

Javaプログラミング基礎

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Version 0.9.007

本ドキュメントについて



- この作品は、クリエイティブ・コモンズの表示-改変禁止 2.1 日本ライセンスの下でライセンスされています。この使用許諾条件を見るには、<http://creativecommons.org/licenses/by-nd/2.1/jp/> をチェックするか、クリエイティブ・コモンズに郵便にてお問い合わせください。住所は: 559 Nathan Abbott Way, Stanford, California 94305, USA です。
- 本ドキュメントの最新版は、<http://www.knowledge-ex.jp/opendoc/javaprogramming.html> より入手することができます。

あなたは以下の条件に従う場合に限り、自由に



本作品を複製、頒布、展示、実演することができます。

あなたの従うべき条件は以下の通りです。



表示. あなたは原作者のクレジットを表示しなければなりません。



改変禁止. あなたはこの作品を改変、変形または加工してはなりません。

- 再利用や頒布にあたっては、この作品の使用許諾条件を他の人々に明らかにしなければなりません。
- 著作[権]者から許可を得ると、これらの条件は適用されません。
- Nothing in this license impairs or restricts the author's moral rights.

Agenda

- Javaの特徴
- Javaプログラムを動かしてみよう
- Javaプログラムの基本構造を知ろう
- 基本文法
- クラスの連携
- オブジェクト指向でJavaを活用する
- 例外処理
- APIを使う
- デバッグの方法
- その他の文法

Javaの特徴

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

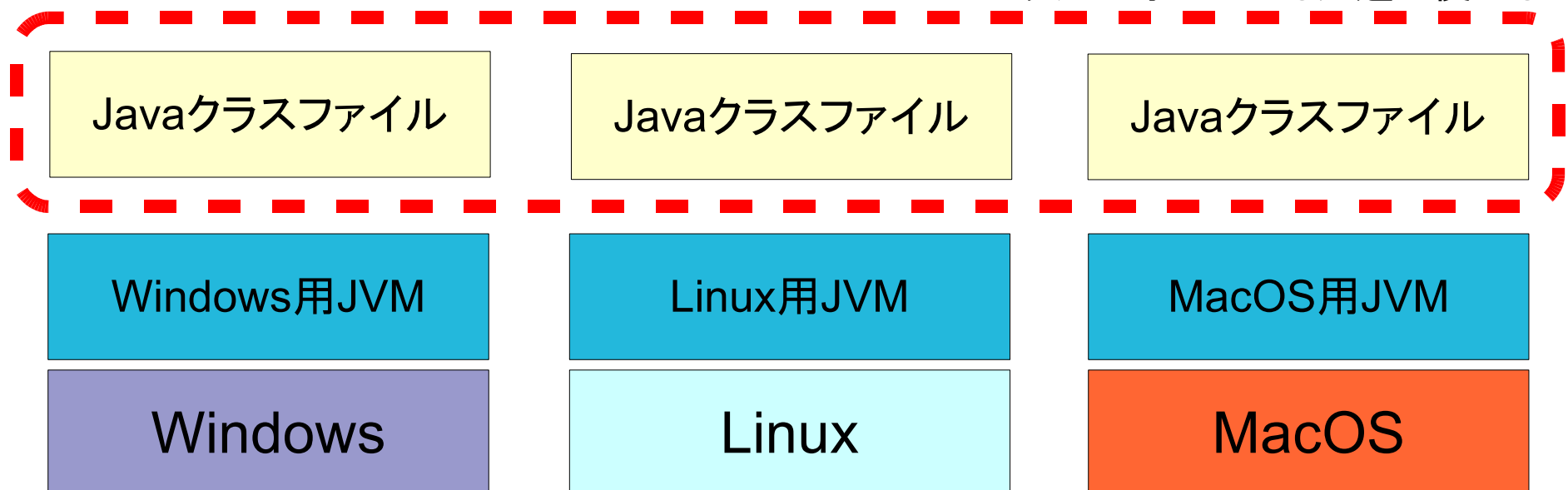
Javaの特徴

- オブジェクト指向言語 (Object Oriented)
 - JavaはOOで最も普及した言語のひとつ
- インタプリタ方式
 - 各OS専用のバイナリは作らず、中間コードをJavaVMに解釈させて実行させる
- プラットフォーム非依存
 - 同じコード・バイナリがOSを問わずそのまま動作する
 - WindowsでもMacでもUnixでも携帯でも
 - JavaVM (Java Virtual Machine) の提供

Java Virtual Machine(JVM/JavaVM)

- Javaのバイナリ形式をそのまま解釈して実行できる仮想的な機械 (Java Virtual Machine) を、プラットフォームごとに提供
- 各OSでは専用のJavaVMを使うことにより共通な実行結果を得ることができる

どのプラットフォームでも共通に使える



3つのエディション

- Java SE(J2SE)
 - Java Standard Edition
 - Javaの基本となるプラットフォーム
- Java EE(J2EE)
 - Java Enterprise Edition
 - エンタープライズ向けの機能を含んだプラットフォーム
- Java ME(J2ME)
 - Java Micro Edition
 - 携帯端末や組込み向けの機能を含むプラットフォーム

JDKとJRE

- JDK(Java Development Kit)
 - Javaの開発用環境
 - Java VM・コンパイラ・デバッガなどが含まれるパッケージ
- JRE(Java Runtime Environment)
 - Javaの実行用環境
 - Java VMなどが含まれているパッケージ
 - コンパイラやデバッガなどは含まれない

Javaのバージョン

- 1995 Java言語が発表される
- 1996 JDK 1.0リリース
- 1997 JDK 1.1リリース
- 1998 Java2(JDK1.2)リリース
- 2000 Java2 1.3リリース
- 2002 Java2 1.4リリース
- 2004 J2SE 5.0リリース
- 2006 JavaSE 6リリース

Javaプログラムを動かしてみよう

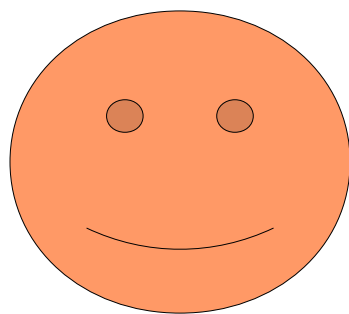
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Javaプログラムの作成から実行まで

- ソースファイルの作成
- コンパイル
- 実行

ソースファイルの作成

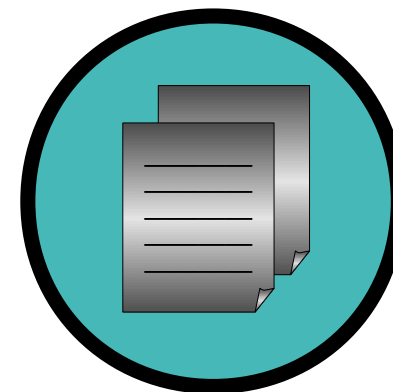
- ソースファイル
 - 拡張子は「.java」
 - エディターを使って記述する
 - 通常のテキストエディターでも編集可能



プログラマ



作成

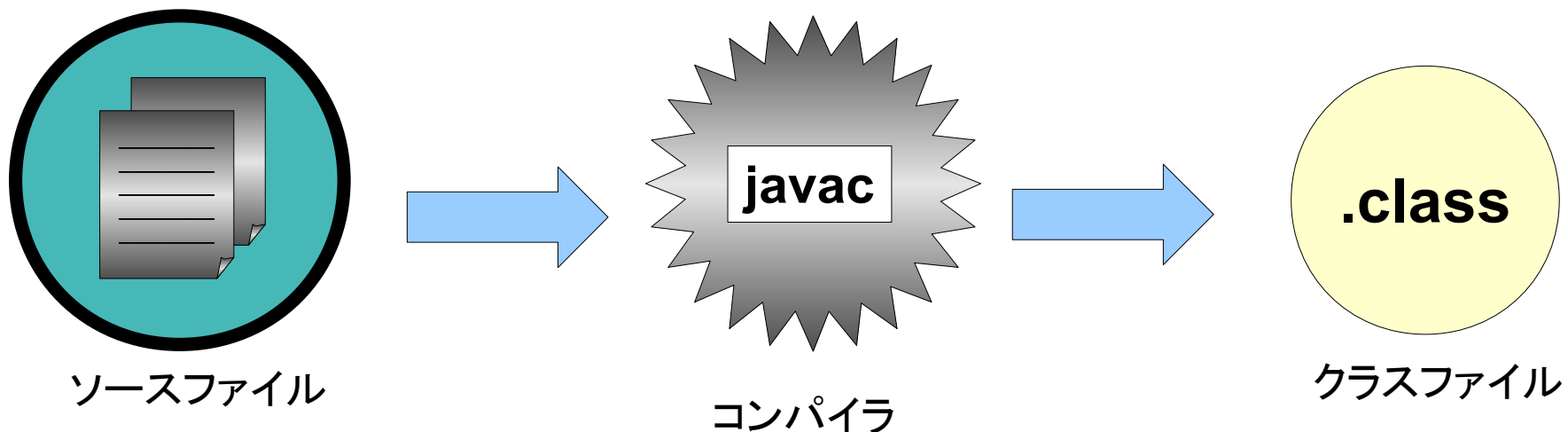


ソースファイル

コンパイル

- コンパイル

- javac というコマンド(コンパイラ)を使う
- コンパイラが実行される(コンパイル)とソースファイルの内容をJavaVMで実行することのできる「クラスファイル」が作成される
 - 拡張子は「.class」
 - クラスファイルはOSネイティブなバイナリではないので、JavaVM上でしか実行できない

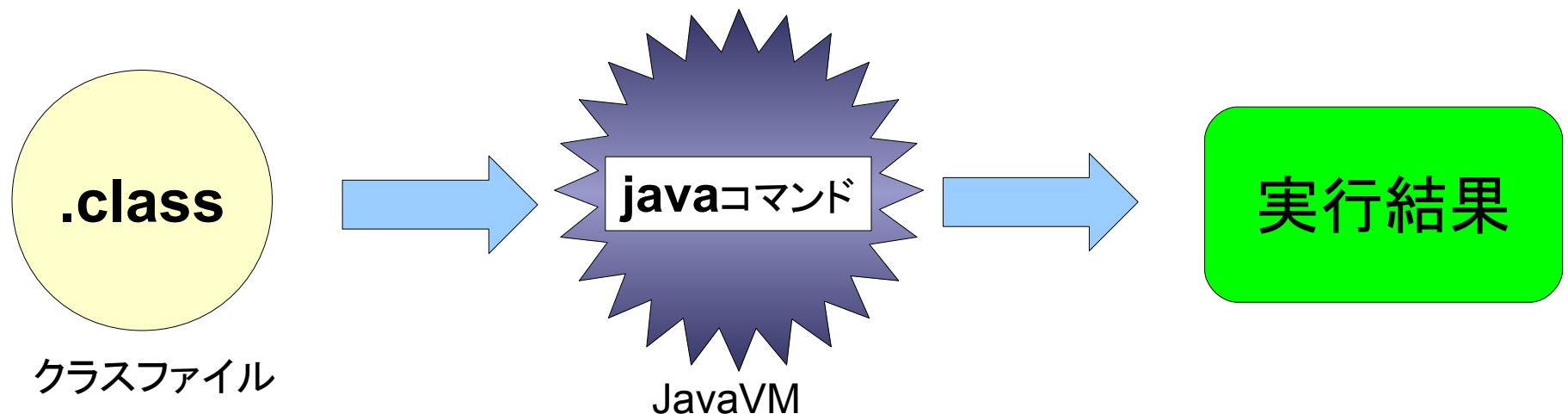


実行

- 実行

- java というコマンド(インタプリタ)を使う

- クラスファイルを指定すると、そのファイルをJavaVM上で実行してくれる



プログラムを作成してみる

- 開発環境
 - IDEというソフトウェアを使うのが一般的
- 代表的なJava用統合開発環境
 - Eclipse (Eclipse Foundation)
 - www.eclipse.org
 - NetBeans (NetBeans.org)
 - www.netbeans.org

IDE (統合開発環境) =
Integrated(統合された)
Development(開発)
Environment(環境)

プログラムを作成してみる

- EclipseでJavaプログラムを作成する
 - Eclipseを起動
 - プロジェクトの新規作成
 - クラスの新規作成
 - コンパイル(保存)
 - 実行

EclipseでJavaプログラムを作成

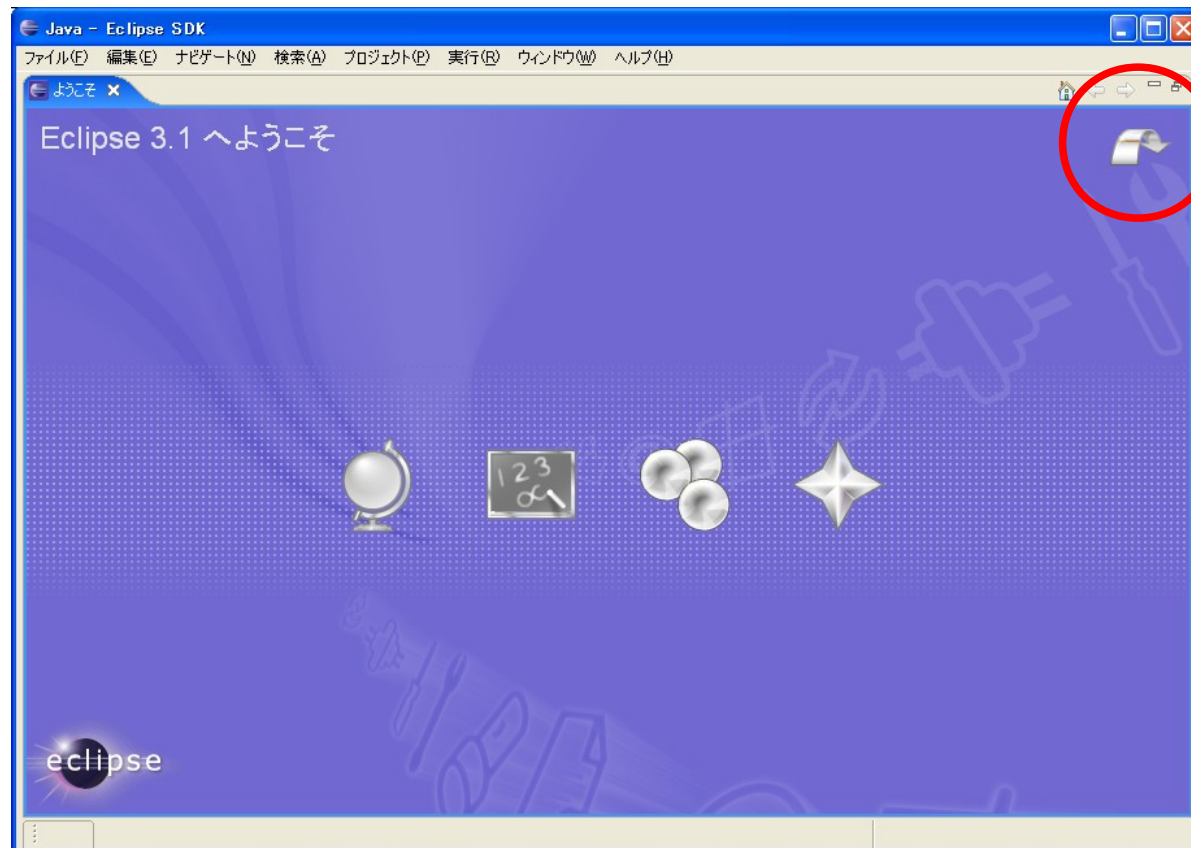
- Eclipseを起動
 - eclipse.exeをダブルクリック



eclipse.exe

EclipseでJavaプログラムを作成

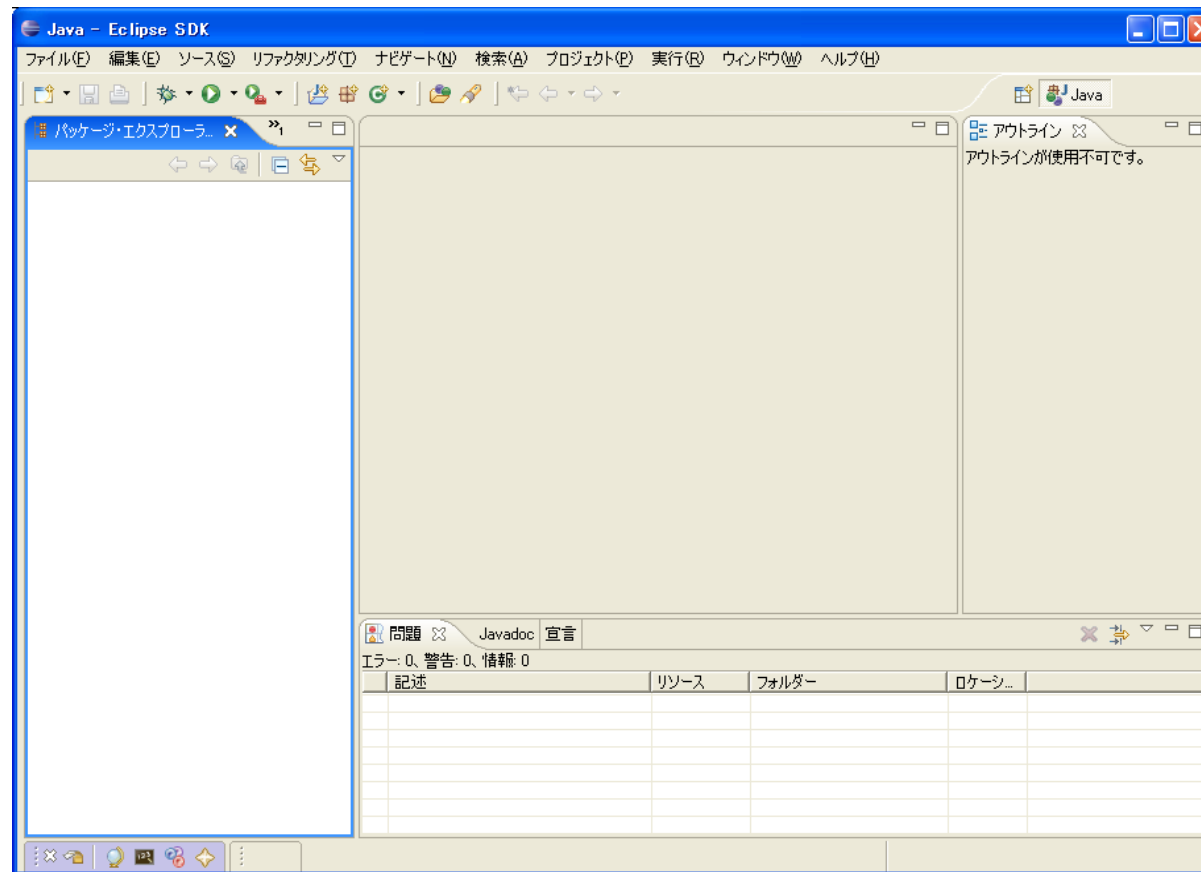
- Eclipseのメイン画面(ワークベンチ)が表示される
– 「ようこそ」画面



ここをクリック

EclipseでJavaプログラムを作成

- Javaプログラム作成用の画面構成 (Javaパースペクティブ) に切り替わる

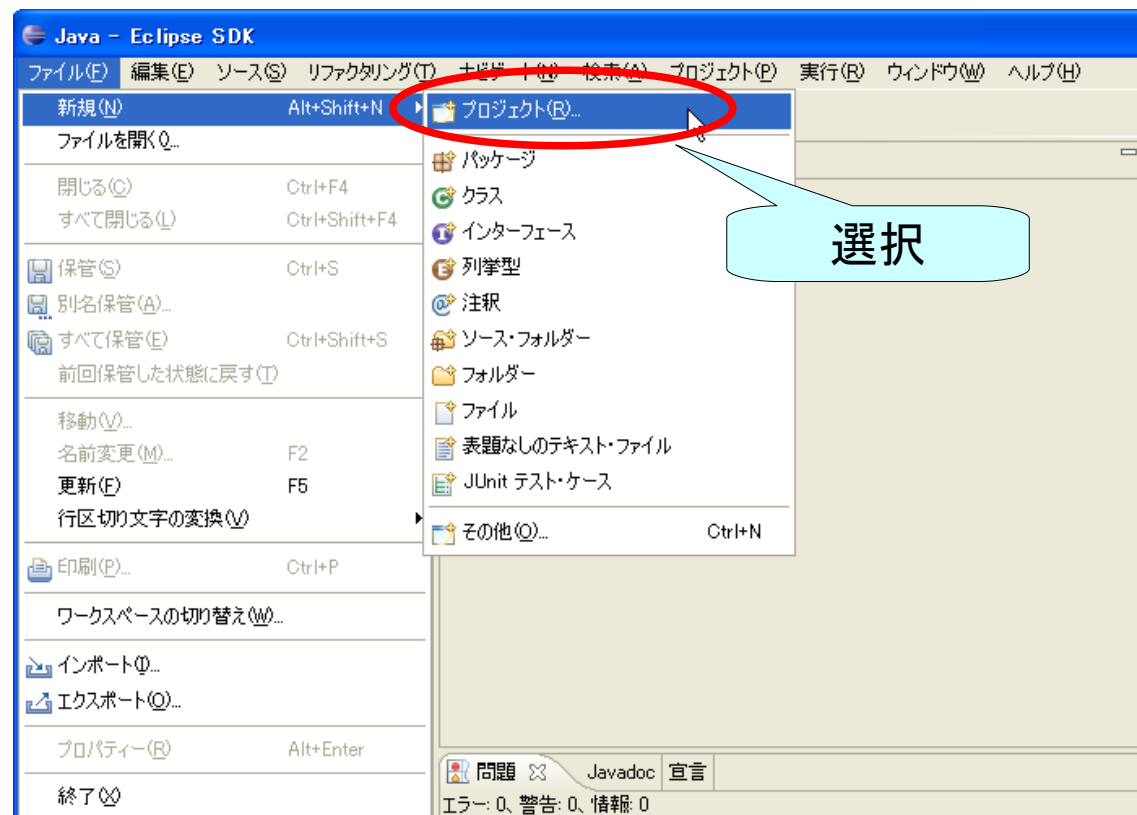


プロジェクトの新規作成

- プロジェクトとは？
 - 作成するアプリケーションの単位
 - Eclipse内で複数のアプリケーションを平行して開発できるように、「プロジェクト」という単位で分けて管理する
 - 異なるプロジェクトどうしは、特に設定しなければ互いに干渉しない

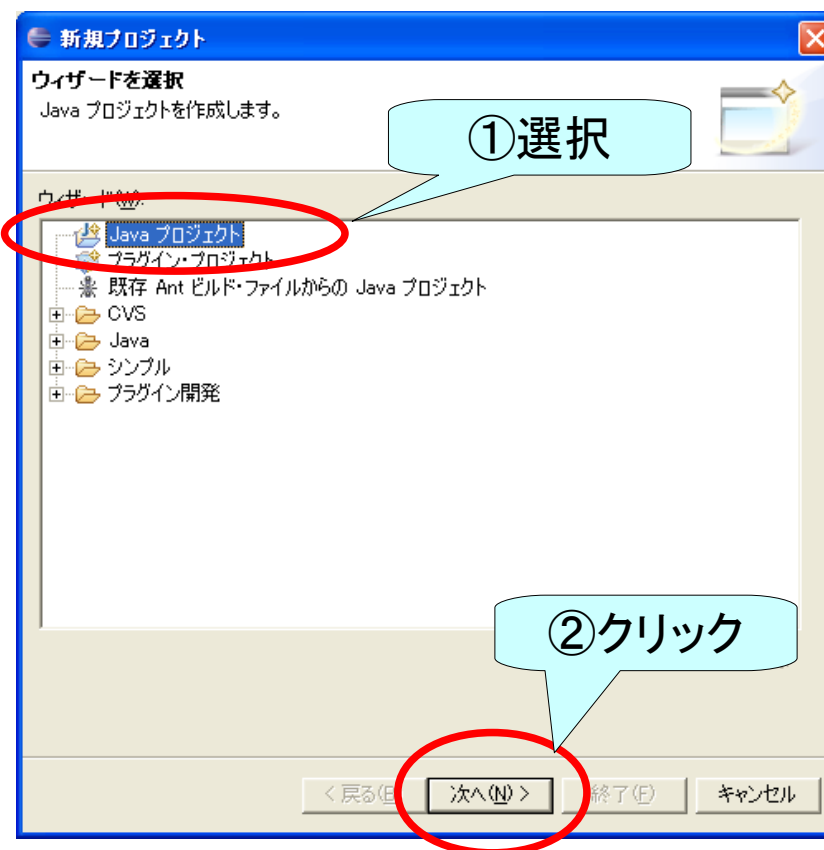
プロジェクトの新規作成

- Eclipseでプロジェクトを作成する
 - メニューから[ファイル]→[新規]→[プロジェクト]を選択



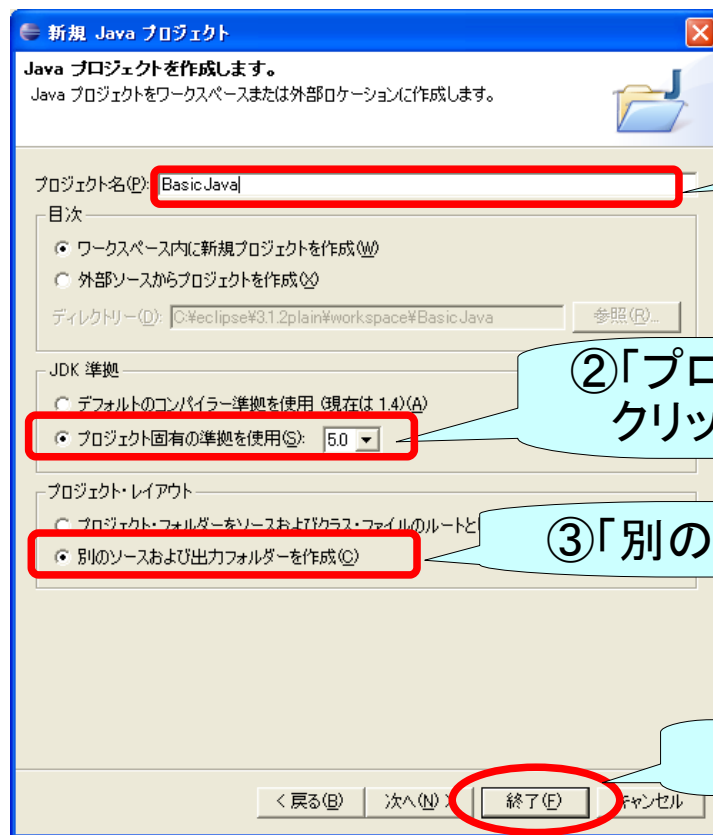
プロジェクトの新規作成

- Eclipseでプロジェクトを作成する
 - 「Javaプロジェクト」を選択して「次へ」をクリック



プロジェクトの新規作成

- Eclipseでプロジェクトを作成する
 - 「プロジェクト名」「JDK準拠」「プロジェクト・レイアウト」を編集



①プロジェクト名を入力(名称は任意)

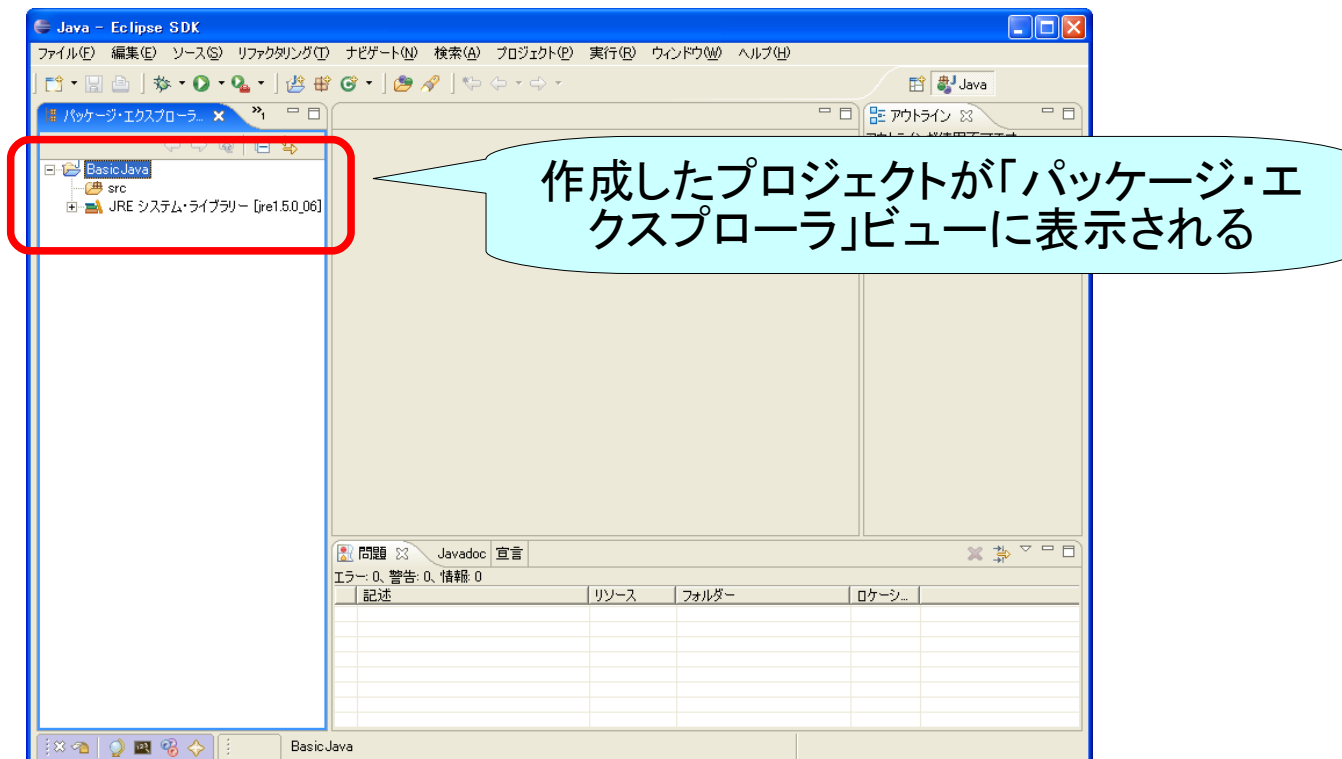
②「プロジェクト固有の準拠を使用」をクリックし、リストから「5.0」を選択

③「別のソースおよび出力フォルダーを作成」を選択

④クリック

プロジェクトの新規作成

- Eclipseでプロジェクトを作成する
 - プロジェクトの作成が完了



EclipseでJavaプログラムを作成

- クラスの作成

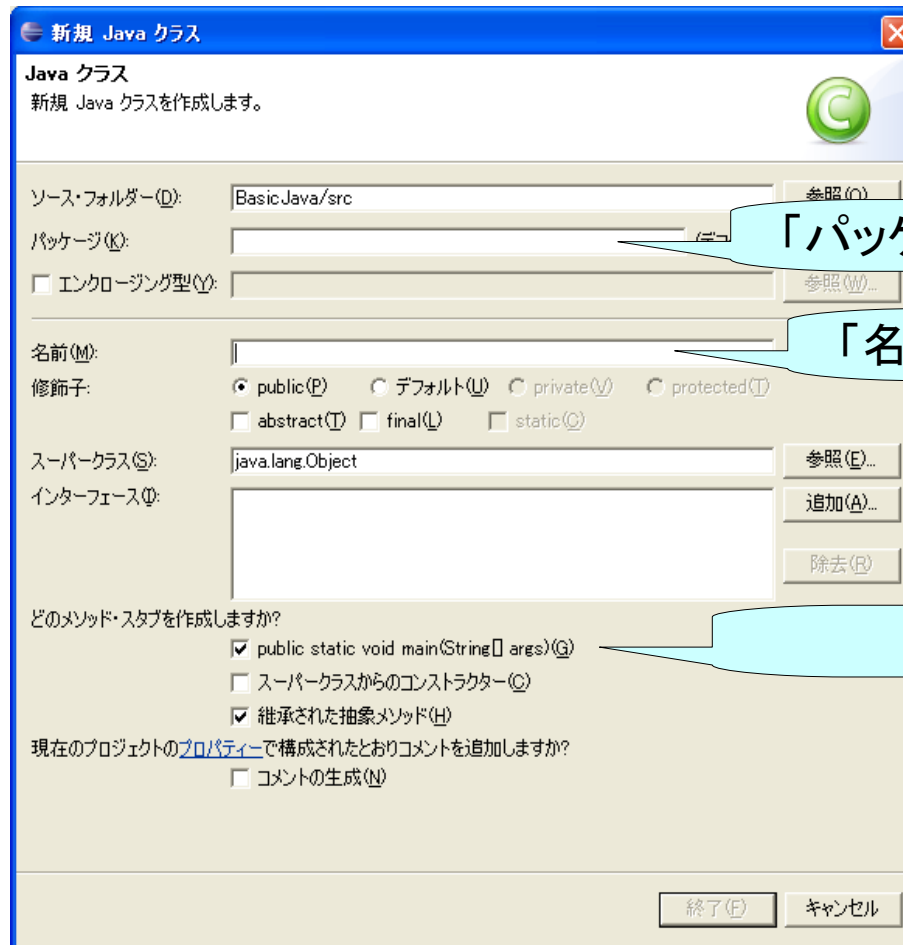
- クラスの新規作成メニューを使ってクラスを作成
 - ボタンバーの「新規Javaクラス」ボタンをクリック
 - または、「ファイル」→「新規」→「クラス」メニューを選択



