れ子クラス、メタクラス、自己反映機能、…
- この節では「基本部分」として、Simula が持っていた機能プラスインタフェースの範囲を取り上げる。例題は Java 言語による。

4.1 クラスによるオブジェクト定義

□ オブジェクト: さまざまな「種類」がある (はず)。例: 「乗用車」「トラック」「バス」

□ 1 つの種類のオブジェクト: 複数ある (はず)。例: 「私の車」「実家の車」

□ 「種類」を「クラス」(と呼ばれる単位) で記述し、「クラス」を雛型にインスタンス (個々のオブジェクト) を 1 つ以上生成

□ クラスに規定されるもの

- インスタンス変数 (状態変数とも呼ぶ) →各インスタンスの「状態」ないし「固有の値」を保持
- メソッド (C++用語では「メンバ関数」) →各インスタンスに付随する手続き。この手続きの中では、インスタンス変数の読み書きが可能。

□ Simulaの用語ではインスタンスは「永続的なブロック」、インスタンス変数は「ブロックの実行が終わっても値が消滅しないようなブロックローカル変数」と位置付けていた。しかし意味的には上記のように、現在のオブジェクト指向言語と同じ。

```
public class Sample21 {
  public static void main(String[] args) {
    Car c1 = new Car("My Car", 50);
    Car c2 = new Car("Father's Car", 80);
    c1.showInfo(); c2.showInfo();
    c1.changeSpeed(-20);
    c1.showInfo();
  }
}

class Car {
  String name;
  double speed;
  public Car(String n, double s) {
    name = n; speed = s;
  }
  public void changeSpeed(double d) {
    speed += d;
  }
  public void showInfo() {
    System.out.println("Car: " + name +
                       " speed: " + speed);
  }
}
```

```
% javac Sample21.java
% java Sample21
Car: My Car speed: 50.0
Car: Father's Car speed: 80.0
Car: My Car speed: 30.0
%
```

4.2 クラスの実現方法

□ クラスの実現はごく簡単→レコード型だと思って変数の割り当て等を計算すればよい。

- インスタンスの生成時にその大きさの領域をヒープから割り当てる
- 必要なら初期設定を行う

□ 変数の読み書きは領域先頭からの固定オフセットに対して行えばよい

□ 動的な言語では変数の位置も探索する場合がある

- 多くの変数があり、一部にしか値を設定しない場合には有利かも
- 多重継承の場合にも有利 (後述)

□ 通常、動的分配のための手当てが必要 (後述)

- 動的分配を使わないならデータ領域だけでよい (C++など)

4.3 カプセル化

□ インスタンス変数の値は、そのインスタンスが持っているメソッドの中からしか読み書きできない→カプセル化

- カプセル化によってインスタンス変数群に決まった制約を持たせ続けることができ、外部からそれを破壊されないことが言語仕様上保障される

□ たとえば: 「v = 複雑な計算 (x, y)」→ v の値を保存しておけば計算効率がよくなる→ただし x や y の値が変化したら必ず再計算

- 「x や y を変更したら必ず再計算」を保証することができるか?

```
changeX(...) { x = ○; recalcV(); }
changeY(...) { y = ○; recalcV(); }
useV() { return v; }
relalcV() { v = 複雑な計算 (x, y); }
```

□ 約束を変更したときにも変更が及ぶ範囲が限定される

- 「v が参照されないのに再計算は無駄」→この無駄を省けるか?

```
changeX(...) { x = ○; vChanged = true; }
changeY(...) { y = ○; vChanged = true; }
useV() { if(vChanged) {
           recalcV(); vChanged = false; }
         return v; }
```

□ その後のオブジェクト指向言語では、外部からインスタンス変数をアクセスできる「ようにも」指定可能に…(あまりよくないと思う)

4.4 カプセル化の実現

□ カプセル化は単なる「スコープの問題」だからコンパイル時のみで処理

- C や C++のように「別の型だと強制的に見なすキャスト (reinterpret cast)」があると問題
- 動的な (弱い型の) 言語では実行時にクラス情報を参照して検査

□ インスタンスを「その場に」取るような言語ではカプセル化はしてもコンパイラには利用するクラスの中が見えないとまずい。例:C++ (他にも Ada や Eiffel など)

```
class Human {
  private:  ←以下は見ちゃだめ
    char *name; float height, weight;
  public:   ←以下は見ていい
    char *getName();
    float getWeight();
    ...
}
```

□ こういう情報がヘッダファイルに入っている。

- つまり「見ていい」「見ちゃだめ」を区別…とはいってもそこに書いてあるという嫌らしさ
- さらに、後で変数を増やすとコンパイルし直しになる。

4.5 インタフェース

□ 先に出て来た「見ていい部分」だけを定義として記述したもの→オブジェクトの「外側から見える側面」→インタフェース (界面)

- インタフェースを独立した記述対象とした言語→普及したものでは Java が最初

□ 例: Figure(図) というインタフェースを考える

- 図は「画面に描ける」「X 座標/Y 座標を持ち、変更できる」

```
interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public int getX();
  public int getY();
}
```

□ このような「メソッドの名前、引数/返値の型の情報を合わせたもの」を「シグニチャ(signature)」と呼ぶ。

- シグニチャ情報があればコンパイル時に型検査できる

□ 上のインタフェースを使ったアプレット

□ アプレットは「円を 2 つ描く」もの

□ 「円」はクラスで実現

□ 「円」は Figure インタフェースに従う

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

public class Sample22 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Figure f2 = new Circle(80, 90, 50, Color.blue);
  public void paint(Graphics g) {
    f1.moveTo(f1.getX()+3, f1.getY()+2);
    f2.moveTo(f2.getX()-1, f2.getY()+1);
    f1.draw(g); f2.draw(g);
    try { Thread.sleep(500); repaint();
    } catch(Exception e) { }
  }
}
```

```
interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public int getX();
  public int getY();
}

class Circle implements Figure {
  int gx, gy, rad; Color col;
  public Circle(int x, int y, int r, Color c) {
    gx = x; gy = y; rad = r; col = c;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(gx-rad, gy-rad, rad*2, rad*2);
  }
  public void moveTo(int x, int y) {
    gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}
```

□ ちなみにこれを動かすために必要なのは…

- Sample22.java を打ち込み、コンパイルする
- 「javac -target 1.1 Sample22.java」←MS VM、Netscape 4 等で動かす場合は jdk1.1 互換を指定すること
- その同じディレクトリに以下の Sample22.html を置き、ブラウザまたは appletviewer(JDK に付属してくるアプレット表示ツール) でこの HTML ファイルを開く

