

# Javaプログラミング1

## プログラミングの基礎

---

# 目次

|                              |                           |
|------------------------------|---------------------------|
| ➡ 第1章 Javaとは.....5           | ➡ 第5章 条件分岐.....99         |
| ➡ プログラミング言語とは.....6          | ➡ if文.....100             |
| ➡ Javaとは.....11              | ➡ switch文.....120         |
| ➡ 第2章 開発環境のインストール.....19     | ➡ 第6章 繰り返し.....125        |
| ➡ JDKのインストール.....20          | ➡ while文.....126          |
| ➡ サクラエディタのインストール.....28      | ➡ do-while文.....129       |
| ➡ 第3章 プログラムの作成・コンパイル・実行...33 | ➡ for文.....133            |
| ➡ プログラムの作成.....34            | ➡ 繰り返しのネスト.....138        |
| ➡ コンパイルと実行.....46            | ➡ 条件分岐と繰り返しの組み合わせ.....141 |
| ➡ 第4章 変数とデータ型.....53         | ➡ 繰り返し制御.....144          |
| ➡ 変数.....54                  | ➡ 第7章 配列.....149          |
| ➡ データ型.....68                | ➡ 1次元配列.....150           |
| ➡ 標準入力.....74                | ➡ 多次元配列.....161           |
| ➡ 演算.....81                  |                           |
| ➡ 定数.....89                  |                           |
| ➡ 基本データ型の変換(キャスト).....91     |                           |
| ➡ 文字列・基本データ型間の変換.....95      |                           |

# 目次(つづき)

- 第8章 メソッド.....165
  - メソッドとは.....166
  - メソッドの引数・戻り値.....170
  - メソッドのメリット.....177
- 第9章 プログラムの品質.....179
  - 品質の基準.....180
  - プログラミングの原則.....183

# 開発環境など

- ▶ このテキストでは、以下の開発環境を前提としています。
  - ▶ Windows 7(64bit版)
  - ▶ JDK 8

# 第1章

## Javaとは

---

# プログラミング言語とは

---

# コンピュータの3大原則

1. コンピュータは、入力、演算、出力を行う装置である
2. プログラムは、命令とデータの集合体である
3. コンピュータの都合は、人間の感覚と異なる場合がある

※「コンピュータはなぜ動くのか」(矢沢久雄著、日経BP)より引用

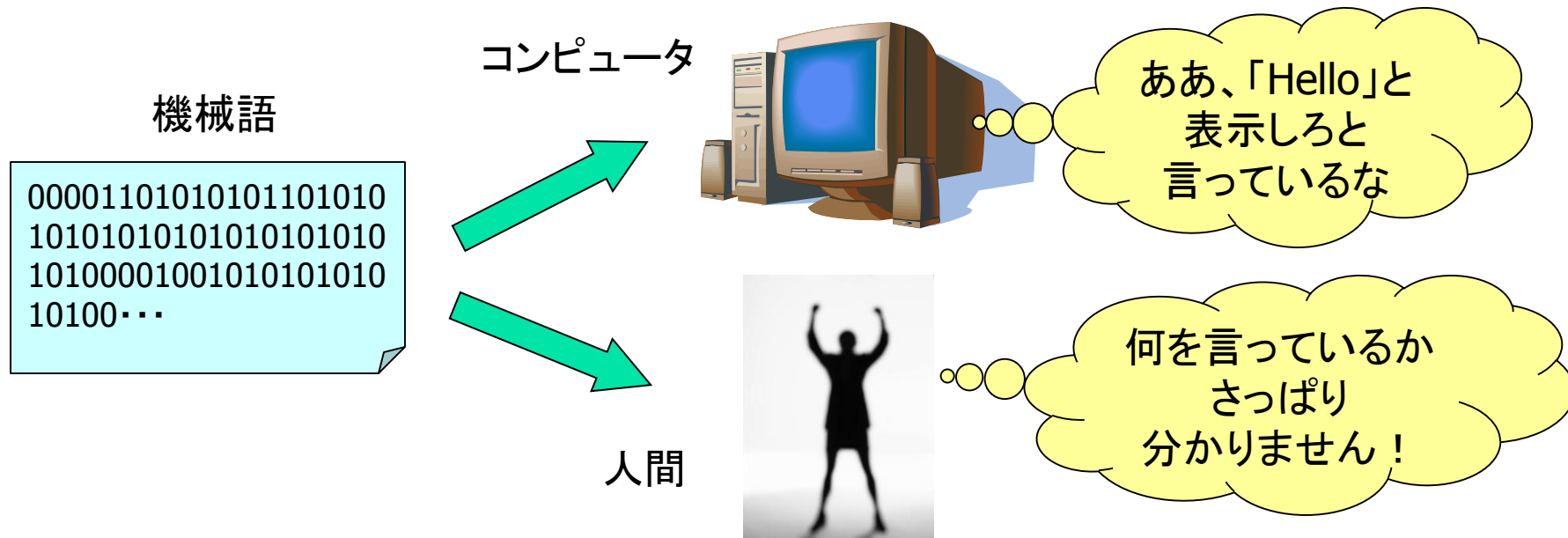
# コンピュータの都合

- ▶ 数値しか理解できない
- ▶ 自分で「考える」ことが出来ない
- ▶ 一から十まで指示されないと分からない



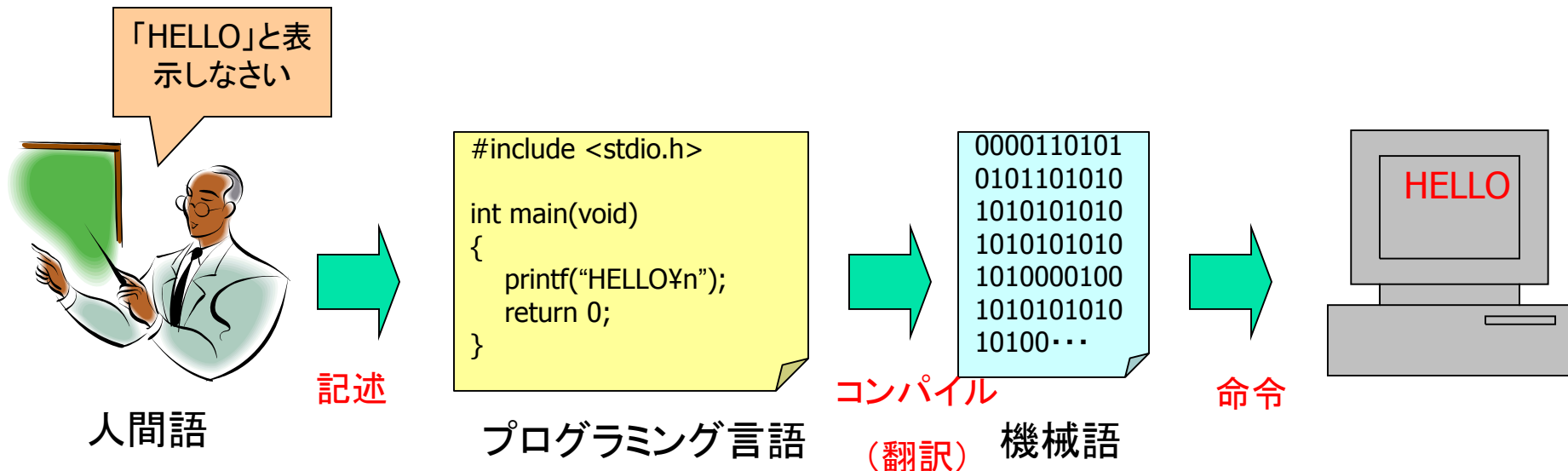
# 機械語

- ▶ コンピュータが理解できる唯一の言語
- ▶ 全て2進数で表現される
- ▶ 人間が直接使うのは非常に難しい



# 「プログラミング言語」とは？

- ▶ 機械語と人間の言葉の中間
- ▶ 人間がコンピュータへの命令・データをプログラミング言語で記述し、それを機械語に翻訳することで、コンピュータを操作する





# Javaとは

# Java言語とは？

---

- ▶ 米SunMicrosystems社のJames Gosling氏が1990年に開発
- ▶ C/C++の文法を参考に作られた

# Java言語の特徴

## ▶ OS非依存

- ▶ Windows、Linux、Solaris、Mac OSなど、様々なOS上で動作する

## ▶ オブジェクト指向

- ▶ 保守性・再利用性の高いプログラムを作成できる

# Java仮想マシン(JVM)とは？

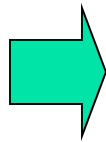
- ▶ 各OS用のJVMが存在する
  - ▶ Windows用、Linux用、Solaris用、Mac OS用など
- ▶ Java言語で記述されたプログラムは、各OS用JVMで共通の「**バイトコード**」にコンパイルされる
- ▶ JVMはバイトコードを解釈し、プログラムを実行する

# CとJavaの違い

## ▼ C言語

```
#include <stdio.h>
int main(void)
{
    printf("HELLO\n");
    return 0;
}
```

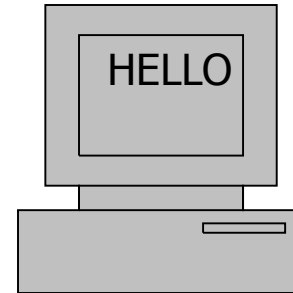
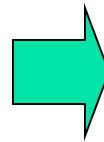
C言語



```
0000110101
0101101010
1010101010
1010101010
1010000100
1010101010
10100...
```

機械語

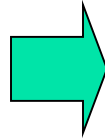
各OSで、解釈できる機械語が違う



## ▼ Java言語

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

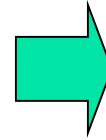
Java言語



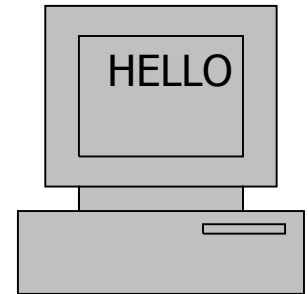
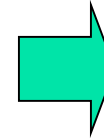
```
0000110101
0101101010
1010101010
1010101010
1010000100
1010101010
10100...
```

バイトコード

各OS用JVMで共通



JVM  
が実行



# Javaの用途

## ▶ サーバーサイド

- ▶ Webアプリケーションにおいて、サーバー側で動くプログラム

## ▶ 組み込み

- ▶ 携帯電話アプリ、Androidアプリ、家電製品内のプログラムなど



# Javaのバージョン

| 年    | バージョン       | 備考                                     |
|------|-------------|--|
| 1996 | Java 1.0    |  |
| 1997 | Java 1.1    |  |
| 1998 | Java 1.2    | 1999年に「Java 2」と改称                      |
| 1999 | Java 2      |  |
| 2000 | J2SE 1.3    |  |
| 2002 | J2SE 1.4    |  |
| 2004 | Java SE 5.0 | ジェネリクス、拡張for文、アノテーションなどの追加             |
| 2006 | Java SE 6.0 | 2009年にOracle社がSun Microsystems社を買収     |
| 2011 | Java SE 7.0 | try-with-resources、NIO.2などの追加          |
| 2014 | Java SE 8.0 | ラムダ式、Stream API、Date and Time APIなどの追加 |

# Javaのエディション

- ▶ Java SE (Standard Edition)

  - ▶ 標準版

- ▶ Java EE (Enterprise Edition)

  - ▶ 企業の基幹システムなどの大規模システム向け

- ▶ Java ME (Micro Edition)

  - ▶ モバイル、組み込み(家電製品内のプログラムなど)向け

## 第2章

# 開発環境のインストール

---

# JDKのインストール

---

# JDKとは

---

- ▶ Java Development Kit (Java開発キット) の略
- ▶ Javaのコンパイラなど、開発に最低限必要なものがそろっている

# インストーラーのダウンロード

▼ 下記のWebサイトにアクセス

▼ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

▼ [Java SE 8u31]-[JDK]の[Download]をクリック

※ダウンロード時の最新バージョンが最上位に表示される



# インストーラーのダウンロード

- ▶ [Java SE Development Kit 8u31]の[Accept License Agreement]にチェックを入れ、使用しているOSに合ったものをクリック
- ▶ ダウンロードが開始するので、適当なフォルダに保存

| Java SE Development Kit 8u31  |           |   |
|---|-----------|---|
| You must accept the <a href="#">Oracle Binary Code License Agreement for Java SE</a> to download this software. |           |   |
| <input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement       |           |   |
| Product / File Description  | File Size | Download  |
| Linux x86   | 135.24 MB | <a href="#">jdk-8u31-linux-i586.rpm</a>         |
| Linux x86   | 154.91 MB | <a href="#">jdk-8u31-linux-i586.tar.gz</a>      |
| Linux x64   | 135.62 MB | <a href="#">jdk-8u31-linux-x64.rpm</a>          |
| Linux x64   | 153.45 MB | <a href="#">jdk-8u31-linux-x64.tar.gz</a>       |
| Mac OS X x64  | 209.17 MB | <a href="#">jdk-8u31-macosx-x64.dmg</a>         |
| Solaris SPARC 64-bit (SVR4 package)   | 136.91 MB | <a href="#">jdk-8u31-solaris-sparcv9.tar.Z</a>  |
| Solaris SPARC 64-bit  | 97.11 MB  | <a href="#">jdk-8u31-solaris-sparcv9.tar.gz</a> |
| Solaris x64 (SVR4 package)  | 137.51 MB | <a href="#">jdk-8u31-solaris-x64.tar.Z</a>      |
| Solaris x64   | 94.82 MB  | <a href="#">jdk-8u31-solaris-x64.tar.gz</a>     |
| Windows x86   | 157.96 MB | <a href="#">jdk-8u31-windows-i586.exe</a>       |
| Windows x64   | 170.36 MB | <a href="#">jdk-8u31-windows-x64.exe</a>        |

# インストールの実行

- ▶ ダウンロードしたインストーラーを実行
- ▶ 設定などは全てデフォルト(選択肢は全て[次へ])
- ▶ 図のような画面が表示されれば完了

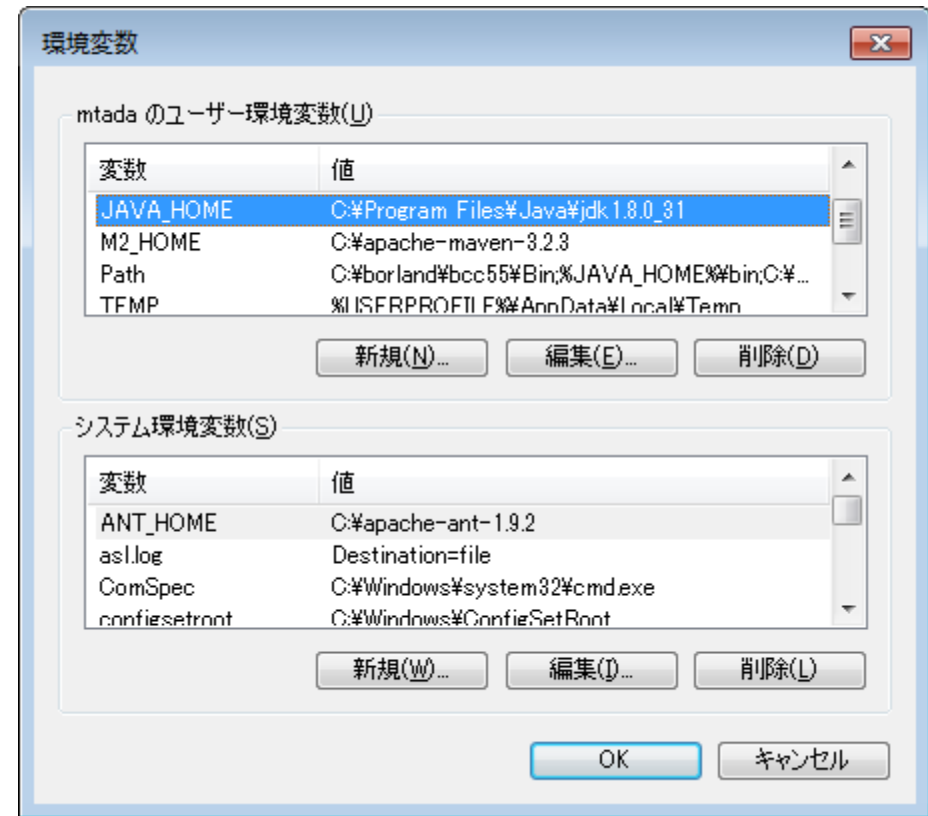


※完了後、ブラウザが開きユーザー登録を求められるが、必須ではない



# 環境変数の設定

- ▶ スタートメニューの[コンピュータ]を右クリック→[プロパティ]をクリック→[システムの詳細設定]をクリック
- ▶ [システムのプロパティ]画面の[詳細設定]タブ→[環境変数]をクリック