「新規Javaクラス」
ボタンをクリック

EclipseでJavaプログラムを作成

- クラスの作成
 - クラスの新規作成メニューを使ってクラスを作成



「パッケージ」はとりあえず空でよい

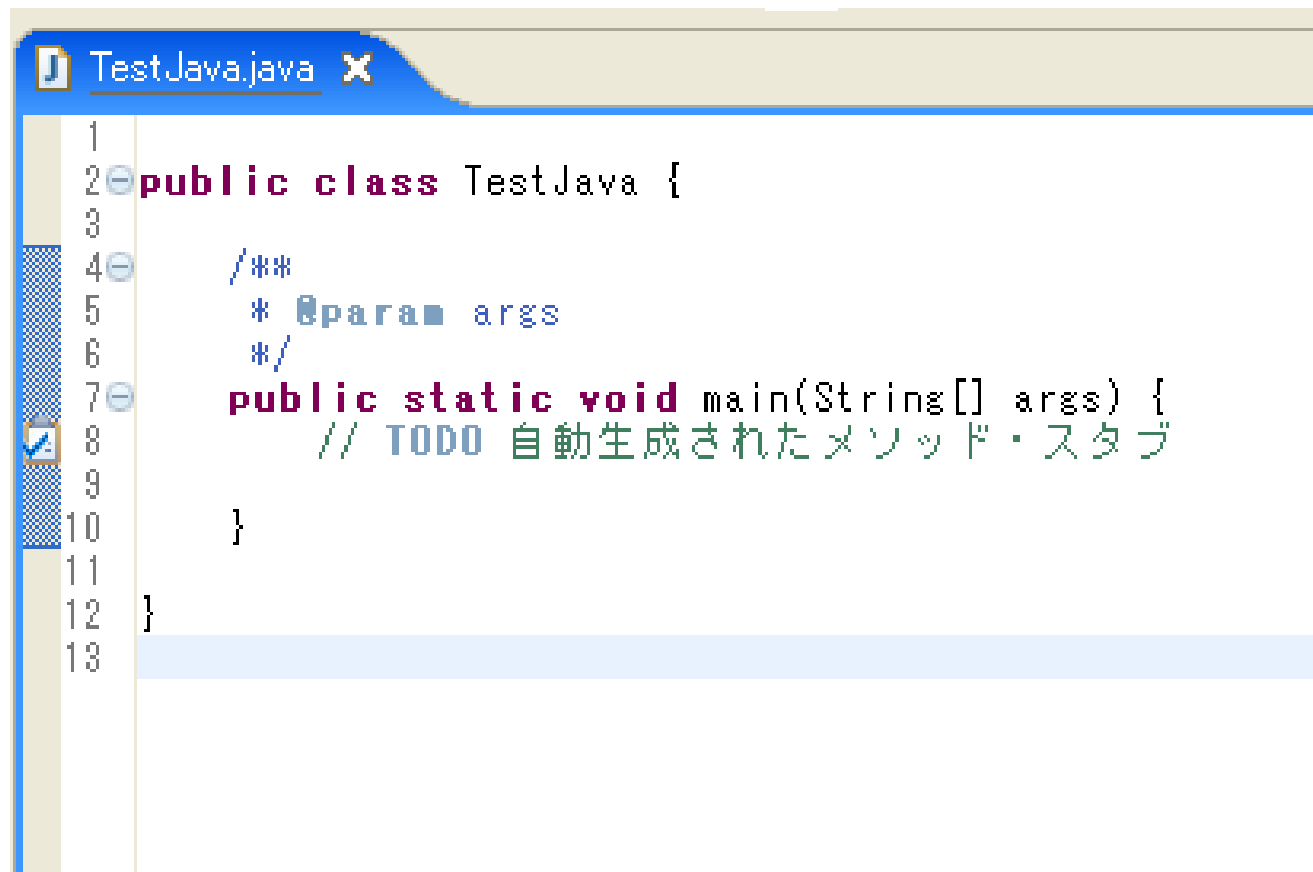
「名称」(クラス名)の先頭は大文字

チェックをつけておく

EclipseでJavaプログラムを作成

- クラスの作成

- 作成すると中央にエディターが開くので、そこにソースを記述していく



```
1
2 public class TestJava {
3
4     /**
5      * @param args
6      */
7     public static void main(String[] args) {
8         // TODO 自動生成されたメソッド・スタブ
9
10    }
11
12 }
13
```

EclipseでJavaプログラムを作成

- 処理内容を記述
 - mainメソッドの内容を入力

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

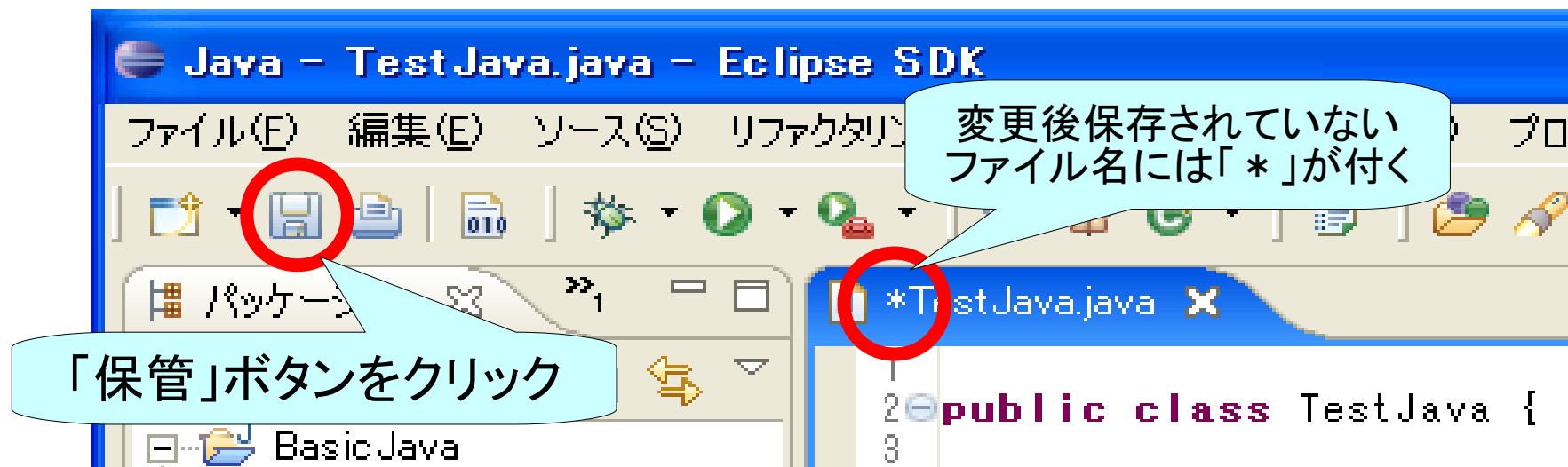
この行を入力

System.out.println()・・・()内に記述したものを、コンソールに出力してくれる

EclipseでJavaプログラムを作成

- コンパイル(保存)

- ソースを記述できたら、保存する
 - ボタンバーの「保管」ボタンをクリックするか、メニューの「ファイル」→「保管」を選択、もしくは、「CTRL+S」を押下
- Eclipseではソースを保存すると同時にコンパイルが行われる
 - Eclipse上からは見えないが、保存が終わると所定のフォルダにクラスファイルができています

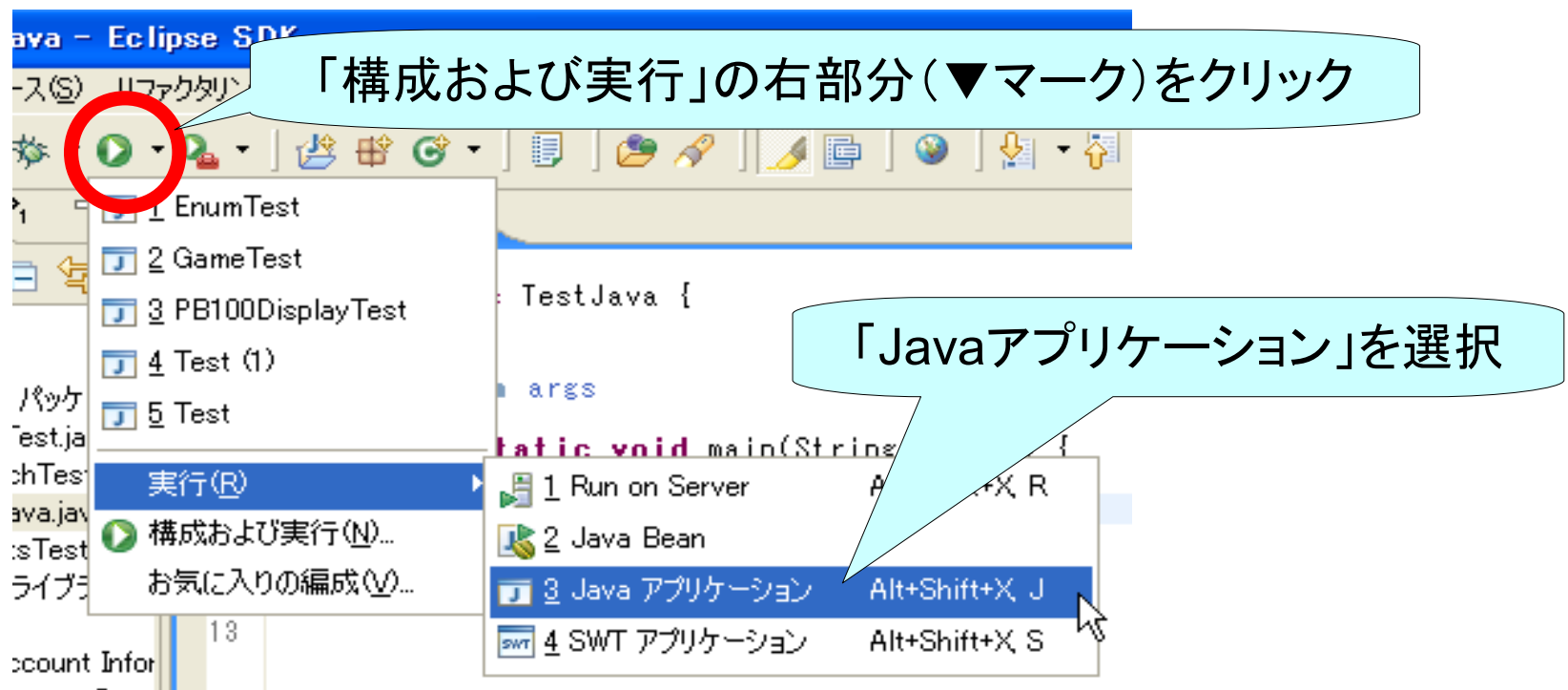


EclipseでJavaプログラムを作成

- 実行

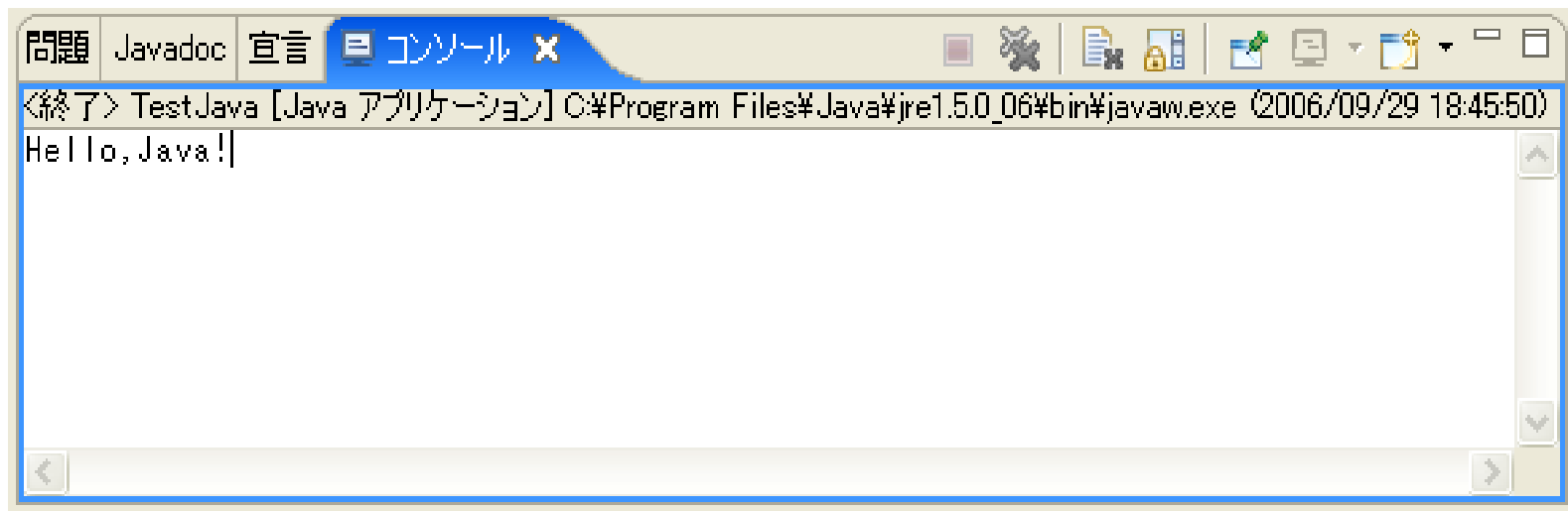
- Eclipseのメニューから実行する

- ボタンバーの「構成および実行」の右部分(▼マーク)を押下し、「実行」→「Javaアプリケーション」を選択



EclipseでJavaプログラムを作成

- 実行
 - 実行結果は「コンソール」ビューに表示される



Javaプログラムの基本構造を知ろう

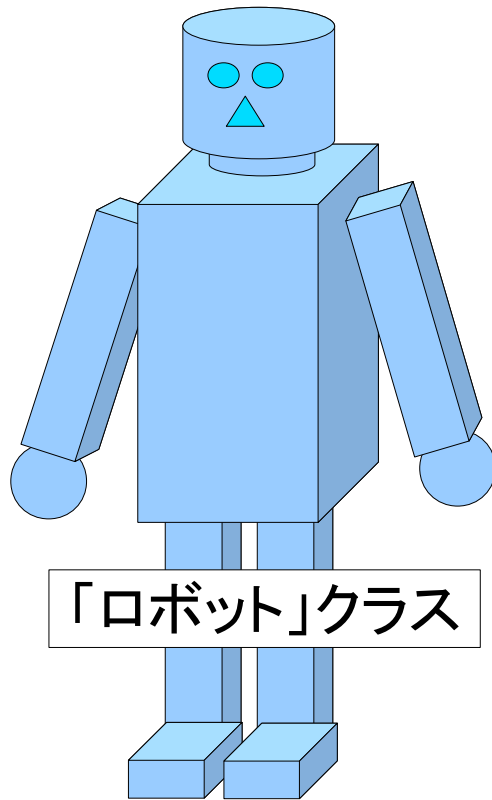
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

- クラス
- メソッド
- パッケージ
- データ型
- 変数
 - フィールド
 - ローカル変数

クラス

- クラスとは
 - 機能やデータ(状況)をまとめるひとつの単位



機能

- ・歩く
- ・話す
- ・計算する

データ

- ・名前
- ・バッテリー残量

クラス定義

- クラス定義部

- class [クラス名] の記述をクラス定義部という
- そこに定義されたプログラムは[クラス名]という名称のクラスであることが示される

クラス定義部 … 「class クラス名」

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

クラス定義

- クラスの範囲

- クラスの範囲は、クラス定義部に続く「{」から「}」まで

```
class HelloJava
```

```
{
```

ここから

```
    public static void main(String[] args) {
```

```
        System.out.println("Hello, Java!");
```

```
    }
```

```
}
```

ここまで

クラス定義

- クラス定義

凡例

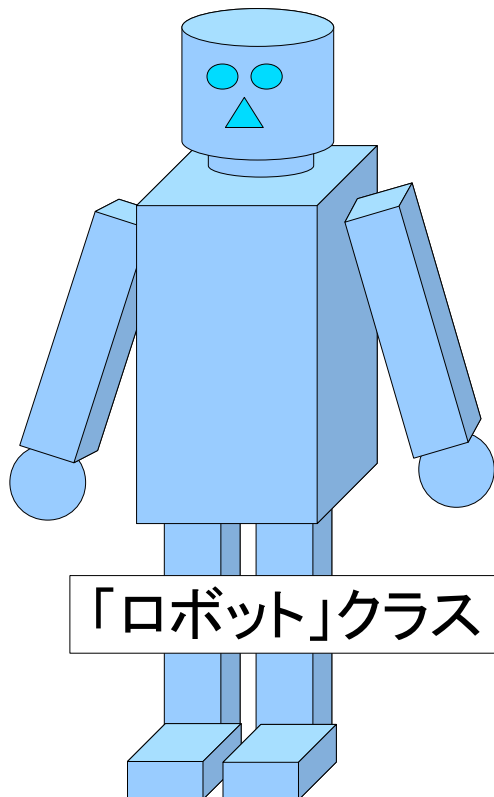
```
class クラス名 {  
  
}
```

クラス定義

- クラスは複数定義可能
 - ひとつのアプリケーションで、複数のクラスを用いることも可能(方法は後ほど)

メソッド

- メソッドとは
 - クラスがもつ機能を表現したもの
 - メソッド＝機能



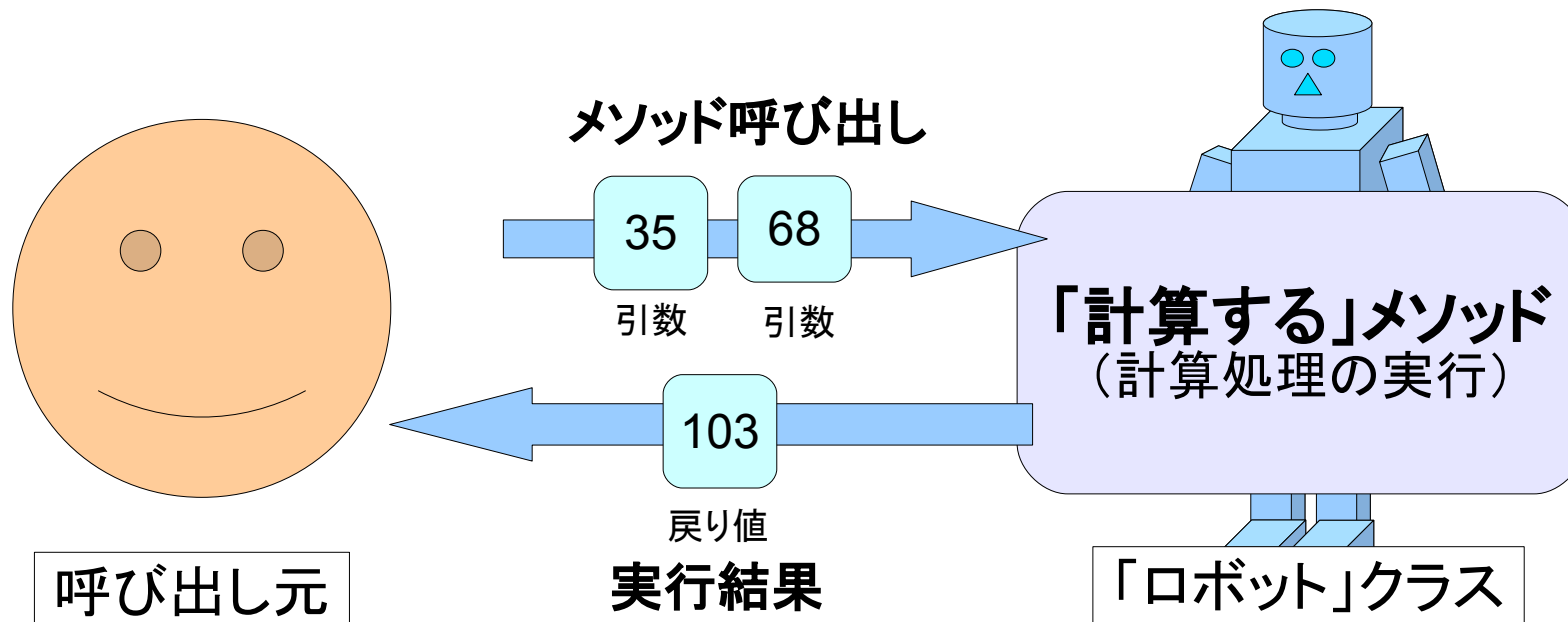
メソッド

- ・歩く
- ・話す
- ・計算する

メソッド

• メソッドを使う

- メソッドを呼び出すとそこに記述された処理が実行される
- メソッドを呼び出す際には、処理に必要な情報を「引数」として渡すことができる
- メソッドの実行が終わると、呼び出し元には「戻り値」と言われる結果が渡される(戻り値のないメソッドも作れる)



メソッド

- メソッド定義

凡例

```
戻り値の型 メソッド名(引数リスト) {  
    実行したい処理  
}
```

- ・メソッド定義は、必ずクラス定義の内側に記述する
- ・戻り値の型(データ型)については後ほど詳しく
- ・戻り値の型に「void」と指定すると、戻り値がないことを示す
- ・引数リストの部分には複数の引数を指定することができる

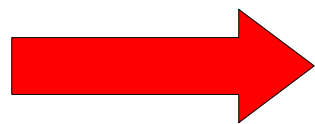
※「戻り値 メソッド(引数リスト)」の部分だけを「メソッドのシグネチャ」と呼ぶこともあります

メソッド

- メソッド定義の記述例

```
void walk() {  
    System.out.println("walk.");  
}
```

```
int plus(int a,int b) {  
    return a+b;  
}
```



上記のメソッドを、先ほど作成したクラスに書き加えてみましょう

メソッド

- mainメソッド

- プログラムを起動したとき、最初に実行されるメソッド
- メソッド定義の書式はあらかじめ決まっている
- 「public」「static」修飾子については後述

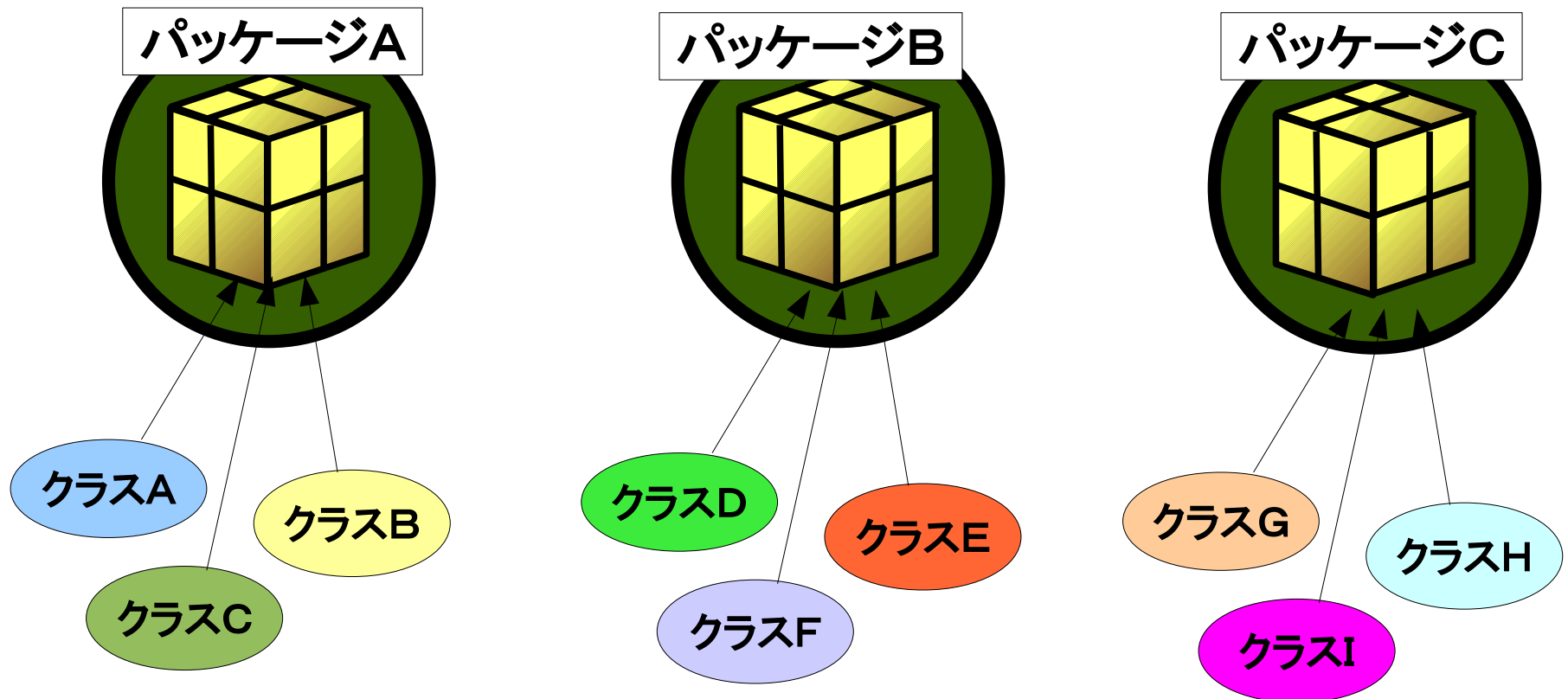
mainメソッドの定義

```
public static void main(String[] args) {  
  
}
```

パッケージ

- パッケージ

- クラスは「パッケージ」という単位で分けて管理することができる
- 意味や役割などのまとまりごとにパッケージで分類できる



パッケージ

- パッケージの命名基準
 - 公開して用いるクラスの場合
 - インターネットドメイン名をパッケージに必ず含める
 - ドメイン名は末尾から順に並べる
 - ドメイン名から後ろは、好きな名称をつけられる
 - 公開しない(テストや学習目的で作る)クラスの場合
 - 特に決まりはなく好きな名称をつけられる
- パッケージ名の区切り
 - パッケージ名は「.(ドット)」で区切った名称をつけられる

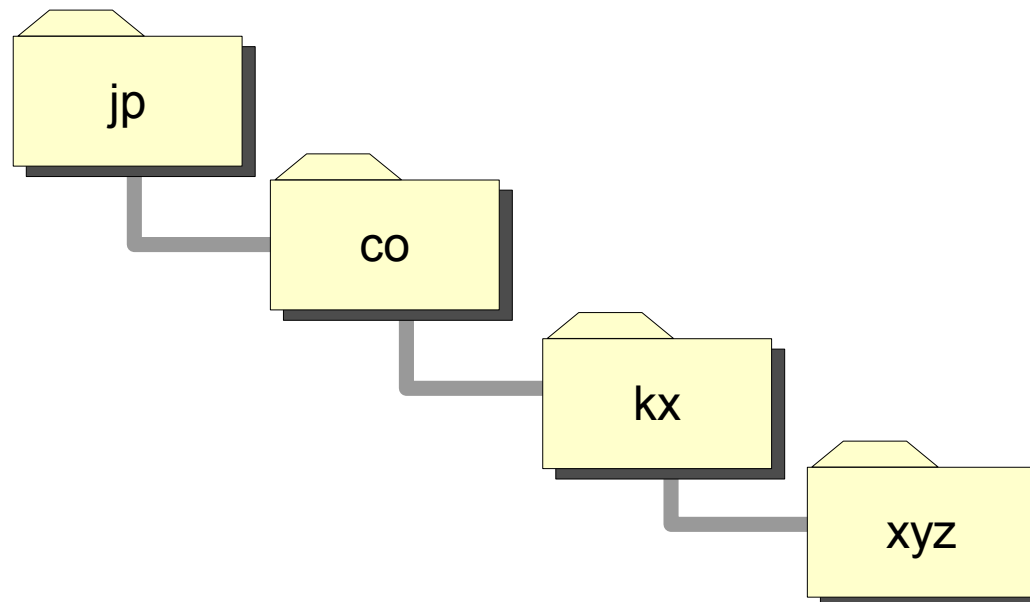
パッケージ

- パッケージ名の例
 - 「kx.co.jp」ドメインの組織が作る公開用パッケージの場合
 - jp.co.kx.xxx
 - jp.co.kx.yyy.zzz
 - など
 - 公開目的ではない、テスト用のパッケージの場合
 - util
 - test.model
 - database.connect

パッケージ

- パッケージの格納場所
 - パッケージ名は同名のフォルダ内に格納される
 - 「.」で区切るとフォルダ階層が作られる

jp.co.kx.xyzパッケージの場合



パッケージ

- パッケージを定義する
 - あるクラスをあるパッケージに含めるには、クラスの前頭に「package宣言」を記述する
 - 「パッケージ」だけを別個に作ることはない
 - 1クラスにpackage宣言はひとつだけ

凡例

```
package パッケージ名;  
  
class クラス名 {  
  
    ...  
  
}
```


パッケージ

- パッケージ宣言の例

```
package jp.knowledge_ex.util;  
  
class MailUtil {  
  
    void sendMail(String message) {  
        ...  
    }  
  
    ...  
}
```



適当なパッケージ名のクラスを新たに作成してみましょう。
Eclipse上で、パッケージに属するクラスはどのように表現されるでしょうか。
また、パッケージ構成を反映したフォルダが作られるかどうか確認してみましょう。

データ型

- データ型
 - Javaで扱うさまざまな値には「型」が決められている

データ型

- Javaのデータ型
 - 基本データ型
 - 整数、小数などよく用いられるデータのために用意された型
 - 参照型/クラス型(のちほど詳しく)
 - もっともよく使うクラス型…String型

データ型

- 基本データ型一覧

型名	内容	リテラルの範囲
int	32ビットの整数	-2147483648～2147483647
long	64ビットの整数	-9223372036854775808～9223372036854775807
byte	8ビットの整数	-128～127
short	16ビットの整数	-32768～32767
float	32ビットの単精度実数	$\pm(2-2^{-23}) \times 2^{127}$
double	64ビットの倍精度実数	$\pm(2-2^{-52}) \cdot 2^{1023}$
char	文字1字分を表す	'¥u0000'～'¥uffff'の2バイトUnicode文字
boolean	2値の論理値を表す	trueまたはfalse

データ型

- 数値表現のデータ型について
 - Javaでは、特に指定しなければ整数はint型、実数はdouble型で扱われる
 - そのため、変数の型についても、特に理由がなければ、整数はint型、実数はdouble型で定義するのが慣例となっている

データ型

- char型(基本データ型)
 - char型の値1つで、Unicode文字1文字を表現できる
 - 文字は、「'」(シングルクォート)で囲んで表す
 - 例) 'あ' 'a' '亜' など
 - 複数の文字から構成される「文字列」はchar型では表現できない

データ型

- String型(クラス型)
 - Unicode文字列(複数の文字)を表現できる
 - 文字列は、「”」(ダブルクオート)で囲んで表す
 - 例) ”あいうえお” ”abcde” など
 - String型は基本データ型ではないので、基本データ型と扱いの異なる部分があるいくつかあるので注意(後述)

データ型

- boolean型
 - リテラル(実現値)は、trueとfalseの2種類のみ
 - true ... 真(条件が成立、正しいの意)
 - false ... 偽(条件が成立しない、正しくないの意)

変数

- 変数

- あるデータを格納しておく入れ物
- Javaの変数には、必ず型定義が必要
- 変数を使う前には、必ず宣言を行う必要がある
- 定義した型以外の値を変数に代入することはできない
（「静的な型付け」という）

凡例

型名 変数名 = 初期値;

- ・「=初期値」は省略可能
- ・同じ型名の変数は、「,」で区切って複数同時に宣言可能

フィールドとローカル変数

- クラス内での変数の定義
 - フィールド
 - クラス定義の内部に宣言した変数
 - そのクラス内で共通に利用できる
 - 他のクラスから操作することも可能(くわしくは後ほど)
 - ローカル変数
 - あるメソッドの内部で定義した変数
 - そのメソッド内だけで利用できる
 - さらに範囲を限定して定義することも可能(くわしくは後ほど)

フィールドとローカル変数

- フィールド定義の例

```
class Person {  
    String name;  
    int age;  
}
```

フィールドとローカル変数

- ローカル変数定義の例

```
class Person {  
  
    String name;  
    int age;  
  
    void walk() {  
        int a = 10;  
        int b = 20;  
    }  
}
```

