オブジェクト指向言語
Javaプログラミング（2010年度）

桑原 寛明
情報理工学部 情報システム学科

Javaの基礎

Java言語

- クラスに基づくオブジェクト指向言語
- 制御構造、式、演算子はC,C++に類似
- 豊富なライブラリ

- Java仮想機械(Java VM)と呼ばれる実行環境
  - Javaのバイナリは OSから独立
  - 各OSごとにJava VMを用意すればJavaプログラムを変更せずに異なるOSで実行できる
    - Write once, Run anywhere.（建前的には）

ソースファイルとクラスファイル

- ソースファイル
  - Javaでプログラムが書かれているファイル
    - プログラムはクラスを単位に書く
  - 拡張子は".java"

- クラスファイル
  - Java VMの命令で書かれたバイナリファイル
    - 通常はソースファイルをコンパイルして生成
  - 拡張子は".class"

Javaプログラムの作成と実行

1. プログラムをソースファイルに書く
   – エディタやIDEを利用する
2. コンパイルしてクラスファイルを作る
   – % javac Sample.java
3. クラスファイルを指定してJava VMで実行する
   – % java Sample

プログラム例

Hello.java

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

```
% javac Hello.java
% java Hello
Hello World!
```


ソースファイルの構成

```
package jp.ac.ritsumei.cs;

import java.io.File;
import java.util.*;

public class Book {
  private File file;

  public String read() {
    ...
  }

  ...
}

class Folder {
  ...
}
```

- package文
- import文
- クラス定義
- フィールド
- メソッド
- クラス定義（1ファイルに複数書ける）

コメント

```
// 一行コメント

/*
  複数行コメント
*/

/**
  ドキュメンテーションコメント
  javadocコマンドによる文書生成に利用
*/
```

識別子

- クラス、メソッド、変数などの名前
- 英数字、アンダーバー、ドル記号、Unicode文字
  - 数字は先頭文字に使えない
  - ドル記号は慣習として内部的に利用される
  - 文字コードをUTF-8にすれば

```
public class サンプル {
  private int 氏名;
  public void 表示() {
  }
}
```

みたいなのも可能

命名規則

- 識別子に関する慣習的なルール
- パッケージ名
  - 小文字アルファベット（と数字）
  - ドメイン名の逆順
  - 例： jp.ac.ritsumei.cs.ClassName
- クラス名・インタフェース名
  - 1つ以上の単語
  - 各単語の先頭の1文字を大文字で
  - 例： Timer, FutureTask

命名規則

- フィールド名・メソッド名
  - 1つ以上の単語
  - 先頭の単語の一文字目は小文字で
  - メソッド名の先頭の単語は通常は動詞
  - 例： remove, ensureCapacity

予約語

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	if	goto	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

型

- 基本データ型(プリミティブ型)
  - boolean　true/false
  - char　16ビットUnicode文字
  - byte　8ビット符号付整数
  - short　16ビット符号付整数
  - int　32ビット符号付整数
  - long　64ビット符号付整数
  - float　32ビット単精度浮動小数点
  - double　64ビット倍精度浮動小数点
- 参照型
  - 基本データ型以外のすべて(クラス、配列など)

リテラル

- いわゆる定数
  - Nullリテラル(参照型)　null
  - ブール値リテラル(boolean)　true, false
  - 文字リテラル(char)　'Q', '%', '¥n'
  - 整数リテラル(int)　21, 035, 0x1a
  - 浮動小数点リテラル(float)　12.0, 2.17, 1.98E4
  - 文字列リテラル(String)　"Java"
- 定数変数
  - final付で宣言された変数　`final int MAX = 1024;`
  - 参照型の`final`には注意する

演算子

- 算術演算　+, -, *, /, %, ++, --
- 関係演算　<, <=, >, >=, ==, !=
- 論理演算　&, |, ^, !, &&, ||
- ビット演算　&, |, ^, <<, >>, >>>
- 代入演算　=, +=, -=, *=, /=, %=
- 条件演算　?:
- その他　new, instanceof, [], ()

文

- if-else, switch
- for, while, do-while
- break
- continue
- return
- try-catch-finally
- throw
- gotoは予約語だがgoto文はない

ラベル

外側のforにラベルを付けて

```
SEARCH:
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; j++) {
    if (matrix[i][j] == 0) {
      found = true;
      break SEARCH;
    }
  }
}
```

二重ループの中から一気に脱出

ローカル変数

- ローカル変数はメソッド内のどこでも宣言できる
  - スコープは宣言を含む最内のブロック
  - 通常は必要になったときに初めて宣言する
- for 文の初期化式で宣言できる

  ```
  for (int i = 0; i < 10; i++) { ... }
  ```
  - スコープは for 文全体

クラスとインスタンス

クラス

- 属性と操作をまとめて定義した抽象データ型
  - 属性 → フィールド
  - 操作 → メソッド
  - フィールドとメソッドをまとめてメンバともいう
- Javaプログラムの基本構成単位
- クラスを元に生成されたオブジェクト（インスタンス）が動作する

