TSSI 基盤技術研修コース
— プログラミング言語 — ＃3

久野　靖*

2011.6.10

1　はじめに

1.1　前回のアンケートを拝見して…

□ 前回も練習問題の時間を取ったのは好評だったようです。今回もできればやりたかったのですが、資料には間に合わなかったので…

□ オブジェクト指向についてあまり納得していなかったのが分かったというご意見もあったので、よかったと思います。よく知っている人は逆につっこみたい所もありそうですが、とりあえず分かりやすさ優先ということでやっていますので…

□ グラフィクス関係の例題は Java のみでしたが、それはそれでよいという感じのようでした。

1.2　今回の内容

□ オブジェクト指向言語およびプログラミング言語全般で周辺の話題

- JavaScript
- 総称的プログラミング (テンプレート、型パラメタ)
- 自己反映機能 (時間があれば)
- 関数型プログラミング言語 (時間があれば)

2　JavaScript - 弱い型の OOPL

□ スクリプト言語

□ 弱い型のオブジェクト指向言語

□ プロトタイプ方式

□ 組み込み型処理系

2.1　スクリプト言語

□ って、聞いたことありますか?　何を意味すると思う?

□ たとえば…

- (答 1) コマンドを並べたもの←シェルスクリプト?
- (答 2) いい加減な/適当な言語←そんなの目標にするか?
- (答 3) インタプリタ方式の言語← Lisp なんかもそうなの?
- (答 4) ささっと書いて動かせる言語←コンパクト、インタプリタ
- (答 5) くっつける (glue) 言語←組み込み型

□ スクリプト言語世代説

- 第 1 世代→シェルスクリプト、Awk →コンパクト、適当に書ける
- 第 2 世代→ Perl →それ 1 つで結構何でも書ける (汎用)、自立した言語。ただしキタナイ
- 第 3 世代→ Python、Ruby →言語デザインを整理、オブジェクト指向
- JavaScript も第 3 世代でいいと思うが…

2.2　JavaScript の由来

□ Netscape Navigator 2 →当時の「最先端の」ブラウザ

- Netscape 社はブラウザにスクリプト言語を組み込んで様々なことができるようにしようとした
- 言語は当初「LiveScript」という名前で開発→おりからの Java ブーム→ Sun にお金を払って「JavaScript」にした
- 文法はできるだけ Java そっくりにしたが、言語としてはまったく別の言語。よく JavaScript と Java を混同している人がいるので迷惑
- ブラウザ組み込みのスクリプト言語として「事実上の標準」
- MSIE もこれと互換の「JScript」を搭載

□ 標準化の必要性→ ECMA(欧州情報通信システム標準化機構) による標準化→ ECMA-262 規格として成立

- その後、ブラウザ組み込みだけでなく Flash のスクリプト言語 (AcitonScript)、MS ASP(Active Server Page) のスクリプト言語としても採用

*筑波大学大学院経営システム科学専攻

- ECMA によるコア部分は共通だが組み込まれている追加機能はそれぞれの環境で異なる

□ 以下では (当然) ブラウザ組み込みの JavaScript を扱う (どこにでもあって簡単に試せる)

2.3 JavaScript 入門

□ HTML の説明はしませんので…

□ JavaScript コードを書ける場所…HTML ファイル中の次の 3 箇所

□ その 1: script 要素 (埋め込み、別ファイル)

```
<script type="text/javascript">
コード...
</script>

<script type="text/javascript" src="ファイル">
</script>
```

- この場合、コード中で「document.write(...)」を実行して出力したものは HTML 中のその場所 (script 要素のある場所) に埋め込まれる

```
<pre><script type="text/javascript">
document.writeln(' 乱数: ' + Math.random());
</script></pre>
```

□ その 2: HTML タグの「onmouseover="..."」(マウスが上に乗ったら〜) 「onclick="..."」(クリックしたら) などの指定 (イベントハンドラ) の中

- 長いものは書きづらいのでよそに関数を定義して呼ぶことが多い

```
<p onmouseover="window.alert('　乗った ね')">ここ</p>
```

□ まとめると…

- 常に実行するもの (関数定義、ページ中への埋め込み) → script 要素
- ユーザの操作に対応するもの→イベントハンドラ

```
<html>
<head>
<title>文書のタイトル…</title>
<script type="text/javascript">
  常に実行するコード (関数定義等)
  ...
</script>
</head>
<body>
ページ内容…
<script type="text/javascript">
  document.write() を使うようなコード…
</script>
... onclick="..." ...
</html>
```

2.4 JavaScript の特徴

□ 制御構造の構文は Java にソックリ

- while、for、do-while、if、try-catch
- 文字列との連結の「+」も同じ

```
var x1 = 1, x2 = 1;
while(x1 < 50) {
  document.writeln(x1);
  var z = x1+x2; x1 = x2; x2 = z;
}
```

□ オブジェクト指向言語

```
o = new Object();
document.writeln("...");
```

□ 弱い型→型宣言はない、変数に型がない

```
var x = "ABC", y = 1;
```

- 文字列/文字の区別はなく「'x'」でも「"x"」でも同じこと

□ 「値」と「オブジェクト」の 2 種類がある (Java と同じ)

□ 「値」は数値型 (1 つだけ)、論理値型 (Java と同じ)、文字列型 (文字型も兼ねる。Java では文字型は値、文字列はオブジェクト)

□ 3 種類の「値」それぞれに対応するオブジェクト種別もある (Java と同じ)

□ 値に対してメソッドを呼ぼうとすると自動的にオブジェクトに

```
var n = 3.141592;
document.writeln(n.toExponential(4));
```

□ 関数 (function) がある、関数もオブジェクト

□ 関数リテラルがある。下の 2 つは同じこと

```
function add(x, y) { return x+y; }
var sub = function(x, y) { return x-y; }
document.writeln(add(3, 5) + ', ' + sub(3, 5));
```

□ 関数オブジェクトはクロージャ (まわりの環境を一緒に持つ)

```
function test(n) {
  return function() { return ++n; };
}
var f = test(5), g = test(10);
document.writeln(f() + ' : ' + g());
document.writeln(f() + ' : ' + g());
document.writeln(f() + ' : ' + g());
```

□ オブジェクトはすなわち連想配列 (任意の型の値を添字として取れる配列)

□ オブジェクトのプロパティ (フィールド) は連想配列の要素と同じ

□ 一般に「x.y」と「x['y']」は同じものを指す

```
var x = new Object();
x['a'] = 10; x['bc'] = 11; x[100] = 12;
x.b = 13; x.de = 14;
for(var i in x) document.writeln('x['+i+'] == '+x[i]);
```

2.5 JavaScript のオブジェクト指向機能

□ オブジェクトのプロパティとして関数を入れる→メソッド

□ メソッドの中ではオブジェクトを「this」で参照できる

```
var o = new Object();
o.count = 100;
o.show = function(msg) {
  document.writeln(msg + this.count);
}
o.show('My number is: ');
```

