プログラミング言語論'10 ＃2

久野　靖 *

2010.4.15

はじめに

□ 前回やったこと

- プログラミング言語の位置づけ
- プログラミング言語の各種機能
- 裸の計算機モデル→構造化→モジュール→抽象データ型→ (OO)

□ 今回やること

- オブジェクト指向言語 (ただし細かく検討する)
- C++と Java の比較 (言語設計とメカニズム部分)
- 基本的な書き方 (抽象データ型の部分メインで)
- オブジェクト指向の考え方
- 基本的な実現方法
- (この後さまざまな言語などの話題も一応用意)

1　オブジェクト指向言語入門

□ 本講座では実際にオブジェクト指向言語を使ってレポートをやって頂きたい→具体的な言語の紹介も必要

- 言語として「C++」「Java」の両方を取り上げる
- 例題･課題ともできるだけ両方を対比させて用意する
- ただし題材によってはどちらか片方だけになるものもある

1.1　C++入門

□ C++ --- Bjane Stroustrup　に　よって　、「C with class」として始まった

- C からスムーズに移行できるオブジェクト指向言語として普及
- C に (ほぼ) 上位互換な言語。型検査とかは厳しくなっている
- +α されているもの…「オブジェクト指向」「例外」「テンプレート」など
- C の「裸のマシンがそのまま使える」に「よい構造化ができる」を追加

□ C++の設計思想…

- フルスピードで動く (C に負けない)
- そのために足枷となるものは言語に導入しない
- どのようなプログラミングスタイルでもサポートする
- プログラマが分かっていてやることは禁止しない
- ユーザ定義型も組み込み型と同様な見た目で使える

□ 例題:　2 つの数を読み込んで足す

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

int main(void) {
  int x; cout << "x? "; cin >> x;
  int y; cout << "y? "; cin >> y;
  cout << "x+y = " << (x+y) << '\n';
}
```

- 「cin」「cout」は入出力ストリーム
- 「>>」「<<」(もともとはシフト演算子) を演算子定義により入出力演算として使っている
- 「>>」の結果はまたストリームなので連続して使える
- 多重定義により、「<<」を文字列用、整数用、実数用と多数用意している

□ C++の処理系は…

- Windows 用…Borland C++、MS Visulal C++など色々ある
- フリーなもの…GNU C++ (G++)。Unix (Linux) には普通ついている。Windows なら Cygwin を入れればそこに一緒にいれて動かせる

1.2　Java 入門

□ Java --- 「C++よりも安全なオブジェクト指向言語」として SunMicrosystems 社で開発された

- 当初 Web ブラウザの上で動く「アプレット」のための言語として普及

*筑波大学大学院経営システム科学専攻

- この場合の「安全」…プログラムが悪さをできないような防壁がある (例:　ファイルの読み書きができない、勝手にネットワーク接続ができない、など)
- 現在では通常のソフトウェア開発用言語として使われている
- C(や C++) とは制御構造の構文が似ているだけであとは別物

□ Java の設計思想…

- C++の危険さ、複雑さを減少させ単純さを求める
- C や C++との互換性は捨てたので言語仕様は単純化できた
- 安全さを第一とし、危険なことはどうやってもできない
- すべてのオブジェクトはヒープ上に割り当て、ごみ集めを行う
- Smalltalk 的、伝統的なオブジェクト指向言語ふう
- これらの代償として見た目は長く、動作は遅くなりがち

□ 例題:　2 つの数を読み込んで足す

```
import java.util.*;

public class Sample1 {
  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in);
    System.out.print("x? ");
    int x = sc.nextInt();
    System.out.print("y? ");
    int y = sc.nextInt();
    System.out.println("x+y = " + (x+y));
  }
}
```

1.3　オブジェクト指向言語

□ オブジェクト指向言語とは?→いろいろな定義があると思うがここでは一応次のように考える。

- 「オブジェクト指向」→プログラムが扱う対象をそれぞれ自律的な/完結した「もの」(オブジェクト) であるとする「考え方」
- 「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するようなプログラミング言語

□ 抽象的でよく分からない?　まあそうだと思いますが…

```
outstream ostream = ...;
outstream_println(ostream, "Hello, World.\n");
↑関数名が長い　　↑操作対象も引数 : 従来のスタイル

outstream ostream = ...;
ostream.println("Hello, World.\n");
↑オブジェクト . メソッド (引数…) : オブジェクト指向ぽい
```

- 同じことを 2 度言わなくて済む感じ
- 同じことには同じ名前が使える
- 名前空間の節約

□ あと、とりあえず関連する用語を紹介しておく

□ 「クラス」→オブジェクトの種類に対応する構文要素。

- 新しい種類のオブジェクトを定義したければ、クラスを書く。

□ クラス定義に含まれるもの…

- オブジェクトはどういう動作 (メソッド、メンバ関数) を持っているか
- オブジェクトはどういう属性を持っているか←インスタンス変数、メンバ変数
- そのほかクラスはモジュールとしての役割りも←クラスメソッド変数、static メソッド/変数

□ 「インスタンス」→クラス定義に基づいて作り出された「もの」(オブジェクト)

□ 逆に見ると→「新しいモノ」を定義できるのがオブジェクト指向

- 世の中はさまざまなモノから成り立っている→それをそのまま言語に移せる (プログラムだから現実世界よりは杓子定規だけど…) →考えやすい (考えやすさは重要)

□ どういうモノを作るか考える→オブジェクト指向における設計

- (1) どんなモノを作るか
- (2) それぞれのモノはどういう操作/インタフェースを持つか
- (3) その実装はどうするか←ある意味どうでもいい/別建て
- (上記は ADT の考え方と言った方が正確← ADT は重要)

1.4　例:じゃんけんの手

□ 「じゃんけん」のデータを多数扱うアプリがあるとする

- 従来手法:　たとえば"Goo"、"Paa"、"Choki"の文字列で扱う→何がよくない?
- 従来手法:　たとえば 0、1、2 をグー、パー、チョキに対応→何がよくない?
- じゃあどうするか?

□ 「じゃんけんの手」をデータ型とする

- 常に「3 つの手のどれか」であることが保証される

- どういう操作を持たせるか?

□ 必要な操作

- 作る (文字列→手、ランダムな手)
- 表示 (手→文字列)
- 勝ち負け引き分けの判定

1.5 じゃんけんの手:C++版

□ C++のクラス:インタフェース (ヘッダに入れる部分)