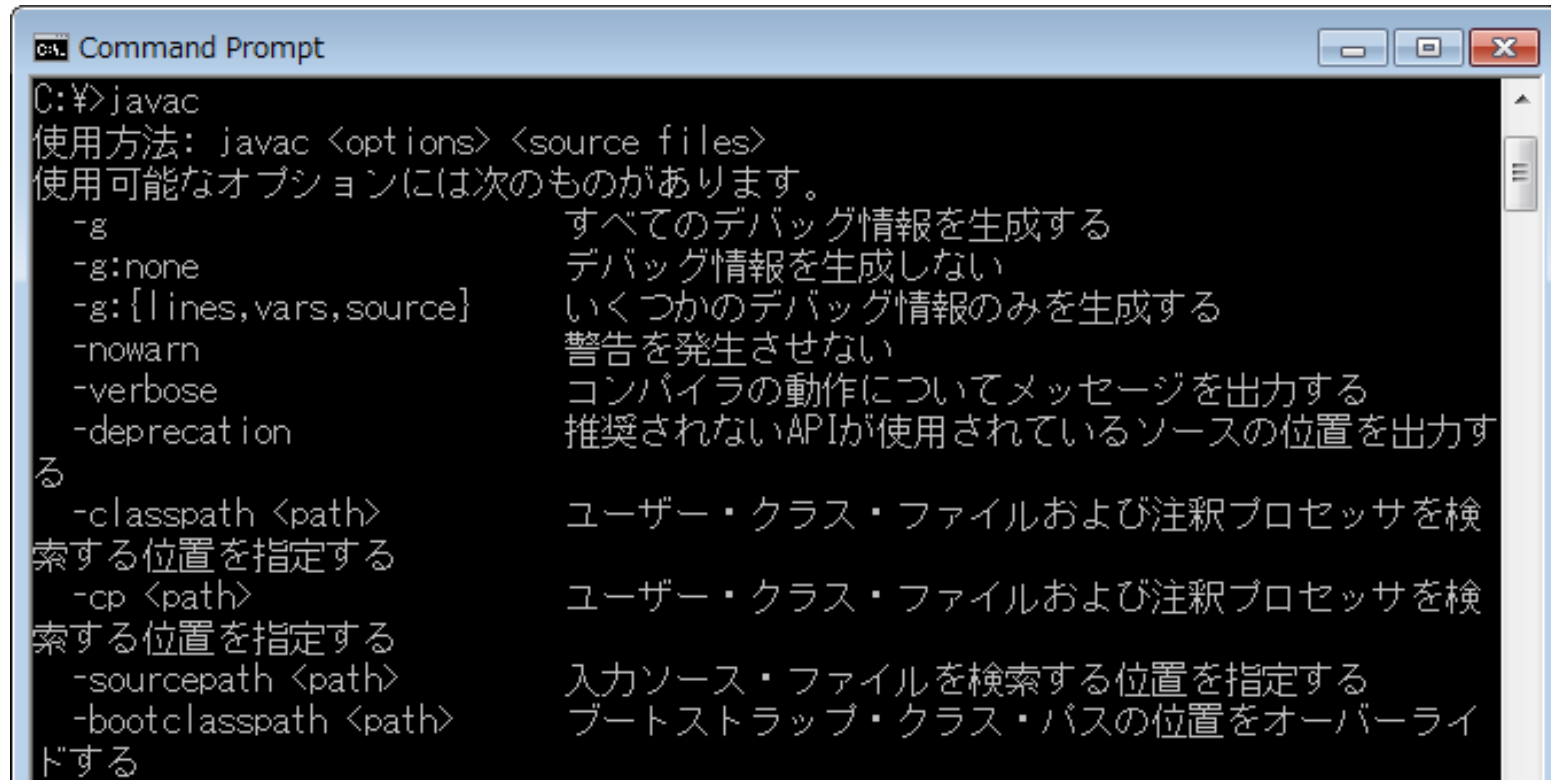
# 環境変数の設定

- ▶ [〇〇のユーザー環境変数]-[新規]をクリックし、以下2つの環境変数を追加→[OK]をクリック

| 変数名       | 変数値                               |
|-----------|-----------------------------------|
| JAVA_HOME | C:¥Program Files¥Java¥jdk1.8.0_31 |
| PATH      | %JAVA_HOME%¥bin                   |

# インストール完了の確認

- ▶ コマンドプロンプトで「javac」コマンドを実行
- ▶ 図のようなメッセージが表示されればOK



```

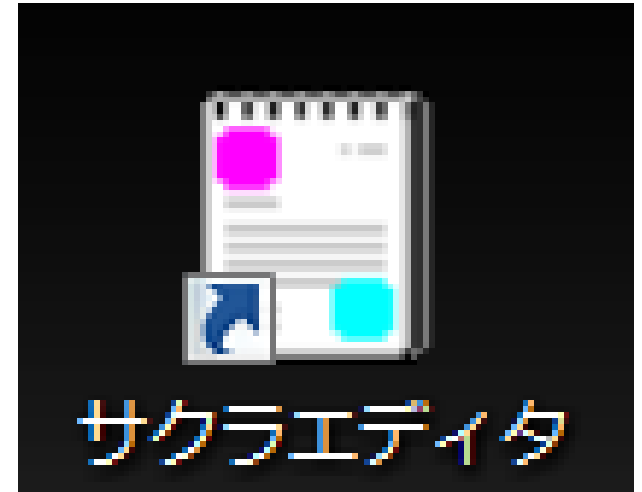
C:\¥>javac
使用方法: javac <options> <source files>
使用可能なオプションには次のものがあります。
-g                  すべてのデバッグ情報を生成する
-g:none            デバッグ情報を生成しない
-g:{lines,vars,source} いくつかのデバッグ情報のみを生成する
-nowarn            警告を発生させない
-verbose           コンパイラの動作についてメッセージを出力する
-deprecation       推奨されないAPIが使用されているソースの位置を出力する
-classpath <path>   ユーザー・クラス・ファイルおよび注釈プロセッサを検索する位置を指定する
-cp <path>          ユーザー・クラス・ファイルおよび注釈プロセッサを検索する位置を指定する
-sourcepath <path>   入力ソース・ファイルを検索する位置を指定する
-bootclasspath <path> ブートストラップ・クラス・パスの位置をオーバーライドする

```

# サクラエディタのインストール

# サクラエディタとは

- ▶ オープンソースのテキストエディタ
- ▶ 無料で使用できる
- ▶ 各プログラミング言語に合った表示ができる



# インストーラーのダウンロード

- ➡ 下記のWebサイトにアクセス
  - ➡ <http://sakura-editor.sourceforge.net/download.html>
- ➡ [最新版ダウンロード]をクリック
- ➡ ダウンロードが開始するので、適当なフォルダに保存

**V2(UNICODE版)**

サクラエディタの **Version 2** です。  
内部データを **UNICODE** で保持し、**SJIS** で表現できない文字も扱えます。  
日本語版 **Windows 2000/XP/Vista/7** で動作します。

最新版ダウンロード [sinst2-0-5-0.exe](#) (2.0.5.0)

**主な新規機能**

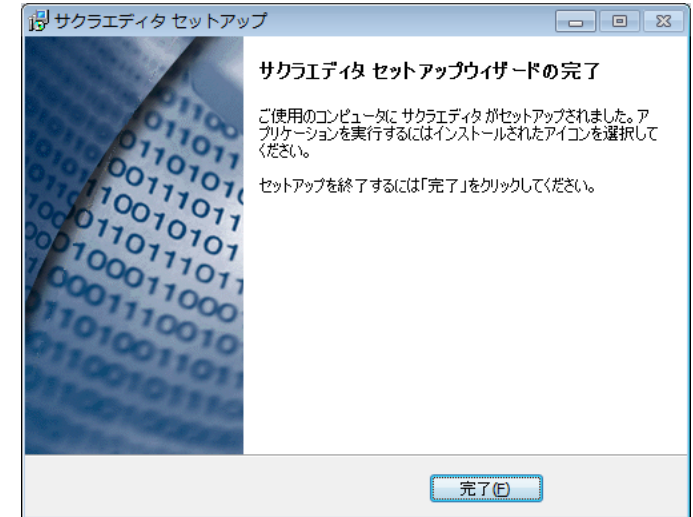
- タブバーのフォント指定
- 保存時に改行コードの混在を警告する
- ステータスバーのカーソル位置文字コード表示欄ダブルクリックでエンコード別コード表示ダイアログを出す

詳細は[更新履歴](#)を御確認ください。

1つ前 [sinst2-0-4-0.exe](#) (2.0.4.0)

# インストールの実行

- ▶ ダウンロードしたインストーラーを実行
- ▶ [デスクトップにアイコン作成]と[「SAKURAで開く」メニューの追加]にチェックを入れる
- ▶ 図のような画面が表示されれば完了







# 第3章 プログラムの 作成・コンパイル・実行

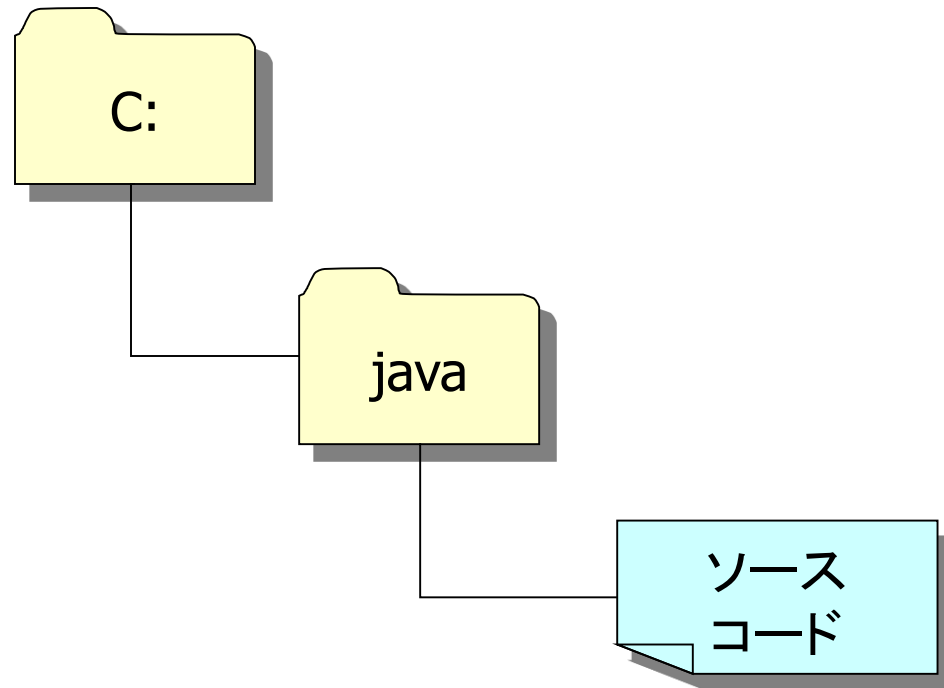
---

# プログラムの作成

---

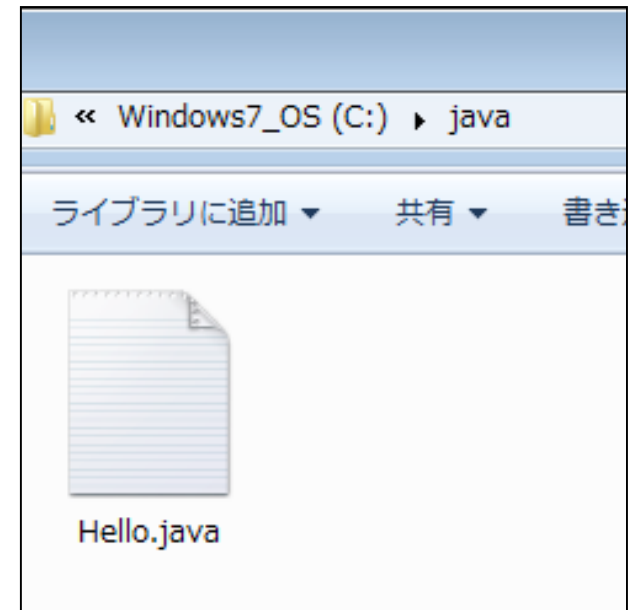
# フォルダの作成

- ▶ Cドライブ直下に「java」フォルダを作成
  - ▶ 以降、ソースコードは全て上記のフォルダ内に作成



# ソースコードの作成

- ▶ C:\javaフォルダ内で右クリック
  - [新規作成]-[テキスト ドキュメント]をクリック
  - ファイル名を「Hello.java」と変更  
(拡張子は必ず「.java」)



# ソースコードの作成

- ▶ Hello.javaを右クリック→[SAKURAで開く]をクリック→以下のようなコードを作成・保存

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

# インデント例

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

→ :TAB1つ    | :半角スペース1つ

# インデントしていない例

- ▶ クラスやメソッドなどのかたまりが分かりづらく、読みにくい

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

# Javaプログラムの構造

クラス



メソッド



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

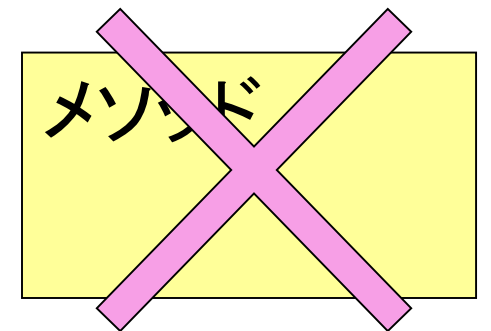
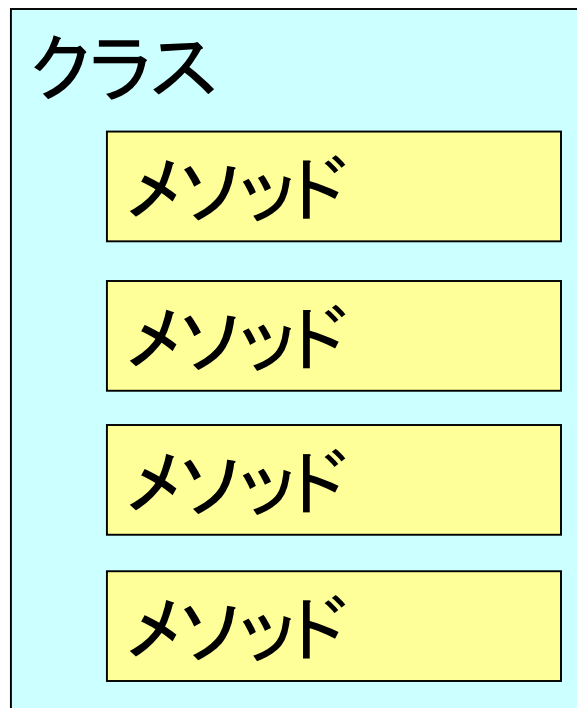
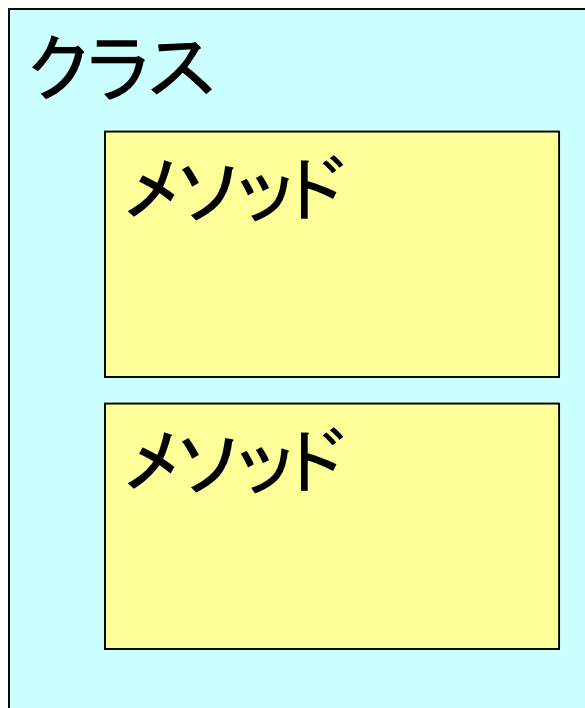


# クラス

- ▶ Javaプログラムの最小単位
- ▶ クラス名の先頭文字は大文字にするのが慣習
- ▶ ファイル名は「クラス名.java」とするのが基本
- ▶ 1ファイルに1クラスが基本

# メソッド

- ▶ 行いたい具体的処理を記述する
- ▶ 全てのメソッドはクラスに属する



クラスに属さないメソッド  
は存在しない

# main()メソッド

---

- ▶ プログラム実行時には、mainメソッドから処理が実行される

# System.out.println()メソッド

- ▶ 文字列を表示して改行するメソッド
- ▶ Javaに標準で用意されている
- ▶ System.out.print()メソッドを利用すると改行しない

# コメント

▶ コメント＝プログラム内の説明書き

▶ コンパイル時には無視される→処理には無関係

▶ 書き方

▶ 一行コメント `// コメントです`

▶ 範囲コメント `/* コメントです */`

# コンパイルと実行

---

# コンパイル(javacコマンド)

- ▶ コマンドプロンプトで以下のように実行
  - ▶ cdコマンドで、ソースコードのあるフォルダに移動
  - ▶ 「javac <ソースファイル名>」と実行