□ コンストラクタも実はただの関数だが、「new 関数名(...)」の形で呼び出すと中で this が使える

```
function Counter(n) {
  this.count = n;
  this.add = function(n) { this.count += n; }
  this.getCount = function() { return this.count; }
}
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

□ これで Java と遜色なくオブジェクト指向…だと思う?

□ ちゃんとあるもの

- オブジェクト、インスタンス変数 (プロパティ)、メソッド
- 同種のオブジェクト、コンストラクタ←かなり ad hoc
- メッセージ送信記法
- 動的分配←型がないので簡単

□ 足りないと思うもの…

- クラス→クラスは必要なのか? (単なる構文単位?)
- カプセル化→確かに保護は「全然」ない→スクリプトだから? (ブラウザと組み合わせた場合、ブラウザ側での保護はアリ)
- 継承 (のようなもの) →以下で説明

2.6 プロトタイプ方式オブジェクト指向言語

□ オブジェクトの種類 (形) を表す (定義する) やり方→おもに 2 通り

- 「こういう種類のオブジェクトはこういう形」という定義 (==クラス) を書く→クラス方式
- 必要なプロパティ、メソッドを持つオブジェクトをまず作り、あとはそれをコピーする (概念的には) →プロトタイプ方式
- 実際には全部コピーすると領域が無駄なので、プロトタイプ (親) へのポインタを持っておき、「自分が持っていないものは親が持っているものを使う」ようにする

□ JavaScript ではどうなっているか…

- 「new 関数 (...)」によってオブジェクトが作成されるときに、
- その「関数」に prototype というプロパティが定義されていると、
- その値が作成されたオブジェクトの「親」としてセットされる
- (実際には関数を作ると prototype プロパティには「new Object()」の結果が自動的にセットされる→そこに色々設定すればよい)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3), c2 = new Counter(5);
c1.add(2); c2.add(2);
document.writeln(c1.getCount() + ', ' +
                 c2.getCount());
```

□ プロトタイプ方式の特徴…

- いつでも親にメソッドを追加したりしていじることができる (単なるオブジェクトだから)
- 場合によっては親を指しているポインタをとり替えることで、オブジェクトをまったく別物に「変身」させられる (ECMA-262 では不許可、ただし一部の実装ではこれを許している)

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
var c1 = new Counter(3); c1.add(4);
Counter.prototype.sub =
  function(n) { this.count -= n; }
c1.sub(2);
document.writeln(c1.getCount());
```

□ 継承みたいなのはどうするの?

- 親 (プロトタイプ) を探して見つからない場合はさらにその親を探しに行く。
- 例: Counter オブジェクトを継承してメソッド sub を追加するには…

```
function Counter(n) { this.count = n; }
Counter.prototype.add =
  function(n) { this.count += n; }
Counter.prototype.getCount =
  function() { return this.count; }
function ExCounter(n) { this.count = n; }
ExCounter.prototype = new Counter(0);
ExCounter.prototype.sub =
  function(n) { this.count -= n; }
var c1 = new ExCounter(7);
c1.add(4); c1.sub(2);
document.writeln(c1.getCount());
```

□ プロトタイプ方式の特徴をまとめると…

- 実行系のデザインはわりとシンプルにできる

- コンパイラよりインタプリタ向き。動的
- その分、何をやっているのか分かりにくい
- (きちっと構文を作って行くとクラス方式みたいに…)

2.7 組み込み型スクリプト

□ スクリプト言語の 1 つの目的→組み込み型処理系

- 大きなアプリケーションがあったとして、それをカスタマイズすることを考える→どうしますか?

□ 沢山パラメタがあって、それを設定して調整すればいいか?

□ パラメタは静的

- 「こういう場合はこう」というのが書きにくい
- 「動き」はつけにくい

□ →パラメタを設定する代わりに「プログラム」を設定する

- →そのプログラムが動いてさまざまな動作をすればよい
- →組み込み型スクリプト
- (例は既に出て来た…「マウスが乗ったら〜をする」)

□ 組み込み型スクリプトに使われる言語…

- tcl →もともとこのような用途のために作られた。しかし言語仕様があまり美しくないので普及はいまいち (tcl/tk としてだけ有名)
- scheme →もともとは Lisp 系の汎用言語だが、コンパクトな処理系が作れるので組み込みスクリプト用としても使われるように。gwm(ウィンドウマネージャ)、gimp(画像加工ソフト)
- ECMA-262(JavaScript) →元はブラウザ用の組み込みスクリプト言語だが、他の用途にも進出 (Flash ActionScript など)

□ JavaScript の場合、オブジェクト指向ならではの利点→具体的には?

□ アプリケーション自体および、アプリケーションが扱うデータ→オブジェクトであると考えることができる

- オブジェクトを自由に操作するには、オブジェクト指向言語がよい (アタリマエ)
- これまでと違うところ→オブジェクトは既に大量に (アプリやそのデータとして) 存在している→既存のオブジェクトを操るのが主目的
- 従来のプログラムでは自前でオブジェクトを設計したり生成したりしてからそれを使っていた→大きな変化 (どんな?)

□ できあいのオブジェクトを操作する場合の考慮点

- 自分が設計したデータ構造でないのでよく分からない
- 向こう側はチェックして例外を投げる側、こちらはだましだまし使う側
- きちんと型を書くのが煩雑 (しかし型検査はあった方がよいと思うが…)

□ 具体例は次の DOM で

2.8 Document Object Model (DOM)

□ DOM →ブラウザ内のページをオブジェクトの組み合わせで表現

- W3C による標準化→ http://www.w3.org/DOM/
- 現在のブラウザでは DOM level 2(DOM2) が中心
- DOM2 の標準は Core、Views、Events、Style、Traversal、HTML と分かれている
- 以下では例題とともに簡単に説明

□ DOM2 では文書全体は Node オブジェクトの木構造として表される

- Node オブジェクトのプロパティ/メソッドを用いて木構造を自由に変更できる

```
<script type="text/javascript">
function rot() {
  var n = document.body.lastChild;
  document.body.removeChild(n);
  document.body.insertBefore(n,
    document.body.firstChild);
}
function dup() {
  var n = document.body.firstChild;
  var m = n.cloneNode(true);
  document.body.appendChild(m);
}
function del() {
  var n = document.body.firstChild;
  document.body.removeChild(n);
}
```

□ HTML 要素に対応するノードのスタイル→その Node オブジェクトの style プロパティに値をつけることで色々操作できる

- 位置の変更→ CSS の位置指定機能で実現できる

```
<script type="text/javascript">
var elt, time = 0.0;
function change() {
  elt = document.getElementsByTagName('h2')[0];
  elt.style.position = 'relative';
  elt.style.backgroundColor = 'yellow';
  setInterval(step, 50);
}
function step() {
  time += 0.05;
```

```
  var p = Math.sin(time) * 200;
  elt.style.left = p + 'px';
}
</script>
<button onclick="change()">Start</button>
```

□ その他できること…

- テキストノードの中の文字列を操作できる (挿入、削除、…)
- マウスイベント、キーイベントを受け止めて処理できる
- フォーム部品の内容 (値など) を操作できる (わりと昔からある機能)

□ 結局、DOM で何ができるかというと…

- ブラウザが表示している内容→内部のデータ構造が対応
- データ構造 (オブジェクト) を任意に操作→自由に内容が変更できる
- →究極のカスタマイズ (?)

2.9 本節のまとめ

□ スクリプト言語→ささっと書ける言語、glue 言語

□ JavaScript →ブラウザ内蔵むけスクリプト言語が発端

- 弱い型の言語
- プロトタイプ方式のオブジェクト指向言語

□ 組み込み型スクリプト

- ブラウザ組み込み→ブラウザの制御、DOM によるページ内容の制御

3 総称的プログラミング

□ 配列型→「int の配列」「char の配列」など型をパラメタに持てる

- 見かたを変えれば…1 つの「もの」が複数の型に対し利用できる←「総称的」(generic) という
- 組み込み演算子なども総称的。5 / 2 == 2, 5.0 / 2.0 = 2.5。
- 総称関数 … max(int,int) → int, max(double,double) → double 等。← オーバローディングである程度はできるが、有限個数
- array[int]、array[double] …型がパラメタになっている (型パラメタ) →無限の場合に対応。

□ そういうものを「自前で」(ユーザ定義で) 作れるようにするには?

- 以下で説明する Java Generics を使うには JDK 1.5 以降必要。C++ テンプレートは最近の C++コンパイラはだいたい実装ずみ。

3.1 コンテナクラス

□ C++でも Java でも配列は「個数を最初に指定」→途中で大きさが伸び縮みできるものが欲しい→ライブラリで用意。

□ Java (JKD1.4 まで) では → 例: ArrayList クラス。任意のオブジェクトを格納できる配列。

