プログラミング言語論'10 ＃3

久野　靖 *

2010.4.22

はじめに

□ 前回やったこと

- C++と Java の「抽象データ型までの (基本部分の)」設計比較
- オブジェクトの配置方法の違い
- ガベージコレクション
- 代入演算子、コピーコンストラクタ
- 書き換え可能/不能、例外

□ 今回やること

- オブジェクト指向の考え方
- クラス方式のオブジェクト指向言語 (実現方法を含む)
- インタフェース、継承、動的分配
- Smalltalk-80
- ドリトル (実現方法を含む)
- (Lisp 系、型などの話も一応用意)

1　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

1.1　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」

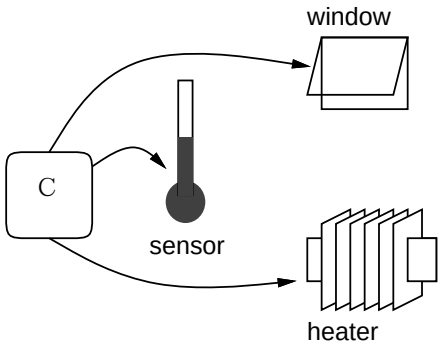
□ オブジェクト指向が取り入れられている分野はいろいろある

- 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

*筑波大学大学院経営システム科学専攻

1.2　オブジェクト指向的な考え方

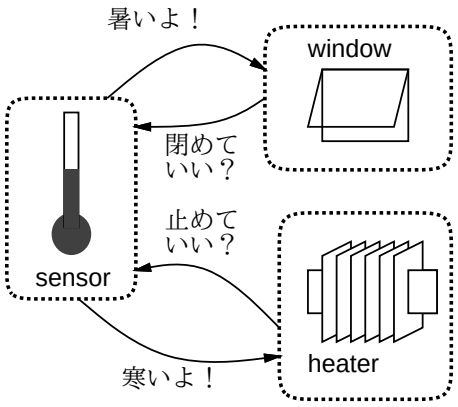
□ たとえば、「温室の温度調節システム」を考える。



□ 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)。

1.3 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

2 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。

2.1 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、C++ではメンバ変数) →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++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。

```
// Java
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
  %
```

```
// C++
#include <iostream>
#include <string>
```

```
#include <cstdlib>
using namespace std;

class Car {
  const string name;
  double speed;
public:
  Car(string n, double s);
  void change_speed(double d);
  void show_info();
};

Car::Car(string n, double s) :
  name(n), speed(s) {
}
void Car::change_speed(double d) {
  speed += d;
}
void Car::show_info() const {
  cout << "Car: " << name << " speed: "
    << speed << "\n";
}

int main() {
  Car c1 = Car("My Car", 50);
  Car c2 = Car("Father's Car", 80);
  c1.show_info(); c2.show_info();
  c1.change_speed(-20); c1.show_info();
}
```

□ 今回は string クラスを利用。文字列定数 (char*) から string への自動的な初期設定 (=変換) が使われている

2.2 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。

- オブジェクト生成時にその大きさの領域を用意し、初期設定する (コンストラクタを書かない場合は標準の初期設定)

□ 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)

2.3 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)

2.4 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 { ↓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。

2.5 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」

```
// Java
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

□ C++はインタフェースの代わりに「純粋な抽象クラス」を使う

```
// C++
class Figure {
public:
  virtual void draw(Graphics g) const = 0;
  virtual void moveTo(int x, int y) = 0;
  virtual int getX() const = 0;
  virtual int getY() const = 0;
};
```

- データ定義はない
- virtual は「このメソッドは動的分配 (後述) を行う」
- 「= 0」は「このメソッドはこのクラスでは定義しない」

□ 上のインタフェースを使ったアプレット (以下しばらく動く例題は Java のみになります)

□ 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 Sample22.java」
- 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>
```