クラス定義の例

```
class Book {
  String title;
  int price;

  Book(String title, int price) {
    this.title = title;
    this.price = price;
  }

  public String getTitle() {
    return title;
  }

  public int getPrice() {
    return price;
  }

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

クラスの名前はBook

フィールド

コンストラクタ

メソッド

メソッド

メソッド

フィールド

- 属性を表す変数
  - インスタンス変数ともいう
- 型名＋フィールド名で宣言
- 同じクラスのどのコンストラクタ、メソッドからでも参照可能

メソッド

- オブジェクトに対する操作
- C言語の関数に類似
  - 引数、返り値
- 値を返す他にフィールドの値を変更できる

コンストラクタ

- クラス名と同じ名前のメソッド
- 返り値がないので返り値の型もない
- インスタンス生成時に実行される
  - 通常はインスタンスの初期化を行う
- this(適切な引数) で別のコンストラクタを呼び出せる

インスタンス生成

- new演算子を使って
  new クラス名(引数)
  のようにする
- 引数に応じて適切なコンストラクタが実行される
  - 適切なコンストラクタが見つからない場合はコンパイルエラー
- 生成されたインスタンスは変数に代入できる

メンバアクセス

- ドット演算子を使って
  book.title
  book.getTitle()
  のようにする
- bookはオブジェクトを指す変数
  - そのオブジェクトのtitleフィールドの値を読む
  - そのオブジェクトのgetTitleメソッドを実行する

this

- thisはインスタンス自身を指す予約語
- ドット演算子を使って
  this.title
  this.getPrice()
  のようにすることで自分自身のフィールドやメソッドにアクセスできる
  - thisによって同名のフィールドとローカル変数を区別できる

クラスの使用例

```
public class Sample {
  public static void main(String[] args) {
    Book my = new Book("The Java Tutorial", 55);
    System.out.println("My book title = " + my.getTitle());
    System.out.println("My book price = " + my.getPrice());

    Book your = new Book("The Java Programming Language", 55);
    your.setPrice(40);
    System.out.println("Your book title = " + your.getTitle());
    System.out.println("Your book price = " + your.getPrice());
  }
}
```

```
[実行結果]
My book title = The Java Tutorial
My book price = 55
Your book title = The Java Programming Language
Your book price = 40
```

オーバーロード（多重定義）

- 同一クラス内に同名のメソッド、コンストラクタを複数定義できる
- ただし、引数の型や数で区別できなければならない

```
Book(String title, int price) {
  this.title = title;
  this.price = price;
}

Book(String title) {
  this(title, 0);
}
```

Bookクラスのコンストラクタ（引数2個）

Bookクラスのコンストラクタ（引数1個）
this()は適切なコンストラクタの呼出し

static

- staticなフィールドやメソッドはクラスに属する
  - クラス変数、クラスメソッドと呼ぶ
  - 通常のフィールドやメソッドはオブジェクトに属する
- クラス変数とクラスメソッドは同じクラスのオブジェクトで共有される
- static修飾子を付けて宣言・定義する

クラス変数・クラスメソッドの例

```
class Book {
  static long nextID = 0;
  String title;
  int price;
  long id;

  Book(String title, int price) {
    this.title = title;
    this.price = price;
    this.id = nextID++;
  }
}
```

Bookクラスのすべての
インスタンスで共有

1つ目のインスタンスでは0、
2つ目は1、3つ目は2、…

合成と継承

合成

- 別のオブジェクトを参照あるいは包含することで機能を拡張する方法
- 自身に対するメソッド呼出しを包含するオブジェクトに丸投げ（委譲・転送）する
- あるいは、包含するオブジェクトのメソッド呼出しの結果を加工して拡張する

合成の例

```
class OnlineBook {
  Book book;
  String website;

  OnlineBook(String title, int price, String website) {
    book = new Book(title, price);
    this.website = website;
  }

  public String getWebsite() {
    return website;
  }

  public String getTitle() {
    return "Online: " + book.getTitle();
  }

  public int getPrice() {
    return book.getPrice();
  }
}
```

Bookクラスのオブジェクトを包含

Bookクラスのオブジェクトを生成

getPriceの呼出しはBookクラスのオブジェクトに委譲

継承

- 既存のクラスのインスタンス変数、メソッドを引き継いで新しいクラスを定義すること
- 新しい機能を追加したり、一部の機能を変更することができる
- 予約語extends
  - OnlineBookクラスがBookクラスを継承する場合
    class OnlineBook extends Book { ... }
  - extendsを省略すると暗黙にjava.lang.Objectを継承

継承の例

```
class Book {
  String title;
  int price;

  Book(String title, int price) {
    this.title = title;
    this.price = price;
  }

  public String getTitle() {
    return title;
  }

  public int getPrice() {
    return price;
  }

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

```
class OnlineBook extends Book {
  String website;

  OnlineBook(String title,
             int price,
             String website) {
    super(title, price);
    this.website = website;
  }

  public String getWebsite() {
    return website;
  }

  public String getTitle() {
    return "Online: " + title;
  }
}
```

継承の例

- OnlineBookクラスはBookクラスのフィールドtitle, priceやメソッドgetPrice, setPriceを持っている
  - コンストラクタは継承しない
  - Bookクラスの親クラス(java.lang.Object)のフィールドやメソッドも引き継ぐ
- getTitleメソッドは独自機能で上書き（オーバーライド）
- super()は親クラスのコンストラクタ呼出し

継承での注意

- 親クラスは1つのみ
  - 多重継承はできない
- finalなクラスは継承できない
- finalなメソッドはオーバーライドできない
- 親クラスのフィールドと同名のフィールドを宣言すると親クラスのフィールドが隠蔽される
  - super.fieldname でアクセス可能

抽象クラスと多態性

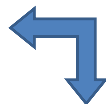
抽象クラス

- 抽象メソッドを持つクラス
- 抽象メソッド
  - 実装のないクラス
  - シグネチャ（メソッド名、引数の型と順番）はある
  - 実装は継承したクラスで行う
- 機能は同じだが実現方法が異なるクラスの親クラスとする

