Java 学習教材

筑波大学 システム情報系 三谷純
最終更新日 2023/4/19

　　図表出典：三谷純著『プログラミング学習シリーズ Java』（翔泳社 刊）

本資料の位置づけ

本資料は

『Java 第3版 入門編
ゼロからはじめるプログラミング』

を専門学校・大学・企業などで教科書として採用された教員・指導員を対象に、教科書の内容を解説するための副教材として作られています。

上記に該当する場合は、自由にご使用ください。授業の進め方などに応じて、改変していただいて結構です。※ このページを削除して構いません

ただし、民間企業が商用、ビジネス目的で利用する際には別途許諾が必要ですので、著者までご連絡ください。

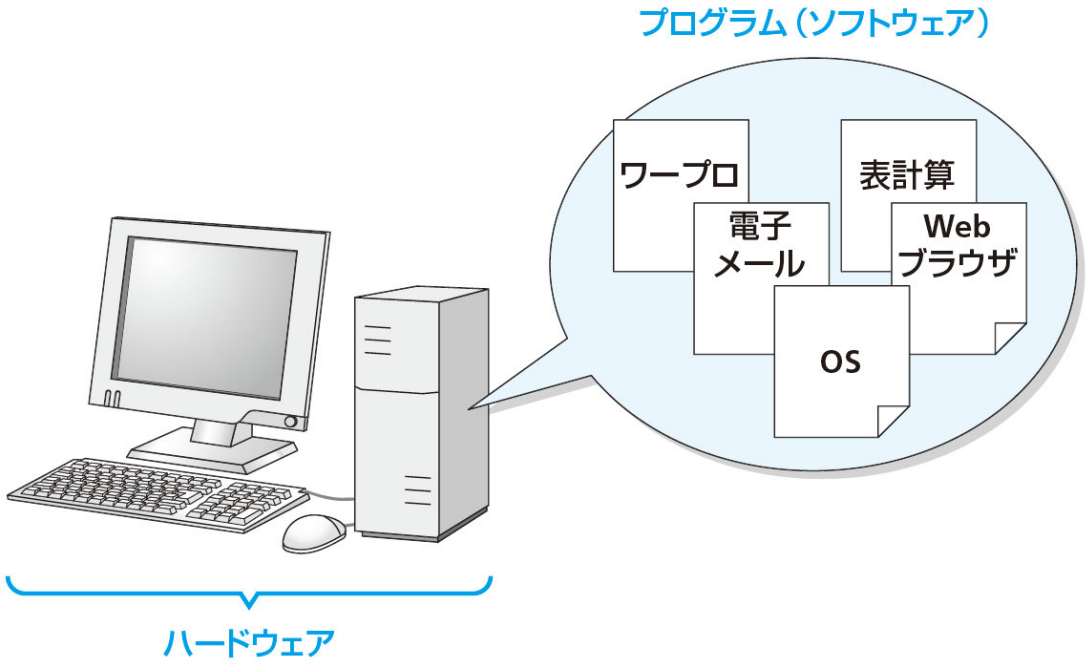
Java 第3版 入門編
ゼロからはじめるプログラミング
（三谷純 著）

出版社： 翔泳社
発売日： 2021/1/18
ISBN ： 9784798167060

第1章 Java言語に触れる

プログラムとは

- コンピュータに命令を与えるものが「プログラム」
- プログラムを作成するための専用言語が「プログラミング言語」
- その中の1つに「Java言語」がある



Java言語のプログラムコード

Java言語のプログラムコードを見てみよう

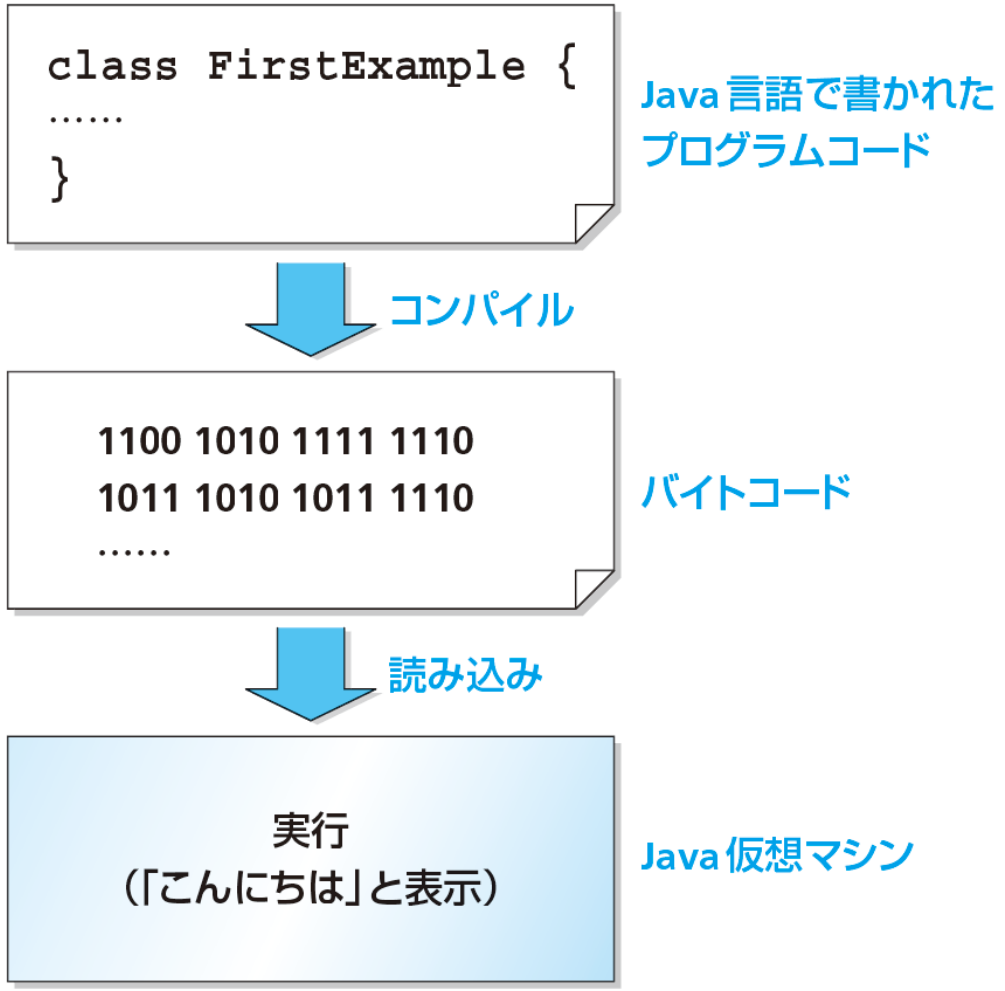
```
class FirstExample {
  public static void main(String[] args) {
    System.out.println("こんにちは");
  }
}
```