```
class RPS { // Rock Paper Scissors
  int hand;
  enum { rock = 0, paper = 1, scissors = 2 };
  static const char* const names[3];
  static const int wl[3][3];
public:
  RPS();
  RPS(const char* const s);
  const char* c_str() const;
  bool operator==(const RPS& r) const;
  bool operator!=(const RPS& r) const;
  bool operator<(const RPS& r) const;
  bool operator<=(const RPS& r) const;
  bool operator>(const RPS& r) const;
  bool operator>=(const RPS& r) const;
  class Unknown_hand { };
};
const char* const RPS::names[3] = {
  "Rock", "Paper", "Scissors"};
const int RPS::wl[3][3] =  {
  {0,-1,1}, {1,0,-1}, {-1,1,0}};
```

- 「class クラス名 { ... };」がクラスを定義
- public:より前は private 部分→クラス外からは見えない (実際にはコンパイラは見るが) →カプセル化、ADT の保護
- 変数:「この型のデータ (インスタンス) は何と何を組にしたものか」を表す
- static は全インスタンスに共通のデータ (共有される表とか)

□ public:以下にメソッド (メンバ関数) のインタフェースを書く→これを (ヘッダファイルとして) 取り込むことで型検査可能に

- クラス名と同名のメソッド→「コンストラクタ」(初期化用の特別なメソッド)。
- メソッドは同名のものが複数あっても引数の数と型で区別できれば OK(オーバーローディング、多重定義)
- 「変更しない」ことを const で表す (型もメソッドも)。
- 「operator △」で「△」という演算子を定義可能。
- unknown_hand というクラスは「例外の種別を表す」ために使う。

□ C++のクラス:実装部分

```
RPS::RPS() {
  hand = int((double(rand()) / RAND_MAX)* 3);
}
RPS::RPS(const char *s) {
  switch(*s) {
case 'R': case 'r': case 'G': case 'g':
    hand = rock; break;
case 'P': case 'p':
    hand = paper; break;
case 'S': case 's': case 'C': case 'c':
    hand = scissors; break;
default: throw Unknown_hand();
  }
}
const char* RPS::c_str() const {
  return names[hand];
}
bool RPS::operator<(const RPS& r) const {
  return wl[hand][r.hand] < 0;
}
bool RPS::operator<=(const RPS& r) const {
  return wl[hand][r.hand] <= 0;
}
bool RPS::operator>(const RPS& r) const {
  return wl[hand][r.hand] > 0;
}
bool RPS::operator>=(const RPS& r) const {
  return wl[hand][r.hand] >= 0;
}
bool RPS::operator==(const RPS& r) const {
  return hand == r.hand;
}
bool RPS::operator!=(const RPS& r) const {
  return hand != r.hand;
}
```

- 「クラス名::名前」でクラスに付属するもの (メンバ) の名前が指定できる。定義自体は class { ... } とは別に書く。
- static な変数も定義を外の 1 個所に置く必要。
- メソッド (メンバ関数) の中は「クラスの中」だから private なものも参照可能。
- 文字列はここでは「C の文字列」を返す。string クラスとか色々あるが。

□ C++のクラス:メイン

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

//ここにヘッダ部分
//ここに実装部分 (別ファイルにしてもよい)

int main(void) {
  const int n = 5;
  RPS a[n], b[n];
  for(int i = 3; i < n; ++i) {
    char sa[n], sb[n];
    cout<<"a hand? "; cin>>sa; a[i] = RPS(sa);
```

```
    cout<<"b hand? "; cin>>sb; b[i] = RPS(sb);
  }
  for(int i = 0; i < n; ++i) {
    char *s = "draw";
    if(a[i] > b[i]) s = "a win";
    if(a[i] < b[i]) s = "b win";
    cout << a[i].c_str() << ":" <<
      b[i].c_str() << " --- " << s << "\n";
  }
}
```

- RPS 型 5 個の配列→コンストラクタにより初期化→ランダムな手が最初から入っている
- 2 つだけ手を読み込む→文字列を読んでコンストラクタで RPS 型を作り代入。
- 最後にどちらが勝っているか/引き分けかを表示。「>」「<」等は定義された演算子を呼ぶ。

1.6 じゃんけんの手:Java 版

□ Java では JDK 1.5 以降で「Enum 型」を導入。Enum 型とは要するに C++で作ったようなクラスを「ひとことで」作る機能。

```
enum RPS { ROCK, PAPER, SCISSORS }
```

- これにより以下のようなメソッドを持つクラスが作られる:
- static RPS RPS.ROCK, RPS.PAPER, RPS.SCISSORS --- 3 つの値に対応するインスタンスを保持する final 変数 (定数)
- public static RPS[] values() --- 3 つの値を格納した配列
- public static valueOf(String s) --- 文字列から 3 つの値に変換
- public int ordinal() --- 何番目の値かを返す

□ enum はクラスなのでメソッドを追加したりできる→それを利用したじゃんけんの手の main()

```
import java.util.*;

public class Sample2 {
  public static void main(String[] args)
      throws Exception {
    final int n = 5;
    RPS[] a = new RPS[n], b = new RPS[n];
    Scanner sc = new Scanner(System.in);
    for(int i = 0; i < 3; ++i) {
      a[i] = RPS.newHand(); b[i] = RPS.newHand();
    }
    for(int i = n-2; i < n; ++i) {
      System.out.print("a hand? ");
      a[i] = RPS.valueOf(sc.nextLine());
      System.out.print("b hand? ");
      b[i] = RPS.valueOf(sc.nextLine());
    }
    for(int i = 0; i < n; ++i) {
      String s = "draw";
      if(a[i].winlose(b[i]) > 0) s = "a win";
      if(a[i].winlose(b[i]) < 0) s = "b win";
      System.out.println(a[i] +  ":" + b[i] +
        " --- " + s);
    }
  }
}
```

- Java ではヘッダファイルがなく、前方参照も OK(コンパイラの作りがより新しい感じ)
- 「配列オブジェクトを作る」ことと「そこに入れるオブジェクトを作る」ことは常に別。(値の配列なら 0 が入るのだけど…)
- 構文上の細工はあまりなく、基本的にすべてメソッド呼び出し形式だけで書く。

□ クラス RPS は…enum なので制約があるがインスタンス変数やメソッドを持たせることができる

```
enum RPS {
  ROCK, PAPER, SCISSORS;
  static final int[][] wl = {
    {0,-1,1}, {1,0,-1}, {-1,1,0}};
  public int winlose(RPS r) {
    return wl[ordinal()][r.ordinal()];
  }
  public static RPS newHand() {
    RPS[] hands = values();
    return hands[(int)(Math.random()*hands.length)];
  }
}
```