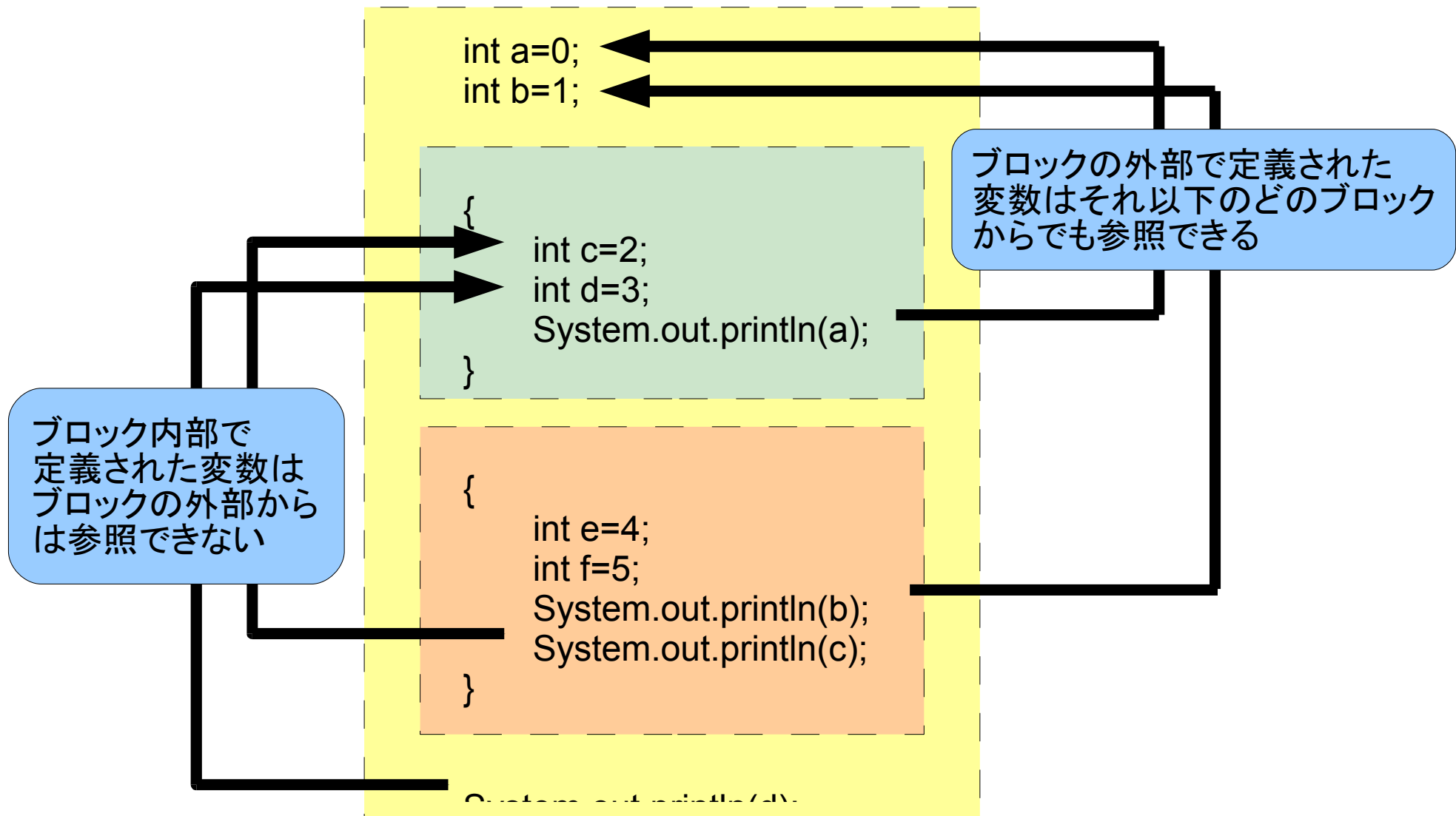


クラス内に、フィールドとローカル変数を両方定義してみましょう。

ブロックとスコープ

- Javaコードにはたびたび { } が登場するが...
- { ... } を「ブロック」と呼ぶ
- 変数の有効範囲(スコープ)は、ブロックによって決まる
 - 変数のスコープは、ブロックの内側である
 - ブロックの内側に定義した変数は、外側では無効

ブロックとスコープの関係



→ メソッド内に上記のコードを入力して、コンパイルしてみましょう

Javaの基本文法

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

- 文
- 式
- 演算子
- 制御文
- 配列

文

- 文
 - Javaの処理の最小単位
 - 「;」で終わるまでが一つの文
 - 制御文など一部例外もある
 - 文で表記する内容
 - メソッドの呼び出し
 - 演算や代入など

「文」の例

```
System.out.println("Hello!");  
int a = 100;  
c = a*a;
```

文とブロック

- 文とブロックの関係

- ブロックの範囲内には、複数の文を含めることができる
- メソッド定義にはブロックがあるので、メソッド内には複数の文を書くことができる
- 制御文でもブロックが多用される

```
public static void main(String[] args) {  
    int a = 100;  
    int c;  
    c = a*a;  
    System.out.println("Hello!");  
}
```

コメント

- プログラム中にコメントを記述する
 - コメントの部分は実行時には無視される

1行だけ有効なコメントを書きたい場合

```
// (コメントの内容...)
```

「//」から後ろはコメントとみなされる

任意の範囲で有効なコメントを書きたい場合

```
/* (コメントの内容...) */
```

「/*」から「*/」までがコメントとみなされる
コメントの開始と終了は複数行にまたがっていてもよい

式

- 式

- 演算子を用いて、演算を行う表現を「式」と呼ぶ
- 式だけで文を作ることは通常できないが、代入を伴う式は文とみなされる

```
a + b
256 * 16
c * ( a - b )
a = 100;
a++;
```

演算子

- 演算子
 - 演算を行うための記号

単項演算子 ... ひとつの値に対して演算するもの
二項演算子 ... 2つの値を使って演算するもの
比較演算子 ... 2つの値を比較し結果をtrueまたはfalseで表現するもの
論理演算子 ... 演算結果がtrueまたはfalseで表現されるもの
代入演算子 ... ある値を他の変数に代入するもの
再帰代入演算子 ... 二項演算結果の代入を略記する
インクリメント／デクリメント演算子 ... 1加算・減算する
条件演算子 ... 条件に応じて演算結果が変化する演算子
キャスト演算子 ... データの型を変換する演算子

単項演算子

- 単項演算子
 - ひとつの値に対して演算するもの

演算子	内容	表記例
-	正負の反転	- a

二項演算子

- 二項演算子
 - 二つの値をもちいて演算するもの

演算子	内容	表記例
+	加算	$a + b$
-	減算	$a - b$
*	乗算	$a * b$
/	除算	a / b
%	除算の余り	$a \% b$

比較演算子

- 比較演算子
 - 二つの値を比較し、結果をtrue/falseで表現するもの
 - 演算子による式が成立すればtrue、不成立ならばfalse

演算子	内容	表記例
==	等しい	a == b
!=	等しくない	a != b
>	より大きい	a > b
<	未満	a < b
>=	以上	a >= b
<=	以下	a <= b

参考 : instanceof 演算子

- instanceof 演算子
 - 二つのインスタンスを比較し、左辺が右辺と同一クラスまたは子クラスの場合にtrue、そうでない場合にfalseを返す

演算子	内容	表記例
instanceOf	左辺が右辺と同一クラスか、子クラスである	a instanceof b

論理演算子

- 論理演算子

- 一つまたは二つの論理値を論理演算し、結果を true/false で表現するもの
- 一つまたは二つの数値をビット演算し、結果を数値で表現するもの

演算子	内容	表記例
!	NOT演算(否定)	!a
&	AND演算(かつ)	a & b
	OR演算(または)	a b
^	XOR演算(どちらか一方)	a ^ b

参考：論理演算

- AND/OR/XOR演算
 - 左辺(a)、右辺(b)の値と演算結果の対応

	式	その値			
演算対象	a	true	false	true	false
	b	true	true	false	false
演算結果	a & b	true	false	false	false
	a b	true	true	true	false
	a ^ b	false	true	true	false

- NOT演算

式	その値	
a	true	false
!a	false	true

参考: OR演算子の記号

- OR演算子の記号はどこにある？
 - 「|」は、縦棒記号とよばれ、シフト+「¥」で入力します



ショートサーキット論理演算子

- ショートサーキット論理演算子
 - AND/OR演算の結果が、左辺の内容だけで判定できる場合に、右辺の評価を省略する演算子

演算子	内容	表記例
& &	AND演算(かつ)	a & & b
	OR演算(または)	a b

ショートサーキット論理演算

「a & & b」・・・aがfalseのときは、bの評価を省略→演算結果はfalse

「a | | b」・・・aがtrueのときは、bの評価を省略→演算結果は true

代入演算子

- 代入演算子
 - 右辺の値を、左辺の変数に代入する演算
 - 代入演算式は演算式だけで文として記述することができる

演算子	内容	表記例
=	右辺を左辺に代入	a = 10

再帰代入演算子

- 再帰代入演算子

- ある変数に対して二項演算を行い、結果をその変数に再び代入する演算子

演算子	内容	表記例	再帰代入演算子を使わずに書くと
+=	右辺を加算した値を代入	a += 10	a = a + 10
-=	右辺を減算した値を代入	a -= 10	a = a - 10
*=	右辺を乗算した値を代入	a *= 10	a = a * 10
/=	右辺を除算した値を代入	a /= 10	a = a / 10
%=	右辺で除算した余りを代入	a %= 10	a = a % 10

インクリメント/デクリメント演算子

- インクリメント/デクリメント演算子
 - 対象の値を1加算、1減算する

演算子	表記例	内容
++	++a (前置)	1加算してから値を評価
	a++ (後置)	値を評価してから1加算
--	--b (前置)	1減算してから値を評価
	b-- (後置)	値を評価してから1減算

インクリメント/デクリメント演算子

- インクリメント/デクリメント演算子の特徴
 - 主に変数に対して使う
 - 数値に対しても使えるがあまり意味がない
 - 通常、代入演算子以外の演算子は演算に使われる値そのものは変化しないが、インクリメント/デクリメント演算子では変数の値そのものが加算、減算される
 - そのため、インクリメント/デクリメント演算式だけで文として成立する

インクリメント/デクリメント演算子

- 前置と後置の違い
 - サンプルコード

```
public class ZenchiKouchi {  
    public static void main(String[] args) {  
        int a = 100;  
        int b = 100;  
        System.out.println(a++);  
        System.out.println(--b);  
    }  
}
```



上記のクラスを作成して、結果の違いを確認してみましょう

条件演算子

- 条件が成立するかしないかで演算結果が変わる演算子
 - if文(後ほど詳しく)で同じ内容を書くことも可能

演算子	内容	表記例
条件式 ? 値1 : 値2	条件式が成立すれば値1、成立しなければ値2を演算結果とする	(a > b) ? a : b

キャスト演算子

- ある値の型を別の型に変換する演算子
 - ある変数に、型の異なる変数や値を代入したい場合などによく使われる

演算子	内容	表記例
(型名)値	指定された値を、()内に記述された型に変換する	(int) a

キャスト演算子

- 暗黙のキャスト

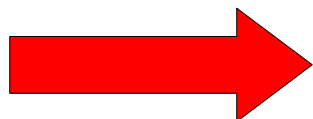
- 精度の低いものから高いものへのキャスト演算子は省略することができる
 - `int → double` = 小数部が0の実数として変換
 - `double → int` = 小数部を切り捨てないと整数にできない

省略できる	省略できない
<code>int → double</code>	<code>double → int</code>
<code>int → long</code>	<code>long → int</code>
<code>float → double</code>	<code>double → float</code>

コード例

```
int a = 100;  
double b = a;
```

```
double c = 3.1415;  
int d = (int)c;
```



上記のコード例を適当なメソッド内に実装しコンパイルしてみましょう

配列

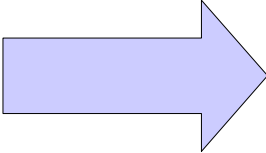
- 配列とは
 - 同じ変数名で、複数の値を管理する仕組み

配列を使わずに、複数の年齢データを表現

```
age_yamada = 30;  
age_tanaka = 26;  
age_suzuki = 43;  
age_nakamura = 19;  
⋮
```

人ごとに変数名が違うので、扱いが面倒…

配列を使って、複数の年齢データを表現



age[0]	30
age[1]	26
age[2]	43
age[3]	19
age[4]	⋮

変数名が同じなので、扱いやすい

配列

- 配列の作り方と使い方
 - 配列を生成する

```
型名[] 変数名 = new 型名[配列の大きさ]
```

または

```
型名 変数名[] = new 型名[配列の大きさ]
```

例) `int[] age = new int[5];`

配列

- 配列の作り方と使い方
 - 配列の生成と同時に値を設定する
 - 配列の大きさは、自動的に要素の数に設定される

型名[] 変数名 = {要素1,要素2,要素3···}

または

型名 変数名[] = {要素1,要素2,要素3···}

例)

```
int[] age = {23, 46, 51, 30, 19}
```


配列

- 配列の使い方

- 配列は各要素(それぞれの値)を数字で管理している
 - ※ 配列の[]内の数字を「添え字」と呼びます
 - ※ 添え字は0から始まることになっています
- 各要素を指し示す場合は、添え字に番号を書いて指定
 - 例1) `age[3] = 30;` → `age[3]`に30を代入
 - 例2) `System.out.println(age[1]);` → `age[1]`の値をコンソールに出力
- 定義時よりも大きい添え字を指定することはできない
 - `int[] age = new age[5];`と定義した場合は、5個分の領域が生成されたので、添え字番号は0～4を指定できる
- 一度生成した配列の大きさは、後から変更することができない(固定長配列)

配列

- 多次元配列
 - 2次元配列…表形式のデータを扱いたい場合

型名[][] 変数名 = new 型名[配列の大きさ1][配列の大きさ2]

例) int[][] area = new int[3][4]; と記述した場合

	0	1	2
0	100	89	36
1	0	13	47
2	92	21	68
3	75	54	-33

area[1][2] → 21
area[2][3] → -33
area[0][1] → 0

配列

- 多次元配列
 - 2次元より大きい配列＝3次元、4次元…の配列
 - [] の数を増やして定義すれば2次元配列と同じように生成可能
 - 次元数の最大値は255

制御文

- 処理の実行の順序を制御する文
 - 制御文を使わないと逐次処理(順次処理)しかできない
 - 逐次処理(順次処理)
 - 書かれている順に最初から最後までひとつずつ順番に処理すること
 - 制御文を使うことによって、
 - 分岐処理(条件によって処理する内容を変える)
 - 反復処理(同じ処理を繰り返し実行する)
 - その他
- のような処理をさせることが可能

制御文

- Javaにおける制御文
 - if文
 - switch～case文
 - for文
 - while文、do～while文
 - break文
 - continue文
 - return文
 - try～catch～finally文

if文

- if文 ～ 分岐処理の実現
 - 凡例①基本構文

```
if (条件式)  
    文1;
```

() 内の条件式が成立(演算結果がtrue)するとき、文1を実行

```
if (条件式) {  
    文1;  
    文2;  
    :  
}
```

() 内の条件式が成立(演算結果がtrue)するとき、ブロック内の文1、文2...を順に実行

if文

- if文を使ったコード例

```
int a=10,b=20;

if (a>b)
    System.out.println("a is greater than b");
```

```
int a=10,b=20;

if (a>b) {
    System.out.println("a is greater than b");
    a++;
    b--;
}
```

if文

- if文 ～ 分岐処理の実現

- 凡例② if ～ else構文

```
if (条件式) {  
    文1;  
    文2;  
    :  
}  
else {  
    文3;  
    文4;  
    :  
}
```

() 内の条件式が成立(演算結果がtrue)するとき、文1、文2…を順に実行

() 内の条件が成立しないときは、文3、文4…を順に実行

分岐させたい処理が1文ずつしかない場合は、ブロックを使わずに書くことも可能
(以下のif文の凡例でも同様)

if文

- if～else文を使ったコード例

```
int a=10,b=20;

if (a>b)
    System.out.println("a is greater than b");
else
    System.out.println("a is smaller than b");
```

```
int a=10,b=20;

if (a>b) {
    System.out.println("a is greater than b");
    a++;
} else {
    System.out.println("a is smaller than b");
    b--;
}
```

if文

- if文 ～ 分岐処理の実現
 - 凡例③ if ～ elseif ～ else構文

```
if (条件式1) {  
    文1; ...  
} else if (条件式2) {  
    文2; ...  
} else if (条件式3) {  
    文3; ...  
} else {  
    文4; ...  
}
```

(条件式1)が成立すれば文1…を順に実行して終了

(条件式1)が成立しなければ次のelse ifにある(条件式2)の式を判定し、(条件式2)が成立すれば文2…を順に実行して終了

(条件式2)が成立しなければ次のelse ifにある(条件式3)の式を判定し、(条件式3)が成立すれば文3…を順に実行して終了

どれも成立しなければ文4…を順に実行して終了

※ else ifは、複数繰り返して記述することも可能

※ else は、省略することも可能

if文

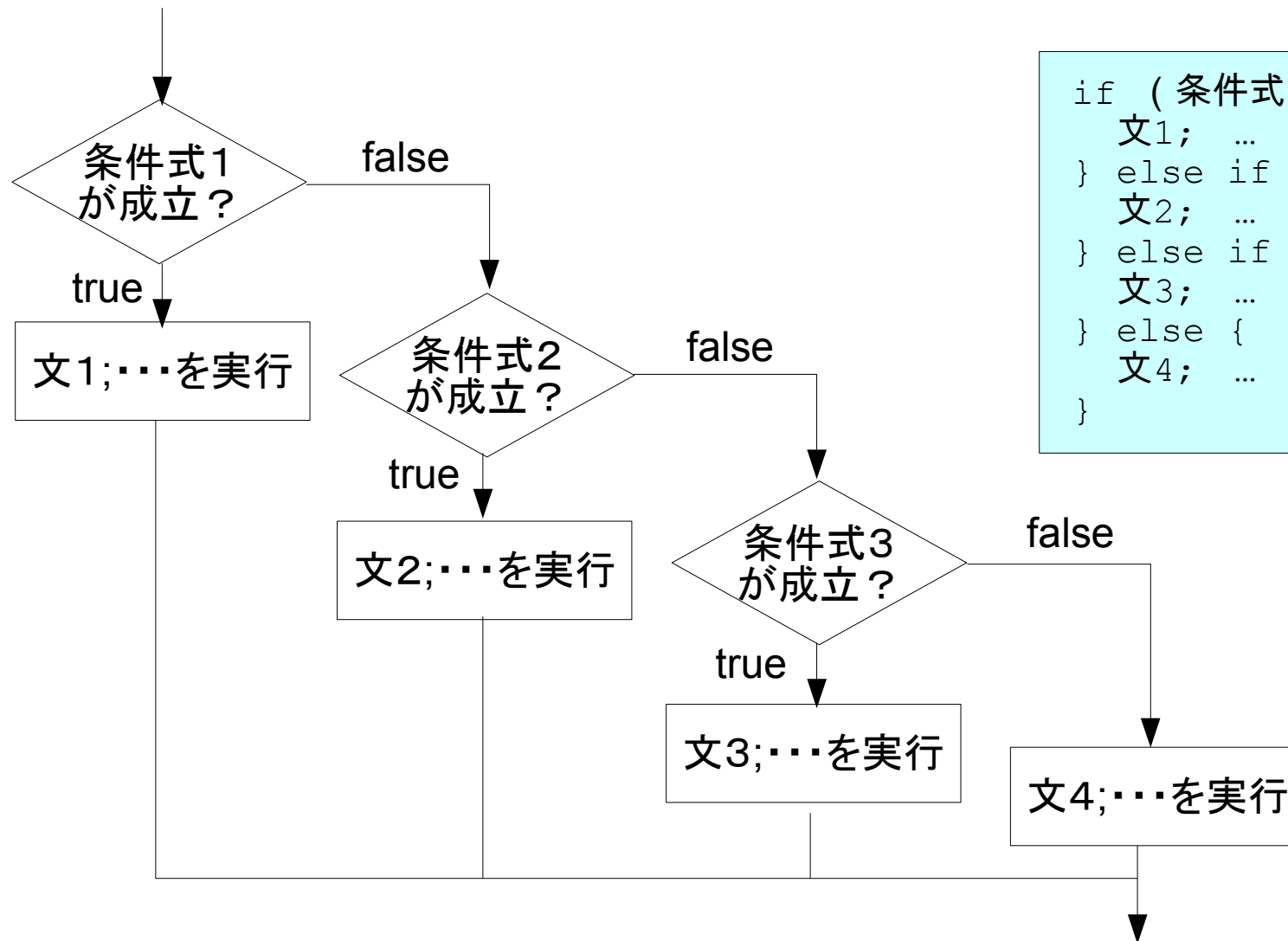
- if～else if～else文を使ったコード例

```
int a=10,b=20;

if (a>b) {
    System.out.println("a is greater than b");
} else if (a<b) {
    System.out.println("a is less than b");
} else {
    System.out.println("a equals b");
}
```

if文

- if ~ elseif ~ else構文の制御フロー



```
if ( 条件式1 ) {  
    文1; ...  
} else if ( 条件式2 ) {  
    文2; ...  
} else if ( 条件式3 ) {  
    文3; ...  
} else {  
    文4; ...  
}
```

switch～case文

- switch ～ case文
 - ある値の内容によって、処理を分岐する

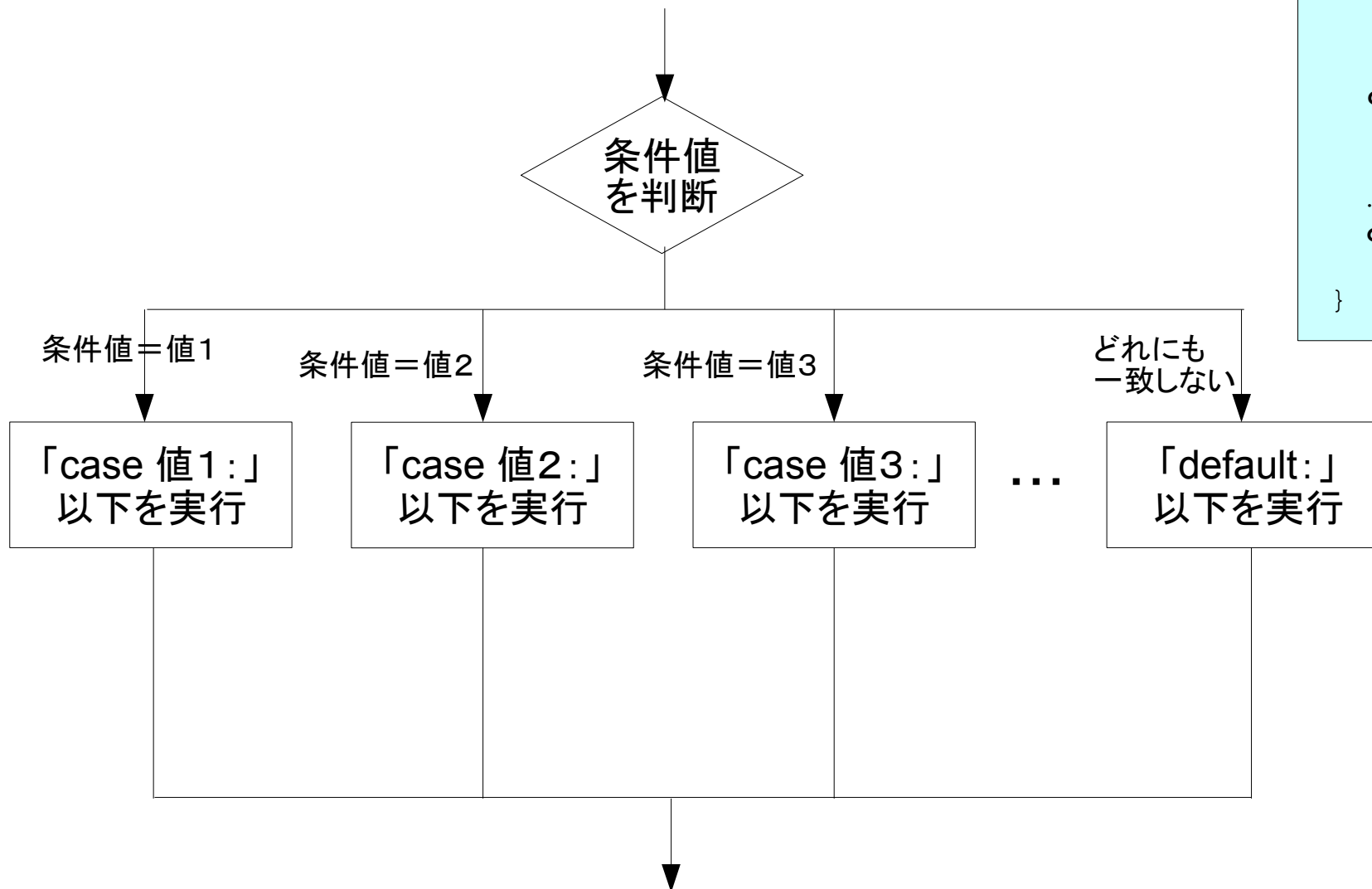
```
switch ( 条件値 ) {  
    case 値1:  
        文1; ...  
        break;  
    case 値2:  
        文2; ...  
        break;  
    ...  
    default:  
        文3; ...  
}
```

switch()内の値によって、

- 値1と等しければ、case 値1:からbreak;までを実行
 - 値2と等しければ、case 値2:からbreak;までを実行
 - どの値とも一致しなければ、default:からbreak;までを実行
-
- switch()内の値(条件値)は、int型およびenum型(後述)のみ指定可能
 - case～break;の間は、ブロックがなくても複数の文が記述可能
 - case節は、複数記述が可能
 - default節は、省略することも可能
 - 最後に書いたcaseまたはdefault節では、break;は省略しても良い

switch～case文

- switch ～ case文の制御フロー




```
switch ( 条件値 ) {  
    case 値1:  
        文1; ...  
        break;  
    case 値2:  
        文2; ...  
        break;  
    ...  
    default:  
        文3; ...  
}
```

switch～case文

- switch～case文を使ったコード例

```
public class SwitchCaseTest {  
    public static void main(String[] args) {  
        int a = 0;  
        switch(a) {  
            case 0:  
                System.out.println("Apple");  
                break;  
            case 1:  
                System.out.println("Orange");  
                break;  
            case 2:  
                System.out.println("Grape");  
                break;  
            default:  
                System.out.println("Others");  
        }  
    }  
}
```



上記のコードを作成し実行してみましょう。さらに、①aの値を適当に変更して、実行するとどうなるか②break;を削除して実行するとどうなるか、確認してみましょう。

for文

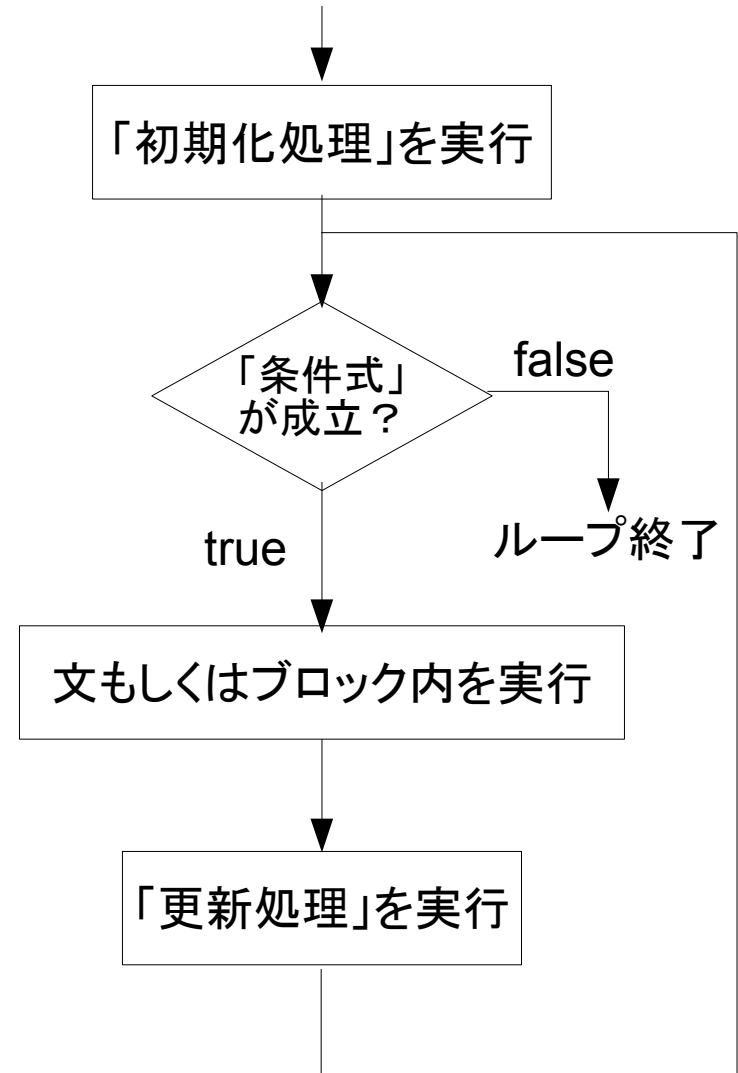
- for文
 - 反復処理を行う

```
for (初期化処理; 条件式; 更新処理)  
    文1;
```

```
for (初期化処理; 条件式; 更新処理) {  
    文1;  
    文2;  
    ...  
}
```

for文の処理手順

- ① () 内の「初期化処理」を実行
- ② () 内の「条件式」を判定し、
 - ②-1 条件式がtrueならば、文1; または、ブロック内の文 を順に実行
 - ②-2 条件式がfalseならば、for文を終了
- ③ () 内の「更新処理」を実行し、②へ



for文

- for文を使ったコード例

```
public class ForTest {  
    public static void main(String[] args) {  
        for(int i=0;i<5;i++) {  
            System.out.println(i);  
        }  
    }  
}
```



上記のコードを作成し実行してみましょう。
表示される数値から、ループの初期化文、条件式、更新処理の関係を考えましょう。

演習

- 全員の名前を順に表示するプログラム
 - 名前のデータはString型の配列で定義(下記参照)
 - 配列の内容をforループで表示する

定義する配列

添字	値
0	A.Jolie
1	B.Pitt
2	C.Diaz
3	D.Washington
4	E.Murphy

解答例

```
public class ForArray {  
    public static void main(String[] args) {  
        String[] a = {"A.Jolie","B.Pitt","C.Diaz","D.Washington","E.Murphy"};  
        for(int i=0;i<a.length;i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

拡張for構文

• 拡張for構文

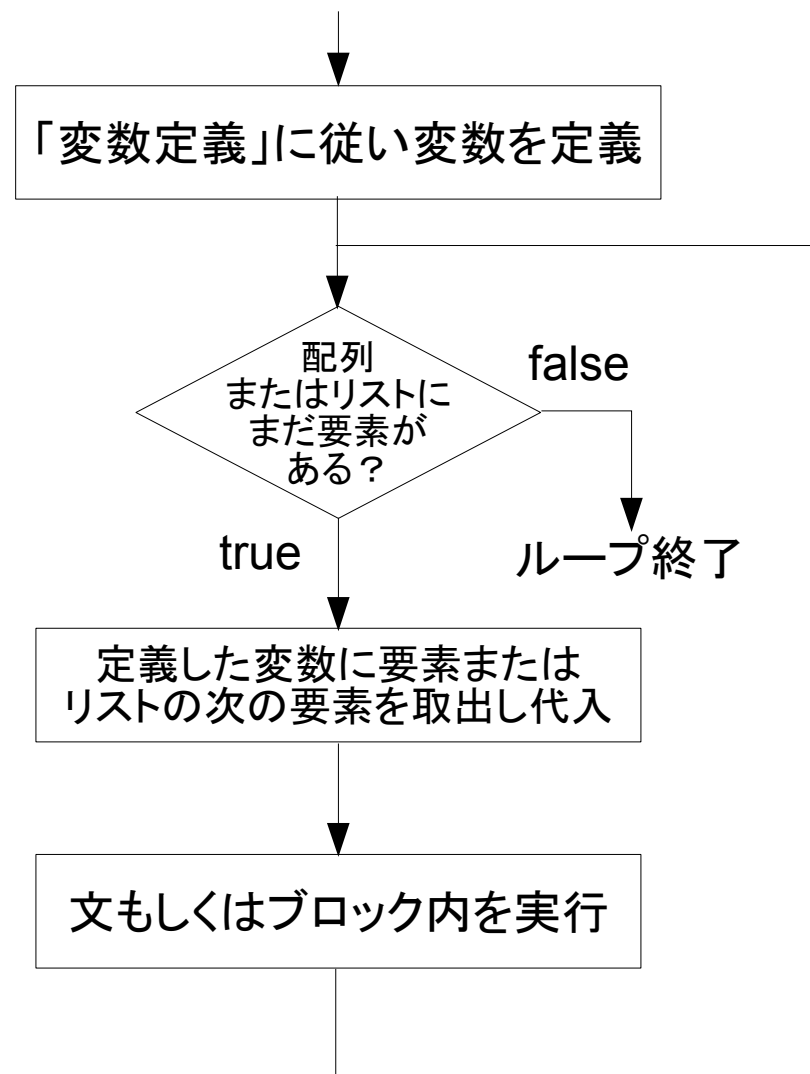
- 配列やリスト(後述)を対象とする
反復処理を行う(※J2SE 5.0から追加)

```
for(変数定義:配列名またはリスト名)  
    文1;
```

```
for(変数定義:配列名またはリスト名){  
    文1;  
    文2;  
    ...  
}
```

拡張for構文の処理手順

- ①()内の「変数定義」に従い変数を定義
- ②()内の「配列またはリスト」から、1番目の要素を取り出し、①の変数に代入
- ③ 文1;または、ブロック内の文 を順に実行
- ④()内の「配列またはリスト」から、次の要素を取り出し、①の変数に代入して③へ
- ⑤()内の「配列またはリスト」から取り出す要素がなくなったらループを終了



拡張for構文

- 拡張for構文を使ったコード例

```
public class ForEachTest {  
    public static void main(String[] args) {  
        int[] age = {19,23,30,45,58};  
        int sum = 0;  
        for(int a:age) {  
            sum += a;  
        }  
        System.out.println(sum);  
    }  
}
```

演習

- スライド#106で作成したクラスを、拡張for構文を用いたものに書き直してみましょう

解答例