```
import java.util.*;

public class Sample31 {
  public static void main(String[] args) {
    ArrayList a = new ArrayList(); //サイズ 0
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(int i = 0; i < a.size(); ++i) {
      String s = (String)a.get(i); // 取出し
      System.out.println(s);
    }
  }
}
```

- 何か疑問な点はありますか?
- この方法の弱点は何だと思う?

□ ダウンキャスト … 「元の型にキャストし戻す」こと。

- a.add("abc") --- Object 型のパラメタに String を渡す→ OK
- String s = a.get(i) --- Object 型は String 型に入れられない。×。
- String s = (Object)a.get(i) --- ダウンキャスト:親クラスの型から子クラスの型へのキャスト。このとき「元々なに型だったか」がチェックされる。String でなかったものは String にはキャストできない (ClassCastException が投げられる)。

□ 弱点 (まずいこと) のまとめ

- a. ダウンキャストは繁雑。
- b. 実行時にチェックが必要→オーバヘッドになる (Java はそれを気にしない言語だということはあるが)。
- c. オブジェクトでないと入れられない。たとえば int は Integer クラスのインスタンスにして入れなければならない (これもオーバヘッド)。
- d. 配列だから「均一なものの並び」にしたいのに、何でも入ってしまう。
- e. 間違ってヘンなものを入れてもコンパイラは怒ってくれない。
- f. そのヘンなものが取り出されて来たところで実行時にエラー。

□ 実行してみないとエラーが出ないというのはコンパイルする言語にとっての敗北。

□ 根本的な問題…本来「型パラメタ」によって複数の型 (クラス) を使用するべきところを 1 個の ArrayList クラスで済ませていること。

□ JDK 1.5から「型パラメタ」が使えるようになった (Java Generics)。ArrayList<T>のようにパラメタは「<>」の中に指定する。T のところには任意の型 (ただしクラス) を入れてよい。

□ 動かすときは…当然、JDK 1.5 の javac を使う。逆に 1.5 の javac で 1.4 のソースを動かすときはオプション指定必要

- javac -source 1.4 Sample31.java ← JDK1.4 ソース
- javac Sample31b.java ← JDK1.5 ソース (Generics)

```
import java.util.*;

public class Sample31b {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(int i = 0; i < a.size(); ++i) {
      String s = a.get(i);
      System.out.println(s);
    }
  }
}
```

- これで先の弱点のうち a、b、e、f が解消。残る c はダメ。
- 実はこのコードを実行するとき内部的には先と同様のコードが使われている。つまり ArrayList の実装は 1 つ (均一な実装、homogeneous implementation)。だから c の制約が残っている。
- c の緩和策として、JDK 1.5 で int と Integer、double と Double 等の自動変換 (AutoBoxing/Unboxing) を追加。

```
ArrayList<Integer> a = ...
a.add(3); // JDK1.5 で○
...
int i = a.get(0); // 〃
```

□ C++では→言語設計上のポリシーとして、c は受け入れられない。また、C++では配列は「直接そのオブジェクトが埋め込まれた」ものになるので、コンテナクラスも同様でなければ受け入れられない。←フルスピードで動く C に負けないことが原則。

```
//sample31
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(int i = 0; i < a.size(); ++i)
    cout << a[i] << '\n'; // operator[]
}
```

- C++では型パラメタをソースのその位置に埋め込んで全体をコンパイルするという実装 (非均一な実装、heterogenous implementation)。だから型パラメタごとに別のコードができる。その代わり、その場展開ができる→極めて高速。

3.2 型パラメタと継承関係

□ X ⊃ Y で継承関係を表すものとする。たとえば Object ⊃ String。

□ パラメタつきクラス C<T>を考える。

□ クイズ: X ⊃ Y ならば、C<X>⊃ C<Y> だといえるか?

- たとえば、ArrayList<Object> ⊃ ArrayList<String> か? YES/NO?

□ 次のように考えるべき

- X ⊃ Y とは、X 型オブジェクトの代わりに Y 型オブジェクトを使っても OK であること (互換性があること)
- 例: 「自動車」クラスの変数に「乗用車」を格納しても OK(互換性がある)

□ ArrayList<Object> a = new ArrayList<String>; は OK?

- a.get(0) ... String が取り出されるから OK。
- a.set(0, Z) ... Object が入れられないと困るが String しか駄目!!!

□ よって先の質問の答えは「NO」であるべきなんだけど…

□ 実は配列でも同じことが起こる。

□ しかし! Java では X ⊃ Y ならば X[] ⊃ Y[] になっている

- 「X[] a = new Y[10];」が可能。そのあとで「a[0] = new X();」→ ArrayStoreException 発生

□ 一般にこういうのを contravariance/covariance problem というらしい

3.3 Java Generics の制約

□ Java Generics では実行時には型パラメタ情報がなくなっている→そのため、型検査の抜け道が起きる可能性→それを防ぐための制約

- たとえば、「パラメタ型の配列は作れない」「パラメタつき型の配列は作れない」

```
ArrayList<String>[] lsa = new ArrayList<String>[5];
// ↑これは本来は許されない
Object[] oa = (Object[])lsa; // Java では OK
oa[0] = new ArrayList<Integer>(); // !!!
...
String s = lsa[0].get(0); // 実行時例外!!!
```

3.4 イテレータ

□ コンテナの種類→可変長配列 (ArrayList<T>、vector<T>)、連結リスト、B 木、ハッシュ表、などなど。

- どれでも「順番に要素を処理する」などの操作は必要→それをどれでも同等に扱えるようにしたい (後でコンテナを取り替えたりしても利用コードは変えないで済むように)。
- 「イテレータ (iterator、反復子)」→次々に要素にアクセスしていくためのオブジェクト。

□ Java では共通の Iterator インタフェースとして規定。JDK 1.5 では Iterator<T>のように各要素の型が型パラメタになる。

```
bool hasNext() --- 終わりかどうか調べる
T next() --- 次の値 (T:パラメタ型) を取り出す
void remove() --- 現在値を削除 (今回は使わない)
```

```
import java.util.*;

public class Sample31c {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(Iterator<String>
        i = a.iterator(); i.hasNext(); ) {
      String s = i.next();
      System.out.println(s);
    }
  }
}
```

□ さらに! Iterable<E>を実装するオブジェクト X については

```
for(E v: X) { ...
```

という for ループ (for-in ループ) を書くと以下と同等に動作

```
for(Iterator<E> i = X.iterator(); i.hasNext(); ) {
  E v = i.next(); ...
```

□ たとえば上の例も ArrayList<E> が Iterable<E> を実装しているので次のように書ける。

```
import java.util.*;

public class Sample31d {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    for(int i = 0; i < 10; ++i) a.add("X" + i);
    for(String s: a) System.out.println(s);
  }
}
```

□ C++では共通のインタフェースはない (各コンテナごとに違う) が形としては次のもの。

```
*p --- 現在の要素を参照
++p、p++ --- 次の要素に進める
p == q --- 同じ要素を指すかどうか判定
```

```
//sample31b
#include <vector>
#include <iostream>

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i+9);
  for(vector<int>::iterator
      i = a.begin(); i != a.end(); )
    cout << *i++ << '\n';
}
```

- 「vectior<int>::iterator」→ vector<int>クラスにおいて typedef で定義されている iterator という型名。
- 普通の「配列要素を指すポインタ」もイテレータとして使える
- コンテナにもよるが単なるポインタではないイテレータが大半

3.5 型パラメタつきクラスを作る

□ 例: 「最後の 2 つの値を記憶するバッファ」(前回やったもの) ただし、記憶される値の型は「任意」

□ C++版 (クラス定義の中に直接コード本体も書いている。こうするとインライン展開が可能に。または従来通り分けて書いてもいいが、その場合は inline と指定。)