抽象クラスの例

```
class Article {
  private String title;
  private String journal;

  public Article(String title, String journal) {
    this.title = title;
    this.journal = journal;
  }

  public void showInfo() {
    System.out.println(title + "in" + journal);
  }
}
```

共通部分を親クラスにまとめたい



```
class TechReport {
  private String title;
  private String institution;

  public Article(String title, String institution) {
    this.title = title;
    this.institution = institution;
  }

  public void showInfo() {
    System.out.println(title + "by" + institution);
  }
}
```


抽象クラスの例

```
class Publication {
  protected String title;

  public Publication(String title) {
    this.title = title;
  }
}
```

タイトルを格納するフィールドtitleは親クラスが持つ

```
class Article extends Publication {
  private String journal;

  public Article(String title, String journal) {
    super(title);
    this.journal = journal;
  }

  public void showInfo() {
    System.out.println(title + "in" + journal);
  }
}

// TechReportクラスも同様
```

Publicationクラスに対してshowInfoメソッドが実行できるべき

showInfoメソッドの実装はクラスによって異なる

抽象メソッドにする

抽象クラスの例

```
abstract class Publication {
  protected String title;

  public Publication(String title) {
    this.title = title;
  }

  public abstract void showInfo();
}
```

showInfoは抽象メソッド
抽象メソッドを持つので abstract class

```
class Article extends Publication {
  private String journal;

  public Article(String title, String journal) {
    super(title);
    this.journal = journal;
  }

  public void showInfo() {
    System.out.println(title + "in" + journal);
  }
}

// TechReportクラスも同様
```


動的束縛・多態性

```
public void static main(String[] args) {
  Publication pub;
  pub = new Article("Java Compiler", "IEEE Software");
  pub.showInfo();

  pub = new TechReport("Java Programming Explained", "Ritsumeikan Univ.");
  pub.showInfo();
}

[実行結果]
Java Compiler in IEEE Software
Java Programming Explained by Ritsumeikan Univ.
```

- pub.showInfo()の結果はpubが指すインスタンスのクラスによって決まる
  - 実行時に呼び出されるメソッドが決定される ⇒ 動的束縛
  - 同じメソッド呼出しshowInfo()で異なる動作を実現する ⇒ 多態性(多相性)

インタフェース

- 抽象メソッドのみのクラス
  - インタフェースを実装したクラスが「何ができるか」のみを表す
  - 「どうやるか」は実装したクラスが決める

```
interface Printable {
  void print();
}

class Article extends Publication implements Printable {
  public void print() {
    ...
  }
}
```


パッケージとアクセス制御

パッケージ

- クラス名やインタフェース名の衝突を避けるために名前空間を作る仕組み
  - 例：java.lang.Objectはjava.langパッケージのObjectクラス
- パッケージ名まで含めたクラス名を完全修飾（限定）名と呼ぶ
  - 完全修飾名が異なれば異なるクラス
    - java.io.Fileとritsumei.Fileは違う

```
package ritsumei;

class File {
  ...
}
```


import

- 異なるパッケージのクラスは完全修飾名で指定
  - import文を利用すればパッケージ名を省略可能
- 同じパッケージのクラスは暗黙にimportされる
  - クラス名だけで利用可能

```
import ritsumei.File;

class Directory {
  File file1;          // ritsumei.File を利用
  ritsumei.File file2; // もちろん完全修飾名でもよい
  java.io.File file3;  // 完全修飾名で異なるパッケージのFileクラスを利用
}
```

import

- import ritsumei.* のように * も利用できる
  - プログラム中に単にFileとあった場合にritsumei.Fileを自動的に探してくれる
  - どこを探すかは別途指定する
    - クラスパス
- java.lang.* は暗黙に import される
  - java.langパッケージのクラスはimport不要

パッケージとディレクトリ構成

- Javaの基本は完全修飾名
  - 完全修飾名がディレクトリツリーにマッピングされる
- 例えば、jp.ac.ritsumei.cs.ClassName のクラスファイルは jp/ac/ritsumei/cs/ClassName.class に置かれる

アクセス制御

- パッケージやクラスの外部からクラスやメンバにアクセスできるか制御する
- クラス・インタフェースはpublic, packageの2段階

```
package ritsumei;

public class File {
  ...
}
```

ritsumei.Fileクラスはritsumeiパッケージ以外のパッケージで利用可能

```
package ritsumei;

class File {
  ...
}
```

ritsumei.Fileクラスはritsumeiパッケージでのみ利用可能

アクセス制御

- フィールド・メソッドはpublic, protected, package, privateの4段階

```
package ritsumei;

public class File {
  public void pub() {...}      // どのクラスからでもアクセス可能
  protected void pro() {...}   // ritsumeiパッケージとサブクラスから
                               // アクセス可能
  void pac() {...}             // ritsumeiパッケージ内からのみアクセス可能
  private pri() {...}          // ritsumei.File内からのみアクセス可能
}
```

アクセス制御の方針

- 制限はできるだけ厳しくする
- publicなインスタンスフィールドは利用しない
- サブクラスからアクセス可能なメソッドのみオーバーライド可能

例外処理

例外

- 正常系と異常系を分離するための仕組み
  - 正常系から送出された例外を捕捉することで異常系に移行する
- 関数の返り値と条件分岐を利用するよりも明確に異常系を分離できるのが利点

