TSSI基盤技術研修コース
— プログラミング言語 — ＃2

久野　靖*

1999.5.14

はじめに

□ 第1回の内容について…

- ご意見を拝見しましたが、結構「知っていた」方が多かったようです。まあ、体系的に整理するとか知らなかったことが多少でもプラスされればそれでよかったかと思います。
- あまり知らなかったとか難しかったというご意見もありましたが、全体的には適切なレベルだったかと思います。少々速い/内容が多いという感想もいただきましたので気を付けます。
- Javaでの例題を読んでいただく部分はやっぱり時間がかかりますね。ざっと説明してあとで再度各自に読んでもらうという形を取るのがいいかと思います。

□ 第2回の構成:

- プログラミング言語の進化→前回の積み残し
- オブジェクト指向言語の進化と諸概念を並行して見て行く(歴史的な順序におおむね従った方がわかりやすい)
- オブジェクト指向言語の実装技術→必要な箇所にそのつど挿入(諸概念が出て来たところで説明した方が納得しやすい)
- オブジェクト指向言語の利用技術→最後にまとめて解説
- Javaによる例題/演習→それに適した話題のところに適宜挿入

1　手続き型言語の進化

□ 結局、現在でもメジャーな(プログラマ人口が多い、ソフトウェア製品の開発に使われることが多い)言語は手続き型言語の末裔。

- ただし、手続き型言語の系列もかなり変化してきている
- その系譜を眺めてみよう

1.1　構造化プログラミング

□ 初期の手続き型言語はGOTO文化(COBOL、FORTRAN、…)

- スパゲティプログラムになりがち

□ 構造化プログラミング(1960年代末〜) → GOTOを使わず、もっと整った制御構造(if-then、while、repeat-until)を使おう

□ 段階的詳細化(トップダウン開発)

□ いずれも、教条的にやってもうまく行かないという結論

□ 実現上の技術は、特にない(コンパイラとしては当たり前)

1.2　手続きと引数機構

□ アセンブリ言語開発での手続き→広域変数に依存することが多かった

□ 構造化設計(モジュール間の存関係を減らす)→広域変数より引数

□ 引数渡し機構…値渡し、名前渡し(Algol)→参照渡し、copy/restore(Fortran)→オブジェクト渡し(オブジェクト指向)

- 参照渡しの引数として渡した変数をそのまま作業変数としてさまざまな計算にも利用するのはよくない(なぜか?)

```
call subr(x)
...
x = 1.0
y = .... x ....
```

*筑波大学大学院経営システム科学専攻

1.3 例外

□ 例外とは→通常の制御の流れと違う制御の移行 (例外的状況/エラー)

- 通常の制御構造でまかなえない goto の代替機能として…
- エラー処理の統一的な枠組みとして…

□ Lisp などで古くからある→最近は C++、Java、…

```
try {
  ...
  int i = Integer.parseInt(s).intValue();
  ...
} catch(NumberFormatException e) { ... }
```

□ 利点:

- 同種のエラーを 1 箇所でまとめて処理できる
- 異種のエラーをそれぞれ別の場所で処理できる
- コードの主要部分がエラー処理でごちゃごちゃしない
- 手続きに無理矢理「エラーコード」を返させなくてすむ
- 受け止められていない例外をコンパイラがチェック

□ 例外機構のバリエーション

- 手続き内/手続き間
- 例外の伝播…そのまま/汎用の例外に変換/必要なら変換
- 例外からの復帰/後ろへ抜けるだけ

□ 例外機構はどうやって実装する?