↑「こんにちは」という文字を画面に表示するプログラムのプログラムコード

- 半角英数と記号で記述する
- 人が読んで理解できるテキスト形式

プログラムコードが実行されるまで

1. プログラムコードがコンパイルされてバイトコードが作られる
2. バイトコードがJava仮想マシンによって実行される



Java言語の特徴

- コンパイラによってバイトコードに変換される
- バイトコードがJava仮想マシンによって実行されるので、WindowsやMac OS、Linuxなどの各種OS上でコンパイルし直さずに動作する
- オブジェクト指向型言語

Java言語のプログラム構成

```
public class クラス名 {
  public static void main(String[] args) {
    命令文
  }
}
```

- クラス名は自由に設定できる。先頭の文字はアルファベットの大文字
  例：Example

- `public static void main(String[] args) { }`
  の { } の中に命令文を書く

Java言語のプログラム構成

```
public class FirstExample {
  public static void main(String[] args) {
    System.out.println("こんにちは");
  }
}
```

- 命令文の末尾にはセミコロン(;)をつける
- 空白や改行は好きなところに入れられる
- 大文字と小文字は区別される

ブロックとインデント

```
public class FirstExample {
（インデント）public static void main(String[] args) {
        （インデント）    System.out.println("こんにちは");
（インデント）}
}
```

- { と } は必ず1対1の対応を持っている
- { } で囲まれた範囲を「ブロック」と呼ぶ
- プログラムコードを見やすくするための先頭の空白を「インデント」と呼ぶ

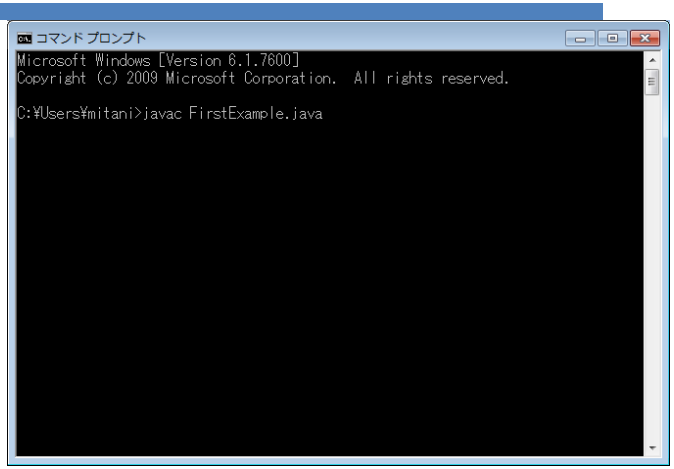
コメント文

```
/*
    こんにちはという文字を画面に表示するプログラム
    作成日：2017年1月1日
*/
class FirstExample {
  public static void main(String[] args) {
    // 画面に文字を表示する
    System.out.println("こんにちは");
  }
}
```

- プログラムコードの中のメモ書きを「コメント」と呼ぶ
- 方法1 /* と */ で囲んだ範囲をコメントにする
- 方法2 // をつけて、1行だけコメントにする

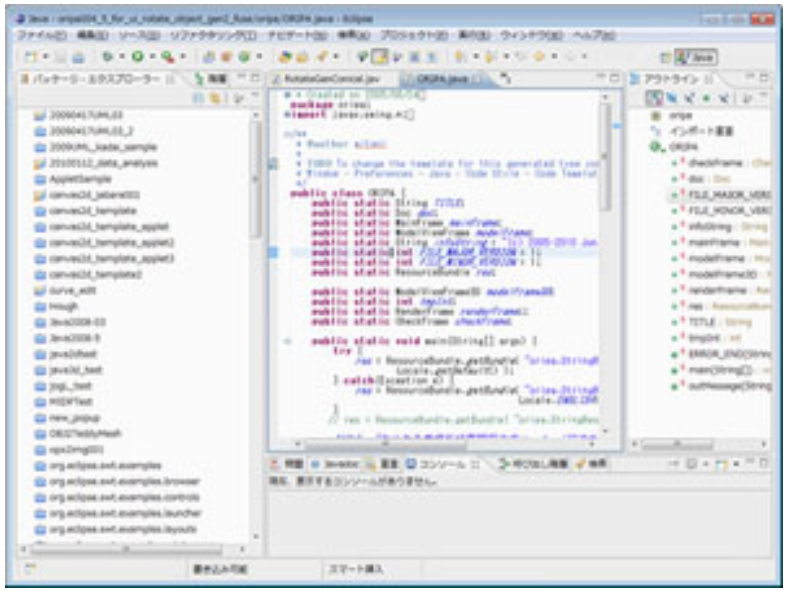
プログラムの作成

【方法1】
コマンドラインでコンパイルして実行する




```
> javac FirstExample.java   ←コンパイル
> java FirstExample         ←実行
   こんにちは                ←実行結果
```

【方法2】
Eclipseなどの統合開発環境を使用する



Eclipseでの実行の手順

1. プロジェクトを作成する
   （[ファイル]→[新規]→[Javaプロジェクト]）

2. プログラムコードを作成する
   （[ファイル]→[新規]→[クラス]）

3. プログラムの実行
   （[実行]→[実行]→[Javaアプリケーション]）

エラー（Compile Error）が発生したら

- キーワードの綴りミス、文法上の誤りが原因
- 単純なミスに気を付ける

  全角の文字、空白を使用しない

  似た文字の入力間違い

  ゼロ(0) 小文字のオー(o) 大文字のオー(O)
  イチ(1) 大文字のアイ(I)　小文字のエル(l)
  セミコロン(;) コンマ(:)
  ピリオド(.)　カンマ(,)

.javaファイルと.classファイル

- プログラムコードは拡張子が.javaのファイルに保存する
  例：FirstExample.java

- プログラムコードをコンパイルすると拡張子が.classのファイルが生成される
  例：FirstExample.class

- Eclipseでは、最初に指定したworkspaceフォルダの中に自動生成される

演習

1. Java言語の歴史についてインターネットで調べてみよう
2. 実際にJavaプログラムが使用されているシステムにはどのようなものがあるかインターネットで調べてみよう
3. FirstExample.java を入力し、実際に動かしてみよう
4. .javaファイルと.classファイルがどこにあるか確認してみよう

第2章 Java言語の基本

出力

文字列を標準出力（Eclipseの場合はコンソールビュー）に出力する命令

```
System.out.println(出力する内容);
```

実際のコード

```
class FirstExample {
  public static void main(String[] args) {
    System.out.println("こんにちは");
  }
}
```

エスケープシーケンス

特別な記号や出力方法を制御するために
記号（¥）を使う。 環境によっては（＼）記号

```
例：System.out.println(
        "これから¥"Java言語¥"を学習します。");
```

記号の組み合わせ	意味
¥'	'
¥"	"
¥¥	¥
¥n	改行（ラインフィード：LF）
¥t	水平タブ
¥r	復帰（キャリッジリターン：CR）

演習

1. 自分の名前を出力してみよう
2. 複数の`System.out.println`命令文を入れて、実行結果を確認しよう
3. `System.out.println(2+3);` と入れたらどうなるか確認しよう

