

## 211.Javaソースの構成要素

Javaプロジェクトの構成

Javaソースファイルとクラスの名前

命名規則

インデント

Javaソースの構成要素-変数とデータ型

- データ型

- 変数が保持される場所

- 参照型の注意事項

Javaソースの構成要素-文字列(Stringクラス)

Javaソースの構成要素-定数

## Javaソースの構成要素-演算子

- 算術演算子（加減乗除＋余）

- インクリメント・デクリメント

- 代入演算子

- 複合代入演算子

- 関係演算子

- 論理演算子

- 3項演算子

- 演算子の優先順位と結合順

## Javaソースの構成要素-文

- コメント

- 順次

- 選択

- 繰り返し

## Javaソースの構成要素-クラス

## 目的：

Javaプログラミングについて、ソースコードを構成する最も基本的な要素を学びます。

- コーディングルール（命名規約、インデント）
- データ型と変数
- 演算子
- 制御構文
- クラス(※詳しくは次回以降)

また、Java言語の制御構文（順次・選択・繰り返し）を学び、  
if文とswitch文の処理の違い、for文とwhile文の処理の違いについて学ぶ。

## ゴール：

Javaソースコードの基本的な構成要素を理解し、実際にJavaソースコードを書いて実行できる。

## Javaプログラミングの構成イメージ

### ■ Javaプロジェクト

#### ◆ ソースファイル

ソースコード(プログラム)        …… ソースコードの構成要素

#### ◆ 中間コードファイル(※クラスファイル)

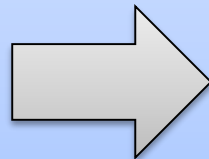
中間コード(バイトコード)

### プロジェクト

#### ソースファイル

```
class クラス名 {  
    ソースコードの構成要素  
  
    ソースコードの構成要素  
  
    ソースコードの構成要素  
}
```

コンパイル



#### 中間コードファイル (※クラスファイル)

中間コード  
(バイトコード)

## Javaプロジェクト

Java言語でアプリケーションを開発する時には、プロジェクトを作成する。

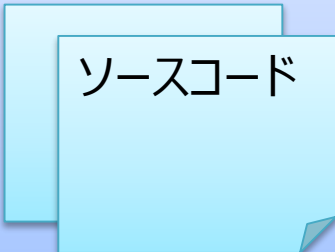
(プロジェクトの名称は、作成するアプリケーションの種類や、IDEによって、Javaプロジェクト、Webプロジェクト、Mavenプロジェクトなど異なる)

プロジェクトは、Javaソースファイル、中間コードファイル等で構成される。

通常、開発者はJavaソースファイルを作成し、JavaソースファイルにJavaプログラム(ソースコード)を記述してゆき、Javaコンパイラで中間コードファイルを生成する。

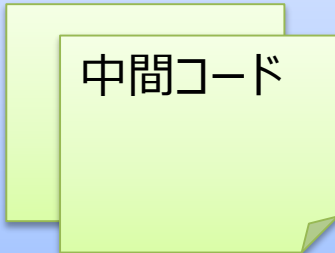
### プロジェクト

#### ソースファイル



ソースコード

#### 中間コードファイル (※クラスファイル)



中間コード

## Javaソースファイルとクラスの名前

Javaソースファイルのファイル名は**クラス名.java**とする。

(Javaでは**クラス**というものを作成してプログラムを作っていく)

※ 課題などで様々なクラスを作成して頂くが、上記の決まりを忘れない様に

※ 基本的に 1 つのJavaソースファイルに 1 つのクラスを書くが、複数のクラスを記述することもある

※ **クラスは後ほど詳しく説明する**

### **クラス名.java**

```
class クラス名 {  
    ... ..  
    ... ..  
    ... ..  
  
}
```

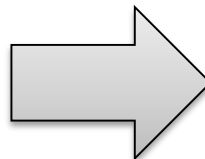
## Javaクラスファイルとクラスの名前

コンパイルした後に生成される中間コードファイルは**クラスファイル**と呼ぶ。  
クラスファイルのファイル名は**クラス名.class**となる。

**クラス名.java**

```
class クラス名 {  
    ... ..  
    ... ..  
    ... ..  
}
```

コンパイル



**クラス名.class**

中間コード (バイトコード)



Javaソースコードは、様々な構成要素で構成されている。

開発者は構成要素に名前を付けてゆくが、この名前の事を**識別子**という。(クラス名も識別子)

**識別子**には幾つか決まり(命名規則)がある。

※ 言語仕様に基づく命名規則なので、沿っていない場合はコンパイルエラーになる。

- ① 先頭の文字は「a～z」「A～Z」「\_」「\$」のいずれかで開始しなければならない
- ② 2文字目以降は「a～z」「A～Z」「\_」「\$」「0～9」を利用できる
- ③ Java言語のキーワード（**予約語**）は使用できない。（名前の一部になっている場合は使用可能）
- ④ 英字の大文字と小文字は区別される
- ⑤ 識別子は重なってはいけない
- ⑥ 予約語ではないが**true, false, null** も使えない

先の決まりに加えて識別子の命名は以下とするのが一般的である。

※ 言語仕様では無いので、沿っていなくてもはコンパイルエラーにならない。

- ① クラス名は「名詞」とする。「大文字」で始める
- ② メソッド名は「動詞」で始める。「小文字」で始める
- ③ 変数は英字の「小文字」で始める
- ④ 定数で使う英字は「大文字」のみ
- ⑤ 複数の単語からなる場合はキャメル記法(二単語目から頭を大文字)