```
public class ForArray {  
    public static void main(String[] args) {  
        String[] a = {"A.Jolie","B.Pitt","C.Diaz","D.Washington","E.Murphy"};  
        for(String s:a) {  
            System.out.println(s);  
        }  
    }  
}
```

while文・do～while文

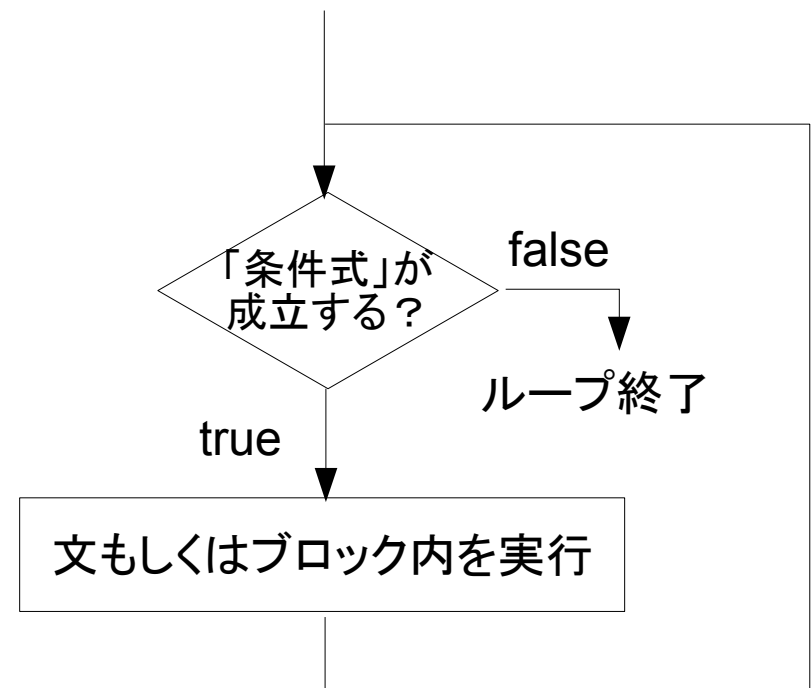
- while文
 - 反復処理を行う

```
while(条件式)  
    文1;
```

```
while(条件式) {  
    文1;  
    文2;  
    ...  
}
```

while文の処理手順

- ① () 内の「条件式」を判定し、
 - ①-1 条件式がtrueならば、文1; または、ブロック内の文 を順に実行
 - ①-2 条件式がfalseならば、while文を終了
- ② 文またはブロック内の処理が終了したら①へ戻る



while文・do～while文

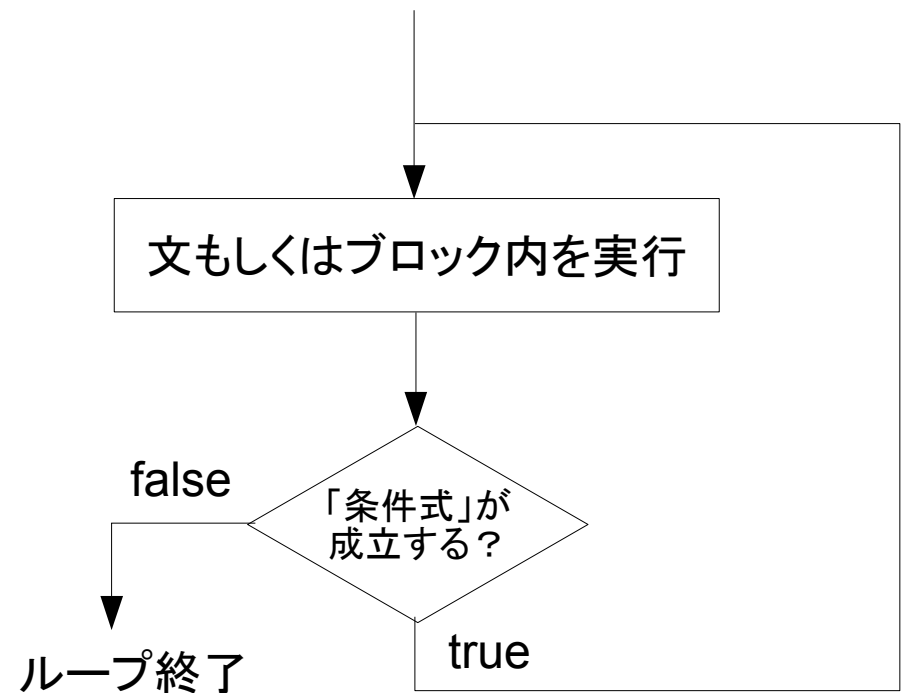
- do～while文
 - 反復処理を行う

```
do  
  文1;  
while(条件式)
```

```
do {  
  文1;  
  文2;  
  ...  
} while(条件式);
```

do ～ while文の処理手順

- ① 文またはブロックを処理を実行
- ② () 内の「条件式」を判定し、
 - ②-1 条件式がtrueならば、①に戻る
 - ②-2 条件式がfalseならば、do～while文を終了



while文とfor文、do～while文

- while文とfor文の使い分け
 - for文…ループする回数が定まっている場合に有効
 - while文…回数が不定で、ループ処理した結果で繰り返すかどうか決まるような場合に有効
- while文とdo～while文の違い
 - while文…最初に条件判断をするので、状況によっては、一度もループが実行されないことがある
 - do～while文…ループ処理を実行した後で条件判断をするので、状況にかかわらず少なくとも1回はループが実行される

break文

- break文

- 制御を中断する

- 反復処理を途中で中断して終了
 - switch～caseの分岐処理を終了し、構文の外へ
 - 制御文が入れ子になっている場合は、今いる制御文だけを中断し、外側へ

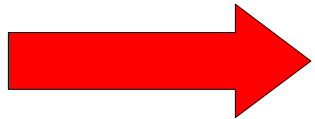
コード例

```
for(int i=0;i<10;i++) {  
    System.out.println(i);  
    if (i>5) {  
        break;  
    }  
}
```

break文

- 入れ子の制御文でbreak文を用いた例

```
for(int i=1;i<10;i++) {  
    for(int j=1;j<10;j++) {  
        System.out.println(i+"x"+j+"="+ (i*j));  
        if (j>5) {  
            break;  
        }  
    }  
}
```



上記のコードを実行し、どのような結果になるか確認してみましょう。

continue文

- continue文
 - その回のループを中断し、次のループから継続する
 - 反復処理そのものが終了するわけではない
 - breakは制御文自体を終了させる

return文

- return文

- メソッドの処理を終了し、呼び出し元に制御を戻す
 - メソッドの途中であっても、その時点で終了する
- メソッドに戻り値が必要な場合は、return文で戻り値を指定する
 - 戻り値がvoidの場合は、メソッドの末尾で自動的にメソッド処理が終了するので、return文は省略しても良い

凡例

戻り値がvoid(戻り値なし)のメソッドを終了する場合

```
return;
```

戻り値がvoid以外のメソッドを終了する場合

```
return 戻り値;
```

return文

- return文のコード例

```
public void walk(boolean b) {  
    System.out.print("I can walk ");  
    if (!b) {  
        return;  
    }  
    System.out.println("very well.");  
}
```

```
public int multiply(int a,int b) {  
    return a*b;  
}
```

try～catch～finally文

- try～catch～finally文
 - 例外処理を扱うための制御文
 - 詳しくはのちほど

クラスの連携

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

クラスの連携

- クラスの連携

- クラスは単独で動作することが可能
- ひとつのクラスにアプリケーションの全てのコードを記述すると、コードが大きくなり過ぎ、管理が大変
- クラスは単独ではなく、複数を連携して動作させることもできる

クラスの連携

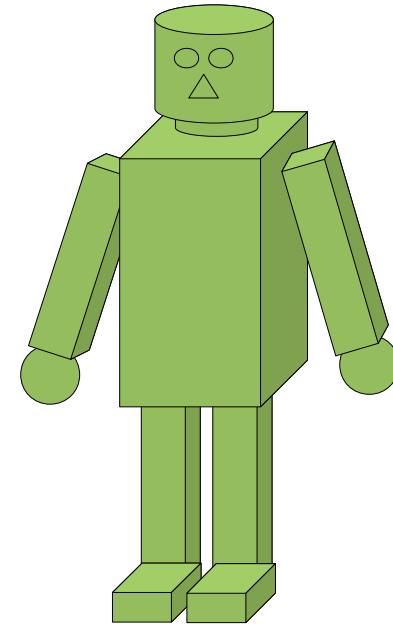
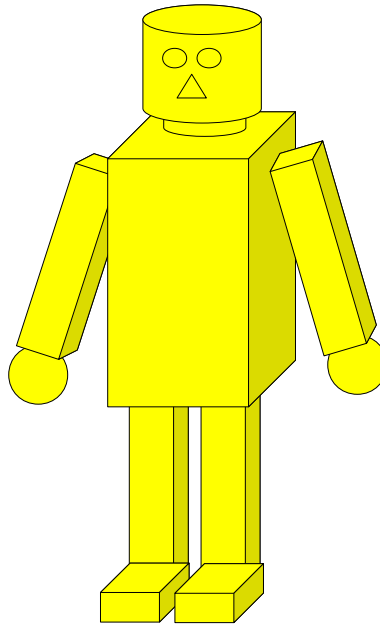
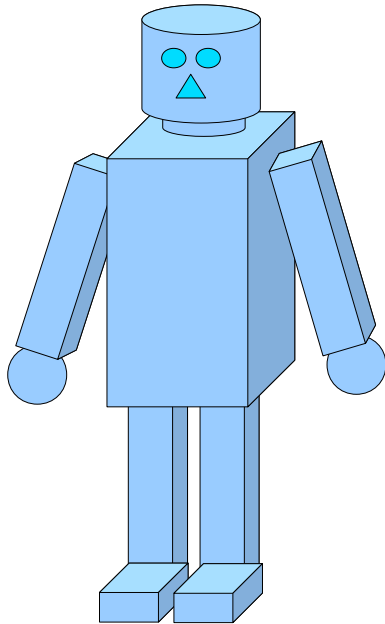
- どのように連携するか
 - ①あるクラスから、他のクラスのメソッドを呼び出して実行
 - ②あるクラスから、他のクラスのフィールドを参照、変更する

クラスの連携

- 相手のクラスへアクセスする手順
 - A) 対象クラスのインスタンスを作成してから、メソッドやフィールドにアクセス
 - B) static定義されたメソッドやフィールドに直接アクセス

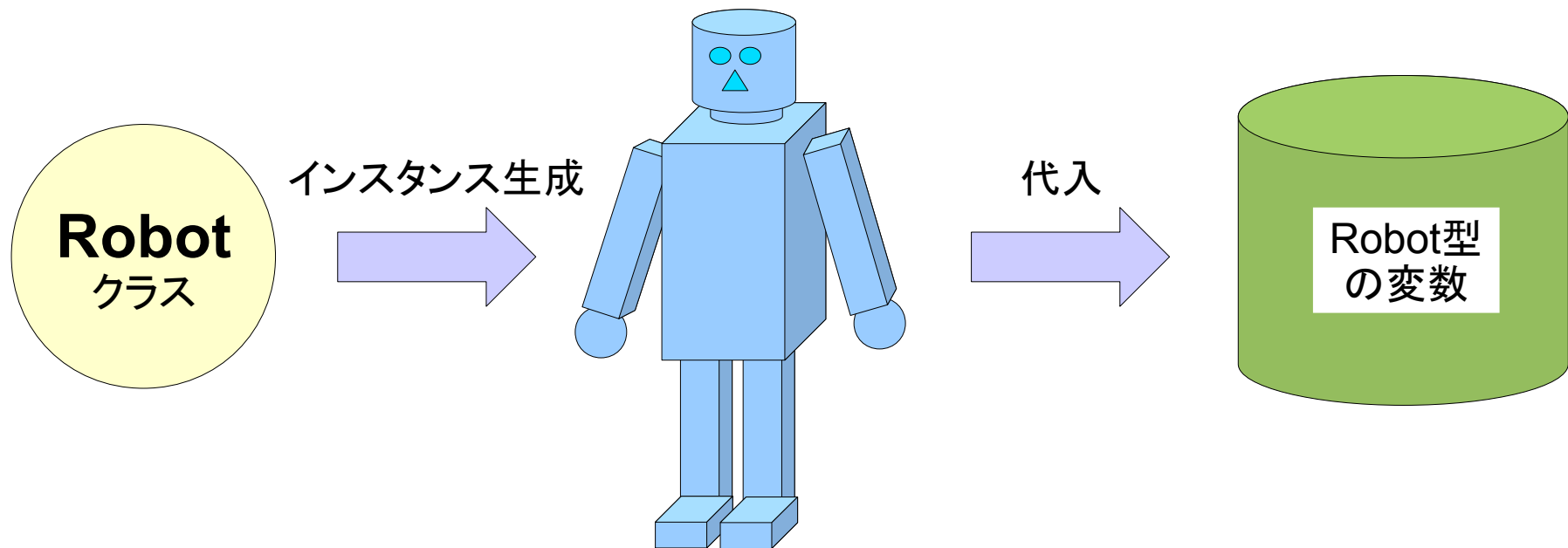
クラスの連携

- インスタンスとは何か
 - 同じクラスで、状態の異なるものを複数扱いたい
 - 例えば、デザインや性能の異なる「ロボット」クラスを複数作って処理を実行させたいなど



クラスの連携

- インスタンスとは何か
 - Javaではクラスに対して、それぞれ独立した固有の状態をもつ実体→インスタンスを作ることができる
 - インスタンスは複数作成できるため、それぞれを区別するために異なる変数に代入して扱うことができる



クラスの連携

- インスタンスの生成手順
 - 凡例

クラス名 変数名 = new コンストラクタ指定;

↑
クラス名(引数リスト)

※ 基本データ型の変数に対して、クラス名を型名とする変数を「クラス型」「参照型」と呼びます

クラスの連携

- コンストラクタとは？
 - インスタンスを生成する際、初期化処理をするために定義する特別なメソッド
- コンストラクタの特徴
 - メソッド名がクラス名と必ず同じである
 - 戻り値は指定できない
 - 引数の異なるコンストラクタを複数定義可能
 - コンストラクタ自体を省略することも可能

クラスの連携

- コンストラクタ定義されたクラスの例

```
public class Person {  
    public int age;  
  
    public Person() {  
        age = 0;  
    }  
}
```



上記のコードを作成し、さらに別のクラスのメソッド内で、Personクラスのインスタンスを作成するコードを記述してみましょう。

クラスの連携

- 複数のコンストラクタを使い分ける
 - インスタンス生成時の「コンストラクタ指定」で、どのような引数を指定するかで呼ばれるコンストラクタが決まる
 - コンストラクタは引数リストによって区別されている

クラスの連携

- コンストラクタを複数定義したクラスの例

```
public class Person {  
    public int age;  
  
    public Person() {  
        age = 0;  
    }  
  
    public Person(int a) {  
        age = a;  
    }  
  
}
```



上記のコードを先ほどのPersonクラスに追加し、さらに別のクラスのメソッド内で、それぞれのコンストラクタを使ってPersonクラスのインスタンスを作成するコードを記述してみましょう。

クラスの連携

- 作成したインスタンスにアクセスする手順

内容	凡例
メソッドを呼び出す	インスタンスを格納した変数名・メソッド名(引数リスト)
フィールドにアクセスする	インスタンスを格納した変数名・フィールド名

クラスの連携

- インスタンスの特徴
 - それぞれのインスタンスで、独立したフィールドの値を持つことができる
 - あるインスタンスのフィールドを書き換えても、他のインスタンスのフィールドには影響しない(後述するstaticフィールドを除く)
 - インスタンス内のメソッドから、自分自身の持つメソッドやフィールドにアクセスする場合は、「メソッド名(引数リスト)」「フィールド名」のみで指定可能

クラスの連携

- 作成したインスタンスにアクセスするコード例

```
Person p1 = new Person();  
System.out.println(p1.age);  
p1.printAge();
```



先ほどのPersonクラスに、自分の年齢(ageフィールド)をコンソールに表示するprintAgeメソッドを追加したのちに、上記のコードを他のクラスで実行してみましょう。

クラスの連携

- static定義されたメソッド・フィールドへのアクセス
 - staticキーワードをつけて定義されたメソッド（staticメソッド）やフィールド（staticフィールド）には、インスタンスを生成しなくても、直接アクセスすることができる

凡例

staticフィールド

```
static 型名 変数名（＝初期値）；
```

staticメソッド

```
static 戻り値の型名 メソッド名（引数リスト）；
```

クラスの連携

- static定義されたメソッド・フィールドにアクセスする手順
 - インスタンスが存在しているとは限らないので、クラス名を使ってアクセスする

内容	凡例
メソッドを呼び出す	クラス名 . メソッド名 (引数リスト)
フィールドにアクセスする	クラス名 . フィールド名

- インスタンスが存在していれば、格納した変数名でアクセスすることも可能(ただしあまり推奨されない)
- 同一クラスのstaticメソッド/フィールドにアクセスする場合は、クラス名は省略可能

クラスの連携

- static定義されたメソッド、フィールドをもつクラスの例

```
public class MoneyCalc {  
  
    static double rate = 117.94;  
  
    public static double dollarToYen(int dollar) {  
        return dollar*rate;  
    }  
  
    public static double yenToDollar(int yen) {  
        return yen/rate;  
    }  
  
}
```

クラスの連携

- static定義されたメソッド、フィールドへアクセスするクラスの例

```
public class CalcTest {  
  
    public static void main(String[] args) {  
        int dollar = 150;  
        int yen = 25800;  
        System.out.println(dollar+"ドルは、"+  
            MoneyCalc.dollarToYen(dollar)+"円です");  
        System.out.println(yen+"円は、"+  
            MoneyCalc.yenToDollar(yen)+"ドルです");  
    }  
  
}
```



上記のクラスを作成して実行し、結果を確認してみましょう。

クラスの連携

- staticフィールド・staticメソッドの制限
 - staticフィールド
 - インスタンスの有無に関わらず、値を1カ所でしか保持できない
 - staticメソッド
 - staticメソッド内からは、同一クラス内のstatic定義でないメソッド(インスタンスメソッドという)や、static定義でないフィールド(インスタンス変数という)に直接アクセスすることはできない
 - staticメソッド内でインスタンスを作れば、アクセス可能

クラスの連携

- staticメソッド、staticフィールドの制限を確認するコード例(1)

```
public class MoneyCalc {  
  
    double rate = 117.94;  
  
    public static double dollarToYen(int dollar) {  
        return dollar*rate;  
    }  
  
    public static double yenToDollar(int yen) {  
        return yen/rate;  
    }  
  
}
```

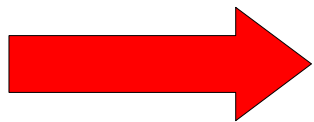


先ほど作成したMoneyCalcクラスのrateフィールドのstatic修飾子を削除し、クラスをコンパイルするとどうなるでしょうか。

クラスの連携

- staticメソッド、staticフィールドの制限を確認するコード例(2)

```
public class StaticTest {  
  
    public double dollarToYen(int dollar) {  
        return dollar*rate;  
    }  
  
    public static void main(String[] args) {  
        double d = dollarToYen(100);  
    }  
  
}
```



上記のコードを作成しコンパイルすると、どうなるでしょうか。

クラスの連携

- 他のパッケージのクラスを使う場合
 - 自分と異なるパッケージに属するクラスを利用する場合、クラス名は「パッケージ名.クラス名」で指定する

```
package animal;  
  
public class Monkey {  
    int age;  
    public Monkey() {  
        age = 0;  
    }  
}
```

異なるパッケージのクラスを利用

```
package human;  
  
public class Person {  
    public static void main(String[] args) {  
        animal.Monkey m = new animal.Monkey();  
    }  
}
```

クラスの連携

- import宣言によるパッケージ指定の省略
 - クラスの先頭に「import宣言」を行っておくと、該当するパッケージ名の指定を省略することができる

1クラスについて指定

```
import パッケージ名.クラス名;
```

あるパッケージに属する全クラスについて指定

```
import パッケージ名.*;
```

クラスの連携

- import宣言を使ったコード例

```
package human;  
  
import animal.Monkey;  
  
public class Person {  
    public static void main(String[] args) {  
        Monkey m = new Monkey();  
    }  
}
```


クラスの連携

- static import
 - static定義されたフィールドやメソッドに対してimport宣言すると、パッケージ名だけでなくクラス名の指定も省略することができる
 - J2SE 5.0で追加された新機能

1フィールド、1メソッドについて指定

```
import static パッケージ名.クラス名.フィールド名(メソッド名);
```

あるクラスに属する全staticフィールド、staticメソッドについて指定

```
import static パッケージ名.クラス名.*;
```

クラスの連携

- static importを使ったコード例

```
package util;

public class MoneyCalc {
    static double rate = 117.94;
    public static double dollarToYen(int dollar) {
        return dollar*rate;
    }
    public static double yenToDollar(int yen) {
        return yen/rate;
    }
}
```

```
import static util.MoneyCalc.*;