変数

「変数」とは、値を入れておく入れ物

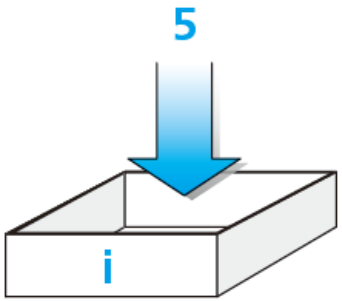
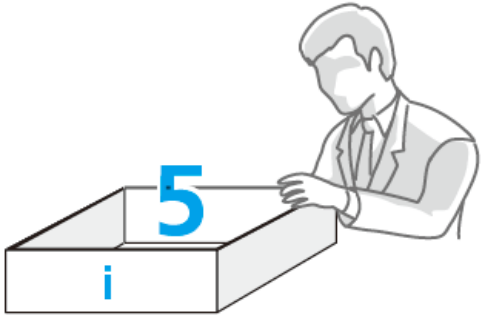
```
int i; // 変数の宣言
i = 5; // 値の代入
System.out.println(i); // 値の参照
```

変数の宣言：変数を作成すること

値の代入：変数に値を入れること

値の参照：変数に入っている値を見ること

変数の使用

命令文	意味
`int i;`	 iという名前の入れ物を作成します。
`i = 5;`	 iという名前の入れ物に5を入れます。
`System.out.println(i);`	 iという名前の入れ物の中身を見て、その中身をコンソールに出力します。

変数の宣言と型

変数の宣言では、変数に入れる値のタイプ（型）をはじめに指定する。

構文：型名 変数名;

例1 `int i;`
例2 `double d;`
例3 `boolean boo = false;`
例4 `char c = 'あ';`

Javaで使用できる型

型	値の例	格納できる値の範囲
char	'a', 'b', 'c', … 'あ', 'い', …	1文字（16ビット、Unicode文字）
boolean	true, false	真偽値。true（真）または false（偽）のどちらか
byte	整数（…, -1, 0, 1, …）	8ビット符号付き整数。 -2^7 (-128) ～ 2^7-1 (127)
short		16ビット符号付き整数。 -2^{15} (-32,768) ～ $2^{15}-1$ (32,767)
int		32ビット符号付き整数。 -2^{31} (-2,147,483,648) ～ $2^{31}-1$ (2,147,483,647)
long		64ビット符号付き整数。 -2^{63} (-9,223,372,036,854,775,808) ～ $2^{63}-1$ (9,223,372,036,854,775,807)
float	小数点を含む数値 （… -0.5, …, 0.0, …, 0.5, …）	32ビット符号付き浮動小数点数（注❷-8）
double		64ビット符号付き浮動小数点数

演習

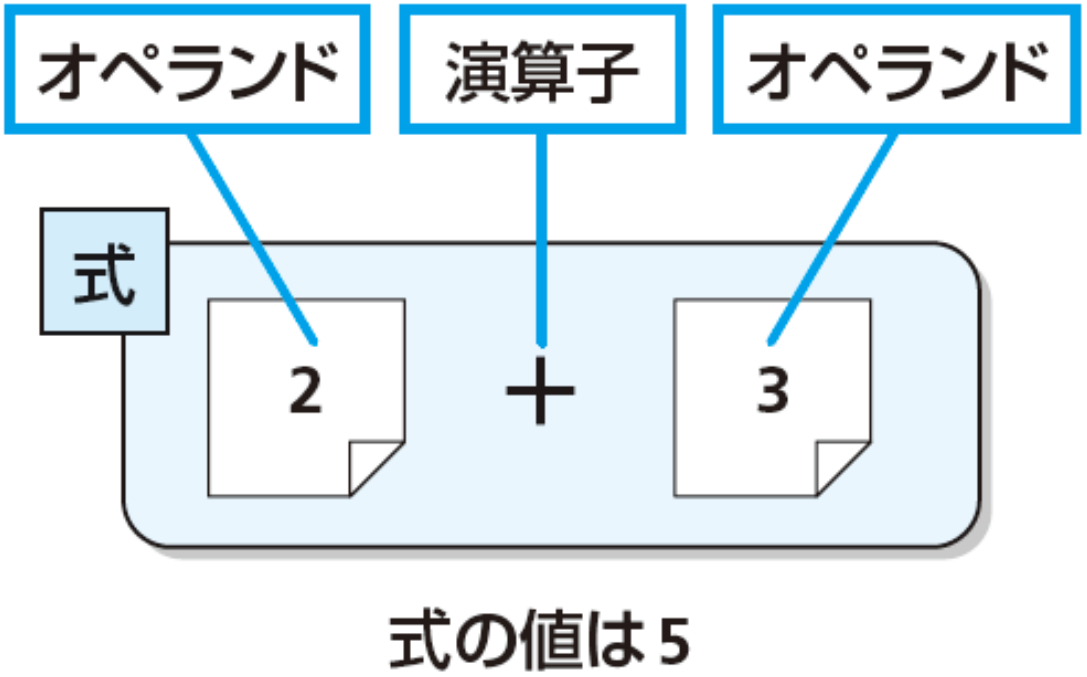
次のプログラムコードの赤字部分を様々に変更して実行してみよう
例：double型、boolean型、char型

```
class Example {
  public static void main(String args[]) {
    int i;
    i = 5;
    System.out.println(i);
  }
}
```

算術演算子と式

算術演算子を用いた計算

```
System.out.println(2 + 3);
```



算術演算子と優先順位

演算子	演算の内容	説明	使用例
+	加算（足し算）	左辺と右辺を足します	1 + 2（式の値は3）
-	減算（引き算）	左辺から右辺を引きます	2 - 1（式の値は1）
*	乗算（掛け算）	左辺と右辺を掛けます	2 * 3（式の値は6）
/	除算（割り算）	左辺を右辺で割ります	4 / 2（式の値は2）
%	剰余	左辺を右辺で割った余りを求めます	7 % 3（式の値は1）

数学と同じように、加算と減算（+,-）より乗算と除算（*,/）が優先される

```
System.out.println(3 + 6 / 3);   // 5
System.out.println((3 + 6) /3); // 3
```

演習

次のプログラムコードの赤字部分を変更して、
様々な計算をしてみよう

例：加算、減算、乗算、除算、剰余

```
class Example {
  public static void main(String args[]) {
    System.out.println(2 + 3);
  }
}
```

変数を含む算術演算子

```
int i = 10;
int j = i * 2;
System.out.println(j); // 20
```

```
int i = 10;
i = i + 3;
System.out.println(i); // 13
```

```
int i = 10;
i += 3; // 短縮表現
System.out.println(i); // 13
```