- 自明な選択肢: 手続きの戻りコードにして戻り側で検査→わかりやすいが通常の呼び/戻りがのろくなる→避けたい
- 一般的な選択肢: 例外スコープの表を作っておき、例外が起きた時に自前で戻り処理をやりながら戻り先を検索する→例外が起きた時の速度は遅くなるが、通常の実行は遅くならない→例外はそう頻繁におきないはず→よく使われる手法

1.4 Java の例外機構

□ 例外の種別→ Throwable クラスのサブクラスとして表す (分類を提供)

- 共通のメソッドとして、printStackTrace() を提供
- 次のような階層構造

```
Throwable ←例外すべて
  Error ←エラー: 起こってはならない事象 (システムエラー等)
  Exception ←普通の例外
    RuntimeException: 通常の実行時にいつでも起こり得る例外
      ...
```

□ 受け止める時→クラスを指定するとそのサブクラスの例外も捕捉

```
try {
  ...
  try {
    ...
    ※ここで例外が起きた時
    ...
  } catch(IOException e) { ... } ← IO 例外を捕捉
  ...
} catch(Exception e) { ... } ←すべての Exception を捕捉
```

□ 捕捉しなかった例外→外 (メソッド呼び出し元) に伝播

□ あるメソッドから例外が投げられるためには、メソッドの頭書きでその例外を投げると宣言しておかなければならない。

```
public void read(...) throws IOException {
  ...
}
```

- 効果: あるメソッドを呼んだ場合に発生し得る (処理する必要がある) 例外は何と何かが明確になる

□ あるメソッドを呼んだ場合に発生し得る例外は、必ず受け止めて処理するか、または自分も同じ例外を返すように宣言する必要がある

```
public void myread(...) throws IOException {
  ... read(...) ...
}
または
public void myread(...) {
  try {
    ... read(...) ...
  } catch(IOExceptoin e) { ... }
}
```

- 効果: 処理すべき例外が処理されないまま終ることがないよう、言語機構としてサポートしてくれている

□ ただし、すべてのメソッドは予め「throws Error, RuntimeException」と宣言ずみであるものと見なす

- 理由: これらの例外はどこででも発生し得るため、いちいち宣言させていては大変すぎる

□ 自前の例外を作る簡単な例題:

```
public class Sample6 {
  public static void main(String args[]) {
    try {
      int i = (new Integer(args[0])).intValue();
      System.out.println("sqrt(i) = "+mysqrt(i));
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  private static double mysqrt(int i)
      throws Negative {
    if(i < 0) throw new Negative(i);
    return Math.sqrt((double)i);
  }
}

class Negative extends Exception {
  int value;
  public Negative(int i) { value = i; }
  public String toString() {
    return "Negative: <"+value+">";
  }
}
```

1.5 型

□ 型とは?

- 値の集合→古典的な定義。例: 整数型→{1, 2, 3, ...}
- 一群の操作を提供する値の集合→抽象データ型/オブジェクト指向に合う。例: 整数型→加算、減算、…

□ プログラミング言語における型…

□ ごく初期の型

- 機械語、アセンブリ言語 (原始的な場合) には型がない
- (旧)Fortran →演算命令の区別のための型 (整数、実数) →文字型がない、配列も「単なる繰り返し」であり型ではない
- COBOL →レコード定義も「データの集まり」であり型ではない
- Algol68 → ref T 型 (C 言語でいう「*」) →ポンインタを型にした

□ Pascal →これらさまざまなものを型として整理し、「ユーザ定義の型」を大幅に導入した。

- 基本型 (int、real、char、boolean、列挙型)
- 部分範囲型 (「1..100」)
- 配列型、ポインタ型、レコード型、集合型
- 範囲型、列挙型、集合型は現在の言語ではすたれている
- 集合型の実装→ビットベクター。いくつかの主要な Pascal 処理系では 32 ビットに限定されていたので「set of char」が使えなかった (今でも日本語だと無理がある…)

□ C → Pascal と同時代のライバル

- 範囲、列挙、集合に加え、論理型がなく、配列とポインタが混同されている (わざと) →アドホックな設計
- 「int sub(int *a)」と「int sub(int a[])」の a の型は同じ? (Y/N)
- 「int a[100]」と「int *a」は意味は違うが…a の型は同じ? (Y/N)

□ 結局、型は何のためにある?

