プログラミング言語論'10 ＃4

久野　靖*

2010.5.6

はじめに

□ 前回やったこと

- オブジェクト指向の考え方
- 継承、動的分配などの重要な概念
- 基本的な実装方式
- 別のオブジェクト指向言語: Dolittle

□ 今回やること

- Lisp 系のオブジェクト指向言語
- オブジェクト指向と型
- 継承と委譲
- プロトタイプ方式 (Dolittle でやった…)
- (Java の入れ子クラスと内部クラス) ←説明省略?
- 実例: Clojure

1　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)

1.1　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

1.2　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

1.3　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

*筑波大学大学院経営システム科学専攻

- 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)

1.4 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

1.5 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

2 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)

2.1 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

2.2 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?

2.3 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)

2.4 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";
}
```

2.5 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

2.6 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…

2.7 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)

3 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

3.1 継承の意味づけ

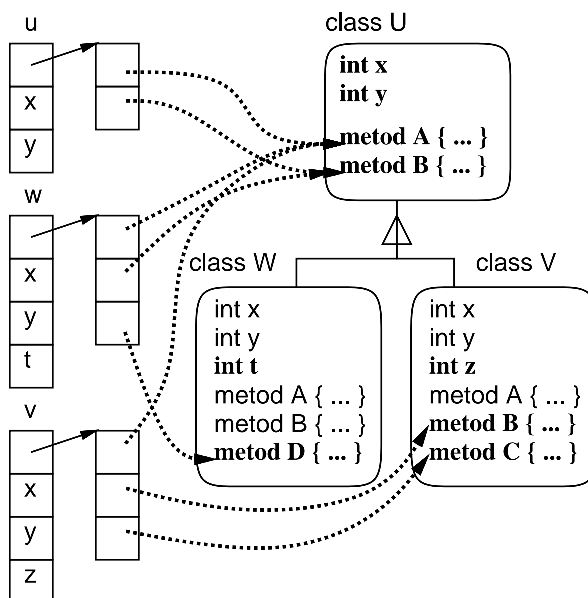
□ 継承がやってることは何かというと…

- 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

3.2 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

3.3 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

3.4　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。

3.5　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

3.6　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

3.7 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)

3.8　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

4　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

4.1　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)

4.2　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

4.3 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

4.4 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++)

5 実例: Clojure

☐ Lisp ベースの言語

- Lisp、記号計算
- JVM に強く統合→ Java API を自由に使用可能
- 関数型 (普通の Lisp と違い副作用は非常に限定)
- 遅延評価
- 並行処理に対する対応

5.1 Clojure 入門

☐ Read-Eval-Print-Loop (REPL) から使う

- Read: 式を読む。Eval: 式を評価 (実行)。Print: 結果を打ち出す。Loop: 以上 3 つを無限に繰り返す。
- 動かし方: コマンド「clojure」→プロンプト「user=>」が出る
- 止め方: Control-D を打つと終わる
- 打ち込むもの: Lisp の式
- 基本形: 「(関数名 パラメタ…)」例: (+ 1 2 3) → 6
- カンマをつけてもよい (C) 例: (+ 1, 2, 3) → 6

☐ 基本的な型と評価方法

- 数値や文字列は評価してもそのまま。 3 → 3、"abc" → "abc"
- 単一の文字は「\文字」。
- 名前は「記号 (symbol)」。記号は値を持てる。「(def x 1)」
- 記号を評価すると記号が持っている値になる。x → 1
- リスト: 「(…)」。リストは「並び」だがプログラムを表すのにも使う。
- リストは関数呼び出しとして評価される。(* 1 2 3) → 6
- 記号やリストを「そのまま」(評価せずに) 使いたい場合は「'」を前置。'x → x、'(1 2 3) → (1 2 3)、(first '(1 2 3)) → 1、(rest '(1 2 3)) → (2 3)
- 「'…」は「(quote …)」の短縮記法 (読み込み時に展開)。''x → (quote x)
- 「:」で始まる名前は「キーワード」。評価するとそのまま。プログラム上のさまざまな指定のために使うことが多い。:a → :a
- [値…] --- ベクタ。リストとは別種の「並び」。

☐ true、false、nil --- 真、偽、「ない」ことを表す値。nil と '() は別のもの (C)。

- 「?」で終わる関数は述語 (最近の流行) --- (nil? nil) → true, (nil? '()) → false, (even? 10) → true, (char? :a) → false
- if などで論理値として使う時は「false」「nil」は偽、それ以外は真として扱われる
- {キー 値 キー 値 …} --- マップ、# {値…} --- セット。これらは関数としても使える