2.6 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

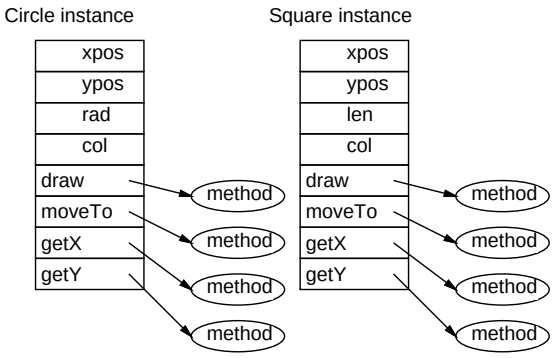
```
import java.applet.*;
import java.awt.*;

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)

2.7 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 (メソッド表) 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

□ C++では「virtual」指定の関数が 1 個以上ある場合にのみ、メソッド表が生成され、各オブジェクトにそこへのポインタが追加される

- virtual が 1 個もないクラス→「具象クラス (concrete class) →単純な ADT として使い、効率がよい
- virtual を持つクラス→動的分配によるオブジェクト指向的なプログラミングのためのもの

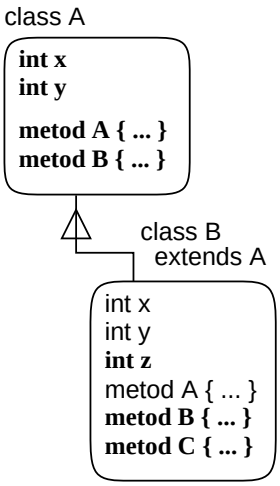
□ ただし、そのようなことをやる場合は「ポインタ型を」使う必要がある。なぜか分かりますか?

□ (ポインタでなく) 直接値を扱うと→その値の領域サイズはコンパイル時に決定されて確保される→そこに「さまざまなデータを」入れることはできない (入り切らない部分は切捨てられてしまう!!!)

- …ということは、オブジェクト指向ぽく使うばあいはどのみちポインタにするしかない→そうなるとJavaの「全部ポインタで統一」が魅力的に見える

2.8 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)

□ Javaでは継承もインタフェースと同様、動的分配の対象になる(むしろ一般的には継承の方が先)

□ C++では継承しても親クラスでvirtualと指定していないメソッドは動的分配しない

□ 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)

□ C++における継承→グラフィクスの例題は無いので、概要だけ。

- 親クラスは次の形で指定する

```
class クラス名 : public 親クラス {
  ...
}
```

- public の代わりに private とも指定できる→継承したものはそのクラス内でしか参照できない。つまり外から見たら継承していないのと同じ (純粋な実装の継承)
- 純粋な抽象クラスの継承→ Java でいう implements と同様の役割り
- 継承は複数指定可能→多重継承 (後述)
- コンストラクタの冒頭で親クラスのコンストラクタを呼ぶ構文がある (メンバ変数も同じ構文で初期化できる…初期化と代入が違うので結構重要)。

```
class Someclass : public Parent1 {
  double val;
public:
  Someclass(...)
    : Parent1(...), val(3.14) { ... }
}
```

2.9 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

□ C++では個別のメソッドに「= 0」を指定するだけでクラスについては特に指定はない

□ 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);
  }
}
```

2.10 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

2.11 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。

3 Smalltalk: オブジェクト指向の再発見

□ 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

□ 「再発見」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/)

3.1 「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

3.2 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

3.3 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)

3.4 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)

3.5 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

3.6 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 (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)

3.7 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

4 Dolittle: 教育用オブジェクト指向言語

□ 兼宗・久野が 2000 年ごろ設計

□ 兼宗: フリーの LOGO 処理系「ロゴ坊」の作者

- LOGO: タートルグラフィクスを搭載し、子供にプログラムを体験させるための言語として一定の地位
- 言語としては「括弧のない Lisp」、再帰とリストを多用 →タートルグラフィクスの「先」が難しすぎる
- これからは「既存の部品を組み合わせて活用」のようなことが必須になりそう → 「子供でも使える教育用オブジェクト指向言語」

□ 久野: 目新しい言語が作れれば嬉しい

4.1 Dolittle の設計方針

□ 日本語を使用できる

- 日本語プログラミング言語…色々なものがある世界
- Dolittle の立場は「言語には依存しない」「標準的な名前として日本語を使用」
- この標準的な名前を別の定義にしておけば「英語版」「韓国語版」などもすぐに作れる

□ 記号類はできるだけ少なく

- Java 等の「object.method(p1, p2, ...)」で既に複雑すぎ

□ オブジェクト指向を活かしたいし、教えたい

- タートルグラフィクス→「タートルオブジェクト」の機能として残す
- タートルグラフィクスで描いたものもオブジェクトとして扱う

□ 分かりやすく、間違えにくく

- プログラムは常に「まっさらの状態で先頭から」動く
- 全角半角の違いなどは最大限許容
- エラーをできるだけ出さない (これはよかったかどうか?)

