22–12 ほぼ自習 ジェネリック型

例題19.6や例22.1で定義したStackOfAnyObjectsクラスに見られる様に、 構成要素をObject型に設定 して様々な種類のデータを扱える汎用データ構造を構成することがある。

しかし、この様にすると汎用性と引き換えに次の様な不満点も発生する。

- コンパイル時のエラーチェックが甘くなる。
  (本来とは違う型のデータを汎用データ構造側に渡しても
  コンパイルエラーとならない。)

- 汎用データ構造を利用するプログラムが多少煩雑になる。
  (汎用データ構造を利用するプログラムの中で 常に要素
  データの型を認識し、汎用データ構造からデータを取り
  出す際は適切な型へのキャストを行う必要がある。)

例22．17（Object型を使った汎用スタックの不満点）　例22.1で定義されているStackOfAnyObjectsクラスを利用する単純なjavaプログラム...

[motoki@x205a]$ cat -n AbuseOfStackOfAnyObjectsMain.java

```
 1 /*
 2  * StackOfAnyObjectsクラスでは、注意深い使用が必要で、
 3  * 誤用してもコンパイル時に見落とされ実行時に初めてエラー
 4  * となることもある、ということを例示
 5  */
 6 public class AbuseOfStackOfAnyObjectsMain {
 7     public static void main(String args[]) {
 8         StackOfAnyObjects stack = new StackOfAnyObjects
 9
10         stack.pushdown("123");                    ←無チェック
11         //  ...(しばらく後に)...      要キャスト
12         Integer someInteger = (Integer) stack.popup();
13         System.out.println("someInteger = "
                                          + someInteger);
14     }
15 }
```

```
[motoki@x205a]$  javac AbuseOfStackOfAnyObjectsMain.java
[motoki@x205a]$  java AbuseOfStackOfAnyObjectsMain
Exception in thread "main" java.lang.ClassCastException:
        java.lang.String cannot be cast to java.lang.Integer
        at AbuseOfStackOfAnyObjectsMain.
                main(AbuseOfStackOfAnyObjectsMain.java:12)
[motoki@x205a]$
```

注目するのは次の2点

- コンパイルは通るが実行時エラーとなる：

- プログラム 12行目 の様に、適宜キャスト演算を施す必要がある：

試しにプログラム 10~ 12行目 を

```
10                  stack.pushdown(new Integer(123));
11                  //  ...(しばらく後に)...
12                  Integer someInteger = stack.popup();
```

という風に変えてコンパイルし直すと 、次の様にエラーが検出

```
[motoki@x205a]$ javac AbuseOfStackOfAnyObjectsMain2.java
AbuseOfStackOfAnyObjectsMain2.java:12: 互換性のない型
検出値   : java.lang.Object
期待値   : java.lang.Integer
        Integer someInteger = stack.popup();
                                    ^
エラー 1 個
```

⇨ 上記の不満点を解消するために、 JDK1.5以降では

ジェネリッククラス... クラスに型パラメータを設け、インスタンス内部で想定する基本要素の型を型パラメータで指定できる

ジェネリック型... ジェネリッククラスから生成されるオブジェクトの型

ジェネリッククラス定義の形式：通常次の様な形式

修飾子 class クラス名 < 型パラメータ名 , ..., 型パラメータ名 > .

{

フィールドの宣言、メソッドの定義、など

}

ここで、

- Java文法上は、各々の 型パラメータ名 に任意の識別子を用いることができる。しかし、慣習上、型パラメータ名 は単一の英大文字、特に...
  - E ··· 要素(element)の型を表す場合
    （「コレクションフレームワーク」でよく使われている。）
  - K ··· キー(key)の型を表す場合
  - N ··· 数(number)の型を表す場合
  - V ··· 値(value)の型を表す場合
  - T ··· 一般的な型(type)を表す場合
  - S,U,V,... ··· 第2, 第3, 第4, ...の型を表す場合

- 各々の 型パラメータ名 を実在する特定の型（型引数という）で置き換えて得られる、

  クラス名 < 型引数 , ..., 型引数 >

  という形のものがジェネリック型。

- 型パラメータに対応付ける型引数を限定する書き方 もある。例えば、Comparableインタフェースを実装した型引数に限定したい場合は

  型パラメータ名 の部分を

  ```
  E extends Comparable<E>
  ```

  や

  ```
  E extends Comparable<? super E>
  ```

- クラス定義の本体部では、具体的な型名を書けるほとんどの場所に型パラメータを書くことができる。(例外もある→次の項)

- 1つのジェネリッククラスの定義によって導入されるクラスは1個だけ
  ⇨ クラス定義の本体部の書き方に次の様な制約
  ◇ staticフィールドの型に型パラメータを使用不可。

  ◇ staticメソッド内やstatic初期化子で型パラメータを使用不可。

  ◇ 型パラメータで指定された型のオブジェクトを直接生成できない。
  例えば、 `new E[size]` という書き方は許されず 、代わりに
  `(E[]) new Object[size]`
  といった書き方をする。

- 型パラメータや型引数の情報は、オブジェクトが正しく使われているかどうかをチェックするためにコンパイラによって使用される。しかし、個々のインスタンスは自分自身の属するジェネリック型の情報を内部に持たない。

例22．18（ジェネリック版スタック）例22.1で示したStackOfAnyObjectクラスをジェネリック化してStackGenericというクラスを定義

```
[motoki@x205a]$  cat -n StackGeneric.java
     1 import java.util.*;
     2
     3 /**
     4  *  ジェネリック版 pushdownスタックのクラス
     5  * @author 元木達也
     6  * @version 0.0
     7  */
     8 public class  StackGeneric<E>  {
     9     /** 初期容量のデフォルト値 */
    10     private static final int
                              DEFAULT_INITIAL_CAPACITY = 100;
    11
    12     /** 容量不足の際に増やす容量のデフォルト値 */
    13     private static final int
                            DEFAULT_CAPACITY_INCREMENT = 100;
    14
```

```
15      /** Eインスタンス(への参照)を格納するための配列領域 */
16      private  E [] stack;
17
18      /** スタックの最も上部の要素が格納されている位置... */
19      private int indexOfTopEle;
20
21      /**
22       * 空のスタックを構成する
23       */
24      public  StackGeneric () {
25          this(DEFAULT_INITIAL_CAPACITY);
26      }
27
28      /**
29       * 空のスタックを構成する
30       * @param initialCapacity スタックの初期容量
31       */
32       @SuppressWarnings("unchecked")
33      public  StackGeneric (int initialCapacity) {
```

```
34                  //配列stackにはpushdown()メソッドによってEクラスの
35                  //インスタンスのみが格納されるので、次のキャストは安全。
36          stack =  (E[])  new Object[initialCapacity];
37          indexOfTopEle = -1;
38      }
39
40      /**
41       * スタックオブジェクトの標準的な文字列表現を求める
42       * @return 標準的な文字列表現
43       */
44      @Override
45      public String toString() {
46          return "pushdownStack ( generic_type , capacity
47                  + ", currentNumOfEle="
                            + (indexOfTopEle+1) + ")";
48      }
49
```

```
50      /**
51       * スタックに格納されている要素の情報を得る
52       * @return スタックの内容を表す文字列
53       */
54      public String getDetailedConfig() {
55          String result = "stack contains "
                    + (indexOfTopEle+1) + " elements: {";
56          for (int i=0; i<indexOfTopEle; ++i)
57              result += "\n    " + stack[i] + ",";
58          if (indexOfTopEle >= 0)
59              result += "\n    " + stack[indexOfTopEle];
60          result += " }";
61          return result;
62      }
63
64      /**
65       * スタックが空かどうかを調べる
66       * @return スタックが空かどうか
67       */
```

```
68      public boolean isEmpty() {
69          return indexOfTopEle == -1;
70      }
71
72      /**
73       * 新しい E 要素をスタックにpush-downする
74       * @param element スタックにpush-downする新要素
75       */
76      public void pushdown( E  element) {
77          if (indexOfTopEle+1 == stack.length) {
78              stack = Arrays.copyOf(stack, stack.length
79                                    + DEFAULT_CAPACITY_INCREMENT);
80              System.out.printf("###Stack capacity is inc
81                                        "#<New> %s%n", this);
82          }
83          stack[++indexOfTopEle] = element;
84      }
85
86      /**
```

```
 87      * スタックから最も上部の要素を取り出す
 88      * @return スタックの最も上部の要素
 89      */
 90     public  E  popup() {
 91         if (indexOfTopEle < 0)
 92             throw new EmptyStackException();
 93          E  element = stack[indexOfTopEle];
 94         stack[indexOfTopEle--] = null;
 79                                  //取り出した要素への参照を解除
 95         return element;
 96     }
 97
 98     /**
 99      * スタックに格納された要素の個数を調べる
100      * @return スタックに格納された要素の個数
101      */
102     public int getNumOfEle() {
103         return indexOfTopEle+1;
104     }
```

```
105
106     /**
107      * スタックのtop要素をのぞき見
108      * @return スタックのtop要素(への参照)
109      */
110     public  E  peepTop() {
111         if (isEmpty())
112             return null;
113         else
114             return stack[indexOfTopEle];
115     }
116
117     /**
118      * スタックの指定要素をのぞき見
119      * @param index のぞき見したいスタック要素の番号
120      * @return 番号indexのスタック要素(への参照)
121      */
122     public  E  peepEleOfIndex(int index) {
123         return stack[index];
```

```
   124     }
   125 }
[motoki@x205a]$  cat -n TestStackGenericMain.java
     1 /**
     2  * StackGenericクラスの動作を確認するためのJavaプログ...
     3  */
     4 public class TestStackGenericMain {
     5     public static void main(String args[]) {
     6         StackGeneric<String> stack1 =
                                    new StackGeneric<String>();
     7         stack1.pushdown("a");
     8         stack1.pushdown("bcd");
     9         stack1.pushdown("efg");
    10         stack1.pushdown("hij");
    11         System.out.println(stack1.popup() + ", " +
    12                            stack1.popup() + ", " +
    13                            stack1.popup());
    14
    15         StackGeneric<Integer> stack2 =
```

```
                                        new StackGeneric<Integer>(2);
    16          stack2.pushdown(new Integer(1));
    17          stack2.pushdown(new Integer(2));
    18          stack2.pushdown(new Integer(3));
    19          System.out.println(stack2.popup() + ", " +
    20                             stack2.popup() + ", " +
    21                             stack2.popup());
    22      }
    23 }
[motoki@x205a]$  javac TestStackGenericMain.java
[motoki@x205a]$  java TestStackGenericMain
hij, efg, bcd
###Stack capacity is increased###
#<New> pushdownStack (generic_type, capacity=102, currentNumOf
3, 2, 1
[motoki@x205a]$
```