## 実行結果

```
C:\¥java>javac Hello.java
```

# コンパイルエラー

- ▶ プログラムに誤りがあった場合、コンパイル時にエラーメッセージが表示される

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

セミicolon忘れ

実行結果

```
C:\java>javac Hello.java  
Hello.java:4: エラー: ';'がありません  
    System.out.println("World")  
                        ^
```

エラー1個



# 起きやすいコンパイルエラー

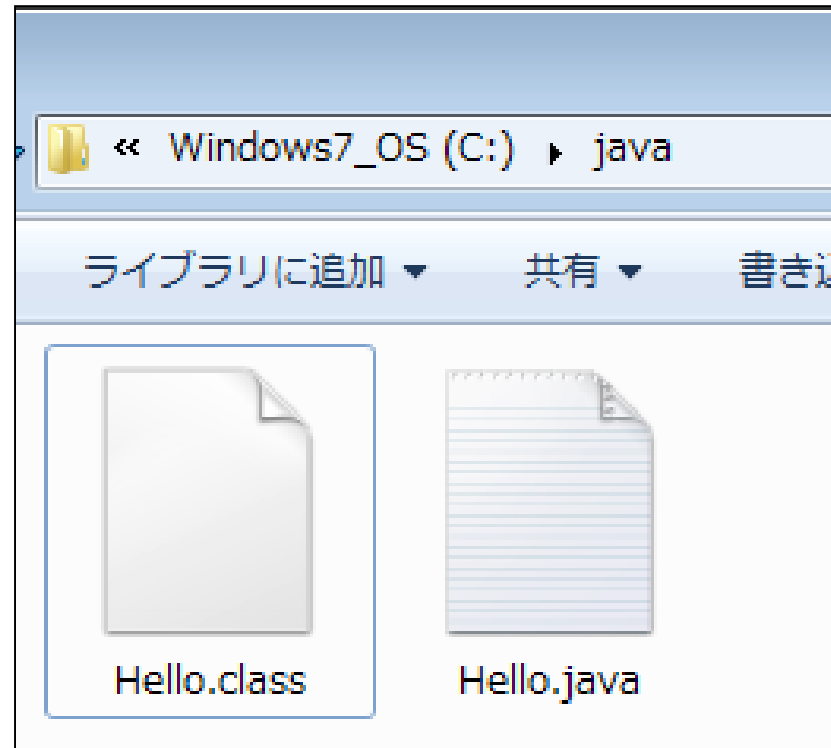
- ▶ 波カッコ{} のとじ忘れ
- ▶ ダブルクォテーション“ ” のとじ忘れ
- ▶ 文末のセミicolon ; の忘れ
- ▶ 大文字・小文字の違い
- ▶ “ ” 内以外での全角文字の使用
  - ▶ 特に全角スペース(空白)は要注意

これらのミスをなくすだけでも、  
飛躍的にプログラミングが上達します！



# クラスファイル

- ▶ コンパイルに成功すると、「**クラス名.class**」というファイル(**クラスファイル**)が作成される



# 実行 (javaコマンド)

- ▶ コマンドプロンプトで以下のように実行
  - ▶ cdコマンドで、ソースコードのあるフォルダに移動
  - ▶ 「**java <クラス名>**」と実行

## 実行結果

```
C:\¥java>java Hello  
  
Hello  
  
World
```

# プログラムの流れ

- ▶ メソッドに記述した処理が、上から順番に実行される

プログラム

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

実行結果

```
Hello  
World
```

# 第4章

## 変数とデータ型

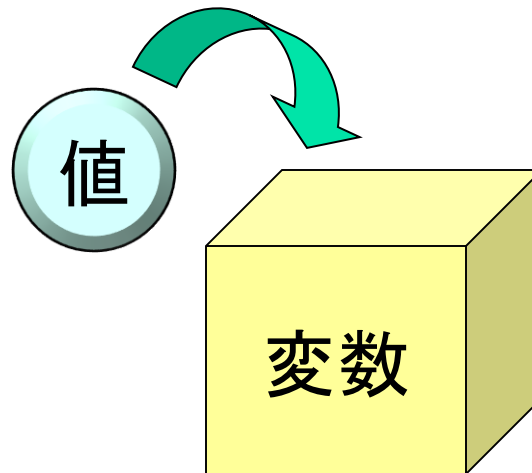
---



# 変数

# 変数とは

- ▶ プログラム内でデータ(値)を保存しておくための、メモリ上の「箱」のようなもの



# サンプルプログラム

```
class VarSample1 {  
    public static void main(String[] args) {  
        int num;  
        num = 10;  
        System.out.println("num = " + num);  
    }  
}
```

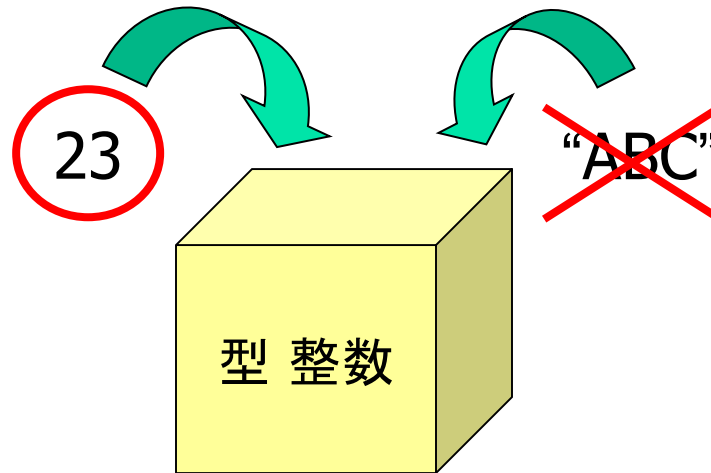
実行結果

```
C:\¥java>java VarSample1  
num = 10
```



# データ型とは

- ▶ データの種類（整数、小数、文字列など）
- ▶ 変数には必ずデータ型の指定が必要
- ▶ 指定したデータ型以外の値は、変数に代入できない



# int型

---

▶ 整数を表すデータ型

# 変数名

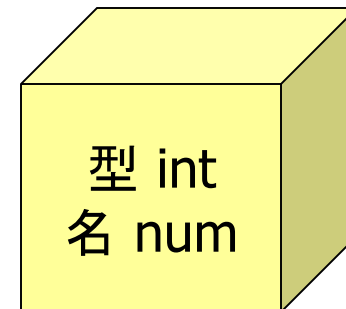
---

- ▼ 変数に付ける名前
- ▼ メソッド内での名前の重複は禁止

# 変数の宣言

- ▶「こんなデータ型の、こういう名前の変数を使います」ということを明示する
- ▶C言語と違い、メソッドのどの部分にでも記述できる

```
// 宣言  
int num;
```



# 値の代入

- ▶ 変数に値を代入する際は  
= (イコール) 演算子を利用する
  - ▶ 「等しい」ではない！
- ▶ 変数に最初に値を代入することを初期化という
- ▶ 宣言と初期化は同時に行うことも可能

```
// 宣言  
int num;  
  
// 値の代入(初期化)  
num = 10;  
  
// 10 = num; は×
```

```
// 宣言と初期化を同時に  
int num = 10;
```

# 変数名のつけ方(規約)

- ▶ 以下の規約を守らない場合、コンパイルエラーとなる
  - ▶ 使える文字は英数字、アンダーバー、ドルマーク(すべて半角)
  - ▶ 先頭文字は数字以外
  - ▶ Javaの予約語と同じ変数名は使えない
    - ▶ int、class、public、static、void、など
  - ▶ 同一メソッド内で、同じ変数名は使えない
  - ▶ 大文字・小文字は区別される

# 変数名のつけ方(慣習)

- ▶ 以下の慣習を守らなくてもコンパイルエラーにはならないが、大概は従う
  - ▶ 先頭文字は小文字
  - ▶ 2つ以上の単語をつなげる場合、2単語目以降の頭文字は大文字(studentNameなど)

# 変数名のつけ方(その他)

- ▶ その変数の意味が分かるような名前を付ける  
(年齢→age、氏名→nameなど)
- ▶ 英単語は略さず付ける  
(生徒→○ student × stdnt)



# 変数への上書き代入

- ▶ 変数に新しい値を代入すると、古い値は削除される

```
class VarSample2 {  
    public static void main(String[] args) {  
        int num;  
        num = 10;  
        System.out.println("num = " + num);  
        num = 20; // 再代入  
        System.out.println("num = " + num);  
    }  
}
```

実行結果

```
C:¥java>java VarSample2  
num = 10  
num = 20
```

# 複数の変数の利用

▼ 変数は複数利用  
できる

```
class VarSample3 {  
    public static void main(String[] args) {  
        int num1 = 10;  
        System.out.println("num1 = " + num1);  
        int num2 = 20;  
        System.out.println("num2 = " + num2);  
    }  
}
```

実行結果

```
C:\¥java>java VarSample3  
num1 = 10  
num2 = 20
```

# 2つの変数の値を交換するプログラム

```
class VarSample4 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        System.out.println("a = " + a + ", b = " + b);  
        int temp = a; // 一時退避用の変数  
        a = b;  
        b = temp;  
        System.out.println("a = " + a + ", b = " + b);  
    }  
}
```

実行結果

```
C:\¥java>java VarSample4  
a = 10, b = 20  
a = 20, b = 10
```



# データ型

# Javaの基本データ型(プリミティブ型)

| 種類  | 型名      | バイト数など              |
|-----|---------|---------------------|
| 論理値 | boolean | trueまたはfalseのみ      |
| 文字  | char    | 2バイトUnicode文字→実体は整数 |
| 整数  | byte    | 1バイト整数              |
|     | short   | 2バイト整数              |
|     | int     | 4バイト整数              |
|     | long    | 8バイト整数              |
| 小数  | float   | 4バイト浮動小数            |
|     | double  | 8バイト浮動小数            |

# String型

- ▶ 文字列を表すデータ型
- ▶ ダブルクォテーション "" で囲うと、文字列 (String 型) と認識される
- ▶ String型はプリミティブ型ではない

# サンプルプログラム

```
class TypeSample1 {  
    public static void main(String[] args) {  
        int i = 100;  
        double d = 3.14;  
        char c = 'あ'; // charはシングルクォテーション  
        String str = "ほげほげ"; // Stringはダブルクォテーション  
        System.out.println("i = " + i);  
        System.out.println("d = " + d);  
        System.out.println("c = " + c);  
        System.out.println("str = " + str);  
    }  
}
```

実行結果

```
C:\¥java>java TypeSample1  
i = 100  
d = 3.14  
c = あ  
str = ほげほげ
```

# 文字列の結合

▶ + 演算子で文字列を結合できる

▶ 異なるデータ型とStringの結合も可能

```
class TypeSample2 {  
    public static void main(String[] args) {  
        String s1 = "あいう" + "かきく";  
        String s2 = "値は" + 5;  
        String s3 = "和は" + (10 + 20);  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println(s3);  
    }  
}
```

実行結果

```
C:¥java>java TypeSample2  
あいうかきく  
値は5  
和は30
```



# エスケープシーケンス

## ◀ 改行など特殊な文字を表す

| エスケープシーケンス | 意味          |
|------------|-------------|
| ¥n         | 改行          |
| ¥t         | タブ          |
| ¥u         | 16進ユニコード    |
| ¥'         | シングルクォテーション |
| ¥"         | ダブルクォテーション  |
| ¥¥         | ¥ 記号        |

```
class TypeSample3 {  
    public static void main(String[] args) {  
        String str = "あいう¥nかきく";  
        System.out.println(str);  
    }  
}
```

実行結果

```
C:¥java>java TypeSample3  
あいう  
かきく
```



# 標準入力

---

# 整数の入力

## ScannerクラスのnextIntメソッドを利用する

```
import java.util.Scanner;

class ScannerSample1 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int num = s.nextInt();
        System.out.println("num = " + num);
    }
}
```

実行結果

```
C:\¥java>java ScannerSample1
10 (キーボードから入力)
num = 10
```

# 注意点

- ▶ クラス記述より上部に「`import java.util.Scanner`」というインポート宣言を記述しなければならない
- ▶ Scannerクラスを利用するために必要

# InputMismatchException

- ▶ nextInt()実行時に、整数以外の文字列が入力された際に発生する例外

```
System.out.print("入力→");  
Scanner s = new Scanner(System.in);  
int num = s.nextInt();
```

## 実行結果

入力→あああ（キーボードから入力）

```
Exception in thread "main" java.util.InputMismatchException  
at java.util.Scanner.throwFor(Unknown Source)  
at java.util.Scanner.next(Unknown Source)  
at java.util.Scanner.nextInt(Unknown Source)  
at java.util.Scanner.nextInt(Unknown Source)  
at .main(Foo.java:11)
```

# 小数の入力

## ScannerクラスのnextDoubleメソッドを利用する

```
import java.util.Scanner;

class ScannerSample2 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        double d = s.nextDouble();
        System.out.println("d = " + d);
    }
}
```

実行結果

```
C:\¥java>java ScannerSample2
3.14 (キーボードから入力)
d = 3.14
```

# 文字列の入力

## Scannerクラスのnextメソッドを利用する

```
import java.util.Scanner;

class ScannerSample3 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String str = s.next();
        System.out.println("str = " + str);
    }
}
```

実行結果

```
C:\¥java>java ScannerSample3
あいうえお (キーボードから入力)
str = あいうえお
```

# 文字列の入力(スペースを入力したい場合)

## ScannerクラスのnextLineメソッドを利用する

```
import java.util.Scanner;

class ScannerSample4 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String str = s.nextLine();
        System.out.println("str = " + str);
    }
}
```

実行結果

```
C:\¥java>java ScannerSample4
あいう かきく (キーボードから入力)
str = あいう かきく
```





# 演算

# 算術演算子

| 演算子 | 意味  | 例                                   |
|-----|-----|-------------------------------------|
| +   | 足し算 | $a = 7 + 3$ (aは10になる)               |
| -   | 引き算 | $a = 7 - 3$ (aは4になる)                |
| *   | 掛け算 | $a = 7 * 3$ (aは21になる)               |
| /   | 割り算 | $a = 7 / 3$ (aは2になる)                |
| %   | 剰余算 | $a = 7 \% 3$ (7を3で割った余り<br>→aは1になる) |

# サンプルプログラム

```
class OperationSample1 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        int sum = a + b;  
        System.out.println("和は" + sum);  
    }  
}
```

実行結果

```
C:¥java>java OperationSample1  
和は30
```

# ArithmeticException

◀ 0での割り算を行った際に発生する例外

```
int num = 2 / 0;
```

実行結果

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Foo.main(Foo.java:3)
```

# 代入演算子

- ▶ aに2を加える →  $a = a + 2$  または  $a += 2$
- ▶ aに2を減じる →  $a = a - 2$  または  $a -= 2$
- ▶ aに2をかける →  $a = a * 2$  または  $a *= 2$
- ▶ aを2で割る →  $a = a / 2$  または  $a /= 2$
- ▶ aを2で割ったときの余りをaに代入する  
→  $a = a \% 2$  または  $a \% = 2$

# サンプルプログラム

```
class OperationSample2 {  
    public static void main(String[] args) {  
        int a = 10;  
        a = a + 2; // a += 2と同様  
        System.out.println("a = " + a);  
    }  
}
```

実行結果

```
C:\¥java>java OperationSample2  
a = 12
```

# インクリメントとデクリメント

▶ インクリメント(変数の値を1増やす)

▶  $a = a + 1$  または  $a++$  ( $++a$ )

▶ デクリメント(変数の値を1減らす)

▶  $a = a - 1$  または  $a--$  ( $--a$ )

# サンプルプログラム

```
class OperationSample3 {  
    public static void main(String[] args) {  
        int a = 10;  
        a++; // a += 1と同様  
        System.out.println("a = " + a);  
    }  
}
```

実行結果

```
C:\¥java>java OperationSample3  
a = 11
```





# 定数

# final修飾子

- ▶ 宣言時に**final修飾子**を付けた変数は、**値の変更が出来なくなる**(=定数になる)
  - ▶ 値を変更しようとするときコンパイルエラーになる

```
class ConstSample1 {  
    public static void main(String[] args) {  
        final int num = 100;  
        // この後「num = 200;」とするとコンパイルエラー  
        System.out.println("num = " + num);  
    }  
}
```

実行結果

```
C:\¥java>java ConstSample1  
num = 100
```

# 基本データ型の変換(キャスト)

# キャストとは

- ▶ データ型を変換すること
- ▶ (型名)で変換できる

```
class CastSample1 {  
    public static void main(String[] args) {  
        int i1 = 100;  
        long l = (long) i1;  
        System.out.println("l = " + l);  
        double d = 3.14;  
        int i2 = (int) d;  
        System.out.println("i2 = " + i2);  
    }  
}
```

実行結果