4.2 DEMO(体験)

□ 「1 時間で学ぶソフトウェアの仕組み」

http://kanemune.eplang.jp/diary/2008-11-06-1.html

□ 以下のコードを 1 行ずつ解説しながら生徒にもやってもらう

```
かめた＝タートル！作る。
左ボタン＝ボタン！"左" 作る。
左ボタン：動作＝「かめた！30　左回り」。
右ボタン＝ボタン！"右" 作る。
右ボタン：動作＝「かめた！30　右回り」。
時計＝タイマー！作る。
時計！「かめた！10　歩く」実行。
タートル！作る "tulip.png" 変身する
          ペンなし 100 100 位置。
タートル！作る "tulip.png" 変身する
          ペンなし 100 -100 位置。
タートル！作る "tulip.png" 変身する
          ペンなし -100 100 位置。
かめた：衝突＝「|相手|　相手！消える」。
```

□ 中学・高校などでうまく授業ができているとの報告

- どのようなことが分かるか?

□ このような体験で分かってもらえる (と期待される) こと

- プログラムがどんなものかわかる
- ゲームなどのソフトはプログラムで作られている
- プログラムは人間が書いている
- プログラムは特別な「言語」で書く
- 文法が違うとエラーになる
- 間違って書くと間違って動く
- 書かれていないことは実行されない
- 上から順に実行される
- ある状態になったときに実行される命令もある (ボタン、衝突)
- ソフトは自分たちで作れる
- キー入力やマウスカーソルもプログラムが表示している (OS)

4.3 Dolittle 構文設計

□ メッセージ送信式

- 「object!p1 p2 ...　method.」メソッド名が最後だと日本語で読む場合にはそれらしい。
- さらに続けて次のメッセージ送信を書けるようにし　た　。「object!p1 p2 ...　meth1 p1 p2 ...　meth2 ...　methN.」
- 問題:　メッセージセレクタは「名前」。パラメタのところが数値ならいいが、変数名を書きたければどうするの?

□ 答え:　「かっこで囲む」

- 式は「中置記法」「メッセージ式」のどちらか
- どちらであるかは「!」が現れるかどうかで曖昧さなく区別できる
- 「!」の後ろではすべての名前はメッセージセレクタ
- 「()」で囲めばその中の部分式がどちらの種別かはまた独立

□ 制御構造は「言語に組み込みでは存在していない」「ブロックとメッセージ式によってすべて実現」← Smalltalk-80 が好きだから

□ ブロックとは…

- 「実行を遅延するコードの断片」…ブロック内のコードは必要に応じて別途起動
- 「パラメタを持てる」…手続きとしても使いたいから
- 「周囲の環境を保持 (クロージャ)」…制御構造として使う場合必須

□ ブロックの典型的な使い方

- 「…」!実行 --- 実行させる

- 「|x| …」!値 実行 --- パラメタを指定して実行
- 「|i| …」!回数 繰り返し --- 計数ループ
- 「…」!の間「…」実行 --- while ループ
- 「…」!なら「…」そうでなければ「…」実行 --- if 文
- 配列!「|x|…」それぞれ実行 --- foreach

4.4 Dolittle の実行方式設計

□ 基本的な設計…JavaScript に類似 ← たまたま JavaScript の勉強をしたから (でも、全部同じではない)

- オブジェクトは複数の名前つきプロパティの集合
- オブジェクトのインスタンス変数はプロパティにより表現
- オブジェクトのメソッド: 組み込み (native) とユーザ定義がある
- ユーザ定義のメソッドは「プロパティにブロックが入れてある」だけ

□ 「継承」の実現方式 --- プロトタイプ方式

- 子供に「クラス」とか「インスタンス」とか言っても無理そうだから…
- 基本的な考え方: 「あるオブジェクトに似たオブジェクト」を作りたいなら、そのあるオブジェクトをコピーすればできる。
- 実際に全部コピーしていると領域が無駄＋共有がなされない
- 「コピー」オブジェクトから「元の」オブジェクトへのリンクを張っておく (プロトタイプリンク→連鎖→プロトタイプチェーン)
- あるオブジェクトに「ないもの」はプロトタイプ (親) チェーンをたどって見つかるまで探していく