演算子	演算の内容	説明	使用例
+=	加算代入	左辺の変数の値と右辺の値を足した値を、左辺の変数に代入します	a += 2 （変数aの値は2増えます。a = a + 2と同じです）
-=	減算代入	左辺の変数の値から右辺の値を引いた値を、左辺の変数に代入します	a -= 3 （変数aの値は3減ります。a = a - 3と同じです）
*=	乗算代入	左辺の変数の値に右辺の値を掛けた値を、左辺の変数に代入します	a *= 2 （変数aの値は2倍になります。a = a * 2と同じです）
/=	除算代入	左辺の変数の値を右辺の値で割った値を、左辺の変数に代入します	a /= 3 （変数aの値は3分の1になります。a = a / 3と同じです）
%=	剰余代入	左辺の変数の値を右辺の値で割った余りを、左辺の変数に代入します	a %= 2 （変数aの値はそれを2で割った余りになります。a = a % 2と同じです）
++	インクリメント	左辺の変数の値を1増やします	a++ （変数aの値は1増えます。a = a + 1と同じです）
--	デクリメント	左辺の変数の値を1減らします	a-- （変数aの値は1減ります。a = a - 1と同じです）

演習

次の命令文を短い表現に書き換えてみよう

```
1.  a = a + 5;
2.  b = b - 6;
3.  c = c * a;
4.  d = d / 3;
5.  e = e % 2;
6.  f = f + 1;
7.  g = g - 1;
```

演習

次のプログラムコードの実行結果を予測し、確認しよう

```
class CalcExample3 {
  public static void main(String[] args) {
    int i;
    i = 11;
    i++;
    i /= 2;
    System.out.println("iの値は" + i);

    int j;
    j = i * i;
    System.out.println("jの値は" + j);
  }
}
```

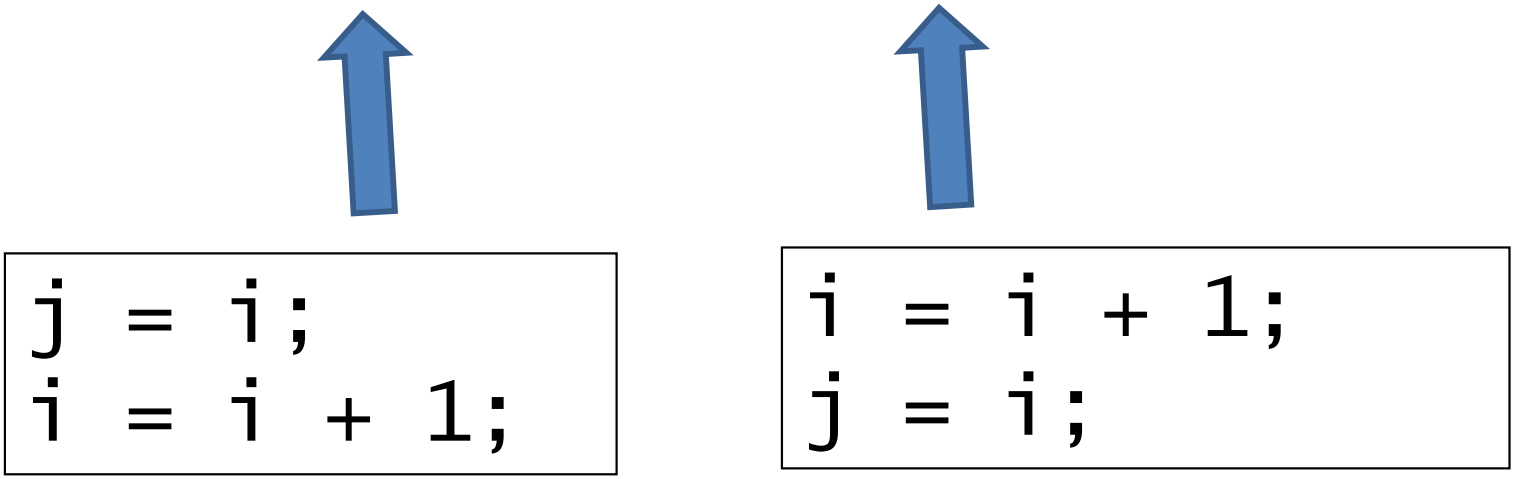
ワン・モア・ステップ（文と式）

- 「`i = 2 + 3;`」は文
- 「`i = 2 + 3`」は式（代入式）
- 式は値を持つ
- 代入式は左辺に代入される値を持つ

```
int i;
int j = (i = 2 + 3) * 2;
System.out.println(i); // 5
System.out.println(j); // 10
```

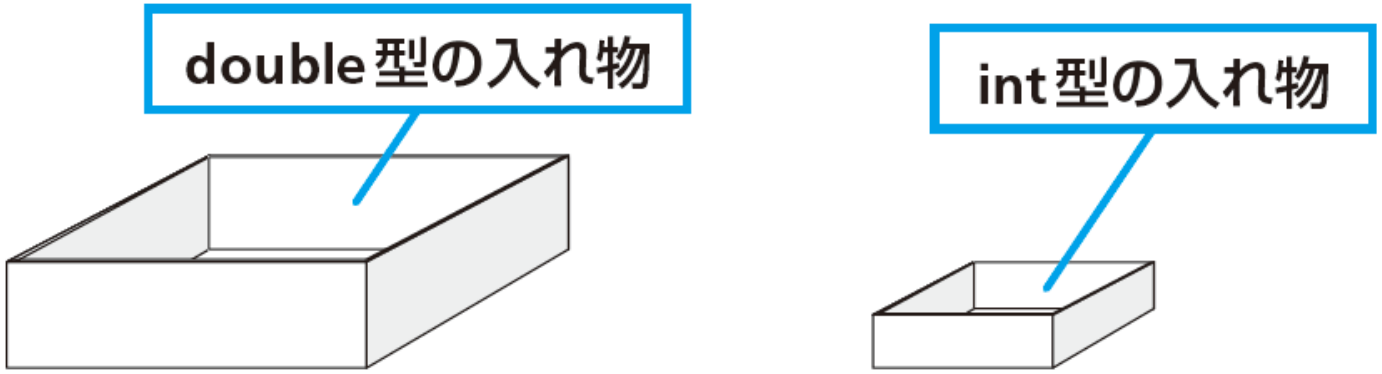
ワン・モア・ステップ（前置と後置）

- 後置「`i++;`」
- 前置「`++i;`」
- どちらも`i`の値を1だけ増やす
- 「`j=i++;`」と「`j=++i;`」では`j`の値が異なる。



型と大きさ

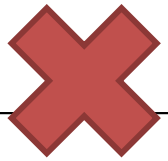
型によって変数の大きさが異なる



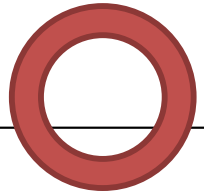

```
double > int
```

型変換

```
double d = 9.8;
int i = d;
```



```
double d = 9.8;
int i = (int)d;
```



- 大きな変数(double)の値を小さな変数(int)に代入できない
- カッコを使って型変換できる
- 型変換を「キャスト」とも呼ぶ

異なる型を含む演算

```
int i = 5;
double d = 0.5;
System.out.println(i + d); // 5.5
```

型の異なる変数や値の間で演算を行った場合は、最も大きい型（上の例ではdouble型）に統一されて計算される

整数同士の割り算

```
int a = 5;
int b = 2;
double c = a / b;
System.out.println(c); // 2.0
```