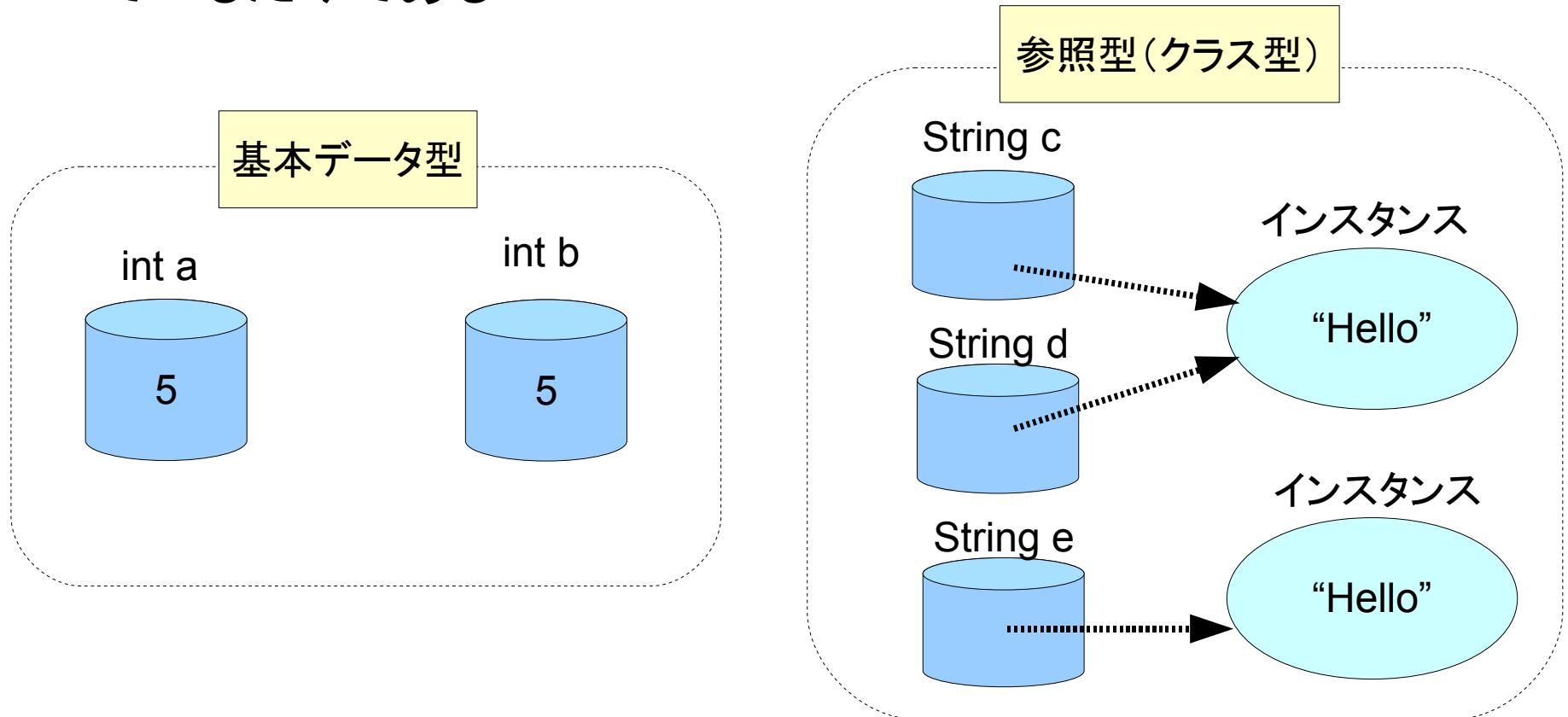
public class CalcTest {

    public static void main(String[] args) {
        int dollar = 150;
        System.out.println(dollarToYen(dollar));
    }
}
```

「基本データ型」と「参照型」の違い

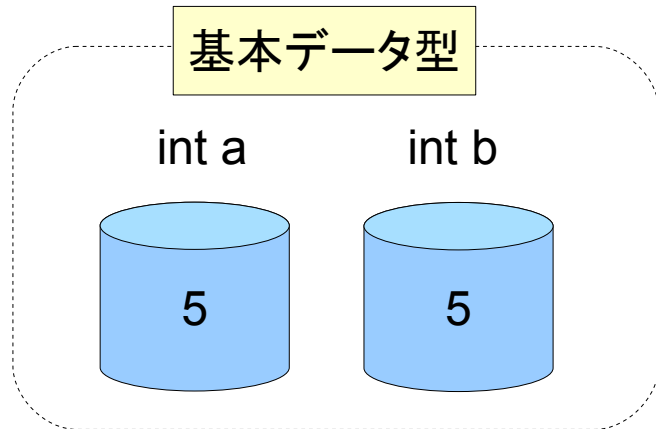
- 変数の格納の概念の違い

- 基本データ型の変数は、それぞれが独立した格納場所を持っているが、参照型(クラス型)の変数は、インスタンスの所在を示しているだけである

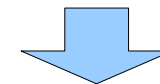


「基本データ型」と「参照型」の違い

- 比較演算子(等号)の意味の違い
 - 基本データ型の比較演算(==演算子)
 - それぞれの値の内容を比較



(a == b)の演算結果



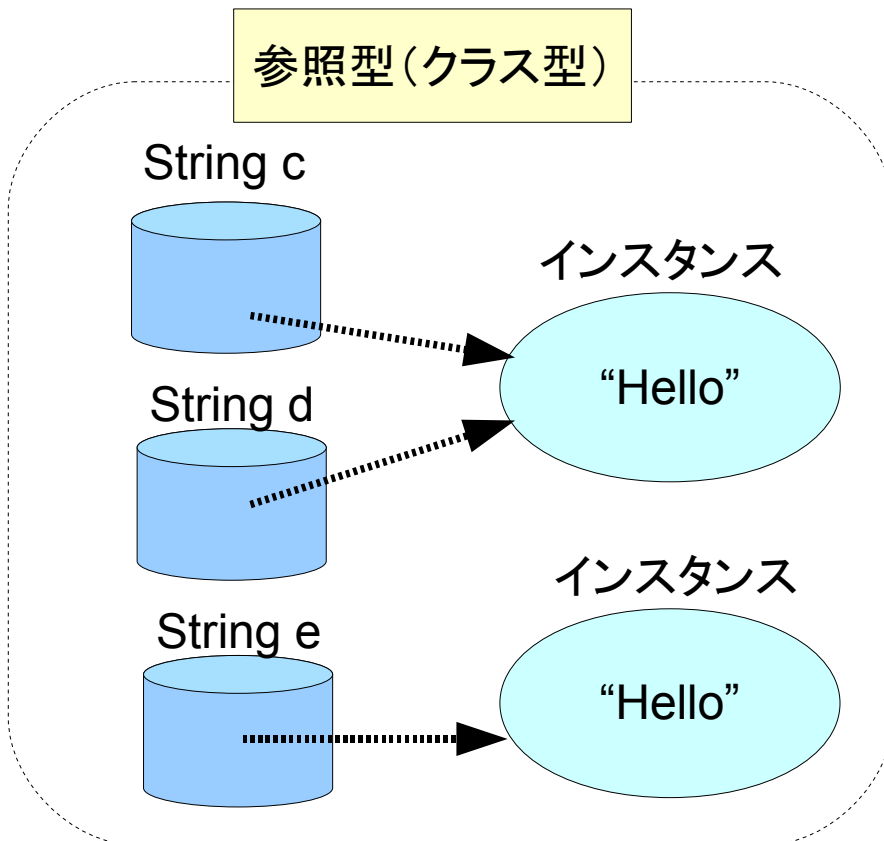
true

「基本データ型」と「参照型」の違い

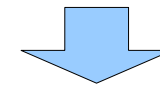
- 比較演算子(等号)の意味の違い

- 参照型の比較演算(==演算子)

- それぞれの示すインスタンスが同一かどうかを比較

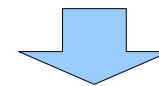


(c == d)の演算結果



true

(c == e)の演算結果

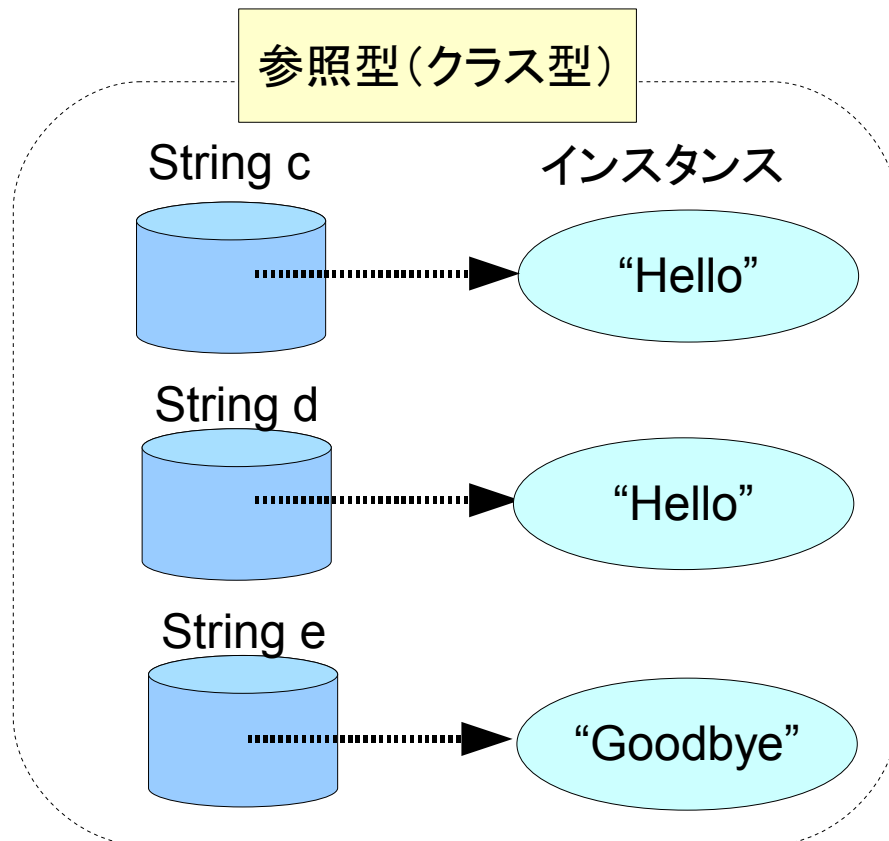


false

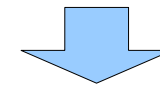
「基本データ型」と「参照型」の違い

- 参照型の比較演算

- 参照型の値が等しいかどうか比較するにはequalsメソッドを使う

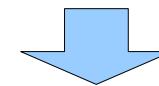


(c.equals(d))の演算結果



true

(c.equals(e))の演算結果

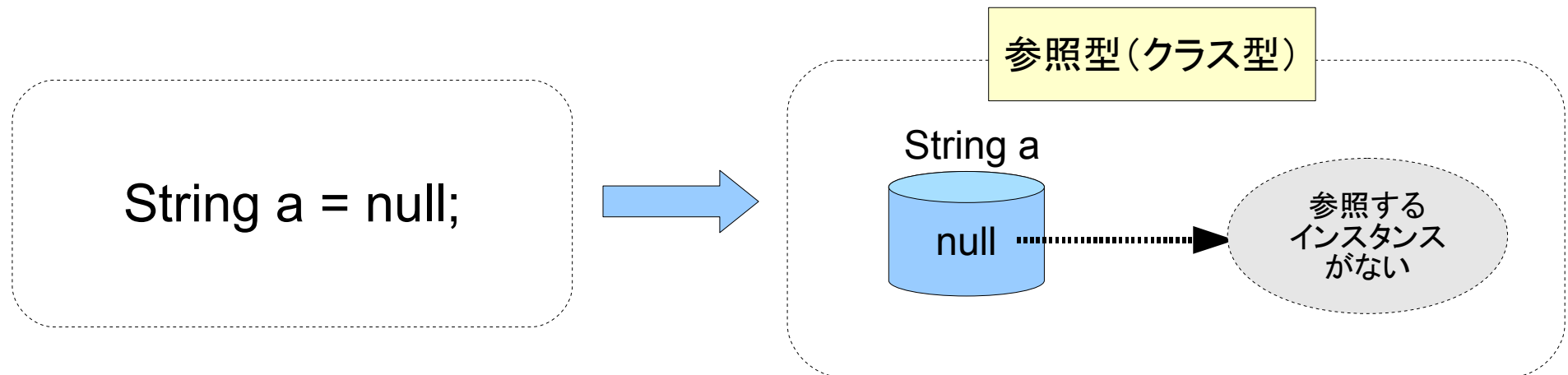


false

「基本データ型」と「参照型」の違い

• 「null」の存在

- 参照型の変数には、所在を示すインスタンスがないことを表す「null」という特別な値を代入することができる
- nullが代入されている変数に対してメソッドやフィールドにアクセスすることはできない



※ ある変数がnullかどうかを判定する場合は「==」演算子を使います
例) if (a == null) System.out.println("aはnullです");

オブジェクト指向でJavaを活用する

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Agenda

- オブジェクト指向でJavaを活用する
 - オブジェクト指向とは何か
 - 継承
 - オーバーライド
 - 抽象クラス
 - インターフェース
 - カプセル化
 - アクセス制限の修飾子
 - ポリモルフィズム
 - オーバーロード

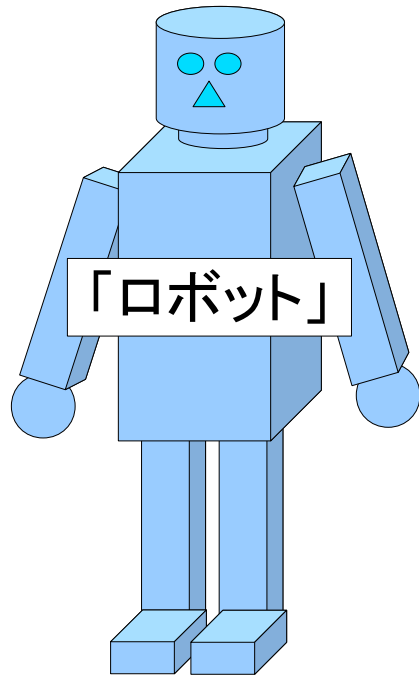
オブジェクト指向でJavaを活用する ～オブジェクト指向とは何か～

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

オブジェクト指向とは何か

- 「オブジェクト(Object)」とは？

- あらゆる事物(概念的なものか、実体を伴うものかは問わない)
- 世の中に存在する事物はオブジェクトとみなすことができる



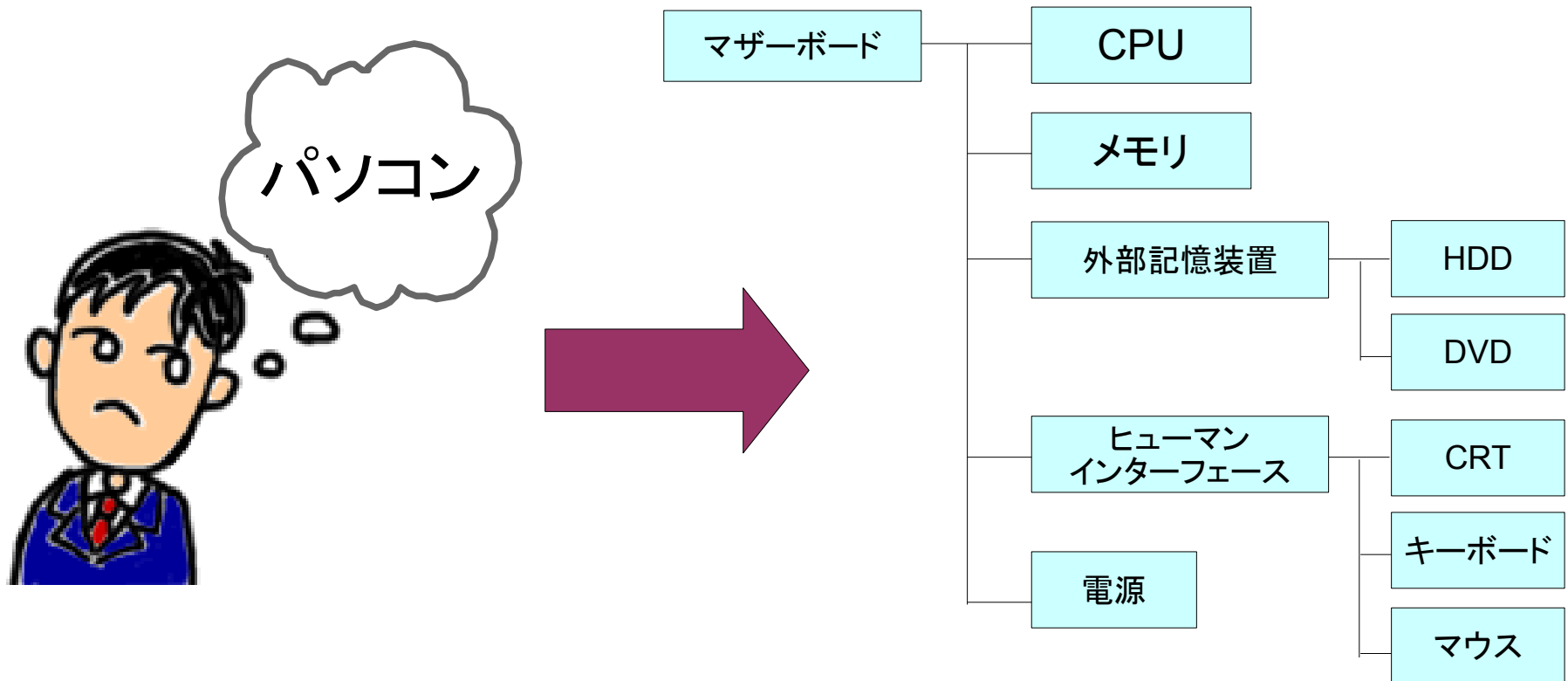
物理的な実体を伴うオブジェクト

会社

概念的なオブジェクト

オブジェクト指向とは何か

- 「オブジェクト指向(Object-Oriented)」とは何か
 - システム化する対象を、なるべくそのままの構造(概念や実体を活かした)で表現しようとするプログラミング手法



オブジェクト指向に登場する概念

- クラス
 - 状態(状況)
 - 振る舞い(動作)
- インスタンス

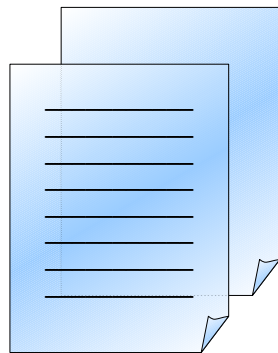
オブジェクト指向に登場する概念

- クラス

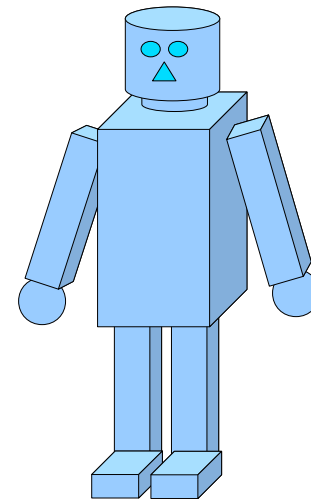
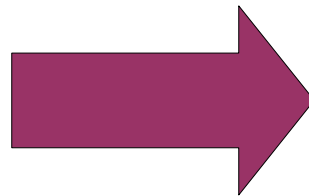
- 表現したい事物について定義したもの
- 定義の内容は、大きく二つに分類される
 - 状態(状況) … フィールド
 - 振る舞い(動作) … メソッド

オブジェクト指向に登場する概念

- クラスの「実体」としてのインスタンス
 - クラスそのものは「定義」でしかなく、実体がない
 - クラスはよく「設計書」のようなものと例えられる
 - 設計書をもとにした実体が「インスタンス」



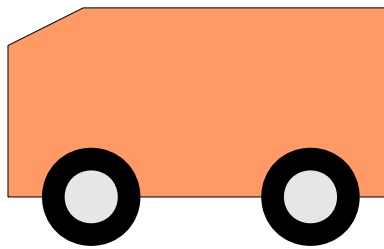
ロボットの設計書
（「ロボット」クラス）



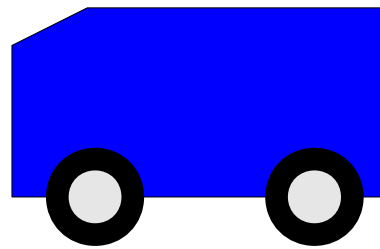
ロボットの实体
（「ロボット」インスタンス）

オブジェクト指向に登場する概念

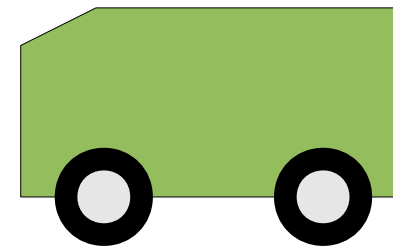
- なぜ「インスタンス」が必要なのか
 - クラスは「それぞれの事物を定義したもの」であるが、個体そのものではない
 - 「人間」とひとことで言っても、いろいろな人(個体)がいる
 - 「車」とひとことで言っても、いろいろな車(個体)がある



オレンジ色の車



青い車



緑色の車

オブジェクト指向に登場する概念

- 「クラス」が「インスタンス化」すると何が起きるか
 - クラスには「状態」「振る舞い」が定義されている
 - つまり、それぞれのインスタンスが、固有の「状態」を持つことができる
 - お金をたくさん持っている人、あまり持っていない人
 - 乗車定員が2名の車、定員が5名の車
 - 「状態」が異なると「振る舞い」も異なってくる
 - お金のたくさんある人の買い物と、そうでない人の買い物
 - スポーツカーとファミリーカーの走り方

オブジェクト指向に登場する概念

- クラス間の関連

- クラスは単独で動作するのではなく、他のクラスと協調して動作することができる
 - 「人間」が「道具」を使って行動する
 - 「車」は「エンジン」を使って走る
- クラス間の関連
 - あるクラスから、他のクラスのメソッドを実行
 - あるクラスが、他のクラスのフィールドを参照(更新)



「人間」が「パソコン」
を使って行動する



オブジェクト指向プログラミング

- これらの概念を駆使することによって、対象となるものの構造を実際のままだに表現することができる
- 実行するコンピュータにとっては、効率的な表現ではないが、設計する人間にとっては理解しやすいモデルとなる

オブジェクト指向でJavaを活用する ～継承～

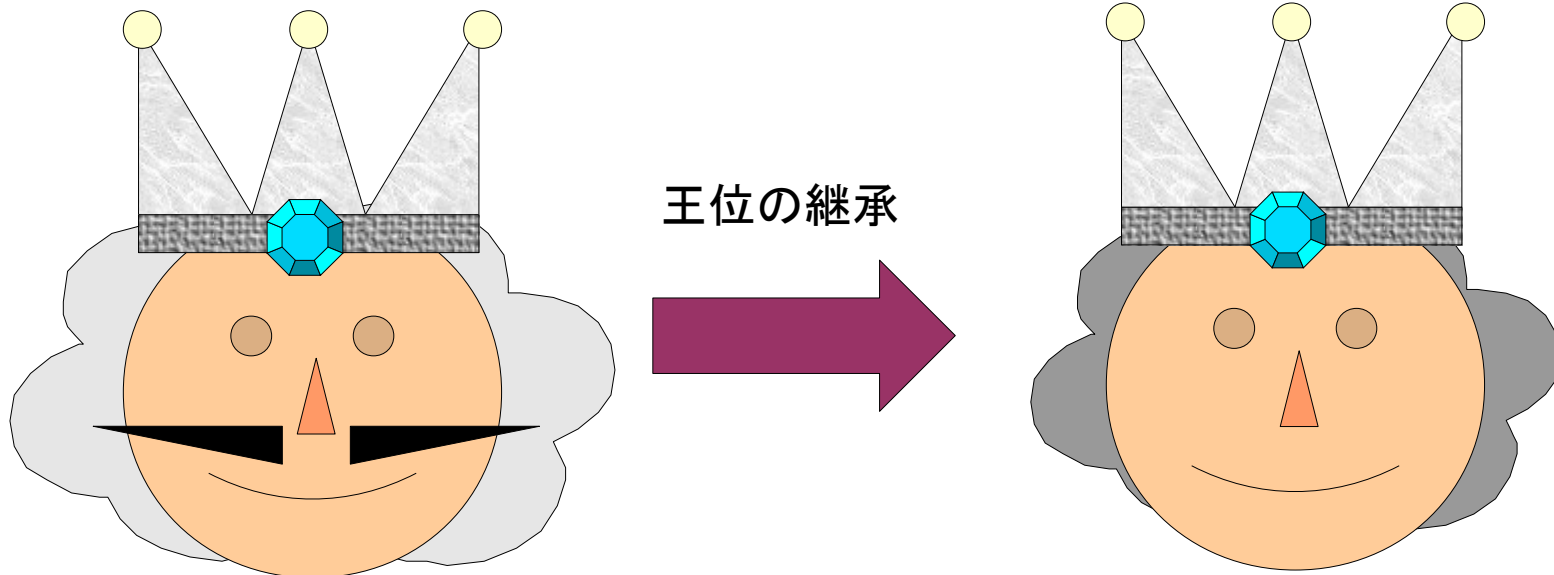
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

継承とは何か

- 継承【けいしょう】

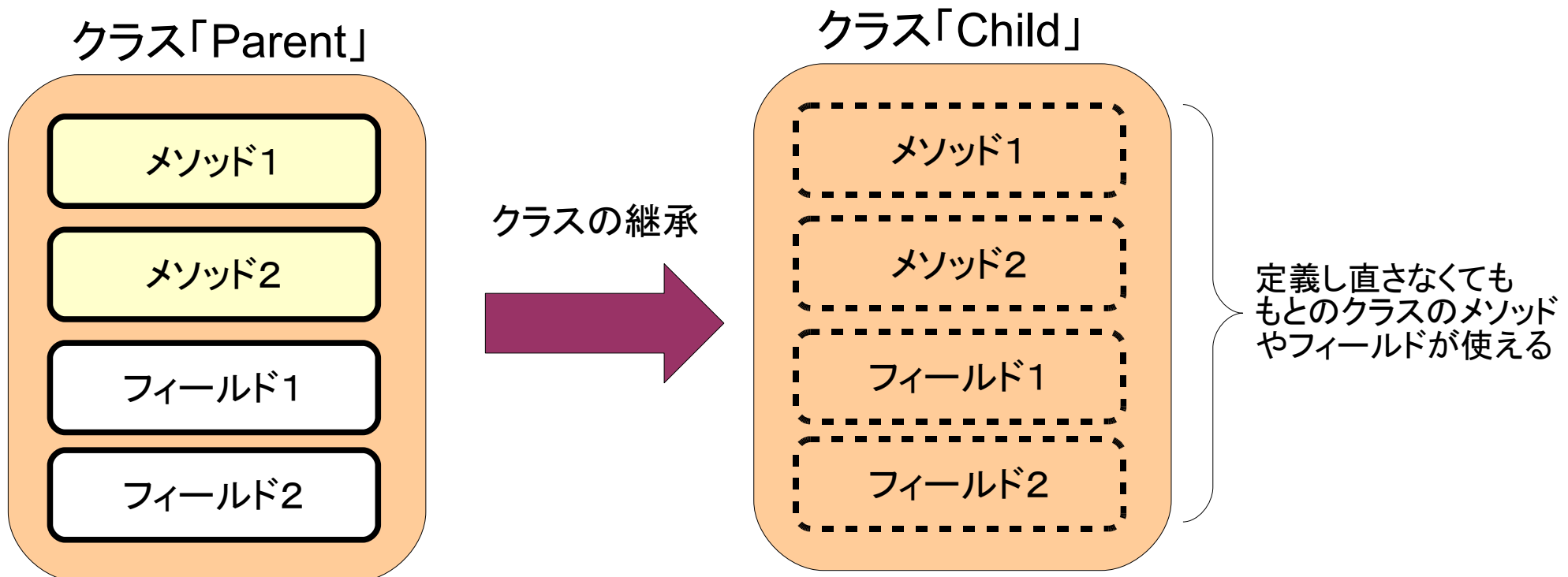
- 先の人の子分・権利・義務・財産などを受け継ぐこと。
(三省堂提供「デイリー 新語辞典」より)

- 「王位を継承する」
 - 「師匠の技を継承する」
 - 「買収先会社の人員や資産を継承する」



継承とは何か

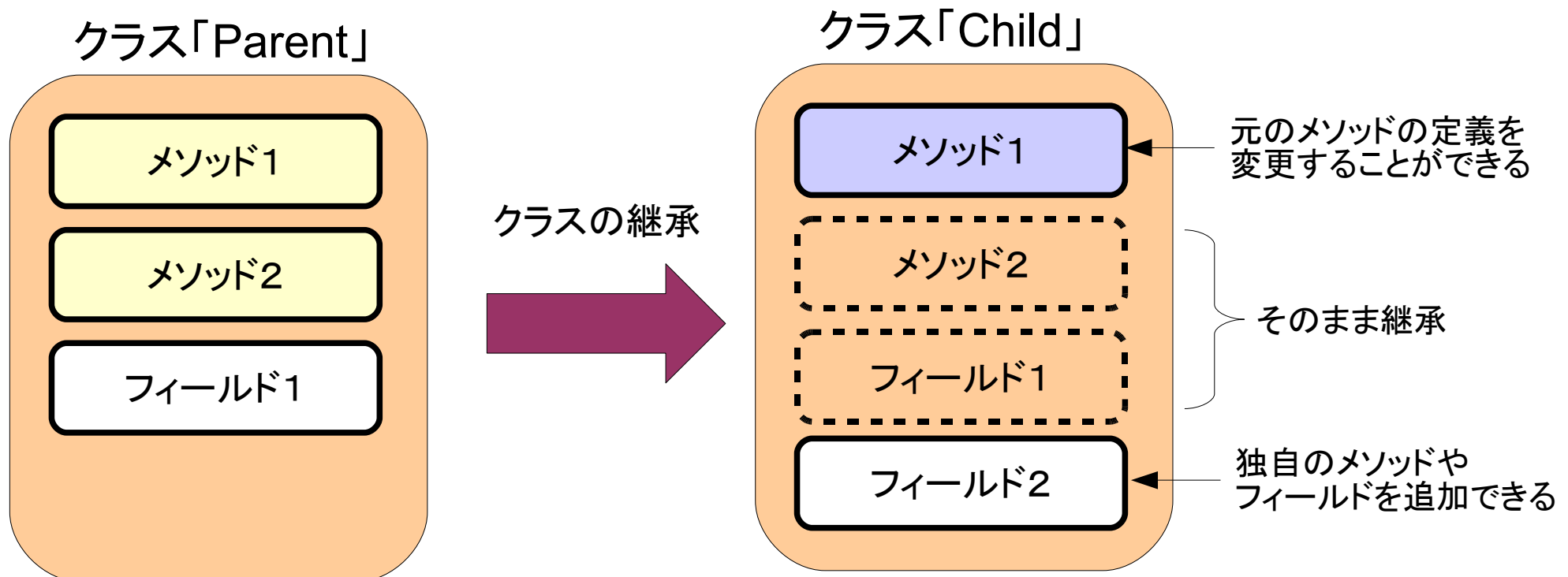
- Javaにおける継承
 - あるクラスの定義内容を受け継ぐこと
 - メソッド・フィールドを定義し直さなくてもそのまま使える



継承とは何か

• Javaにおける継承

- 継承したクラスで独自のメソッドやフィールドを追加することもできる
(=拡張)
- 継承したクラスで元のメソッド・フィールドの定義を変更することもできる
(=再定義・オーバーライド)



継承とは何か

- 継承関係の用語
 - 元のクラス(継承されるクラス)
 - 親クラス
 - スーパークラス
 - 継承元
 - 元のクラスを継承したクラス
 - 子クラス
 - サブクラス
 - 派生クラス
 - 継承クラス
 - 継承先

※「継承」のことを「派生」と呼ぶこともあります

Javaでの継承の表現

- 継承

- extendsキーワードによって継承を表現
- 子クラスでは、特に必要なければ親クラスのフィールド・メソッドを定義し直さなくて良い(何も書かなくて良い)
- クラス定義には親クラスはひとつしか指定できない(単一継承)


```
class 子クラス名 extends 親クラス名 {  
  
}
```

Javaでの継承の表現

- 継承のコード例

```
public class Animal {  
    public void eat(String food) {  
        System.out.println(food+"を食べます。");  
    }  
    public void sleep() {  
        System.out.println("眠ります");  
    }  
    public void wakeUp() {  
        System.out.println("起きました");  
    }  
}
```

```
public class Bird extends Animal {  
  
}
```



上記のAnimalクラスとBirdクラスを作成し、さらに別のクラスのmainメソッドでBirdクラスに対してsleepメソッド、wakeUpメソッドを呼び出してみましょう。

Javaでの継承の表現

- 継承（＋拡張）
 - 親クラスにないフィールド・メソッドを追加したい場合は、子クラスのクラス定義にそれらを記述することができる

```
class 子クラス名 extends 親クラス名 {  
    子クラス独自のフィールド定義  
    子クラス独自のメソッド定義  
}
```

Javaでの継承の表現

- 継承(＋拡張)のコード例

```
public class Bird extends Animal {  
    public static int wing = 2;  
  
    public void fly() {  
        System.out.println("飛びます");  
    }  
}
```

子クラスで拡張された
メソッド・フィールド

Javaでの継承の表現

- 継承（＋オーバーライド）
 - 親クラスのフィールド・メソッドの内容を子クラスで変更したい（オーバーライド）場合は、子クラスでそれらを定義し直すことができる
 - 変更したいフィールド・メソッドだけを再定義すればよい
 - そのままでよいフィールド・メソッドは何も記述しなくてよい
 - 拡張とオーバーライドは併用してもよい

```
class 子クラス名 extends 親クラス名 {  
    親クラスのフィールド再定義  
    親クラスのメソッド再定義  
}
```

Javaでの継承の表現

- superとthis
 - オーバーライドを行う際、親クラスの定義を利用したい場合がよくある
 - オーバーライドは、親クラスの状態や振る舞いを基本に、一部を追加したり変更したりする場合が多いため
 - オーバーライドした子クラスのメソッドの中から、親クラスのメソッドやフィールドにアクセスしたい場合
 - superキーワードを利用する

super. 親クラスのフィールド名
super. 親クラスのメソッド名(引数リスト)

Javaでの継承の表現

- thisキーワード

- superが親クラスを示すのに対し、thisは自分自身のクラスを示す
 - 自分自身のメソッドや、自分自身のフィールドを指定する場合は、thisを使用する
 - しかし、簡便のためにthisは省略して良いことになっているため、メソッド呼び出しではほとんど使われることがない
 - ローカル変数と、フィールドを区別するために、thisをつけてフィールドを示す方法はよく用いられる


```
this. 自クラスのフィールド名  
this. 自クラスのメソッド名(引数リスト)
```

Javaでの継承の表現

- 継承(＋オーバーライド)のコード例

```
public class Bird extends Animal {  
  
    public static int wing = 2;  
  
    public void fly() {  
        System.out.println("飛びます");  
    }  
  
    public void sleep() {  
        System.out.println("木の上で眠ります");  
    }  
  
}
```

子クラスで
オーバーライド
された部分




上記に従いBirdクラスでsleepメソッドをオーバーライドし、再び別のクラスのmainメソッドでBirdクラスに対してsleepメソッドを呼び出してみましょう。

Javaでの継承の表現

- superキーワードを使ったオーバーライドの例

```
public class Bird extends Animal {  
  
    public static int wing = 2;  
  
    public void fly() {  
        System.out.println("飛びます");  
    }  
  
    public void sleep() {  
        System.out.print("木の上で");  
        super.sleep();  
    }  
  
}
```




上記に従いBirdクラスのsleepメソッドを修正し、どのような結果が得られるか確認してみましょう。

Javaでの継承の表現

- thisキーワードを使った例

```
public class Bird extends Animal {  
  
    public static int wing = 2;  
  
    public void fly() {  
        int wing = 3;  
        System.out.println(this.wing+“枚の羽根で飛びます”);  
    }  
  
    public void sleep() {  
        System.out.print(“木の上で”);  
        super.sleep();  
    }  
  
}
```



上記に従いBirdクラスのflyメソッドを修正し、どのような結果が得られるか確認してみましょう。

暗黙の親クラス

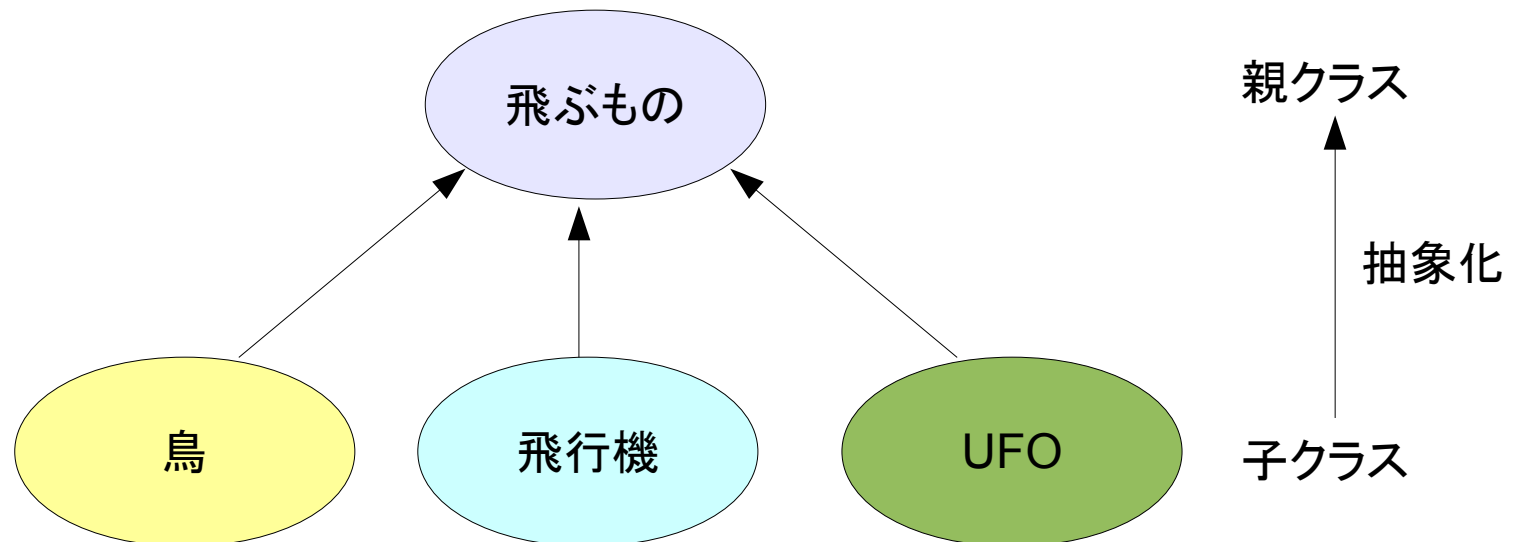
- Javaでは、クラス定義に明示していなくとも、全てのクラスは「java.lang.Object」というクラスを継承していることになっている＝暗黙の親クラス
- java.lang.Objectクラスにも、メソッドやフィールドが定義しており、全てのクラスから利用が可能

継承の活用

- どのような場合に継承を使うべきか
 - クラス間の関連を考慮する
 - 通常、子クラスは親クラスを特殊化したものであるべき
 - 「車」と「消防車」、「ほ乳類」と「ヒト」
 - 継承を用いると、少ないコードで特殊化したクラスを作成でき、同じコードを何度も書き直す必要がない
 - 機能が似ていると言うだけで継承を使わないようにする
 - 「馬」と「車」…走るという機能は似ているが、継承はふさわしくない

抽象クラス

- 抽象クラスとは
 - 具体的な実体を持たないが、親クラスとなりうるもの
 - すでにあるいくつかのクラスを「抽象化」してできるもの



抽象クラス

- 抽象クラスの特徴
 - 抽象クラス自身はインスタンス化できない
 - 子クラスにオーバーライドを強制する「抽象メソッド」を定義することができる
 - 抽象メソッドでない通常のメソッドやフィールドも定義可能

抽象クラス

- 抽象クラスの定義

```
abstract class クラス名 {  
  
}
```

- 抽象メソッドの定義

```
abstract 戻り値 メソッド名(引数リスト);
```

オブジェクト指向でJavaを活用する ～インターフェース～

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

インターフェースとは何か

- インターフェース【interface】
 - コンピューター本体と各種周辺装置やコンピューターどうしを接続し、電気信号の大きさを調整したり、データの形式を変換したりして、両者間のデータのやりとりを仲介する回路や装置。
 - また、人間がコンピューターなどの装置を円滑に使用できるようにするための操作手順。
(三省堂提供「デイリー 新語辞典」より)

Javaにおける「インターフェース」

- インターフェース
 - クラスに対する入出力を規定するもの
 - 入出力＝メソッドの名前、引数、戻り値のこと
 - 工業製品の「規格」に近い概念
 - 工業製品の例
 - 「USB」という規格に従う製品であれば、(性能は違うかもしれないが)どのメーカーの製品でも同じように使うことができる
 - Javaのインターフェースの例
 - 「ABC」というインターフェースに従うクラスであれば、(実際の処理の詳細は違うかもしれないが)誰が作ったクラスでも同じように使うことができる

Javaにおける「インターフェース」

- インターフェース利用の例 (JDBC API)

もしデータベースにアクセスするメソッドの使い方が、データベースごとにバラバラだとしたら・・・
(例: SELECT文の実行)

クライアント
クラス

PostgreSQL用アクセスクラス

`select("SELECT * FROM EMP")`

PostgreSQL

MySQL用アクセスクラス

`fetchRecord("", "EMP")`

MySQL

`query(SELECT, "* FROM EMP")`

Firebird用アクセスクラス

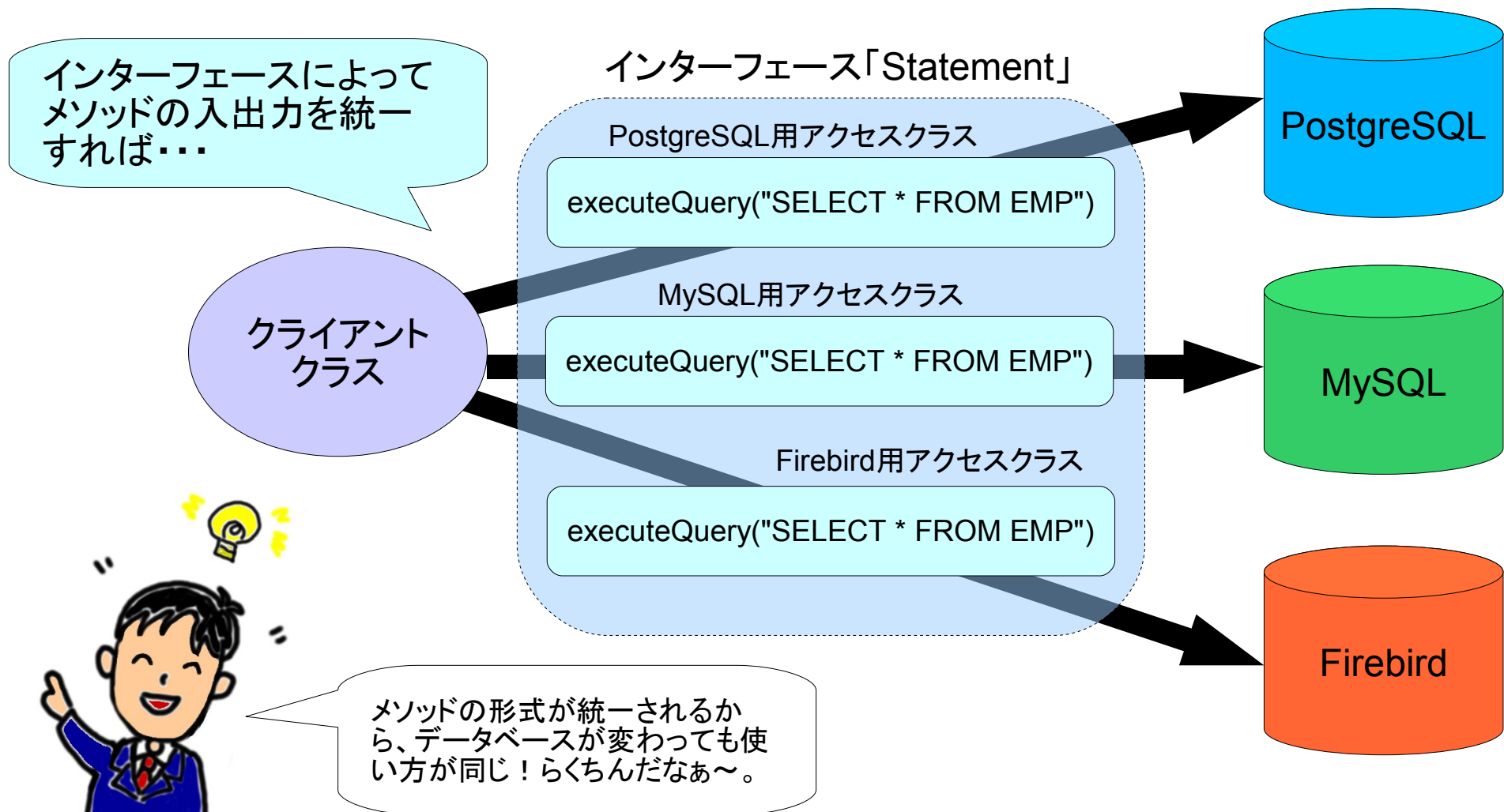
Firebird



やりたい操作はほとんど同じなのに、データベースごとにいちいちおぼえるのは面倒だなあ・・・

Javaにおける「インターフェース」

- インターフェース利用の例 (JDBC API)



Javaにおける「インターフェース」

- インターフェースの定義

```
interface インターフェース名 {  
  
}
```

Javaにおける「インターフェース」

- インターフェースに定義できるもの

- 定数

- インターフェースでは、変数定義には暗黙に「public static final」がつくとみなされるため定数扱いとなる

```
型名 変数名 = 初期値;
```

- 抽象メソッド

- インターフェースでは、メソッド定義には暗黙に「public abstract」がつくとみなされるため抽象メソッド扱いとなる