```
C:\¥java>java CastSample1  
l = 100  
i2 = 3
```

# 暗黙的キャスト・明示的キャスト

- ▶ 範囲が小さい型から範囲が大きい型へのキャスト(アップキャスト)は、暗黙的に行われる

- ▶ 例

- ```
int i = 500;
```

- ```
long l = i; // 暗黙的に型変換が行われる
```

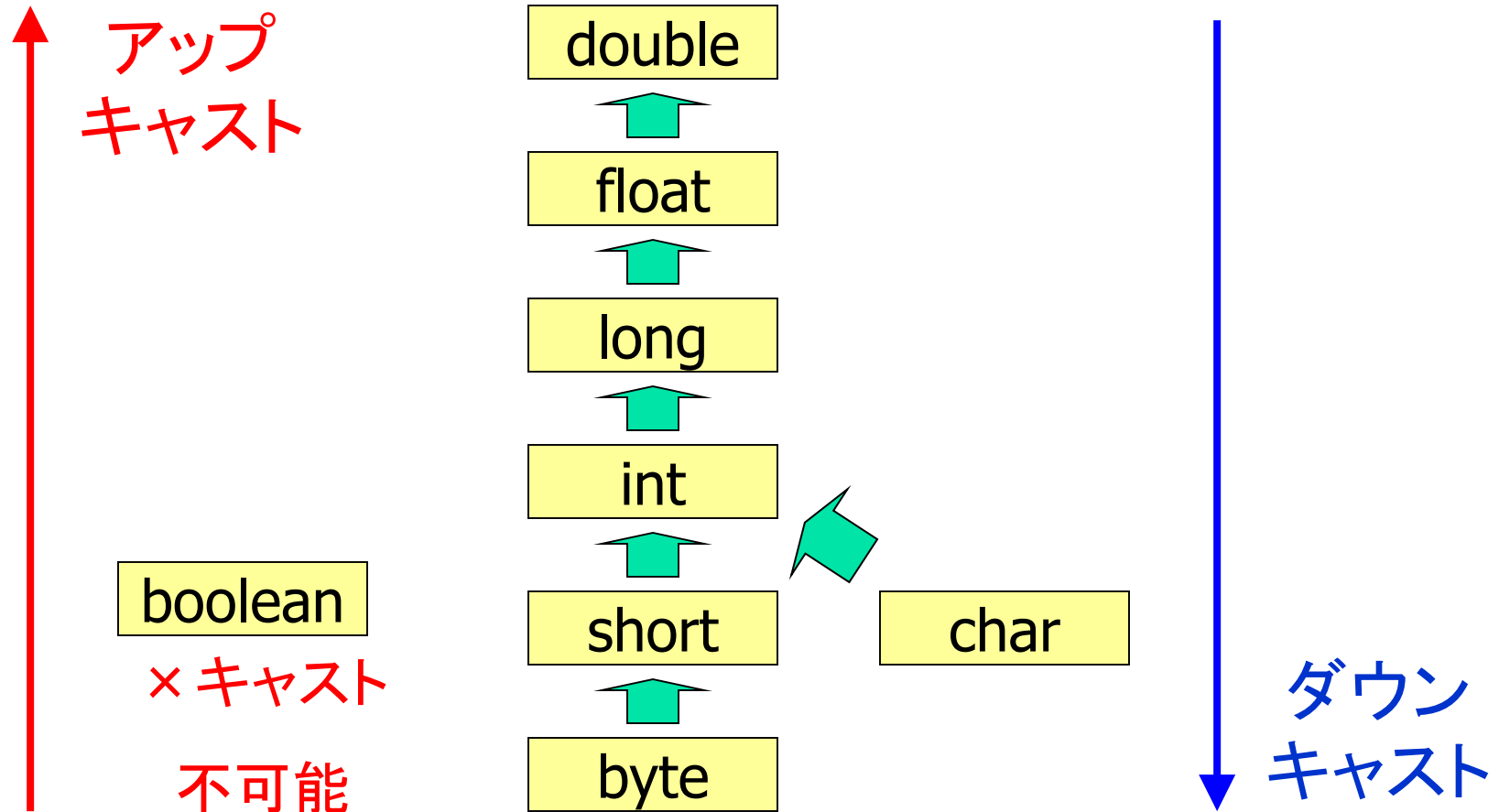
- ▶ 範囲が大きい型から範囲が小さい型へのキャスト(ダウンキャスト)は、明示しなければコンパイルエラーになる

- ▶ 例

- ```
long l = 1000L;
```

- ```
int i = l; // コンパイルエラー。int i = (int) l と書けばOK
```

# 基本データ型のキャスト順位



# 文字列・基本データ型間の変換

# 文字列→基本データ型

- Integerクラスの  
parseIntメソッド、およびDoubleクラスの  
parseDoubleメソッド  
を利用する

```
class ParseSample1 {  
    public static void main(String[] args) {  
        // 文字列を整数に変換  
        String s1 = "12345";  
        int i = Integer.parseInt(s1);  
        System.out.println("i = " + i);  
        // 文字列を小数に変換  
        String s2 = "3.14";  
        double d = Double.parseDouble(s2);  
        System.out.println("d = " + d);  
    }  
}
```

実行結果

```
C:\java>java ParseSample1  
i = 12345  
d = 3.14
```



# NumberFormatException

- ▶ 整数(または小数)でない文字列を変換した際に発生する例外

```
int num = Integer.parseInt("あああ");
```

## 実行結果

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "あああ"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at Foo.main(Foo.java:8)
```

# 基本データ型→文字列

## ▶ StringクラスのvalueOf メソッドを利用する

```
class ParseSample2 {  
    public static void main(String[] args) {  
        // 整数を文字列に変換  
        int i = 12345;  
        String s1 = String.valueOf(i);  
        System.out.println("s1 = " + s1);  
        //小数を文字列に変換  
        double d = 3.14;  
        String s2 = String.valueOf(d);  
        System.out.println("s2 = " + s2);  
    }  
}
```

実行結果

```
C:\¥java>java ParseSample2  
s1 = 12345  
s2 = 3.14
```

# 第5章

## 条件分岐

---



if文

# if文のサンプルプログラム

```
class IfSample1 {  
    public static void main(String[] args) {  
        int age = 22;  
        System.out.println("あなたの年齢:" + age);  
        if (age >= 20) {  
            System.out.println("成年です。");  
        }  
        System.out.println("終了します。");  
    }  
}
```

実行結果(ageが20以上の場合)

```
C:\java>java IfSample1
```

```
あなたの年齢:22
```

```
成年です。
```

```
終了します。
```

実行結果(ageが20未満の場合)

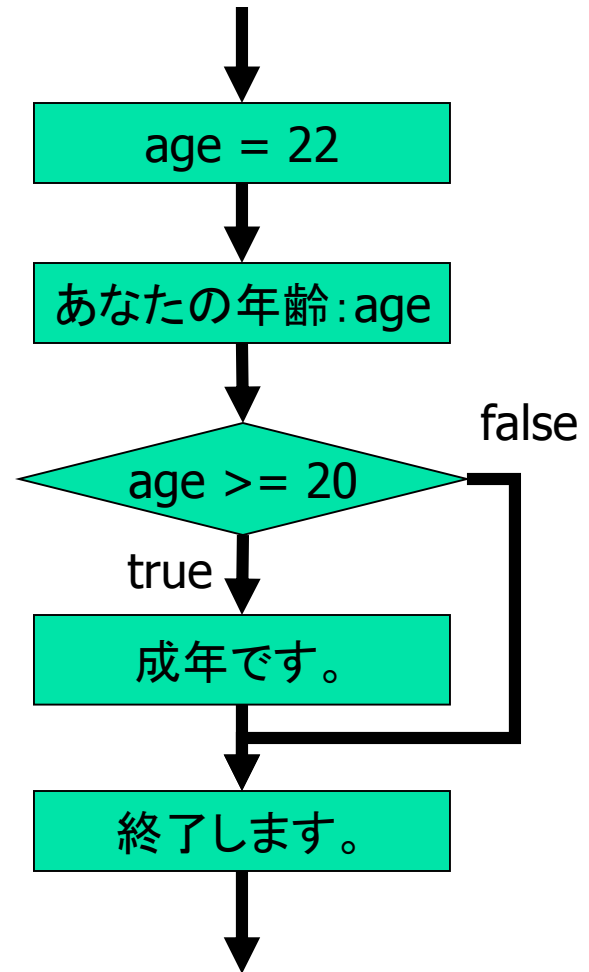
```
C:\java>java IfSample1
```

```
あなたの年齢:18
```

```
終了します。
```

# フローチャート

```
int age = 22;  
System.out.println("あなたの年齢:"  
    + age);  
if (age >= 20) {  
    System.out.println("成年です。");  
}  
System.out.println("終了します。");
```



# 関係演算子

| 関係演算子      | 意味              |
|------------|-----------------|
| $a > b$    | aがbより大きいときtrue  |
| $a < b$    | aがbより小さいときtrue  |
| $a \geq b$ | aがb以上のときtrue    |
| $a \leq b$ | aがb以下のときtrue    |
| $a == b$   | aとbが等しいときtrue   |
| $a != b$   | aとbが等しくないときtrue |

# 文字列の比較

▶ equals()メソッドを利用する

×

```
String str = "ほげほげ";  
if (str == "ふがふが") {  
    ...  
}
```

○

```
String str = "ほげほげ";  
if (str.equals("ふがふが")) {  
    ...  
}
```



# if-else文のサンプルプログラム

```
class IfSample2 {  
    public static void main(String[] args) {  
        int age = 22;  
        System.out.println("あなたの年齢:" + age);  
        if (age >= 20) {  
            System.out.println("成年です。");  
        } else {  
            System.out.println("未成年です。");  
        }  
        System.out.println("終了します。");  
    }  
}
```

実行結果(ageが20以上の場合)

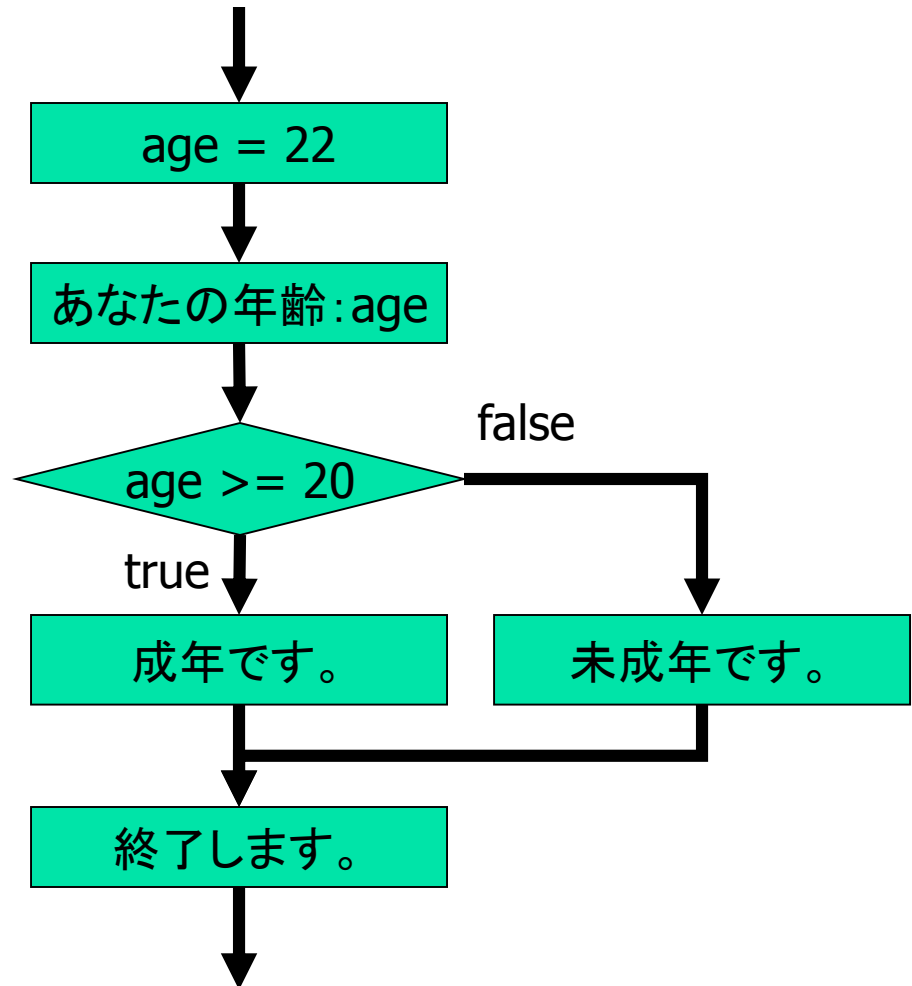
```
C:\java>java IfSample2  
あなたの年齢:22  
成年です。  
終了します。
```

実行結果(ageが20未満の場合)

```
C:\java>java IfSample2  
あなたの年齢:18  
未成年です。  
終了します。
```

# フローチャート

```
int age = 22;  
System.out.println("あなたの年齢:"  
    + age);  
if (age >= 20) {  
    System.out.println("成年です。");  
} else {  
    System.out.println("未成年です。");  
}  
System.out.println("終了します。");
```



# if-else if-else文のサンプルプログラム

```
class IfSample3 {  
    public static void main(String[] args) {  
        int age = 22;  
        System.out.println("あなたの年齢:" + age);  
        if (age >= 60) {  
            System.out.println("高齢者です。");  
        } else if (age >= 20) {  
            System.out.println("成年です。");  
        } else {  
            System.out.println("未成年です。");  
        }  
        System.out.println("終了します。");  
    }  
}
```

実行結果 (ageが60以上の場合)

```
C:\¥java>java IfSample3
```

あなたの年齢:65

高齢者です。

終了します。

実行結果 (ageが60未満20以上の場合)

```
C:\¥java>java IfSample3
```

あなたの年齢:22

成年です。

終了します。

実行結果 (ageが20未満の場合)

```
C:\¥java>java IfSample3
```

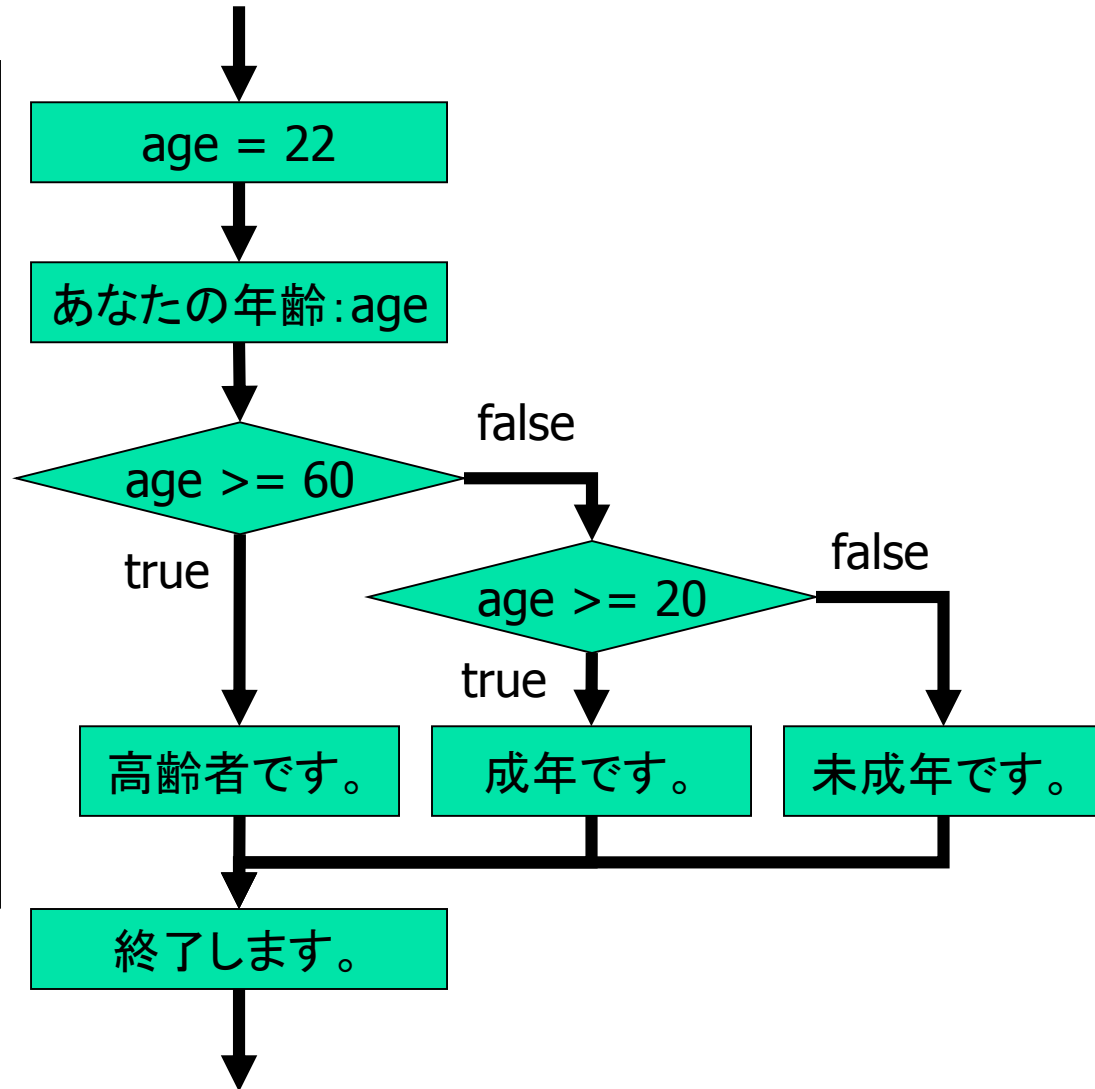
あなたの年齢:18

未成年です。

終了します。

# フローチャート

```
int age = 22;  
System.out.println("あなたの年齢:"  
    + age);  
if (age >= 60) {  
    System.out.println("高齢者です。");  
} else if (age >= 20) {  
    System.out.println("成年です。");  
} else {  
    System.out.println("未成年です。");  
}  
System.out.println("終了します。");
```