□ ドリトルでは上記の意味の「コピー」は「作る」。「ペン=タートル！作る。」

□ 興味深いポイント: 参照するときはチェーンをたどって探すが、書き込む時は手もとに同名のプロパティを作る

- 理由: オブジェクトを分けた意味がなくなるから＋共有されたままだと不都合
- 他のプロトタイプ方式の言語 (例: JavaScript) でも同様
- 理詰めで考えるとそうなのだが、ちょっとびっくりする

□ Dolittle 固有の特徴: 「すべてを」オブジェクトとプロトタイプチェーンだけでやるという方針で設計 ← 簡単で面白そう

- 「ルート」オブジェクト: すべてのオブジェクトの親 (グローバル変数もここに保持される)
- メソッドの実行時→「実行環境オブジェクト」が作られ、そこにローカル変数やパラメタが用意されている
- ブロックの内側から外側の変数をアクセスできる…ブロック内側の実行時に使われる「実行環境オブジェクト」のプロトタイプチェーンが外側の環境オブジェクトを指している (クロージャ)
- ただし「実行環境オブジェクト」には実行中には新たなプロパティは定義されない (なぜか?)

□ メソッド定義の場合はどうするか?

- メソッド中で変数に書き込んだらそれはインスタンス変数 == オブジェクトのその名前のプロパティ
- ブロックを「メソッドとして実行するときは」「実行環境オブジェクトの親はそのオブジェクト」
- ブロックを「メソッドとしてでなく実行するときは」元々の外側の変数 (クロージャ)

4.5 Dolittle 言語のまとめ

□ 教育用オブジェクト指向言語 --- タートルグラフィクス、GUI、アニメーションなど教育に使いやすいようにライブラリを設計

□ 言語自体はプロトタイプ方式のオブジェクト指向言語

□ 「ブロックだけで制御構造を実現」「プロトタイプとオブジェクトで何でもやる」などのちょっと変わった実験が盛り込まれている

4.6 Dolittle の実習＋演習問題

□ Dolittle の実行環境: 「sma」の窓を開いてその中で「dol」で開始

- プログラムは Emacs を開いて「なんとか.dtl」というファイル名で作る
- 文字コードは UTF-8。Emacs で新たなファイルを作る時に「C-x RET f utf-8-u RET」を実行しておく
- Dolittle の「読み込み」ボタンを押してそのファイルを指定
- 「実行」で実行開始
- Dolittle のエディタはうちの環境だと日本語がうまく入らない (泣) けど、Emacs と窓 2 枚の方が使いやすい (負け惜しみ?)

□ 演習問題

- 上で説明したような「Dolittle の特徴」を確認するようなプログラムを作って実行結果について解説せよ。
- 「Dolittle の特徴を活かしていると思われる」プログラムを作成し、解説せよ。

5 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)

5.1 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

5.2 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

5.3 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)

5.4 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

5.5 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

6 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)

6.1 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

6.2 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?

6.3 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

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

- このキャストは「実行時の型検査を伴うキャスト (ダウンキャスト)」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)

6.4 型情報を利用した自動型変換

□ すべての式や変数に型がある→それらに「不一致」があることも

- 数値型など基本型の間では一定範囲で「自動変換」が提供されている
- オブジェクト型どうしでは「上位の変数に下位のオブジェクトが入れられる」という規則が「原則」

- オブジェクトと基本型→相互変換行わないのが「原則」
- 便利さのためのこの「原則」をやめて変換を可能とする場合がある

□ Java の AutoBoxing/Unboxing

- Java では「1」(値) と「new Integer(1)」のように基本型とそれに対応するオブジェクト型の両方を用意→相互に変換することが多い
- Boxing --- 「箱に入れる」: 値をオブジェクトにする
- Unboxing --- 「箱から出す」: オブジェクトから値を取り出す

```
Integer i1 = new Integer(1);  ← boxing
int j1 = i1.intValue();       ← unboxing
```

- JDK 1.5 以降でこれらを「自動化」:boxing / unboxing が必要になったときは自動的に行う (演算などの場合も)

```
Integer i1 = 100, i2 = 200; ← auto boxing
int j1 = i1, j2 = i1+i2;  ← auto unboxing
int j3 = i2++;  ← auto unboxing+boxing
```