※クラス・メソッド・定数・変数などは後述

※プロジェクトによって決まっている場合もあるので、それに従って下さい

## インデント

一般的にJavaソースファイルはインデント（字下げ）して見やすく記述する。

```
class forSample {  
    public static void main(String[] args) {  
        int i;  
        for(i = 0; i < 10; i++) {  
            System.out.println("iは" + i );  
        }  
        System.out.println("forを抜けました。iは" + i );  
    }  
}
```

※ Eclipseでは Ctrl+Shift+f でフォーマットを整えてくれる。

Javaソースコードの構成要素には以下がある。

- 変数と定数、データ型
- 演算子（主なもの）
  - 算術演算子
  - インクリメント・デクリメント
  - 代入演算子
  - 複合代入演算子
  - 関係演算子
  - 論理演算子
  - 3項演算子
- 文
  - コメント
  - 順次
  - 選択
  - 繰り返し
- クラス

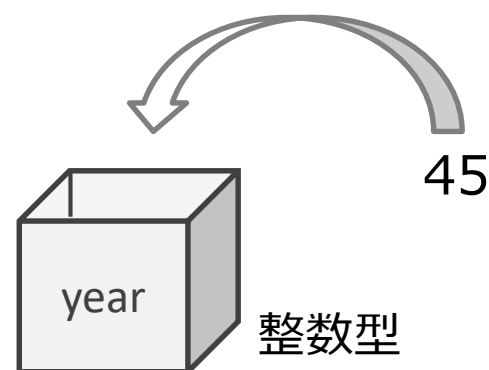
## 変数とデータ型

**変数**：

プログラムで扱うデータを格納しておくもの

**データ型**：

変数には扱うデータによって幾つかのデータ型がある



変数は「**宣言**（名称とデータ型を指定）」することで利用(**代入**や**参照**)できる  
変数に最初に値を代入する事を「**初期化**」という  
「宣言」と「初期化」を同時に行うこともできる

例)

```
class AssignmentSample {  
    public static void main(String[] args) {  
        int num;                // 整数型の変数の宣言  
        num = 10;               // 変数へ代入  
        String str = "文字列";  // 文字列の変数の初期化  
        System.out.println( "numの値は" + num + "、strの値は" + str );  
    }  
}
```

## データ型

データ型はプログラムでの保持のされ方による分類で「**基本型(プリミティブ型)**」と「**参照型(オブジェクト型)**」があり、扱うデータの形によって

**基本型の「整数型」「小数型」「文字型」「論理型」と「参照型(オブジェクト型)」がある。**

### 主なデータ型（基本型）

型名	サイズ	値の範囲	備考
byte	8 bit	-128 ~ 127	整数型
short	16 bit	-32768 ~ 32767	整数型
int	32 bit	※省略	整数型
long	64 bit	※省略	整数型
float	32bit	有効桁 6 桁	単精度浮動小数点型
double	64bit	有効桁15 桁	倍精度浮動小数点型
char	16 bit	1 文字	文字型 (Unicode)
boolean		true / false	論理型

※ 論理型 : boolean

…… **真偽値**を持つ(true または false)