ここで、例22.1で示したStackOfAnyObjects.javaから変更した箇所を下線で表している。

- **32行目 ··· “SuppressWarnings("unchecked")” というアノテーションを挿入。これによって次のコンストラクタをコンパイル時に「unchecked」という種類の警告が出るのを抑制する。**

補足： この行をコメントアウトしてコンパイルすると、次の様に警告が出る。

```
|[motokix205a]$  javac StackGeneric.java
|注:StackGeneric.java の操作は、未チェックまたは安全ではありません。
|注:詳細については、-Xlint:unchecked オプションを指定して
                                        再コンパイルしてください。
```

更に、この警告文に従って再コンパイルすると、...

```
|[motokix205a]$  javac -Xlint:unchecked StackGeneric.java
|StackGeneric.java:36: 警告:[unchecked] 無検査キャストです
|検出値 : java.lang.Object[]
|期待値 : E[]
|        stack = (E[]) new Object[initialCapacity];
|                      ^
|警告 1 個
```

22–13 ほぼ自習 jarファイル，クラスパスの指

C言語では
.oファイル群を1つのライブラリファイル(.aファイル)に纏めることができた。⟶Javaでも同様のことが可能

jarファイル： Javaでは、
複数のクラスファイルやリソース(e.g.gifファイル)を単一のアーカイブファイル(jarファイルという)に纏めておくことができる。

- jarファイルの拡張子は .jar 。
- jarファイルはクラスファイル群の公開・配付に利用されている。実際、単に「ライブラリ」と言えばjarファイルのことを指す。
- jarファイルはZIP形式で内容を保持する。
- jarファイルを作成したり操作したりするために、JDKの中にjarコマンドが用意されている。(コマンドの書式はtarコマンドに類似。)

jarファイルの作成： jarコマンドを次の様な形式で使う。

jar cvf [jarファイル名] [jarファイルに含めたいファイルのリスト]

ここで、

- jarコマンドのキーの意味は次の通り。
  - c ··· create(作成)。
  - v ··· verbose(詳細報告)。省略可。
  - f ··· file(標準出力でなくファイルに出力を送る)。

- [jarファイルに含めたいファイルのリスト] は
  jarファイルに含めたいファイルの名前を
  空白で区切って並べた文字列
  を表す。この中で、
  - * ··· 全ファイルの意味
  - ディレクトリ ··· ディレクトリ以下の内容が再帰的に追加

---

jarファイルの内容表示： jarコマンドを次の様な形式で使う。

```
jar tvf [jarファイル名]
```

ここで、

- jarコマンドのキーの意味は次の通り。
  - t ··· table(一覧表)。
  - v ··· verbose(詳細報告)。省略可。
  - f ··· file(内容表示するjarファイルがコマンドラインで指定されていることを表す)。

- [jarファイル名] には内容表示したいjarファイルの(パスと)名前を指定する。

---

jarファイルの内容抽出： jarコマンドを次の様な形式で使う。

jar xvf [jarファイル名] [jarファイルから抽出したいファイルのリスト, 省略可]

ここで、

- jarコマンドのキーの意味は次の通り。
  - x ··· extract(抽出)。
  - v ··· verbose(詳細報告)。省略可。
  - f ··· file(抽出元のjarファイルがコマンドラインで指定されていることを表す)。

- [jarファイル名] には抽出元のjarファイルの(パスと)名前を指定する。

- [jarファイルから抽出したいファイルのリスト, 省略可] は
  jarファイルから抽出したいファイルの名前を
  空白で区切って並べた文字列
  を表す。
  省略した場合 ⟶jarファイル中の全ファイル

例22．19（自分専用のライブラリを用意）

例題22.7 で定義した HeapsortIntArrayクラス,
BubblesortIntArrayクラス,
LListsortIntArrayクラス,
例題22.15 で定義した StopWatchクラス,
例22.18で定義した StackGenericクラス
は汎用性があり、将来再利用する可能性も大いにある。

⇨ ◇ これらに関連するクラスの.classファイルを自分専用のライブラリとして ../mylib というディレクトリの中にまとめて置いておく。

◇ 整列化関連の.classファイルは全てsortIntArray.jar という名前のjarファイルにまとめて保管する。

```
[motoki@x205a]$  ls *.class
ls: *.class にアクセスできません: そのようなファイルやディレクトリはありません
[motoki@x205a]$  javac StopWatch.java
[motoki@x205a]$  javac *sortIntArray.java
[motoki@x205a]$  javac StackGeneric.java
[motoki@x205a]$  ls *.class
BubblesortIntArray.class  LinkedListOfInt$Node.class   StackGe
HeapsortIntArray.class    LinkedListOfInt.class        StopWat
LListsortIntArray.class   SortModuleForIntArray.class
[motoki@x205a]$  jar cvf sortIntArray.jar
                                        BubblesortIntArray.class \
    HeapsortIntArray.class LListsortIntArray.class
                                        LinkedListOfInt\$Node.class \
    LinkedListOfInt.class SortModuleForIntArray.class
マニフェストが追加されました。
BubblesortIntArray.class を追加中です。(入 = 1441) (出 = 840)(41%
```

縮されました)
HeapsortIntArray.class を追加中です。(入 = 1612) (出 = 958)(40%
縮されました)
LListsortIntArray.class を追加中です。(入 = 1511) (出 = 891)(41%
縮されました)
LinkedListOfInt$Node.class を追加中です。(入 = 1049) (出 = 612)(4
縮されました)
LinkedListOfInt.class を追加中です。(入 = 851) (出 = 540)(36% 収
縮されました)
SortModuleForIntArray.class を追加中です。(入 = 363) (出 = 252)(3
縮されました)
[motoki@x205a]$ jar tvf sortIntArray.jar
     0 Sat Jan 21 18:20:36 JST 2012 META-INF/
    71 Sat Jan 21 18:20:36 JST 2012 META-INF/MANIFEST.MF
  1441 Sat Jan 21 17:25:18 JST 2012 BubblesortIntArray.class
  1612 Sat Jan 21 17:25:18 JST 2012 HeapsortIntArray.class
  1511 Sat Jan 21 17:25:18 JST 2012 LListsortIntArray.class

```
  1049 Sat Jan 21 17:25:18 JST 2012 LinkedListOfInt$Node.class
   851 Sat Jan 21 17:25:18 JST 2012 LinkedListOfInt.class
   363 Sat Jan 21 17:25:18 JST 2012 SortModuleForIntArray.clas
[motoki@x205a]$  mkdir ../mylib
[motoki@x205a]$  mv sortIntArray.jar ../mylib
[motoki@x205a]$  cp StackGeneric.class ../mylib
[motoki@x205a]$  cp StopWatch.class ../mylib
[motoki@x205a]$  ls ../mylib
StackGeneric.class  StopWatch.class  sortIntArray.jar
[motoki@x205a]$
```

クラスパスの指定： 標準パッケージやimport宣言されたパッケージの他に別クラスへの参照がある場合、 コンパイルや実行の際に 、
デフォルトでは、⟶カレントディレクトリ内から
必要なクラス定義や.classファイルが探される。

カレントディレクトリ以外の場所を探してもらいたい場合は、
⟶次のいずれかの方法 （両方指定⟶(方法1)が優先）

(方法1) コマンドのオプション指定 ：
javacコマンドやjavaコマンドを実行する際に
-classpath クラスパスの指定
または
-cp クラスパスの指定
という形のオプションを指定する。ここで、クラスパスの指定 の部分は探してもらいたいディレクトリやjarファイルのパス指定をコロン(:)で区切って並べた文字列で表す。（途中の空白は許されない。）