- static の意味は C++と同じ。
- winlose は勝ち負けに応じて「正負零を返す」ようにした。
- enum でなく普通のクラスの例→次の節で

1.7 整数を出し入れするバッファ

□ いきなり難しかったですか? そう言われると思ってもっと易しい練習問題ぽいものを用意してあります。

□ 数を「ある決まった規則で」保持するバッファというものを作ります。

- 数を入れる→「buf.put(値)」
- 数を取り出す→「x = buf.get()」

□ Mybuf1 は次の規則により保持するものとします。

- 「最近に入れた 2 つの数を保持する (初期値は 0)。1 回取り出すと、一番最後に入れた値が出て来る。もう 1 回以上取り出すと、それより 1 つ前に入れた値が出て来る」。

□ C++版の Mybuf1

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

class Mybuf1 { // memorize last two
  int val1, val2;
public:
  Mybuf1();
  void put(int x);
  int get();
};

Mybuf1::Mybuf1() {
  val1 = val2 = 0;
}
void Mybuf1::put(int x) {
  val2 = val1; val1 = x;
}
int Mybuf1::get() {
  int x = val1; val1 = val2;
  return x;
}

int main(void) {
  Mybuf1 buf;
  while(true) {
    char c; int v;
    cout << "? "; cin >> c;
    switch(c) {
case 'q': return 0;
case 'p': cin >> v; buf.put(v); break;
case 'g': cout << buf.get() << '\n'; break;
    }
  }
}
```

□ Java 版の Mybuf1

```
import java.util.*;

public class MybufTest1 {
  public static void main(String[] args) {
    Mybuf1 buf = new Mybuf1();
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("? ");
      char c = sc.next().charAt(0);
      switch(c) {
case 'q': return;
case 'p': buf.put(sc.nextInt()); break;
case 'g': System.out.println(buf.get());
      }
    }
  }
}

class Mybuf1 {
  int val1, val2;
  public Mybuf1() {
    val1 = val2 = 0;
  }
  void put(int x) {
    val2 = val1; val1 = x;
  }
  int get() {
    int x = val1; val1 = val2;
    return x;
  }
}
```

- sc.next() は「1 単語」読む。その先頭の文字を charAt(0) で取り出し、使用する。(Scanner には 1 文字だけ読むというメソッドがないためちょっと不便。)

□ では練習問題をやってみましょう。C++でも Java でも好きな方で (ただしできるだけ普段使っていない言語で) 作成してください。

- 練習 1: 「最近 3 つの値を覚えるバッファ」を Mybuf2 という名前で作りなさい。
- 練習 2: 「最後に入れた値を覚えるが、取り出すごとに覚えている値が 1 つずつ減るバッファ」を Mybuf3 という名前で作りなさい。
- 練習 3: 「入れた値が累計され、取り出すとその累計値が取れるバッファ」を Mybuf4 という名前で作りなさい。

□ C++はこういう感じ

```
(冒頭省略)

class Mybuf2 {
  (インスタンス変数をここで定義)
public:
  Mybuf2();
  void put(int x);
  int get();
};

Mybuf2::Mybuf2() {
  (初期化動作を記述)
}
void Mybuf2::put(int x) {
  (入れる動作を記述)
}
int Mybuf2::get() {
  (x に取り出す動作を記述)
  return x;
}

(main は同様なので省略)
```

□ Java はこういう感じ

```
(冒頭+main 部分省略)

class Mybuf2 {
  (インスタンス変数の定義)

  public Mybuf1() {
    (初期化動作を記述)
  }
  void put(int x) {
    (入れる動作を記述)
  }
  int get() {
    (x に取り出す動作を記述)
```

```
    return x;
  }
}
```

1.8 分数電卓:C++版

□ 分数をあらわすクラスを用意する。方針としては2つの数 a、b で分数 a/b を表す。まず宣言部。

```
class Rational { // a/b
  int a, b;
  int gcd(int x, int y);
public:
  Rational(int x);
  Rational(int x, int y);
  Rational operator+(const Rational& r) const;
  Rational operator-(const Rational& r) const;
  Rational operator*(const Rational& r) const;
  Rational operator/(const Rational& r) const;
  friend ostream& operator<<(ostream& o, Rational& r);
  friend istream& operator>>(istream& i, Rational& r);
};
```

- private 部分には変数と補助関数 GCD(最大公約数)。
- コンストラクタは値1つ (分母 1) と2つ。
- 四則は演算子。
- 分母 0 になる場合は NaN (Not a Number)。
- 入出力は単独関数として定義した演算子「>>」と「<<」。(なぜ単独関数かというと、クラス istream や ostream に後からメソッドの追加はできないから。) 第1引数は入出力ストリーム。firend 宣言は「この関数は特別にこのクラス定義の中身 (private 部分) にアクセスできる」という意味。

□ 実装部分。

```
int Rational::gcd(int x, int y) {
  while(x != y) if(x > y) x -= y; else y -= x;
  return x;
}
Rational::Rational(int x) { a = x; b = 1; }
Rational::Rational(int x, int y) {
  if(y == 0) { a = b = 0; return; }
  if(x == 0) { a = 0; b = 1; return; }
  if(y < 0) { x = -x; y = -y; }
  if(x == 0) {
    a = x; b = y;
  } else if(x > 0) {
    a = x / gcd(x,y); b = y / gcd(x,y);
  } else {
    a = x / gcd(-x,y); b = y / gcd(-x,y);
  }
}
Rational Rational::operator+(const Rational& r) const {
  return Rational(a*r.b + r.a*b, r.b*b);
}
Rational Rational::operator-(const Rational& r) const {
  return Rational(a*r.b - r.a*b, r.b*b);
}
Rational Rational::operator*(const Rational& r) const {
  return Rational(a*r.a, r.b*b);
}
Rational Rational::operator/(const Rational& r) const {
  return Rational(a*r.b, r.a*b);
}
ostream& operator<<(ostream& o, Rational& r) {
  if(r.b == 0) o << "NaN";
  else o << r.a << '/' << r.b;
  return o;
}
istream& operator>>(istream& i, Rational& r) {
  int a, b; i >> a >> b;
  r = Rational(a, b); return i;
}
```

- 面倒な「約分」の計算はコンストラクタで。

□ メイン部分。

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

// ここにヘッダ部分
// ここに実装部分

int main(void) {
  Rational r(1), x(0);
  while(true) {
    char c; cout << "? "; cin >> c;
    if(c == 'q') return 0;
    switch(c) {
case 'q': return 0;
case '=': cin>>x; r = x; break;
case '+': cin>>x; r = r + x; break;
case '-': cin>>x; r = r - x; break;
case '*': cin>>x; r = r * x; break;
case '/': cin>>x; r = r / x; break;
default: continue;
    }
    cout << r << '\n';
  }
}
```