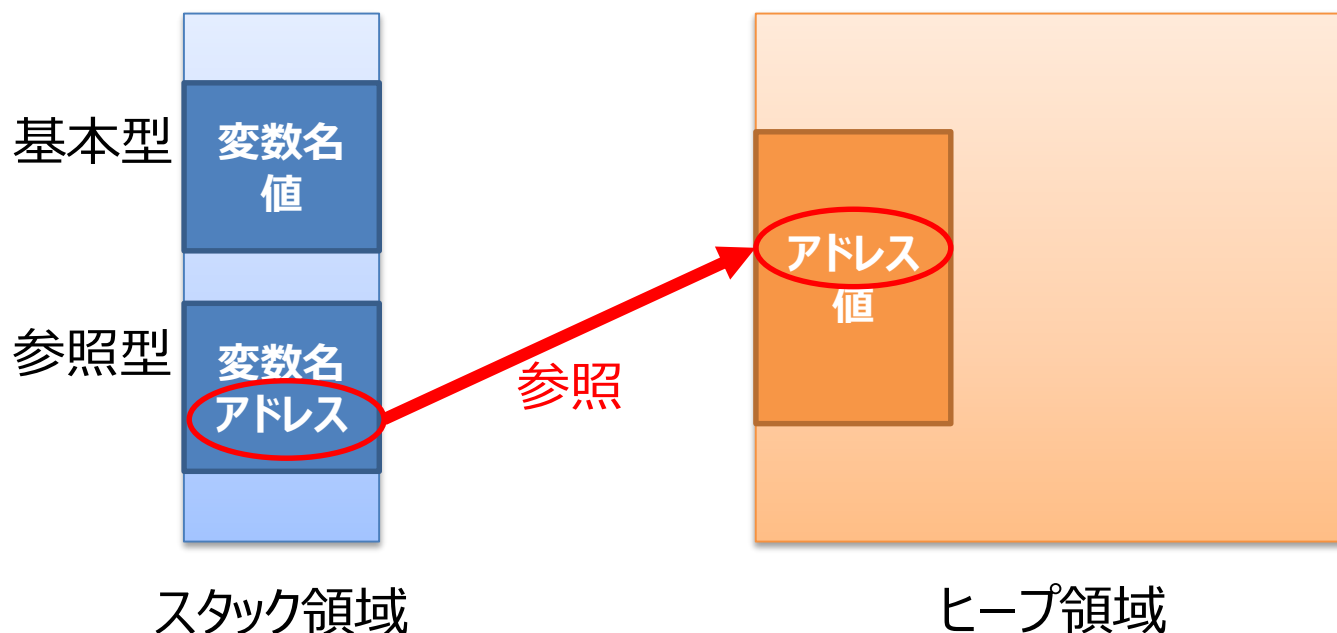
## 変数が保持される場所

「基本型（プリミティブ型）」と「参照型」は、メモリへの保存され方が異なる。

- 「基本型（プリミティブ型）」 → スタック
- 「参照型」 → スタック+ヒープ

※ JavaVMは、メモリを主に2つの領域に分けてプログラムを実行する。

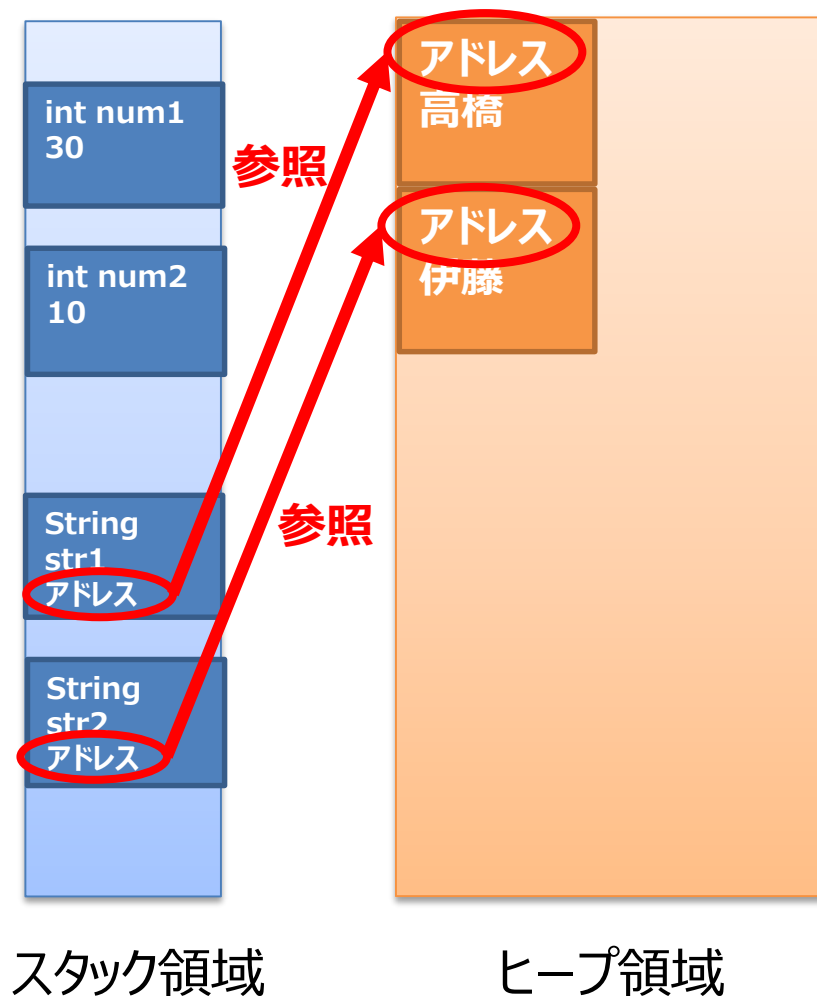
※ 小さなメモリ領域をスタック、大きなメモリ領域をヒープという。





## 基本型と参照型、スタックとヒープのイメージ

```
class Sample {  
    public static void main(String[] args) {  
        int num1;  
        num1 = 30;  
        int num2 = 10;  
  
        String str1;  
        str1 = "高橋";  
        String str2 = "伊藤";  
    }  
}
```



基本型は値をスタックに持つ

参照型は値をヒープに、スタックはヒープを参照するアドレスを持つ

※ 簡単にいうとスタックは小さなメモリ、ヒープは大きなメモリ

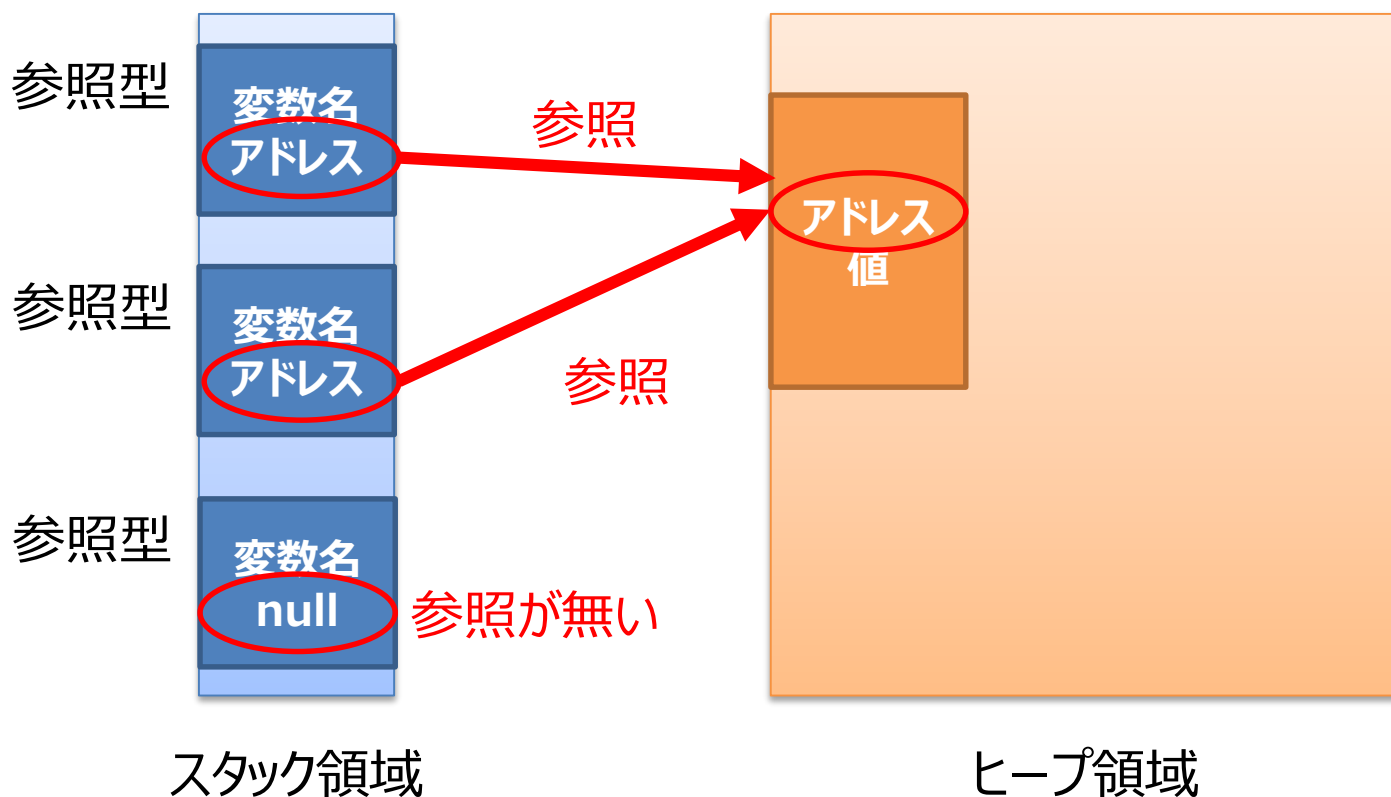
※ 先程の例で「String型（文字列）」は文字列の長さ(データサイズ)が分からない  
→ヒープを使う