(方法2) 環境変数CLASSPATHの設定 ：

（方法2）環境変数CLASSPATHの設定 ：

bash を使っている場合はコマンドライン上で例えば

CLASSPATH= クラスパスの指定; export CLASSPATH

とし、 tcsh を使っている場合はコマンドライン上で

setenv CLASSPATH クラスパスの指定

とする。ここでも、クラスパスの指定 の部分はパスの明示されたディレクトリやjarファイルをコロン(:)で区切って並べた文字列である。

例22．20（クラスパスを指定してライブラリ内のStackGenericを利用）

先の例22.19で自分専用のライブラリの要素として登録した
../../mylib/StackGeneric.classを利用して、
例22.18で定義したTestStackGenericMain.java をコンパイル・実行

```
[motoki@x205a]$  ls
TestStackGenericMain.java          TimerForSortModuleIntArray.j
TimeSortModulesIntArrayMain.java
[motoki@x205a]$  ls ../../mylib
StackGeneric.class  StopWatch.class  sortIntArray.jar
[motoki@x205a]$  javac -cp ../../mylib TestStackGenericMain.ja

[motoki@x205a]$  java -cp .:../../mylib TestStackGenericMain
hij, efg, bcd
###Stack capacity is increased###
#<New> pushdownStack (generic_type, capacity=102, currentNumOf
3, 2, 1
[motoki@x205a]$
```

ここで、

- 最後の java コマンド で、
  クラスパスにカレントディレクトリ (.) を追加しないと、...

```
[motoki@x205a]$ java -cp ../../mylib TestStackGenericMain
Exception in thread "main" java.lang.NoClassDefFoundError: T
Caused by: java.lang.ClassNotFoundException: TestStackGeneri
at java.net.URLClassLoader$1.run(URLClassLoader.java:217)
at java.security.AccessController.doPrivileged(Native Method
at java.net.URLClassLoader.findClass(URLClassLoader.java:205
at java.lang.ClassLoader.loadClass(ClassLoader.java:321)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:
at java.lang.ClassLoader.loadClass(ClassLoader.java:266)
Could not find the main class: TestStackGenericMain. Program
[motoki@x205a]$
```

例22．21（クラスパスを指定してライブラリ内の各種ファイルを利用）

先の例22.19で構成した自分専用のライブラリ
(../../mylib内の.classファイルとjarファイル)を利用して、例題22.15 で定義した TimeSortModulesIntArrayMain.java (とそれに関連しているjavaソースファイル)をコンパイル・実行

```
[motoki@x205a]$  ls
TestStackGenericMain.class  TimeSortModulesIntArrayMain.java
TestStackGenericMain.java   TimerForSortModuleIntArray.java
[motoki@x205a]$  ls ../../mylib
StackGeneric.class  StopWatch.class  sortIntArray.jar
[motoki@x205a]$
     javac -cp .:../../mylib:../../mylib/sortIntArray.jar \
                              TimeSortModulesIntArrayMain.java
[motoki@x205a]$  ls
TestStackGenericMain.class          TimerForSortModuleIntArray$
```

```
TestStackGenericMain.java          TimerForSortModuleIntArray.
TimeSortModulesIntArrayMain.class  TimerForSortModuleIntArray.
TimeSortModulesIntArrayMain.java
[motoki@x205a]$
      java -cp .:../../mylib:../../mylib/sortIntArray.jar \
                                    TimeSortModulesIntArrayMain
Clocking the average execution time of the module
that sorts 5, 10, 25, 50, 100, or 200 elements.
  (*** Heapsort module ***)
Input a random seed (0 - 92233720368547758O7):  333

        ** time for sort **    **time for initialize**
size    cpu_time   real_time    cpu_time   real_time
         (m sec)    (m sec)     (m sec)    (m sec)
----    ---------  ---------   ---------  ---------
   5      0.00008    0.00006     0.00014    0.00015
  10      0.00023    0.00026     0.00025    0.00024
```

```
  25        0.00100     0.00098        0.00056     0.00058
  50        0.00250     0.00244        0.00113     0.00115
 100        0.00550     0.00573        0.00250     0.00233
 200        0.01300     0.01295        0.00450     0.00470
            (途中省略)
 200        0.02300     0.02265        0.00400     0.00425
[motoki@x205a]$
```

22–14 ほぼ自習 パッケージ管理

- プログラムが大規模になり
- 関連する.classファイルやjarファイルが
  あちこちのディレクトリに分散配置される様になると、...

⇨ 大量のソースコードの管理も難しくなる。

- ◇ クラスパスの指定も大変。
- ◇ 目的のクラスを探し出す手間が増える。
- ◇ クラス名の衝突の可能性も出てくる。

⇨ 「パッケージ」と呼ばれる仕組み

- ◇ クラス間のアクセス制御を行うためでもある

パッケージ管理の機構：

機能面，目的面で互いに関連したクラスやインタフェースを1つのグループ（パッケージ）にまとめて管理することができる。

- 階層構造を持つことができる。
  但し、名前を区切るための記号としてピリオド（.）

- ソースプログラムの先頭にpackage文
  ....定義したクラス（やインタフェース）の所属パッケージを指定
  package文のないプログラムの場合 $\longrightarrow$ 無名パッケージに所属

  補足（無名パッケージの用途）：
  小規模，一時的なアプリケーション、開発の初期段階

- コンパイルや実行の際には 、 ...

パッケージ管理の機構：

機能面，目的面で互いに関連したクラスやインタフェースを1つのグループ（パッケージ）にまとめて管理することができる。

- 階層構造を持つことができる。
  但し、名前を区切るための記号としてピリオド（.）
- ソースプログラムの先頭にpackage文
  ....定義したクラス（やインタフェース）の所属パッケージを指定
- コンパイルや実行の際には 、
  - ◇ パッケージ名中のピリオド（.）で区切られた名前の各々はディレクトリ名として解釈される。
  - ◇ 環境変数CLASSPATHで指定されたクラスパスを起点に、パッケージ名の表すディレクトリ階層をたどって必要な.classファイルが探されることになる。

  ⇨ パッケージ名の階層構造に合致した形で、実際のファイルシステム上に.classファイル群を階層的に構成しておく必要

標準パッケージ：

多くの標準パッケージが備わっている。
（全てjavaパッケージのサブパッケージ） 例えば、

- java.lang ··· Object, String, Math, Thread, Class 等の基本的なクラスを含むコアパッケージで、この中で定義されたクラスは（import宣言無しでも）自由に使える。
- java.io ··· 入出力, ファイル操作に関連したクラスから成る。
- java.util ··· 一般的なユーティリティのためのクラスから成る。
- java.awt ··· GUIを記述するためのクラスから成る。
  （Abstract Window Toolkit）
- java.applet ··· アプレットを記述するためのクラスから成る。
- java.beans ··· JavaBeansコンポーネントアーキテクチャにおける、独立したソフトウェアコンポーネントを記述するためのクラスから成る。

- java.lang.instrument ··· 仮想マシン上で動作しているアプリケーションを計測できるエージェント定義するためのクラスから成る。
- java.lang.management ··· 仮想マシンとその上のOSを監視・管理するためのクラスから成る。
- java.math ··· 任意精度の算術計算等のためのクラスから成る。
- java.net ··· ソケットやURL等のネットワークの基盤を扱うためのクラスから成る。
- java.nio ··· 通常より複雑だが高性能な入出力(New I/O)のためのクラスから成る。
- java.nio.charset ··· 文字セットとそのエンコーディングを定義しているクラスから成る。
- java.rmi ··· Remote Method Invocation。他ホスト上の他仮想マシンからのメソッド呼び出しも可能なオブジェクトを生成す

るためのクラスから成る。

- java.security ··· セキュリティに関連したクラスから成る。
- java.sql ··· 関係データベースを使用するためのJDBC(Java Database Connectivity)パッケージ。
- java.text ··· 数字や日付の書式・解析、文字列のソート、キーによるメッセージ検索等のためのクラスから成る。
- java.util.concurrent ··· 効率的なマルチスレッドのアプリケーションを書くためのユーティリティから成る。
- java.util.jar ··· jarファイルを読み書きするためのクラスから成る。
- java.util.logging ··· コード内からロギングするためのフレームワークを提供する。
- java.util.prefs ··· アプリケーションの実行状況やユーザとシステムの設定を管理するための仕組みを提供する。

- java.util.zip ··· ZIPファイルを読み書きするためのクラスから成る。
- java.util.regex ··· 正規表現を扱うためのクラスから成る。

更に、標準拡張と呼ばれている次の様なパッケージもある。

- javax.accessibility ··· 障害者が使用可能なGUIを開発するためのフレームワークを提供する。
- javax.naming ··· ディレクトリとネーミングサービスを扱うためのクラス, サブパッケージから成る。
- javax.sound ··· ディジタルサウンドを扱うためのサブパッケージから成る。
- javax.swing ··· GUIコンポーネントを扱うためのクラスから成る。(awtはネイティブのGUIに依存しているが、こちらは全てのシステム上で出来るだけ同じ様に見え振舞う様に書かれている。)

所属パッケージの指定：

クラスやインタフェースの所属するパッケージを指定したい場合は、ソースプログラムの先頭に次の形式のpackage文を書く。

```
package パッケージ名 ;
```

ここで、

- パッケージ名 の付け方 に関しては、次の様な指針が一般的
  - ◇ ピリオド(.)で区切られた名前の部分には英小文字だけを使う。
  - ◇ 会社や組織の場合、そのインターネットドメイン名の構成要素を逆順に並べた文字列(e.g. jp.ac.niigata_u.ie.ce.)でパッケージ名を開始する。その後に、地域やプロジェクトの名前も挿入して、名前の衝突を未然に防ぐ。

> 補足：
> 世界中の様々な人達の作ったプログラムが公開される可能性
> $\Longrightarrow$ 面倒な手間無しでこれらを安全に利用したい

所属パッケージの指定：

クラスやインタフェースの所属するパッケージを指定したい場合は、
ソースプログラムの先頭に次の形式のpackage文を書く。

package　パッケージ名　;

ここで、

- パッケージ名 の付け方 に関しては、次の様な指針が一般的

. . . . . .