例外処理

- 例外を送出する可能性のあるコードを try ブロックで囲む
- 捕捉したい例外とその後の処理を catch ブロックに書く
  - catch された例外は消滅する
- 対応する catch ブロックのない例外は、その例外を送出したメソッドの呼出し元に送られる
- 例外が発生しなかった場合、catch で処理された場合は try-catch ブロックに後続するコードが実行される

try-catch-finally

```
try {

  // ここに例外を送出する可能性のあるコードを記述

} catch (例外クラス名 捕捉した例外オブジェクトを代入する変数) {

  // 例外処理
  // 例外クラス名の例外が送出されたときに実行される

} catch () { // 異なる例外に対応するcatchが書ける

} finally {

  // tryブロックあるいはcatchブロックを抜ける際に必ず実行される

}
```


例外処理の例

```
class ExceptionExample {
  public static void main(String[] args) {
    try {
      Reader r = new FileReader("NotExistedFile");
      System.out.println(r.read());

    } catch (FileNotFoundException e) {
      System.err.println("File Not Found: " + e);

    } catch (IOException e) {
      System.err.println("I/O Exception occurred: " + e);
    }

    System.our.println("Finish.");
  }
}
```

エラー発生
指定されたファイルがない
FileNotFoundExceptionを送出

ここで捕捉して処理

Error系例外

- 回復不可能なエラーの伝達に利用される
- 発生したら通常はプログラムの実行を停止する
- java.lang.Errorクラスの直接・間接のサブクラス
- 例
  - OutOfMemoryError
  - StackOverflowError
- 慣習的にJavaVM内部でのみ利用される

Exception系例外

- 回復可能なエラーの伝達に利用される
- 捕捉して適切な処理を行い実行を続行する
- java.lang.Exceptionクラスのサブクラス
- 例：
  - IOException
  - InterruptedException
  - NullPointerException

チェックされない例外

- RuntimeExcetion, Errorのサブクラス
- 対応するtry-catchは書かなくてもよい
- 通常はプログラミングエラー（主に事前条件違反）を表す
- 例：
  - ArrayIndexOutOfBoundsException
    - 配列の添字が不正
  - IllegalArgumentException
    - メソッド呼出しの引数が不正
  - NullPointerException
    - 本来nullではいけない参照がnullになっている

チェックされる例外

- チェックされない例外以外の例外
- コンパイル時に対応する例外処理が存在するか検査される
  - try-catch あるいは throws が必要
- 回復可能な状況を表すのに利用される
- 例：
  - IOException
    - 入出力時にエラーが発生
  - InterruptedException
    - スレッドに割り込みが発生

throws

- 呼び出したメソッドがチェックされる例外を送出する可能性があるか知るための仕組み
  - メソッド定義については該当する例外に対する try-catch が不要であることを示す
  - メソッド呼出し側については送出されてくる例外に対する処理が必要であることを示す

throwsの例

```
class ExceptionExample {
  public static void main(String[] args) {
    try {
      Reader r = openFileReader("NotExistedFile");
      System.out.println(r.read());

    } catch (FileNotFoundException e) {
      System.err.println("File Not Found: " + e);

    } catch (IOException e) {
      System.err.println("I/O Exception occurred: " + e);
    }

    System.our.println("Finish.");
  }

  private static Reader openFileReader(String filename)
    throws FileNotFoundException {
    return new FileReader(filename);
  }
}
```

FileNotFoundExceptionが送出されたらここで捕捉

FileNotFoundExceptionを送出するかもしれないことを示す

例外の送出

- throw 文を用いて例外クラスのインスタンスを送出することができる

独自の例外クラス

- java.lang.Exceptionクラスを継承して独自の例外クラスを作ることができる

```
class Manual {
  private String title;

  public void setTitle(String title) throws NoTitleException {
    if (title.length() < 1) {
      throw new NoTitleException();
    }

    this.title = title;
  }
}

class NoTitleException extends Exception {
}
```

インスタンス化して送出

独自例外クラスの定義

配列とコレクション

配列

- 同じ型の要素を並べたもの
- 配列も一種のオブジェクトなので new する
  - new するときに長さを指定する
- 例：長さ10のFileの配列

```
File[] files = new File[10];
```

- 配列の長さとして定数だけでなく式が書ける

```
int num = 10;
File[] files = new File[num];
```

- 配列の長さを取得するフィールド length

```
for (int i = 0; i < files.length; i++) {
```

コレクションフレームワーク

- 複数のオブジェクトをまとめて扱うためのライブラリ
  - インタフェースとクラス群
- リスト、集合、マップなどが提供されている
- java.util パッケージ

コレクションインタフェース

Collection ：コレクションの基礎インタフェース
List ：要素が特定の順序で並ぶ
Set ：重複した要素を含まなず順不同
SortedSet ：ソートされたSet
Map ：キーから高々1つの値へのマップ
SortedMap: キーがソートされたMap

Iterator ：要素1つずつへのアクセスを提供

実装クラス

ArrayList ：配列のように扱えるList
LinkedList ：双方向リンクによるList
HashSet ：ハッシュテーブルによるSet
TreeSet ：バランスした二分木によるSortedSet
HashMap ：ハッシュテーブルによるMap
TreeMap ：バランスした二分木によるSortedMap

ArrayListの例

```
List list = new ArrayList();

list.add("Java");
list.add("Programming");
list.add("Language");

System.out.println("size = " + list.size());

for (int i = 0; i < list.size(); i++) {
  String str = (String)list.get(i);
  System.out.println("i : " + str);
}
```

ArrayListのインスタンスを生成

末尾に追加

格納された要素に応じたキャストが必要

先頭の要素から順に表示

反復子(Iterator)