- 1文字読んでコマンド文字によって動作を切替え。例:

```
? = 1 3  ← 1/3 を入れる
1/3      ←現在値
? + 1 6  ← 1/6 を足す
1/2      ←現在値
q        ←おしまい
```

1.9 分数:Java 版

□ C++版と同じ方針で実装。こちらはメイン側から。

```
import java.util.*;

public class Sample3 {
  public static void main(String[] args)
      throws Exception {
    Scanner sc = new Scanner(System.in);
    Rational r = new Rational(1);
    while(true) {
      System.out.print("? ");
      String s = sc.nextLine();
```

```
      if(s.charAt(0) == 'q') return;
      Rational x = new Rational(s.substring(1));
      switch(s.charAt(0)) {
case '=': r = x; break;
case '+': r = r.add(x); break;
case '-': r = r.sub(x); break;
case '*': r = r.mul(x); break;
case '/': r = r.div(x); break;
default: continue;
      }
      System.out.println(": " + r);
    }
  }
}
```

- こちらはコンストラクタに文字列を渡せるように作った。文字列を空白で区切って扱うクラス StreamTokenizer を活用。
- 動かし方は C++版と同じ。

□ Rational クラスは内容的にはほぼ同じ。

```
class Rational {
  int a = 0, b = 0; // b == 0 -> Not a Number
  public Rational(String s) {
    try {
      StringTokenizer tok = new StringTokenizer(s);
      if(tok.countTokens() == 1) {
        a = Integer.parseInt(tok.nextToken()); b = 1;
      } else if(tok.countTokens() == 2) {
        a = Integer.parseInt(tok.nextToken());
        b = Integer.parseInt(tok.nextToken());
      }
    } catch(Exception ex) { }
  }
  public Rational(int x) { a = x; b = 1; }
  public Rational(int x, int y) {
    if(y == 0) { return; }
    if(x == 0) { a = 0; b = 1; return; }
    if(y < 0) { x = -x; y = -y; }
    if(x == 0) {
      a = x; b = y;
    } else if(x > 0) {
      a = x / gcd(x,y); b = y / gcd(x,y);
    } else {
      a = x / gcd(-x,y); b = y / gcd(-x,y);
    }
  }
  public Rational add(Rational r) {
    return new Rational(a*r.b + r.a*b, r.b*b);
  }
  public Rational sub(Rational r) {
    return new Rational(a*r.b - r.a*b, r.b*b);
  }
  public Rational mul(Rational r) {
    return new Rational(a*r.a, r.b*b);
  }
  public Rational div(Rational r) {
    return new Rational(a*r.b, r.a*b);
  }
  public String toString() {
    if(b == 0) return "Nan"; else return a + "/" + b;
  }
  private int gcd(int x, int y) {
    while(x != y) if(x > y) x -= y; else y -= x;
    return x;
  }
}
```

- 演算子定義はないので適当なメソッド名をつける。
- 出力も演算子はないので toString() で文字列への変換方法を定める（文字列が必要なときは自動的にこれが呼ばれる）。

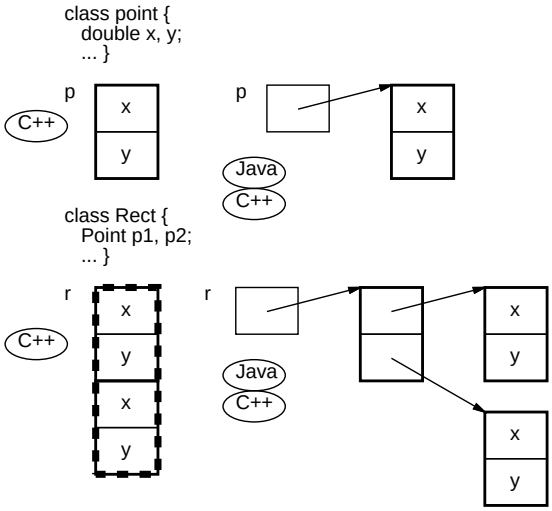
1.10 値とオブジェクト

□ C++も Java も「値」と「オブジェクト」を区別している。

- 値〜数値、文字、論理値。
- オブジェクト→ C++でも Java でもクラスにより定義。
- ポインタ型/参照型→ C++にのみ存在。Java ではオブジェクト値は全部ポインタのようなもの。

□ C++と Java で一番違うところは:

- C++ではオブジェクトは「配列や実行スタック上に直接その記憶域を配置できる。外に配置するときはポインタを使う」
- Java ではオブジェクトは「すべてヒープ上のオブジェクト。オブジェクトはすべてポインタで扱う」（Java の用語では「参照」）→最も重要な単純化。動的データ構造や多態のためにはどのみち参照が必要。そのため、すべてポインタに統一している。そのために遅い面も。



- そのため、Java ではごみ集め（GC、後述）が必須。C++ではごみ集めはオーバヘッドがあるので言語本体としては提供しない→手動メモリ管理が前提（うまくやれば効率はよいが面倒、間違えるとメモリリークしたり落ちたりする）。

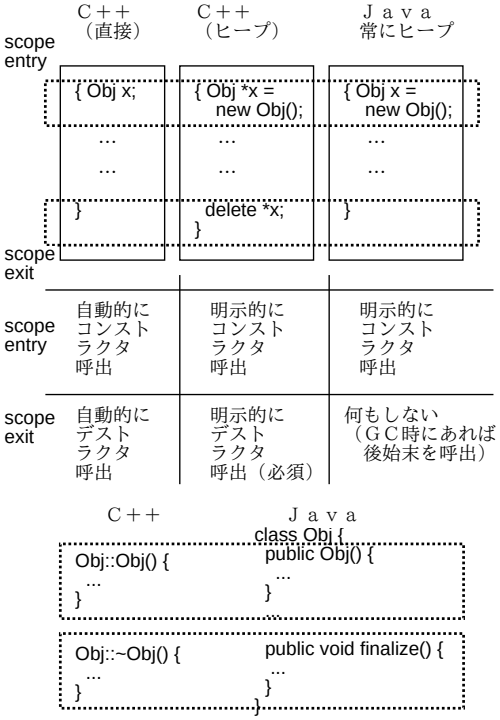
□ C++はオブジェクトも書き方を値に近くできる。Java はそれはしない。

- C++では「==」などの演算子を定義できる→オブジェクトでも演算子を使って書く。
- Java では「値→演算子」「オブジェクト→メソッド」ときっぱり区分されている。「v1 == v2」(値)、「o1.equals(o2)」。もしオブジェクトに「o1 == o1」を使うと「ポインタ比較 (同一のオブジェクトかどうかを調べる)」になる。