- 効率のよいコードを出すため…△ (ごく特別な場合) → cf. ML の型推論
- 値の種別に応じた操作を人間がいちいち指定しなくて済む…◎
- 値の種別の勘違いを検出してくれる (強い型の場合) …◎
- プログラムの構造を設計/記述する手段…◎

□ 強い型による保護

- 型検査で許された操作しかできない→メモリ保護として利用可。この方式の代表格: Java
- 型検査を抜ける (escape) 方法があると役に立たない。代表: C や C++のキャスト
- マシン語を直接書けばどのような操作でも可能なのでやはり問題
- Java では仮想マシン語の検証器によってこのような操作をはねる (ただし native メソッド --- 他言語/機械語で書かれたコードをリンクして呼び出す --- があると無意味)

1.6 モジュール

□ 手続きが単位では「カタマリ」が小さすぎる

□ 複数の手続き群が協調して 1 つのデータ構造を維持する →よくある

□ これらの機能を言語機構として取り入れ→モジュール→「データ構造と、一群の手続きとをひとまとめにして、外からアクセスできるもの/できないものを定める機能」

- Modula、ConcurrentPascal、Modula-2、…
- C などでも「ファイル」はモジュール的に使える (static 変数/関数はそのファイル内だけで参照可能)

□ モジュールの機能は「隠すこと」。なぜ「不自由さ」が重要なのか?

- 外部から「見えてしまう」と「使ってしまう」。「使ってしまう」と「依存関係が増えてしまう」(それで後から変更しようとするとはまる)。
- たとえ変数が見えても「型定義が見えない」ようにできる言語も。そうすると「操作する方法がない」ので「その型の変数は作れるが、そのモジュールの手続きを呼ぶ以外のことは何もできない」状態になる→情報隠蔽 (encapsulation)→抽象データ型につながる考え。
- モジュールの実装→「隠すこと」だけだから特になし (コンパイラで正しくないアクセスをはねればよい)

1.7 抽象データ型

□ 1 つのモジュールに 1 つのデータ構造、ではなく、あるモジュールが定義するデータ構造を「いくつも作りたい」場合も→つまり、新しい「値の種類」が増える→型→「抽象データ型」

- 抽象データ型をサポートする言語→ 1970 年代に抽象データ型が脚光を浴びた時にいくつか作られた。CLU、Alphard、…、Ada、…
- 現在ではオブジェクト指向言語にその機能が引き継がれていると言える (由来はちょっと違っているが…)

□ CLU による「整数の集合」型

```
intset = cluster is create, insert, is_in
  node = record[i:int, l,r:intset]
  rep = variant[n:null, r:node]
  create = proc() returns(cvt)
    return(rep$make_n(nil)) end create
  insert = proc(r:cvt, i:int)
    tagcase r
    tag n : rep$change_r(node$[i:i,
      l:intset$create(), r:intset$create()])
    tag r(n:node) :
      if n.i = i then     % do nothing
      elseif n.i < i then intset$insert(n.l, i)
      else                intset$insert(n.r, i)
      end
    end
  end insert
  ...
end intset
```

□ 抽象データ型の実装方法…

- すべてヒープ上のデータ構造へのポインタとするなら難しくはない (CLU)
- スタック上のデータなどにしたい場合は、データ構造の大きさを知らなければならない→抽象データ型の定義において「外から見える」部分と「見えない」部分に分けて記述し、「見えない」部分は「見えないふり」をする→ Ada、C++→しかし「見えない」部分の変更でも再コンパイルが必要だったりしていまひとつ使いやすくない。現在のオブジェクト指向言語ではすべてポインタとするのが主流。

1.8 型パラメタ

□ たとえば上のようなコードで、格納される内容は int でなくてもよいはず→「型」を「パラメタ」として扱えるような抽象データ型 (型生成子) が使えるとよい。

- たとえば通常の言語の「配列」なども要素型をパラメタとする型生成子だと言える。
- ものによっては、パラメタ型が「どのような操作を提供しているか」を指定する必要。