- コレクションに格納された要素に順にアクセスするための仕組み
- コレクションの種類によらず統一的

```
List list = new ArrayList();
list.add("Java");
list.add("Programming");
list.add("Language");

for (Iterator it = list.iterator(); it.hasNext(); ) {
  String str = (String)it.next();
  System.out.println(str);
}
```

反復子を取得

次の要素があればtrue

次の要素を取得

総称(generics)

- Java5以降では総称によってクラスやメソッドの型をパラメータとして定義できる
- 総称によってコレクションの要素となるクラスを指定できる
- 例えば、要素がStringのリスト

```
List<String> list = new ArrayList<String>();
```


総称(generics)

- 異なるクラスのインスタンスを追加しようとするとコンパイルエラー

```
List<String> list = new ArrayList<String>();
list.add(new Integer(10));   // error
```

- 取り出すときのキャストが不要

```
String str = list.get(0);
```


反復子と for-each

```
List<String> list = new ArrayList<String>();
list.add("Java");
list.add("Programming");
list.add("Language");

for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
  String str = it.next();
  System.out.println(str);
}

for (String str : list) {
  System.out.println(str);
}
```

キャストが不要

このように書いても同じ意味
(配列でも同じように書ける)

boxing と unboxing

```
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(100));
list.add(new Integer(200));
list.add(300);

for (Integer i : list) {
  System.out.println(i);
}

for (int i : list) {
  System.out.println(i);
}
```

プリミティブ型は対応する参照型に自動的にラップされる

取り出すときに自動的にプリミティブ型に変換される

入出力処理

入出力処理

- 入出力はストリーム（順序付データ列）で扱う
- java.io パッケージ
- 文字指向ストリーム
  - データを文字列として扱う
  - java.io.Reader, java.io.Writer
- バイト指向ストリーム
  - データをバイト列として扱う
  - java.io.InputStream, java.io.OutputStream

入出力処理の基本

1. ストリームを new して開く
2. 入力ならば read、出力ならば write メソッドを呼び出す
3. 最後に close メソッドを実行してストリームを閉じる

文字指向ストリームの基本抽象クラス

- 入力：Reader
  - read()
    - 1文字読込み
  - read(char[] buf, int offset, int count)
    - count文字読み込んでbuf[offset]からbuf[offset+count-1]に格納
- 出力：Writer
  - write(int ch)
    - 1文字書込み
  - write(char[] buf, int offset, int count)
    - buf[offset]からcount文字書込み

文字指向ストリームクラスの例

- 入力
  - FileReader
    - ファイルから入力
  - StringReader
    - 文字列（Stringオブジェクト）から入力
- 出力
  - FileWriter
    - ファイルに出力
  - StringWriter
    - 文字列に出力
  - PrintWriter

バイト指向ストリームの基本抽象クラス

- 入力：InputStream
  - read()
    - 1バイト読込み
  - read(byte[] buf, int offset, int count)
    - countバイト読み込んでbuf[offset]からbuf[offset+count-1]に格納
- 出力：OutputStream
  - write(int b)
    - 1バイト書込み
  - write(byte[] buf, int offset, int count)
    - buf[offset]からcountバイト書込み

バイト指向ストリームクラスの例

- 入力
  - FileInputStream
    - ファイルから入力
  - DataInputStream
    - バイト列からJavaの基本型データを入力
- 出力
  - FileOutputStream
    - ファイルに出力
  - DataOutputStream
    - Javaの基本型データを出力
  - PrintStream

バイト指向から文字指向への変換

- InputStreamReader
  - InputStream を Reader に変換
- OutputStreamWriter
  - OutputStream を Writer に変換
- バイト列を解釈する文字コードが指定できる

バッファリング

- データをバッファリングして効率よく入出力する
- BufferedReader, BufferedWriter
- BufferedInputStream, BufferedOutputStream
- 例: new BufferedReader(new FileReader(...))

標準入力・標準出力・標準エラー出力

- 標準入力
  - 通常はキーボード
  - System.in : InputStream
    - in は java.lang.System クラスの static フィールド
- 標準出力、標準エラー出力
  - 通常はディスプレイ
  - System.out : PrintStream
  - System.err : PrintStream

ファイル

- パスを表す java.io.File クラスを利用してファイル・ディレクトリの操作が可能
- new File(path), new File(dir, file)
- exists(), isFile(), isDirectory(), length(), delete(), etc.

例

```
PrintWriter pw = null;
try {
  pw = new PrintWriter(new BufferedReader(new FileWriter(filename)));
  pw.println("Java");
  pw.println("Programming");
  pw.println("Language");

} catch (IOException e) {
  System.err.println(e);

} finally {
  if (pw != null) {
    try {
      pw.close();
    } catch (IOException e) {
    }
  }
}
```

ファイルに出力

バッファを挿入

例外の発生に関わらずcloseするためにfinallyを利用

スレッド

スレッド

- 1本の糸のように実行される処理の流れ
- シングルスレッド・プログラム
  - 1つのスレッドのみを実行するプログラム
- マルチスレッド・プログラム
  - 複数のスレッドを実行するプログラム
- 各スレッドが異なる処理を行うことで、複数の処理を（見かけ上）同時に行うことができる

Javaのスレッド

- プログラムの起動時に1つのスレッド（メインスレッド）が生成されて実行される
- スレッドとしての基本的な機能を備えている java.lang.Thread クラス
  - スレッドは Thread （のサブクラス）のインスタンス
- 実行中のスレッドはThreadクラスのクラスメソッドcurrentThreadで取り出せる