- 整数と整数の割り算は整数型として扱われる。
  上の例では 5/2 が 2 になる
- 正しい値を求めるには、double型にキャストする必要がある
  例：`double c = (double)a/(double)b;`

演習

7÷2の計算結果が正しく3.5になるように修正しよう

```
class Example {
  public static void main(String[] args) {
    int a = 7;
    int b = 2;
    double d = a / b;
    System.out.println(d);
  }
}
```

String 型

文字列はString型の変数に代入できる

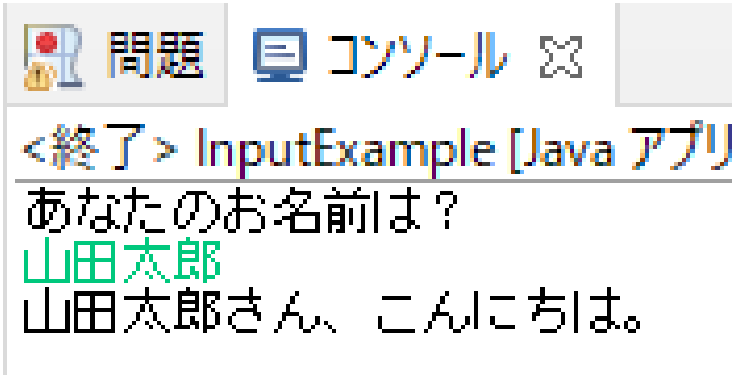
```
String s;
s = "こんにちは";
System.out.println(s);
```

文字列は「+」演算子で連結できる。

```
String s1 = "こんにちは。";
String s2 = "今日はよい天気ですね。";
String s3 = s1 + s2;
System.out.println(s3);
```

キーボードからの入力を受け取る

キーボードからの入力を受けとることで対話的なプログラムを作成できる




```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("あなたのお名前は？ ");
        String name = in.next();
        System.out.println(name + "さん、こんにちは。");
        in.close();
    }
}
```

コンソールに入力された文字列を受け取り、変数nameに代入します

コンソールからの受け取りを終了します

キーボードからの入力の受け取り方

1. 先頭行に次のように記述する

```
import java.util.Scanner;
```

2. main を含む行の下に次のように記述する

```
Scanner in = new Scanner(System.in);
```

3. 入力を受け取りたいところに次のように記述する

文字列を受け取る場合

```
String s = in.next();
```

整数を受け取る場合

```
int i = in.nextInt();
```

小数点を含む数を
受け取る場合

```
double d = in.nextDouble();
```

第3章 条件分岐と繰り返し

条件分岐

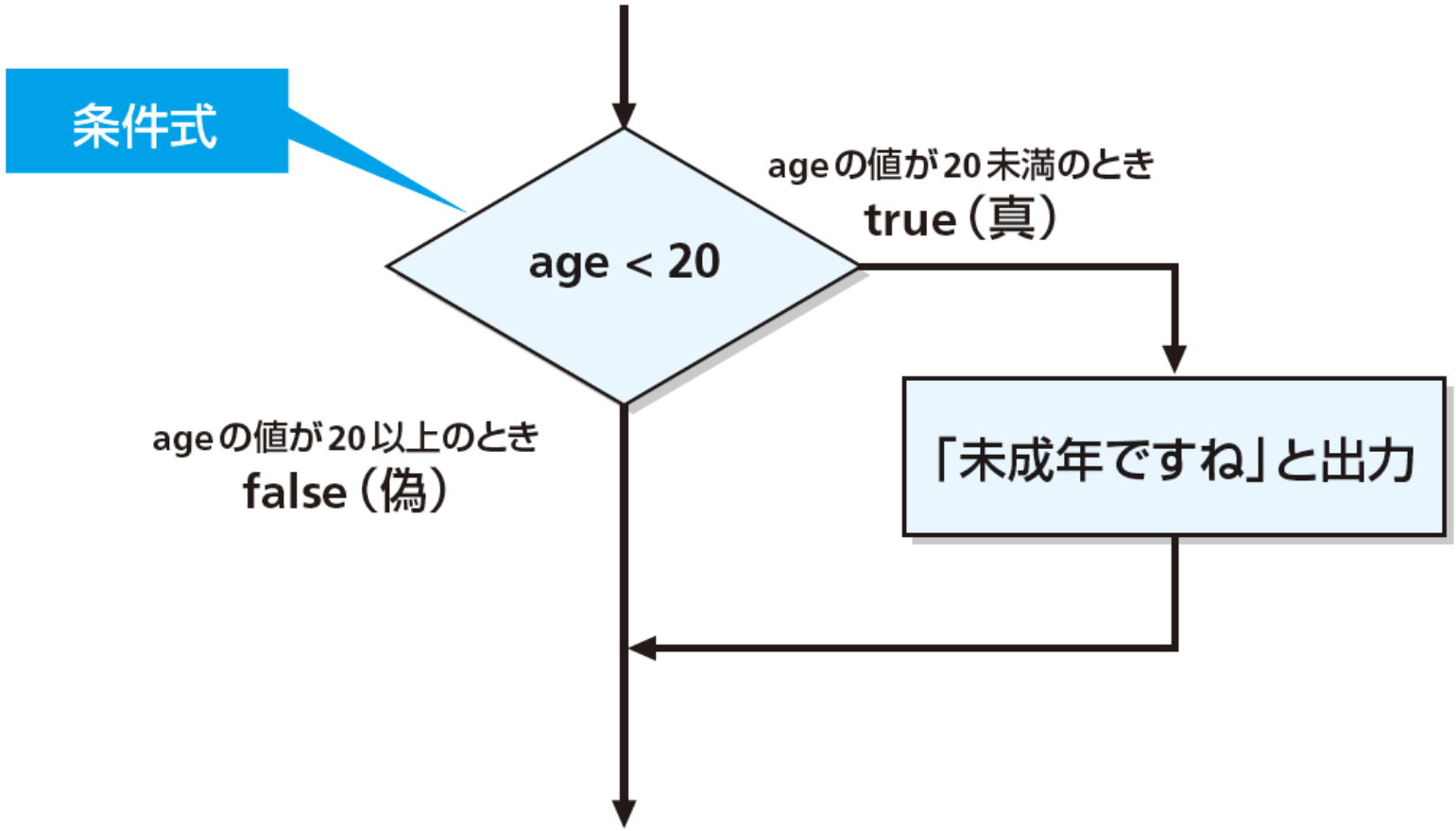
条件による処理の分岐
「もしも○○ならば××を実行する」

```
if(○○) {
    ××;
}
```

```
if(条件式) {
    命令文; //条件式がtrueの場合に実行される
}
```

条件分岐の例

```
if(age < 20) {
    System.out.println("未成年ですね");
}
```



関係演算子

- 関係演算子を使って、2つの値を比較できる
- 比較した結果は true または false になる

演算子	説明	例
==	左辺と右辺が等しい	a == 1（変数aが1のときにtrue）
!=	左辺と右辺が等しくない	a != 1（変数aが1でないときにtrue）
>	左辺が右辺より大きい	a > 1（変数aが1より大きいときにtrue）
<	左辺が右辺より小さい	a < 1（変数aが1より小さいときにtrue）
>=	左辺が右辺より大きいか等しい	a >= 1（変数aが1以上のときにtrue）
<=	左辺が右辺より小さいか等しい	a <= 1（変数aが1以下のときにtrue）