```
<html><head><title>sample</title></head><body>
<applet code="Sample22.class" width="300" height="200">
</applet></body></html>
```

4.6 動的分配

□ 変数 x にオブジェクトが格納されているとして、x.m(...) はメソッド m を呼び出す。

□ 変数 x がクラス A のインスタンスであれば、クラス A で定義されているメソッド m が呼ばれる。クラス B のインスタンスであれば、クラス B で定義されているメソッド m が呼ばれる。→どのメソッド m であるかは、実行時に x に格納されているオブジェクトのクラスに応じて動的に定まる→動的分配 (dynamic dispatch)

- 「円の『描く』、矩形の『描く』、線分の『描く』はすべて実装としては別のものだが、機能としては同じに扱える」→ 1 つのコードで区別なく記述できるようにしたもの
- 動的分配がないと、「if 円 then 円.描く () elif 矩形 then 矩形.描く () else ... 」となってしまう→コードがごちゃごちゃ、プログラマが一時に

考えることが増える。これと対比すると、動的分配は極めて強力な機能だと言える

□ 先のアプレットに「正方形」を増やした

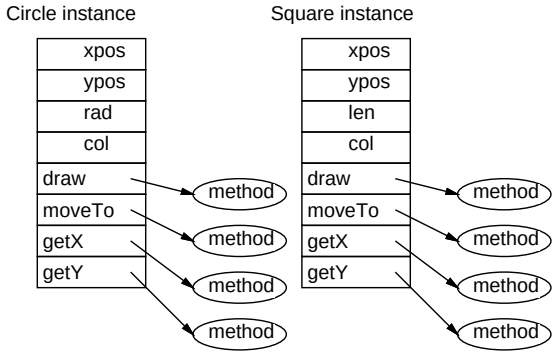
```
import java.applet.*;
import java.awt.*;

public class Sample23 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Figure f2 = new Circle(80, 90, 50, Color.blue);
  Figure f3 = new Square(120, 40, 40, Color.green);
  public void paint(Graphics g) {
    f1.moveTo(f1.getX()+3, f1.getY()+2);
    f2.moveTo(f2.getX()-1, f2.getY()+1);
    f3.moveTo(f3.getX()+2, f3.getY()+3);
    f1.draw(g); f2.draw(g); f3.draw(g);
    try { Thread.sleep(500); repaint();
    } catch(Exception e) { }
  }
}

interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public int getX();
  public int getY();
}

class Circle implements Figure {
  int gx, gy, rad; Color col;
  public Circle(int x, int y, int r, Color c) {
    gx = x; gy = y; rad = r; col = c;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(gx-rad, gy-rad, rad*2, rad*2);
  }
  public void moveTo(int x, int y) {
   gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}

class Square implements Figure {
  int gx, gy, len; Color col;
  public Square(int x, int y, int l, Color c) {
    gx = x; gy = y; len = l/2; col = c;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect(gx-len, gy-len, len*2, len*2);
  }
  public void moveTo(int x, int y) {
    gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}
```

□ 強い型の言語の場合、変数 x に対してメソッド m が使えるかどうかは型検査でチェックされる→変数 x に実行時に何が入れられるかを規定しておく必要 (先の例で出て来たインタフェースなど。型については後でとりあげる)

4.7 動的分配の実現

□ 最も原理的には…

- メソッドへのポインタの表を各オブジェクトに持たせる
- この表を検索してメソッドを見つけてから呼び出す

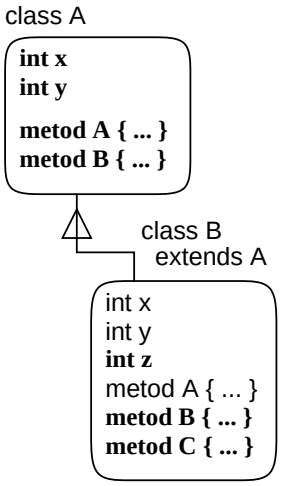


□ 実際にはいくつかの点で最適化

- 同じクラスのオブジェクトならメソッドは同じ→メソッド表はクラスごとに 1 つだけ作成し、各オブジェクトは自分が属するクラスを表す情報 (クラスオブジェクトへのポインタ) を持つ
- 一定の条件が整えば表を探索しなくても「表の何番目か」はコンパイル時に分かる
- 表の探索が必要な場合も、一度探索した結果をキャッシュしておくと次回はそれが再利用できることが多い

4.8 継承

□ 継承とは→あるクラスから別のクラスに定義を「引き継ぐ」こと



- 典型的には、インスタンス変数定義とメソッド定義 (実現の継承)
- 追加や差し替え (オーバライド) も可能

- しかし厳密な「継承の定義」をしはじめると難しい (後述)

□ 継承も動的分配の対象になる (現在のところ普及しているすべての言語において、親クラスはインタフェースの機能を兼ねている)

□ 継承は何が嬉しいか?

- 類似したクラス群を少ない記述で作成できる、定義の共有
- 新しいクラスを作るとき、違うところだけ書けばよい→差分プログラミング
- 共通の親を持つクラスのオブジェクトを総称的に扱える (なぜなら同じ変数群、同じメソッド群を持つから)(ただしそのためには動的分配も必要)
- 強い型のオブジェクト指向言語では、子クラスの値は親クラスの型に互換→型の問題 (後述)

□ たとえば先の例で Circle と Square の違うところは「ほんの少し」→その「差分」だけ書いてすませることもできる。

```
class Circle implements Figure {
  int gx, gy, rad; Color col;
  public Circle(int x, int y, int r, Color c) {
    gx = x; gy = y; rad = r; col = c;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(gx-rad, gy-rad, rad*2, rad*2);
  }
  public void moveTo(int x, int y) {
    gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}

class Square extends Circle {
  int len;
  public Square(int x, int y, int l, Color c) {
    super(x, y, 0, c); len = l/2;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect(gx-len, gy-len, len*2, len*2);
  }
}
```

□ この「super(...)」とは? → 親クラスのコンストラクタを呼ぶ。

- コンストラクタを呼ばないと初期設定が行なわれないので危ないから、呼ばないと許されないことになっている。
- 引数なしのコンストラクタがあれば、自動でそれを呼んでくれる
- クラスにコンストラクタを書かなければ、引数なしのコンストラクタが自動的に作られる
- →全体として、初期設定を必ず通る方向でチェックされる

□ それはそうと、上のような「差分プログラミング」はどう思う?

□ Java アプレットも継承を使った実例になっている

- アプレットは java.applet.Applet クラスからメソッド群を継承
- 自分の領域を再描画するときはメソッド paint() を呼ぶ
- Applet のサブクラスで paint() を差し替えることで独自の画面を作成

□ このように「ある決まった部分だけオーバライドにより差し替えて使えるような構造」→「アプリケーションフレームワーク」

- 実際にはもっと大きいプログラムで使われる (例: MFC)

4.9 抽象クラス

□ 抽象クラス: 機能の一部を子クラスの実装に任せるようなクラス

- その「子クラスに任されている」メソッドを抽象メソッドという
- Smalltalk では、抽象メソッドは例外を投げることで示す

□ Java では、抽象クラス/抽象メソッドを宣言→コンパイラが検査 (冒頭に修飾子「abstract」) をつける

□ 先の例題で「円と長方形の共通部分」を SimpleFigure という抽象クラスにくくり出してみる

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

public class Sample25 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Figure f2 = new Line(80, 90, 50, 70, Color.blue);
  Figure f3 = new Square(120, 40, 40, Color.green);
  public void paint(Graphics g) {
    f1.moveTo(f1.getX()+3, f1.getY()+2);
    f2.moveTo(f2.getX()-1, f2.getY()+1);
    f3.moveTo(f3.getX()+2, f3.getY()+3);
    f1.draw(g); f2.draw(g); f3.draw(g);
    try { Thread.sleep(500); repaint();
    } catch(Exception e) { }
  }
}
```

```
interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public int getX();
  public int getY();
}

abstract class SimpleFigure implements Figure {
  int gx, gy; Color col;
  public SimpleFigure(int x, int y, Color c) {
    gx = x; gy = y; col = c;
  }
  public abstract void draw(Graphics g);
  public void moveTo(int x, int y) {
    gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}

class Circle extends SimpleFigure {
  int rad;
  public Circle(int x, int y, int r, Color c) {
    super(x, y, c); rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(gx-rad, gy-rad, rad*2, rad*2);
  }
}

class Square extends SimpleFigure {
  int len;
  public Square(int x, int y, int l, Color c) {
    super(x, y, c); len = l/2;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect(gx-len, gy-len, len*2, len*2);
  }
}

class Line extends SimpleFigure {
  int dx, dy;
  public Line(int x1, int y1, int x2, int y2, Color c) {
    super(x1, y1, c); dx = x2-x1; dy = y2-y1;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.drawLine(gx, gy, gx+dx, gy+dy);
  }
}
```

4.10 クラス階層の設計

□ (抽象クラスや継承を含めた) よいクラス階層のデザイン→なかなか難しい重要な部分

□ 指針としては「抽象的なもの (上位概念) →親クラス」と言われているが…

□ 「円」と「楕円」はどっちが親クラス? またはどちらでもない?