- 所属パッケージの指定されたクラスについては、

  パッケージ名 . クラス名

  という名前（完全限定名という）で一意にクラスを特定できる

⇨ 環境変数 CLASSPATHが適切に設定されている場合 、

◇ 完全限定名を使えば、別パッケージからでもクラスを参照可

◇ どのディレクトリにいても、

java　クラスの完全限定名

というコマンド入力で指定クラスのmainメソッドを起動可

例22．22（パッケージへの登録）

例題22.7 で定義した HeapsortIntArrayクラス,
Bubble sortIntArrayクラス,
LListsortIntArrayクラス,
例題22.15 で定義した StopWatchクラス,
例22.18で定義した StackGenericクラス
は汎用性があり、将来再利用する可能性も大いにある。

⇨ ◇ 関連するクラスを mypackage というパッケージに登録して、
◇ 色々な場所からこのパッケージの中のクラスを利用できる
様にしたい。

⇨ 次のことを行えば良い。

. . . . . .

例22.22(パッケージへの登録)

例題22.7 で定義した HeapsortIntArrayクラス,

.....

例22.18で定義した StackGenericクラス

は汎用性があり、将来再利用する可能性も大いにある。

⇨ .....

⇨ 次のことを行えば良い。

(1) 登録したいクラスを定義したソースプログラム群をディレクトリ

~/C-Java2012/Programs-Java/mypackage

の中にコピー

CLASSPATHで~/C-Java2012/Programs-Java が指定されていると仮定

(2) 各々のソースプログラムの先頭に次の行を挿入。

```
package mypackage;
```

(3) コマンドライン上で

```
CLASSPATH=~/C-Java2012/Programs-Java; export CLASSPATH
```

(4) コンパイル。

補足： .classファイルが出来た後は、ソースファイルは別の場所で管理しても良い。

実際に以上の (1),(2) を行った状況下で 、
簡単な確認作業と (3),(4) の作業を行っている様子

```
[motoki@x205a]$  pwd
/home/motoki/C-Java2012/Programs-Java/mypackage
[motoki@x205a]$  ls
BubblesortIntArray.java  LinkedListOfInt.java          StopWatch
HeapsortIntArray.java    SortModuleForIntArray.java
LListsortIntArray.java   StackGeneric.java
[motoki@x205a]$  cat BubblesortIntArray.java
package mypackage;
```

以下、例題 22.7 の BubblesortIntArray.java と同じ

```
[motoki@x205a]$  cat HeapsortIntArray.java
package mypackage;
```

以下、例題 22.7 の HeapsortIntArray.java と同じ

```
[motoki@x205a]$  cat LListsortIntArray.java
```

```
package mypackage;
```

以下、例題22.7 の LListsortIntArray.java と同じ

```
[motoki@x205a]$  cat LinkedListOfInt.java
package mypackage;
```

以下、例題22.7 の LinkedListOfInt.java と同じ

```
[motoki@x205a]$  cat SortModuleForIntArray.java
package mypackage;
```

以下、例題22.7 の SortModuleForIntArray.java と同じ

```
[motoki@x205a]$  cat StackGeneric.java
package mypackage;
```

以下、例題22.18 の StackGeneric.java と同じ

```
[motoki@x205a]$  cat StopWatch.java
package mypackage;
```

以下、例題22.15 の StopWatch.java と同じ

```
[motoki@x205a]$
     CLASSPATH=~/C-Java2012/Programs-Java; export CLASSPATH
[motoki@x205a]$  javac *.java
[motoki@x205a]$  ls
BubblesortIntArray.class     LinkedListOfInt.java
BubblesortIntArray.java      SortModuleForIntArray.class
HeapsortIntArray.class       SortModuleForIntArray.java
HeapsortIntArray.java        StackGeneric.class
LListsortIntArray.class      StackGeneric.java
LListsortIntArray.java       StopWatch.class
LinkedListOfInt$Node.class   StopWatch.java
LinkedListOfInt.class
[motoki@x205a]$
```

- 環境変数 CLASSPATH が適切に設定されていないと、...

```
[motoki@x205a]$  javac BubblesortIntArray.java
BubblesortIntArray.java:7: シンボルを見つけられません。
シンボル: クラス SortModuleForIntArray
public class BubblesortIntArray extends SortModuleForIntArra
                                        ^
BubblesortIntArray.java:25: メソッドはスーパータイプのメソッド
をオーバーライドまたは実装しません
    @Override
    ^
BubblesortIntArray.java:31: メソッドはスーパータイプのメソッド
をオーバーライドまたは実装しません
    @Override
    ^
エラー 3 個
[motoki@x205a]$
```

別パッケージ内のクラスの利用：

ソースプログラムの最初の方に次の形式のimport文を置いておくと、クラスの完全限定名の代わりに単純なクラス名でクラスを参照できる様になる。

import パッケージ名 . クラス名 ; または import パッケージ名

ここで、

- 状況に応じて使い分けるのが良い。
  - ◇ クラス名も示す 書き方は、どのクラスを使うのかが明確になる。
  - ◇ ワイルドカードを使う 書き方は、パッケージ内の多数のクラスを利用したい時に簡潔にimport宣言できる。

- 同一パッケージ内のクラスを利用する場合 はimport宣言は不要である。
  （java.lang標準パッケージと同様に自動的にimportされる。）
  しかし、コンパイル前に環境変数CLASSPATHの設定を忘れずに行っておく必要がある。

例22．23（パッケージ内のクラスの利用）

プログラムの中から先の例22.22で構成したmypackageパッケージ内のStackGeneric というクラスを利用できる様にするためには、
プログラムの前の方に

```
import mypackage.StackGeneric;
```

という行を挿入すれば良い。
これを例22.18で定義したTestStackGenericMain.java に対して行なった状況下で 、
簡単な確認作業とその改変版をコンパイル・実行している様子

```
[motoki@x205a]$  pwd
/home/motoki/C-Java2012/Programs-Java/objectoriented/example_p
[motoki@x205a]$  ls
TestStackGenericMain.java          TimerForSortModuleIntArray.j
TimeSortModulesIntArrayMain.java
```

```
[motoki@x205a]$  cat TestStackGenericMain.java
import mypackage.StackGeneric;
```

以下、例題22.18 の TestStackGenericMain.java と同じ

```
[motoki@x205a]$
   CLASSPATH=~/C-Java2012/Programs-Java:.; export CLASSPATH
[motoki@x205a]$  javac TestStackGenericMain.java
[motoki@x205a]$  ls
TestStackGenericMain.class  TimeSortModulesIntArrayMain.java
TestStackGenericMain.java   TimerForSortModuleIntArray.java
[motoki@x205a]$  java TestStackGenericMain
hij, efg, bcd
###Stack capacity is increased###
#<New> pushdownStack (generic_type, capacity=102, currentNumOf
3, 2, 1
[motoki@x205a]$
```

22–15 ほぼ自習 アクセス制御とカプセル化

情報隠蔽・カプセル化を進めるためには適切なアクセス制御が必要

⇨ アクセス修飾子の働きを次にまとめておく。

クラス、インタフェースに対するアクセス制御：

クラス定義等に付ける アクセス修飾子	効果
private	（指定不可）
（なし）	同一パッケージからのみ利用可
protected	（指定不可）
public	他パッケージからも利用可

クラスのメンバー(フィールド, メソッド, コンストラクタ, 入れ子クラス, 等)に対するアクセス制御:

<table>
<tr><td rowspan="2">メンバーに付ける
アクセス修飾子</td><td colspan="2">効果</td></tr>
<tr><td>クラスがpublicの場合</td><td>クラスが
publicでない場合</td></tr>
<tr><td>private</td><td colspan="2">クラス内からのみアクセス可</td></tr>
<tr><td>(なし)</td><td colspan="2" rowspan="3">同一パッケージからのみアクセス可</td></tr>
<tr><td>protected</td><td>同一パッケージ、
及びサブクラス
からのみアクセス可</td></tr>
<tr><td>public</td><td>他パッケージからもアクセス可</td></tr>
</table>

インタフェースのメンバー(フィールド, メソッド, 等)に対するアクセス制御:

メンバーに付けるアクセス修飾子	効果	
	インタフェースがpublicの場合	インタフェースがpublicでない場合
private	(指定不可)	
(なし)	(public宣言されたものとして扱われる)	
protected	(指定不可)	
public	他パッケージからもアクセス可	同一パッケージからのみアクセス可

次に、
アクセス修飾子を利用して情報隠蔽・カプセル化を進めている例を、これまでに示したプログラムの中から幾つか抜き出し再確認

例22．24（アクセス制御; 内部作業用クラス）

例題19.5... TowerOfHanoiConfigクラスのメンバーとして
privateでstaticな入れ子クラス Disk を用意

例題19.6... NumberWith1000DecimalPlacesクラスのメンバーとして
privateでstaticな入れ子クラス Digit を用意

例題19.7... BinaryTreeOfStringIntクラスのメンバーとして
privateでstaticな入れ子クラス Node を用意

いずれの入れ子クラスも 、外側のクラスの中で使うことしか想定していないので、private宣言し外部からの利用は出来なくしている。

```
[motoki@x205a]$  cat -n TowerOfHanoiConfig.java
     1 /* Hanoiの塔の問題における、途中の円盤の配置状況を表す...
     2
     3 public class TowerOfHanoiConfig {
     4     //棒に挿す円盤を表すオブジェクトのクラス
     5     private static class Disk {
                 ......
    23     }
             ......
   115 }
---
```

例22．25（アクセス制御; 内部作業用フィールド, メソッド）

例題19.6(と例22.1)... StackOfAnyObjectsクラスのメンバーとして
privateなインスタンスフィールド DEFAULT_INITIAL_CAPACITY, DEFAULT_CAPACITY_INCREMENT, stack, indexOfTopEle を用意

例題22.7... HeapsortIntArrayクラスのメンバーとして
privateなインスタンスメソッド heapify() を用意

これらは、いずれもインスタンス外から自由に利用させる必要がないものなので、private宣言して情報隠蔽

```
[motoki@x205a]$  cat -n StackOfAnyObjects.java
     1 /* Objectインスタンスを格納するpushdownスタックオブ... *
     2
     3 import java.util.*;
     4
     5 public class StackOfAnyObjects {
     6     private static final int DEFAULT_INITIAL_CAPACITY =
     7     private static final int DEFAULT_CAPACITY_INCREMENT
     8
     9     private Object[] stack;
    10     private int indexOfTopEle;
           ......
___ 107 }
```

例22．26（アクセス制御; アクセッサを用いたフィールドの管理）

例題19.4... Rectangleクラスのメンバーとして

protected,finalなインスタンスフィールド id と

protectedなインスタンスフィールド width, height を用意

これらのフィールドのアクセス修飾子をpublicや「なし」にすると 、

外部から中身の閲覧だけでなく自由に書き換えができてしまい問題

⇨ アクセス修飾子をprotectedとしてこれらのフィールドへの

直接のアクセスを自クラスとサブクラス内に限定し、

代わりに必要に応じてゲッターメソッドやセッターメソッドを用意

例えばidについては 、

インスタンス生成以降値を変更することはない

$\Longrightarrow$ ゲッターメソッド(getId())だけを用意, 更にfinal宣言も

ここでpublic宣言されたアクセッサについても 、

将来の状況に応じてアクセス修飾子を変更して

情報隠蔽の度合いを調節可

```
[motoki@x205a]$  cat -n Rectangle.java
     1 /* 長方形を表すオブジェクトのクラス */
     2
     3 public class Rectangle {
     4     protected final int id;    //長方形インスタンスに付け.
     5     protected double width;    //長方形の幅
     6     protected double height;   //長方形の高さ
             ......
    28     //ゲッターメソッド
    29     public int getId() {
    30         return id;
    31     }
             ......
    58 }
```

例22．27（アクセス制御;インスタンス生成を抑制）

例題22.7 ... HeapsortIntArrayクラスのコンストラクタを
private宣言し外部からのインスタンス生成を出来なくしている。
(複数のインスタンスを生成してもメモリの無駄にしかならないため)

外部からのインスタンス生成を抑制する代わりに 、
クラス内部でインスタンス1個を生成し保持した上で、
そこへの参照値を外部に対して教えるpublic，staticなメソッド
getInstance() を用意

```
[motoki@x205a]$  cat -n HeapsortIntArray.java
     1 /**
     2  * int配列内の要素をheapsort手法で昇順に並べ替える機能
     3  * を備えた整列化モジュールを作り出すためのクラス
     4  */
     5 public class HeapsortIntArray extends SortModuleForIntA
     6     //クラス内部でインスタンスを1個だけ生成
     7     //  （コンストラクタはprivate宣言してあるので、  ）
```

```
 8      //  （生成されるインスタンスはこの1個だけになり、）
 9      //  （これが使い回されることになる。　　　　　　　）
10      private static final HeapsortIntArray INSTANCE
                                        = new HeapsortIntArray();
11
12      //コンストラクタ(外部からインスタンス生成不可)
13      private HeapsortIntArray() {
14          super();
15      }
16
17      /** コンストラクタの代わりに外部に整列化モジュールを...
18      public static HeapsortIntArray getInstance() {
19          return INSTANCE;
20      }
          ......
86  }
```

22–16 ほぼ自習 オブジェクト指向のまとめ、利

オブジェクト指向の特徴と利点を以下に列挙する。

オブジェクト指向の基本的な特徴：

- クラスを定義し、そのクラスのインスタンス（ソフトウエア部品）を必要なだけ生成して利用する。
  ⇨ （利点0）コードの簡素化 ··· 類似コードをあちこちに書かなくて済むので。

オブジェクト指向の3大要素：

- カプセル化 ··· 情報隠蔽を進めてオブジェクト（ソフトウェア部品）の独立性を高める。
  ⇨ （利点1.1）モジュール性 ··· 他のオブジェクトと切り離して、オブジェクト毎にソースコードを作成・保守することができる。（19.2節の記述）
  （利点1.2）情報隠蔽の恩恵 ··· 内部の実装方式がちゃんと隠蔽さ

れていて隠蔽しているはずの事柄に依存したコードが他のオブジェクト中に現れないことが保証されるなら、オブジェクト内部の実装方式を自由に変更することができる。(19.2節の記述)

アクセッサメソッドを用いて情報隠蔽を行なっている場合
情報の保持方法の変更はアクセッサメソッドの処理内容を変更するだけで済む。

(利点1.3) コードの再利用 ··· 一般的な処理を行うオブジェクトの場合、他からの独立性を高めることにより、コード再利用の可能性が高まる。(19.2節の記述)

(利点1.4) ソフトウェア全体の保守の容易さ ··· 1つのオブジェクトで異常が発生した場合でも、そのオブジェクトを代替オブジェクトに差し替えたり、場合によってはそのオブジェクトを全体から切り離して残りの部分を運用したり(fail soft)、ということを行い易い。(19.2節の記述)

- 継承 ··· 既存のクラスの内容を引き継いで新たな別のクラスを定義できる。

  ⇨ （利点2.1）効率的なプログラミング ··· 簡単に既存クラスを拡張できる。また、類似クラスができそうな場合は、それらの共通部分を親クラスとして構成することにより、類似コードをあちこちに書かなくて済む。

  （利点2.2）間違いの可能性の減少 ··· 各種機能を整理して配置し、類似コードが多数に場所に分散するのを極力避けることが出来るので、コードの修正忘れも少なくなる。

- 多態性 ··· 同じメソッドに対してオブジェクトごとに異なる振る舞い。

  ⇨ （利点3）コードの簡素化 ··· 同種のインスタンスを統合的に扱えるので。

注意： ちゃんと書けばこういう利点の恩恵に与れる、という話である。以上の利点を充分に引き出せてないプログラムはJavaで書いてあっても非オブジェクト指向と言える。

例22．28（手続き指向プログラムvs.オブジェクト指向プログラム）

同じ問題に対して

- 手続き的に構成されたプログラムと
- オブジェクト指向の考え方で構成されたプログラム

を対比することによって、オブジェクト指向の特徴と利点の認識を深めて下さい。

扱った問題： 2種類の書式

"Book,[本の書名],[著者名],[価格],[在庫数]"

と

"DVD,[DVDのタイトル],[価格],[在庫数]"

に従った文字列データを要素とする配列を引数として受け取り、中に書かれた在庫データを分析した上で

本の総冊数 = [調査結果]

DVDの総数 = [調査結果]

10冊以上在庫がある本のタイトル数 = [調査結果]

総金額 = [調査結果]