演習

次の条件を満たす時に命令文が実行されるような条件式を作成しよう

1. 変数 a の値が 20 である
2. 変数 a の値が 20 でない
3. 変数 a の値が正である
4. 変数 a の値が負である
5. 変数 a の値が 3 の倍数である
6. 変数 a の値が偶数である
7. 変数 a の値を 5 で割った余りが 2 である

if ～ else 文

「もしも○○ならば××を実行し、そうでなければ△△を実行する」

```
if(○○) {
    ××;
} else {
    △△;
}
```

```
if(条件式) {
    //条件式がtrueの場合
    命令文1;
} else {
    //条件式がfalseの場合
    命令文2;
}
```

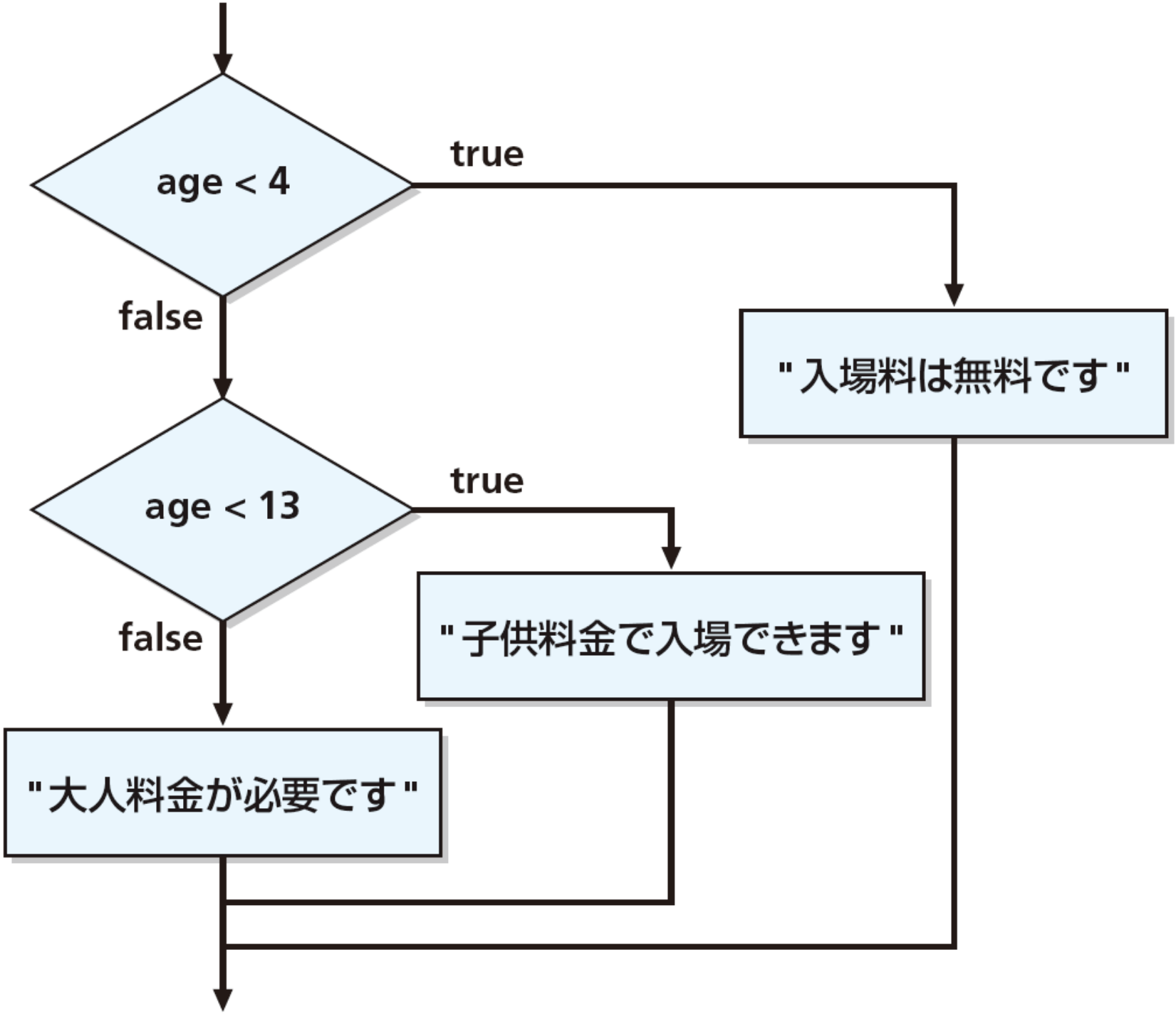
if～else文の使用例

```
int age;
age = 20;
if(age < 20) {
  System.out.println("未成年ですね");
} else {
  System.out.println("投票に行きましょう");
}
```

複数の if ～ else 文

if～else文を連結して、条件に応じた複数の分岐を行える

```
int age;
age = 20;
if(age < 4) {
  System.out.println("入場料は無料です");
} else if(age < 13) {
  System.out.println("子供料金で入場できます");
} else {
  System.out.println("大人料金が必要です");
}
```


age < 4
true
false
" 入場料は無料です "
age < 13
true
false
" 子供料金で入場できます "
" 大人料金が必要です "

演習

日常生活のなかで、条件に応じて処理が変化するものを探し、それをif～else文で表現してみよう。日本語を使ってかまいません。複雑なものにもチャレンジしてみよう。

例：

```
if(お腹の状態 == 空腹) {
   if(ダイエット中である == true) {
      低カロリーのものを食べる
   } else {
      好きなお菓子を食べる
   }
} else {
   勉強を続ける
}
```

演習

aの値が 3, 5, 8, 9, 10, 15, 20 のときに、何が出力されるか予測し確認しよう

```
if(a < 5) {
  System.out.println("A");
} else if(a < 9) {
  System.out.println("B");
} else if(a < 15) {
  System.out.println("C");
} else {
  System.out.println("D");
}
```