□ 型パラメタを持つ言語→ CLU、Ada、C++

□ CLU による「任意の型 T の集合」型

```
intset = cluster[t:type]
      where t has lt:proctype(t,t) returns(bool)
    is create, insert, is_in
  node = record[e:t, l,r:intset]
  rep = variant[n:null, r:node]
  create = proc() returns(cvt)
    return(rep$make_n(nil)) end create
  insert = proc(r:cvt, e:t)
    tagcase r
    tag n : rep$change_r(node$[e:e,
      l:intset$create(), r:intset$create()])
    tag r(n:node) :
      if n.e = e then     % do nothing
      elseif n.e < e then intset[t]$insert(n.l, e)
      else                intset[t]$insert(n.r, e)
      end
    end
  end insert
  ...
end intset
```

□ 型パラメタの実現方法

- 自明な方法: コンパイル時にマクロ展開する→コードが大きくなる、厳密なチェックが抜けることがある、再帰的な型指定はできない (無限に展開)、その反面最適化はやりやすい。

- パラメタ型の情報を実行時に保持する方法→利点と欠点は上記のほぼ反対。

□ 「配列」→ほぼすべての言語にある「組み込みの」型生成子。たとえば Java では配列だけがこのような形で特別扱いになっている。

1.9 オブジェクト指向

□ オブジェクト指向については、この後詳しく取り上げますが、ちょっとだけ。

□ オブジェクト指向の定義→「プログラムが扱う対象を『もの』として取り扱う見かた/考えかた」

- だから、「`intset[real]$insert(s, 3.14)`」では抽象データ型だが、「`s!insert(3.14)`」と書ければそれはオブジェクト指向、という考え方も成り立つ (CLU にそういう拡張を施したものがあった)。

1.10 この節のまとめ

□ アセンブラでも高水準言語でもプログラマの生産能力 (lines/日) は同じ→高水準であるほど効率はよい

□ 考えなければならないこと (依存関係、関連性) が多いほど生産効率は悪くなる

□ そのため、次のことが重要

- 大きな「かたまり」で考える
- 「かたまりの中」と「かたまりどうし」を分けて考える
- 見なくてよいことは「見えない」ようにすることが大切

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)
- オブジェクト指向ソフトウェア開発 (コンポーネント技術)
- オブジェクト指向データベース (マルチメディア)
- 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→「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→「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。

3 Simula: オブジェクト指向言語の始祖

□ 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 (以下に列挙) →極めて先進的

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」

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

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

□ クラスに規定されるもの

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

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

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。

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

□ 変数の読み書きは領域先頭からの固定オフセットに対して行えばよい

□ 動的な言語では変数の位置も探索する場合がある

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

□ 通常、動的分配のための手当てが必要 (後述)

- 動的分配を使わないならデータ領域だけでよい (C++など)

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

□ 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++のように「キャスト」されてしまうと問題がある
- 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

3.5 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)

- 「犬の『前進』、自転車の『前進』、バスの『前進』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 犬 then 犬.前進 () elif 自転車 then 自転車.前進 () 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
- ただし、静的分配 (見ためは上と同じだが場所ごとに型は決まっている) でもそれなりに使えると思う

□ 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 (Simula の場合どうかはすぐ次で説明)

3.6 動的分配の実現

□ 原理的には「メソッドへのポインタを各レコードに持たせる」

□ 実際には 1 つのクラスに属するオブジェクトはすべて同じメソッド群→クラスデータ構造にメソッドポインタを持たせ、インスタンスにはクラスデータ構造へのポインタを持たせるのが普通。

□ 各インスタンスに持たせると、メソッドを置換できる (実際にどれくらいそうしたいかは?)