```
型名 メソッド名(引数リスト);
```

インターフェースの利用

- インターフェースはクラスと組み合わせて利用する
 - ①あるクラスがインターフェースに従うことを宣言
 - クラス定義にimplements宣言を追加しインターフェースを指定
 - インタフェース名は1クラスに対し複数を同時に定義することも可能
 - あるインターフェースに従うクラスを「実装クラス」と呼ぶ

```
class クラス名 implements インターフェース名 {  
  
}
```

インターフェースの利用

- インターフェースはクラスと組み合わせて使用する
 - ②そのクラスはインターフェースに定義された抽象メソッドを全て必ず実装しなければならない
 - クラスがインターフェースに定義されたメソッドを自分自身に定義することを「実装する」と呼ぶ
 - インターフェースに定義されていないメソッドを独自に追加してもよい
 - インターフェースの定数はそのまま好きなときに利用することができる

```
class クラス名 implements インターフェース名 {  
  
    public 戻り値 メソッド名(引数リスト) {  
        (処理内容を定義)  
    }  
  
}
```


インターフェースの利用

- クラスがインターフェースに従うことの効果
 - あるインターフェースに従う全てのクラスでインターフェースに定義されたメソッドが必ず実行できることが保証される
 - 継承関係にないクラスでも、共通の動作をさせることができる
 - あるインターフェースに従う全てのクラスのインスタンスは、自分の型名を「インターフェース名」とみなすことができる
 - 実際のクラス名を意識せずにインターフェース名だけでそのクラスを扱うことができる
 - 型名が「インターフェース名」の場合は、インターフェースに定義されたメソッド以外は利用できない(クラス独自のメソッドは利用できない)

インターフェースの例

- インターフェースのコード例

```
interface Robot {  
    int speed = 100;  
    void sayHello();  
    void walk();  
}
```

インターフェースの例

- 実装クラスの例

```
public class RobotImpl implements Robot {  
    String name = "Mike";  
  
    public void sayHello() {  
        System.out.println("Hello! My Name is "+name);  
    }  
  
    public void walk() {  
        System.out.println("I can walk very well.");  
    }  
  
}
```

インターフェースの例

- インターフェースを型とみなした例①
 - クラスのインスタンスをインターフェース型として定義

```
Robot rbt = new RobotImpl();  
rbt.walk();
```

インターフェースの例

- インターフェースを型とみなした例②
 - インターフェースを引数や戻り値の型に定義した例

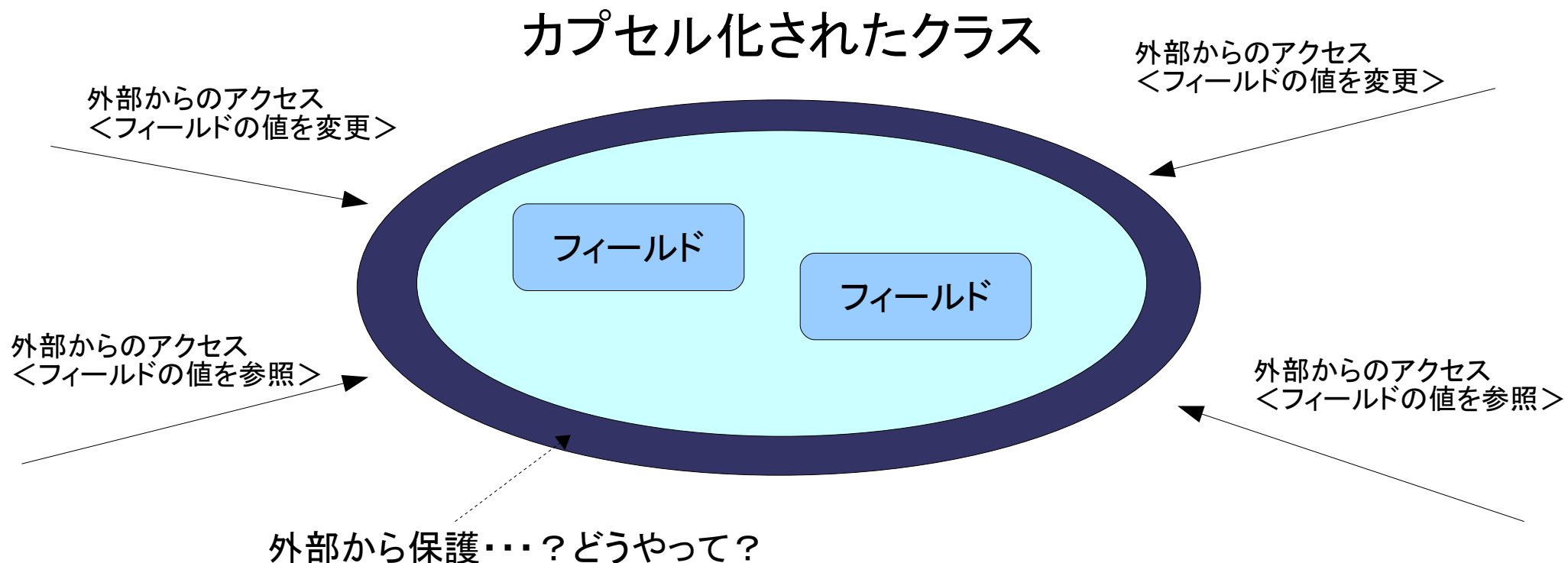
```
public Robot getRobot() {  
    return new RobotImpl();  
}  
  
public void robotSayHello(Robot r) {  
    r.sayHello();  
}
```

オブジェクト指向でJavaを活用する ～カプセル化～

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

カプセル化とは何か

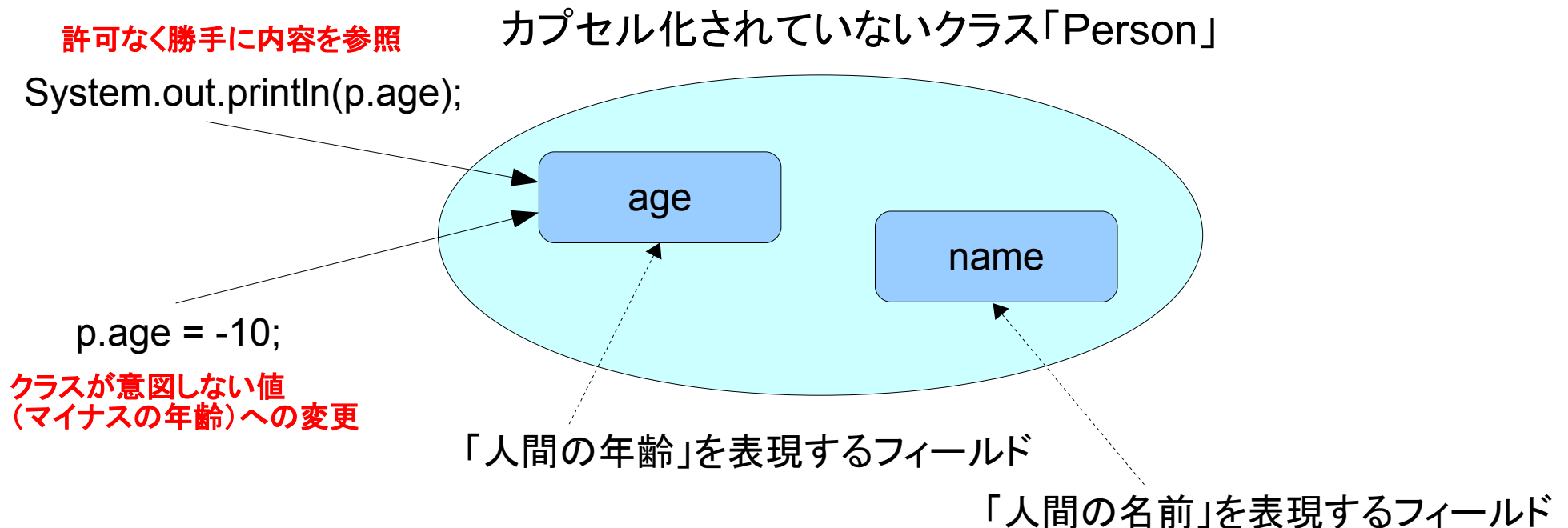
- クラスが持つフィールドの内容を、外部から保護するための仕組み



カプセル化とは何か

- カプセル化が必要な理由

- フィールドの内容を、クラス自身が意図しない内容に変更されてしまったり、勝手に内容を参照されてしまうことを防ぎたい



カプセル化を実現するために

- カプセル化を実現するために①
 - フィールドのアクセス制限
 - 外部からのフィールドへのアクセスを文法的に制限するために、Javaでは「アクセス制御の修飾子」が用意されている
 - カプセル化を実現するには、アクセス制限の修飾子を使って、フィールドを外部からアクセスできないようにする

アクセス制限の修飾子

- アクセス制限の修飾子
 - クラス・メソッド・フィールドの先頭に記述し、外部からのアクセスを制限できる

修飾子	アクセスできる範囲
public	どのクラスからでもアクセス可能
protected	同じパッケージに属するクラスか、自分の子クラスからのみアクセス可能
private	自分自身のみがアクセス可能で、外部からはアクセス不可能
無指定	同一パッケージに属するクラスからのみアクセス可能

カプセル化を実現するために

• カプセル化を実現するために②

– フィールドを操作するためのメソッドを用意する

- フィールドに外部からアクセスできないようにすると、クラスの意図に沿った内容の変更や参照もできなくなってしまうため、代わりにフィールドを操作するためのメソッドを用意する
- メソッド内で変更内容をチェックするコードや、参照する値の内容を加工するコードを書いておけば、クラスにとって意図しない値への変更や参照を防ぐことができる

– 例)

- ageフィールドを変更するメソッドとして、setAgeメソッドを用意
- ageは年齢なので、setAgeメソッドの引数が負数だったら、フィールドへの代入を行わないようにし、意図しない値への変更を防ぐ

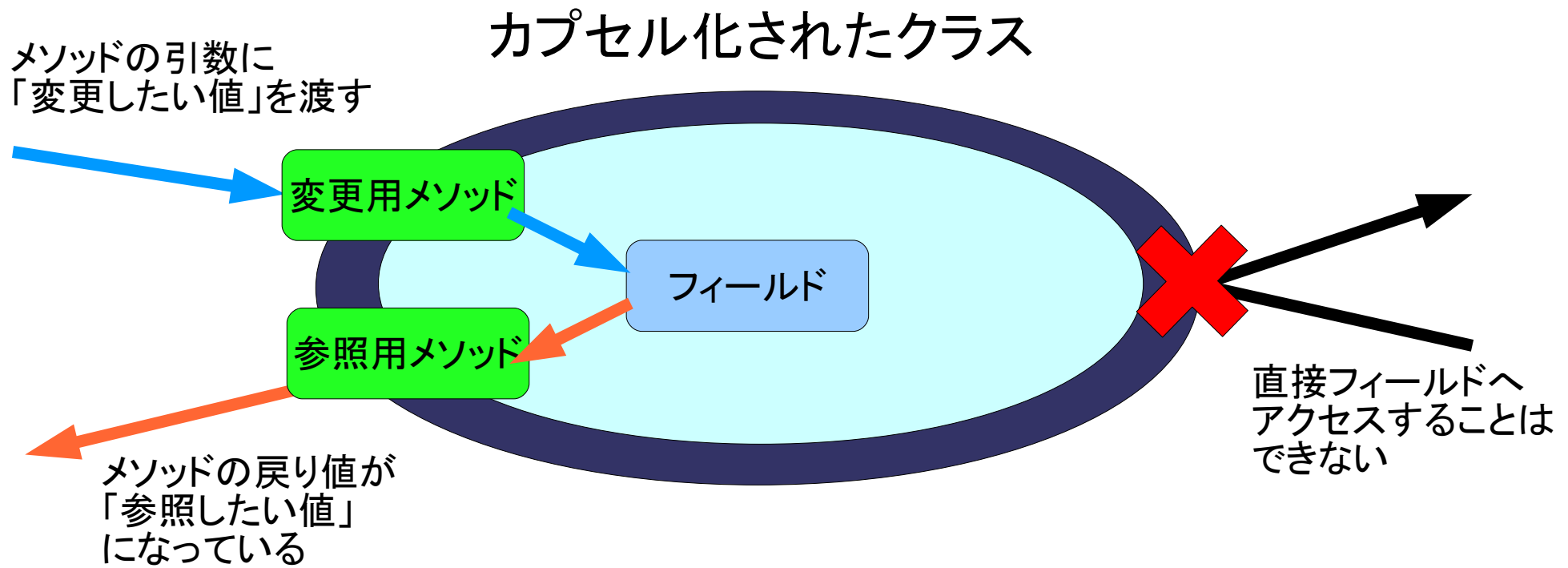
カプセル化の例

- カプセル化されたクラスの例

```
public class Person {  
    private String name;  
    private int age;  
  
    public void setAge(int newAge) {  
        if (newAge>=0) {  
            age = newAge;  
        }  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setName(String newName) {  
        if (newName!=null)&&(!newName.equals("")) {  
            name = newName;  
        }  
    }  
    public String getName() {  
        return name;  
    }  
}
```

カプセル化の完成

- フィールドへのアクセス制限と、メソッドによる変更・参照内容の監視によって、カプセル化が完成する



カプセル化のメリット

- カプセル化のメリット
 - クラスが想定していない状態を引き起こすことを防げる
 - フィールドの名称や存在が変更されても、外部には影響を与えない

オブジェクト指向でJavaを活用する ～ポリモルフィズム～

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

ポリモルフィズムとは

- ポリモルフィズム【polymorphism】
 - 【化】(結晶構造の)多形, 同質異像; 【生物】多形性.
(三省堂提供「EXCEED 英和辞典」より)
- オブジェクト指向におけるポリモルフィズム
 - (1)異なるオブジェクトが、あたかも同じオブジェクトであるかのように振る舞うこと
 - (2)同じクラス内の異なるメソッドが、あたかも同じメソッドであるかのように振る舞うこと

ポリモルフィズム(1)

- 「異なるオブジェクトが、あたかも同じオブジェクトであるかのように振る舞う」とは？
 - ①あるクラスAが他のクラスBを継承している場合、クラスAは自分の型がBであるかのように振る舞うことができる
 - ②あるクラスCが他のインターフェースDを実装している場合、クラスCは自分の型がDであるかのように振る舞うことができる

ポリモルフィズム(1)

- 「①あるクラスAが他のクラスBを継承している場合、クラスAは自分の型がBであるかのように振る舞うことができる」の例

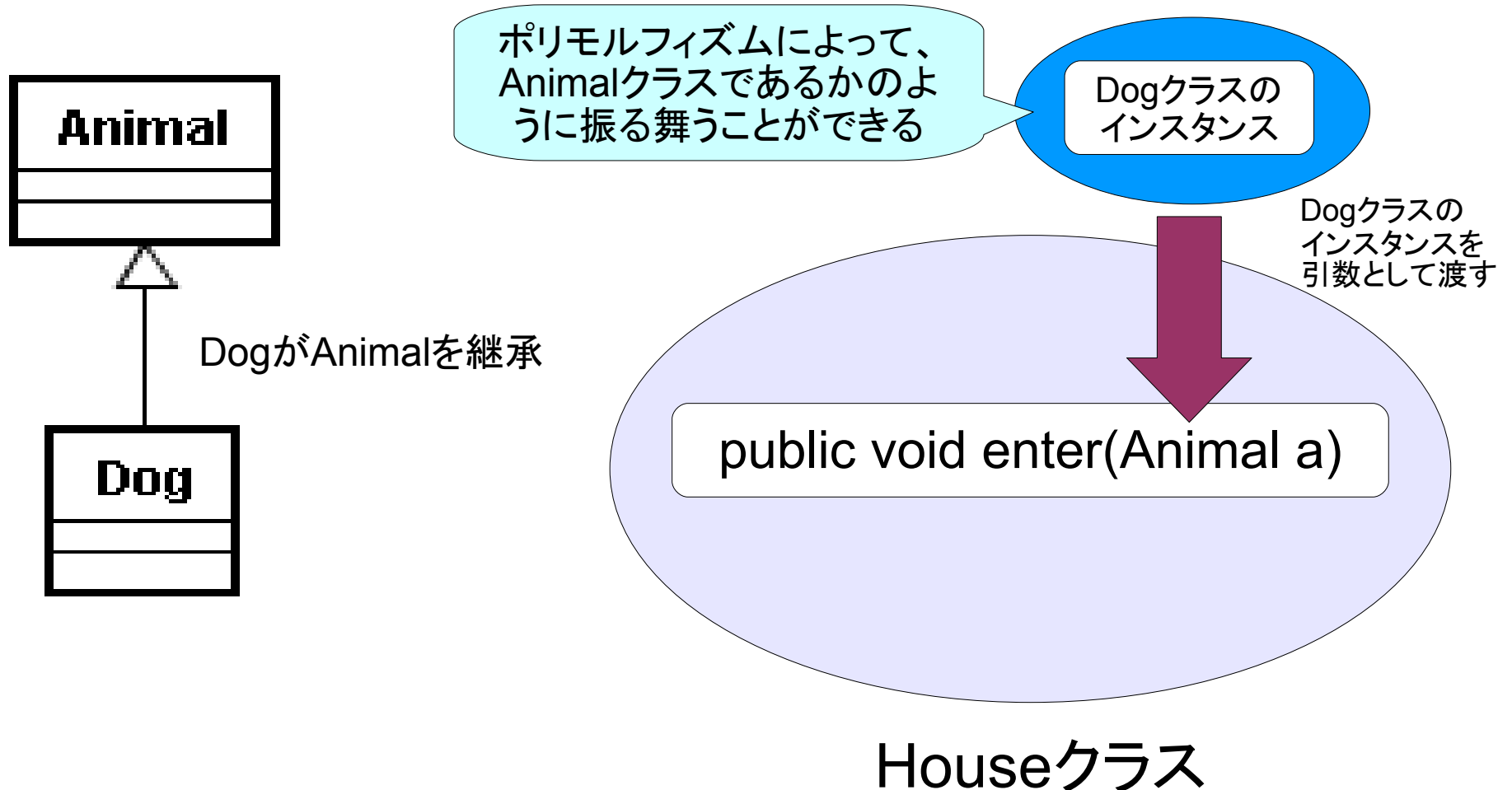
```
public class Dog extends Animal {  
    . . .  
}
```

```
public class House {  
    public void enter(Animal a) {  
        . . .  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        House dogHouse = new House();  
        dogHouse.enter(d);  
    }  
}
```

ポリモルフィズム(1)

- 「①あるクラスAが他のクラスBを継承している場合、クラスAは自分の型がBであるかのように振る舞うことができる」の例



ポリモルフィズム(1)

- 「②あるクラスCが他のインターフェースDを実装している場合、クラスCは自分の型がDであるかのように振る舞うことができる」の例

```
public interface FlyingObject {  
    void fly();  
}
```

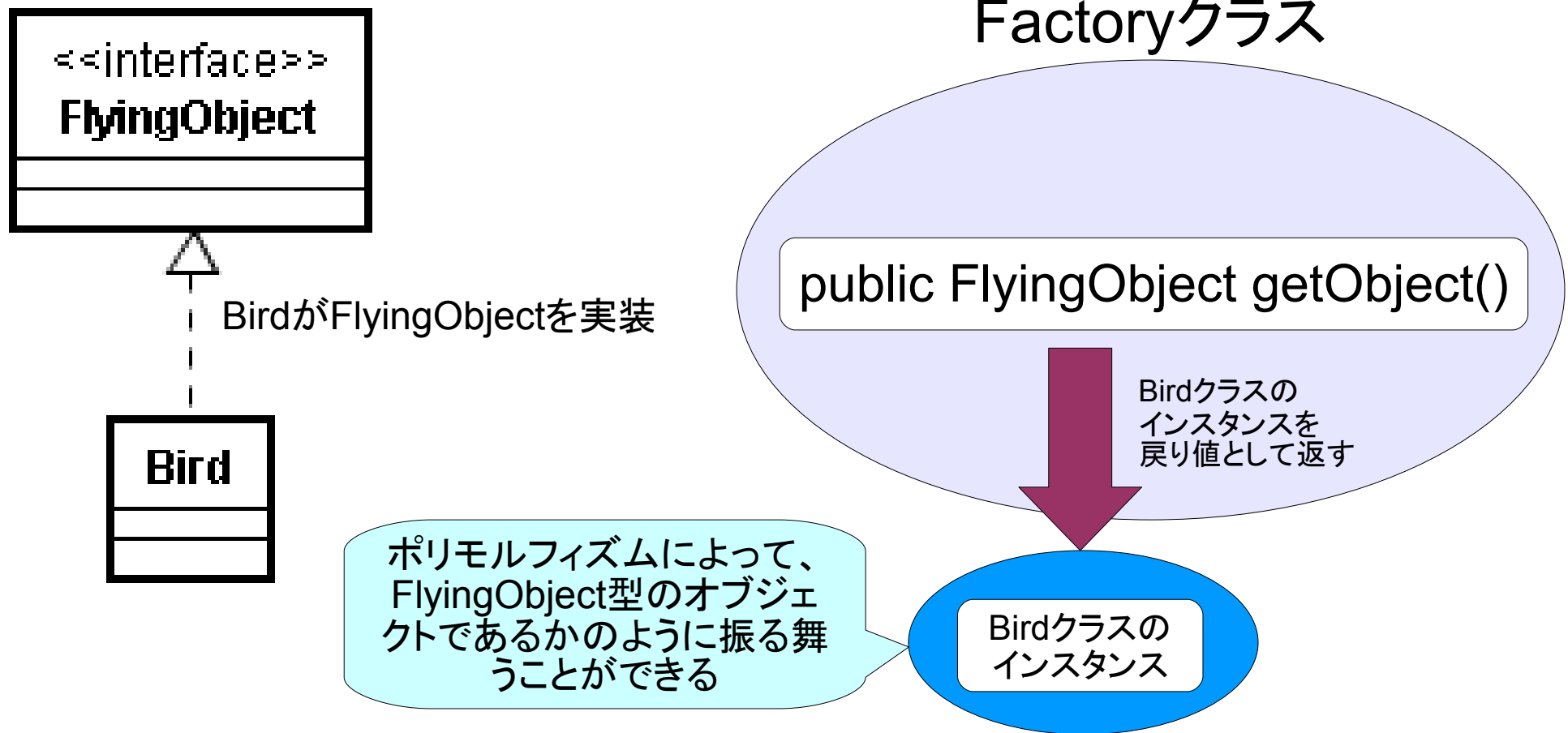
```
public class Bird implements FlyingObject {  
    public void fly() {  
        System.out.println("I'm Flying.");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Factory f = new Factory();  
        FlyingObject obj = f.getObject();  
        obj.fly();  
    }  
}
```

```
public class Factory {  
    public FlyingObject getObject() {  
        Bird b = new Bird();  
        return b;  
    }  
}
```

ポリモルフィズム(1)

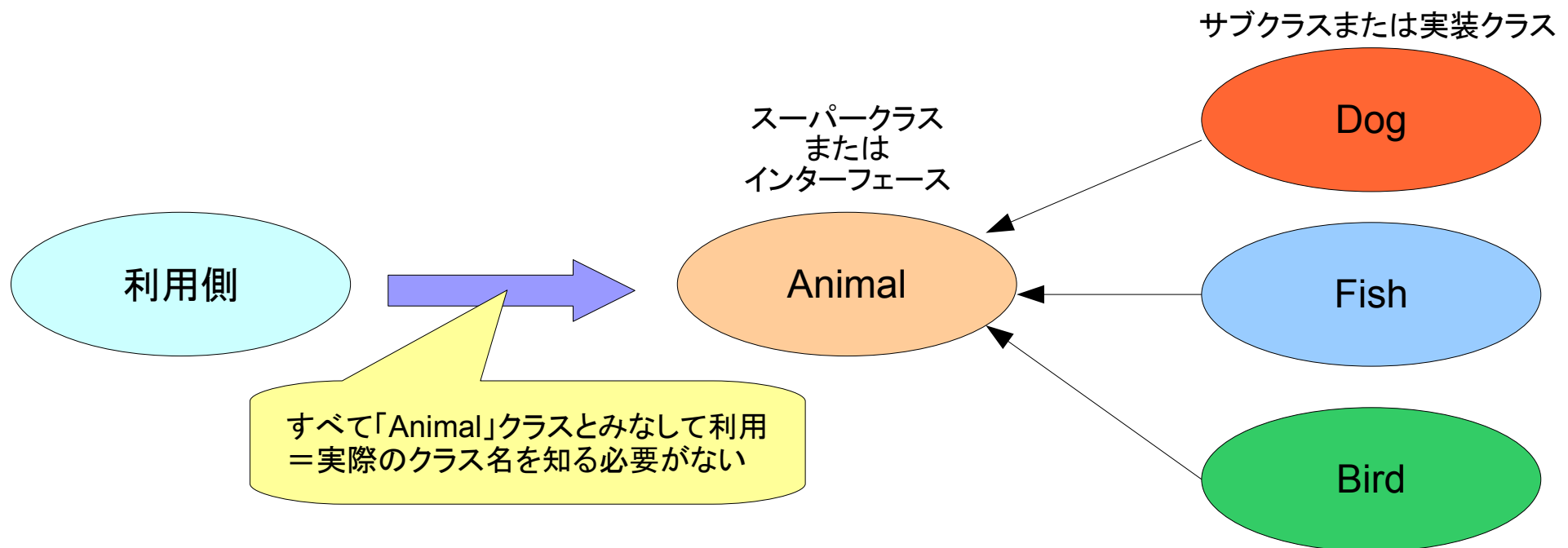
- 「②あるクラスCが他のインターフェースDを実装している場合、クラスCは自分の型がDであるかのように振る舞うことができる」の例



ポリモルフィズム(1)

• ポリモルフィズムのメリット(1)

- 継承やインターフェースにポリモルフィズムを適用することで、実際のクラス名が何であるかを気にせずにそのオブジェクトを共通の名称で利用することができるようになる
- 戻り値や引数の異なるメソッドを多数用意せずに済む



ポリモルフィズム(1)

- ポリモルフィズムとキャスト

- ポリモルフィズムによってスーパークラスやインターフェースを型としているオブジェクトを、本来の型に戻して使いたい場合は、キャスト演算子を使う

```
public class Factory {  
  
    public FlyingObject getObject() {  
        Bird b = new Bird();  
        return b;  
    }  
  
}
```

```
public class Test {  
  
    public static void main(String[] args) {  
        Factory f = new Factory();  
        Bird b = (Bird)f.getObject();  
        b.fly();  
    }  
  
}
```

ポリモルフィズム(2)

- 「同じクラス内の異なるメソッドが、あたかも同じメソッドであるかのように振る舞う」とは？
 - 同じクラス内に、「戻り値とメソッド名が同一で、引数リストの異なるメソッド」を複数定義することができる
- ＝「オーバーロード」

ポリモルフィズム(2)

- オーバーロードの例

```
public class Tax {  
  
    public static final double rate = 0.05;  
  
    public static int calc(int price) {  
        return (int)(price * rate);  
    }  
  
    public static int calc(double price) {  
        return (int)((int)price * rate);  
    }  
  
    public static int calc(String price) {  
        return (int)(Integer.parseInt(price) * rate);  
    }  
  
}
```

ポリモルフィズム(2)

- オーバーロードの使い方
 - 「メソッド名が同じ」=「機能が同じ」と解釈されるので、
ほぼ同じ機能で引数が異なる場合に使うようにする
 - 全く異なる機能なのにオーバーロードを使うと誤用を引き起こしやすい

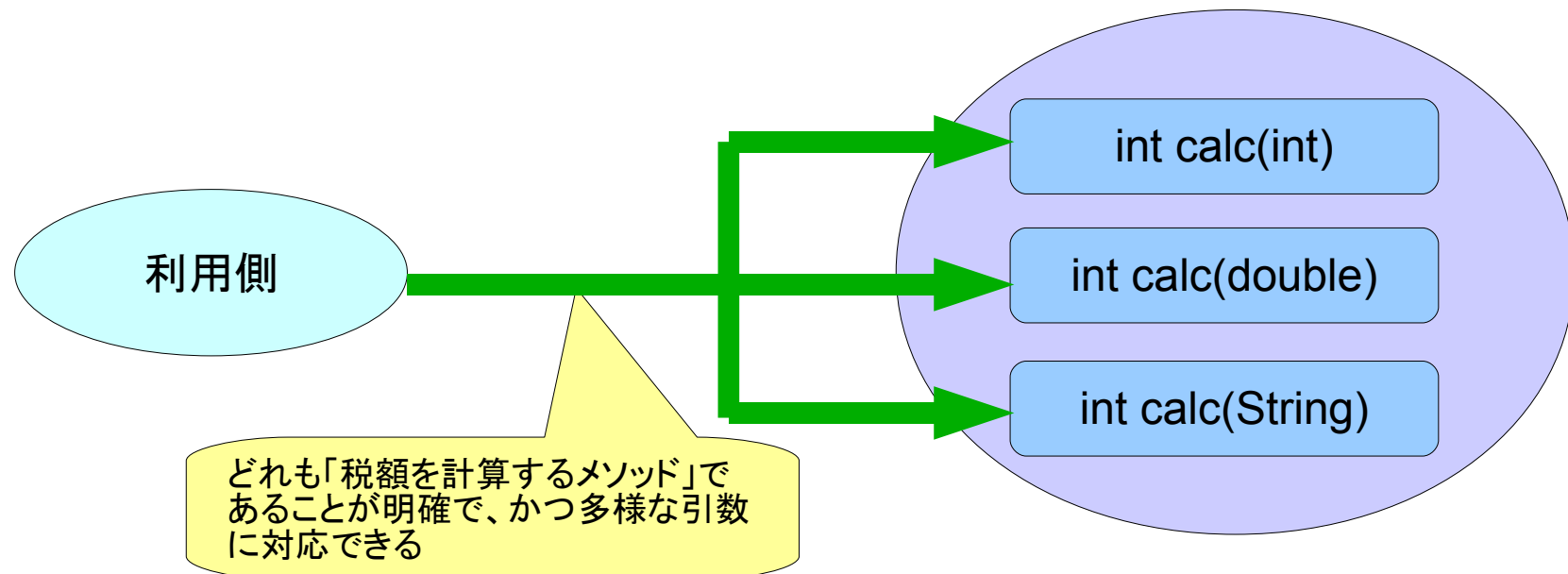
ポリモルフィズム(2)

- 引数のみが異なる、ほぼ同じ内容のメソッドに対してオーバーロードを使う場合の実装例
 - ロジックを1箇所にまとめ、コードの重複を防いでいる

```
public class Tax {  
  
    public static final double rate = 0.05;  
  
    public static int calc(int price) {  
        return (int)(price * rate);  
    }  
  
    public static int calc(double price) {  
        return calc((int)price);  
    }  
  
    public static int calc(String price) {  
        return calc(Integer.parseInt(price));  
    }  
  