- 円は楕円の特殊な場合である
- 円よりも楕円の方が (長径と短径があるので) 機能は多い

□ 「円」が使われるところすべてに「楕円」をあてはめてよいか? それが YES でないようなアプリケーションなら、独立したクラスにした方がよい。

- YES であれば、「円」に「縦横比率」を追加したものが楕円、という位置付けで楕円をサブクラスにすることはあっていいかも
- 結局、アプリケーション (やライブラリの使用想定場面) 次第

4.11 本節のまとめ

□ オブジェクト指向言語の基本部分→ Simula にほとんどある (一部ないものもある)

- クラス、インスタンス
- コンストラクタ→初期設定
- インタフェース→オブジェクトの「切り口」「使い方」
- 動的分配→オブジェクト指向言語の特に重要な機能
- 継承→差分プログラミング、フレームワーク、(インタフェースを兼ねる)
- 抽象クラス→クラス階層の構造化

□ ここまでで課題の 1 番目はできるように配慮しました。

5 Smalltalk: 中興の祖

□ Smalltalk システム: Alto システム上の言語処理系+実行環境

- Alto: ゼロックス社の Palo Alto 研究所で開発された「世界最初の」「パーソナルワークステーション」
- Smalltalk にはいくつかバージョンがあるが、1980 年の Smalltalk-80 が一応完成された版→その後も少しずつ改良
- Alan Key らが Dynabook 構想の実現の一歩として開発した
- ビットマップディスプレイ、対話的グラフィクス、サウンド、ウィンドウシステムなど、当時の水準から見ると極めて先進的なシステム

□ オブジェクト指向によるプログラミングの容易さがあってはじめて可能だった、とされている

□ 「中興の祖」という意味→ Simula にはじまるオブジェクト指向言語について、世の中に再認識させた

- プログラミング言語屋にとっては「センセーション」だった

□ Smalltalk の「見ため」の多くは Apple の Lisa → Macintosh に引き継がれた (Smalltak 言語は引き継がれなかった)

□ Smalltalk 言語は製品としてずっと存在し続けたが開発言語としてはマイナーであり続けた。有償だったし。

□ 1997 年、オリジナル Smalltalk の開発者たちが再結集して開発したフリーの処理系 Squeak が公開に (http://www.squeak.org/)

5.1 「純粋な」オブジェクト指向言語

□ 「純粋」という意味→すべてがオブジェクトである。たとえば整数や文字や論理値も (C++、Java 等ではこれらは基本型でありオブジェクトではない)

- そのため、極めて統一的な言語仕様とできる (すべてのもののふるまいはサブクラスを作って改良可能)
- たとえば「整数」のふるまいを変えたものも作れる
- ただしリテラルがもとの Integer クラスのものなので…
- コンパイラを変更してしまえば変えられる

5.2 特徴的な構文

□ すべては「メッセージ送信式」(と変数代入)

- キーワードセレクタ型:

```
オブジェクト セレクタ.
valu ← aPoint x.
オブジェクト セレクタ: 引数 セレクタ: 引数 ….
anArray at: 10 put: x.
```

- 演算子セレクタ型:

```
オブジェクト 演算子 オブジェクト.
x ← x + 1.
```

□ 「メッセージ送信」とは要するにメソッド呼び出しのこと

- オブジェクトが指定されたセレクタに対応するメソッドを持たないときは、messagenotunderstood というセレクタのメッセージに変換されて送り直される (このメソッドは Object クラスで定義されている) →自前でエラー処理したければこれをオーバライド
- 並列性や分散性は Smalltalk にはない (注: スレッドはあるが並列実行ではない)。この面で拡張を行う研究は多数あり → Concurrent Smalltak、その他

5.3 コードブロックの多用

□ コードブロック: コードの断片だが、それ自身オブジェクト。他の言語でいえば「クロージャ」に相当する

- メッセージ valu(引数つきの場合は「value: 引数」等) を送ると、そのコード内容を実行して return 文で指定した値を返す

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

- ブロックはブロックの周囲の環境をアクセスできる (だからクロージャ)
- 一方で、副作用だらけになるという問題点も

□ Java では、コードブロックは無いがその代り「内部クラス」や「無名の内部クラス」が使える (後述)

5.4 制御構造

□ 制御構造もブロックとメソッドで構成

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

- なぜ上は「(...)」で下は「[...]」なのか分かりますか?

□ そのほか「このような値が見つかるまで探す」といった指定にもブロックを利用→ Lisp 系の言語に近い (記号型もある)

5.5 先進的なプログラミング環境

□ ウィンドウシステムがほとんど普及していない時期からウィンドウ環境だった

□ クラスブラウザ、バックトレーサ、デバッガなどが組み込まれた統合プログラミング環境だった

- ソースを追加/修正すると環境全体が変化していしまう→環境全体のダンプを取って保存 (もちろんソース単独でも保存とロードはできたが)
- 全体的に言語、環境とも Lispっぽいと言える。cf. InterLisp-D

5.6 MVC フレームワーク

□ 画面に見える「もの」(ウィンドウの内容や部品) を M/V/C に分けて構築

- Model: 「もの」の状態を表すオブジェクト。たとえばスライドレバーであれば「現在の値」

- View: 「もの」の状態を表示するオブジェクト。たとえばスライドレバーであれば、「レバーの絵」や「数値表示窓」
- Controller: 「もの」を操作するための動作を提供するオブジェクト。たとえばスライドレバーであれば「レバーをドラグする」「プラス/マイナス押しボタン」。

□ 1 つのモデルに対してビュー、コントローラは複数あってよい (上の例)

□ その後の多くのグラフィカルなシステムにおいて MVC フレームワークが採用された

- View と Controller を分離する必要はどれくらいあるか? Java 2(Swing) などではこれらを一体化した delegate というものを使用
- モデルを分離する、という考え方はいずれにせよとても有効 (後述)

5.7 本節のまとめ

□ Smalltalk → オブジェクト指向の強力さを世に知らしめた

- 「純粋な」オブジェクト指向言語 (整数等もオブジェクト)
- コードの集まり (ブロック) もオブジェクト
- 制御構造はブロック等のメソッド

□ MVC フレームワーク→パターンの元祖

6 Lisp 系のオブジェクト指向言語

□ Smalltalk は最初から Lisp によく似た側面を備えていた

- そのため、Lisp 屋は Lisp にオブジェクト指向を導入することで Smalltalk のようなよい言語/環境を入手できるのではと考えた
- 実際、多くの Lisp ベースのオブジェクト指向言語が作られた
- その際、Smalltalk や Simula になかった新しい概念も多く考案された

□ 現在でも標準として残っているのは CLOS(Common Lisp Object System) →ただし今後の Lisp はどれもオブジェクト指向機能は持つようになる (CLOS 方式かどうかは分からないが)