if文の後の{}の省略

if文の後の命令文が1つなら、{}を省略できる

次の2つは同じ

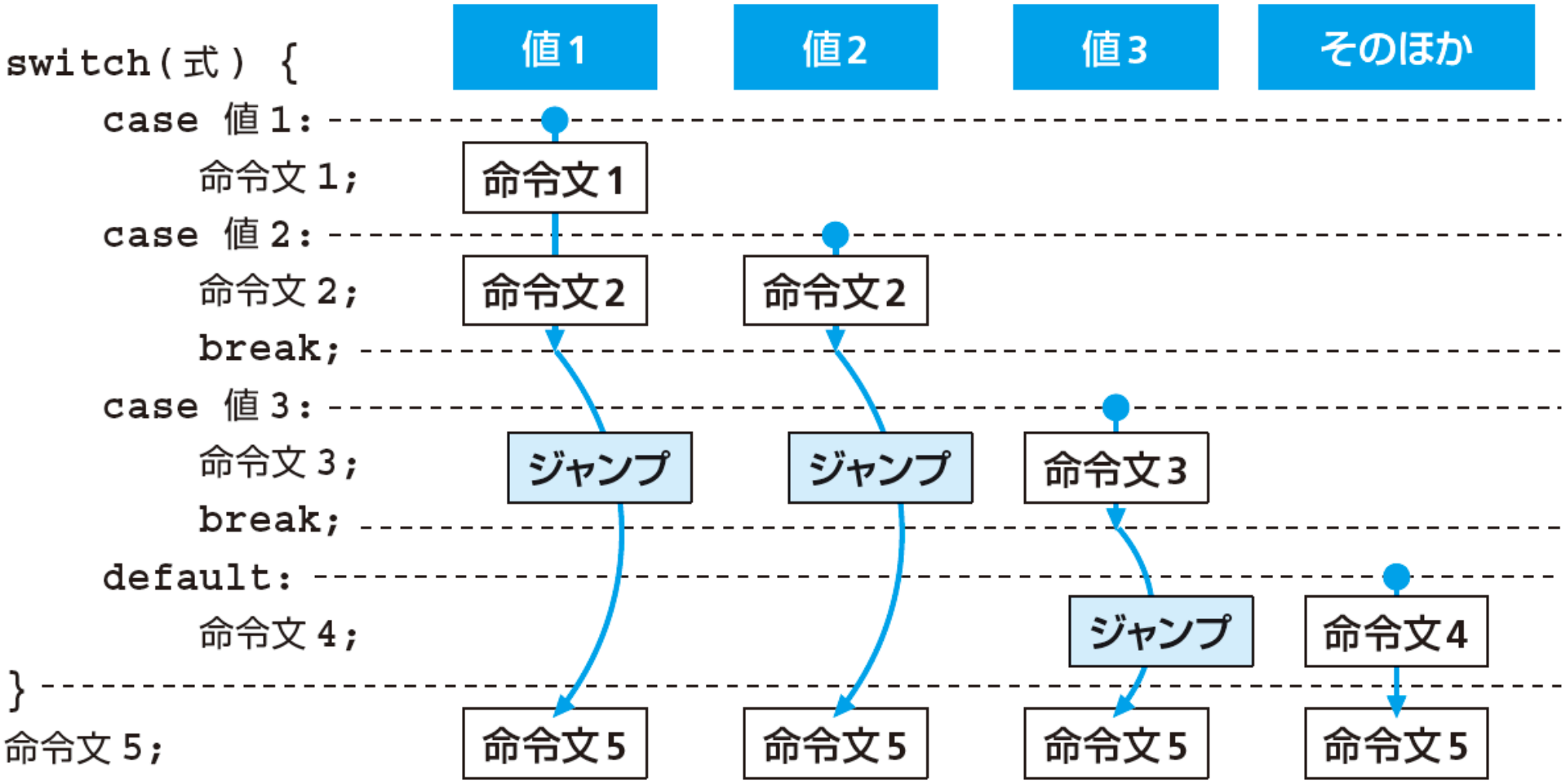
```
if(age >= 20)
    System.out.println("二十歳以上ですね");
```

```
if(age >= 20) {
    System.out.println("二十歳以上ですね");
}
```

ただし{}を省略する時は注意が必要。

```
if(age >= 20)
    System.out.println("二十歳以上ですね");
    System.out.println("お酒を飲めますね");
```

switch文



式の値によって処理を切り替える。break文でブロックを抜ける。

switch文の例(1)

```
switch (score) {
case 1:
        System.out.println("もっと頑張りましょう");
        break;
case 2:
        System.out.println("もう少し頑張りましょう");
        break;
case 3:
        System.out.println("普通です");
        break;
case 4:
        System.out.println("よくできました");
        break;
case 5:
        System.out.println("大変よくできました");
        break;
default:
        System.out.println("想定されていない点数です");
}
System.out.println("switchブロックを抜けました");
```

switch文の例(2)

```
switch (score) {
case 1:
case 2:
	System.out.println("もっと頑張りましょう");
	break;
case 3:
case 4:
case 5:
	System.out.println("合格です");
	break;
default:
	System.out.println("想定されていない点数です");
}
```

演習

次のswitch文では、変数iの値が1,2,3,4,5のとき、それぞれどのような結果が得られるか予測し確認しよう

```
switch(i) {
case 1:
  System.out.println("A");
case 2:
  break;
case 3:
  System.out.println("B");
case 4:
default:
  System.out.println("C");
}
```

ワン・モア・ステップ 3項演算子

```
int c;
if(a > b) {
    c = a;
} else {
    c = b;
}
```

```
int c = (a > b) ? a : b;
```

構文： 条件式？値1：値2

条件式がtrueの場合に、式の値が値1に、falseの場合には値2になる

演習

日常の生活を見まわして、条件に応じて値が変化するものを探してみよう。それを3項演算子を使って表現してみよう。日本語を使って構いません。

例：

```
夕ご飯 = 所持金 > 1000円 ? 外食 : 自炊;
```

演習

cの値が何になるか推測し確認しよう

(a)

```
int a = 5;
int b = 3;
int c = (a > b) ? a : b;
```

(b)

```
int a = 5;
int b = 3;
int c = (a > b * 2) ? a + 1 : b - 3;
```

(c)

```
int a = -5;
int c = (a > 0) ? a : -a;
```

論理演算子

論理演算子を使って複数の条件式を組み合わせられる

演算子	演算の名前	式がtrueになる条件	使用例
&&	論理積	左辺と右辺の 両方がtrueのとき	a > 0 && b < 0 (変数aが0より大きく、かつ bが0より小さい場合にtrue)
||	論理和	少なくとも 左辺と右辺のどちらかが trueのとき	a > 0 || b < 0 (変数aが0より大きい、または 変数bが0より小さい場合にtrue)
^	排他的論理和	左辺と右辺の どちらかがtrueで 他方がfalseのとき	a > 0 ^ b < 0 (変数aが0より大きく、かつ bが0より小さくない場合にtrue。 またはaが0より大きくなく、かつ bが0より小さい場合にtrue)
!	否定	右辺がfalseのとき (左辺はなし)	!(a > 0) (変数aが0より大きくない場合に true)

論理演算子の例

ageが13以上 かつ ageが65未満

```
age >= 13 && age < 65
```

ageが13未満 または age が65以上

```
age < 13 || age >= 65
```

ageが13以上 かつ age が65未満 かつ 20でない

```
age >= 13 && age < 65 && age !=20
```

演算子の優先度

算術演算子が関係演算子より優先される

```
a + 10 > b * 5   ＝   (a + 10) > (b * 5)
```

関係演算子が論理演算子より優先される

```
a > 10 && b < 3   ＝   (a > 10) && (b < 3)
```

カッコの付け方で論理演算の結果が異なる

```
x && ( y || z )   ≠   (x && y ) || z
```

演習

日常生活のなかで、複数の条件の組み合わせに応じて処理が変化するものを探してみよう。それを論理演算子を使って表現してみよう。日本語を使って構いません。
複雑なものにもチャレンジしてみよう。