※「基本型（プリミティブ型）」は大きさ(データサイズ)が分かっている  
→スタックを使う

## 参照型の注意事項

別の変数で同じ参照先の時の動きに注意

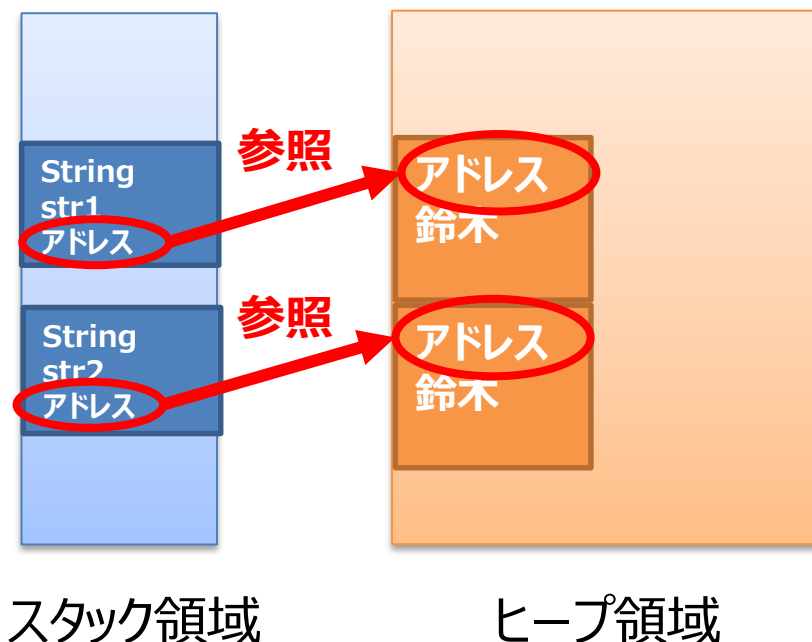
**null** (どこも参照していない)



## Javaの文字列

Javaでは1文字はchar型になり基本型(プリミティブ型)になるが、文字列はString型(参照型)になる。

- 同じ文字列でも、別々の参照の場合がある
- Stringクラスのメソッド(API)が利用できる



Javaで文字列を操作する場合は、主に以下のクラスを使う。

Stringクラス

StringBuilderクラス

① 同じ文字列内容か調べる。

```
String s1, s2;  
boolean b = s1.equals(s2);
```

② 文字列の長さを調べる。

```
String s1;  
int len = s1.length();
```

③ 指定位置の文字を取り出す。

```
String s1;  
char c = s1.charAt(0);
```

④ 指定書式の文字列を得る。

```
String str;  
str = String.format("%s%c%d%5d%05d", "文字列", 'A', 100, 200, 300);
```

⑤ 文字列を連結する。その1

```
String s1, s2, s3;  
s3 = s1 + s2;
```

⑥ 文字列を連結する。その2

```
StringBuilder sb = new StringBuilder();  
sb.append(s1);  
sb.append(s2);
```

⑦ 他のデータ型の文字列表現を得る。

```
String s1 = String.valueOf(100);
```

## 定数(リテラル) (1/2)

変数と同様、プログラムで扱うデータを格納しておくが、変更しないもの。

キーワード：**final**を付けることで定数となる

例)

```
int num;                // 整数型の変数の宣言
final int num = 10       // 整数型の定数の宣言
```

例)クラス変数の場合(後の講義で詳しく説明します)

```
public final int num = 10        // final修飾子のみで定数宣言できますが
public static final int num = 10 // 定数の性質上staticにすべきです。
```

## 定数(リテラル) (2/2)

変数に代入する値などは、直接値を書きますが、Javaではこれらも定数(リテラル)です。

### 定数 (リテラル) の書き方

- 整数 (10進数、16進数など)
- 少数
- 文字 (シングルクォートで括る、UNICODE表記、エスケープシーケンス (¥n, ¥', ¥", ¥¥) )
- 文字列 (ダブルクォートで括る)
- 論理型 (true, false)



変数を使って計算したり、変数に値を代入したりする際に利用するもの  
「評価結果」を持つ

## 算術演算子（加減乗除＋余）

$+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$

評価結果：2つの項の算術結果

## インクリメント・デクリメント

$++$ ,  $--$

評価結果：前置き 1加算・1減算した後の結果

評価結果：後置き 1加算・1減算する前の結果

※ あまり他の演算子と一緒に使わないようにする方が良い。紛らわしいため。

## 代入演算子・・・代入も演算

=

評価結果：代入した結果（要するに左辺）

## 複合代入演算子

`+=, -=, *=, /=, %=`

評価結果：代入した結果（要するに左辺）

## 関係演算子

<, >, <=, >=, ==, !=

評価結果：真偽値 真true または 偽false

## 論理演算子

&&, ||, !