```
(def s1 #{:a :e :i :o :u})
(s1 :a) → :a; (s1 :x) → nil
(def s2 {:a 1 :e 2 :i 3 :o 4 :u})
(s2 :e) → 2; (s2 :x) → nil
```

☐ 関数定義 --- (defn 名前 [パラメタ] 本体)

```
(defn fact [n] (if (< n 1) 1 (* n (fact (- n 1)))))
```

パラメタをベクタで表現するところは変わっている (C)

- 上記は if による枝分かれ。多方向の枝分かれは cond だが、普通の Lisp よりかっこが少ない (Lisp の「かっこのお化け」をできるだけ避けようとしている)

```
(defn test1 [x]
  (cond (> x 0) 'positive
        (< x 0) 'negative
        true 'zero))
```

- 長くなった場合は→ファイル「なんとか.clj」に入れておき「(load "なんとか")」で読み込ませる
- 無名関数は「(fn [パラメタ] 本体)」(C)

```
(filter (fn [x] (> x 2)) '(1 2 3 4)) → (3 4)
```

□ もう 1 つの関数定義の書き方として、パラメタ個数が異なる複数のケースを選択する形で定義するものがある。

```
(defn test ([] 1)
          ([x] (inc x)))
```

□ パラメタの「ある位置以降」をリストとして受け取る仕組み (L)

```
(defn sum [& l] (apply + l))
(sum 1 2 3) → 6
```

□ 再帰関数

- 普通の関数型言語処理系だと→単純再帰はループに翻訳される

```
(defn nums
  ([n] (nums n []))
  ([n r] (if (zero? n)
             r
             (nums (dec n) (conj r n)))))
```

- このように 1 引数と 2 引数を分けて扱える→補助関数を別にしないでもすむ
- しかし! Clojure は JVM に頼っているので単純再帰を自動的にループにはできない→上記の定義は「10000」とかでも動かない (スタックオーバフロー)
- recur を使って「これは再帰だよ」と明示すればループとして動作

```
(defn nums
  ([n] (nums n []))
  ([n r] (if (zero? n)
             r
             (recur (dec n) (conj r n)))))
```

- 次のような「素朴な」版だと recur に書き換えられない (前の値を取っておいて再帰から戻ってきてから追加する必要)

```
(defun nums [n]
  (if (zero? n) [] (conj (nums (dec n)) n)))
```

- loop という形式を使うと関数の先頭でなく途中の位置に recur で戻れる

```
(defun nums [n]
  (loop [x n r []]
        (if (zero? x)
            r
            (recur (dec x) (conj r n)))))
```

- この loop では変数を書き換えてはいないが、次々に書き換えているかのようなイメージでプログラムすることが可能

5.2 シーケンス

□ 「並んだもの」(リスト、ベクタ、ファイル、ディレクトリ、パターンマッチ結果、…) → すべて「シーケンス」として統一

- すべてのシーケンスを操作する共通の関数

```
(first '(1 2 3)) → 1
(rest '(1 2 3)) → (2 3)
(cons 1 '(2 3)) → (1 2 3)
```

- シーケンスに「都合のいいように」要素 (群) を追加する関数

```
(conj '(1 2 3) 4 5)) → (5 4 1 2 3)
(conj [1 2 3] 4 5)) → [1 2 3 4 5]
(into '(1 2 3) '(4 5)) → (5 4 1 2 3)
(into [1 2 3] [4 5])) → [1 2 3 4 5]
```

- シーケンスの具体的な型は様々だし、操作によって変わる
- 静的な型検査のない言語→「使い方」さえ合っていればどの型でも動く
- こういうのを「Duck Typing」と呼ぶこともある (強い型でも可能という説もあり)

5.3 遅延評価と遅延シーケンス

□ 遅延評価 (lazy evaluation) --- 式の値を「必要になった時はじめて」評価すること

- 対義語: 先行評価 (eager evaluation)。普通の言語はこれ
- (def x (+ 1 y)) --- 通常 (E):1 と y を計算して x に束縛。遅延 (L): x に「この式を」保持。x の値を取り出そうとした時に計算
- 遅延評価の利点 --- 必要の無い計算は行わないで良い、複数回同じ式を計算しても最初の 1 回だけ計算、無限の長さのデータ構造を扱うことができる
- 副作用のない言語では、いつ評価しても値は同じ (参照透明性) →遅延評価が導入しやすい
- 例: Haskell --- 「すべて遅延評価」な言語

□ Clojure は…「ある特定の」データ構造の中だけ遅延評価。具体的には一部のシーケンスの中身

- 例: (iterate f i) : (i (f i) (f (f i)) …) を計算 (無限列)