```
//sample32
#include <iostream>
using namespace std;

template<typename T> class last2 {
  T v1, v2;
public:
  last2(T x, T y) { v1 = x; v2 = y; }
  void put(T v) {
    v2 = v1; v1 = v;
  }
  T get() {
    T x = v1; v1 = v2; return x;
  }
};

int main() {
  last2<int> a(0, 0);
  a.put(1); a.put(2); a.put(3);
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  cout << a.get() << '\n';
  last2<char*> b("", "");
  b.put("ab"); b.put("cd"), b.put("ef");
  cout << b.get() << '\n';
  cout << b.get() << '\n';
```

```
  cout << b.get() << '\n';
}
```

□ Java 版。

```
import java.util.*;

public class Sample32 {
  public static void main(String[] args) {
    Last2<Integer> a = new Last2<Integer>(0, 0);
    a.put(1); a.put(2); a.put(3);
    System.out.println(a.get());
    System.out.println(a.get());
    System.out.println(a.get());
    Last2<String> b = new Last2<String>("", "");
    b.put("ab"); b.put("cd"); b.put("ef");
    System.out.println(b.get());
    System.out.println(b.get());
    System.out.println(b.get());
  }
}

class Last2<T> {
  T v1, v2;
  public Last2(T x, T y) {
    v1 = x; v2 = y;
  }
  public void put(T v) {
    v2 = v1; v1 = v;
  }
  public T get() {
    T x = v1; v1 = v2; return x;
  }
}
```

3.6 Standard Template Library (STL)

□ C++の標準ライブラリの土台となった、テンプレートを駆使したライブラリ群→現在では C++標準ライブラリのコンテナクラス部分のことをそう呼ぶ。重要なアイデア:

- テンプレートを使った効率のよいコンテナクラス群
- 要素アクセスはコンテナクラスのメソッドではなくイテレータ経由 (その方が高速化しやすい)
- イテレータに対して働くアルゴリズム群。find (線形探索)、sort (整列)、count(計数)、などなど。
- アルゴリズム群はテンプレート関数 (型パラメタつき関数) →高速。

□ 例: 普通の swap(C 版)

```
void swap(int *x, int *y) {
  int z = *x; *x = *y; *y = z;
}
int a[100]; ... swap(&a[i], &a[j]) ...
```

- 関数呼び出しのオーバヘッド、ポインタ経由のアクセス→遅い。しかも型ごとに用意する必要。

□ 関数テンプレート版の swap(C++版)

```
template<typename T> inline void swap(T& x, T& y) {
  T z = x; x = y; y = z;
}
float a[100]; ... swap(a[i], a[j]) ...
```

- 関数テンプレートの型パラメタは推定される (a[i] 等が float だから T は float) →繁雑な指定が不要
- 「float z = a[i]; a[i] = a[j]; a[j] = z;」がこの場所に埋められているのと同様→高速

□ アルゴリズム関数を利用すれば制御構造をこちら側で書かなくてもよくなる。たとえば for_each を使うとループが不要になる。

```
//sample33
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

template<typename T> void f(T x) {
  cout << x << '\n';
}

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i);
  for_each(a.begin(), a.end(), &f<int>);
}
```

- これで for_each の中で「f(値)」が繰り返し呼ばれる→「ベクタの各要素を出力」ができる。ループは不要。
- しかし「合計を取る」だとどうするか?

□ 関数オブジェクト… operator() を持つようなオブジェクト。先の「f(値)」というのは関数でなく operator() の呼び出しでもよいので。

```
//sample33b
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template<typename T> class sum {
  T res;
public:
  sum(T init) { res = init; }
  void operator()(T x) { res += x; cout<<res<<"\n"; }
  T result() const { return res; }
};

int main() {
  vector<int> a;
  for(int i = 0; i < 10; ++i)
    a.push_back(i);
  sum<int> s(0);
  s = for_each(a.begin(), a.end(), s);
  cout << s.result() << '\n';
}
```

- ループ周回を通じて保存されて欲しいものは関数オブジェクトのインスタンス変数にすればよい。→ ループ構造と処理の分離

3.7 テンプレートの特殊化

□ テンプレートで一般向けの実装のほかに「特殊ケース」を用意したいことがある。

- たとえば bool(0/1) のバッファだったら 1 ビットずつ入れれば 32 個の値が 1 語で入るとか…
- コンテナクラスの場合、「ポインタ用」はそれなりのポインタ用のものを使う方がよさそう…（「ポインタ用」のクラスを 1 つ用意して残りはすべてそれを呼ぶ→コードを何回も展開しなくて済む→コード量の爆発を防ぐ上で重要なテクニック）

□ そのような「特殊ケース」を別途用意できる→特殊化

```
template <typename T> class last32 {
  //一般
}
template <typename T> class last32<T*> {
  // ポインタ用
}
template <> class last32<bool> {
  // bool 用
}
```

3.8 型パラメタに対する制約

□ 例: maxbuf<T>というコンテナを作る。

- 次々に put(T x) で値を入れていく。
- T get() でその時点での最大値を取得できる。

```
//sample34
#include <iostream>

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  double a2[] = { 3.0, -2.7, 6e2, 4e1, 0.5 };
  maxbuf<double> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

□ 型 T には演算子「<」が必要。これを満たさないと…

```
//sample34b
#include <iostream>

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(max < x) max = x; }
  T get() { return max; }
};

class point {
  double x, y;
public:
  point(double a, double b)
    : x(a), y(b) { }
};

int main() {
  point a3[] = { point(0,0), point(1,1),
    point(2,2), point(3,3), point(4,4) };
  maxbuf<point> m3(a3[0]);
  for(int i = 1; i < 5; ++i) m3.put(a3[i]);
}
```

□ 上記のをコンパイルすると…

```
In method 'void maxbuf<point>::put(point)':
22:   instantiated from here
7: no match for 'point & < point &'
```

□ 確かに「<」がない、と言われる… 次のを入れればよくなるが。

```
bool operator<(point &p) {
  return x*x+y*y < p.x*p.x+p.y*p.y;
}
```

- しかし「コンパイルしてみたら無かった」では実装の中に立ち入っていてエラーメッセージとしてあまり良くないのでは…
- 実際、極めて分かりづらいエラーメッセージに遭遇することがよくある
- 本来だったら「この型パラメタにはこういう型しかあてはめてはいけませんよ」という指定ができた方がいいのでは? → 賛否両論。否定派: 指定が繁雑になり自由度が減る。ともあれ、Java Generics ではこちらの立場。

□ 上記の Java Generics 版。整数は Integer クラスを使う必要。

```
import java.util.*;

public class Sample33 {
  public static void main(String[] args) {
    int[] a1 = { 5, 3, 8, 2, 4 };
    Maxbuf<Integer> m1 = new Maxbuf<Integer>(a1[0]);
    for(int i = 0; i < 5; ++i) m1.put(a1[i]);
    System.out.println(m1.get());
    String[] a2 = { "this", "by", "foo", "z", "x" };
    Maxbuf<String> m2 = new Maxbuf<String>(a2[0]);
    for(int i = 0; i < 5; ++i) m2.put(a2[i]);
```

```
    System.out.println(m2.get());
  }
}

class Maxbuf<T extends Comparable<T>> {
  T max;
  public Maxbuf(T x) { max = x; }
  public void put(T x) {
    if(max.compareTo(x) < 0) max = x;
  }
  public T get() { return max; }
}
```