- しかし「==」「!=」はオブジェクトどうしが等しいという意味がもともとあるので auto unbox されないことに注意
- メソッド呼び出しのために autoboxing されることはない。「3.toString()」はダメ

□ C++の自動変換機能

- 多くの場合「変換」しなくても演算子オーバロードで済む
- コンストラクタを定義することで自動型変換が可能になる
- 代入時も別扱いしたければ代入演算子も定義
- 基本型や既にあるクラスを「変換先」にしたい…コンストラクタは定義できない→その場合は「型変換演算子」を定義 (「operator T()」…自分クラスの値を T 型に変換)
- これらの複数の機能のどれを使うか→言語仕様で規定 (複雑)

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

class Test {
  int val;
public:
  Test() { val = 9; }
//Test(int i) { val = i+1; }
//void operator=(const int& i) { val = i+2; }
//operator int() const { return val+3; }
  int value() { return val; }
};

int main() {
  Test x;
//Test x = 10;
//x = 20;
  cout << x.value() << "\n";
//cout << (x+0) << "\n";
}
```

6.5 実行時の型情報

□ 前記のキャストのようなことができる←実行時にもある程度の型情報が保持されている

□ Java の場合: 上記のダウンキャスト、およびある型に属するかどうかを判定する 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)

□ C++の場合: 実行時の型情報は virtual メソッドのあるクラスに対してのみ利用可能。そうでない場合は実行時には情報がない。

- チェックのあるダウンキャスト → dynamic_cast<T>(値)
- 実行時に情報がない場合のキャスト → static_cast<T>(値)
- const のある型から const を外す → const_cast<T>(値)
- 何を何にでも変換できるキャスト → reinterpret_cast<T>(値) ← これまでの「(型)値」というキャストと同等。これまでのキャストは危険な上に目立たないのでやめさせたいという考え

6.6 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…

6.7 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)

7 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

7.1 継承の意味づけ

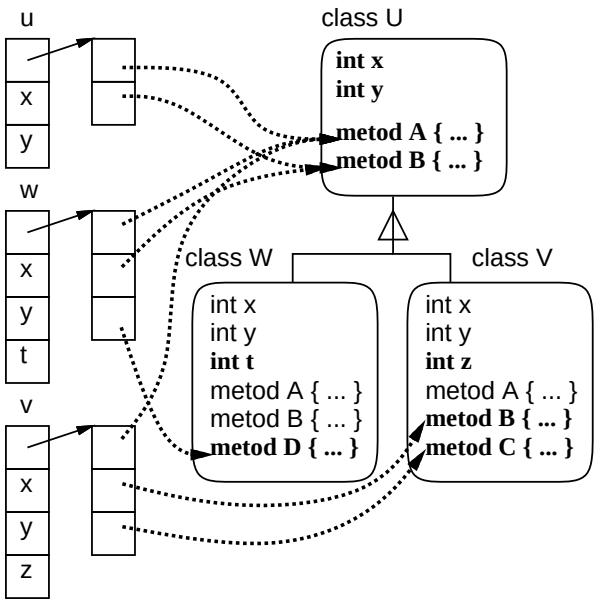
□ 継承がやってることは何かというと…

- 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

7.2 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

7.3 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、メソッド表を作ってそれに基づいて分岐すればよい。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

7.4 多重継承の実装

□ 多重継承→ C++ではサポート。Java では単純化のため禁止。

□ 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?

- 案 1:　親のインスタンスが 2 個埋め込まれる→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)(C++)
- 案 2:　共通部分は「統合」される→「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必要 (C++)
- 案 2 の別解→最初に名前で探索し、その場所を覚える (Flavors)

□ C++では案 2 は「virtual 親クラス」と呼ばれている。両方あるのは C++らしいが複雑。

- とくに、コンストラクタによる初期設定が複雑。「子」が共通の「親」を初期設定すると 2 回初期設定されてしまう。このため、virtual 親クラスになるものには「引数無しの」ものが用意されている必要。これが最初に呼ばれる。「子」のコンストラクタでは親のコンストラクタは呼ばないのが通例。

7.5　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

7.6　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

7.7 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)

7.8 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

8 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

8.1 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)

8.2 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

8.3 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

8.4 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

□ 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++）