□ C++でのオブジェクトの初期設定と後始末は…

- コンストラクタとデストラクタ (名前が「〜クラス名」) による
- ローカル変数のオブジェクトはスコープに入った時にコンストラクタが呼ばれ、出るときにデストラクタが呼ばれる。
- ヒープ上のものは「new クラス名 (…)」でメモリ割り当て+コンストラクタ、「delete オブジェクト」でデストラクタ+メモリ開放
- 配列の場合「new クラス名 [大きさ]」で割り当て+デフォルトコンストラクタ (引数なし)、「delete[] ポインタ」でデストラクタ+メモリ開放。(ポインタが単一要素か配列参照か分からないという弱点は C 言語から引き継いでいる。)

□ Java の場合は「new クラス名 (…)」でコンストラクタというのは C++と同じ。配列はポインタ配列なので初期値 null。領域回収は GC なのでずっと簡単。後始末用にはメソッド finalize()、ただし GC が領域を回収するときに呼ばれるので、呼ばれない場合も。



1.11 Java とごみ集め (GC)

□ Java のコード→どんどん必要なオブジェクトを生成し、使わないものは GC で回収、というスタイル。

- GC のオーバヘッドは織り込み済みという割り切りが必要。
- 自分で領域開放するよりずっと楽。自前でやるとコードが複雑化し、また重大な誤りの原因になりやすい。
- GC は Lisp あたりでは古くから使われているが、普通の手続き型言語で標準としたのは Java がはじめて

□ GC の原理: 変数から参照できるオブジェクト群を順次たどって行き、たどったという印をつける。印がつかなかった領域はごみとして回収

- その他、使っているオブジェクトだけコピーする方式、オブジェクトがそれぞれ何個所から指されているか常に数えておく方式などもある。
- GC で自動的に回収するためには「不要なデータ構造は指さないようにする」(指している所に null を入れるなどしてつながりを切る) ことが必要
- 「もう要らないからこれ回収して」とは言えない (なんで?)

1.12 整数の集合: C++版

□ より複雑なものの例として「整数の集合」型を作ってみる。

```
static const int maxsize = 100;

class Intset { // set of strings
  int count;
  int arr[maxsize];
  void add1(int x); // internal use only
public:
  Intset();
  int size() const;
  bool is_in(const int i) const;
  Intset operator+(const Intset &s) const;
  Intset operator-(const Intset &s) const;
  Intset operator*(const Intset &s) const;
  class Overflow { };
  friend ostream& operator<<(ostream& o, Intset& s);
  friend istream& operator>>(istream& i, Intset& s);
};
```

- 実装の方針→配列を用いて整数の列を保持。とりあえず最大固定。
- インタフェースはとりあえず最低限:

□ 実装:

```
Intset::Intset() { count = 0; }
void Intset::add1(int x) {
  if(count+1 >= maxsize) throw Overflow();
  arr[count++] = x;
```

```
}
int Intset::size() const { return count; }
bool Intset::is_in(const int i) const {
  for(int k = 0; k < count; ++k)
    if(arr[k] == i) return true;
  return false;
}
Intset Intset::operator+(const Intset &s) const {
  Intset r = s;
  for(int k = 0; k < count; ++k)
    if(!s.is_in(arr[k])) r.add1(arr[k]);
  return r;
}
Intset Intset::operator-(const Intset &s) const {
  Intset r;
  for(int k = 0; k < count; ++k)
    if(!s.is_in(arr[k])) r.add1(arr[k]);
  return r;
}
Intset Intset::operator*(const Intset &s) const {
  Intset r;
  for(int k = 0; k < count; ++k)
    if(s.is_in(arr[k])) r.add1(arr[k]);
  return r;
}
ostream& operator<<(ostream& o, Intset& r) {
  o << "{ ";
  for(int k = 0; k < r.count; ++k) o << r.arr[k] << ' ';
  o << '}'; return o;
}
istream& operator>>(istream& i, Intset& r) {
  r.count = 0;
  while(i.peek() != '\n') {
    int x; i >> x; r.add1(x);
  }
  i.get(); return i;
}
```

- 下請けメソッド add1() はあふれチェックして要素を追加する。
- 演算のメソッドは 2 つの集合に含まれているかどうかをそれぞれチェックしながら動作。
- 出力は書くだけ。入力は読み込みながら add1() を呼ぶ。

□ メイン部分は先の例とほとんど代わらない

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

// ここにヘッダ部分
// ここに実装部分

int main(void) {
  Intset r, x;
  while(true) {
    char c; cout << "? "; cin >> c;
    if(c == 'q') return 0;
    switch(c) {
case 'q': return 0;
case '=': cin>>x; r = x; break;
case '+': cin>>x; r = r + x; break;
case '-': cin>>x; r = r - x; break;
case '*': cin>>x; r = r * x; break;
default: continue;
    }
    cout << r << '\n';
  }
}
```

1.13 整数の集合: Java 版

□ main 側。これもこれまでと同様。

```
import java.util.*;

public class Sample14 {
  public static void main(String[] args)
      throws Exception {
    Scanner sc = new Scanner(System.in);
    IntSet r = new IntSet();
    while(true) {
      System.out.print("? ");
      String s = sc.nextLine();
      if(s.charAt(0) == 'q') return;
      IntSet x = new IntSet(s.substring(1));
      switch(s.charAt(0)) {
case '=': r = x; break;
case '+': r = r.add(x); break;
case '-': r = r.sub(x); break;
case '*': r = r.mul(x); break;
default: continue;
      }
      System.out.println(": " + r);
    }
  }
}
```

□ クラス側