という風に出力するメソッドを作る。

手続き指向プログラム：

```
[motoki@x205a]$  cat StockTakingNonOOP.java
/**
 * 在庫状況を把握するためのクラス(手続き指向版)
 */
public class StockTakingNonOOP {
    /** 在庫状況の概要を出力する */
    public static void printOutline(String[] lines) {
        int countOfBooks = 0,                    //本の総冊数
            countOfDVDs = 0,                     //DVDの総数
            countOfTeemingBookTitles = 0,
                                  //10冊以上在庫がある本のタイトル数
            amount = 0;                          //総金額

        for (String line : lines) {
            if (line.startsWith("Book")) {
                String[] data = line.split(",");
                int count = Integer.parseInt(data[4]);
                countOfBooks += count;
```

```
            if (count >= 10) {
                ++countOfTeemingBookTitles;
            }
            int price = Integer.parseInt(data[3]);
            amount += price * count;
        } else {
            String[] data = line.split(",");
            int count = Integer.parseInt(data[3]);
            countOfDVDs += count;
            int price = Integer.parseInt(data[2]);
            amount += price * count;
        }
    }

    System.out.println("本の総冊数 = " + countOfBooks);
    System.out.println("DVDの総数  = " + countOfDVDs);
    System.out.println("10冊以上在庫がある本のタイトル数 =
                        + countOfTeemingBookTitles);
    System.out.println("総金額     = " + amount + "円");
```

```
    }

    //---------------------単体での動作テスト用-----------------
    public static void main(String[] args) {
        String[] stockData = {
            "Book,ゼロから学ぶ!最新Javaプログラミング,日経...,25
            "DVD,サウンド・オブ・ミュージック,1490,10",
            "Book,プログラミング言語Java第4版,K.Arnold他,441(
            "Book,Javaチュートリアル第4版,S.Zakhour他,5040,5"
        printOutline(stockData);
    }
}
[motoki@x205a]$  javac StockTakingNonOOP.java
[motoki@x205a]$  java StockTakingNonOOP
本の総冊数 = 27
DVDの総数  = 10
10冊以上在庫がある本のタイトル数 = 1
総金額      = 99320円
[motoki@x205a]$
```

オブジェクト指向プログラム：

```
[motoki@x205a]$  cat StockTakingOOP.java
/**
 * 在庫状況を把握するためのクラス(オブジェクト指向版)
 */
public class StockTakingOOP {
    /** 在庫状況の概要を出力する */
    public static void printOutline(String[] lines) {
        int countOfBooks = 0,                    //本の総冊数
            countOfDVDs = 0,                     //DVDの総数
            countOfTeemingBookTitles = 0,
                                  //10冊以上在庫がある本のタイトル数
            amount = 0;                          //総金額

        for (String line : lines) {
            Item item = null;
            if (line.startsWith("Book")) {
                item = new Book(line);
                countOfBooks += item.getCount();
```

多態変数

```
                if (((Book)item).isTeeming()) {
                    ++countOfTeemingBookTitles;
                }
            } else {
                item = new DVD(line);
                countOfDVDs += item.getCount();
            }
            amount += item.getAmount();
        }

        System.out.println("本の総冊数 = " + countOfBooks);
        System.out.println("DVDの総数  = " + countOfDVDs);
        System.out.println("10冊以上在庫がある本のタイトル数 =
                           + countOfTeemingBookTitles);
        System.out.println("総金額     = " + amount + "円");
    }

    //--------------------単体での動作テスト用-----------------
    public static void main(String[] args) {
```

```
        String[] stockData = {
            "Book,ゼロから学ぶ!最新Javaプログラミング,日経...,25
            "DVD,サウンド・オブ・ミュージック,1490,10",
            "Book,プログラミング言語Java第4版,K.Arnold他,4410
            "Book,Javaチュートリアル第4版,S.Zakhour他,5040,5"
        printOutline(stockData);
    }
}

//----------------------補助的なクラス定義-----------------------
/** 商品のクラス */
class Item {
    private int count = 0;
    private int price = 0;

    public int getCount() {
        return count;
    }
```

```
    public int getPrice() {
        return price;
    }

    protected void setCount(int count) {
        this.count = count;
    }

    protected void setPrice(int price) {
        this.price = price;
    }

    public int getAmount() {
        return price * count;
    }
}

/** 本のクラス */
class Book extends Item {
```

```
    private static final int THRESHOLD_FOR_TEEMING_BOOK = 10

    public Book(String bookInfo) {
        String[] data = bookInfo.split(",");
        setPrice(Integer.parseInt(data[3]));
        setCount(Integer.parseInt(data[4]));
    }

    public boolean isTeeming() {
        return (getCount() >= THRESHOLD_FOR_TEEMING_BOOK);
    }
}

/** DVDのクラス */
class DVD extends Item {
    private static final int THRESHOLD_FOR_TEEMING_BOOK = 10

    public DVD(String dvdInfo) {
        String[] data = dvdInfo.split(",");
```

```
            setPrice(Integer.parseInt(data[2]));
            setCount(Integer.parseInt(data[3]));
        }
}
[motoki@x205a]$  javac StockTakingOOP.java
[motoki@x205a]$  java StockTakingOOP
本の総冊数 = 27
DVDの総数  = 10
10冊以上在庫がある本のタイトル数 = 1
総金額      = 99320円
[motoki@x205a]$
```

補足： オブジェクト指向にするとコード量も増え、かえって複雑になった様に見えないこともない。しかし、

- 抽象化を進め、
- 類似部分を抽出し親クラスとしてまとめる、

等してオブジェクト指向化することによって、プログラムが幾つかの独立な部品に分けられ、商品の種類が増える等の時にも関連する部品だけに修正範囲を留めることができる様になっている。

オブジェクト指向設計の原則:

{日経ソフトウェア編「ゼロから...」第3部1章p.181}

- 単一責務の原則(SRP, Single Responsibility Principle) ···
  分割された個々のプログラムに複数の目的・機能を負わせるべきでない、という指針。(その方が、将来の部分的な変更もやり易い。)

- 開放閉鎖の原則(OCP, Open-Closed Principle) ···
  構築するクラス群は、
  機能拡張可能(open)で、
  拡張の際には既存コードには手を加えなくて済む(closed)、
  様なものが良い、という指針。

補足（良いプログラムを構築するための考え方と実践方法）:

{日経ソフトウェア編「Javaツール完全理解」第2部2章}

- ソフトウェアの価値の3条件
  - ◇ シンプル ··· プログラムの理解や修正が容易になる。
  - ◇ コミュニケーション可 ··· プログラムの書き手と読み手がソースコードを通じて十分にコミュニケーションできる。
  - ◇ 柔軟性 ··· 変更に対する柔軟性がある。（初期開発費用より修正費用の方が大きいので、これも重要。）
- プログラミングの原則
  - ◇ YAGNI (You Aren't Going to Need It.) ···
    今必要なことだけをやる。
  - ◇ DRY (Don't Repeat Yourself.) ···
    コードの重複を避け、必要があればできるだけ再利用する。
  - ◇ PIE (Program Intently and Expressively.)···
    意図が明確に伝わる様にコードを書く。
- 代表的なベストプラクティス ..... （次ページ）

補足(良いプログラムを構築するための考え方と実践方法,続き):

- ソフトウェアの価値の3条件 ..... (前ページ)
- プログラミングの原則 ..... (前ページ)
- 代表的なベストプラクティス
  - ◇ リファクタリング ··· 外部に対する振舞いを変えずに、ソースコードの内部構造を整理し簡素化すること。
  - ◇ テストファースト ··· 実装者の視点で実装コードを書く前に、利用者の視点でテストコードを書く。これによって余分な複雑さを排除できることを期待する。
  - ◇ ドメイン駆動設計 (Domain-Driven Design, DDD) ··· 問題領域(domain)をモデル化し、それを中心に据えてソフトウェアを設計する。(モデル自体もドメイン知識を使って反復的に深化させていく。)

22–17 ほぼ自習 ソフトウェアの部品化と再利用

プログラミング言語の進化 プログラミング言語に備わっているべき事柄は、...

- アルゴリズムを容易にコード化できるための表現能力
- ソフトウェアの寿命が伸びた
  ⟹ 保守も大事
  ⟹ 出来上がったプログラムの理解や修正の容易さも重要
- ソフトウェアには高い品質が必要
  ⟹ プログラムの中に余計な複雑さや単純な間違いが入り込みにくいということも大切
- ソフトウェアの生産性を上げたい
  ⟹ 実績のあるプログラムを再利用する仕組み

⇨ プログラミング言語がどの様に進化してきたのかを、
--- 次の様にまとめることができる。

	表現能力	保守性	品質保証	再利用性	導入された機構/考え方
機械語					
アセンブリ言語	△				
高級言語	○			△	• サブルーチン ⟹ 表現能力, 再利用性向上
構造化	○	△	△	△	• 3つの基本構造 ⟹ 保守性向上 • サブルーチンの独立性を高める ⟹ 再利用性多少向上
オブジェクト指向	○	○	○	○	• クラス(関連する変数とメソッドをまとめる仕掛け) ⟹ 保守性向上, 再利用性向上 • 例外機構 ⟹ 品質向上 • ガベージコレクション ⟹ 品質向上 • 型チェックの強化 ⟹ 品質向上 • 多態性 ⟹ 再利用性向上 • 継承 ⟹ 再利用性促進

時間 ↓

ソフトウェア再利用技術発展の流れ

オブジェクト指向より前の構造化言語では、

$\Longrightarrow$ 再利用できるソフトウェアと言えばサブルーチンだけ
コード変換，入出力処理，数値計算，. . . の汎用ライブラリ程度

オブジェクト指向の考えが導入されると、

(関連性の強いサブルーチンや大域変数を1つのクラスとしてまとめて粒度の大きいソフトウェア部品を作り出す仕組みがある)