□ 変数と同様、動的な言語では実行時に探すことも有り得る

□ 継承があると、さらに探索が必要な場合も

□ C 言語でちょっと動的分配な例を書いてみました。

```
typedef struct obj {
  char *name;
  void (*sound)(), (*pinfo)(); } *obj_t;
typedef struct obj_car {
  char *name;
  void (*sound)(), (*pinfo)();
  int speed; } *obj_car_t;
typedef struct obj_dog {
  char *name;
  void (*sound)(), (*pinfo)();
  char *kind; } *obj_dog_t;

void car_sound(obj_t o) {
  printf("Broom!\n");
}
void car_pinfo(obj_t o) {
  obj_car_t c = (obj_car_t)o;
  printf("name: %s, max_speed: %d\n",
    c->name, c->speed);
}
obj_car_t new_car(char *s, int m) {
  obj_car_t c =
    (obj_car_t)malloc(sizeof(struct obj_car));
  c->name = s; c->speed = m;
  c->sound = car_sound; c->pinfo = car_pinfo;
  return c;
}
void dog_sound(obj_t o) {
  printf("Vow! Vow!\n");
}
void dog_pinfo(obj_t o) {
  obj_dog_t d = (obj_dog_t)o;
  printf("name: %s, kind: %s\n",
    d->name, d->kind);
}
obj_dog_t new_dog(char *s, char *k) {
  obj_dog_t d =
    (obj_dog_t)malloc(sizeof(struct obj_car));
  d->name = s; d->kind = k;
  d->sound = dog_sound; d->pinfo = dog_pinfo;
  return d;
}

main() {
  int i;
  obj_t a[3];
  a[0] = (obj_t)new_car("my-car", 200);
  a[1] = (obj_t)new_dog("my-dog", "bulldog");
  a[2] = (obj_t)new_car("your_car", 300);
  for(i = 0; i < 3; ++i) {
    obj_t o = a[i]; (o->sound)(o); (o->pinfo)(o);
  }
}
```

- 共通部分とクラスごとの固有部分が持てる
- メソッドポインタ部分はクラスごとに共通化できる
- 動的分配のない言語で動的分配をやるのは面倒で間違いやすい (X Toolkit は今でもこれをやっている)

3.7 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)
- しかし厳密な「継承の定義」をしはじめると難しい (後述)

□ 継承は何が嬉しいか?

- 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
- 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)

□ 例: 継承を用いた構造化グラフィクス (別資料 Sample8.java)

3.8 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-80 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

4 Smalltalk-80: 中興の祖

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

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

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

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

- プログラミング言語屋にとっては「センセーション」だった
- 多くの「見ため」は Apple の Lisa → Macintosh に引き継がれた (Smalltak-80 言語は引き継がれなかった)

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-80 にはない。この面で拡張を行う研究は多数あり

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: [....].
```

□ 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

5 Lisp 系のオブジェクト指向言語

□ Smalltalk-80 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-80 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-80 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-80 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-80 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 (実際にはもっといろいろ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

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-80、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

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

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

□ ある型に属するかどうかを判定→ instanceof 演算子

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

□ もっと自由に型の情報そのものを扱う→自己反映機能

7 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)

7.1 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」

7.2 Smalltalk-80 のクラスとメタクラス

□ Smalltalk-80 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-80 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-80 システム全体の動作が変化させられる (!!)

7.3 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

7.4 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 Sample9 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    while(true) {
      try {
        System.out.print("Class Name? ");
        System.out.flush();
        String cname = br.readLine();
        if(cname.equals("")) break;
        Class cls = Class.forName(cname);
        Constructor[] cons = cls.getConstructors();
        for(int i = 0; i < cons.length; ++i)
          System.out.println(""+i+": "+cons[i]);
        System.out.print("Constructor Number? ");
        System.out.flush();
        int cno = Integer.parseInt(br.readLine());
        Object obj =
          cons[cno].newInstance(new Object[]{});
        Method[] meths = cls.getMethods();
        for(int i = 0; i < meths.length; ++i)
          System.out.println(""+i+": "+meths[i]);
        System.out.print("Method Number? ");
        System.out.flush();
        int mno = Integer.parseInt(br.readLine());
        Object res =
           meths[mno].invoke(obj, new Object[]{});
        System.out.println("Result Class:"+
           (res.getClass()));
        System.out.println("Result: "+res);
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
}
```