}
```

ポリモルフィズム(2)

- ポリモルフィズムのメリット(2)
 - メソッド定義にポリモルフィズムを適用することで
 - 同じ機能を持つメソッドに同じ名称をつけることができクラスの内容を理解しやすくなる
 - 多様な引数に対応した機能を実装しやすくなる



例外処理

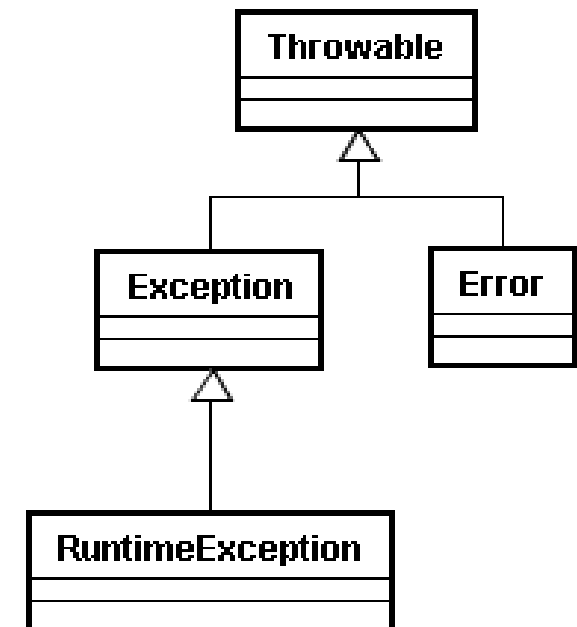
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

例外処理

- 例外とは
 - 実行時に発生するエラー(ランタイムエラー)をJavaで表現したもの
- 単なるランタイムエラーとの違い
 - ランタイムエラー
 - 発生するとそこでプログラムが異常停止してしまう
 - 例外
 - プログラム中で自らそれに対処する処理を書くことが出来る
 - 開発者自身で「例外を発生する」コードを書くことも出来る

例外処理

- 例外の種類
 - Exception
 - 通常の例外
 - RuntimeException
 - コード実行時にのみ発生する例外
 - Error
 - 対処が難しいような致命的な例外

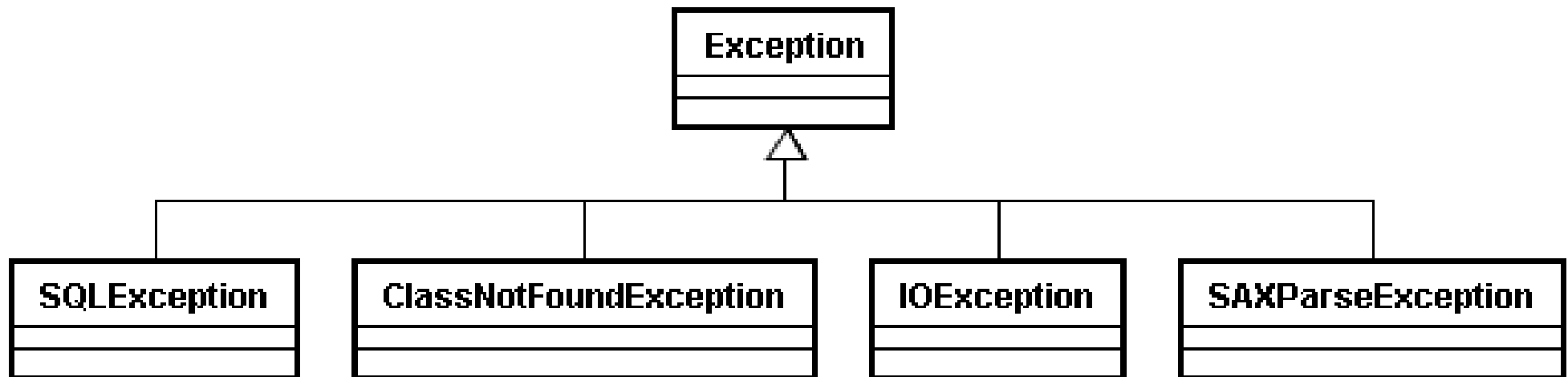


例外処理

- Javaでの「例外」の表現
 - 例外はクラスで表現される(例外クラス)
 - 例えば、「Exception」という例外クラスが存在している
 - 継承関係を持った子クラスも多数定義されている
 - クラスの型名を見ることで、どのような現象が起きたか判別することができる

例外処理

- 例外クラスの継承関係～Exceptionクラス

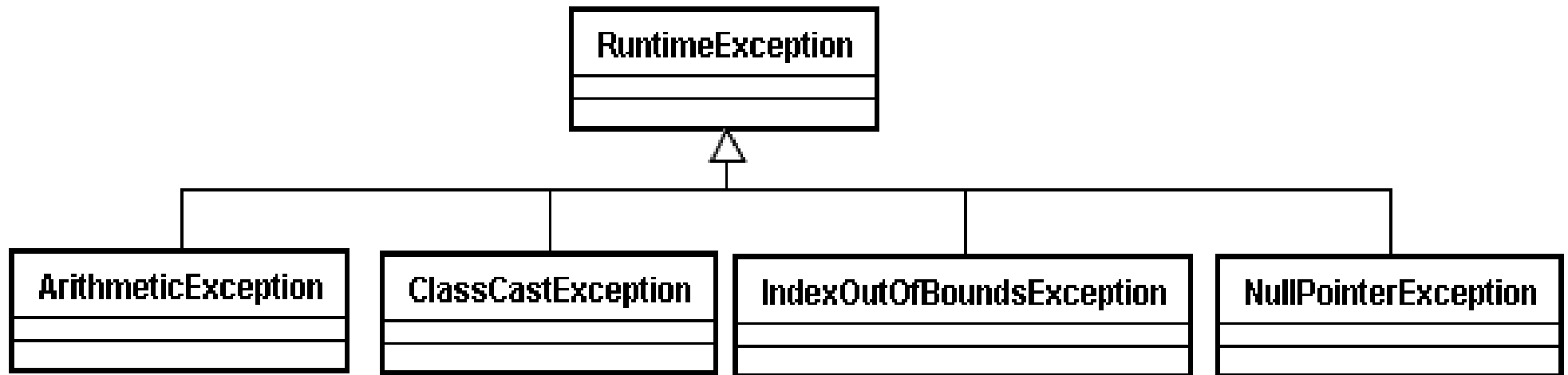


クラス名	内容
ClassNotFoundException	指定されたクラスが見つからないことを示す
IOException	入出力に関する不具合が生じたことを示す
SQLException	データベースアクセスに関する不具合が生じたことを示す
SAXParseException	XML文書を解釈している最中に不具合が生じたことを示す

※ 上記のクラスはException継承クラスの一部です

例外処理

- 例外クラスの継承関係～RuntimeExceptionクラス

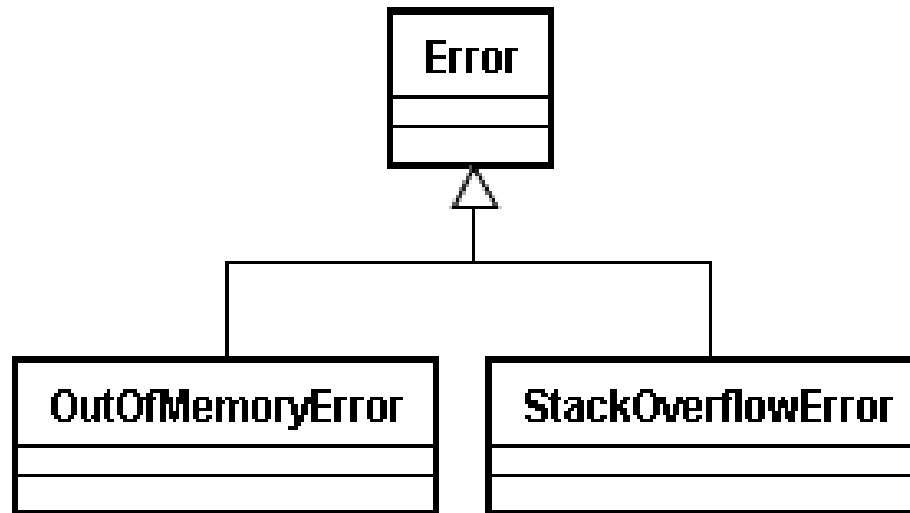


クラス名	内容
ArithmeticException	算術演算に関する処理で例外的条件が発生したことを示す
ClassCastException	クラスを他のクラスにキャストできなかったことを示す
IndexOutOfBoundsException	配列などのインデックスの指定が範囲外であったことを示す
NullPointerException	nullが代入された変数に対してメソッド・フィールドへのアクセスが行われたことを示す

※ 上記のクラスはRuntimeException継承クラスの一部です

例外処理

- 例外クラスの継承関係～Errorクラス



クラス名	内容
OutOfMemoryError	JavaVMに割り当てられた実行用メモリが不足したことを示す
StackOverflowError	再帰の回数が多すぎてスタック領域があふれたことを示す

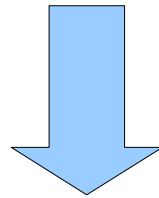
※ 上記のクラスはError継承クラスの一部です

例外処理

- 例外が「発生する」とは？
 - あるクラスの、あるメソッドを呼び出して実行
 - 呼び出し先のメソッド内で例外が発生する
 - 呼び出し元には、戻り値は返されず、代わりに「例外クラスのインスタンス」が返される

例外処理

- エラーとは予期しないもの
- 予期しないものを、どうして対処できるのか？



例外クラス(Exceptionとその派生クラス)が発生する可能性のあるメソッドは、そのことを宣言しておかなければならない

※ ただし、RuntimeException/Errorクラスを継承した例外クラスについてはその必要はありません

例外処理

- throws節
 - そのメソッドで発生する可能性のある例外クラスを列挙して宣言するための節
 - throws節に列挙されたクラスを見ることで、このメソッドを使う側が、どんな例外が発生する可能性があるか知ることができる

凡例

```
戻り値 メソッド名(引数リスト) throws 例外クラス名1,例外クラス名2... {  
  
}
```

例外処理

- try～catch節

- 呼び出したメソッドで例外が発生した場合に、それに対処する処理を記述(例外の捕捉)することの出来る制御文

凡例

```
try {  
    文1;  
    文2;  
    :  
} catch (例外クラス 引数名) {  
    文3;  
    文4;  
    :  
}
```

try { ... } のブロック内には、例外が発生する可能性のある処理を記述

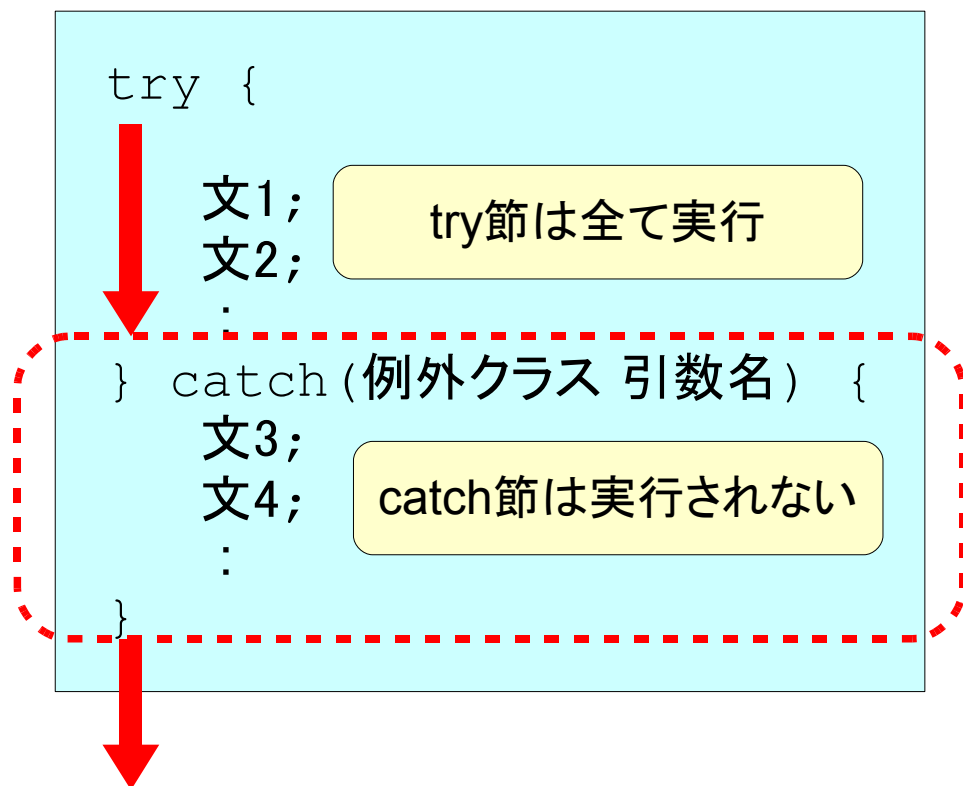
catch(例外クラス 引数名) には、対処したい例外クラスの型名と、例外を受け取ったときの引数名を記述

catch() { ... } のブロック内には、受け取った例外に対処するための処理を記述
※ catch() { ... } は複数種類記述してもよい

例外処理

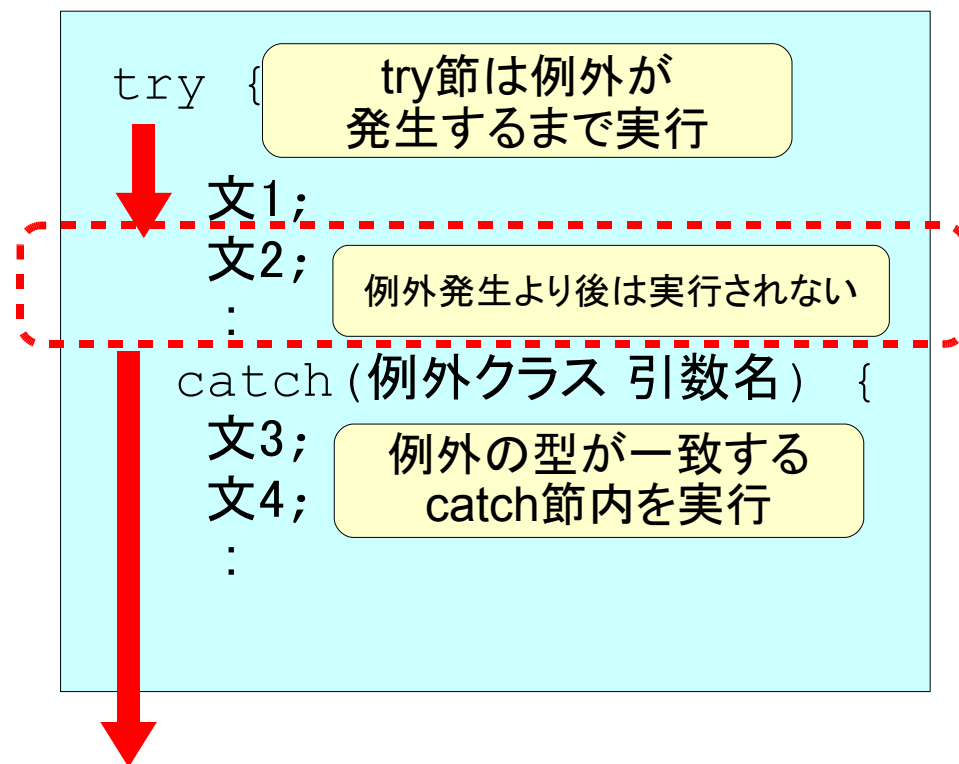
- 例外が発生したとき、しないときのtry～catch節の処理の流れ

例外が発生しなかったとき



例外が発生しなければ、try節内の処理は全て実行され、catch節内は実行されない

文1で例外が発生したとき



例外が発生すると、発生以降のtry節内の処理は実行されず、例外の型が一致するcatch節内の処理が実行される

例外処理

- finally節

- 例外発生の有無に関わらず、必ず実行させたい処理を記述する
- finally節に達する前にtry節やcatch節の内部でreturnが実行されても、finally節の内容は必ず実行される

凡例

```
try {  
    ...  
} catch (例外クラス 引数名) {  
    ...  
} finally {  
    実行したい処理  
}
```

例外処理

- try～catch節とthrows節の関係
 - throws節が宣言されているメソッドを呼ぶ場合、以下のどちらかの対処をしなければならない
 - try～catch節でその例外を捕捉する
 - 自分自身のメソッドのthrows節の宣言でその例外を列挙する
 - 自身のメソッド内にtry～catch節が無くても、throws節が宣言されていれば、発生した例外をそのまま呼び出し元に送出することが可能

例外処理

- throw文
 - 自分の作ったメソッド内で例外を発生させることができる

凡例

```
throw 例外のインスタンス;
```

コード例

```
throw new Exception();
```

```
Exception e = new Exception();  
throw e;
```

演習

- 下記のPersonクラスに年齢を変更するsetAgeメソッドを定義してください
 - setAgeメソッドは、int型の引数を持ちます
 - その引数が負の値だった場合は、Exception型の例外が発生します

```
public class Person {  
    public int age;  
    public Person() {  
        age = 0;  
    }  
}
```

次に、PersonTestクラスを作成し、mainメソッドでPersonクラスのインスタンスを作成してください。

作成したインスタンスに対しsetAgeメソッドを呼び出してください。
ただし、引数は負の値を与えるものとします。
try～catchで例外を捕捉し、catch節内で「例外が発生しました」とコンソールに表示してください。

解答例

- Personクラス

```
public class Person {  
    private int age;  
  
    public Person() {  
        age = 0;  
    }  
  
    public void setAge(int newAge) throws Exception {  
        if (newAge < 0) {  
            throw new Exception();  
        }  
        age = newAge;  
    }  
}
```

解答例

- PersonTestクラス

```
public class PersonTest {  
    public static void main(String[] args) {  
        try {  
            Person p = new Person();  
            p.setAge(-20);  
        } catch (Exception e) {  
            System.out.println("例外が発生しました");  
            e.printStackTrace();  
        }  
    }  
}
```

例外処理

- 例外クラスのコンストラクタ・メソッド
 - コンストラクタ
 - Exception(String)
 - 例外についての詳細メッセージを指定できる
 - Exception(Throwable)
 - この例外が発生する原因となった例外クラスを指定できる

例外処理

- 例外クラスのコンストラクタ・メソッド
 - メソッド
 - printStackTrace()
 - この例外のトレースを表示できる
 - getCause()
 - この例外が発生する原因となった例外のインスタンスを取得する
 - getMessage()
 - この例外の詳細メッセージを取得する

演習

- さきほど作成したPersonクラスのsetAgeメソッドで発生するException型の例外に、詳細メッセージ「指定された年齢が不正です」を追加してください。
- 次に、PersonTestクラスを作成し、mainメソッドでPersonクラスのインスタンスを作成してください。

解答例

- Personクラス

```
public class Person {  
  
    private int age;  
  
    public Person() {  
        age = 0;  
    }  
  
    public void setAge(int newAge) throws Exception {  
        if (newAge < 0) {  
            throw new Exception("指定された年齢が不正です");  
        }  
        age = newAge;  
    }  
  
}
```

例外処理

- 例外トレースとは

例外トレースの例

```
java.lang.Exception: 年齢の値が不正です -10  
    at Person.setAge(Person.java:11)  
    at StackTraceTest.main(StackTraceTest.java:8)
```

例外トレースの凡例

発生した例外クラス名 : 例外の詳細メッセージ

at 例外が発生したクラス名.メソッド名(ソースファイル名:行番号)

at 上記のメソッドの呼び出し元クラス名.メソッド名(ソースファイル名:行番号)

...

例外処理

- 例外トレースと読み方
 - 先頭行の例外クラス名と詳細メッセージで、何が起こったかを判断する
 - 2行目以降を見て、自分が作成したソースの何行目で例外が発生しているかを調べる
 - 例外発生箇所のソースを調べ、例外発生の原因を考える

例外処理

- 独自の例外クラスを作る
 - Exception/RuntimeException/Errorクラスを継承することで、独自の例外クラスを作ることも可能

コード例

```
public class MyException extends Exception {  
  
}
```

Javadocの読み方

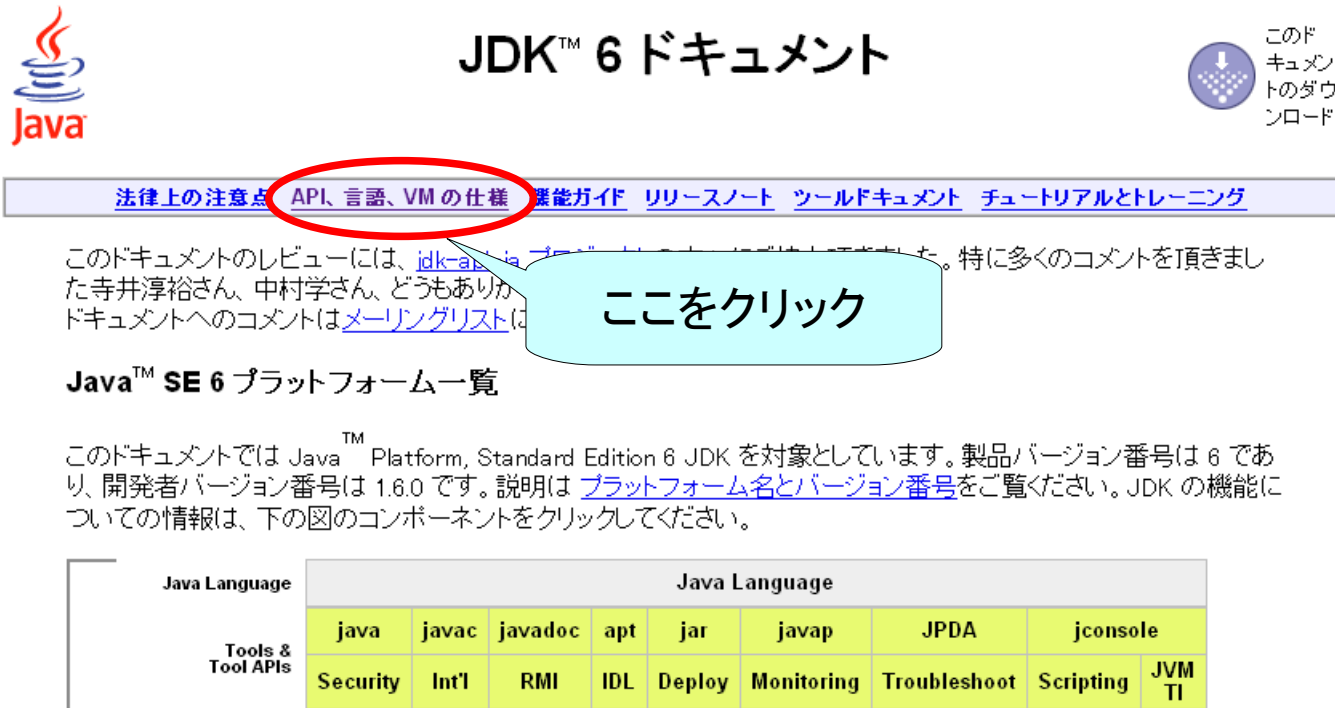
株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

Javadocとは何か

- Javadocとは
 - APIなどのクラスやインターフェースの仕様について解説したドキュメントのこと
 - ソースコード中に所定の形式でコメントを記入しておき、javadocコマンドを使用すると、HTML形式のJavadocが自動的に生成される
 - Javaの標準APIについても用意されているが、自分が作成したプログラムについてのJavadocを独自に作成することも可能

Javadocはどこにある？

- Javadocはどこにある？
 - JavaSE 6 SDKの標準APIについてのJavadoc
 - <http://java.sun.com/javase/ja/6/docs/ja/>



The screenshot shows the JDK 6 documentation page. The Java logo is on the left. The title 'JDK™ 6 ドキュメント' is in the center. On the right, there is a download icon and the text 'このドキュメントのダウンロード'. Below the title, there is a navigation bar with links: '法律上の注意点', 'API, 言語、VM の仕様', '機能ガイド', 'リリースノート', 'ツールドキュメント', and 'チュートリアルとトレーニング'. The link 'API, 言語、VM の仕様' is circled in red. A callout bubble with the text 'ここをクリック' points to this link. Below the navigation bar, there is a paragraph of text and a section titled 'Java™ SE 6 プラットフォーム一覧'. Below this, there is a table with two columns: 'Java Language' and 'Tools & Tool APIs'. The 'Java Language' column contains links to 'java', 'javac', 'javadoc', 'apt', 'jar', 'javap', 'JPDA', and 'jconsole'. The 'Tools & Tool APIs' column contains links to 'Security', 'Int'l', 'RMI', 'IDL', 'Deploy', 'Monitoring', 'Troubleshoot', 'Scripting', and 'JVM TI'.

このドキュメントのレビューには、[jdk-6u10](#) プラットフォーム名とバージョン番号を指定してください。特に多くのコメントを頂きました寺井淳裕さん、中村学さん、どうもありがとうございます。ドキュメントへのコメントは[メーリングリスト](#)に

ここをクリック

Java™ SE 6 プラットフォーム一覧

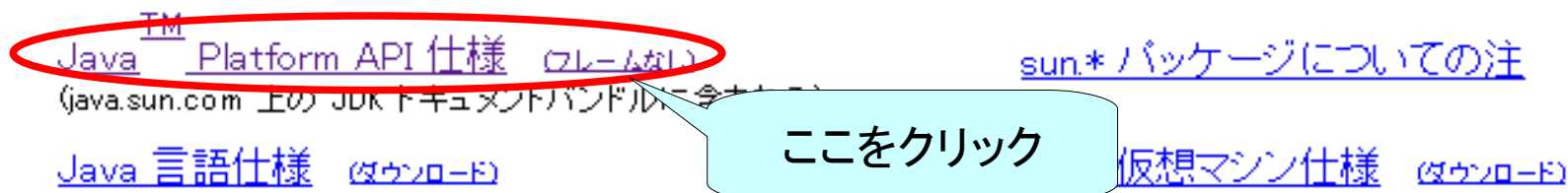
このドキュメントでは Java™ Platform, Standard Edition 6 JDK を対象としています。製品バージョン番号は 6 であり、開発者バージョン番号は 1.6.0 です。説明は [プラットフォーム名とバージョン番号](#) をご覧ください。JDK の機能についての情報は、下の図のコンポーネントをクリックしてください。

Java Language	Java Language								
Tools & Tool APIs	java	javac	javadoc	apt	jar	javap	JPDA	jconsole	
	Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI

Javadocはどこにある？

- Javadocはどこにある？（続き）

API、言語、仮想マシンのドキュメント



The screenshot shows the top of the Java Platform API Specification page. A red circle highlights the text "JavaTM Platform API 仕様 (クレームなし)". A blue speech bubble with the text "ここをクリック" (Click here) points to this link. Other visible links include "sun.* パッケージについての注" (Notes on the sun.* packages), "Java 言語仕様 (ダウンロード)" (Java Language Specification (Download)), and "仮想マシン仕様 (ダウンロード)" (Virtual Machine Specification (Download)).

機能リファレンスガイド - Java プラットフォーム

以下のガイドは、特に説明がない限り、java.sun.com 上の Web サイトおよびドキュメントのダウンロードバンドルにあります。

[Java SE 6 概要](#)
[新機能と拡張機能](#)

(java.sun.com の Web サイトのみ)

標準APIのJavadoc

Java™ Platform
Standard Ed. 6

[すべてのクラス](#)

パッケージ

[java.applet](#)

[java.awt](#)

[java.awt.color](#)

[java.awt.datatransfer](#)

[java.awt.dnd](#)

[java.awt.event](#)

すべてのクラス

[AbstractAction](#)

[AbstractAnnotationValue](#)

[AbstractBorder](#)

[AbstractButton](#)

[AbstractCellEditor](#)

[AbstractCollection](#)

[AbstractColorChooserFactory](#)

[AbstractDocument](#)

[AbstractDocument.AttributeKey](#)

[AbstractDocument.ContentChange](#)

[AbstractDocument.ElementChange](#)

概要 [パッケージ](#) [クラス](#) [使用](#) [階層ツリー](#) [非推奨 API](#) [索引](#) [ヘルプ](#)

前 次

[フレームあり](#) [フレームなし](#)

Java™ Platform
Standard Ed. 6

Java™ Platform, Standard Edition 6 API 仕様

このドキュメントは、Java Platform Standard Edition 6 の API 仕様です。

参照先:

[説明](#)

パッケージ

[java.applet](#)

アプレットの作成、およびアプレットとアプレットテキストとの通信に使用するクラスの作成に必要なクラスを提供します。

[java.awt](#)

ユーザーインターフェースの作成およびグラフィクスとイメージのペイント用のすべてのクラスをします。

[java.awt.color](#)

カラースペースのクラスを提供します。

アプレット、およびアプレットとアプレットとの通信に使用するクラスの作成に必要なクラスを提供します。

Javadoc画面のみかた

• トップ画面の構成

索引に表示する クラスを絞り込む

Platform
rd Ed. 6

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)
- [java.awt.dnd](#)
- [java.awt.event](#)

すべてのクラス

- [AbstractAction](#)
- [AbstractAnnotationValue](#)
- [AbstractBorder](#)
- [AbstractButton](#)
- [AbstractCellEditor](#)
- [AbstractCollection](#)
- [AbstractColorChooserFactory](#)
- [AbstractComponent](#)

クラス名索引

[概要](#) [パッケージ](#) [クラス](#) [使用](#) [階層ツリー](#) [非推奨](#) [API](#) [索引](#) [ヘルプ](#)

前 次 フレームあり フレームなし

**JavaTM Platform
Standard Ed. 6**

Java™ Platform, Standard Edition
API仕様

J2SEに含まれる パッケージの概要

このドキュメントは、Java Platform Standard Edition 6 の API 仕様です。

参照先：
説明

パッケージ

<u>java.applet</u>	アプレットの作成、およびアプレットとアプレットテキストとの通信に使用するクラスの作成に必要なクラスを提供します。
<u>java.awt</u>	ユーザーインタフェースの作成およびグラフィクスとイメージのペイント用のすべてのクラスをます。
<u>java.awt.color</u>	カラースペースのクラスを提供します。
	アプレットの起動、またはアプレットの起動内

Javadocの読み方

- パッケージごとの詳細をみるには



The screenshot shows the Java Platform Standard Ed. 6 Javadoc site. On the left, there is a sidebar with a list of packages under the heading 'すべてのクラス' (All classes). The packages listed are: `java.applet`, `java.awt`, `java.awt.color`, `java.awt.datatransfer`, `java.awt.dnd`, and `java.awt.event`. Below this, there is another list of classes under the heading 'すべてのクラス' (All classes), including `AbstractAction`, `AbstractAnnotationValue`, `AbstractBorder`, `AbstractButton`, `AbstractCellEditor`, `AbstractCollection`, `AbstractColorChooser`, `AbstractDocument`, `AbstractDocument.Attribute`, `AbstractDocument.Content`, `AbstractDocument.Element`, `AbstractElementVisitor`, and `AbstractExecutorService`.

On the right, there is a table of packages and their descriptions. The packages listed are: `java.awt.print`, `java.beans`, `java.beans.beancontext`, `java.io`, `java.lang`, `java.lang.annotation`, `java.lang.instrument`, `java.lang.management`, and `java.lang.ref`. The `java.lang` package is highlighted with a red circle, and a callout bubble points to it with the text 'ここをクリック' (Click here).

Package	Description
java.awt.print	このパッケージは、印刷用 API を使用するためのクラスおよびインタフェースを提供します。
java.beans	Beans (JavaBeans TM アーキテクチャーに基づくコンポーネント) の開発に関連するクラスが提供されています。
java.beans.beancontext	Bean コンテキストに関連するクラスおよびインタフェースを提供します。
java.io	このパッケージは、データストリーム、直列化、フィルタシステムによるシステム入出力用に提供されています。
java.lang	Java プログラム言語の設計にあたり基本的なクラスを提供します。
java.lang.annotation	Java プログラミング言語の注釈機能をサポートするライブラリを提供します。
java.lang.instrument	Java プログラミング言語エージェントが JVM 実行されているプログラムを計測できるようにサービスを提供します。
java.lang.management	Java 仮想マシンの管理および Java 仮想マシンが実行されているオペレーティングシステムの管理を監視する管理インタフェースを提供します。
java.lang.ref	ガベージコレクタとの制限付きの対話をサポートする、参照オブジェクトクラスを提供します。

Javadocの読み方

• パッケージごとのインデックス画面

Java™ Platform
Standard Ed. 6

すべてのクラス

パッケージ

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)
- [java.awt.dnd](#)
- [java.awt.event](#)

すべてのクラス

- [AbstractAction](#)
- [AbstractAnnotationValue](#)
- [AbstractBorder](#)
- [AbstractButton](#)
- [AbstractCellEditor](#)
- [AbstractCollection](#)
- [AbstractColorChooserFactory](#)
- [AbstractDocument](#)
- [AbstractDocument.AttributeKey](#)
- [AbstractDocument.ContentChange](#)
- [AbstractDocument.Element](#)
- [AbstractElementVisitor](#)

概要 **パッケージ** クラス 使用 階層ツリー 非推奨 API 索引 ヘルプ

[前のパッケージ](#) [次のパッケージ](#) [フレームあり](#) [フレームなし](#)

パッケージ java.lang

Java プログラム言語の設計にあたり基本的なクラスを提供します。

参照先:
[説明](#)

インタフェースの概要	
Appendable	char シーケンスと値を追加できるオブジェクトです。
CharSequence	CharSequence は char 値の読むことのできるオブジェクトです。
Cloneable	Object.clone() メソッドに対して、そのメソッドのインスタンスのフィールド対フィールドの複製を作成できることを示すために、Cloneable インタフェースを実装したクラスです。
Comparable<T>	このインタフェースを実装する各クラスのオブジェクトに全体的順序付けを強制します。

Java™ Platform
Standard Ed. 6

パッケージ名称と
パッケージについて
の簡単な説明

パッケージに含まれる
インタフェースの
一覧と簡単な説明

Javadocの読み方

- パッケージごとのインデックス画面(続き)

Java™ Platform
Standard Ed. 6

すべてのクラス

パッケージ

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)
- [java.awt.dnd](#)
- [java.awt.event](#)

すべてのクラス

- [AbstractAction](#)
- [AbstractAnnotationValue](#)
- [AbstractBorder](#)
- [AbstractButton](#)
- [AbstractCellEditor](#)
- [AbstractCollection](#)
- [AbstractColorChooserFactory](#)

クラスの概要	
Boolean	Boolean クラスは、プリミティブ型 boolean の値をオブジェクトにラップします。
Byte	Byte クラスは、プリミティブ型 byte の値をオブジェクト内にラップします。
Character	Character クラスは、プリミティブ型 char の値をオブジェクトにラップします。
Character	Character クラスは、Unicode 文字セットの特定のサブセットを表す文字サブセットのファミリー。
Character.UnicodeBlock	Character.UnicodeBlock クラスは、Unicode 文字セットの特定のサブセットを表す文字サブセットのファミリー。
Class<T>	Class クラスのインスタンスは、実行中のクラスおよびインタフェースを表します。
ClassLoader	クラスローダーは、クラスのロードを担当します。
Compiler	Compiler クラスは、Java からネイティブコードへのコンパイラおよび関連サービスをサポートします。

ここをクリックすると、クラスごとのインデックス画面が表示される

パッケージに含まれるクラスの一覧と簡単な説明

※ 同様に、例外やエラークラスの一覧などもあります

Javadocの読み方

・クラスごとのインデックス画面

概要 [パッケージ](#) **クラス** [使用](#) [階層ツリー](#) [非推奨 API](#) [索引](#) [ヘルプ](#)

[前のクラス](#) [次のクラス](#)

概要: [入れ子](#) | [フィールド](#) | [コンストラクタ](#) | [メソッド](#)

[フレームあり](#) [フレームなし](#)

詳細: [フィールド](#) | [コンストラクタ](#) | [メソッド](#)

Java™ Platform
Standard Ed. 6

パッケージ名とクラス名

java.lang

クラス String

クラスの継承関係

[java.lang.Object](#)

↳ [java.lang.String](#)

実装しているインターフェース

すべての実装されたインターフェース:

[Serializable](#), [CharSequence](#), [Comparable<String>](#)

ソース上のクラス定義部