6.1 Flavors

□ Zetalisp (Lisp Machine System で採用した Lisp の方言) 上のオブジェクト指向機能。クラスのことを flavor と呼ぶ

- (defflavor フレーバ名 各種情報…) で「クラス」を定義
- flavor のインスタンス→オブジェクト
- (send オブジェクト セレクタ 引数…) → メッセージ送信

□ ここまでのところは Smalltalk と本質的におなじ

6.2 多重継承

□ Flavors による重要な拡張の 1 つ。flavor には複数の親 flavor が指定できる→多重継承

- 多重継承では小クラスは親クラスすべてからインスタンス変数、メソッド群を引き継ぐ→「混ぜる」ことによる干渉もあり使い方は難しい

□ 実際には、「通常の (インスタンスを作る) クラス」と、「他のクラスにまぜて機能を追加するクラス」(mixin クラス) を区別して使い分けることが通例

- 例: ウィンドウクラスに対し、「窓枠をつける mixin クラス」などを混ぜて機能の増えたウィンドウを作って行く
- このような操作を mixin 操作と呼ぶ

6.3 メソッド結合

□ Flavors で提案されたもう 1 つの重要な拡張。

- Smalltalk では子クラスのメソッドは親クラスの同名メソッドを置き換え→親クラスのメソッドの動作「も」利用したい場合は「super セレクタ …」により明示的に呼び出し
- 多重継承では親が複数あるから上の方法ではいまいち
- C++では「どの親の同名メソッド」という形で呼べるが、この方法で十分かどうかは?

□ Flavors では、通常のメソッド (primary) のほかに、deamon メソッド (before daemon、after daemon) がある (実際にはもっとあるがこれらが主に使われる)

□ 多重継承とメソッドのオーバライドがある状態では…、次の順でメソッド群が呼ばれる

- まず、before daemon が親クラスから子クラスへの順で呼ばれる

- primary method はこれまで通りのオーバライドなので一番最近に定義された子クラスのものだけが呼ばれる
- 最後に after daemon が子クラスから親クラスへの順で呼ばれる

□ 何のためにこうなっている? →before daemon は「前しまつ」、after daemon は「後しまつ」を行い、それらは順番に結合されて各レベルのクラスの仕事を実行する

□ 使いこなせば便利なのかも知れないが、やっぱり難しい (と思う)

6.4 CLOS とマルチメソッド方式

□ CLOS (Common Lisp Object System) → CommonLisp の言語仕様のうちの、オブジェクト指向機能部分をいう (後から追加されたもの)

□ 最大の変化→汎用関数 (generic function) に基づくメソッド呼び出し

- Flavors: (send オブジェクト セレクタ …) →「どのメソッドか」は「オブジェクト」と「セレクタ」で決まっていた
- 最初の引数 (レシーバ) のみを重視しすぎ? →「すべての引数がメソッドの決定に関与する」

```
(defclass X ....)
(defclass Y ....)
(defmethod method1 ((a X) (b Y)) ... *1 )
(defmethod method1 ((a X) (b X)) ... *2 )
...
(method1 anX anY) → *1 が呼ばれる
(method1 anX anX) → *2 が呼ばれる
```

- 「メソッドがクラスに付属していない」「構文的には普通の関数みたいな見え方」→特徴的 (好みも分かれる)
- 多重継承やメソッド結合は Flavors 以来引き継がれている

6.5 本節のまとめ

□ Lisp 系のオブジェクト指向言語→メジャーではないが様々な試み

- 多重継承
- メソッド結合
- メタオブジェクトプロトコル (MOP)
- マルチメソッド

7 オブジェクト指向と型

□ Simula は強い型の言語だったが、その後、Smalltalk、Flavors、等はすべて弱い型の言語

□ C 言語にオブジェクト指向を→やはり「オブジェクト型」はすべて一緒、というタイプが多かった (例: Objective-C) →強い型ではない

□ 10 年以上たって、ようやく「強い型のオブジェクト指向言語」が当たり前になった (C++が代表的)

7.1 強い型の概念と利点/弱点

□ 強い型とは? → コンパイル時にすべての式や変数の型が定まっている

- 利点: コンパイル時検査、設計の手段
- 弱点: 繁雑、めんどくさい???

□ 中庸もある: 例 CommonLisp → 型はなくてもいいけど、指定してもいい。指定すると効率がよいかも/コンパイル時検査が可能

7.2 弱い型のオブジェクト指向言語

□ 変数にも式にもコンパイル時の型はない

□ しかし、実行時には型 (==クラス) がある

- 「anObject message.」→「OK である」か「そのメソッドはない!」かどちらか。
- 「そのメソッドはない」がどこで起こり得るかを予め知る方法はない→製品としてソフトを作るときには弱点となり得る
- 「OK である」ならよいのか? → たまたまそういう名前のセレクタが利用可能、だったらもっとたちが悪い?

7.3 強い型のオブジェクト指向言語

□ Simula が既にそうであった

- 「型」と「クラス」は同じものとみなす (ちょっとは違うが)
- 「ある型の変数/式」は実行時に「そのサブクラスの値も持つことができる」

□ この規則は通常の「強い型の言語」からはだいぶ離れている

```
Pascal ... x:integer := 1;
    x の型:integer、1 の型:integer
Java ... o:Object := new Integer(1);
    o の型:Object、式の型: Integer
                (Object のサブクラス)
```

- Object 型には任意のオブジェクトが入れられてしまう
- 代入の左辺と右辺の型は同じでなく包含関係→複雑さの原因
- ただし、メソッドを呼ぶときは型が合わないとだめ

```
i:int := o.intValue(); ... ×
i:int := ((Integer)o).intValue(); ... ○
```

- このキャストは「実行時の型検査を伴うキャスト」であって C のキャストとは違う (もともとのオブジェクトが Integer ないしそのサブクラスでなければ例外が発生)

□ ある型に属するかどうかを判定→ instanceof 演算子

```
if(o instanceof Integer) ...
```

□ もっと自由に型の情報そのものを扱う→自己反映機能 (後述)

7.4 インタフェースと型

□ 単一継承でサブクラス階層だけだと型の包含関係は非常にシンプル

□ 多重継承があると 1 つのインスタンスが複数の親の型に含まれることになるが、まだ包含関係は明らか

□ インタフェースではそれを実装しているすべての型を「くくる」という点で不規則な構造が作れてしまう (ただしループは許さない)

- インタフェース自体にも包含関係があるのでさらに面倒なことに…

7.5 本節のまとめ

□ 強い型はコンパイラによる誤り検査や設計の手段として有用

□ 「お仕事の言語」では強い型のものが主流

□ オブジェクト指向以前の「強い型」とオブジェクト指向の「強い型」はやや違っている (型の包含関係)

- より広い型の変数に狭い型の値 (インスタンスを入れられる)
- 広い型から狭い型に戻すことができる (実行時チェックが必要)