評価結果：真偽値 真true または 偽false

※「a && b」でaがfalseだとbは評価されない。

※「a || b」でaがtrueだとbは評価されない。

## 3項演算子

条件式 ? 式A : 式B

評価結果：条件式が真trueの場合式A、偽falseの場合式B

例： a = x ? b : c ;

## 演算子の優先順位と結合順

ここまでは1つずつ見てきたが、実際には演算子を複数書くことは多い。  
その際にどの演算子が優先されるのかは重要。

優先度高		結合順
	(引数) [配列添字] . x++ x--	左から右
	! ~ +x -x(単項演算子) ++x --x	右から左
	new (型)x	右から左
	* / %	左から右
	=+ -(算術演算子)	左から右
	<< >> >>>	左から右
	> >= < <= instanceof	左から右
	== !=	左から右
	&	左から右
	^	左から右
		左から右
	&&	左から右
		左から右
	?:	右から左
	= ope=	右から左
優先度低		

乗算のほうが加算より優先順位が高い例を思い出してください。  
主要なものを覚えて、全て覚える必要は無いです。  
不安を覚えたら技術書で確認する癖をつけましょう。  
場合（分かりやすくするため等）によってはカッコをつけましょう。

文には「**順次**」「**選択**」「**繰り返し**」「**コメント**」があり、これを組み合わせてプログラムの流れを作成する。

## コメント

コンパイル時（および実行時も）に無視され何も実行されない文。

何に使うのか？

- プログラムの意味などを記述する
- 一時的に実行しないようにする など

行コメント：     //

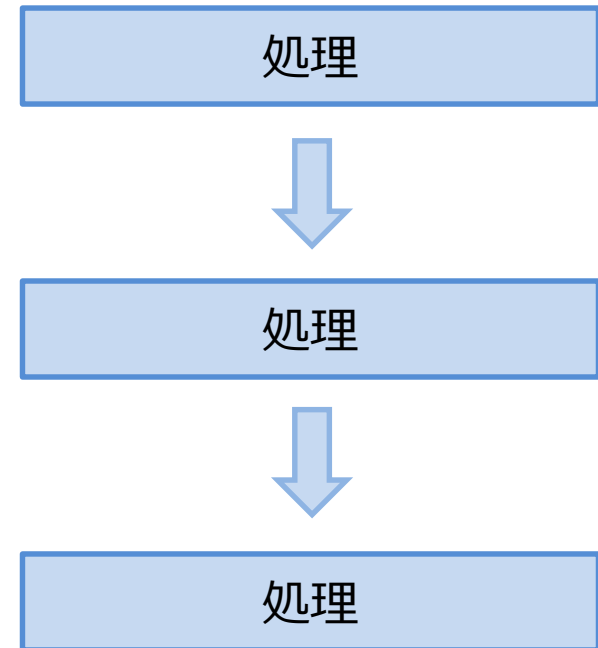
ブロックコメント： /\* ~ \*/

※ブロックコメントの入れ子は出来ません。