□ Comparable<T>というのは x.compareTo(T y) というメソッドを持ち、このメソッドが x と y の大小関係に応じて正/負/零を返す。

□ この方が確かにあらかじめチェックはできるが、窮屈かも…

□ あらかじめ Comparable<T>を implements しているクラスにしか使えない→自分がソースを持っていればいいが、そうでないと…

- 実は C++版にも同様の問題がある。たとえば char* の maxbuf を取ろうとすると「アドレスが最大の」ものが取れてしまう…求めていることと違う。

□ C++の場合は、単独関数テンプレートと特殊化を使えばOK。

```
//sample34c
#include <iostream>
#include <cstring>
using namespace std;

template<typename T> bool lt(T x, T y) {
  return x < y;
}
template<> bool lt(char* x, char* y) {
  return strcmp(x, y) < 0;
}

template<typename T> class maxbuf {
  T max;
public:
  maxbuf(T x) : max(x) { }
  void put(T x) { if(lt(max, x)) max = x; }
  T get() { return max; }
};

int main() {
  int a1[] = { 5, 9, 2, 7, 6 };
  maxbuf<int> m1(a1[0]);
  for(int i = 1; i < 5; ++i) m1.put(a1[i]);
  cout << "a1: " << m1.get() << '\n';
  char* a2[] = { "this", "by", "foo", "z", "x" };
  maxbuf<char*> m2(a2[0]);
  for(int i = 1; i < 5; ++i) m2.put(a2[i]);
  cout << "a2: " << m2.get() << '\n';
}
```

□ 特殊化の応用

- 一般の T →「<」が使われる
- char* →strcmp が使われる
- その他特殊ケースはいくらでも対応可能 … C++の実用性に 1 日の長があるかも?

3.9 テンプレートによる mixin クラス

□ 前回やったように、mixin とは「他のクラスに混ぜるためのクラス」

- それ自体単独でインスタンスを生成しないことから「抽象サブクラス」と呼ぶことも
- 前回の話では flavors などの多重継承を使って混ぜていた
- しかし C++のような静的検査な言語では多重継承ではうまく行かない

□ たとえば、次のようなクラスを考える。

```
class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};
```

□ 「update の回数を数える」

```
class count {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; update(v); }
  int getCount() { return num; }
};
```

□ 「更新履歴を記録する」

```
class history {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = getValue(); update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
};
```

□ これを次のようにして使いたい。

```
class MyClass :
  public balance, count, history {
};
```

□ flavors ではこれでできる (どっちの update が呼ばれるかの規則があって OK)。しかし C++ではそもそも getValue() や update() の宣言がないと型検査できなくてアウト。

□ ではどうするか…

- count や history で親クラスを指定すればもちろん動くようになる。
- しかしさまざまなものにこれらの機能を組み込もうと思って作ったのに、決め打ちで親クラスを書いてしまうのでは 1 つにしか使えない (または繰り返しコピーになる) のでよろしくな。
- ではどうすればいい?

□ 答: その親クラスをテンプレートパラメタで指定する!!

```
//sample35
#include <iostream>

class balance {
  int value;
public:
  balance() : value(0) { }
  void update(int v) { value += v; }
  int getValue() { return value; }
};

class maxbuf {
  int max;
public:
  maxbuf() : max(0) { }
  void update(int v) { if(v>max) max=v; }
  int getValue() { return max; }
};

template<class T> class count : public T {
  int num;
public:
  count() : num(0) { }
  void update(int v) { ++num; T::update(v); }
  int getCount() { return num; }
};

template<class T> class history : public T {
  int record[1000], nrecs;
public:
  history() : nrecs(0) { }
  void update(int v) {
    record[nrecs++] = T::getValue(); T::update(v);
  }
  void showHistory() {
    for(int i = 0; i < nrecs; ++i)
      cout << ' ' << record[i];
    cout << '\n';
  }
};

int main() {
  int a[] = { 5, 9, 2, 7, 6 };
  history< count< balance > > c1;
  count < history< maxbuf > > c2;
  for(int i = 0; i < 5; ++i) {
    c1.update(a[i]); c2.update(a[i]);
  }
  cout << "c1: " << c1.getCount(); c1.showHistory();
  cout << "c2: " << c2.getCount(); c2.showHistory();
}
```

□ mixin クラスを使うことで、1 つのクラスに盛り込まれている複数の側面を分離して記述し、使う時に組み立てることができる。

□ しかし、実際には 1 つの「側面」が複数のクラスにまたがって存在していることが普通→このような「一群のクラス」を 1 つの「層」として、それを積み重ねてシステムが構成されるというイメージ。

□ そのようなものを実現するには→ C++でもクラスの中にクラスが書けることを利用し、外側クラスをパラメタつきとする。

```
template<class T> class ThisLayer : public T {
public:
  class First : public T::First { ... };
  class Second : public T::Second { ... };
  ...
};
```

□ こうすれば、一群のクラスで Fist は First どうし、Second は Second どうし、…で縦に合成が起きる。

- ThisLayer の中にはこの層が実現する機能のためのデータやコードをまとめて入れることができる。
- このような構成を「mixin layers」と呼ぶ。
- このような「構成法」の代替案→言語自体の拡張 (AOP 言語) →すぐ後で

3.10 テンプレートメタプログラミング (簡単に)

□ テンプレートの展開→関数の評価のようなもの→計算に利用できる

```
//sample36
#include <iostream>

template<int n> class Fact {
public:
  enum { RET = Fact<n-1>::RET * n };
};
template<> class Fact<0> {
public:
  enum { RET = 1 };
};

int main() {
  std::cout << Fact<5>::RET << '\n';
}
```

- これを動かすと当然「120」が表示されるが、それは実行時に計算しているのではなく、コンパイル時に

「5*4*3*2*1*1」という式が生成され、それが計算されている→高速

- もっと複雑なデータを処理したり、複雑な手順を持つ関数を生成したりすることもできる → テンプレートメタプログラミング
- 使いこなすのはとっても難しいと言われている（あまりやりたくない）

3.11 本節のまとめ

□ 総称的プログラミング (型パラメタ、パラメタつき型) → C++ テンプレート、Java Genrics で実現 (内容はかなり違っている)。主な用途: コンテナクラス

□ STL(C++) → イテレータによるアクセス、汎用アルゴリズム

□ 型パラメタの制約 → 賛否分かれるところ。Java ではインタフェース/サブクラスによる制約指定可能 (だが不自由でもある) → C++に対する制約機構 (concepts) は設計途上 → 後で少し紹介

□ mixin クラス (C++) → 親クラスを拡張するための抽象サブクラス

□ 全体として C++テンプレートは強力、しかし難しい。より詳しく知りたい人は:

- アンドレイ·アレキサンドレスク著，村上訳，Modern C++ Design，ピアソン，2001． (むずかしい)
- ε π ι σ τ η μ η，高橋晶，C++テンプレートテクニック，ソフトバンク，2009． (少しやさしい。筆頭著者の読みは「えぴすてーめー」)

4 自己反映機能

□ 自己反映 (reflection): 実行中のコードが、実行系の情報にアクセスしたり、実行系の状態/動作を変更したりできるような機能

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

□ 何のためにそんなことをするのか?