8 継承と委譲

□ 継承 (inheritance) → Simula、Smalltalk 以来の「由緒正しい」やりかた

- 使っているうちに、色々問題もあることが分かって来た

□ 委譲 (delegation) →継承の代替として使えるメカニズム

- 複数オブジェクトの組合せ (composition) と相性がよい→継承よりも委譲を使うことが増えている

8.1 継承の意味づけ

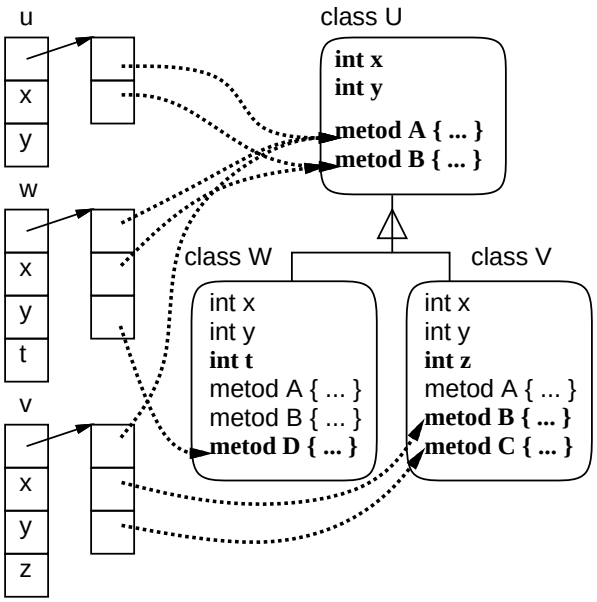
□ 継承がやってることは何かというと…

- インスタンス変数とメソッドを引き継ぐ
- その結果として、呼べるメソッドの集合や外から見たオブジェクトの振る舞いを引き継ぐ
- たとえば B が A のサブクラスであれば、「A のインスタンス」の代りに「B のインスタンス」を与えてもそのまま動く (建前としては)
- 動的分配の前提として、「A 型の変数に A のサブクラスがいろいろ入れられる」ということが必要

□ しかしよく考えると、これは「実装の継承」が先にあり、その結果として「たまたまどれでも同じように取り扱える」ようになっているだけとも思える。この「たまたま」は気持ち悪い

8.2 継承の実装

□ ごく素直な継承の実装方法→オブジェクトの構造が重要。インスタンス変数を定義された順に並べておく→子クラスでインスタンス変数を追加した場合は後ろにつけ加えていく



- この方法であれば、クラス A のどのサブクラスでも A までで定義されている変数のオフセットは同一 →オフセットで直接アクセス可能
- メソッドのコードがそのまま利用可能
- 分かりやすく、効率がよい。ただし多重継承に対応できない

8.3 メソッド探索

□ メソッド呼び出しで実際にどのメソッドが動くかはオブジェクトのクラスによって変化→メソッド探索

- Smalltalk では、最初にそういう呼び出しがあったときに探索を行い、その情報をキャッシュに保持。クラス構造が変化したときはキャッシュをご破算にしてやりなおす。動的にクラスが変化する環境ならでは
- C++、Java などのコンパイルする言語では、分岐表 (メソッドテーブル) を作ってそれに基づいて分岐すればよい。分岐表そのもののスロットも変数と同様にして管理可能
- インタフェースの場合は 1 つのクラスがさまざまなインタフェースを実装しているので面倒。Java では呼び出し時に探索しているが、工夫次第では「表引き」にもできる

8.4 多重継承の実装

□ 多重継承: 2 つの親から継承すること

- 問題: 複数の親が共通の親クラスを持っていたらどうするか?

□ C++の多重継承では 2 つの方式がサポートされているので大変

- 共通の親があったとき 2 重にコピーする→その場所は親クラスとしてそのまま扱える (ただしポインタ逆変換の問題がある)
- 共通の親は統合する→「どこに親が埋まっているか」のポインタをそれぞれのインスタンスに埋めることで対応

□ 「共通の親は統合」でもう 1 つの素直なアプローチ: 最初に名前で探索し、その場所を覚える

8.5 継承の問題点

□ 継承にはさまざまな問題点があった→何が問題か、分かります?

□ 実装と界面の混同

- 継承は実装を引き継ぐことで、インタフェースを共通にすること、多相性 (polymorphism、動的分配) を提供することはまた別の問題だが混同されがち
- Java のようにインタフェースを別にすることが必要 (ただし Java でも継承すると多相性がついてくるので中途半端)

□ カプセル化の破壊

- サブクラスを作ると、その中では親クラスの変数が自由にいじれてしまう→カプセル化の破壊
- C++、Java など→サブクラスでいじれる変数、いじれない変数を区別可能に (private →サブクラスでもいじれない変数、protected →サブクラスでもいじれる変数) →それが問題の解決になってるのかどうか??

□ 機能追加の制約

- 単一継承の場合、1 つずつしか機能を追加して行けない→たとえば図形に「動く機能」「大きさが変わる機能」を追加したいとすると、それぞれの機能のあるなしごとにクラスを分けることに→機能が N 種類あったら 2 の N 乗必要→組合せ爆発
- 多重継承があれば「動く機能」「大きさが変わる機能」などをそれぞれ別のクラスにして「混ぜる」ことで必要なクラスが作れる。ただし混ぜたものの干渉が心配

□ 継承は静的

- 実行時に機能を追加したり外したりといったことはできない。
- しかし場合によっては実行時に機能の調整を行ないたい。たとえば「動かない円」を作っておいたがそれを途中で動くようにしたい等。

□ 例: GUI 部品と動作

- たとえば「ボタン部品」を考えてみる。ボタンには「ボタンを押した時の動作」があるはず。その動作はアプリケーション固有
- しかし汎用の Button クラスには当然、アプリケーション固有の動作は入っていない
- ではどうするか? → 継承を使って、「動作」メソッドをオーバライドして、そのメソッド中でアプリケーション固有の動作を行なわせる
- Java は JDK 1.0.x ではそうしていたが、JDK 1.1 からはやめている→動作 1 つごとにサブクラスを作るのが煩雑、複数の場所 (例: メニュー項目やキーボードショートカット) から同じ動作を起動するときの共有ができない、実行時に割り当てを変更できない

8.6 委譲 (delegation)

□ 継承は「親のインスタンス変数やコードを取り込んで来て自分の一部として実行させる」→カプセル化が壊れる等の問題

□ 別オブジェクトの機能が必要なら、それを別のインスタンスとして持っていて、これを普通に呼び出すことでも利用可能→委譲の考え方

- 簡単に言えば、自分で実装しないメッセージを「たらいまわし」にする

□ 委譲のさまざまな利点

- カプセル化が壊れない
- 委譲先を実行時に動的に切り替えることができる
- 多重継承の実装が容易 (多重継承の実装として、一部に委譲を使うことも)

□ 例: GUI 部品の「ボタン」に動作をつける場合

- JDK 1.1 以降では各ボタンは次のメソッドを持つ

```
public void addActionListener(ActionListener l)
```

- ActionListener は次のようなインタフェース

```
public class ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

- 各ボタンの動作は ActionListener インタフェースを実装するクラスのインスタンス (アダプタオブジェクト) のメソッド actionPerformed() として用意
- ボタンは押されるとアダプタオブジェクトの上記メソッドを呼び出す

□ 例題: 図形を動かすボタンをつけてみる

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;  //← import 追加!

public class Sample26 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Button b1 = new Button("B1"); // ボタン
  Button b2 = new Button("B2"); // ボタン
  public void init() {  //←初期設定メソッド
    setLayout(null);    //←自動配置 off
    add(b1); b1.setBounds(10, 10, 60, 30); //配置
    add(b2); b2.setBounds(10, 50, 60, 30); //配置
    b1.addActionListener(new MyAdapter1(this, f1));
    b2.addActionListener(new MyAdapter2(this, f1));
  }  // ↑ボタンにアダプタを設定
  public void paint(Graphics g) { f1.draw(g); }
}

interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public int getX();
  public int getY();
}

abstract class SimpleFigure implements Figure {
  int gx, gy; Color col;
  public SimpleFigure(int x, int y, Color c) {
    gx = x; gy = y; col = c;
  }
  public abstract void draw(Graphics g);
  public void moveTo(int x, int y) {
    gx = x; gy = y;
  }
  public int getX() { return gx; }
  public int getY() { return gy; }
}

class Circle extends SimpleFigure {
  int rad;
  public Circle(int x, int y, int r, Color c) {
    super(x, y, c); rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(gx-rad, gy-rad, rad*2, rad*2);
  }
}

class MyAdapter1 implements ActionListener {
  Applet app; Figure fig;
  public MyAdapter1(Applet a, Figure f) {
    app = a; fig = f;
  }
  public void actionPerformed(ActionEvent e) {
    fig.moveTo(fig.getX()+5, fig.getY()+5);
    app.repaint();
  }
}

class MyAdapter2 implements ActionListener {
  Applet app; Figure fig;
  public MyAdapter2(Applet a, Figure f) {
    app = a; fig = f;
  }
  public void actionPerformed(ActionEvent e) {
    fig.moveTo(fig.getX()-5, fig.getY()-5);
```

```
    app.repaint();
  }
}
```