□ これをたとえば次のクラスに対して使ってみる…

```
public class Sample9Test {
  int val;
  public Sample9Test() { val = 1; }
  public Sample9Test(int i) { val = i; }
  public Sample9Test add() {
    return new Sample9Test(val+1);
  }
  public Sample9Test sub() {
    return new Sample9Test(val-1);
  }
  public String toString() {
    return "Sample9Test("+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)

8 継承と委譲

□ 継承 (inheritance): Simula、Smalltalk-80 以来の「由緒正しい」やりかた

- 問題点: 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++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→それが問題の解決になってるのかどうか??

□ 委譲 (delegation): 継承の代替案 (実装としての)

8.1 継承の意味づけ

□ 継承がやってることは何かというと…

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

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

8.2 継承の実装

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

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

8.3 メソッド探索

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

- Smalltalk-80 では、最初にそういう呼び出しがあったときに探索を行い、その情報をキャッシュに保持。クラス構造が変化したときはキャッシュをご破算にしてやりなおす。動的にクラスが変化する環境ならでは
- C++、Java などのコンパイルする言語では、分岐表を作ってそれに基づいて分岐すればよい。分岐表そのもののスロットも変数と同様にして管理可能

8.4 多重継承の実装

□ 多重継承: 2 つの親から継承すること

- 問題: 複数の親が共通の親クラスを持っていたらどうするか?

□ C++の多重継承では 2 つの方式がサポートされているので大変

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

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

8.5 委譲: もう 1 つの実装

□ 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

□ 親の機能が必要なら、親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 (delegation) 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
- 多重継承の実装が用意 (多重継承の実装として、一部に委譲を使うことも)

8.6 Self: コピー方式の OOPL

□ 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

□ 継承ではなく委譲を使う

- 委譲した場合でも「元のオブジェクト」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「コピー方式の」「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

9 インタフェース

□ Smallatalk-80 など弱い型のオブジェクト指向言語では、継承→オブジェクトの互換性を意味していた

- 実は Smalltalk-80 の時代から継承にはいく通りもの使われ型 (実装を借りて来る、概念的に共通、等) があることが分かっていた。

□ しかし、C++など強い型の言語オブジェクト指向言語が使われるようになると、「実装の継承」と「外から見た互換性」は一緒でなくてもよいことが知られるようになってきた

- 「外からみた互換性」もまた、階層構造として取り扱いたい (そうでない言語… 例: Emerald や GNU G++の interface 機構) もあるが
- 現在の潮流→実装の継承と界面 (インタフェース) の継承の分離
- インタフェースの継承のことを subtyping と呼ぶことも

9.1 型とシグニチャ

□ シグニチャ(signature): ある型が持っている (1) メソッドの名前、および (2) 各メソッドの引数と返値の型の集まり

- シグニチャに互換性 (conformance) があれば、型 A の代りに型 B を利用可能
- Emerald はこの互換性のみに基づく型検査を用いている
- しかし一般的なオブジェクト指向言語では型の親子関係に基づいて互換性を定義 (Simula 以来の伝統だから? 「たまたま」互換性が生じるのを嫌うから?)

9.2 インタフェース機能

□ インタフェース: シグニチャを定義するもの

- インタフェース機能を持つメジャーな言語: Java が代表的 (ただし通常の継承も持つ→やや中途半端/保守的)

□ インタフェースの継承もあり→インタフェースの階層構造が定義できる→これに基づいて型の互換性が定まる

- インタフェースの継承においては、多重継承の問題は起きない (実装が存在しないから)。ただし継承すると矛盾が起きる場合には継承はできない。

□ 今後→実装の継承とインタフェースの継承はさらに分離される方向?

□ Java でインタフェースを使った簡単な例:

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

interface Function {
  public double calculate(double x);
}

class Quadratic implements Function {
  double a, b, c;
  public Quadratic(double a0, double b0, double c0) {
    a = a0; b = b0; c = c0;
  }
  public double calculate(double x) {
    return a*x*x + b*x + c;
  }
}

public class Sample10 extends Applet
    implements Runnable {
  boolean running;
  double time;
  public void start() {
    running = true; (new Thread(this)).start();
  }
  public void stop() { running = false; }
  public void run() {
    long basetime = System.currentTimeMillis();
    time = 0.0; repaint();
    while(running) {
      try {
        Thread.sleep(100);
      } catch(Exception e) { }
      time = 0.001 *
        (System.currentTimeMillis() - basetime);
      repaint();
    }
  }
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawLine(0, 100, 200, 100);
    g.drawLine(100, 0, 100, 200);
    g.setColor(Color.blue);
    drawFunc(g,
      new Quadratic(1.5*Math.sin(time), 0.7, -0.5));
  }
  public void drawFunc(Graphics g, Function fn) {
    double x0 = -1.0;
    double y0 = fn.calculate(x0);
    for(int i = 1; i <= 20; ++i) {
      double x = 0.1*i - 1.0;
      double y = fn.calculate(x);
      g.drawLine(100+(int)(100*x0), 100-(int)(100*y0),
                 100+(int)(100*x), 100-(int)(100*y));
```

```
      x0 = x; y0 = y;
    }
  }
}
```