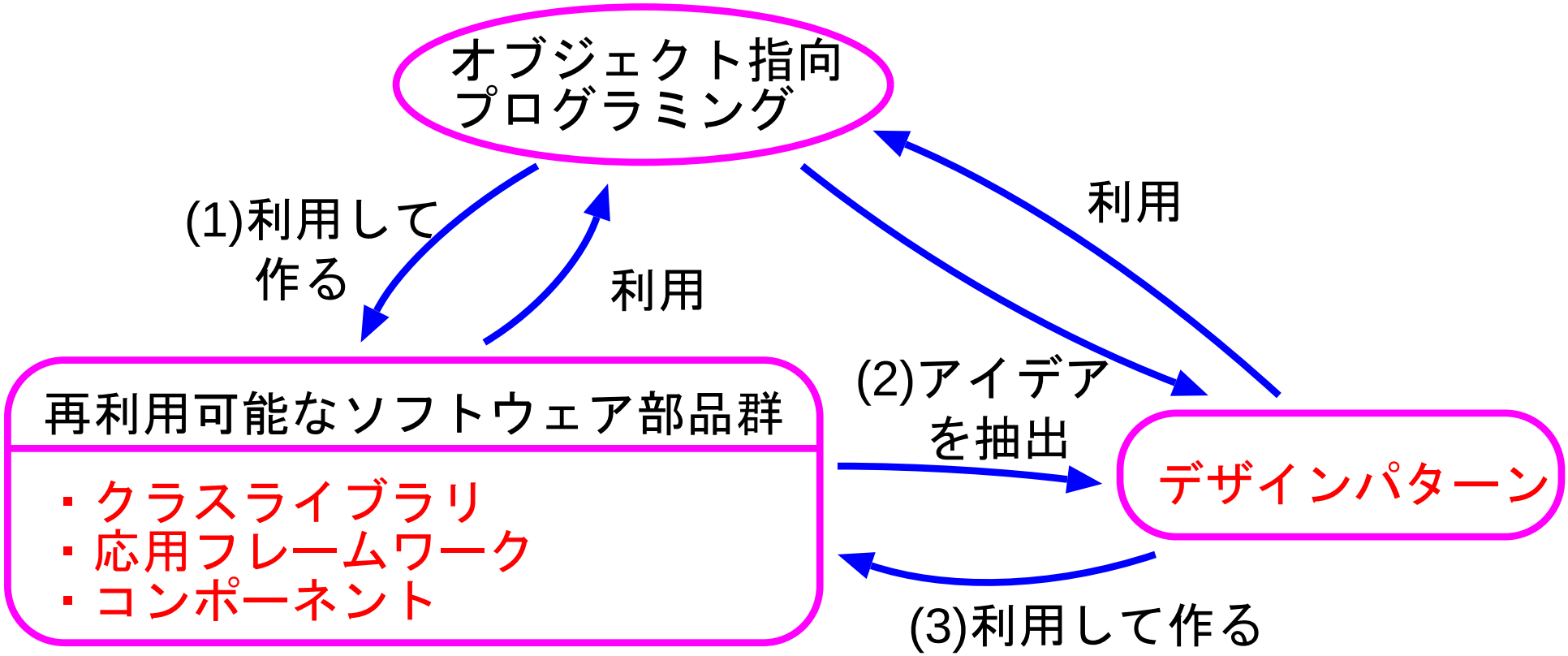
$\Longrightarrow$ ソフトウェア再利用の可能性は大きく広がる。

$\Longrightarrow$ ソフトウェア部品として既に存在しているソースコードや実行形式モジュールを使い回すのが当り前。

再利用可能なソフトウェア部品としては、現在、

- クラスライブラリ、
- (応用) フレームワーク、
- コンポーネント、
- デザインパターン··· ソフトウェア設計のアイデアを後で利用できるように文書化したもの

と呼ばれるものがある。



クラスライブラリ　　・・・汎用的な機能をもつクラスを多数蓄積したもの

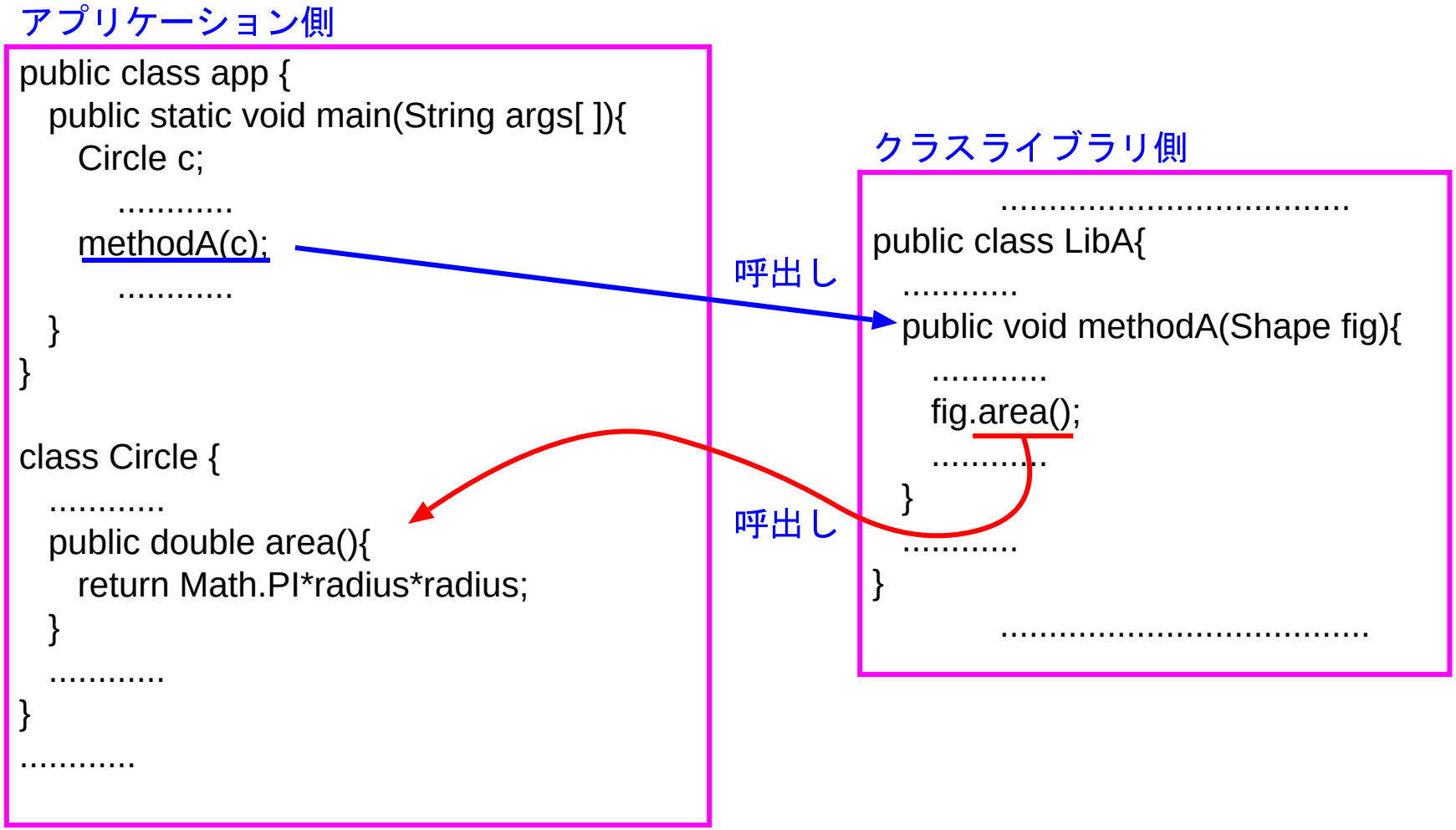
オブジェクト指向の前と比べて

- ソフトウェアが格段に豊富になった。
- ライブラリの利用の仕方も広がった。

クラスライブラリの場合は次の様な3つの利用の仕方が可能

- 用意されたクラスのインスタンスを作成して
  付属のインスタンスメソッド等を利用。
  （クラスの利用，従来のライブラリ関数呼び出しに相当。）

- ライブラリのコードからアプリケーション固有の処理を呼び出す。
  （多態性の利用。）



- ライブラリ内のクラスを拡張・補正して、新しいクラスを作成。（継承の利用。）

特に、現在のJava(J2SE7.0; JDK1.7, Java SE Development Kit 1.7)には、GUI, 入出力, ネットワーキング, ... 等のために、合わせて4000にものぼるクラスライブラリが整備されている。

言語仕様は最小限に抑えて
必要な機能はクラスライブラリとして提供される
$\Longrightarrow$ 言語仕様の互換性を保ちながら
クラスライブラリの拡張によってバージョンアップを行うことが可能。

$\Rightarrow$ この豊富なライブラリを使いこなすことが、
Java習熟への1つの道です。

(応用) フレームワーク

特定の業務分野について、色々な利用者に共通な部分は完成させておいて、個別の要求のある部分だけを追加するだけでそれぞれの利用者に合った応用プログラムを作成できる様にした、いわば「半完成品」を応用フレームワークあるいは単にフレームワークという。