```
(take 10 (iterate inc 1))
→ (1 2 3 4 5 6 7 8 9 10)
(defn twice [x] (* x 2))
(take 10 (iterate twice 1))
→ (1 2 4 8 16 32 64 128 256 512)
(nth (iterate twice 1) 10)
→ 1024
```

- 必要になった時計算される→無限で構わない

- take や nth を使って一部を取り出して打ち出す (そのままだと全部打ち出そうとするので…)
- 頭を押えないようにする (不要な部分を持ったままだとメモリを圧迫)

```
(nth (iterate inc 1) 100000000) : 無問題
(def l (iterate inc 1))
(nth l 100000000) : メモリ不足で死ぬ
```

□ 遅延シーケンスを自分で作る: (lazy-seq 本体)

- 例: フィボナッチ数

```
(defn fib
  ([]
    (concat [0 1] (fib 0 1)))
  ([x y]
    (let [z (+ x y)]
       (lazy-seq (cons z (fib y z))))))
```

- 上記はフィボナッチ数列を返す関数

```
(take 20 (fib))
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
 610 987 1597 2584 4181)
```

□ まとめ: 遅延評価を使うことで「いくつまで」を気にしなくて済む (必要ならその要素を取ろうとすると計算される)

5.4 並行制御

□ 今日の CPU→単一コアの速度向上は頭打ち、マルチコア化によるスループットの向上が主流に

- プログラム内でも並列性がないとマルチコアの恩恵が受けられない
- 多くの言語/システムではマルチスレッドによる並列性の記述
- 並列プログラムの難しさ→スレッド間で共有されるデータの協調的な更新

□ 伝統的な並行制御のツール: ロック

- ロックを取れるのは一人だけ: 「ロックを取る→排他的にデータを参照/更新→ロックを解放」

□ ロックには多くの問題点

- 誰かが間違ってロックを取ったままになると→すべて止まってしまう
- 誰も間違っていなくても取り方が悪いとデッドロック
- 「ある条件が成り立つまで待つ」機能が必要だが難しい

□ 根本的な問題: ロックによる並行制御はデータ抽象などの形でのカプセル化と直行している (カプセル化できていない)

- 久野の研究もあります…
- ここではロックのアンチテーゼとして今日注目されている枠組みについて紹介

□ メッセージパシング (アクター、エージェントなどとも呼ばれる)

- 実行主体どうしはメッセージを送り合うだけでデータは共有しない
- 共有しないから競合もしない
- すべての状態更新をメッセージで表現しないといけないのが面倒 (?)

□ ソフトウェアトランザクショナルメモリ (STM)

- トランザクション --- 一群の共有状態に対する参照/更新をグループ化したもの (データベース、OLTP から来ている): ACI(D)
- 1 つのトランザクションは外からは「一瞬」で終わる (Atomic)
- 各トランザクションは整合した状態のみを見る/見せる (Consistent)
- 各トランザクションは相互に隔離されている (Isolation)
- データベースでは「Commit したトランザクションによる更新は無くならない」(Durable) というのがあるが、プログラミング言語ではそれはなし
- トランザクションは上記の条件が破られそうになったらアボートして再試行。最後まで無事完了したものは commit

□ STM の実現方法

- 悲観的並行制御: 必要なものすべてロックを取ってから開始
- 楽観的並行制御: 各トランザクションはバージョン番号を取得して開始し、参照/更新するデータに対してバージョン番号を記録→更新そのものは手元のコピーに対して実施→最後にバージョン番号が自分の記録したもののままなら競合はないので手元のコピーを書き戻して commit
- 多くの並列実行は互いに独立→楽観的並行制御はオーバヘッドが少なく有利 (とされている)

□ STM の得失

- ロックを取ることを考えなくてよい→「必要なデータの参照/更新をすればいい」だけ→一般のプログラマ向きとの説 (cf. ロックを正しく扱うのは一般のプログラマには極めて難しいとの説)
- 並行制御だけで条件同期が無いが、「条件が合わなければしばらく待って再度実行」などの手法が提案/実装されている (Haskell STM)

- 旧来の言語で書く場合、全ての「読み/書き」を STM オペレーションとして実行するのが繁雑→新しい言語で STM が組み込まれていればとくに問題なし
- STM 操作とその他の副作用を持つ操作が両方書ける言語だと、トランザクションの再試行が外から見えて混乱を招くという説も
- 競合が激しくなった時には性能上問題があるという説も

5.5 Clojure の並行制御機能

□ atom --- 「瞬時」に終わる動作は並行制御しなくても大丈夫

- (def 名前 (atom 初期値)) --- 初期値を指定して atom 値を生成し定義
- (reset! 名前 新しい値) --- 新しい値を書き込む
- (swap! 名前 関数 引数…) --- (関数 前の値 引数…) を計算し、その結果を新しい値として書き込むとともに返値として返す