- さまざまな「関数クラス」はそれぞれ、実装の共通部分はほとんどない→インタフェースのみ共通にするのが自然
- さまざまな「実行可能なクラス」も「実行する」メソッド run() を持つ以外には共通部分はなくてよい → Thread オブジェクト生成時に Runnable オブジェクトを引数として渡すと、その run() をスレッドとして実行させるようになっている

9.3 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
- それらのクラスから元のクラスのメソッドを呼ぶのが面倒

□ このため「クラス内部にクラス定義が書ける」ようになった

```
public class X {
    ...
    return new Y();

  class Y implements I {
    ... ←この中で X のメソッドが呼べる
  }
}
```

□ さらに、いちいち「Y」のような名前を考えないで済ませることもできるように「無名の内部クラス」構文がある

```
public class X {
    ...
    return new I() {
      ... ←先の Y の定義と同じもの
    }
}
```

- 記述が短くなるという点は便利だが、最初はちょっと分かりづらい

10 オブジェクト指向言語の諸側面のまとめ

□ 非常に多数の機能がある、ということは分かったと思う

- それらすべてに「発明された理由」はもちろんある
- しかし「それらがすべてが今いるのか?」は別の問題

□ 言語の多様な機能を使えば使うほど「難しく」もなる→「機能の増加」と「簡潔に/分かりやすく記述できる」のトレードオフについて常に考える (例: Java vs C++)

11 オブジェクト指向の利用技術

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

- それをどのように使うか→なかなか難しい問題

□ どのような方向での利用があるか?

11.1 オブジェクト指向にあった設計

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

11.2 再利用

□ 再利用←書くよりは書かないで済ませた方が生産効率はよいに決まっている

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

11.3 クラスライブラリ

□ Smalltalk-80 →充実したクラスライブラリが付属→クラスライブラリを熟知すれば生産性が高まる、というブームに

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

□ Smalltalk-80 クラスライブラリでは差分プログラミング (子クラスで親クラスの機能を少しずつ拡張していく) を多用→これもよい手法だと一時思われていた

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

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

11.4 アプリケーションフレームワーク

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

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

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

□ たとえば、アプレットもアプリケーションフレームワークの 1 つ

- アプレットはクラス Applet のサブクラスとして作り、初期設定メソッド init() 描画メソッド paint() 等を必要に応じてオーバライドする
- HTML 中にアプレットを埋め込む構文:

```
<APPLET CODD="Sample11.class" WIDTH=300 HEIGHT=200></APPLET>
```