スレッドの作り方

- java.lang.Thread を継承して run メソッドをオーバーライドする
- java.lang.Runnable インタフェースを実装したクラスを Thread に関連付ける
  – Thread クラスが継承できない場合など
- いずれの場合も Thread クラスの start() メソッドを呼び出すと実行開始し、run() メソッドを抜ければ終了

Threadを拡張する例

```
class MyThread extends Thread {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.our.println("my thread: " + i);
    }
  }

  public static void main(String[] args) {
    Thread th = new MyThread();
    th.start();

    for (int i = 0; i < 10; i++) {
      System.out.println("main: " + i);
    }
  }
}
```

独自スレッドを定義

独自スレッドをインスタンス化して開始

メインスレッドでも表示してみる

Runnableインタフェースを実装する例

```
class MyThread implements Runnable {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.our.println("my thread: " + i);
    }
  }

  public static void main(String[] args) {
    Thread th = new Thread(new MyThread());
    th.start();

    for (int i = 0; i < 10; i++) {
      System.out.println("main: " + i);
    }
  }
}
```

Runnableを実装

Runnableを実装したクラスを関連付けてスレッドをインスタンス化して開始

スケジューリング

- スレッドのスケジューリングは Java VM の実装に依存する
- スケジューラに対してヒントを提供することはできる
  – setPriority()
    - スレッドの優先度の変更
  – yield()
    - 現時点でカレントスレッドは実行しなくてもよい
  – sleep()
    - スレッドを指定されたミリ秒の間スリープ

スレッドのキャンセル

- キャンセルしたい側
  - キャンセル対象のスレッドに interrupt() メソッドにより割り込む
- キャンセルされる側
  - 割り込まれていないか interrupted() メソッドによって定期的に確認しながら処理を行う

```
Thread thread = new MyThread();
thread.start();

thread.interrupt();
```

```
class MyThread extends Thread {
  public void run() {
    while (!interrupted()) {
      // 少しだけ処理
    }
  }
}
```


スレッドの完了待ち

```
class MyThread extends Thread {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.our.println("my thread: " + i);
    }
  }

  public static void main(String[] args) {
    Thread[] ths = new Thread[10];
    for (int i = 0; i < 10; i++) {
      ths[i] = new MyThread();
      ths[i].start();
    }

    for (int i = 0; i < 10; i++) {
      try {
        ths[i].join();
      } catch (InterruptedException e) {}
    }
  }
}
```

スレッドを10個生成して開始

10個のスレッドの完了を待つ

同期

- 複数のスレッドが同一のデータを利用する場合は、互いの処理が干渉しないようにする必要がある

```
class Account {                 // 口座
  private long balance;         // 残高

  public void deposite(long amount) {
    balance += amount;
  }
}
```

2つのスレッドが同一のAccountオブジェクトを共有し、近いタイミングでdeposite()メソッドの呼出したらどうなるか？

ヒント: += は加算と代入の2動作

1. スレッドAが加算
2. スレッドBが加算
3. スレッドAが代入
4. スレッドBが代入

⇒ 結果は?

synchronized

- 同期処理を実現するための仕組み
- メソッドを synchronized とする
  - 同時に1つのスレッドのみが実行できる
    - synchronized でないメソッドは同時実行可能
  - メソッドが実行されるインスタンスをロックする
    - インスタンスが異なれば同時実行可能

```
class Account {                 // 口座
  private long balance;         // 残高

  public synchronized void deposite(long amount) {
    balance += amount;
  }
}
```

synchronized ブロック

- メソッド内の一部を同期対象とする

```
synchronized (式) {
    // 同期処理
}
```

- 式には参照型のインスタンスを指定する

スレッド間通信

- wait() メソッドで待機
- notify(), notifyAll() メソッドで再開
  - notify は待機中のスレッドいずれか1つを再開
  - notifyAll は待機中のスレッドすべてを再開
  - 通常は notifyAll を使う

スレッド間通信の例

```
class Queue {
  private Cell head, tail;

  public synchronized void enqueue(Object o) {
    Cell p = new Cell(o);
    if (tail == null) { head = p; }
    else { tail.next = p; }
    p.next = null;
    tail = p;
    notifyAll();
  }

  public synchronized Object dequeue() throws InterruptedException {
    while (head == null) { wait(); }
    Cell p = head;
    head = head.next;
    if (head == null) { tail = null; }
    return p.item;
  }
}
```

キューに何かが入ったことを通知

キューが空なので待機

GUIプログラミング

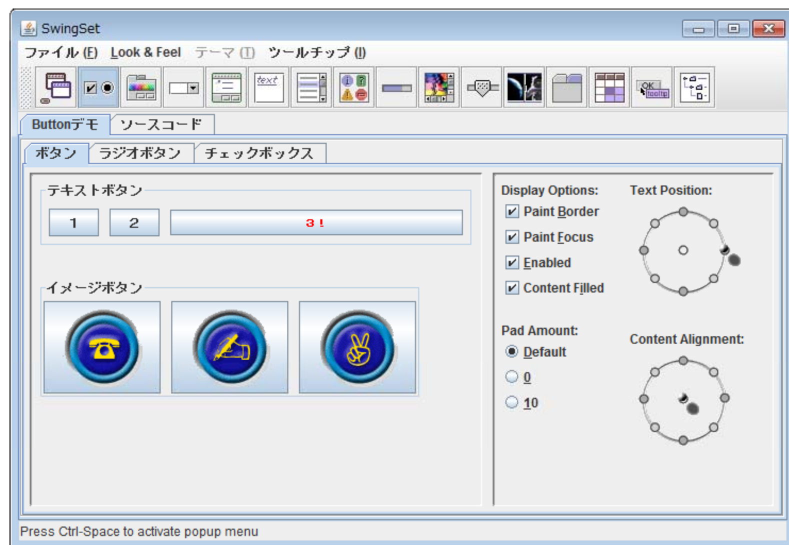
GUI

- グラフィクスを利用してユーザとプログラムが対話する仕組み
- 直感的で親しみやすい Look & Feel

Swing

- Javaが標準で提供するGUIプログラミングのためのフレームワーク
- ウィンドウ、ボタンやテキスト領域などの各種ウィジェット、マウスやキーボードなどを扱うイベントの仕組みを提供