## 順次

処理を連続して実行させる。  
文の終わりは**セミコロン**となる。  
セミコロンだけの文も可能。

例 : `a = a + 1;`



## 選択

条件に従って分岐させる。

条件式の評価結果は必ず真偽値でなければならない。

### 選択その1 if文

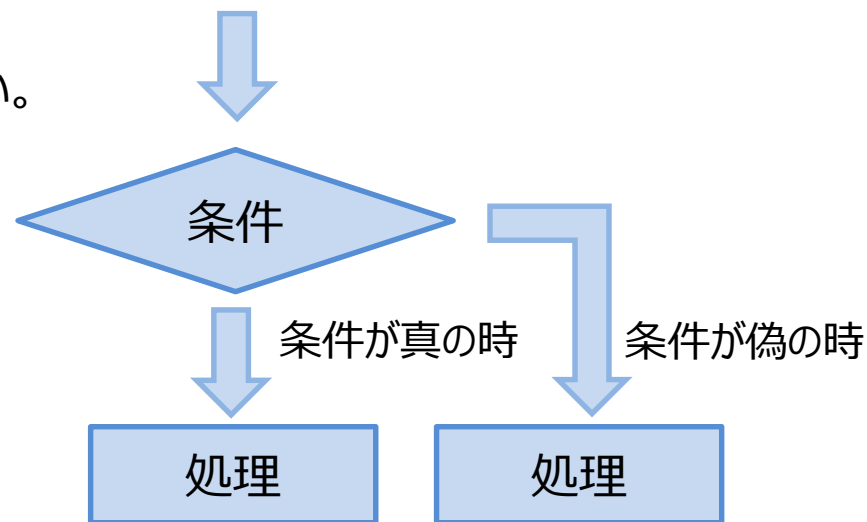
条件の真偽によって分岐させたい。

中括弧で括った範囲をブロック（節）という。

elseブロックは省略可能。

文が1つしかない場合はブロックの中括弧も省略可能。

else ifブロックを使って複数の条件を続けて書くことも可能。





## 選択その2 switch文

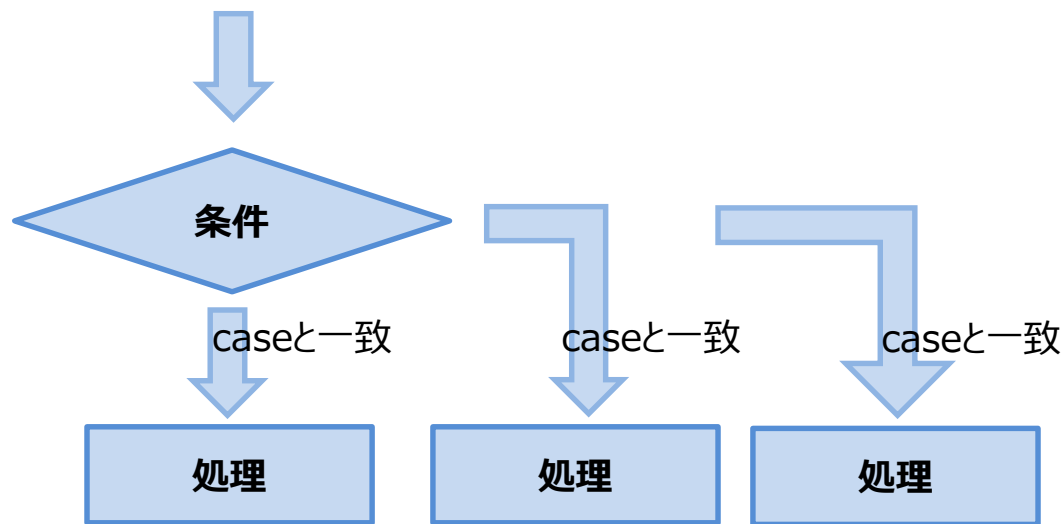
同じ条件式で比較したい結果が複数ある場合に利用する。

breakを書かなかった場合switch文を抜けずに下にある文を実行することになる。

あえてbreakを書かないこともあるが基本的にはbreakは書く。

「case 定数1:」... 末尾はコロン。

「default:」... caseのいずれのケースにも当てはまらない場合



## 繰り返し

条件を満たす間繰り返しを行う。

条件式の評価結果は必ず真偽値でなければならない。

繰り返しの文を書くときは必ず繰り返しを終了することが出来るか確認してください。

繰り返しが終了できないと無限に繰り返すことになります。（無限ループ）

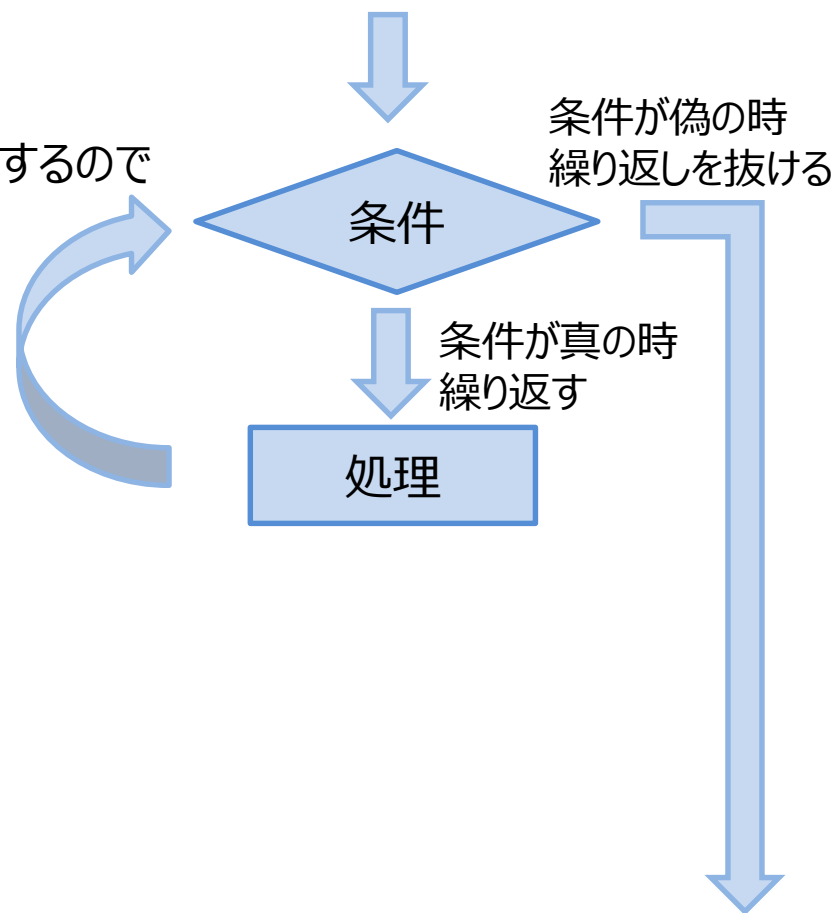
## 繰り返しその 1 **while**文（前判定）

条件式を満たす間繰り返す。

`break`（繰り返しを抜ける）/`continue`（繰り返しの最初に）で繰り返しを抜けることも出来るが、基本的には条件式を満たさなくなった時に抜ける形式で書く。