□ もっと簡単な「表示覧と 2 つのボタンがある」フレームワークを作ってみました。

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

public class Sample11 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  Sample11Model model = new Sample11App();
  Label lab = new Label(model.getStr());
  Button b1 = new Button(model.leftName());
  Button b2 = new Button(model.rightName());
  public void init() {
    setLayout(null);
    add(lab); lab.setFont(fn);
    lab.setLocation(20, 20); lab.setSize(200, 40);
    lab.setBackground(Color.white);
    add(b1); b1.setFont(fn);
    b1.setLocation(20, 80); b1.setSize(60, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        model.left(); repaint();
      }
    });
    add(b2); b2.setFont(fn);
    b2.setLocation(120, 80); b2.setSize(60, 40);
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        model.right(); repaint();
      }
    });
  }
  public void paint(Graphics g) {
    lab.setText(model.getStr());
  }
}

interface Sample11Model {
  public String leftName();
  public String rightName();
  public void left();
  public void right();
  public String getStr();
}

class Sample11App implements Sample11Model {
  String str = "STR";
  public String leftName() { return "A"; }
  public String rightName() { return "B"; }
  public void left() { str += "A"; }
  public void right() { str += "B"; }
  public String getStr() { return str; }
}
```

11.5 コンポーネント

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

□ プログラミングをしない人に使える再利用技術→インスタンスベースの再利用

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

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

- ほかにも国産の IntelligentPad、Java ベースの JavaBeans などいろいろある

- しかし、部品とその配線だけでできるプログラムで十分なの?

11.6 デザインパターン

□ パターン: 「繰り返し現われるようなカタチ」

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

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

□ 例: Command パターン

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

□ 例: Adapter パターン

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

□ 実は上記 2 つは前の例題に含まれていた。

□ 例: Visitor パターン

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

□ Visitor パターンとは:

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

□ 例: AbstractFactory パターン

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

□ AbstractFactory パターンとは:

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

11.7 本節のまとめ

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

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

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

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

12 第 2 回課題

□ 以下の課題から 1 つ以上 (4 を含む場合は 2 つ以上) を選択してプログラムを作成し、実験を行ってレポートを提出せよ。考察まできちんと書くこと。

□ (1) Java の例外機構の処理速度を計測せよ。通常のメソッド呼び出しについても計測し、比較すること。同じ場所で処理する例外の個数、try ... catch の入れ子数、手続きを戻る個数による影響も考慮すること。それに基づき、どのような場合に例外処理が適し、どのよう

な場合には適さないかを考察すること。自分が使用した Java 実行系の例外処理方式が推察できるとなおよい。

- □ (2) 自己反映機能の例題を改良して、「オブジェクト電卓」にせよ。つまり生成したオブジェクトをいくつか「電卓のメモリ」に入れておいて、あるオブジェクトのメソッドを呼ぶのにメモリに入っているオブジェクトを引数として渡せるようにする。これにより、任意の Java オブジェクトをプログラムを書かずにいじって見ることができる。
- □ (3) 「簡単なアプリケーションフレームワークの例題」に入れる別のモデル (アプリケーション) を作って差し替えてみよ。また、もうすこし役に立つことができるようにこのアプリケーションフレームワークを改良してみよ。
- □ (4) その他、Java を用いて「オブジェクト指向言語らしい」プログラム (アプレットでもアプリケーションでもよい) を作成せよ。

参考文献

[1] リメイ，パーキンス著，武舎ほか訳，Java 言語入門，プレンティスホール出版，1995.

[2] モリソン，エイブラン著，福井ほか訳，続· Java 言語入門，プレンティスホール出版，1998.

[3] Goldberg, Robson, Smalltalk-80 --- The Language, Addison-Wesley, 1989.

[4] ガンマ他著，本位田・吉田監訳，オブジェクト指向における再利用のためのデザインパターン，ソフトバンク，1995.