SwingSet2



コンテナとコンポーネント

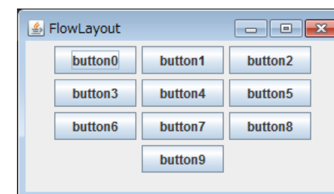
- トップレベルコンテナ
  - コンポーネントを乗せる一番基礎の土台
  - JFrame, JDialog, JWindow など
- 中間コンテナ
  - コンポーネントをまとめたりレイアウトする中間土台
  - JPanel, JScrollPane, JSplitPane, JTabbedPane など
- アトミックコンポーネント
  - 実際に表示されるボタンなどの部品
  - JLabel, JButton, JTextField など

レイアウト

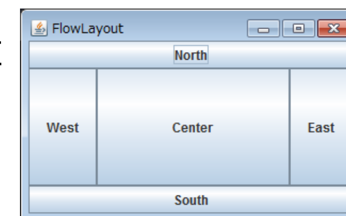
- GUIコンポーネントの配置は座標を具体的に指定するのではなくレイアウトマネージャに任せる
- FlowLayout, BorderLayout, CardLayout, GridLayout, GridBagLayout, BoxLayout など
- コンポーネントの追加やウィンドウのリサイズに対してレイアウトマネージャがコンポーネントを適切に配置する

レイアウトの例

- FlowLayout
  - 左から右、上から下に並べる



- BorderLayout
  - 上下左右、中央の5箇所に配置



イベント

- ボタンのクリックやマウスの移動などユーザの操作をGUIコンポーネントに通知する仕組み
- 操作が行われるとイベントオブジェクトが生成されて、対象コンポーネントに送られる
- コンポーネント側では各種イベントに対して処理を定義しておく

代理人モデル

- イベントの処理をイベントの発生源ではなく、イベント処理のためのオブジェクトに任せる
- イベント発生源となる各種コンポーネントに「イベントリスナ」を登録する
  - イベントが発生すると登録されたイベントリスナが実行される

イベントの例

ハンドラにはオブジェクトが渡される

イベントソース	イベントオブジェクト	イベントリスナ	イベントハンドラ
JButton JCheckBox JComboBox JMenu JTextField	ActionEvent	ActionListener	actionPerformed()
JScrollBar	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
Componentのサブクラス	FocusEvent	FocusListener	focusGained() focusLost()
	KeyEvent	KeyListener	keyPressed() keyReleased()
	MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
		MouseMotionListener	mouseDragged() mouseMoved()

イベント処理の例

```
JButton button = new JButton("Push");
button.addActionListener(new ButtonListener(button));
```

ボタンにイベントリスナを登録

```
class ButtonListener implements ActionListener {
  private JButton button;
  private boolean flag = true;

  ButtonListener(JButton button) {
    this.button = button;
  }

  public void actionPerformed(ActionEvent e) {
    if (flag) {
      button.setText("Push Push");
    } else {
      button.setText("Push");
    }
  }
}
```

イベントリスナの正体

発生したイベント

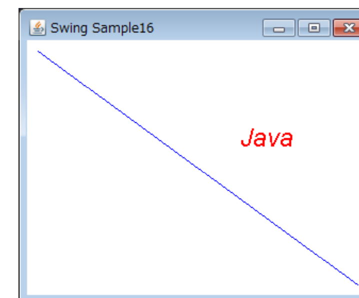
ボタンがクリックされると呼び出されるメソッド

グラフィクス

- グラフィックコンテキスト
  - 描画操作に必要な状態情報
    - 描画対象のコンポーネント
    - 座標
    - 色
    - フォント
  - java.awt.Graphicsクラス
- 試すには JPanel クラスを継承して paintComponent メソッドをオーバーライドするのが簡便

描画の例

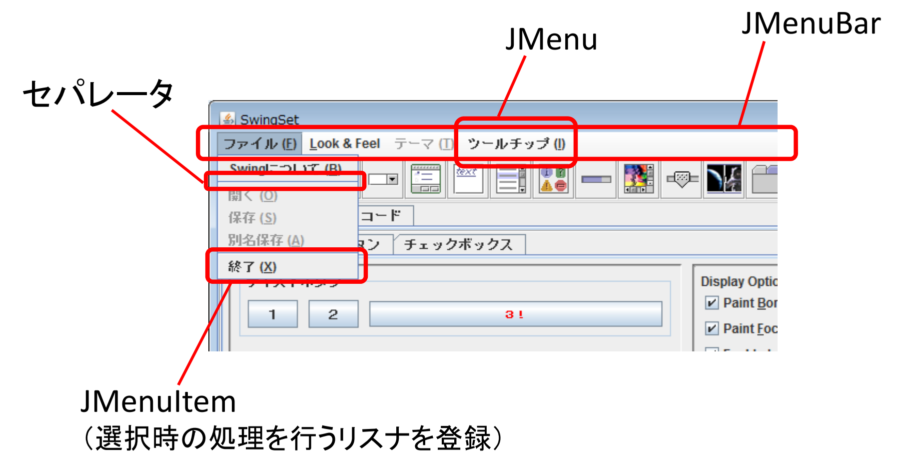
```
class CanvasSample extends JPanel {
  private Font font;

  CanvasSample() {
    setBackground(Color.white);
    setPrefferedSize(new Dimension(320, 240));
    font = new Font("SanSerif", Font.ITALIC, 24);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);

    g.setColor(Color.red);
    g.setFont(font);
    g.drawString("Java", 200, 100);

    g.setColor(Color.blue);
    g.drawLine(10, 10, 310, 230);
  }
}
```