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

4.1 3-Lisp: リフレクションの元祖

□ ベースレベルとメタレベルを区別

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

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

- 実装上は「遡られたところまで自動的にメタレベルを用意」

4.2 Smalltalk のクラスとメタクラス

□ Smalltalk では「クラス」もまたオブジェクト

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

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

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

4.3 メタオブジェクトプロトコル (MOP)

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

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

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

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

4.4 Java の自己反映機能

□ Java の自己反映機能→処理系そのもを変更する、という部分はない。

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

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

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

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

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

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

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

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

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

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

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

public class Sample33 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    while(true) {
      try {
        System.out.print("Class Name? ");
        System.out.flush();
        String cname = br.readLine();
        if(cname.equals("")) break;
        Class cls = Class.forName(cname);
        Constructor[] cons = cls.getConstructors();
        for(int i = 0; i < cons.length; ++i)
          System.out.println(""+i+": "+cons[i]);
        System.out.print("Constructor Number? ");
        System.out.flush();
        int cno = Integer.parseInt(br.readLine());
        Object obj =
          cons[cno].newInstance(new Object[]{});
        Method[] meths = cls.getMethods();
        for(int i = 0; i < meths.length; ++i)
          System.out.println(""+i+": "+meths[i]);
        System.out.print("Method Number? ");
        System.out.flush();
        int mno = Integer.parseInt(br.readLine());
        Object res =
          meths[mno].invoke(obj, new Object[]{});
        System.out.println("Result Class:"+
          (res.getClass()));
        System.out.println("Result: "+res);
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
}
```

□ これをたとえば次のクラスに対して使ってみる…

```
class Sample33Test {
  int val;
  public Sample33Test() { val = 1; }
  public Sample33Test(int i) { val = i; }
  public Sample33Test add() {
    return new Sample33Test(val+1);
  }
  public Sample33Test sub() {
    return new Sample33Test(val-1);
  }
  public String toString() {
    return "Sample33Test("+val+")";
  }
}
```

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

4.5 自己反映機能とコード生成ツール

□ 例: コンポーネントツールや GUI ビルダ

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

□ Java による GUI 部品配置の例

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.lang.reflect.*;

public class Sample34 extends JPanel {
  int x1, y1, x2, y2, x0, y0, width, height;
  Component c0 = null;
  JLabel l0 = new JLabel("Component:");
  JTextField t0 = new JTextField();
  JButton b0 = new JButton("Create");
  JButton b1 = new JButton("Move");
  JLabel l1 = new JLabel();
  JComboBox c2 = new JComboBox();
```

```
JLabel l2 = new JLabel("String:");
JTextField t2 = new JTextField();
JButton b2 = new JButton("Call");
JPanel p1 = new JPanel();
Method[] meths; int[] maps;
public Sample34() {
  setLayout(null);
  add(l0); l0.setBounds(10, 40, 80, 30);
  add(t0); t0.setBounds(100, 40, 200, 30);
  add(b0); b0.setBounds(320, 40, 80, 30);
  add(b1); b1.setBounds(420, 40, 80, 30);
  add(l1); l1.setBounds(10, 70, 500, 30);
  add(c2); c2.setBounds(10, 100, 500, 30);
  add(l2); l2.setBounds(10, 140, 80, 30);
  add(t2); t2.setBounds(100, 140, 200, 30);
  add(b2); b2.setBounds(320, 140, 80, 30);
  add(p1); p1.setBounds(0,0,1,1);
  p1.setBackground(Color.green);
  b0.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      try {
        c0 = (Component)Class
          .forName(t0.getText()).newInstance();
        p1.setBounds(0, 0, 1, 1);
        add(c0); c0.setBounds(x0, y0, width, height);
        t0.setText("");
        meths = c0.getClass().getMethods();
        maps = new int[meths.length];
        c2.removeAllItems();
        for(int i=0,k=0; i < meths.length; ++i) {
          Class[] arg =
            meths[i].getParameterTypes();
          if(arg.length == 1 &&
             (arg[0] == String.class ||
              arg[0] == Object.class)) {
            c2.addItem(meths[i].toString());
            maps[k++] = i;
          }
        }
      } catch(Exception ex) {
        l1.setText(ex.toString());
      }
    }
  });
  b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(c0 != null) {
        p1.setBounds(0, 0, 1, 1);
        c0.setBounds(x0, y0, width, height);
      }
    }
  });
  b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      try {
        Method m = meths[maps[c2.getSelectedIndex()]];
        m.invoke(c0, new Object[]{t2.getText()});
        t2.setText("");
      } catch(Exception ex) {
          l1.setText(ex.toString());
      }
    }
  });
  addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
      x1 = x2 = e.getX(); y1 = y2 = e.getY(); calc();
    }
    public void mouseReleased(MouseEvent e) {
      x2 = e.getX(); y2 = e.getY(); calc();
    }
  });
  addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
      x2 = e.getX(); y2 = e.getY(); calc();
    }
  });
}
private void calc() {
  x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
  width = Math.abs(x1-x2); height = Math.abs(y1-y2);
  p1.setBounds(x0, y0, width, height);
}
public static void main(String[] args) {
  JFrame f = new JFrame("demo");
  f.add(new Sample34());
  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  f.setSize(600, 600);
  f.setVisible(true);
}
}
```

4.6 本節のまとめ

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

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

□ ついでに: オブジェクト指向言語の諸側面のまとめ

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

5 Haskell: 関数型言語

□ 関数型言語 --- 「関数とその適用に基盤を置く言語」

- 「副作用のない言語」 --- こちらの方がぴったりする感じ?
- 「参照透明性」(referencital transparency) とも呼ぶ
- 副作用はバグの原因となり、また並列実行を難しくする
- 副作用がなければ遅延評価を標準としやすい

5.1 関数型言語の歴史

□ Lisp (McCarthy) --- ラムダ式に基盤

- 確かに「関数とその適用」に基盤を置くが…
- 副作用はそこら中で使う「(setq x 1)」「(replaca (cdr x) y)」
- このため関数型と呼ばずに「適用型」(applicative) と呼ぶ向きも
- Lisp 族の言語は多数あり今日でも増えている
- Lisp 1.5, Scheme, CommonLisp, EmacsLisp, ...

□ FP (Backus) --- 最初の「副作用を避ける」試み

- 「副作用がないと出力も結果の保存もできない」問題に直面→「トップレベルは特別」扱い
- 実験的な言語で実用性はない

□ ML (Milner) --- 実用となった最初の「関数型」言語

- 「書き換えられる値」を明示的にその旨定義「ref 1」
- 書き換えが起こる箇所を限定し、残りは参照透明
- 型推論による強い型検査
- 多くの処理系、拡張言語の基盤となる
- SML (Standard ML), Miranda, CAML, OCAML (Objective CAML)

□ Haskell --- 参照透明性をすべてに適用

- 遅延評価をすべてに適用
- 副作用は IO モナドと呼ばれる形で統一して扱う
- 並行プログラムにも複数の機能を通じて対応

5.2 Haskell 入門

□ 基本は「関数定義」

- 以下の実行例は Hugs98 という処理系による

```
add x y = x + y
```

- プログラムはファイルに記述し、それを読ませて、実行
- すべての式は強く型付け。「:type 式」で型を調べられる

```
Main> :load "test1.hs"
Main> add 1 2
3
Main> :type add
add :: Num a => a -> a -> a
```

- これは「ある型 a は Num の一種であり、add は a -> a -> a 型」と読む

□ 定義しないで直接関数を書くこともできる (λ式)

```
Main> (\x -> x + x) 2
4
```

□ 中置記法、前置記法

- 関数名を ‘...‘ で囲むと中置記法にできる。逆に中置演算子は (...) で囲むと普通の関数名と同じに扱える

```
Main> 1 ‘add‘ 2
3
Main> (+) 1 2
3
```

□ カリー化

- 「a -> a」は、「a 型を引数として受け取り、a 型を結果として返す関数」
- すべての関数は引数は 1 つ
- では、先程の add や (+) は? 「a -> a -> a」は「a を引数として受け取り、a -> a を返す」

```
add1 = add 1
add2 = (+) 2
```

- つまり add に 1、(+) に 2 を与えた結果も 1 つの関数

```
Main> :type add1
add1 :: Integer -> Integer
Main> add1 2
3
```