```
(def counter (atom 1))
(swap! counter inc) → 2
```

□ agent --- 後で非同期的に更新する機能

- (def 名前 (agent 初期値)) --- 初期値を指定して agent 値を生成し定義
- (send 名前 関数 引数…) --- atom の更新と同様だが後で実行
- (await 名前…) --- agent(必要なら複数) の更新完了を待つ

```
(def counter (agent 1))
(send counter inc) → agent オブジェクトが帰る
(await counter)
@counter  → 2  …「@名前」で状態を読み出せる
```

□ ref --- 参照変数 (書き換え可能な変数): トランザクションの中でのみ参照/更新が可能

- (def 名前 (ref 初期値)) --- 初期値を指定して ref 値を生成し定義
- (deref 名前) --- ref 値の内容を読み出し…「@名前」でも同じ
- (ref-set 名前 値) --- ref 値に新しい内容を書き込む
- (alter 名前 関数 引数…) --- atom の更新と同様
- (dosync 式 …) --- ref の参照/更新はすべてこの中で行う必要 (トランザクション)

□ Clojure では STM などの並行制御機能を提供

- さまざまな材料が提供されている→分かっていて混ぜるのならいいが、安易に混ぜるとはまりそう

5.6 マクロ

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

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

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

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

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

□ マクロは「Lisp プログラムを組み立てる関数」

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

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

- これを使うと先の unless は次のように書ける

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

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

□ マクロによくある問題→記号の捕捉 (capture)

- マクロの中で識別子を導入したときに起こる

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

- 普通の Lisp だとこれは次のように展開

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

- 基本的な問題: マクロの中で作業用に使っている tmp とパラメタで渡されて埋め込まれる tmp とか衝突/混同

- よくある対処法: マクロ中では「普通使わなそうな」名前を使う→根本的な解決でない
- Scheme の synatx-case --- 非常にややこしい
- Clojure のアプローチ: ランダム番号つきの名前を自動生成する

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

- 必要なら捕捉が起きるようにすることも可能

5.7 Java との連携

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

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

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

□ Java のクラスを作り出すには→ proxy

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

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

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

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

5.8 マルチメソッド

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

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

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

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

6 レポート課題

□ 出席に加えて、レポートを 1 件出していただきます。

□ 課題: 本講義で取り上げた言語、ないしその他の言語のうちから、「自分が普段使い慣れていないもの」を 1 つ選び、その言語の何らかの (自分にとって目新しい、ないし興味深い) 特徴を確認できるようなプログラムを作って解説しなさい。下記の内容が含まれること。

- 動機: どのような言語の、どのような特徴を確認しようと思ったか。また、なぜそれを選んだか。
- 方針: どのようなプログラムを作れば確認が行えると考えたか。
- 本体: 実際に作ってみたプログラム、およびその実行のようす。
- 解説: プログラムおよび実行のようすの解説。
- 考察: 実行のようすからどのようなことが分かるか。とりわけ、確認しようと思った特徴がどのように結果に反映されているか。
- 感想など。

□ 期限: 6 月末日、提出方法: メール (PDF) で久野宛送るか、または紙 (神保町 2F の久野のボックス)