```
class IntSet {
  int[] arr;
  int count = 0;
  public IntSet(String s) {
    try {
      StringTokenizer tok = new StringTokenizer(s);
      arr = new int[tok.countTokens()];
      for(int i = 0; i < arr.length; ++i) {
        int x = Integer.parseInt(tok.nextToken());
        if(!is_in(x)) add1(x);
      }
    } catch(Exception ex) { }
  }
  public IntSet() { arr = new int[0]; }
  private IntSet(int c) { arr = new int[c]; }
  private void add1(int x) {
    if(count+1 >= arr.length) {
      int[] a = new int[arr.length*2 + 1];
      for(int i = 0; i < count; ++i)
        a[i] = arr[i];
      arr = a;
    }
    arr[count++] = x;
  }
  public int size() { return count; }
  public boolean is_in(int x) {
    for(int i = 0; i < count; ++i)
```

```
      if(arr[i] == x) return true;
    return false;
  }
  public IntSet add(IntSet r) {
    IntSet s = new IntSet(size() + r.size());
    for(int i = 0; i < count; ++i) s.add1(arr[i]);
    for(int i = 0; i < r.count; ++i)
      if(!is_in(r.arr[i])) s.add1(r.arr[i]);
    return s;
  }
  public IntSet sub(IntSet r) {
    IntSet s = new IntSet(Math.max(size(), r.size()));
    for(int i = 0; i < count; ++i)
      if(!r.is_in(arr[i])) s.add1(arr[i]);
    return s;
  }
  public IntSet mul(IntSet r) {
    IntSet s = new IntSet(Math.max(size(), r.size()));
    for(int i = 0; i < count; ++i)
      if(r.is_in(arr[i])) s.add1(arr[i]);
    return s;
  }
  public String toString() {
    if(count == 0) return "{ }";
    String s = "";
    for(int i = 0; i < count; ++i)
      s += " " + arr[i];
    return "{" + s.substring(1) + "}";
  }
}
```

- Java では配列は常に「別オブジェクトへの参照」になる
- 必要な容量を見積もってその大きさで用意し、容量が足りなくなったらより大きいものを用意して差し替え。
- 使わなくなった古い配列はごみ集めにより回収

1.14 C++のコピーコンストラクタと代入演算子

□ 整数の集合:C++版でも配列が足りなくなった場合に対処するには?

```
Intset::Intset(int c) {
  count = 0; limit = c; arr = new int[c];
}
Intset::Intset() {
  count = limit = 0; arr = new int[0];
}
Intset::~Intset() { delete[] arr; }
void Intset::add1(int x) {
  if(count+1 >= limit) {
    int *a = new int[limit*2+1];
    for(int i = 0; i < count; ++i)
      a[i] = arr[i];
    delete[] arr; arr = a;
    limit = limit*2+1;
  }
  arr[count++] = x;
}
```

- 初期化時に配列サイズを決めて割り当てる。
- デストラクタが必要に (このオブジェクトが不要になったら指している配列も不要になるから。
- add1() で配列が満杯になったら増やす。

□ クラス定義部分のコードを示す:

```
class Intset { // set of ints
  int count, limit;
  int *arr;
  Intset(int c);
  void add1(int x);
  static int min(int x, int y) { return (x<y)?x:y; }
public:
  Intset();
  Intset(const Intset &s);
  ~Intset();
  Intset& operator=(const Intset& s);
  int size() const;
  bool is_in(const int i) const;
  Intset operator+(const Intset &s) const;
  Intset operator-(const Intset &s) const;
  Intset operator*(const Intset &s) const;
  friend ostream& operator<<(ostream& o, Intset& s);
  friend istream& operator>>(istream& i, Intset& s);
};
```

- ヘンなコンストラクタと演算子が増えている?

□ QUIZ: あるクラス Someclass の変数 x、y があるとする。
□ 次の 2 つは同等だと思う? YES/NO

```
(A) Someclass x = y;  (B) Somclass x;
                          x = y;
```

□ 前問の答え: 「初期化」と「代入」は別物

- 初期化は「ゼロから内部データ構造を作る」
- 代入は「できているデータ構造に格納する」

□ 次の場合は初期化か代入か?

```
(A) int f(Someclass x) (B) Someclass f() {
    ...                        ...
    f(y) ←呼び出し           return y; ←返値
                             }
```

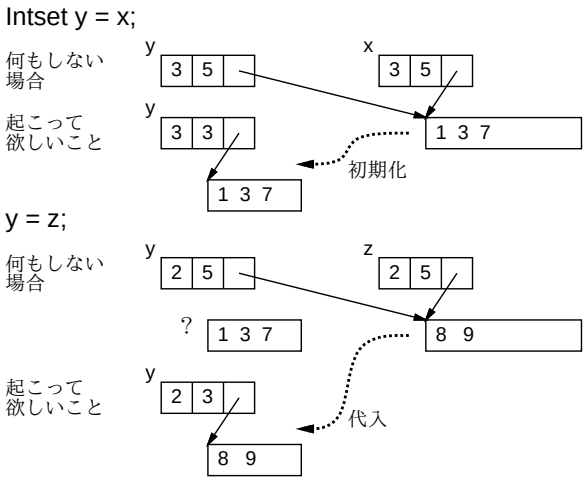
□ 前問の答え: いずれも「初期化」。引数は実引数のコピーで初期化される。返値は返値用のテンポラリが return の式で初期化される。
□ 「初期化」も「代入」も内部データ構造に応じたものを用意しないとまずい*場合がある*→「コピーコンストラクタ」と「代入演算子」

```
  Intset(const Intset &s);
  Intset& operator=(const Intset& s);
```

- 指定しない場合は「各メンバごとのコピー」が用意される。それだとどういう場合にまずいか分かりますか?

□ 通常のコピー (引数渡し等)、代入動作→オブジェクトのそれぞれのインスタンス変数をコピー。先のバージョンではこれでよかったが…

- 変数 arr をコピーする→ポインタのコピーであって配列はコピーされない (共有された状態になる) →それでは困る (副作用とか)。
- このため、「コピーコンストラクタ」と「代入演算子」を作成して、その中で arr が指す配列をコピーする。



□ 具体的なコード (この種のものとしては典型的)