- 「N 引数の関数に 1 つの引数を与えて、残り N-1 引数を受け取って結果を返す関数を得る」ことを「部分適用」と呼ぶ
- このように関数を全部 1 引数として扱うこと→カリー化
- なお、今度は add1 の引数や返値の型は Integer。これは「1 と足す」ことまで分かったから
- このように、演算や引数の値によって式の型を推論すること→型推論 (type inference)
- 中置記法を使えば第 2 引数を部分適用することもできる

```
sub x y = x - y
sub1 = (‘sub‘ 1)  ← (? ‘sub‘ 1) の意味
```

□ 以下では「関数の型」は宣言し、中で使う変数は宣言しないことにする ←型チェックと記述の手間を勘案して

□ 再帰関数 --- 繰り返しの代わりにすべて再帰を使う

- 変数は書き換えられないので別のインスタンスを作るしかない

```
fact1 :: Integer -> Integer
fact1 n
  | n > 0     = n * fact1 (n-1)
  | otherwise = 1
```

- ガードを使った本体の振り分け
- 字下げに意味がある (字下げが増える→上の行の内側)

- この素朴な定義だと再帰 1 回ごとに中間結果を保存する必要がある
- 別バージョン→こちらの方は中間結果の保存不要
- 補助関数 f2sub を where を使って定義

```
fact2 :: Integer -> Integer
fact2 n = f2sub 1 n
  where
    f2sub r n
      | n > 0     = f2sub (n*r) (n-1)
      | otherwise = r
```

- このように「最後に自分自身を呼びその結果を自分の結果とする」場合→実装では関数の先頭に新しい引数を持ってジャンプ (TCO --- tail call optimization)
- TCO を意識して書く必要→ちょっとワザを強制されている感じ?

5.3 リストデータ構造

□ リスト --- Lisp 以来の、関数型言語で広く使われるデータ構造

- Haskell では「:」がリストの「先頭」と「残り」をつなぐ構成子
- [1,2,3] == 1:2:3:[]

□ 素朴には、リストを扱う関数は再帰で書ける

```
addlist1 :: [Integer] -> [Integer]
addlist1 (x:xs) = x+1:addlist1 xs
addlist1 []     = []
```

- 引数部分にパターンを記述→パターンマッチによる枝分かれ

```
Main> addlist1 [1,2,3]
[2,3,4]
```

□ 実は上の例のようなのはいまいち

- 「1 足す」という仕事と「リストにそれぞれ適用」という仕事は別の概念 → 分けた方がいい

```
mymap :: (a -> a) -> [a] -> [a]
mymap fn (x:xs) = fn x : mymap fn xs
mymap fn []     = []

addlist2 :: [Integer] -> [Integer]
addlist2 = mymap (+ 2)
```

- mymap の型は「ある型 a を受け取り a を返す関数と、a の並んだリストを与えると、a の並んだリストを返す」
- この a のところに任意の特定の型が入る ← 型パラメタ
- これに「2 足す」関数を与えるので、できあがる関数は「整数のリストを受け取り、整数のリストを返す」型

```
Main> addlist2 [1,2,3]
[3,4,5]
```

- 実はこれと同じことをする標準関数 map が用意されている →プログラマが書くのは「addlist2 = map (+ 2)」だけ

□ 同様のものとして「リストから特定の条件のものを選別」

```
myfilter fn (x:xs) =
 if (fn x)
  then x:myfilter fn xs
  else myfilter fn xs
myfilter fn []     = []

larger1 = myfilter (> 1)

Main> larger1 [0,1,2,3]
[2,3]
```

□ これも標準関数 filter が用意されているので「larger1 = filter (> 1)」でよい

□ 「合計」のようなものは?

- foldl (+) 0 [w,x,y,z] = ((((0+w)+x)+y)+z)
- foldr (+) 0 [w,x,y,z] = (w+(x+(y+(z+0))))

```
mysum :: [Integer] -> Integer
mysum = foldr (+) 0

Main> mysum [1,2,3,4]
10
```

- ここでは説明しない理由により、foldr の方が効率がいい

□ 例題:「1 から 10 までの奇数の自乗和を求めよ」

- 「1 から 10 までのリスト」は次の記法で作れる

```
Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

- 関数の合成: (f (g x)) = (f.g) x
- 考え方: 「奇数のものを抜き出し、自乗して、合計」

```
oddsqsum :: [Integer] -> Integer
oddsqsum = (foldr (+) 0).(map sq).(filter evenp)
  where
    evenp x = mod x 2 /= 0
    sq x = x * x
```

- このように「引数に言及せず関数だけ合成して行く」やり方 (「pointless な記法」) が関数型らしいところ

```
Main> oddsqsum [1..10]
165
```

□ リスト内包表記

```
[式|変数<-リスト, 変数<-リスト, ..., 条件]

Main> [(a,b)|a<-[1..5],b<-[1..5],mod a b == 0]
[(1,1),(2,1),(2,2),(3,1),(3,3),(4,1),(4,2),(4,4),
(5,1),(5,5)]
```

- quicksort を ML で書くと…

```
qsort :: [Integer] -> [Integer]
qsort (x:xs) =
  (qsort [a|a<-xs,a<x])++[x]++(qsort [a|a<-xs,a>x])
qsort [] = []
```

- 「++」はリストの連結演算

5.4 型クラス

□ ところで、整列は「順序があるもの」のリストなら何でもいいはず

- 「演算子 `>` :: `a -> a -> Bool` を持つような型 a」のように、「ある型についてこういう関数を持つ」ということを表現する →Haskell では「型クラス」を使う (オブジェクト指向のクラスとはまったく別のもの)。
- 上の場合は型宣言を次のように直す:

```
qsort :: Ord a => [a] -> [a]
```

```
Main> qsort "How are you?"
" ?Haeoruwy"
```

- 実は「文字列」は「文字のリスト」なので。

□ 他の型クラス: 「等しい演算 (==) を持つ」: Eq, 「文字列に変換できる関数 show を持つ」: Show など。

5.5 遅延評価

□ 実は「無限のリスト」も作れる。「[1..]」

- ただし、無限のリストを打ち出すと無限に打ち出し始めるが…
- 「先頭から N 個取る」関数 take と組み合わせれば問題なく大丈夫

```
Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

- 無限リストに対して演算することも問題ない

```
Main> take 10 (map (* 2) [1..])
[2,4,6,8,10,12,14,16,18,20]
```

- 普通の言語: 関数 (メソッド) 呼び出し時にパラメタは値に評価される (正格評価, eger evaluation)
- Haskell: 関数適用は「そのままの形」で覚えられている。実際にその値が必要になった時にはじめて評価 (実行) される (遅延評価, lazy evaluation)
- 例: 「フィボナッチ数から成る無限リスト」を計算する関数 fib。(引数がないので `->` がなく結果の型だけ)

```
fib :: [Integer]
fib = fibx [] 1 1
  where
    fibx l a b = l ++ (fibx [a] b (a + b))
```

- 連結演算 (++) の右側の fibx の呼び出しは、必要になった時に行われる

```
Main> take 10 fib
[1,1,2,3,5,8,13,21,34,55]
```

□ 遅延評価の利点

- 再帰を書くのに枝分かれが要らない (先の fib の例) →シンプル
- 「いくつまで用意するか」を考えないでよい (無限に用意して使う時に必要なだけ取る)
- きっちり必要なだけしか計算しないで済む

□ 遅延評価の弱点

- 普通の人は慣れていない
- いつ実行されるか予測しづらい ← しかし、純粋な関数型言語であれば副作用がないので、いつ実行されても結果は同じ (はず)

5.6 データ型の定義

□ 既存の型をもとに新しい型を定義 → data 定義 (代数的データ型の定義。他にもいくつか型を定義する方法がある)