再描画

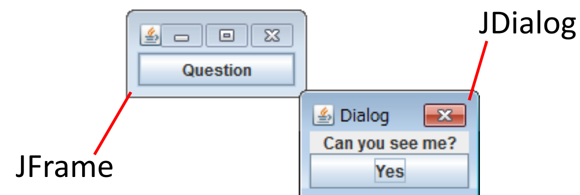
- paintComponent は必要に応じて呼び出される
  - setVisible(true)
  - ウィンドウサイズの変更
  - 重なっていたウィンドウの移動
- 明示的に呼び出すには repaint メソッド
  - 例えばマウスのクリックに対応するイベントハンドラ内で repaint する

メニュー



ダイアログ

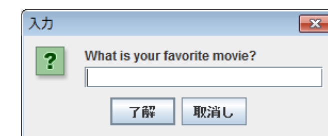
- JDialogクラスのインスタンス
  - 親ウィンドウ、モーダルかどうかを指定して生成
- モーダルダイアログ
  - ダイアログの表示中は元のウィンドウは操作不可
- 非モーダルダイアログ
  - ダイアログの表示中でも元のウィンドウが操作可



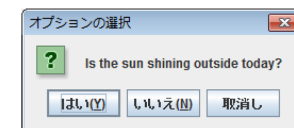
JOptionPane

- 標準的なダイアログを表示するメソッドを提供
  - showMessageDialog
    - メッセージを表示

  - showInputDialog
    - 簡単なテキスト入力

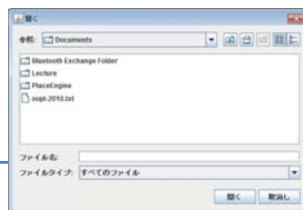


  - showConfirmDialog
    - Yes, No, or Cancel



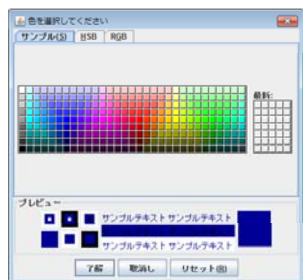
JFileChooser, JColorChooser

- JFileChooser
  - ファイル選択を行う



```
JFileChooser fc = new JFileChooser();
int result = fc.showOpenDialog(親ウィンドウ);
if (result == JFileChooser.APPROVE_OPTION) {
  ...
}
```

- JColorChooser
  - 色選択を行う



```
Color c = JColorChooser.showDialog(
              親ウィンドウ, タイトル, 初期色);
```


付録

練習問題

1. 乗り物を表すクラス Vehicle を作れ。スピード、進行方向(方角)、所有者を格納するフィールドを持たせること。

2. Vehicle クラスに引数なしのコンストラクタと、所有者の名前を引数に取るコンストラクタを追加せよ。他のフィールドは適当な値で初期化すること。

練習問題

3. Vehicle クラスに各フィールドの値を返すメソッドと、各フィールドの値を設定するメソッドを追加せよ。

4. Vehicle クラスに main メソッドを追加せよ。Vehicle クラスのインスタンスを生成してフィールドの値を適当に設定し、その値を表示するようにせよ。

練習問題

5. Vehicle クラスに回転する角度を引数に取る trun メソッドと、定数 Vehicle.TURN_LEFT, Vehicle.TURN_RIGHT を引数に取る turn メソッドを追加せよ。

6. Vehicle クラスにスピードを 0 にする stop メソッドを追加せよ。

練習問題

7. Vehicle クラスに次に生成される乗り物の ID を格納する static フィールドと、個々の乗り物の ID を格納する非 static フィールドを追加せよ。コンストラクタ内で非 static フィールドに ID を代入せよ。

8. Vehicle クラスに今までに使われた ID の最大値を返す static メソッドを追加せよ。

練習問題

9. Vehicle クラスを拡張して、座席数と座っている人数を扱える PassengerVehicle クラスを定義せよ。座席数を引数に取るコンストラクタを用意すること。合成による方法と、継承による方法の両方で作れ。

10. boolean を返す抽象メソッド isEmpty を持つ抽象クラス EnergySource を定義せよ。

練習問題

11. EnergySource クラスを継承して isEmpty メソッドを実装した GasTank クラスと Battery クラスを作れ。

12. Exception クラスを継承して EmptyEnergyException クラスを定義せよ。

練習問題

13.Vehicle クラスに EnergySource を引数に取るコンストラクタを追加せよ。

14.Vehicle クラスにコンストラクタに渡された EnergySource が empty でないことを保証できる start メソッドを追加せよ（isEmpty が true なら EmptyEnergyException をスローせよ）。

15.Vehicle クラスのインスタンスを生成し、start メソッドを呼び出すコードを書け。

練習問題

16.Vehicle クラスのフィールドのアクセス指定をすべて private にせよ。

17.Vehicle クラスの各メソッドに適切なアクセス修飾子を指定せよ。