□ アダプタオブジェクトは「オブジェクト」なので、その中に「呼ばれた」時に必要なデータを持っておくことができる

□ 上の例では 2 つのアダプタクラスはほとんど一緒→ 1 つで済ませる方がスマート。だが、複数あってよいという例のつもりで 2 つ用意した

8.7 Self: プロトタイプ方式のオブジェクト指向言語

□ ここまでは「委譲」をコーディング上の手法として考えて来たが、言語機構として継承の代わりに委譲のみを使う言語も→ Self

□ クラスがなく、「ひな型」のオブジェクトを複数コピーすることでインスタンスを作る

- 委譲した場合でも「元のオブジェクト (プロトタイプ)」を覚えておいて、自分自身に対してメッセージを送った場合元から探す。これがないと次のようなメソッド (抽象メソッド) が使えない

```
moveRight | n |
  self turn 90.
  self forward n.
```

□ Self 言語は「プロトタイプ方式の」「委譲に基づく」オブジェクト指向言語が有用だという実証の意味が大。コードの性能も動的コンパイル (よく使う/高速な組合せだけをコンパイルしていく) などの技術により優れていた

□ もう 1 つの (極めて普及している) プロトタイプ方式のオブジェクト指向言語→ JavaScript(次回に取り上げる)

8.8 本節のまとめ

□ 継承はオブジェクト指向言語の大きな特徴として注目

- もっとも「継承のないオブジェクト指向言語」も可能だしあるが

□ 委譲 (たらいまわし) は最初は「単なる別のオブジェクトの呼び出し」だったが、次第に継承に代わる機構として認識

□ オブジェクト指向プログラミング全体が継承指向から委譲指向に変化

9 Java の入れ子クラスと内部クラス

□ 入れ子クラス (クラスの中にクラス) については既に説明したが、再度整理しておく

□ 入れ子クラスの特別な場合である「内部クラス」についても説明

9.1 入れ子クラス

□ クラスの中に別のクラスを書けるようにしたオブジェクト指向言語はまだあまり多くない。

- クラスはモジュールのようなもので、ある程度の完結性があるから、あるクラスの中に別のクラスを入れると言うことはあんまり必要がないと思っていた?

□ しかし Java ではクラスの中に「そのクラスだけが使う下請けクラス」が書ける→入れ子クラス

- これは、Java が「必要があればどんどんクラスを作って使う」方向だから← Smalltalk などもそういう文化だが、クラスだらけでごちゃごちゃになりがち
- Java ではパッケージ (階層的な名前を持つ) →クラス→入れ子クラスという 3 段階だからだいぶいい。入れ子クラスの中にさらにクラスを入れることもできる (まず見かけないが)

9.2 Java の内部クラス

□ インタフェースを使ってアダプタクラスを作るような場合:

- 小さいクラスが多数できてしまうので面倒→これに対しては入れ子クラスを使えばよい
- アダプタクラスが保持するデータはインスタンス変数として保持する必要→その宣言や初期設定等が結構面倒
- そこでどう考えたか…

□ 2 種類のメソッド

- static のついたメソッド (クラスメソッド) →インスタンス変数にはアクセスできない、独立した関数のようなもの
- static のつかないメソッド (インスタンスメソッド) →インスタンスに付随していて、インスタンス変数を読み書きでき、他のインスタンスメソッドをそのまま呼び出せる

□ これにならって、入れ子クラスも 2 種類にする!

- static のついた入れ子クラス→インスタンス変数にはアクセスできない、独立したクラスと同様
- static のつかない入れ子クラス→*外側の*クラスのインスタンス変数にアクセスでき、外側クラスのインスタンスメソッドをそのまま呼び出せる→つまり、外側クラスのインスタンスを「覚えて」いる→その代わり new で作り出せるのはインスタンスメソッドやコンストラクタの内側→「内部クラス」と名付けた
- さらに内部クラスの中からは、インスタンス変数だけでなく final 指定の引数や局所変数 (つまり値が変更されない変数) も参照できる←ということは必要なら「メソッドの途中」にも書けるわけ
- これらの機能のため、内部クラスを使うといちいちアダプタクラスにインスタンス変数を持たせなくてもよい場合が多い→記述が簡単

□ これを使って先の例を書き換えてみた

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

public class Sample27 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  public void init() {
    setLayout(null);
    add(b1); b1.setBounds(10, 10, 60, 30);
    add(b2); b2.setBounds(10, 50, 60, 30);
    b1.addActionListener(new MyAdapter1());
    b2.addActionListener(new MyAdapter2());
  }
  public void paint(Graphics g) { f1.draw(g); }

  class MyAdapter1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      f1.moveTo(f1.getX()+5, f1.getY()+5);
      repaint();
    }
  }
  class MyAdapter2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      f1.moveTo(f1.getX()-5, f1.getY()-5);
      repaint();
    }
  }
}
(以下同じにつき略)
```

□ 外側クラスのインスタンス変数に直接アクセス→そのためコンストラクタも変数も不要→さっきよりずっとコンパクトに

9.3 無名クラス

□ 上の例では MyAdapter1 等の名前を使う個所はそれぞれ 1 個所ずつしかない

□ 1 個所でしか参照しないようなアダプタクラスの名前をいちいち考えるのは嬉しくない

□ このような場合には、なおかつ「そのクラスが 1 つのクラスを extends しているか、または 1 つのインタフェースを implements しているだけであれば」名前を省略できる→無名クラス

□ 具体的には次のような形

```
public class X {
  ... new Y(); ... ←ここでだけ使用

  class Y implements I {
    内部クラスの定義
  }
}
```

□ このとき、使用しているところに Y の定義本体を直接埋め込んでしまうことで名前を書かなくする

```
public class X {
  ... new I() {
    内部クラスの定義
  } ...
}
```

□ 先の例を無名内部クラスを使うように直すと:

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

public class Sample28 extends Applet {
  Figure f1 = new Circle(50, 50, 30, Color.red);
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  public void init() {
    setLayout(null);
    add(b1); b1.setBounds(10, 10, 60, 30);
    add(b2); b2.setBounds(10, 50, 60, 30);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        f1.moveTo(f1.getX()+5, f1.getY()+5);
        repaint();
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        f1.moveTo(f1.getX()-5, f1.getY()-5);
        repaint();
      }
    });
  }
  public void paint(Graphics g) { f1.draw(g); }
}
(以下同じにつき略)
```