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
  deriving (Show, Eq)
```

- 2 分木を定義。保持するデータは型パラメタ a で表す
- show と (==) を自動的に生成
- データを 1 つだけ持った木を生成

```
newtree :: a -> Tree a
newtree x = Node (Leaf) x (Leaf)
```

- 木への挿入 (副作用はないので挿入した新しい木を返す)

```
instree :: Ord a => Tree a -> a -> Tree a
instree (Leaf) x = newtree x
instree (Node l y r) x
  | x < y = Node (instree l x) y r
  | x > y = Node l y (instree r x)
  | otherwise = Node l y r
```

```
Main> instree (instree (newtree 3) 5) 2
Node (Node Leaf 2 Leaf) 3 (Node Leaf 5 Leaf)
```

- 木の中にめざすデータがあるかどうか検索

```
searchtree :: Ord a => Tree a -> a -> Bool
searchtree (Leaf) x = False
searchtree (Node l y r) x
  | x < y     = searchtree l x
  | x > y     = searchtree r x
  | otherwise = True
```

```
Main>searchtree (instree (instree (newtree 3)5)2)2
True
Main>searchtree (instree (instree (newtree 3)5)2)4
False
```

5.7 IO モナド

□ 最大の問題: 副作用がないのに、どうやって入出力とかするの? (C で getchar() を 2 回呼んだら別の文字が取れて来るし…)

- 答え: 「世界を関数の引数の一部として渡し、世界を返してもらう」
- このために「IO a」という型を使う。これが「世界の状態ともに型 a の値を持っている」ことを表す。
- 「IO a」は Monad という型クラスに属していて、この型クラスは次の 2 種類の演算 (関数) を定義している

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

- 重要なのは (>>=) で、これは「ある世界の状態での a の値と、a の値を処理してある世界の状態での b の値を返す関数とを組み合わせることで、処理を実行した後の世界の状態での b の値を返す」演算子。
- 入出力を行う関数などはすべて、結果が IO a 型になっている

```
Main> :t getLine
getLine :: IO String
Main> :t putStr
putStr :: String -> IO ()
```

- なお型「()」は unit と読み、「何か値を渡す必要があるが値の内容は何もなくていい」時に使う。その値もやはり「()」

□ メイン関数は「IO ()」を返す関数を実行できる (=副作用を持つ動作を実行できる)

- 例題: 2 つ文字列を読み込み、逆順に連結して出力

```
test1 :: IO ()
test1 = putStr "str1? "
        >>=
        \x1 -> getLine
        >>=
        \x2 -> putStr "str2? "
        >>=
        \x3 -> getLine
        >>=
        \x4 -> putStr (x4 ++ x2)
        >>=
        \x5 -> return ()

Main> test1
str1? aaa
str2? bbb
bbbaaa
```

- この「>>=」でつなげて書くのは書きづらくよみづらいので、これと同じことをもっと短く書ける do という構文が用意されている

```
test2:: IO ()
test2 = do
        putStr "str1? "
        s1 <- getLine
        putStr "str2? "
        s2 <- getLine
        putStr (s2 ++ s1)
        return ()
```

□ Haskell は副作用のない言語、というのは本当なのか?

- IO a を使うことで「副作用のある箇所を明示/分離している」という方が正確
- ほとんどの場所は IO a を使わずに書ける→副作用のない世界
- 「少量の副作用のある世界から、大量の副作用のない世界を呼び出してやりたい仕事をこなす」と考えるのがいいかも

5.8 Haskell のまとめ

□ 関数型プログラミング言語 Haskell ---

- 高階関数を駆使するなどにより記述がコンパクト
- 強い型検査、ただし型宣言はしなくても型推論が補ってくれる
- 参照透明性 --- ある式を評価した結果は常に同じ
- 遅延評価 --- 無限データ構造なども作れる。実際に値を計算するのは「使われた時」でよい
- 副作用のある部分は明確に分離して記述 (型が違うの分かる)

□ かなり「違う世界」ですが、どうですか?

6 第 3 回課題

□ 以下の課題から 1 つ以上を選択してプログラム作成ないし実験を行ない、作成したプログラムが課題を達成していることを論ぜよ。課題中に指定した考察をきちんと書くこと。

□ (11) JavaScript を用いて、自分の Web ページに動きや機能をつけるような「ライブラリ」(つまり自分だけにしか使えないのではなく、他人にあげたら使ってもらえるような形のもの) を作ってみよ。その経験に基づいて、JavaScript によるプログラミングと他の言語 (Java 等) によるプログラミングの違いについて考察せよ。「動きや機能」の例としてはたとえば次のものが考えられる (ただしこれらに限定せず自由にアイデアを出してよい)。

- HTML 要素を動かしたり現われさせたり消したりさせる
- ポップアップ/プルダウンメニュー機能 (サブメニューも出せるとなおよい)
- ページ内でクリックすることでクリックした個所を強調するなどページ内容が加工できるようにする

□ (12) C++テンプレートまたは Java Generics (JDK 1.5 以降必要) を用いて、自分なりの「汎用的な役に立つクラス」を開発してみよ (イメージとしては max バッファや last2 バッファのような感じでよい)。実際に複数の型をあてはめて動作させること。また、それを使った場合と使わずに「普通に」書いた場合の性能比較も行うこと。(さらに、C++テンプレートと Java Generics の両方作って両者の比較もするとなおよい。)

□ (13) C++テンプレートメタプログラミングで階乗以外の何らかの値の計算をさせてみよ (組合せの数とかフィボナッチ数とか?)。その上で、普通のコードで計算した場合と性能比較を行ってみよ。もしコンパイルが異常に遅いならその原因も究明せよ。

□ (14) Java 言語の自己反映機能を使った何らかのプログラムを動かしてみよ。

□ (15) Haskell 言語を使って簡単なプログラムを書いてみよ。それと同じ処理をする C++ないし Java のプログラムも書き、比較してみよ。

7 さいごに

□ この講座では、さまざまなプログラミング言語に現れる概念を、その設計思想、用途、実装、特徴、問題点などの観点から整理して見てきたつもりです。

□ 構成としては、1 回目→プログラミング言語一般〜抽象データ型、2 回目→オブジェクト指向、3 回目→より「変わった」「新しい」の話題、のように構成しました。

□ 例年「オブジェクト指向の将来は (次は) 何か」みたいな質問がありますが、私が知りたいです… とりあえず、C++ではさらにテンプレートが多く使われるでしょうとか、それくらいのことは分かっていますが。

□ 関数型プログラミングは最近人気が出て来ているので、昨年度から入れるようにしたのですが、どうだったでしょうか。

□ ともあれ、おつかれ様でした。

8 おまけ: 久野の所属先について

□ 筑波大学 (東京地区) ビジネス科学研究科 --- 社会人対象の夜間大学院

- 「東京地区」というのは文京区大塚にある (旧・東京教育大の跡地)。最寄り駅: 茗荷谷 (東京メトロ丸の内線)1 分。茗荷谷は池袋から 5 分、JR 東京駅から 15 分。
- (2010 年度は建物を建て替え中→神保町に仮校舎)
- 経営システム科学専攻 --- 修士課程
- 企業科学専攻システムズマネジメントコース --- 博士課程
- http://www.gssm.otsuka.tsukuba.ac.jp/

□ 修士では、2006 年度から「4 プログラム制」として、情報系の人も積極的に受け入れるています→興味ある方はぜひご検討を。

□ 博士課程 (入試は 8 月提出/9 月試験。例年だと 2 月にも試験あり) は、修士を持っている人を対象として、在職のまま「博士 (経営学)」または「博士 (システムズ・マネジメント)」を取得。情報系の人も多数受け入れています→修士持っていて「研究」に興味のある方はぜひご検討を。