```
Intset::Intset(const Intset &s) {
  arr = new int[s.count];
  limit = count = s.count;
  for(int i = 0; i < count; ++i)
    arr[i] = s.arr[i];
}
Intset& Intset::operator=(const Intset& s) {
  if(this != &s) {
    count = 0;
    for(int i = 0; i < s.count; ++i)
      add1(s.arr[i]);
  }
  return *this;
}
```

- 代入の場合は「自分への代入は何もしない」を入れておくのが通例。

1.15 書き換え可能/書き換え不能

□ 書き換え不能 (immutable) →オブジェクトの状態が変化しないこと

- Rational オブジェクトも Intset オブジェクトも内部的には変数を持っていて値が変化することは可能だが、外からメソッド経由で使っているぶんには値を変化させることはできない (論理的な不変性)
- 書き換え不能だと副作用の心配がない
- その代わりちょっとでも変更したいときは新しいオブジェクトを作ることになる

□ 書き換え可能 (mutable) →オブジェクトの状態が変化すること

- 例えば先の整数集合でも「add1(x) で直接追加」を公開にすると書き換え可能になる
- 配列オブジェクトなどが書き換え可能なものの典型例
- CLU など「書き換えられない配列」を持つ言語もあった
- 書き換え可能だと変更する時の効率はよい
- その代わり思わぬ副作用が起きないか注意する必要がある

□ 自分がクラスを作る時はどちらにするか十分検討して考える

□ 「論理的には不変」(外からオブジェクトを使っている限りは副作用がない) でも、「物理的には可変」ということがある。

- 例: 文字列表現を常に作るのでは無駄だから最初に必要とされた時に作り、一度作ったら覚えておいて再利用。

□ C++では const を指定することで「不変」を明示できるが指定は厄介

```
const int i = 100; // i は不変
const int *p = &i; // p は不変だが*p は書ける
int *const p = &i; // p は書けるが*p は不変
const int *const p = ... // 両方不変
int size() const;  // メソッドは副作用を持たない
```

□ 「意味的には不変だが内部的には可変」なことも…→何らかの工夫が必要

- 例: 書き換え対象となる一部の変数に「mutable」と指定。
- 例: 書き換え対象となる部分は別オブジェクトにする。

1.16 例外処理

□ 最近の言語で加わった新しい制御構造 (Lisp などでは古くからある)

□ もともとの問題: エラー検出はどうやるのがいいのか?

```
status = some_func(...);
if(status != OK) {
  エラー処理…
}
```

- どういう問題があるか?

□ 返り値が使えなくなってしまう

□ 広域変数にするとマルチスレッドとか問題

□ エラー処理が各所に挿入されると処理の流れが見づらい

- どこか 1 箇所で処理したいが goto はあまり使いたくない
- エラーに飛び出したために後始末が飛ばされたりすると困る

□ エラー処理のための制御構造を導入→「例外」処理

```
try {
  ... 何かあるかも知れない
  ... 処理をいくらでも書く
} catch(Exception ex) { ←エラー情報受け取り
  ここでまとめて処理
}
```

□ 例外を発生させる側は…

```
throw new 例外クラス名 (...); //Java
throw 例外クラス名 (); //C++
```

- ここまでの書き方は Java と C++でほとんど同じ。Java ではオブジェクトは全部ポインタだからこのままでいいが、C++では受け取るところをコピーを避けるため参照で受け取るのが普通。「catch(Exception& ex)」

□ 「必ずやりたい後始末」(例: ファイルの close など) はどうするか? ←途中から飛び出されるのでメソッド末尾を通過しないかも

- Java では finally ブロックにそのような処理を指定

```
try {
  ...
  try {
    ... ←この中で起きたエラーで
  } catch(NumberFormatException e1) {
    ... ←数値書式エラーはここへ来る
  } finally {
    必ずやる後始末
  }
  ...
} catch(Exception ex) {
  ... ←他のエラーは直接ここへ来る
}
```

- C++は finally がない→ローカル変数のデストラクタ等を活用

```
{ Someclass x;
  ... ←途中からどうやって飛び出したとしても
  ...
} ← Someclass::~Someclass() が必ず呼ばれる
```

□ 受け止めなかった例外は?

- メソッド呼び出し側に返される (そこに try-catch があればそこで受け止められる)
- 最後まで受け止められなければプログラム実行を中止

□ 例外種別の分類→階層構造

- Java ではすべての例外が階層構造に統一→Throwable を受け止めれば必ずすべてが受け止められる

```
Throwable ←例外すべて
  Error ←システム上の問題
    OutofMemoryError
    ...
  Exception ←普通の例外
    IOException
    InterruptedException
    RuntimeException ←どこでも起きるような例外
      NumberFormatException
      NullPointerException
      ...
```

- C++では言語上は例外オブジェクトは何でもよい。すべてを受け止めたい場合は「catch(...)」というへンなものを使う。

□ メソッドがどのような例外を返すかはコンパイラがチェック可能。

- いずれもメソッド引数の後に、C++は「throw (名前, …)」Java では「throws 種別, ...」という形。
- C++では「書かないと何でもよい」「受け止めなくてもよい」
- Java では「原則として自分が発生させる例外は明示」「自分が呼ぶメソッドで発生する例外は原則として受け止めて処理」
- Java で受け止めないでもいいのは自分も同じ例外を throws で明示している場合のみ。ただし、すべてのメソッドは「throws Error, RuntimeException」ははじめから指定されているものと見なされる

1.17 この節のまとめ

□ C++は C を土台にオブジェクト指向機能を追加して作った言語

- ポリシーとして「C と同程度の実行効率」「多様な選択肢」
- このため「プログラマが責任を持つ」ことは多め。複雑。

□ Java は C++のアンチテーゼとして「安全優先」で作られた言語

- このため「最初から危ないことはできない」言語設計。効率は犠牲に。

□ いずれも新しい制御構造として「例外」機構を導入している。

□ ここまででは抽象データ型 (ADT) としてのクラスの利用に限定して説明した (これだけでも十分重要だし複雑)。

2　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

2.1　オブジェクト指向の定義

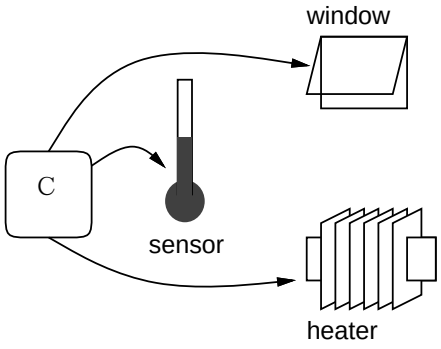
□ オブジェクト指向とは、プログラムが扱う対象のそれぞれを自立した「もの」として扱う「考え方」

□ 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

2.2　オブジェクト指向的な考え方

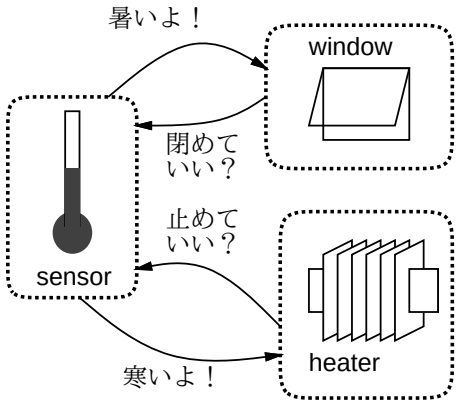
□ たとえば、「温室の温度調節システム」を考える。



□ 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)。

2.3　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

3 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。

3.1 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

3.2 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)

3.3 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)

3.4 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。

3.5 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>
```