□ ボタン以外の GUI 部品を使った例も挙げておこう。

- Button →ボタン
- Label →表示ラベル
- TextField →入力欄
- Choice →選択メニュー

□ 選択メニューについては、選択肢は add() で別途追加する

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

public class Sample29 extends Applet {
  Label l0 = new Label("");
  Choice c0 = new Choice();
  Button b0 = new Button("Calc");
  TextField t0 = new TextField("");
  public void init() {
    setLayout(null);
    add(l0); l0.setBounds(10, 10, 280, 30);
    add(c0); c0.setBounds(10, 50, 60, 30);
    c0.add("F to C"); c0.add("C to F");
    add(b0); b0.setBounds(110, 50, 60, 30);
    add(t0); t0.setBounds(10, 90, 120, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          float x =
            (new Float(t0.getText())).floatValue();
          float y = (c0.getSelectedIndex() == 0) ?
            (5.0f/9.0f * (x - 32.0f)) :
            (9.0f/5.0f * x + 32.0f);
          l0.setText("Result: " + y);
        } catch(Exception ex) {
          l0.setText(ex.toString());
        }
      }
    });
  }
}
```

□ 記述が短く簡潔になるという点は便利だが、最初はちょっと分かりづらい

□ 従来の言語で「クロージャ」と呼ばれる機能を持つものがある

- クロージャ = 関数 + 環境
- Smalltalk のブロックも一種のクロージャ
- そのクラス版が Java の内部クラスだと考えられる

9.4 本節のまとめ

□ Java の入れ子クラス→クラスの中に下請けクラスが書ける

□ 内部クラス→外側インスタンスの情報や、外側メソッドの局所変数などに中から直接アクセスできる→クロージャ機能を実現したものと言える

- うまく使えば記述がコンパクトになるが、一方でどこで何をアクセスしているか分かりにくくなる恐れもある

10 自己反映機能

□ 自己反映 (reflection): 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

- 狭い意味では前者のみ (後者を reification と呼んで区別することも)

□ 何のためにそんなことをするのか?

- 例: デバッガ→実行系の内部状態を調べたり変更する必要
- 例: システムの拡張→「任意の手続き呼び出しを遠隔メッセージに変換」など
- 例: 拡張可能言語 (構文や意味づけ等)

10.1 3-Lisp: リフレクションの元祖

□ ベースレベルとメタレベルを区別

- ベースレベル: 通常の実行
- メタレベル: ベースレベルの実行系の情報 (バインディング、継続等) が見える
- メタレベルを変更→ベースレベルでの対応する状態変化が起こっている→これにより、言語セマンティクス (実行順序の制御等) が拡張可能

□ 3-Lisp のさらに特徴→「メタレベル」はさらに「メタメタレベル」によって制御可能→無限の reflective tower になっている

- 実装上は「遡られたところまで自動的にメタレベルを用意」

10.2 Smalltalk のクラスとメタクラス

□ Smalltalk では「クラス」もまたオブジェクト

- クラスに対してメッセージを送る→メソッドを追加したり修正したり等ができる (実行環境全体が Smalltalk で書かれているので当然といえば当然)

□ クラスオブジェクトは Metaclass というクラスのインスタンス。たとえばクラス Collection のクラスオブジェクトは Collection class(という式でアクセス)。Collection class は Metaclass というクラスのインスタンス

- Metaclass は各クラスオブジェクトを初期設定する機能をおもに提供→ Metaclass を修正すると、Smalltalk システム全体の動作が変化させられる

10.3 メタオブジェクトプロトコル (MOP)

□ 「オブジェクト」を統括する (ふるまいを定義する) オブジェクト→「メタオブジェクト」(例: クラスに対してはメタクラス)

□ メタオブジェクトが提供するサービス、API →メタオブジェクトプロトコル

□ 最近の多くの言語ではメタオブジェクトプロトコルを提供することで自己反映機能をさまざまに利用可能

- CLOS: メソッド呼び出しの意味づけなどを自由に変更可能
- OpenC++: コンパイル時 MOP →メタオブジェクトを定義すると、コンパイル時にメタオブジェクトがソースを変更した上でコンパイル→言語の意味づけが変化させられる

10.4 Java の自己反映機能

□ Java の自己反映機能→処理系そのもを変更する、という部分はない。

- 内部の状態をのぞく
- のぞいた情報を利用して、その場でメソッド呼び出し等を組み立てて実行させられる

□ 強い型の言語は「型が合わなければ扱えない」→リフレクションのような自由自在なことは表しにくい

- Java ではこれらをきちんと型を割り当てた上で可能にしている
- ある意味では、Lisp 等の「eval」(任意のプログラムを合成してその場で走らせる」を強い型の言語上で可能にしたといえる

□ 任意のオブジェクトは getClass() でその Class オブジェクトを取得できる

- または、static メソッド Class.forName("...") でも

□ Class オブジェクトはそのクラスに関する情報を取得するメソッドを持つ

- 例: getConstractors(), getMethods(), getMembers()

□ Constructor オブジェクトの newInstance() を呼ぶとオブジェクトが生成される

□ Method オブジェクトの invoke() を呼ぶとメソッドが実行できる

□ たとえば、任意のクラスを 1 つもってきてオブジェクトを生成しメソッドを呼ぶ (ただし引数はすべて空) というプログラム

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

public class Sample2a {
  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(); }
    }
  }
}
```

□ これをたとえば次のクラスに対して使ってみる…

```
public class Sample2aTest {
  int val;
  public Sample26Test() { val = 1; }
  public Sample26Test(int i) { val = i; }
  public Sample26Test add() {
    return new Sample26Test(val+1);
  }
  public Sample26Test sub() {
    return new Sample26Test(val-1);
  }
  public String toString() {
    return "Sample26Test("+val+")";
  }
}
```

□ なお、この方法で通常取れるのは public なものだけ (セキュリティ上の制約)

10.5 自己反映機能とコード生成ツール

□ 例: コンポーネントツールや GUI ビルダ

- 画面で直接部品を配置したり動かしてみたりしたい
- 動かせるためには「部品そのもの」を操作したい
- しかし、部品オブジェクトは部品ごとに「使い方」が異なる
- →自己反映機能を用いて調べつつ「呼び出せば」よい