# 論理演算子

| 論理演算子 | 意味  | 例          | 説明                      |
|-------|-----|------------|-------------------------|
| !     | 否定  | ! 条件       | 条件の真偽を反転させる             |
| &&    | かつ  | 条件① && 条件② | ①と②が両方ともtrueの場合のみtrue   |
|       | または | 条件①    条件② | ①と②の少なくとも一方がtrueの場合true |

# 論理演算子のサンプルプログラム

```
class IfSample4 {  
    public static void main(String[] args) {  
        int week = 3; // 曜日の日～土を0～6で表す  
        boolean isFemale = true; // 女性ならばtrue  
        int price;  
        if (week == 3 && isFemale == true) {  
            price = 1000;  
        } else {  
            price = 2000;  
        }  
        System.out.println("チケットは"  
            + price + "円です。");  
    }  
}
```

実行結果(水曜日かつ女性の場合)

```
C:\¥java>java IfSample4  
チケットは1000円です。
```

実行結果(水曜日かつ女性でない場合)

```
C:\¥java>java IfSample4  
チケットは2000円です。
```

実行結果(水曜日以外かつ女性の場合)

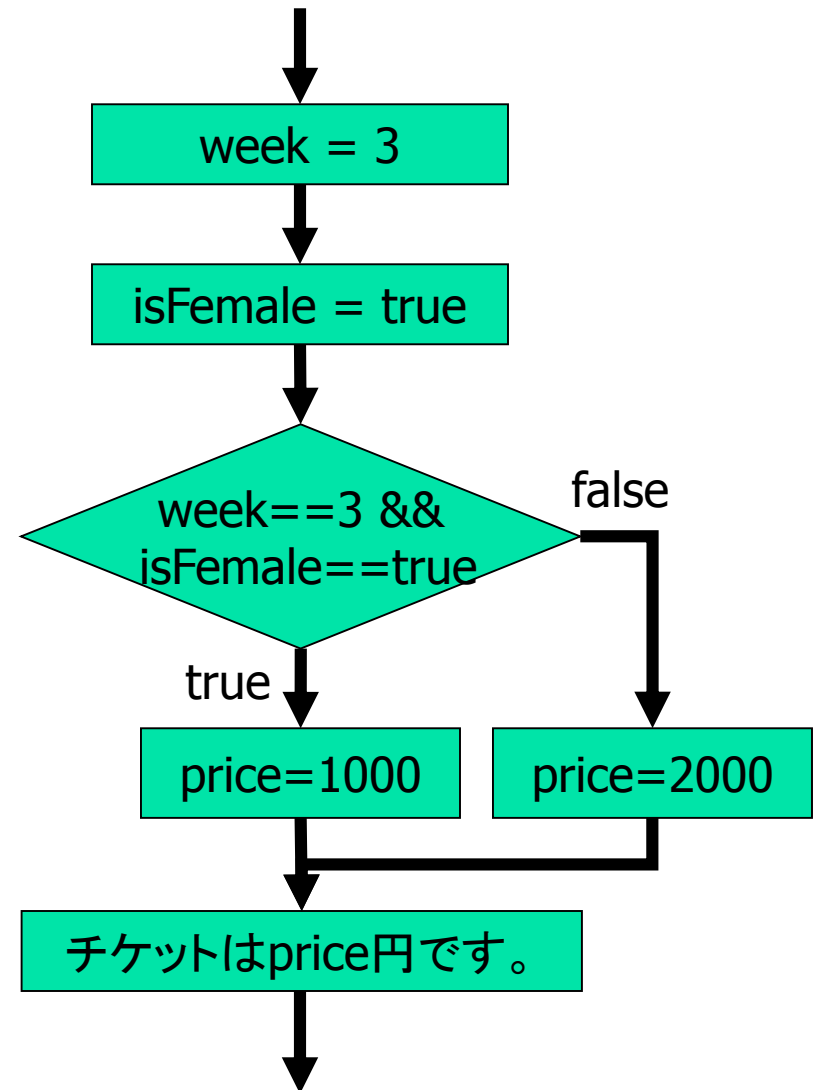
```
C:\¥java>java IfSample4  
チケットは2000円です。
```

実行結果(水曜日以外かつ女性でない場合)

```
C:\¥java>java IfSample4  
チケットは2000円です。
```

# フローチャート

```
int week = 3;  
boolean isFemale = true;  
int price;  
if (week == 3 && isFemale == true) {  
    price = 1000;  
} else {  
    price = 2000;  
}  
System.out.println("チケットは"  
    + price + "円です。");
```



# if文のネスト

▶ if文の中にif文を記述できる

```
if (条件①) {  
    if (条件②) {  
        ...  
    } else {  
        ...  
    }  
} else {  
    ...  
}
```



# ネストのサンプルプログラム

```
class IfSample5 {  
    public static void main(String[] args) {  
        int week = 3;  
        boolean isFemale = true;  
        int price;  
        if (week == 3) {  
            if (isFemale == true) {  
                price = 1000;  
            } else {  
                price = 2000;  
            }  
        } else {  
            price = 2000;  
        }  
        System.out.println("チケットは"  
            + price + "円です。");  
    }  
}
```

実行結果(水曜日かつ女性の場合)

```
C:¥java>java IfSample5  
チケットは1000円です。
```

実行結果(水曜日かつ女性でない場合)

```
C:¥java>java IfSample5  
チケットは2000円です。
```

実行結果(水曜日以外かつ女性の場合)

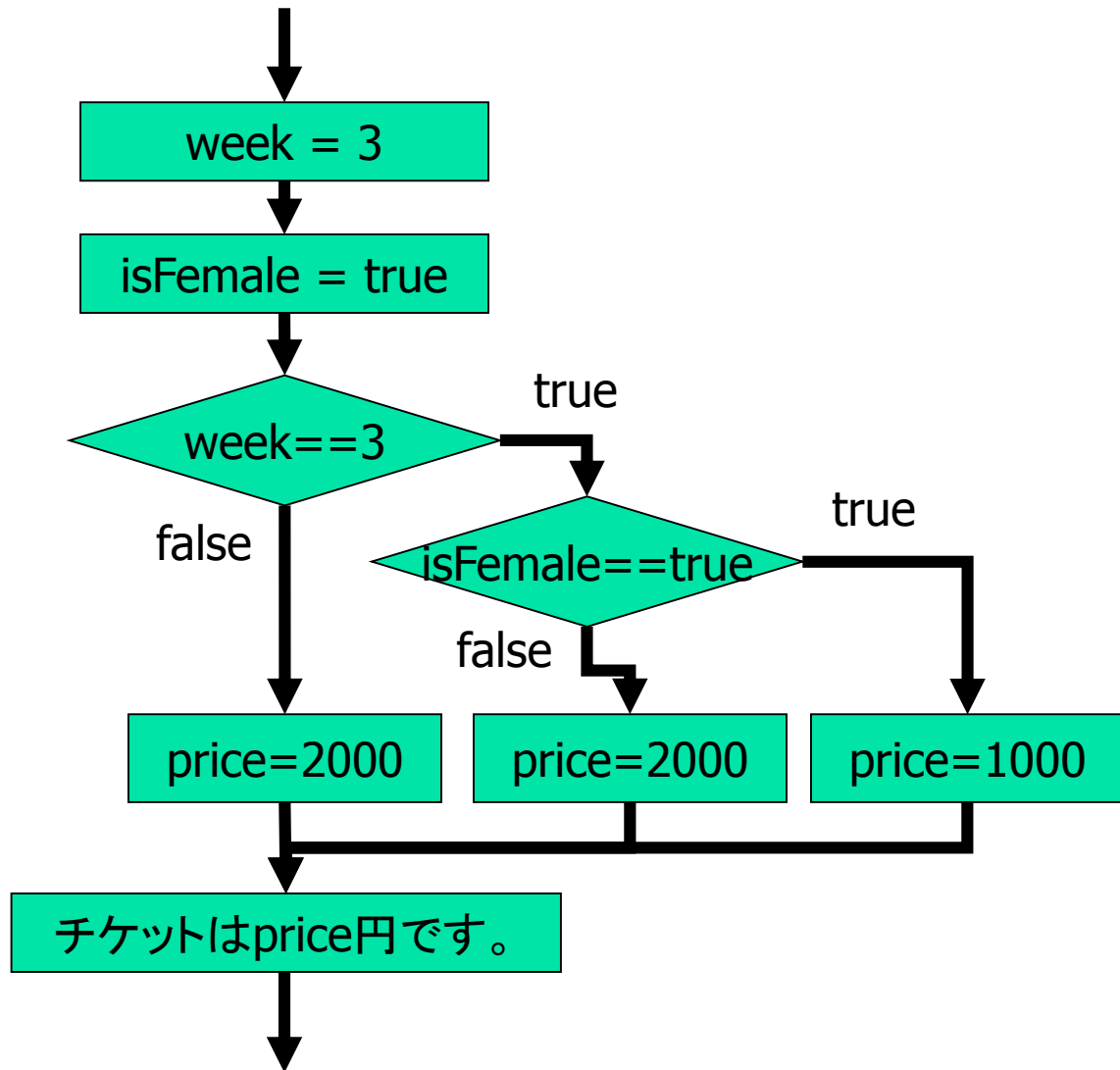
```
C:¥java>java IfSample5  
チケットは2000円です。
```

実行結果(水曜日以外かつ女性でない場合)

```
C:¥java>java IfSample5  
チケットは2000円です。
```

# フローチャート

```
int week = 3;
boolean isFemale = true;
int price;
if (week == 3) {
    if (isFemale == true) {
        price = 1000;
    } else {
        price = 2000;
    }
} else {
    price = 2000;
}
System.out.println("チケットは"
    + price + "円です。");
```



# ネストの注意点

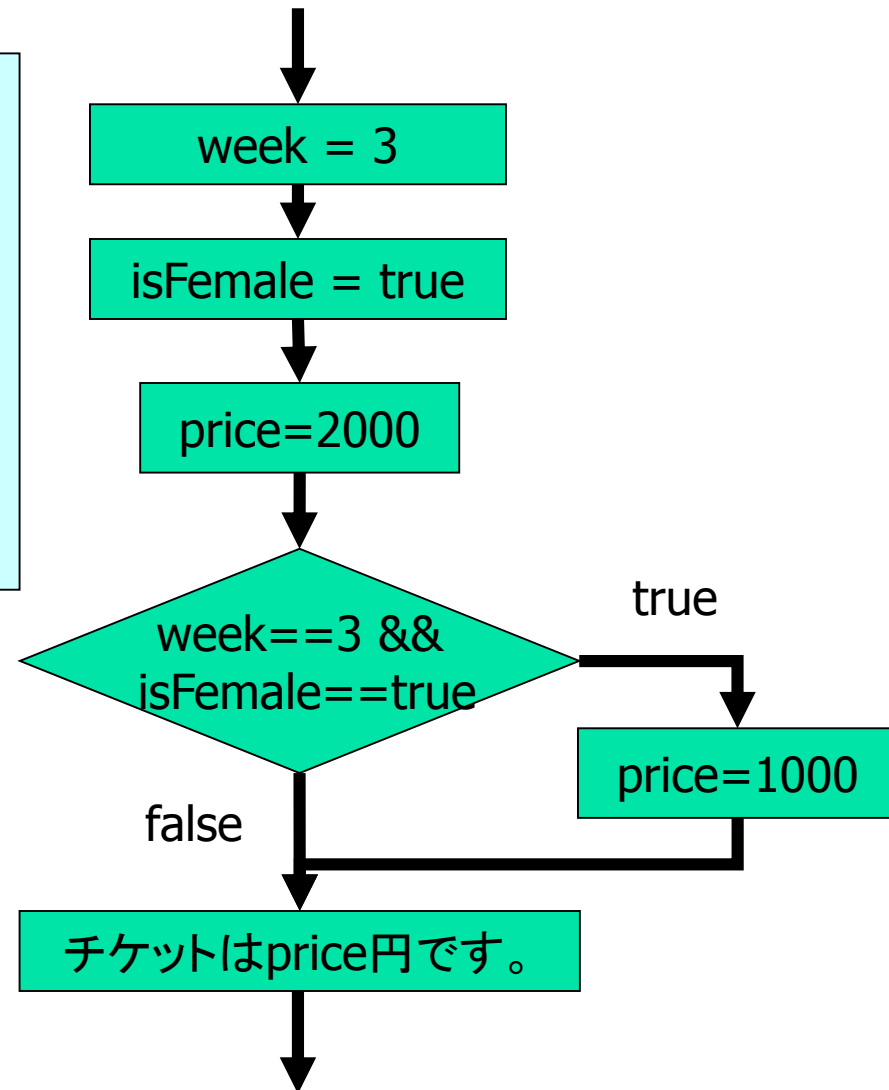
- ▶ ネストが深くなりすぎると、可読性が下がるので注意
- ▶ ネストが深くなった場合の対策
  - ▶ 論理演算子を利用して書き換えるなど、論理構造そのものを見直す
  - ▶ プログラムの一部をメソッド化する

# 論理構造を見直した例

```
class IfSample6 {  
    public static void main(String[] args) {  
        int week = 3;  
        boolean isFemale = true;  
        int price = 2000;  
        if (week == 3 && isFemale == true) {  
            price = 1000;  
        }  
        System.out.println("チケットは" + price + "円です。");  
    }  
}
```

# フローチャート

```
int week = 3;  
boolean isFemale = true;  
int price = 2000;  
if (week == 3 && isFemale == true) {  
    price = 1000;  
}  
System.out.println("チケットは"  
    + price + "円です。");
```



# 中カッコの省略

▶ ブロックの中身が1行のみの場合、{}を省くことが出来る。

▶ ただし、この書き方は推奨されていない

```
if (条件式1)
    処理1;
else if (条件式2)
    処理2;
else
    処理3;
```

=

```
if (条件式1) {
    処理1;
} else if (条件式2) {
    処理2;
} else {
    処理3;
}
```

# 中カッコの省略が推奨されない理由

- ▶ 可読性が下がる
- ▶ バグの温床となる

処理2は、条件式の  
真偽にかかわらず  
実行される

```
if (条件式)
    処理1;
    処理2;
```

コンパイルエラー

```
if (条件式1)
    処理1-A;
    処理1-B;
else if (条件式2)
    処理2;
else
    処理3;
```



switch文

---



# サンプルプログラム

```
class SwitchSample1 {  
    public static void main(String[] args) {  
        int num = 20;  
        switch (num) {  
            case 10:  
                System.out.println("A"); break;  
            case 20:  
                System.out.println("B"); break;  
            case 30:  
                System.out.println("C"); break;  
            default:  
                System.out.println("D");  
        }  
    }  
}
```