※可読性が低くなる為

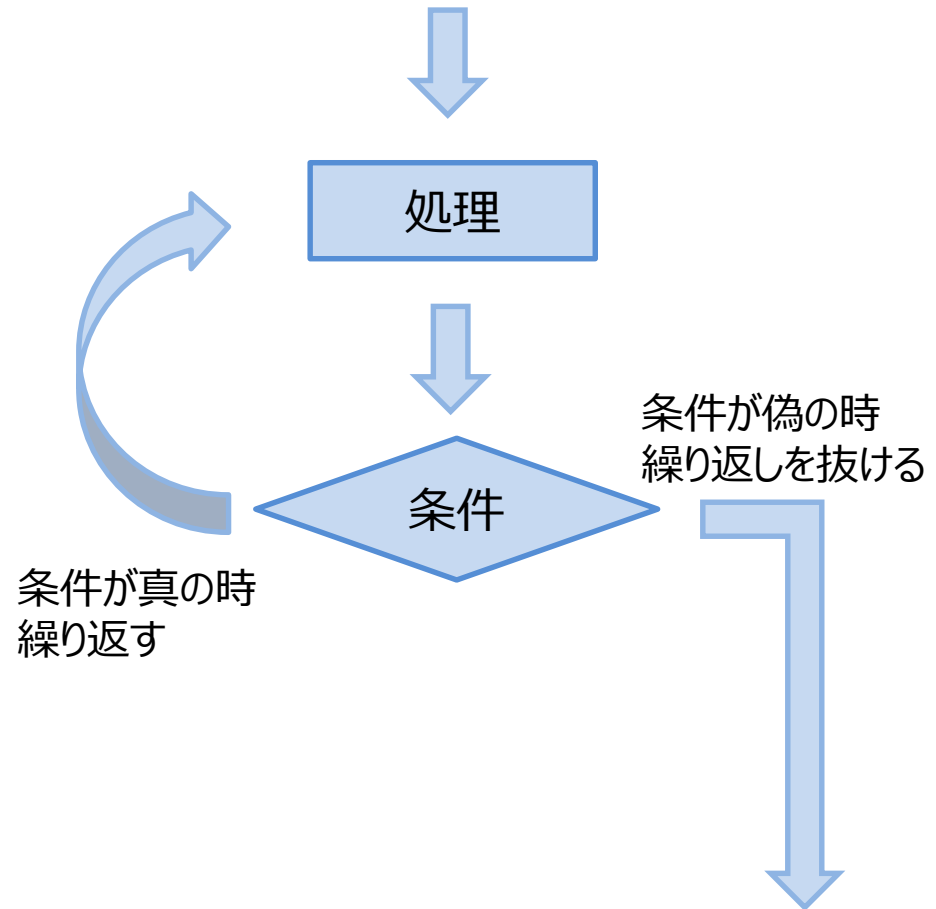
条件式の評価を行ってから処理を繰り返すか判定するので  
処理が一度も実行されない場合もある。



## 繰り返しその2 **do-while文**（後判定）

while文と同じく、条件式を満たす間繰り返す。

しかし必ず一回は処理を行う。

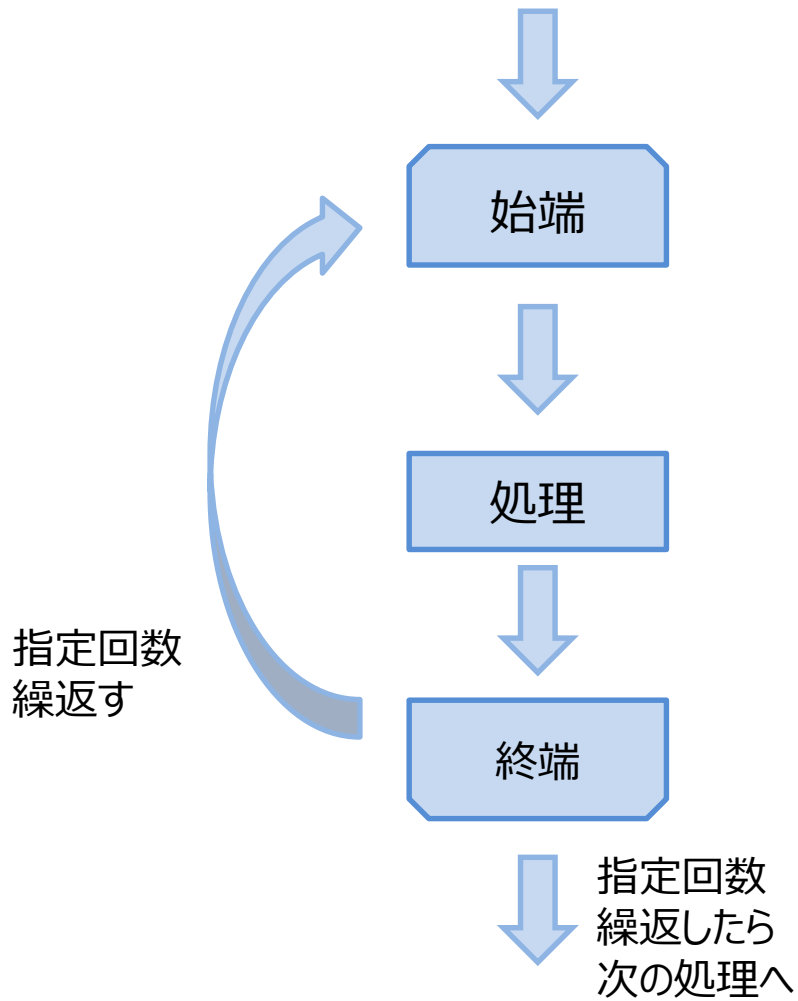


## 繰り返しその3 **for**文

指定回数繰り返す。

while文, do while文と同じく、  
break（繰り返しを抜ける）  
continue（繰り返しの最初に）  
で繰り返しを抜けることも出来るが、  
基本的には指定回数実行したら  
抜ける形式で書く。