□ Java による GUI 部品配置の例

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

public class Sample2b extends Applet {
  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 void init() {
    setLayout(null);
    add(l0); l0.setBounds(10, 10, 60, 30);
    add(t0); t0.setBounds(80, 10, 200, 30);
    add(b0); b0.setBounds(290, 10, 60, 30);
    add(b1); b1.setBounds(360, 10, 40, 30);
    add(l1); l1.setBounds(10, 40, 380, 30);
    add(c2); c2.setBounds(10, 70, 200, 30);
    add(l2); l2.setBounds(10, 110, 60, 30);
    add(t2); t2.setBounds(80, 110, 200, 30);
    add(b2); b2.setBounds(290, 110, 40, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          c0 = (Component)Class
            .forName(t0.getText()).newInstance();
          Sample27.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);
  }
}
```

10.6 本節のまとめ

□ 自己反映機能→「プログラムの中で、プログラムそのものを操作できる」→より柔軟なソフトウェア構成が可能になる

- その反面、プログラムが分かりにくくなる
- どこまで「静的」(コンパイル時)、どこから「動的」(実行時) の線引き→トレードオフを考えることが重要

□ ついでに： オブジェクト指向言語の諸側面のまとめ

- 非常に多数の機能がある、ということは分かったと思う
- それらすべてに「発明された理由」はもちろんある
- しかし「それらがすべてが今いるのか?」は別の問題
- 言語の多様な機能を使えば使うほど「難しく」もなる →「機能の増加」と「簡潔に/分かりやすく記述でき

る」のトレードオフについて常に考える (例: Java vs C++)

11 オブジェクト指向の利用技術

□ オブジェクト指向言語→これまでの言語にないさまざまな「道具だて」(例: サブクラス、動的分配、継承、委譲、インタフェース、…)

- それをどのように使うか→なかなか難しい問題

□ どのような方向での利用があるか?

11.1 オブジェクト指向にあった設計

□ 美しいプログラム構造 (何が美しい???) ←オブジェクト指向言語の特性に合ったプログラム設計←OOSE(オブジェクト指向分析、オブジェクト指向設計、UML、統一メソッド): 本講の範囲外

11.2 再利用

□ 再利用←書くよりは書かないで済ませた方が生産効率はよいに決まっている

- なぜオブジェクト指向で再利用? → クラス、オブジェクトといった単位は従来の「サブルーチン」より固まりが大きく、再利用に向いている
- 再利用するものは何? コード? 設計知識? → さまざまなレベルがある

11.3 クラスライブラリ

□ Smalltalk →充実したクラスライブラリが付属→クラスライブラリを熟知すれば生産性が高まる、というブームに

- 実際にやってみると、よいクラスライブラリの開発/クラスライブラリに熟知した人材の育成ともに簡単ではない

□ Smalltalk クラスライブラリでは差分プログラミング (子クラスで親クラスの機能を少しずつ拡張していく) を多用→これもよい手法だと一時思われていた

- しかしやってみると、よい差分プログラミングは難しい (クラス間の依存関係が大きくなりぐちゃぐちゃになりやすい)

□ 現在では、クラスライブラリはもちろん必要だが、整った機能を一式、分かりやすいインタフェースで提供するという当たり前の結論に

11.4 アプリケーションフレームワーク

□ 抽象メソッド: 親クラスで「自分自身へのメソッド呼び出し」を用いたメソッドを定義→子クラスでそれらのメソッドを具体的なものに差し替え

□ これをさらに発展させて、汎用的なアプリケーション全体の構造を予め定義しておく→その中のいくつかのクラスをサブクラス化してそこに各アプリケーション固有の部分を記述することでアプリケーションを完成させる

- 例: MFC、ET++、Choices、…
- うまく当てはまれば生産性は高まるが、何をどうサブクラス化するか、サブクラスはどのような規約に従う必要があるか、といったことを学ぶのが大変

□ たとえば、アプレットもアプリケーションフレームワークの 1 つ

- アプレットはクラス Applet のサブクラスとして作り、必要なメソッドをオーバライドする
- init() → 初期設定する
- paint() → 描画する
- start()、stop() → ページの表示開始/終了
- アプレットの場合は難しいカスタマイズはしないという感じ

11.5 コンポーネント

□ ここまでの再利用技術→基本的にクラス (群) が対象→プログラマ向け (クラスベースとも言う)

□ プログラミングをしない人に使える再利用技術→インスタンスベースの再利用

- インスタンスを生成し、そのプロパティ(属性、要するにインスタンス変数の値) をカスタマイズする
- カスタマイズしたインスタンス群をディスク等に保存しておき、それを取り出してそのまま動かす
- そのようなインスタンスを「コンポーネント」と呼んでいる。ソフトウェア開発のためのコンポーネント群→「コンポーネントウェア」、コンポーネントウェアに基づくソフトウェア開発→「部品組み立てプログラミング」

□ 代表的な成功例→ VisualBasic (部品: VBX、OCX… COM、DCOM の部品)

- ほかにも国産の IntelligentPad、Java ベースの JavaBeans などいろいろある
- しかし、部品とその配線だけでできるプログラムで十分なの?

11.6 デザインパターン

□ パターン: 「繰り返し現われるようなカタチ」

□ (ソフトウェアにおける) デザインパターン: オブジェクト指向ソフトウェア開発において、有効に使えるようなオブジェクト群の構成のパターン

- 1990 ころから、Peter Cord 他がはじめた。日本では「ガンマ本」が有名になっている。ガンマ本はよく使うパターンを集めた「パターンカタログ」になっている。
- なぜデザインパターン? → オブジェクトの接続関係のノウハウはかなり難しい (ちょっと思いつかないようなものもある) →それを蓄積しておいて流通させると、うまくはまったときに役立つ

□ 例: Command パターン

- メニューの選択、画面上のボタンなどはどれも「何らかの動作」を起動する→「動作をするオブジェクト」を用意して、それをメニューやボタンに結びつけていけばよい。

□ 例: Adapter パターン

- ボタンが押されたときに呼び出されるメソッドはある名前に決まっている。しかし実際に起きて欲しいことを実行するメソッドは別のメソッドである→「仲介するオブジェクト」を用意して、それが橋渡しをすればよい。

□ 例: Visitor パターン

- オブジェクトの階層構造で構造化グラフィクスとか複合文書のようなものを作ったとする。「印刷する」「表示する」「ファイルに保存する」「スペルチェックする」等それぞれの場合について、各クラスにそれ用のメソッドを作るのは面倒である → どうする???

□ Visitor パターンとは:

- 各オブジェクト側には「accept」というメソッドを 1 つだけ用意しておく。その引数として、「印刷用の Visitor」「表示用の Vsitor」などさまざまな Viritor オブジェクトをそのつど渡せばよい

□ 例: AbstractFactory パターン

- Windows でも Mac でも X11/Unix でも同じに動作する GUI アプリケーションを開発するには???

□ AbstractFactory パターンとは:

- Window、Button、Dialog などの汎用的なクラスを用意する
- そのサブクラスとして MacWindow、X11Window などそれぞれ用意する
- WinFactory という抽象クラスを用意し、メソッドとして makeWindow、makeButton 等を用意する
- MacWinFactory、X11WinFactory などの具象クラスでこれらをそれぞれ実装する
- アプリケーションの実行開始時に MacWinFactory などのインスタンスを作って WinFactory 型の変数に格納し、以後それを利用する

11.7 本節のまとめ

□ オブジェクト指向にはさまざまな「道具」が含まれている→その「道具」をどう使うか、についていろいろな工夫がある

□ しかしこれらもまだほんの一部? →これからより多くの「よりよい利用方法」が現われる (はず)

□ それらの利用方法を個別に「新しい」と思って受け入れるだけでは、流行に追われるだけ→必要なこと:

- その「新しい」技術がこれまで行われてきたさまざまなことの中にどのように位置づけられるのかを考える
- 結局、ソフトウェアの生産において「自分が必要とすることは何か」をまず考え、それに照らして必要なものを取捨選択する

12 第 2 回課題

□ 以下の課題から 1 つ以上 (4 を含む場合は 2 つ以上) を選択してプログラムを作成し、報告せよ。考察まできちんと書くこと。

□ (1) 「円」「正方形」などの図形が表示される例題のどれかを改良して、新しい図形を追加してみよ。また、「鉄アレイ型」を追加するものとして、その実装をどのように工夫するか検討し (例: 円と正方形のインスタンスを組み合わせる)、実際に実現してみよ。

□ (2) 「円」「正方形」などの図形が表示される例題では、図形の動きは外側から操作していた。しかしそれぞれの図形が自分固有の「速度」を持って動く方がオブジェクト指向らしい。そのような形にクラス群の機能を拡張してみよ。できれば「速度」などの情報は実際に動く図形にだけ付随していることが望ましい。(複数の実装方法を試してみられるとなおよい。)

□ (3) ボタンその他の GUI 部品を使って、「ボタンを押す度にさまざまな色や大きさの円ができ、位置を調整できる」プログラムを作れ。または「電卓」を作れ。できれ

ば「押した時の動作が場面によって変わってくるようなボタン (ただし if 文による振り分けではなく実現)」をつけてみるとよい。

□ (4) その他、Java を用いて「オブジェクト指向言語らしい」プログラム (アプレットでもアプリケーションでもよい) を作成せよ。