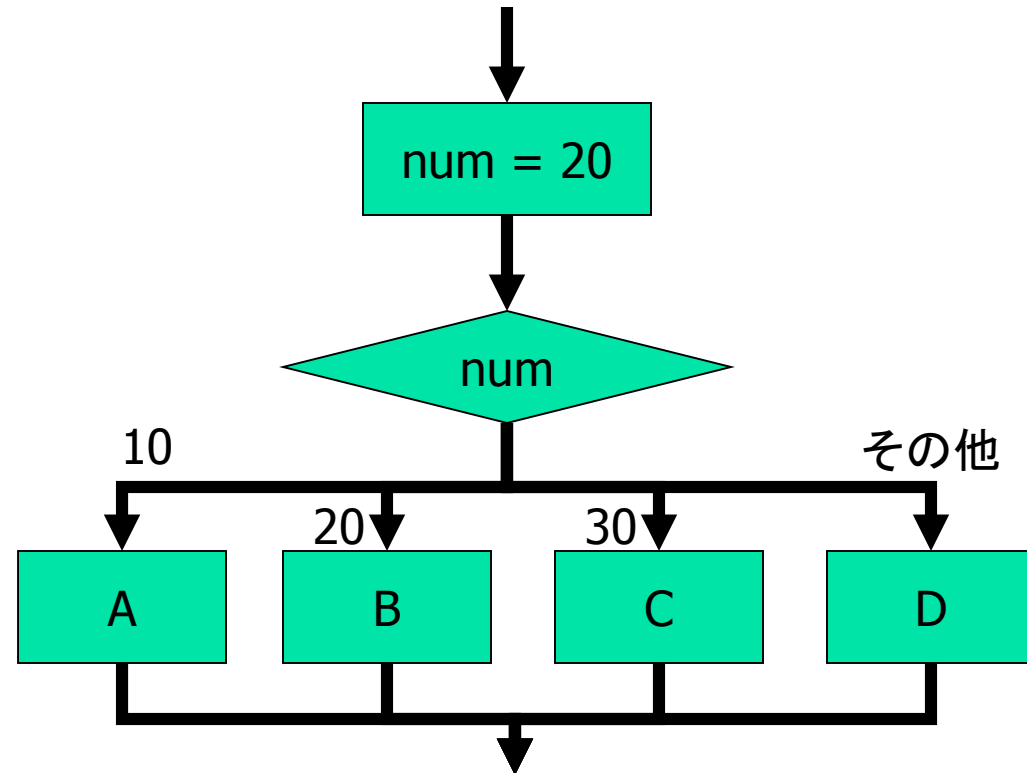
## 実行結果

```
C:¥java>java SwitchSample1
```

```
B
```

# フローチャート

```
int num = 20;  
switch (num) {  
case 10:  
    System.out.println("A"); break;  
case 20:  
    System.out.println("B"); break;  
case 30:  
    System.out.println("C"); break;  
default:  
    System.out.println("D");  
}
```



# 分岐するための変数

- ▶ long以外の整数型 (byte、short、int)
- ▶ 文字 (char型)
- ▶ enum型
- ▶ String型 (Java 7以降)

# breakの役割

---

▶ switchのブロックから抜け出す

# 第6章 繰り返し

---



while文

---

# サンプルプログラム

```
class WhileSample1 {  
    public static void main(String[] args) {  
        int i = 0;  
        while (i < 3) {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

実行結果

```
C:\¥java>java WhileSample1
```

```
0
```

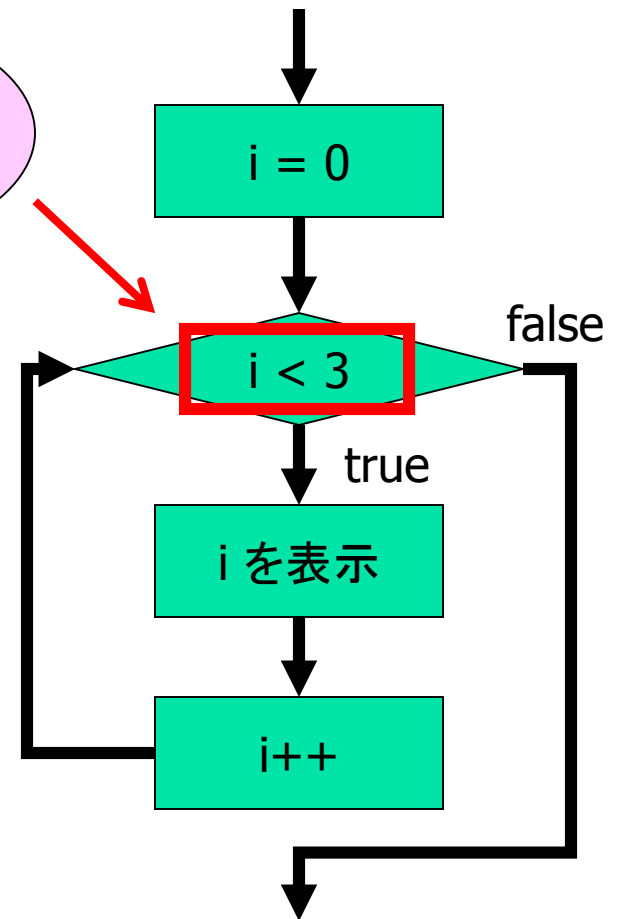
```
1
```

```
2
```

# while文のフローチャート

```
int i = 0;  
while (i < 3) {  
    System.out.println(i);  
    i++;  
}
```

繰り返し  
条件





# do-while文

---

# サンプルプログラム

```
class DoWhileSample1 {  
    public static void main(String[] args) {  
        int i = 0;  
        do {  
            System.out.println(i);  
            i++;  
        } while (i < 3);  
    }  
}
```

実行結果

```
C:\¥java>java DoWhileSample1
```

```
0
```

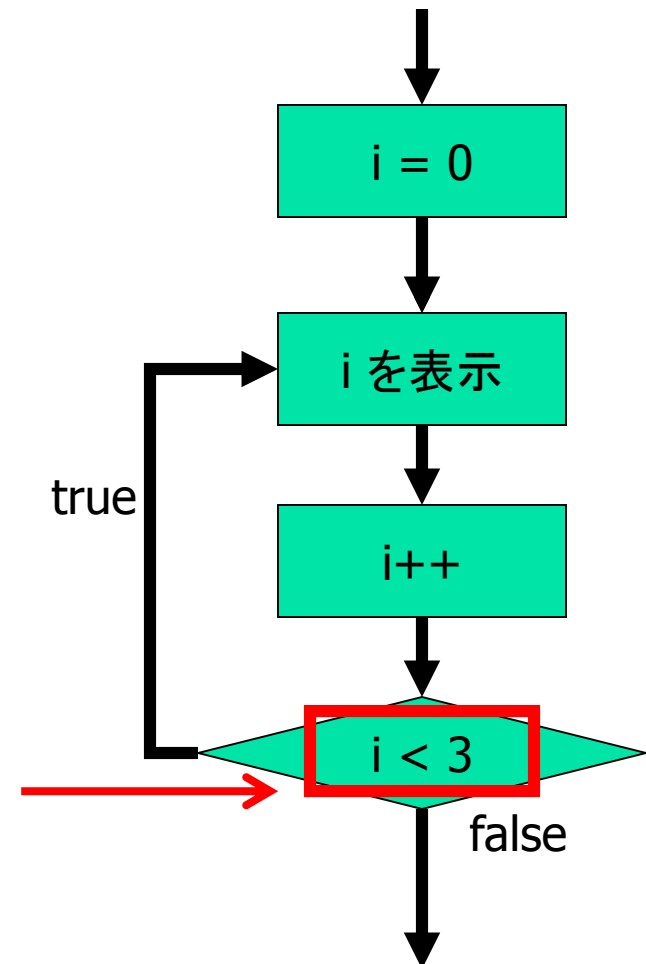
```
1
```

```
2
```

# do-while文のフローチャート

```
int i = 0;  
do {  
    System.out.println(i);  
    i++;  
} while (i < 3);
```

繰り返し  
条件



# whileとdo-whileの違い

- ▶ do-while文は、処理を実行した後に繰り返し条件を評価するため、処理が最低でも1回は必ず実行される



for文

# サンプルプログラム

```
class ForSample1 {  
    public static void main(String[] args) {  
        for (int i = 0; i < 3; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

実行結果

```
C:\¥java>java ForSample1
```

```
0
```

```
1
```

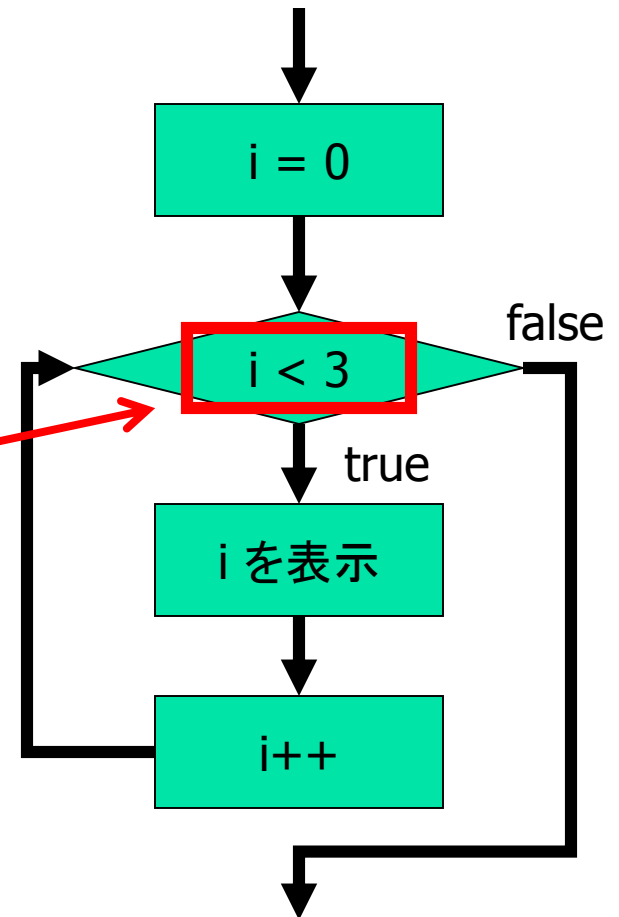
```
2
```

# for文のフローチャート

▼ フローチャートはwhile文と同じ

```
for (int i = 0; i < 3; i++) {  
    System.out.println(i);  
}
```

繰り返し  
条件



# for文の構造 (while文との比較)

for文

```
for (int i = 0; i < 3; i++) {  
    System.out.println(i);  
}
```

while文

```
int i = 0;  
while (i < 3) {  
    System.out.println(i);  
    i++;  
}
```



# 中カッコの省略

▶ ブロックの中身が1行のみの場合、{}を省くことが出来る。

▶ ただし、この書き方は推奨されていない

```
while (条件式)  
  処理;
```

=

```
while (条件式) {  
  処理;  
}
```

```
for ( ; ; )  
  処理;
```

=

```
for ( ; ; ) {  
  処理;  
}
```

```
do  
  処理;  
while (条件式);
```

=

```
do {  
  処理;  
} while (条件式);
```

# 繰り返しのネスト

---

# サンプルプログラム

## ◀ 繰り返しの中に繰り返しを記述できる

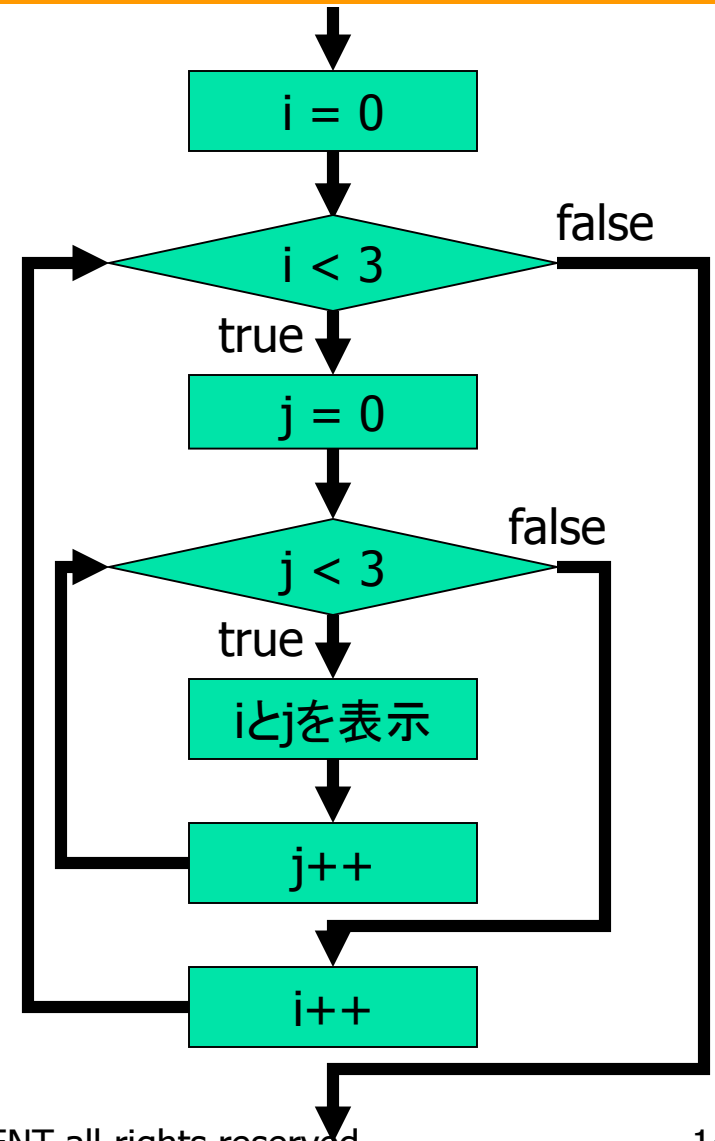
```
class NestForSample1 {  
    public static void main(String[] args) {  
        for (int i = 0; i < 3; i++) {  
            for (int j = 0; j < 3; j++) {  
                System.out.println("i = " + i + ", j = " + j);  
            }  
        }  
    }  
}
```

### 実行結果

```
C:\¥java>java NestForSample1  
i = 0, j = 0  
i = 0, j = 1  
i = 0, j = 2  
i = 1, j = 0  
i = 1, j = 1  
i = 1, j = 2  
i = 2, j = 0  
i = 2, j = 1  
i = 2, j = 2
```

# フローチャート

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        System.out.println(  
            "i = " + i + ", j = " + j);  
    }  
}
```



# 条件分岐と繰り返しの 組み合わせ

---

# サンプルプログラム

▶ 繰り返しと条件分岐を組み合わせて記述できる

```
class ForIfSample1 {  
    public static void main(String[] args) {  
        for (int i = 0; i < 5; i++) {  
            if (i % 2 == 0) {  
                System.out.println(i + "は偶数です");  
            }  
        }  
    }  
}
```

実行結果

C:\¥java>java ForIfSample1

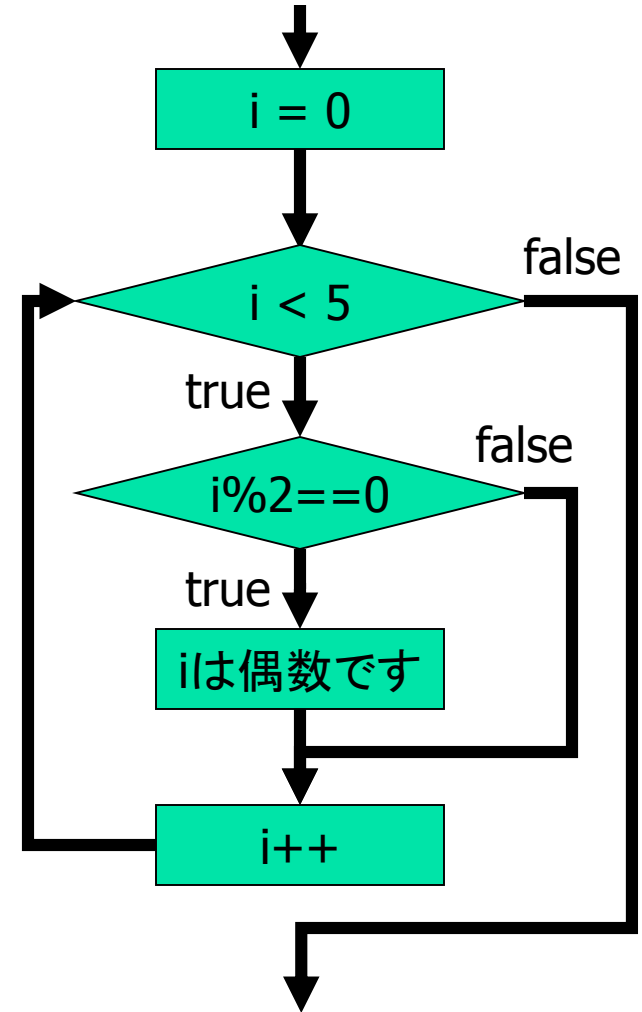
0は偶数です

2は偶数です

4は偶数です

# フローチャート