※可読性が低くなる為



## 繰り返しその4 拡張for文

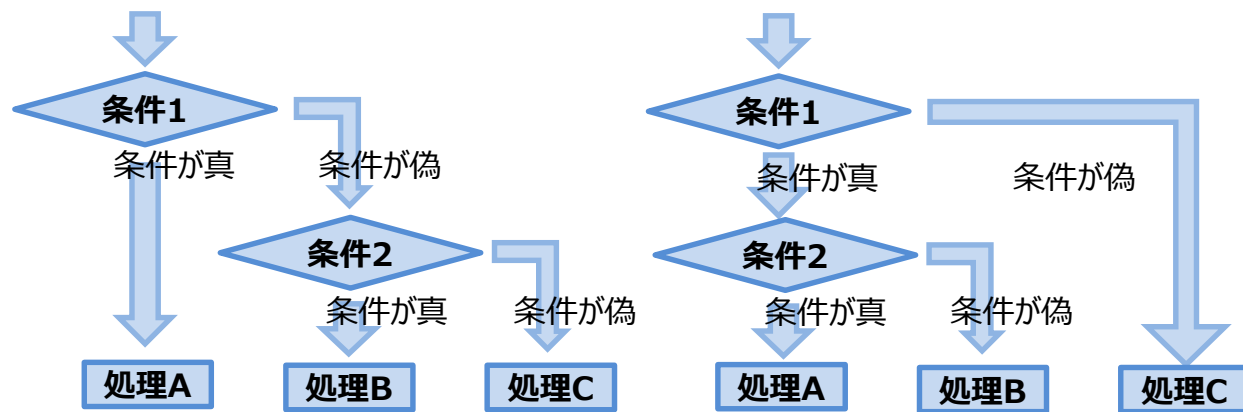
コレクション(後々の講義)で説明。

## ネスト(入れ子)構造

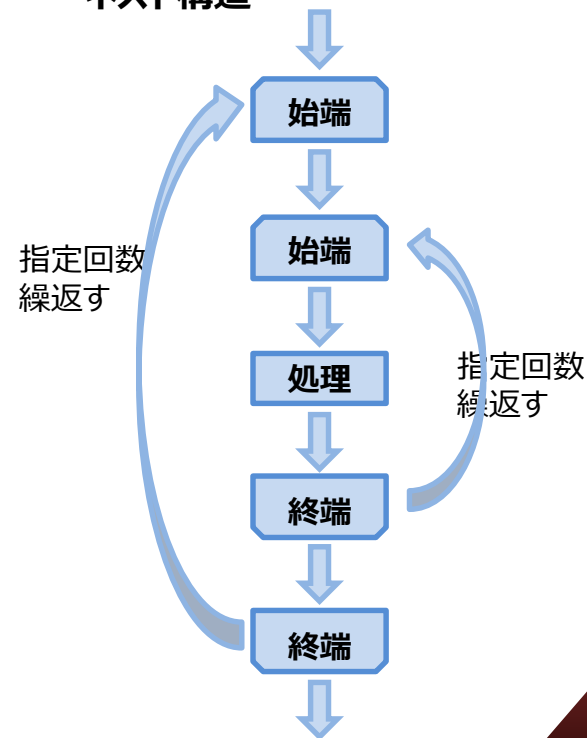
選択ステートメントや繰り返しステートメントのブロック内に、再度、選択ステートメントや繰り返しステートメントを入れる事ができる。

## ネスト(入れ子)構造

選択ステートメントの  
ネスト構造



繰り返しステートメントの  
ネスト構造



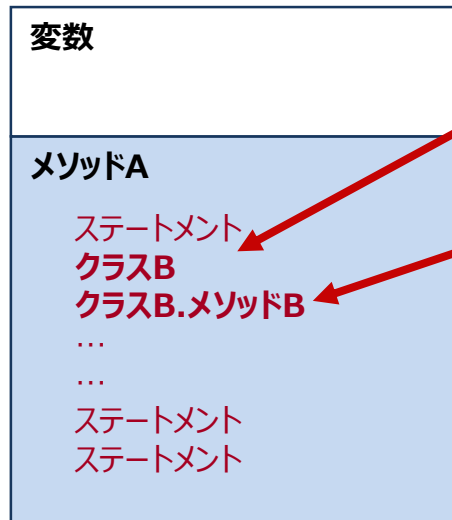
## クラスとメソッド

Javaでは必ずクラスというものを作成する。

クラスは、そのクラスの変数と、処理をまとめたメソッドを持つのが、基本的な形となる。

詳細については次の講義で説明。

クラスA



クラスB

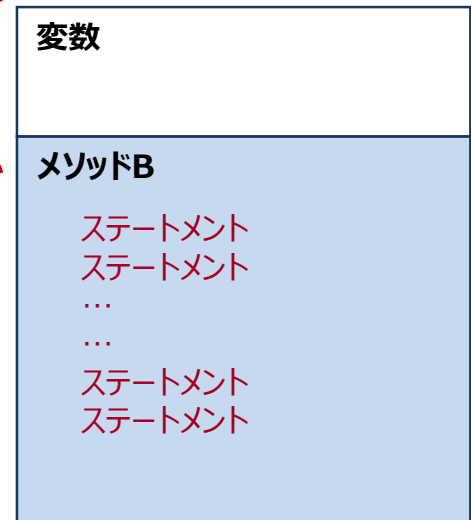


Diagram illustrating the relationship between Class A and Class B. Red arrows indicate that Class A contains references to Class B. Specifically, the arrows point from the text 'クラスB' and 'クラスB.メソッドB' in the 'メソッドA' section of Class A to the corresponding 'メソッドB' section of Class B.