> クラスライブラリも応用フレームワークも再利用可能なソフトウェア部品群という点では同じ。ただ、目的と再利用部品の使われ方が違う。

例22．29（応用フレームワーク, Javaアプレット）

Javaアプレットは応用フレームワークの代表例。

> 実際、Appletクラスのmainメソッドはアプレットの全体的な動作を規定するものとして予め定義されており、我々はmain の中から呼び出される init(), start(), paint(), ... 等の細部を書くだけで、動的なWebページを手軽に作ることができる。

コンポーネント　…(クラスライブラリ，応用フレームワークと全然違う)

平澤 (2004) によれば、

- クラスよりも粒度が大きく、
- (ソースコード形式でなく) バイナリ形式で提供される、

そして、

- ソフトウェア部品の定義情報も提供される、
- 機能的に独立性が高く内部の詳細を知らなくても利用できる、

というものを一般にコンポーネントと呼ぶ。(広く浸透していない。)

利用の仕方も特徴的で、

ソースコードを書くのではなく、
視覚的なツールを用いて直接関連する部品を配置・設定・接続する

ことによって、短時間で応用ソフトウェアを組み立てる。

具体例：

マイクロソフト社VisualBasic(1990年代前半~)で導入 $\longrightarrow$ActiveX

Java環境ではJavaBeansと呼ばれる仕組み

Beansと呼ばれるコンポーネントを

BDK 等の（ビルダ）ツールで接続してソフトウェアを組み立てる。

デザインパターン

..... オブジェクト指向に基づいて
再利用性や柔軟性の高いソフトウェアを開発しようとする際に、
様々な場面で適用される「お決まりの設計指針」

有意義で適用範囲の広いデザインパターンが見つかれば、
①全てのソフトウェア開発者がその恩恵を受けることができ、また
②それが開発者間の共通認識として定着すれば開発者間のコミュニケーションも容易になる。

この様な状況はアルゴリズムやデータ構造の場合と同じ。

具体的なデザインパターンとしては、
E.Gamma, R.Helm, R.Johnson, J.Vlissides という4人の技術者達(GoF the Gang of Four)が発表した次の23種類(GoFのデザインパターンと呼ばれている)が有名で、これらはJavaのクラスライブラリの作成にも大いに利用されている。

GoFによる分類	パターン名	再利用を妨げる要因のどれに効果が期待されるか?						
		クラス名を固定したインスタンスの生成	特定の処理内容への依存	プラットフォームに依存したapplicattion programインターフェースの使用	特定のアルゴリズムへの依存	クラス同士の密接な依存関係	継承によるデメリット	クラス数の急激な増加
生成に関するパターン	Abstract Factory	○		○		○		
	Builder				○			
	Factory Method	○						
	Prototype	○						
	Singleton							
構造に関するパターン	Adapter							
	Bridge			○		○	○	○
	Composite						○	
	Decorator						○	○
	Facade					○		
	Flyweight							
	Proxy							
振舞いに関するパターン	Chain of Responsibility		○			○	○	
	Command		○			○		
	Interpreter							
	Iterator				○			
	Mediator					○		
	Memento							
	Observer					○	○	
	State							
	Strategy				○		○	
	Template Method				○			
	Visitor				○			

GoFの著作においては、

これらのデザインパターンは次の様な項目に分けて説明されている。

- パターン名 ···
- 目的 ··· そのデザインパターンがどの様な設計課題に対処するか、何をもたらすか、原理と意図、等を簡潔に。
- （別名 ···　　）
- 動機 ··· 設計上の問題点、及び、そのパターン内のクラスやオブジェクトの構造がどの様にその問題を解決するか、のシナリオ。
- 適用場面 ··· このデザインパターンを適用できる状況。
- 構造 ··· クラス間の関係、要求のシーケンス、...を図で。
- 構成要素 ··· 使われるクラス、オブジェクトと、各々の役割。
- 協調関係 ··· 各構成要素がどの様に協調して役割を果たすか。
- 結果 ··· そのパターンが要求に対してどの様に効果を発揮するか。
- 実装 ··· 実装方法や注意点。
- サンプルコード ··· そのデザインパターンを使って実装した例。
- 利用例 ··· そのデザインパターンが実際のシステムで利用された例。
- 関連するパターン ··· 別のデザインパターンとの関係。

例22．30（Iteratorパターンの活用）

例えば、配列 arr[] の個々の要素に対してgetName()の実行依頼を出し、戻って来た文字列を全て表示するのに

```
for (int i=0; i<arr.length; i++)
  System.out.println(arr[i].getName());
```

これは、
　　オブジェクトの集合体を表すのに配列を用いるということに依存したコード

⇨ **Iteratorパターンの指針に従えば、上のコードは**

```
Iterator it = 集合体オブジェクト.iterator();
while (it.hasMoreElements()) {
  クラス名 obj = it.nextElement();
  System.out.println(obj.getName());
}
```

という風に、集合体の実装方法に依存しないコードに置き換わる。

補足： J2SE5.0以降では

Iteratorを用いずに拡張for文を用いて

```
for (クラス名 element :  arr)
    System.out.println(element.getName());
```

と書けるが、この拡張for文も内部ではIteratorの仕組みを利用

例22．31（Singletonパターンの活用）

インスタンス生成を1度だけに限定したいクラスもある。

⇨ **Singletonパターン**

これはこの講義ノートの中でも既に

例題22.7 の HeapsortIntArray.java,
BubblesortIntArray.java,
LListsortIntArray.java,

例題22.12 の TesterForSortModuleIntArray.java

を構築する際に利用している。

例22．32（Strategyパターンの活用）

処理の大枠は固定するが、
その中で使うアルゴリズム(strategy)は色々と切り替えて使いたい、
という場合もある。

⇨ **Strategyパターン**

例題22.12で考えた TesterForSortModuleIntArray クラス
... インスタンスには個別の整列化モジュールは持たせなかった

Strategyパターンに従って
個別の整列化モジュールとその名前を
内部のインスタンス変数にもたせる様に変形すると...

```
[motoki@x205a]$  cat TesterForFixedSortModuleIntArray.java
import java.util.Scanner;
import java.util.Random;

/**
```

```
 * 内部に保持するSortModuleForIntArrayモジュールの
 * 「int配列内の要素を昇順に並べ替える機能」が正しく動作するか
 * どうかをテストする機能を備えたモジュールを作り出すためのクラス
 */
public class TesterForFixedSortModuleIntArray {
    private static final int SIZE  =100;
    private static final int WIDTH = 10;

    private final SortModuleForIntArray sortModule;

    //コンストラクタ
    public TesterForFixedSortModuleIntArray(
                          SortModuleForIntArray sortModule) {
        this.sortModule = sortModule;
    }

    /** オブジェクトの説明を答える */
```

```
    @Override
    public String toString() {
        return "Tester for " + sortModule;
    }
```

以下の3行を追加

```
    /** SIZE個のランダムなデータから成る配列に対して
     * 内部で保持する整列化モジュールを実行してみる */
    public void runOnRandomData() {
```

以下、例題22.12 の TesterForSortModuleIntArray.java と同じ

```
[motoki@x205a]$  cat TestFixedSortModulesIntArrayMain.java
/**
 * ・内部に保持する[HeapsortIntArrayオブジェクト]の整列化動作を...
 *    する機能を備えたTesterForFixedSortModuleIntArrayオブジェ...,
 * ・内部に保持する[BubblesortIntArrayオブジェクト]の整列化動作...
 *    する機能を備えたTesterForFixedSortModuleIntArrayオブジェ...,
 * ・内部に保持する[LListsortIntArrayオブジェクト]の整列化動作...
 *    する機能を備えたTesterForFixedSortModuleIntArrayオブジェ...
```

```
 * を生成し、これらを用いて内部に保持されている3つの整列化モジュー..
 * int配列内の要素を正しく昇順に並べ替えるかどうかをテストするJava
 */
public class TestFixedSortModulesIntArrayMain {
    public static void main(String[] args) {
        //HeapsortIntArrayオブジェクトの動作テスト
        TesterForFixedSortModuleIntArray testerForHeapsort
            = new TesterForFixedSortModuleIntArray(
                                HeapsortIntArray.getInstance());
        testerForHeapsort.runOnRandomData();
        System.out.println("---");

        //BubblesortIntArrayオブジェクトの動作テスト
        TesterForFixedSortModuleIntArray testerForBubblesort
            = new TesterForFixedSortModuleIntArray(
                              BubblesortIntArray.getInstance());
        testerForBubblesort.runOnRandomData();
```

```
        System.out.println("---");

        //LListsortIntArrayオブジェクトの動作テスト
        TesterForFixedSortModuleIntArray testerForLListsort
            = new TesterForFixedSortModuleIntArray(
                                LListsortIntArray.getInstance());
        testerForLListsort.runOnRandomData();
    }
}
[motoki@x205a]$  javac TestFixedSortModulesIntArrayMain.java
[motoki@x205a]$  java TestFixedSortModulesIntArrayMain
```

実行の様子は省略