```
for (int i = 0; i < 5; i++) {  
    if (i % 2 == 0) {  
        System.out.println(i + "は偶数です");  
    }  
}
```





# 繰り返し制御

---



# break文

## ループ処理から強制的に抜け出す

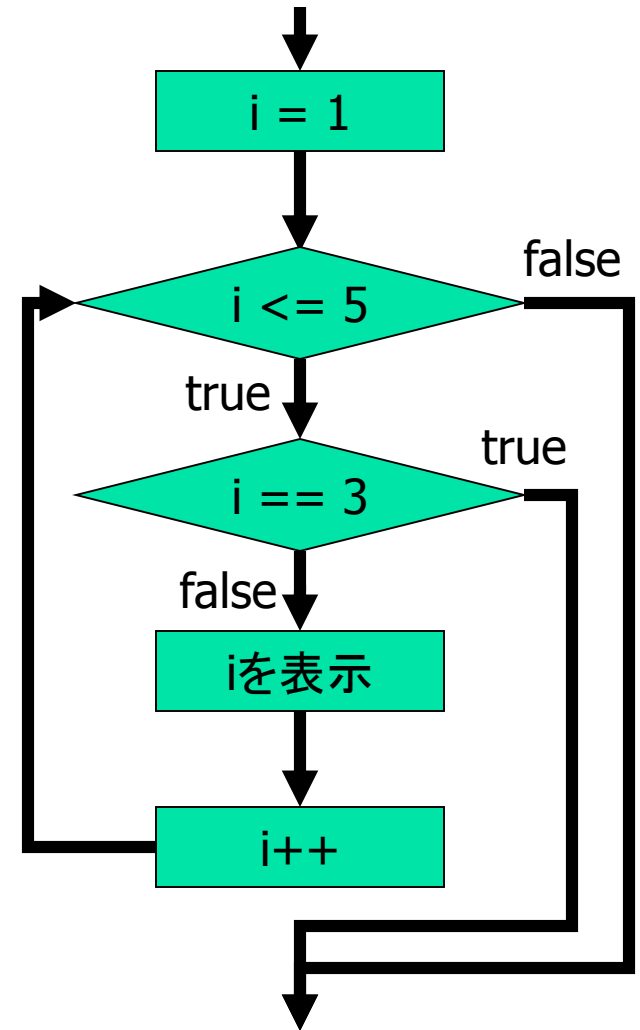
```
class BreakSample1 {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            if (i == 3) {  
                break;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

実行結果

```
C:\¥java>java BreakSample1
```

1

2



# continue文

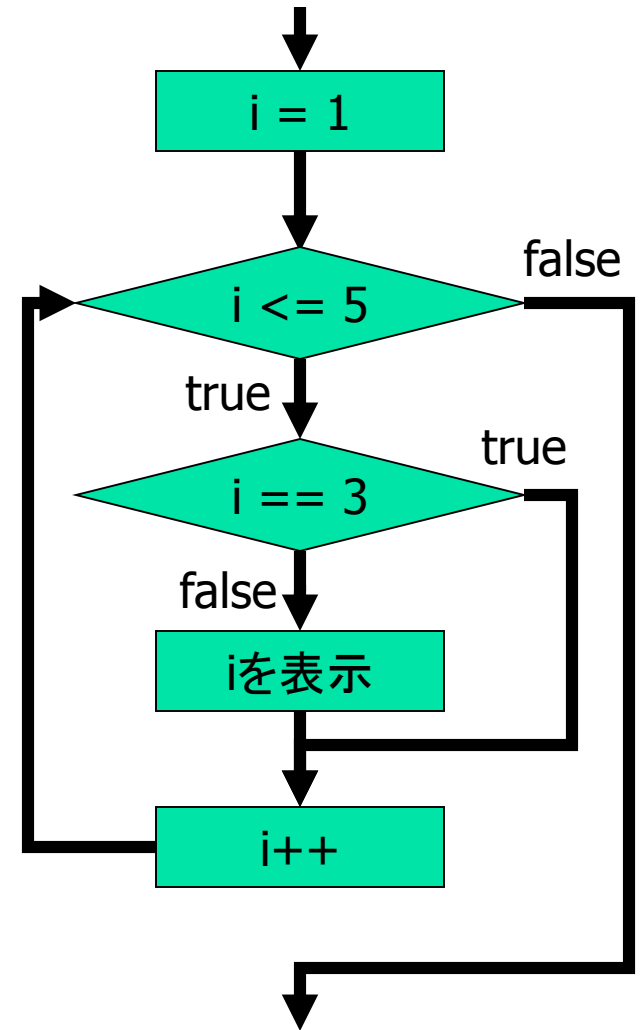
## ループ内の、continue 以下の処理を飛ばす

```
class ContinueSample1 {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            if (i == 3) {  
                continue;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

実行結果

```
C:\¥java>java ContinueSample1
```

```
1  
2  
4  
5
```



# continueの注意点

- ▶ 可読性が下がるので、continueは多用しない
- ▶ 別の書き方で代用可能

```
for (int i = 1; i <= 5; i++) {  
    if (i == 3) {  
        continue;  
    }  
    System.out.println(i);  
}
```



```
for (int i = 1; i <= 5; i++) {  
    if (i != 3) {  
        System.out.println(i);  
    }  
}
```



# 第7章

## 配列

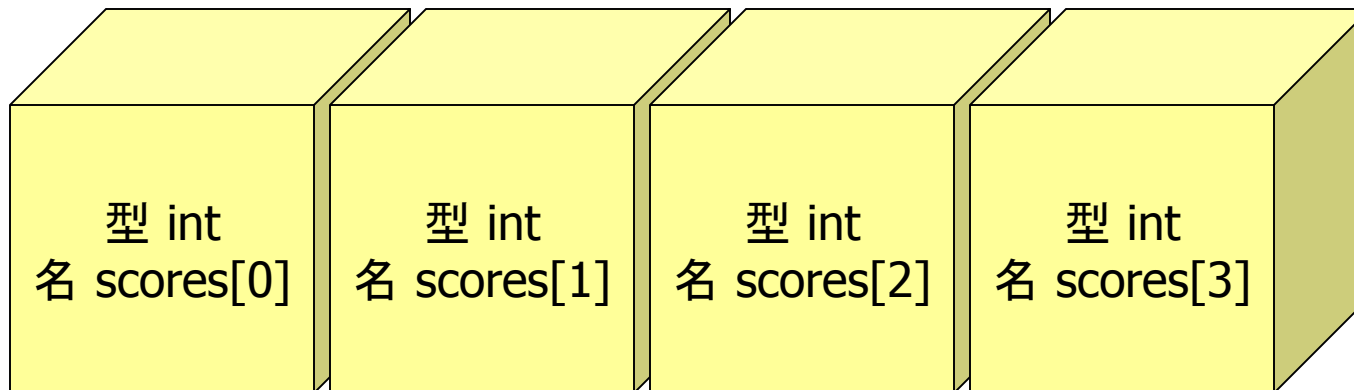
---

# 1次元配列

---

# 配列とは？

- ▶ 同じ型の変数を何個も同時に定義する
  - ▶ 40人の各生徒のテストの点数
    - ▶ `int[] scores = new int[40]`
  - ▶ 365日間の各日の最高気温
    - ▶ `double[] temperatures = new double[365]`
  - ▶ 10000人の各口座残高
    - ▶ `long[] accounts = new long[10000]`
  - ...など



# サンプルプログラム

```
class ArraySample1 {  
    public static void main(String[] args) {  
        int[] a = new int[3];  
        a[0] = 10;  
        a[1] = 20;  
        a[2] = 30;  
        System.out.println(a[0]);  
        System.out.println(a[1]);  
        System.out.println(a[2]);  
    }  
}
```

実行結果

```
C:¥java>java ArraySample1  
10  
20  
30
```

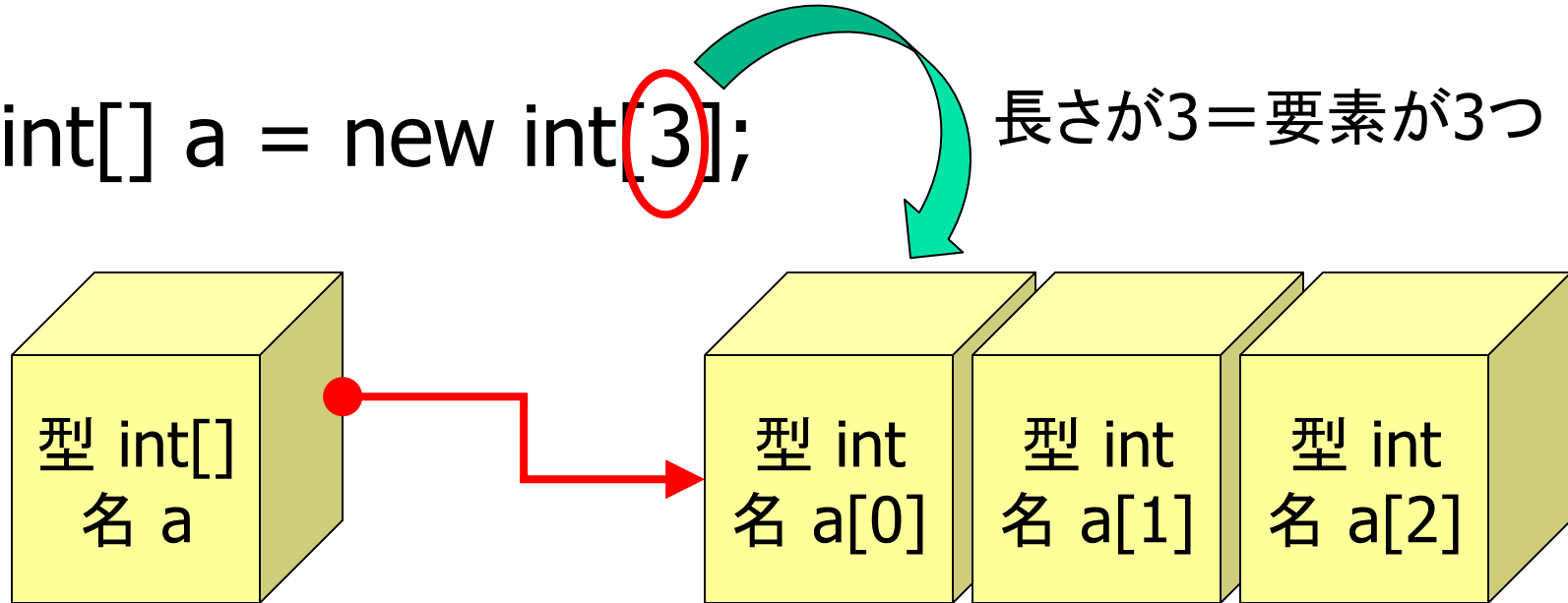


# 配列の生成

- ▶ **new 型名[長さ]**で生成
- ▶ 生成された1つ1つの変数を「要素」という

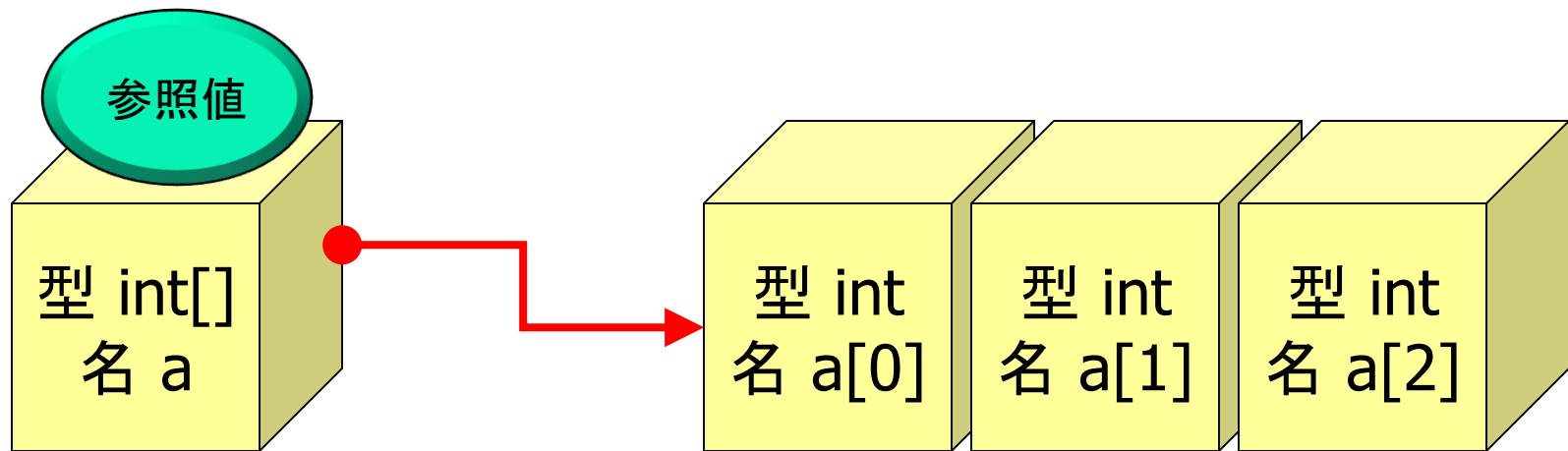
```
int[] a = new int[3];
```

長さが3＝要素が3つ



# 参照型変数

- 配列型の変数には、実際の値ではなく、その値の場所を表す値（参照値）が入っている



# 配列の添え字

▼ 添え字は0～(長さ-1)

int[] a = new int[3];

長さ

a[0] = 10;

添え字(インデックス)

# ArrayIndexOutOfBoundsException

- ▶ 範囲外の添え字（長さ以上や負の値）を指定した際に発生する例外

```
int[] a = new int[3];  
a[3] = 10;
```

実行結果

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at Foo.main(Foo.java:9)
```

# 配列要素のデフォルト値

## ▶ デフォルトの初期値

- ▶ 配列の初期化をした段階で、各要素に最初に入っている値

| データ型                                    | 初期値   |
|---|-------|
| 論理値 (boolean)                           | false |
| 整数、小数 (int、long、float、double、char、byte) | 0     |
| 参照型 (Stringなど)                          | null  |

# その他の初期化方法

▼ 以下は全て、意味的には同じ

▼ `int[] a = new int[3];`

`a[0] = 10;`

`a[1] = 11;`

`a[2] = 12;`

▼ `int[] a = {10, 11, 12};` // 初期化時のみ

▼ `int[] a = new int[]{10, 11, 12};`

# 配列の長さ

▶ 「配列名 **.length**」で取得する

```
class ArraySample2 {  
    public static void main(String[] args) {  
        int[] a = {10, 20, 30, 40, 50};  
        for (int i = 0; i < a.length; i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

実行結果

```
C:¥java>java ArraySample2  
10  
20  
30  
40  
50
```

# 【参考】拡張for文

▶ 配列の全要素を、順番に取得できる

▶ 「for-each文」とも呼ばれる(糖衣構文=シンタックスシュガー)

```
int[] a = {10, 20, 30, 40, 50};  
for (int i = 0; i < a.length; i++) {  
    int num = a[i];  
    System.out.println(num);  
}
```

=

```
int[] a = {10, 20, 30, 40, 50};  
for (int num : a) {  
    System.out.println(num);  
}
```





# 多次元配列

---

# 2次元配列の使用例

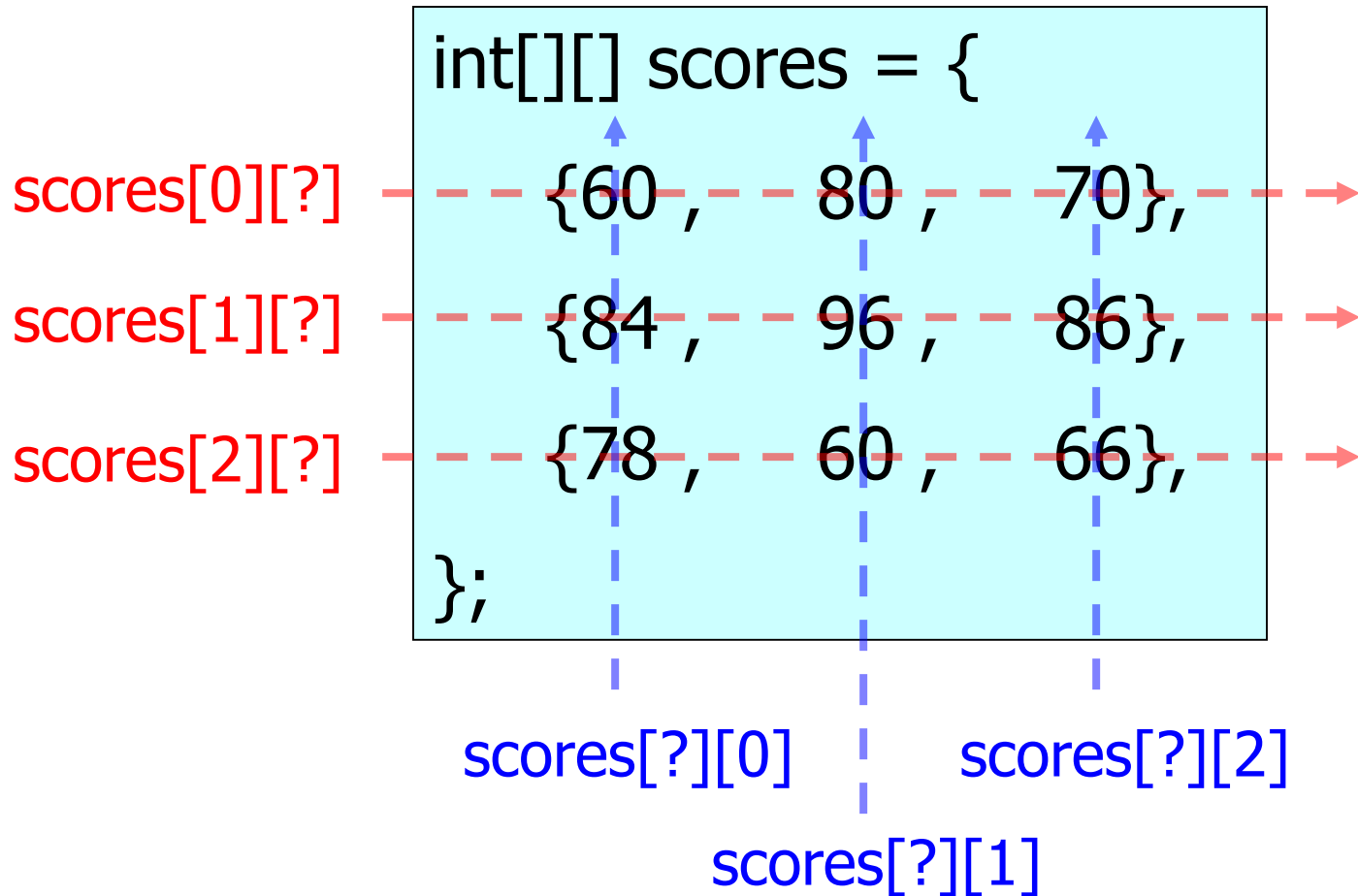
▶ 3人の生徒の国語・数学・英語の成績