3.6 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; }
```

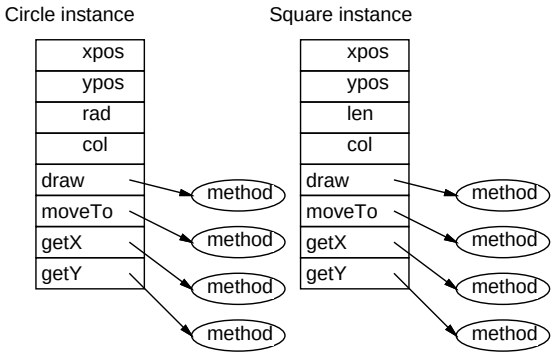
```
  public int getY() { return gy; }
}

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)

3.7 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

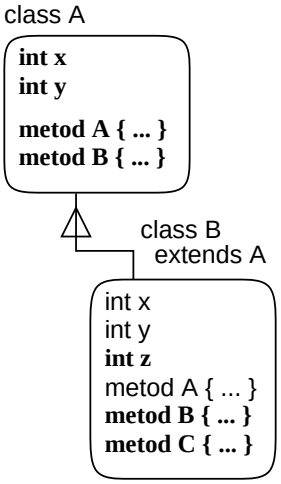
□ ただし、そのようなことをやる場合は「ポインタ型を」使う必要がある。なぜか分かりますか?

□ (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

3.8 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) { ... }
}
```

3.9 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);
  }
}
```

3.10 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

3.11 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。

4 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/)

4.1 「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

4.2 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

4.3 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)

4.4 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)

4.5 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

4.6 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)

4.7 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

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

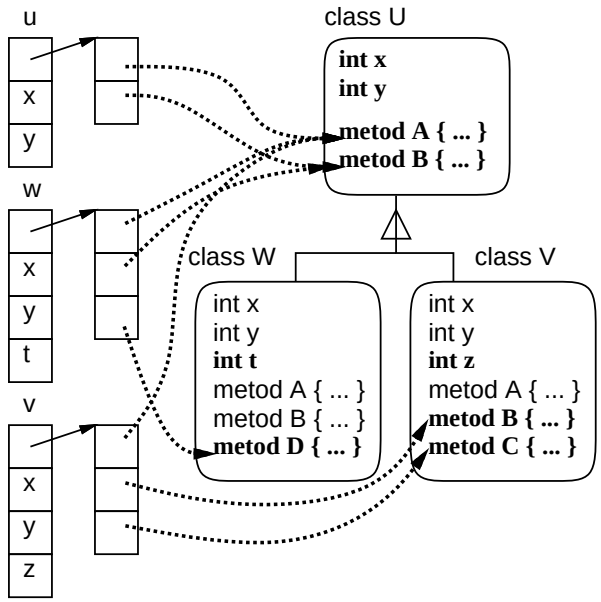
□ 継承がやってることは何かというと…

- 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++)

9 演習問題

□ 下記 (1)〜(4) のうちから 1 つ以上を選び実験を行い、結果をレポートせよ。やったことの説明や書いたコードだけでなく、計測の場合は方法まで説明し、きちんと考察まで書くこと。言語は特に指定していない場合、C++ でも Java でもよい (両方でやって比べてみるとなおよい)。期限はリーダから指定。

□ (1) Rational オブジェクトや Intset オブジェクトの例題を動かし、その「足し算」などの演算が int や double などの数値の演算の何倍遅いか計測してみよ。計測する前に予測し、予測がどれくらい合っていたか検討すること。(予測はあてずっぽうではなく、何らかの根拠が必要)。

- ヒント: C++で時間計測をするには標準関数 clock() を利用するとよい。

```
#include <ctime>
...
clock_t t1 = clock();
計測したい処理
clock_t t2 = clock();
double sec = double(t2-t1)/CLOCKS_PER_SEC;
       ↑秒単位に換算
```

- ヒント: Java の時間計測には System クラスのクラスメソッド System.currentTimeMillis() を利用するとよい。

```
long t1 = System.currentTimeMillis();
計測したい処理
long t2 = System.currentTimeMillis();
long dt = t2 - t1; ←所要時間 (msec)
```

□ (2) 例題の RPS クラス、Rational クラス、Intset クラスと同様な抽象データ型を自分でも作り、「電卓」部分も合わせて直して動かせ (たとえば「複素数」「ベクトル」「方位」「時刻」など自由に考えてよい)。どんな演算やメソッドを用意するかも自分でデザインすること。なぜそうデザインしたかも報告する。

□ (3) 例題の Intset クラスはプログラム内で生成した整数データをもとに集合を作る方法を提供していない。これを追加せよ。次に、2 つの集合の大きさ M、N と和集合や積集合を計算するメソッドの実行時間との関係を、ある程度大きな M、N で複数通り試してグラフに描け。この実行時間を高速化する方法を考えて実装できるとなおよい。

- 注記: 実験用に乱数が必要なら「じゃんけん」の例題を参照。
- ヒント: 配列 arr 内で整数が大小順に整列されるようにしておくと、和集合や積集合の計算が速くできる。
- ヒント: 別の方法として、配列に順に詰めるのではなくハッシュ表などを使ってもよい。API ドキュメントで調べて Java 標準ライブラリに含まれるハッシュ表のクラスを活用してもよい。

□ (4) 2 つの Intset どうしが同じ集合であるかどうかを判断するメソッドを実装せよ。また、その性能について前 2 問と同様に所要時間のモデルを考え、計測により確認し、さらに改良を試みよ。

- 注記: 前 2 問で行なった改良が含まれた状態でやってもよい。
- ヒント: 集合に含まれる文字列群全体のハッシュ値を計算し、集合が異なればハッシュ値も異なるようにしておくと、異なる場合の判断はすごく速くできる。

□ (5) 「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。

□ (6) 「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。)

□ (7) 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。

□ (8) その他、Java または C++を用いて「オブジェクト指向言語らしい」プログラム (アプレットであってもなくてもよい) を作成せよ。