```
public final class String
```

```
extends Object
```

```
implements Serializable, Comparable<String>, CharSequence
```

String クラスは文字列を表します。Java プログラム内の「abc」などのリテラル文字列はすべて、このクラスのインスタンスとして実行されます。

文字列は定数です。この値を作成したあとに変更はできません。文字列バッファは可変文字列をサポートします。文字列オブジェクトは不変であるため、共用することができます。次に例を示します。

```
String str = "abc";
```

これは、次と同じです。

クラスの概要説明

Javadocの読み方

• クラスごとのインデックス画面(続き)

フィールドの概要	
static Comparator < String	CASE INSENSITIVE ORDER compareToIgnoreCase の場合と同じように String オブジェクトを順序付ける Comparator

定義されているフィールドの一覧

ここをクリックすると、そのフィールドの解説へジャンプする

コンストラクタの概要	
String ()	新しく生成された String オブジェクトを初期化して、空の文字シーケンスを表すようにします。
String (byte[] bytes)	プラットフォームのデフォルトの文字セットを使用して、指定されたバイト配列を復号化することによって、新しい String を構築します。
String (byte[] bytes, Charset charset)	指定された 文字セット を使用して、指定されたバイト配列を復号化することによって、新しい String を構築します。
String (byte[] ascii, int hibyte)	推奨されていません。このメソッドでは、バイトから文字への変換が正しく行われません。 JDK 1.1 以降、 String コンストラクタの使用が推奨されています。 Charset 、文字セットの名前を取る String コンストラクタを使用する String コンストラクタの使用が推奨されています。
String (byte[] bytes, int offset, int length)	プラットフォームのデフォルトの文字セットを使用して、指定されたバイト部分配列を復号化することによって、新しい String を構築します。

定義されているコンストラクタの一覧

ここをクリックすると、そのコンストラクタの解説へジャンプする

Javadocの読み方

- クラスごとのインデックス画面(続き)

メソッドの概要		定義されているメソッドの一覧
char	charAt (int index) 指定されたインデックス位置にある char 値を返します。	ここをクリックすると、その メソッドの解説へジャンプする
int	codePointAt (int index) 指定されたインデックス位置の文字 (Unicode コードポイント) を返します。	
int	codePointBefore (int index) 指定されたインデックスの前の文字 (Unicode コードポイント) を返します。	
int	codePointCount (int beginIndex, int endIndex) この String の指定されたテキスト範囲の Unicode コードポイントの数を返します。	
int	compareTo (String anotherString) 2 つの文字列を比較します。	
int	compareToIgnoreCase (String anotherString) 大文字と小文字を区別せずに 2 つの文字列を比較します。	
String	concat (String str) 指定された文字列をこの文字列の最後に連結します。	

Javadocの読み方

• メソッドごとの解説

charAt

```
public char charAt(int index)
```

メソッドの使い方の説明

指定されたインデックス位置にある char 値を返します。インデックスは、0 から length() - 1 の範囲になります。配列のインデックス付けの場合と同じように、シーケンスの最初の char 値のインデックスは 0、次の文字のインデックスは 1 と続きます。

インデックスで指定された char 値がサロゲートの場合、サロゲート値が返されます。

定義:

インタフェース [CharSequence](#) 内の [charAt](#)

パラメータ:

index - char 値のインデックス

引数についての説明

戻り値:

文字列内の指定されたインデックス位置にある char 値。最初の char 値のインデックスが 0 になる

戻り値についての説明

例外:

[IndexOutOfBoundsException](#) - index 引数が負の値、または文字列の長さと同じかこれより大きい値の場合

発生する可能性のある
例外についての説明

※ 例外については、throws節に記述されている例外について解説されています

APIの活用

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

APIの活用

- APIとは？
 - Application Programming Interfaceの略
 - アプリケーションにおいて頻繁に使われる機能をあらかじめ実現したモジュールをまとめて提供するもの
 - Javaの場合、クラスやインターフェースとして提供される

APIの活用

- 標準API

- 開発環境(JDK)にあらかじめ付属しているAPIのこと
 - Javaの標準APIは、パッケージ名が「java.」で始まる
 - 「javax.」で始まるものは「標準拡張API」と呼ばれる

APIの活用

- 標準APIの紹介

- java.langパッケージ

- 基本データ型のラッパクラス
 - Mathクラス
 - Systemクラス
 - Stringクラス

- java.ioパッケージ

- Fileクラス
 - ストリームクラス

- java.textパッケージ

- フォーマッタクラス

- java.utilパッケージ

- ArrayListクラス
 - HashMapクラス
 - Date/Calendarクラス

APIの活用

- 基本データ型のラッパクラス(java.langパッケージ)
 - int,double,booleanなどの基本データ型の値をインスタンスとして保持するためのクラス

基本データ型	対応するラッパクラス
int	Integer
long	Long
short	Short
byte	Byte
double	Double
float	Float
boolean	Boolean
char	Character

APIの活用

• ラップクラスに値を渡す方法

- コンストラクタに、基本データ型の値（またはその値を文字列で表記したもの）を渡す

コード例

```
int a = 100;  
Integer b = new Integer(a);  
  
String c = "true";  
Boolean d = new Boolean(c);
```

- auto-boxing/auto-unboxingを使う

- ラップクラスと基本データ型はそのまま相互に代入することが可能
- キャスト演算子を使って変換することもできる
※J2SE 5.0で追加された新機能

コード例

```
int a = 100;  
Integer b = a; または Integer b = (Integer)a;  
int c = b; または int c = (int)b;
```


APIの活用

- Mathクラス(java.langパッケージ)
 - 数学的な演算を行うメソッドや定数が定義されている
 - Eフィールド
 - 自然対数の値
 - PIフィールド
 - 円周率(π)の値
 - sin(), cos(), tan()メソッド
 - 三角関数
 - random()メソッド
 - 0以上1未満のdouble型の乱数を発生
 - max(), min() メソッド
 - 2値のうちで大きいほう(小さいほう)の値を返す

演習

- Mathクラスのrandomメソッドを利用して、実行するたびに0～9までの乱数をコンソールに表示するクラスを作成してみましょう。

解答例

- RandomTestクラス

```
public class RandomTest {  
    public static void main(String[] args) {  
        int rnd = (int) (Math.random() * 10);  
        System.out.println(rnd);  
    }  
}
```

APIの活用

- Systemクラス(java.langパッケージ)
 - 標準出力へのアクセスや、いくつかの便利なメソッドを持つクラス
 - outフィールド(java.io.PrintStreamクラス)
 - 標準出力への出力を行うオブジェクト(printlnメソッドやprintfメソッドなどを持つ)
 - inフィールド(java.io.InputStreamクラス)
 - 標準入力からの入力を行うオブジェクト(readメソッドなどを持つ)
 - errフィールド(java.io.PrintStreamクラス)
 - 標準エラー出力への出力を行うオブジェクト(使用方法はoutと同様)

APIの活用

- System.outオブジェクトのメソッド
 - print(Object)メソッド
 - 標準出力に、指定されたオブジェクト(を文字列に変換したもの)を表示
 - 表示後の改行は行わない
 - println(Object)メソッド
 - 標準出力に、指定されたオブジェクト(を文字列に変換したもの)を表示
 - 表示後に改行を行う
 - printf(String,Object...)メソッド
 - 標準出力に、Stringで指定したフォーマットに従った形式で、指定されたオブジェクト(を文字列に変換したもの)を表示
 - フォーマット形式は、C言語のprintf関数のものとほぼ同じ
 - 「Object...」は、Object型の引数を複数(任意の数)指定できることを示す
 - 表示後の改行は行わない

APIの活用

- Systemクラス(java.langパッケージ)
 - exit(int)メソッド
 - VMを強制終了させるメソッド
 - getProperty(String)メソッド
 - システムプロパティの値を取得するメソッド
 - gc()メソッド
 - ガベージコレクタを動作させるメソッド
 - currentTimeInMillis()メソッド
 - 現在時刻(1970年1月1日午前0時から経過時間)をミリ秒単位で取得するメソッド

APIの活用

- Stringクラス(java.langパッケージ)
 - charAt(int)メソッド
 - 指定位置の文字1文字をchar型として取得するメソッド
 - length()メソッド
 - この文字列の長さをint型の数値で返す
 - indexOf(String)メソッド
 - 引数の文字列が何文字目に登場するかを返す
 - startsWith(String) / endsWith(String)
 - この文字列が、引数で指定された文字列で始まる(終わる)かどうかをboolean型で返す
 - toUpperCase() / toLowerCase()
 - この文字列を大文字(小文字)に変換して返す

APIの活用

- Fileクラス(java.ioパッケージ)
 - ファイルシステム上のファイルを抽象化したクラス
 - コンストラクタ
 - File(String)
 - 引数に指定されたパスおよびファイル名のファイルを示すFileインスタンスを生成
 - 主なメソッド
 - createNewFile()
 - ファイルを新規に作成
 - delete()
 - ファイルを削除
 - list()
 - このディレクトリに含まれるファイルのリストを配列で取得

APIの活用

- ストリームクラス(java.ioパッケージ)
 - あるデータを読み込んだり、書き込んだりする処理を実現してくれるクラス群
 - InputStream/OutputStream
 - バイナリデータ(byte配列)を読み込み/書き込みする基底クラス
 - Reader/Writer
 - テキストデータ(文字列)を読み込み/書き込みする基底クラス

APIの活用

- InputStream/OutputStreamの派生クラス(例)
 - FileInputStream/FileOutputStream
 - ファイルをバイナリ形式で読み込み/書き込みするためのストリームクラス
 - ByteArrayInputStream/ByteArrayOutputStream
 - バイト配列からデータを読み込んだり、書き込んだりするためのストリームクラス
 - BufferedInputStream/BufferedOutputStream
 - バッファを使ってバイナリデータを読み込み/書き込みするためのストリームクラス
 - 他のストリームクラスと組み合わせて用いられる

APIの活用

- Reader/Writerの派生クラス(例)
 - FileReader/FileWriter
 - ファイルをテキスト形式で読み込み/書き込みするためのストリームクラス
 - CharArrayReader/CharArrayWriterchar
 - 配列からデータを読み込んだり、書き込んだりするためのストリームクラス
 - InputStreamReader/OutputStreamWriter
 - バイナリ形式からテキスト形式に変換してデータを読み込み/書き込みするためのストリームクラス
 - BufferedReader/BufferedWriter
 - バッファを使ってテキストデータを読み込み/書き込みするためのストリームクラス他のストリームクラスと組み合わせて用いられる

APIの活用

- Fileクラスの利用例
 - ファイル・ディレクトリの一覧を表示

```
import java.io.File;

public class DirectoryList {

    public static void main(String[] args) {

        File f = new File("C:¥¥");
        String[] list = f.list();
        for(String s:list)
            System.out.println(s);

    }

}
```

APIの活用

- ストリームクラスの利用例

- Reader/BufferedReaderクラスを利用してファイルの内容を読み込み

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

public class FileReadTest {
    public static void main(String[] args) {
        try {
            File f = new File("C:\\boot.ini");
            FileReader fr = new FileReader(f);
            BufferedReader br = new BufferedReader(fr);
            while(true) {
                String s = br.readLine();
                if (s==null) break;
                System.out.println(s);
            }
            br.close();
            fr.close();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

APIの活用

- java.utilパッケージのクラス
 - ArrayListクラス
 - 可変長配列を実現するクラス
 - HashMapクラス
 - ハッシュ関数を使った配列を実現するクラス
 - Date/Calendarクラス
 - 日付データを扱うためのクラス

APIの活用

- ArrayListクラス

- 可変長配列を実現するクラス

- 通常の配列は固定長なので、いったん定義すると、配列の大きさを変更することが出来なかった
 - ArrayListは可変長配列のため、定義後に大きさを自由に變更できる(必要な時に必要なだけデータを格納できるので、事前に大きさを宣言しておく必要がない)

- 格納できるデータはインスタンスに限られる

- 基本データ型の値を直接格納することはできない

APIの活用

- ArrayListクラスのメソッド
 - add(Object)
 - リストの末尾に引数の内容を追加
 - add(int, Object)
 - リストのintで指定した箇所にObjectの内容を追加
 - set(int, Object)
 - リストのintで指定した箇所の要素をObjectの内容に変更
 - get(int)
 - リストの指定位置にある要素の値をObject型で返す

APIの活用

- ArrayListクラスのメソッド
 - clear()
 - リストを空にする
 - contains(Object)
 - 指定の値がリストに含まれるかどうかをboolean型で返す
 - remove(int)
 - リスト内の指定の位置からオブジェクトを削除
 - remove(Object)
 - リスト内にObjectと同じ値があればその要素を削除
 - size()
 - リストのサイズ(要素の数)を返す

APIの活用

- ArrayListクラスの利用例

```
import java.util.ArrayList;

public class ArrayListTest {

    public static void main(String[] args) {

        ArrayList list = new ArrayList();
        list.add("Java");
        list.add(150);
        list.add(true);

        System.out.println(list.size());
        System.out.println(list.get(2));
    }
}
```

APIの活用

- HashMapクラス

- ハッシュ関数による配列を実現するクラス
 - 通常の配列は添え字に数値以外は使えない
 - HashMapはハッシュ関数による配列なので、添え字(キーともいう)の値は任意のものを利用することが出来る
- キーと格納できるデータはインスタンスに限られる
 - 基本データ型の値をキーとしたり、データとして直接格納することはできない

キー	値
Name	G.Paltrow
Gender	Female
Age	30
Country	America

APIの活用

- HashMapクラスのメソッド
 - put(Object, Object)
 - 第一引数をキー、第二引数を要素の値としてリストに追加
 - get(Object)
 - 指定された値をキーとして、要素の値をObject型で返す
 - clear()
 - リストを空にする
 - remove(Object)
 - 指定されたキーに対応する要素をリストから削除

APIの活用

- HashMapクラスのメソッド
 - containsKey(Object)
 - 指定のキーがリストに含まれるかどうかをboolean型で返す
 - containsValue(Object)
 - 指定の値がリストに含まれるかどうかをboolean型で返す
 - keySet()
 - このリストで使用されているキーの集合を返す
 - size()
 - リストのサイズ(要素の数)を返す

APIの活用

- HashMapクラスの利用例

```
import java.util.HashMap;

public class HashMapTest {

    public static void main(String[] args) {

        HashMap map = new HashMap();
        map.put("Name", "G.Paltrow");
        map.put("Age", 30);
        map.put("Gender", "Female");

        System.out.println(map.get("Gender"));

    }

}
```

Generics(総称)機能の利用

- ArrayList/HashMapの問題点
 - ArrayList/HashMapには任意の型のインスタンスを格納できるが、誤った型変換によってランタイムエラーを起こすことがある
 - 「誤った型変換」はコンパイル時には検出できない
- Genericsの利用
 - Genericsを使うと、ArrayList/HashMapクラスで使用する値の型を、事前に限定することができる
 - 限定された型以外のオブジェクトを使うとコンパイルエラーになるため、誤った型変換でランタイムエラーとなる可能性を排除できる

Generics(総称)機能の利用

- Genericsの記述方法
 - 型名の後ろに「<限定したい型名>」と記述する

ArrayListの場合

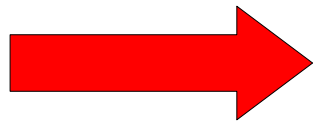
```
ArrayList<String> list = new ArrayList<String>();
```

HashMapの場合

```
HashMap<String,Integer> map = new HashMap<String,Integer>();
```


Generics(総称)機能の利用

- Genericsを利用したコードのコンパイル
 - 限定された型以外を扱おうとすると、コンパイルエラーになる



スライド#268,#272のそれぞれにGenericsで型の限定をして、コンパイルするとどうなるでしょうか？

APIの活用

- Date/Calendarクラス
 - Dateクラス
 - 日付(年月日時分秒)の情報を保持するためのクラス
 - Calendarクラス
 - Date型のオブジェクトを加工して扱う機能を持った抽象クラス
 - GregorianCalendarクラス
 - Calendarクラスの子クラスで、多くの機能をCalendarクラスから継承している
 - 世界各国で採用されているグレゴリオ暦を表現したクラス

APIの活用

- Dateクラスのコンストラクタ・メソッド
 - コンストラクタ
 - Date()
 - 現在のシステム時刻を値として保持するインスタンスを生成
 - Date(long)
 - 引数を1970年1月1日0時0分0秒からの経過時刻(単位＝ミリ秒)とみなした時刻を値として保持するインスタンスを生成

APIの活用

- Dateクラスのコンストラクタ・メソッド
 - メソッド
 - setTime(long)
 - 引数を1970年1月1日0時0分0秒からの経過時刻(単位＝ミリ秒)とみなした時刻をこのインスタンスの値として置き換える
 - getTime()
 - このインスタンスが保持する時刻を、1970年1月1日0時0分0秒からの経過時刻(単位＝ミリ秒)とみなしたlong型の値として返す
 - after(Date)
 - 自分の日付が引数の日付よりも後の日付かどうかをbooleanで返す
 - before(Date)
 - 自分の日付が引数の日付よりも前の日付かどうかをbooleanで返す

APIの活用

- GregorianCalendarクラスのコンストラクタ・メソッド
 - コンストラクタ
 - GregorianCalendar()
 - デフォルトの状態でインスタンスを生成
 - メソッド
 - setTime(Date)
 - 引数に指定された日付データをこのインスタンスに渡す
 - setTimeInMillis(long)
 - 引数を1970年1月1日0時0分0秒からの経過時刻(単位＝ミリ秒)とみなした時刻をこのインスタンスに渡す

APIの活用

- GregorianCalendarクラスのコンストラクタ・メソッド
 - メソッド
 - set(int,int,int)
 - 引数を、それぞれ年・月・日とみなしてこのインスタンスに渡す
 - set(int,int,int,int,int)
 - 引数を、それぞれ年・月・日・時・分とみなしてこのインスタンスに渡す
 - set(int,int,int,int,int,int)
 - 引数を、それぞれ年・月・日・時・分・秒とみなしてこのインスタンスに渡す
 - set(int,int)
 - 第一引数に指定されたフィールドに、第二引数の値を渡す

APIの活用

- GregorianCalendarクラスのコンストラクタ・メソッド
 - メソッド
 - getTime()
 - インスタンスが保持する日付をDate型で返す
 - getTimeInMillis()
 - インスタンスが保持する日付を、1970年1月1日0時0分0秒からの経過時刻(単位＝ミリ秒)とみなした時刻で返す
 - get(int)
 - 引数に指定されたフィールドの値を返す
 - add(int,int)
 - 第一引数に指定されたフィールドに第二引数の値を加算

APIの活用

- Calendarクラスのフィールド
 - add/set/getメソッドで指定するフィールドの値

フィールド名	内容
YEAR	「年」を示すフィールド値です。
MONTH	「月」を示すフィールド値です。
DATE	「日」を示します。
HOUR	「時」(12時間制)を示します。
HOUR_OF_DAY	「時」(24時間制)を示します。
MINUTE	「分」を示します。
SECOND	「秒」を示します。
MILLISECOND	「ミリ秒」を示します。
WEEK_OF_MONTH	現在の月の何週目かを示します。
WEEK_OF_YEAR	現在の年の何週目かを示します。
DAY_OF_MONTH	「日」を示します。(DATEフィールドと同じ)
DAY_OF_WEEK	「曜日」を示します。
DAY_OF_WEEK_IN_MONTH	現在の月の何度目の曜日かを示します。
DAY_OF_YEAR	現在の年の何日目かを示します。

APIの活用

- Calendarクラスのフィールド
 - 「月」を示す値(数値でも指定可能)

フィールド名	内容
JANUARY	「1月」を示す MONTH フィールドの値です。
FEBRUARY	「2月」を示す MONTH フィールドの値です。
MARCH	「3月」を示す MONTH フィールドの値です。
APRIL	「4月」を示す MONTH フィールドの値です。
MAY	「5月」を示す MONTH フィールドの値です。
JUNE	「6月」を示す MONTH フィールドの値です。
JULY	「7月」を示す MONTH フィールドの値です。
AUGUST	「8月」を示す MONTH フィールドの値です。
SEPTEMBER	「9月」を示す MONTH フィールドの値です。
OCTOBER	「10月」を示す MONTH フィールドの値です。
NOVEMBER	「11月」を示す MONTH フィールドの値です。
DECEMBER	「12月」を示す MONTH フィールドの値です。

APIの活用

- Calendarクラスのフィールド
 - 「曜日」を示す値

フィールド名	内容
SUNDAY	日曜日を示す DAY_OF_WEEK フィールドの値です。
MONDAY	月曜日を示す DAY_OF_WEEK フィールドの値です。
TUESDAY	火曜日を示す DAY_OF_WEEK フィールドの値です。
WEDNESDAY	水曜日を示す DAY_OF_WEEK フィールドの値です。
THURSDAY	木曜日を示す DAY_OF_WEEK フィールドの値です。
FRIDAY	金曜日を示す DAY_OF_WEEK フィールドの値です。
SATURDAY	土曜日を示す DAY_OF_WEEK フィールドの値です。

APIの活用

- Date/Calendarクラスの利用例

```
import java.util.Calendar;
import static java.util.Calendar.*;
import java.util.Date;
import java.util.GregorianCalendar;

public class DateTest {
    public static void main(String[] args) {
        Date d = new Date();
        Calendar c = new GregorianCalendar();
        c.setTime(d);
        c.add(DATE, 90);
        System.out.print(c.get(YEAR) + "年");
        System.out.print((c.get(MONTH) + 1) + "月");
        System.out.println(c.get(DATE) + "日");
    }
}
```

APIの活用

- フォーマッタクラス (java.text パッケージ)
 - SimpleDateFormatクラス
 - 日付データを指定の形式でフォーマットして文字列に変換
 - DateFormatクラスを継承している
 - NumberFormatクラス
 - 数値データを標準の形式でフォーマットして文字列に変換
 - DecimalFormatクラス
 - 数値データを指定の形式でフォーマットして文字列に変換
 - NumberFormatクラスを継承している

APIの活用

- SimpleDateFormatクラスの使い方
 - コンストラクタにフォーマット形式を指定し、インスタンス生成
 - もしくはインスタンス生成後にapplyPatternメソッドで指定
 - formatメソッドで、日付データを与え戻り値でフォーマットされた文字列を取得

APIの活用

- 日付のフォーマット文字列

文字	日付または時刻のコンポーネント	表示	例
G	紀元	テキスト	AD
Y	年	年	1996; 96
M	月	月	July; Jul; 07
w	年における週	数値	27
W	月における週	数値	2
D	年における日	数値	189
d	月における日	数値	10
F	月における曜日	数値	2
E	曜日	テキスト	Tuesday; Tue
a	午前/午後	テキスト	PM
H	一日における時 (0 ~ 23)	数値	0
k	一日における時 (1 ~ 24)	数値	24
K	午前/午後の時 (0 ~ 11)	数値	0
h	午前/午後の時 (1 ~ 12)	数値	12

※ J2SE APIドキュメントより一部抜粋

<http://java.sun.com/j2se/1.4/ja/docs/ja/api/java/text/SimpleDateFormat.html>

APIの活用

- SimpleDateFormatクラスの利用例

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateFormatTest {

    public static void main(String[] args) {
        Date d = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
        System.out.println(sdf.format(d));
    }
}
```

APIの活用

- DecimalFormatクラスもSimpleDateFormatと同じ手順で利用できる
- DecimalFormatではフォーマット文字列以外にメソッドでフォーマットに対する指定を行うことも可能
 - setGroupingUsed(boolean)
 - グループ化(カンマ区切り)を使用するかどうか指定する
 - setMaxFractionDigits(int)
 - 小数部の最大桁数を指定する
 - setMinimumFractionDigits(int)
 - 小数部の最小桁数を指定する

APIの活用

• 数値のフォーマット文字列

記号	位置	意味
0	数値	数字
#	数値	数字。ゼロだと表示されない
.	数値	数値桁区切り子または通貨桁区切り子
-	数値	マイナス記号
,	数値	グループ区切り子
E	数値	科学表記法の仮数と指数を区切る。接頭辞や接尾辞内に引用符を付ける必要はない
;	サブパターン境界	正と負のサブパターンを区切る
%	接頭辞または接尾辞	100 倍してパーセントを表す
¥u00A4	接頭辞または接尾辞	通貨記号で置換される通貨符号。2 つの場合は、国際通貨記号で置換される。パターン内にある場合は、数値桁区切り子ではなく、通貨桁区切り子が使用される
'	接頭辞または接尾辞	接頭辞や接尾辞内の特殊文字を引用符で囲む場合に使用される。たとえば、"'###" を使用すると 123 は "'#123'" にフォーマットされる。単一引用符自体を作成するために合は、1 行に 2 つ引用符を使用する ("# o'clock")

※ J2SE APIドキュメントより一部抜粋

<http://java.sun.com/j2se/1.4/ja/docs/ja/api/java/text/DecimalFormat.html>

APIの活用

- DecimalFormatクラスの利用例

```
import java.text.DecimalFormat;

public class DecimalFormatTest {

    public static void main(String[] args) {
        DecimalFormat df = new DecimalFormat("#,##0-");
        System.out.println(df.format(1000000));
    }
}
```

APIの活用

- 自分でフォーマット文字列を指定しなくても、親クラス (NumberFormatクラス) にあらかじめ用意されたフォーマットを使うと簡単な場合もある
 - NumberFormat.getNumberInstance()
 - 汎用的な数値フォーマットを有するNumberFormat型のインスタンスを取得
 - NumberFormat.getCurrencyInstance()
 - 汎用的な通貨フォーマットを有するNumberFormat型のインスタンスを取得
 - NumberFormat.getPercentInstance()
 - 汎用的なパーセントフォーマットを有するNumberFormat型のインスタンスを取得

APIの活用

- その他のAPI

- ここで紹介した以外にも多くの有用なAPIが用意されている
- 詳細はJavaSEのAPIドキュメントで調べられる
 - <http://java.sun.com/javase/ja/6/docs/ja/>

デバッグの使い方

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

デバッガを使ったデバッグ

- デバッガとは何か
 - デバッグを効率的に行うことの出来るツール
 - Eclipseにも「デバッグ・パースペクティブ」という「パースペクティブ」(画面構成)がありデバッガの機能が含まれている
- Eclipseのデバッガの主な機能
 - ブレークポイントによるプログラムの一時停止機能
 - コードを1行ずつ実行するトレース実行機能
 - 一時停止時の各種変数の内容を確認する機能

デバッガを使ったデバッグ

- デバッガはどんなときに必要か？

種類	内容	デバッガが必要か？
コンパイルエラー	プログラムの文法が誤っている	×
ランタイムエラー	予期せぬトラブルによりプログラムが中断する	○
仕様どおり動かない	エラーはでないが、期待する結果が得られない	◎

Eclipseのデバッガを使ってみよう

- ①ブレークポイントの設置

- 実行時に一時停止したい箇所に、ブレークポイントを設置する
- ソースコードの左端でダブルクリック(または右クリックして「ブレークポイントの切り替え」を選択)すると、ブレークポイントのアイコンが表示される

Eclipseのデバッガを使ってみよう

- ②デバッグモードでプログラムを実行
 - パッケージエクスプローラで実行したいソースを選択し、右クリックして「デバッグ」→「Javaアプリケーション」を選択(または、デバッグ実行のアイコンからも選択可能)
 - 「デバッグ・パースペクティブ」に切り替えますか？とダイアログが出たら「はい」を選択
 - 画面構成が「デバッグ・パースペクティブ」に切り替わる







Eclipseのデバッガを使ってみよう

- ③ブレークポイントでプログラムが一時停止
 - 「変数」ビューで現在の変数の値を確認することもできる

Eclipseのデバッガを使ってみよう

④トレース実行

- 「デバッグ」ビューのボタンバーで、この後の実行を制御することができる

アイコン	名称	内容
	ステップ・オーバー	現在の行を実行し、次の行で一時停止
	ステップ・イン	コンストラクタやメソッドの呼び出し先に移り、一時停止
	ステップ・リターン	コンストラクタやメソッドの呼び出し元に戻って、一時停止
	再開	次のブレークポイントに到達するまで停止せず処理を実行
	停止	実行中のプログラムを強制終了させる
	終了した全ての起動を除去	実行終了したプログラムの情報を「デバッグ」ビューから消去する

デバッガを使ったデバッグ

- デバッガで何を突き止めればよいか
 - デバッガで1行ずつ実行し、状況を確認していくことで、「どこまでが正常」で「どこからが異常」となっているかを発見することが出来る
 - 異常が発生している箇所を発見できれば、バグの修正は非常に容易

その他の文法

株式会社ナレッジエックス
<http://www.knowledge-ex.jp/>

その他の文法

- Enum(列挙型)
- 可変長引数

Enum(列挙型) (1)

- Enumとは(1)
 - 定数を安全かつ厳密に扱うための文法

Enum定義

```
enum 型名 {定数1,定数2,定数3···}
```

Enum(列挙型) (2)

- Enumとは(2)
 - Enumは変数の型名としても利用できる

Enum定数の利用

型名 . 定数名

Enum型変数の定義と初期化(代入部は省略可能)

```
Enum型名 変数名 = 型名 . 定数名;
```


Enum(列挙型) (3)

- Enumを使ったコード例

```
public class EnumTest {  
    public enum Status {MITSUMORI, JUCHU, NOUHIN, SEIKYU, NYUKIN};  
    public static void main(String[] args) {  
        Status s = Status.MITSUMORI;  
  
        if (s == Status.JUCHU) {  
            System.out.println("受注処理");  
        } else if (s == Status.MITSUMORI) {  
            System.out.println("見積処理");  
        }  
    }  
}
```

Enum(列挙型) (4)

- Enumのメリット

- Enumには「型」があり、かつ定数名に対応する値が存在しないため、異なる種類の定数を厳密に区別できる
- 従来よく用いられていた「public static final」修飾された変数の場合は、異なる定数でも代入された値が同じため、意味の異なる定数が誤用される可能性がある

Enum(列挙型) (5)

- Enumを使わず「public static final」で定数を定義したコード例

```
public class EnumTest {  
  
    public static final int DOOR_OPEN = 0;  
    public static final int DOOR_CLOSE = 1;  
    public static final int CONNECTION_ESTABLISH = 0;  
    public static final int CONNECTION_TRANSPORT = 1;  
    public static final int CONNECTION_CLOSE = 2;  
  
    public static void main(String[] args) {  
        int door = DOOR.OPEN;  
        door = CONNECTION_CLOSE;  
    }  
}
```

型が同じなのだと、意味の異なる定数でも同じ変数に代入できてしまう
(=誤用を引き起こしやすい)

Enum(列挙型) (6)

- Enumを使ったコード例

```
public class EnumTest {  
  
    public enum DOOR {OPEN,CLOSE};  
    public enum CONNECTION {ESTABLISH,TRANSPORT,CLOSE};  
  
    public static void main(String[] args) {  
        CONNECTION cnct = DOOR.CLOSE;  
    }  
}
```

定数名が同じでも、異なる
Enum型の定数は代入できな
い(コンパイルエラーとなる)

Enum(列挙型) (7)

- Enumのメソッド
 - Enum型の定数にはメソッドが定義されている
 - Enum型名#values()メソッド
 - そのEnumに定義された定数すべてを配列として返す
 - Enum型名#valueOf(String)メソッド
 - 引数の文字列と一致する定数を返す
 - Enum定数#toString()メソッド
 - その定数の定数名を文字列として取得する
 - System.out.println()の引数にEnum定数を与えると定数名が出力されるが、これは内部でtoString()メソッドを呼んでいるためである

Enum(列挙型) (8)

- Enumの各種メソッドを使ったコード例

```
public class EnumTest {  
  
    public enum CONNECTION {ESTABLISH,TRANSPORT,CLOSE};  
  
    public static void main(String[] args) {  
  
        CONNECTION cnct = CONNECTION.valueOf("CLOSE");  
        String str = cnct.toString();  
        CONNECTION[] array = CONNECTION.values();  
        for(CONNECTION c:array)  
            System.out.println(c);  
  
    }  
  
}
```

可変長引数

- 可変長引数とは

- 引数の数が不定なメソッドを作ることができる
- 可変長引数の部分は、0個以上の任意の数を指定できる
- メソッド内では、可変長引数は配列とみなされる

凡例

戻り値 メソッド名(型名... 引数名)

記述例

```
public void foo(String... a)
```

可変長引数

- 可変長引数を使ったコード例

```
public class VarargsTest {  
  
    public static void main(String[] args) {  
        foo();  
        foo("a", "b", "c");  
    }  
  
    public static void foo(String... a) {  
        System.out.println("foo");  
  
        for(String i:a)  
            System.out.println(i);  
    }  
}
```

可変長引数の変数は配列とみなされるため、拡張for構文が使える

可変長引数

- 可変長引数を使う場合の留意点

- 可変長引数と、オーバーロードの併用は、どちらが呼ばれるかわかりにくくなるので、なるべく避ける
- 実際には、可変長引数のメソッドよりも、引数の数が一致するメソッドの呼び出しが優先される

可変長引数

- 可変長引数とオーバーロードを併用したコード例

```
public class VarargsTest {  
  
    public static void main(String[] args) {  
        foo("a","b","c");  
        foo("a");  
        foo("a","b");  
    }  
  
    public static void foo(String... a) {  
        System.out.println("Varargs");  
    }  
  
    public static void foo(String a) {  
        System.out.println("1arg");  
    }  
  
    public static void foo(String a,String b) {  
        System.out.println("2arg");  
    }  
}
```