```
int[][] scores = {  
    {60, 80, 70}, // 生徒①  
    {84, 96, 86}, // 生徒②  
    {78, 60, 66}, // 生徒③  
};
```

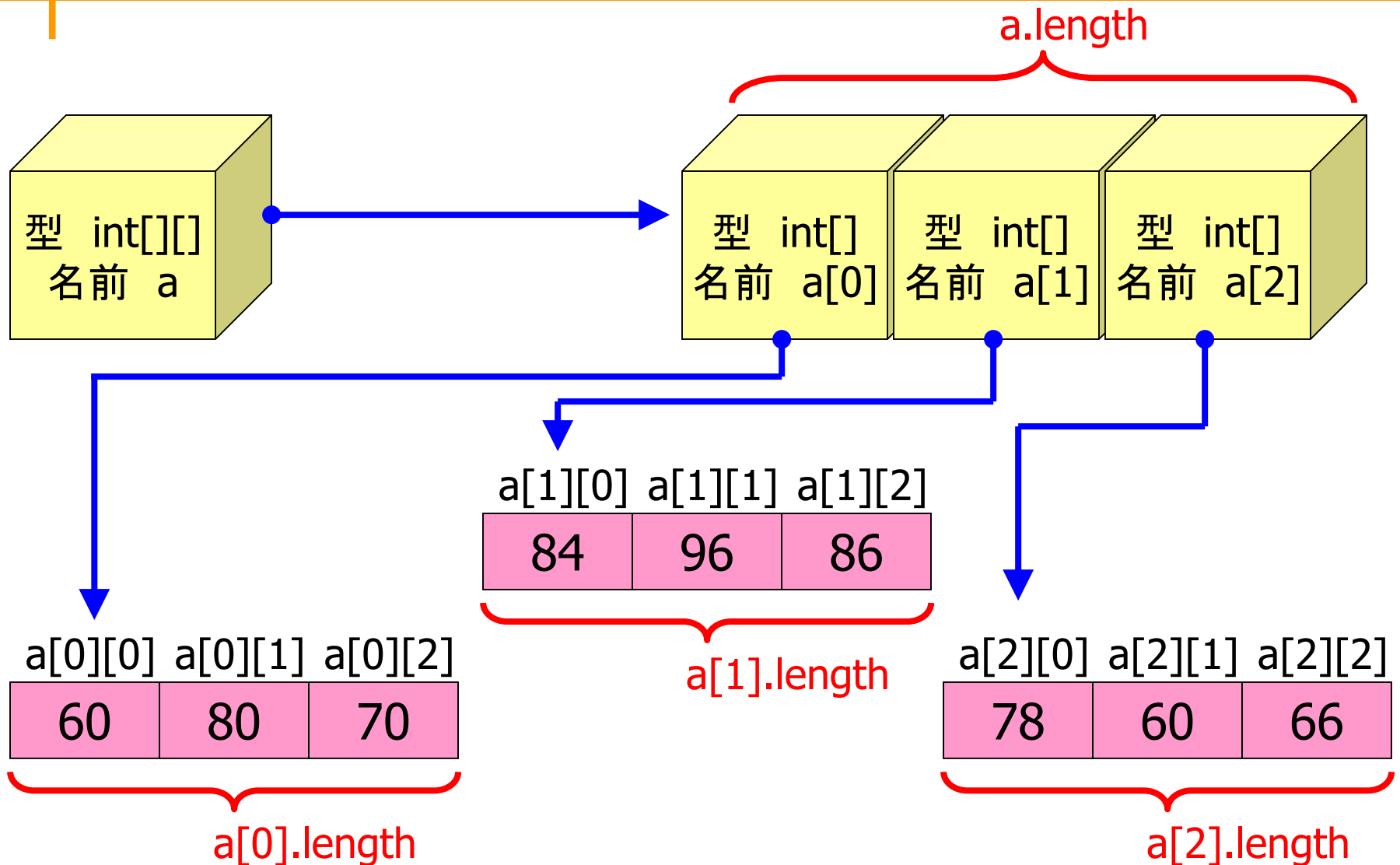
▶ オセロ盤 (何も無い=0、白=1、黒=2)

```
int[][] board = {  
    {0, 0, 0, 0, 0, 0, 0, 0},  
    {0, 0, 0, 0, 0, 0, 0, 0},  
    {0, 0, 2, 0, 0, 0, 0, 0},  
    {0, 0, 2, 1, 2, 0, 0, 0},  
    {0, 0, 2, 2, 1, 1, 0, 0},  
    {0, 0, 0, 0, 1, 0, 0, 0},  
    {0, 0, 0, 0, 0, 0, 0, 0},  
    {0, 0, 0, 0, 0, 0, 0, 0},  
};
```

# 2次元配列の添え字



# 多次元配列の変数の構造



# 第8章 メソッド

---

# メソッドとは

---

# メソッドとは

- ➡ いくつかの処理をひとまとめにしておき、後で呼び出せるようにしたもの
  - ➡ いわゆる「関数」「サブルーチン」「プロシージャ」
  - ➡ `main()`もメソッド
- ➡ 標準で準備されているメソッドもある
  - ➡ `println()`、`Integer.parseInt()`など
- ➡ 自作できる

# サンプルプログラム

```
class MethodSample1 {  
    public static void main(String[] args) {  
        System.out.println("mainを開始します。");  
        method();  
        System.out.println("mainを終了します。");  
    }  
    static void method() {  
        System.out.println("メソッドを開始します。");  
        System.out.println("メソッドを終了します。");  
    }  
}
```

実行結果

```
C:\¥java>java MethodSample1
```

```
mainを開始します。
```

```
メソッドを開始します。
```

```
メソッドを終了します。
```

```
mainを終了します。
```



# 処理の流れ

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

処理1;

method();

処理2;

}

```
static void method() {
```

処理A;

}

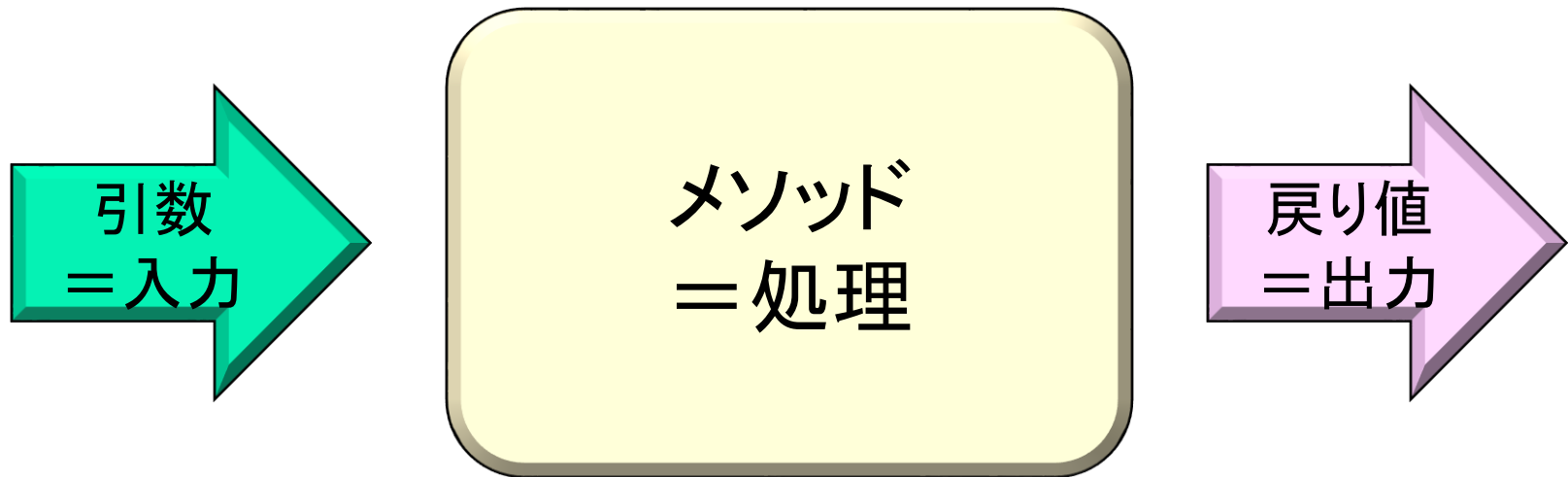
- ▶ 「処理1→処理A→処理2」の順に処理が行われる。
- ▶ メソッドを使うことを「**メソッドを呼び出す**」と言う。
- ▶ static、void、()については後ほど説明

# メソッドの引数・戻り値

---

# 引数・戻り値とは

- ▶ 引数＝メソッドの処理に利用する入力値
- ▶ 戻り値＝メソッドの処理結果を表す出力値



# サンプルプログラム

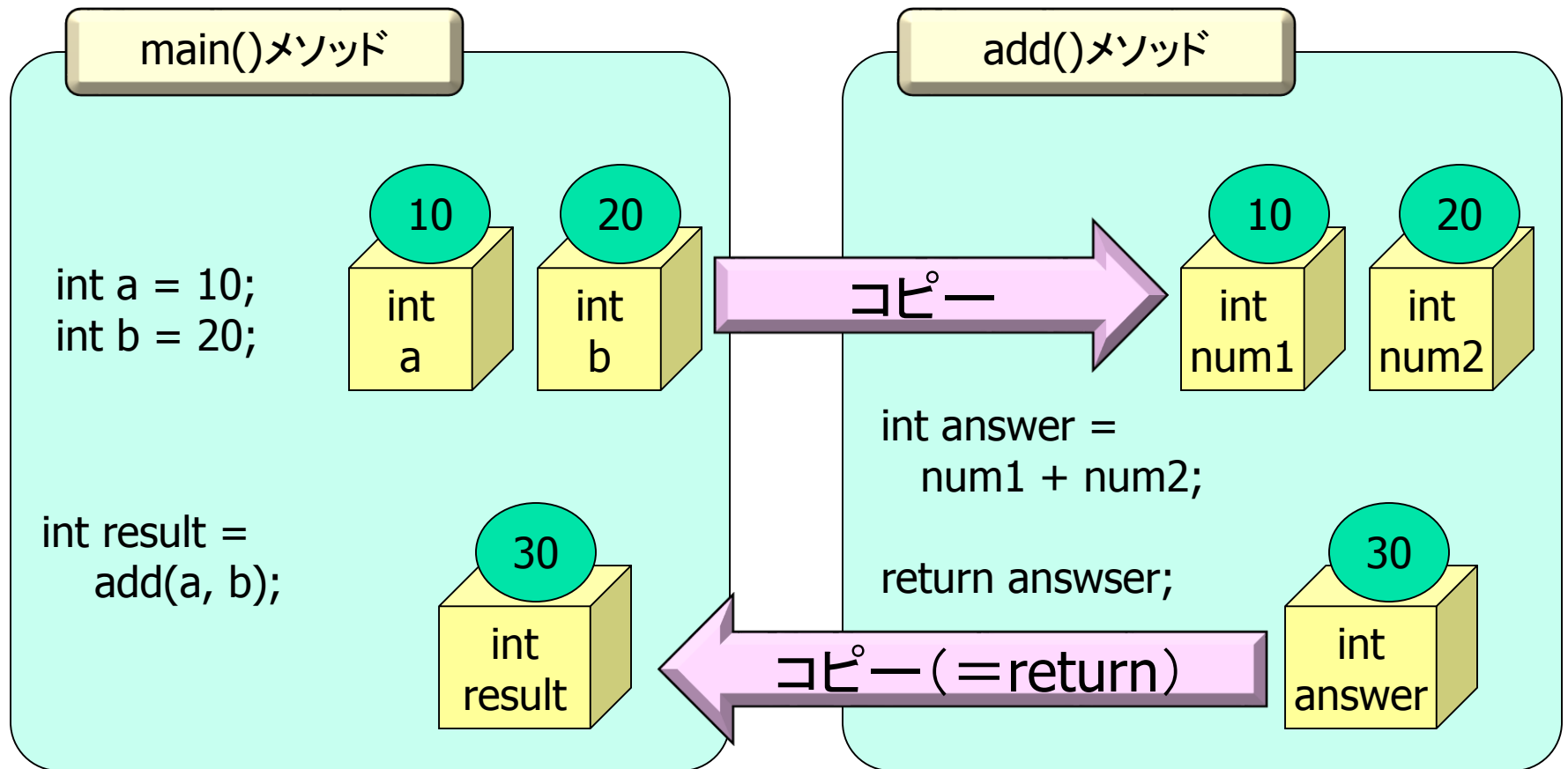
```
class MethodSample2 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        int result = add(a, b);  
        System.out.println("結果は" + result);  
    }  
    static int add(int num1, int num2) {  
        int answer = num1 + num2;  
        return answer;  
    }  
}
```

実行結果

C:¥java>MethodSample2

結果は30

# 引数と戻り値のイメージ



# 戻り値の型

▶ returnで返す値と、メソッドの戻り値の型は一致させる

×コンパイルエラー

```
static int method() {  
    String str = "ABC";  
    return str;  
}  
String型
```

○ OK

```
static double method() {  
    double d = 3.14;  
    return d;  
}  
double型
```

○ これもOK

```
static double method() {  
    int num = 9999;  
    return num;  
}  
int型
```

# void型

- ▶ 戻り値を返さないメソッドの場合、そのメソッドの戻り値の型は「**void**」とする
  - ▶ main()メソッド
  - ▶ 何かを表示するだけのメソッド
  - ...など

# mainメソッドの引数

- ▶ javaコマンドを利用して、mainメソッドの引数に渡すことができる

```
class MethodSample3 {  
    public static void main( String[] args ) {  
        System.out.println(args[0]);  
        System.out.println(args[1]);  
    }  
}
```

実行結果

```
C:¥java>java MethodSample3 xxx yyy  
  
xxx  
  
yyy
```



# メソッドのメリット

---

# メソッドのメリット

- ▶ 保守性・再利用性が向上する
  - ▶ 1つのメソッド当たりの行数を短くできる

# 第9章 プログラムの品質

---



# 品質の基準

---

# 「良い」プログラムとは

---

- ▶仕様通りに正しく動作する
- ▶修正がしやすい

# 【参考】ISO 9126

- ▶ ISO(国際標準化機構)が定めた、ソフトウェアに関する品質基準
  - ▶ 機能性・・・正しく機能する
  - ▶ 信頼性・・・長時間運用した際の故障の少なさ
  - ▶ 使用性・・・システムの使いやすさ(ユーザビリティ)
  - ▶ 効率性・・・システムが利用する資源の効率
  - ▶ 保守性・・・変更のしやすさ
  - ▶ 移植性・・・別環境への移行のしやすさ

# プログラミングの原則

---

# 変数のスコープは狭くする

- ▶ 変数のスコープは最小限であることが原則
- ▶ スコープの広い変数を色々なところで使いまわすと、思いもよらない値が代入され、思ったように動作しないことがある(=バグの温床)



# 分かりやすい名前を付ける

- ▶ その変数に何が代入されているか、名前を読んだだけで分かることが望ましい

# 論理構造をシンプルにする

- ▶ ネストが深すぎないか？
- ▶ 無駄な処理はないか？
- ▶ 別の書き方はないか？

# DRYの原則

- ▶ Don't Repeat Yourself＝同じことを繰り返さない
  - ▶ 同じ処理は2度と書かない→メソッド化・クラス化
- ▶ 基本的にはDRYであることが望ましいが、生産性との兼ね合いに注意