例：

```
if(曜日 == 日曜 && 天気 != 雨) {
    買い物にでかける
}
```

演習

変数a,b,cに関する次の文章を論理演算子を使った条件式で表そう

(a) aはbより大きい、かつ、bはcより大きい
(b) aはbより小さい、または、aはcより小さい
(c) a,b,cの中で、cが一番大きい
(d) c>b>aの大小関係がある
(e) aはcと等しいが、aはbと等しくない
(f) aはbの2倍より大きく、aはbの3倍よりは大きくない

処理の繰り返し

- ある処理を繰り返し実行したいことがよくある
- ループ構文を使用すると、繰り返し処理を簡単に記述できる
- Javaには3つのループ構文がある

  for文

  while文

  do ～ while文

for文

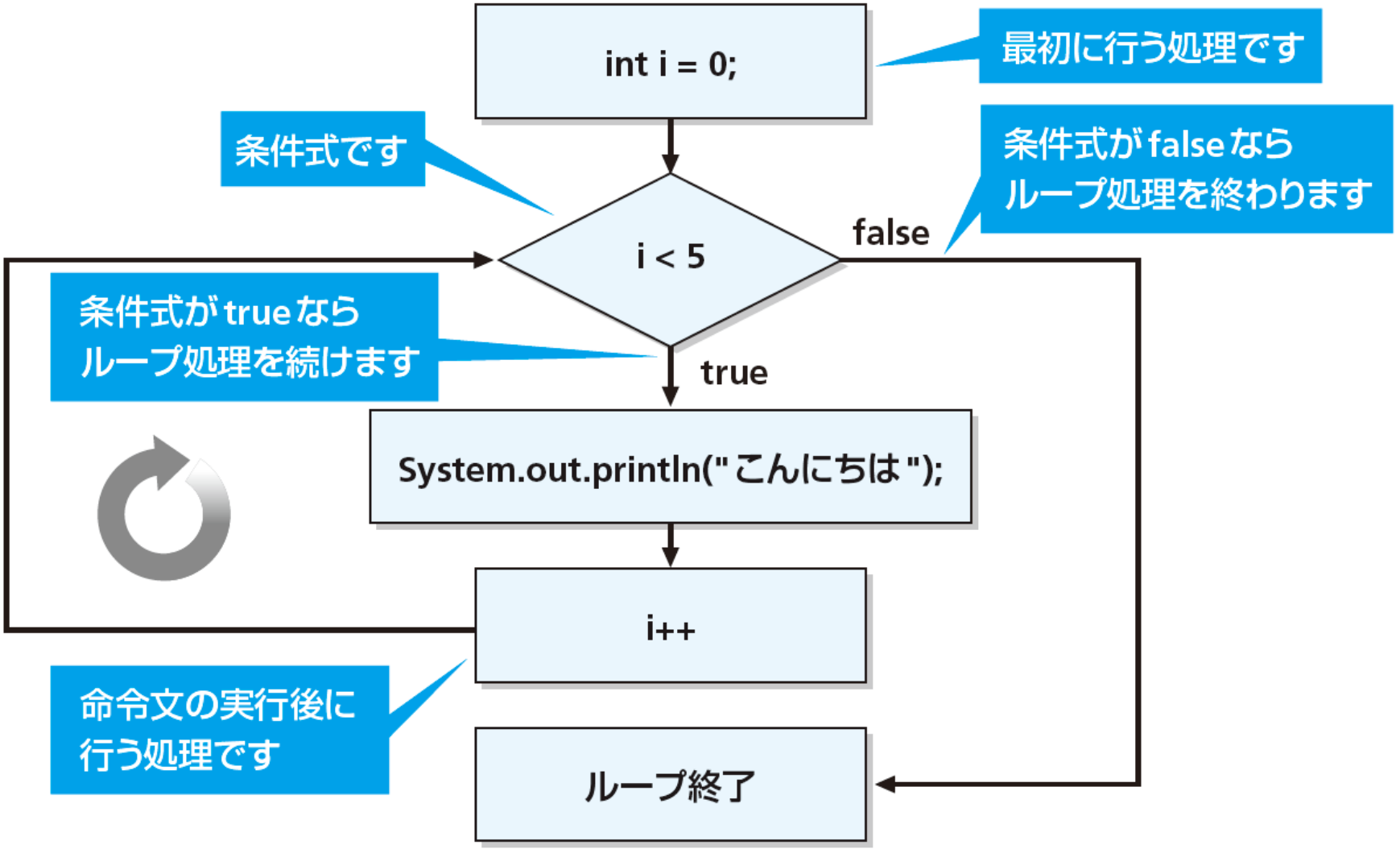
for文の構文

```
for(最初の処理; 条件式; 命令文の後に行う処理){
    命令文
}
```

1. 「最初の処理」を行う
2. 「条件式」がtrueなら「命令文」を行う
   false ならfor文を終了する
3. 「命令文の後に行う処理」を行う
4. 2.に戻る

for文の例

```
for(int i = 0; i < 5; i++) {
   System.out.println("こんにちは");
}
```



forループ内で変数を使う

for ループ内で変数を使用することで、例えば1から100までを順番に足し合わせる計算ができる

```
int sum = 0;
for(int i = 1; i <= 100; i++) {
  sum += i;
  System.out.println(i + "を加えました");
}
System.out.println("合計は" + sum );
```

演習

次の計算をするプログラムを作ろう

1. 1～100までの偶数だけを順番に足し算する

2. 1～100までの2または3の倍数を順番に足し算する。ただし12の倍数は足し算しない。

3. xの値を-10から10まで1ずつ変化させたときの次の式の値を求める。

$$x^2 - 2x + 1$$

変数のスコープ

- 変数には扱える範囲が決まっている。これを「変数のスコープ」と呼ぶ
- スコープは変数の宣言が行われた場所から、そのブロック{ } の終わりまで

```
class ForExample2 {
  public static void main(String[] args) {
    int sum = 0;
    for(int i = 1; i <= 100; i++) {
      sum += i;
      System.out.println(i + "を加えました");
    }
    System.out.println("合計は" + sum );
  }
}
```

while文

while文の構文

```
while(条件式){
        命令文
}
```

1. 「条件式」がtrueなら「命令文」を行う
   false ならwhile文を終了する
2. 1.に戻る

※ for文と同じ繰り返し命令を書ける

while文の例

```
int i = 0;
while(i < 5) {
   System.out.println("こんにちは");
   i++; // この命令文が無いと「無限ループ」
}
```

```
int i = 5;
while(i > 0) {
   System.out.println(i);
   i--;  // この命令文が無いと「無限ループ」
}
```

do ～ while文

do ～ while文の構文

```
do {
    命令文
} while(条件式);
```

必ず1回は実行される

1. 「命令文」を実行する
2. 「条件式」がtrueなら1.に戻る。
   false ならdo～while文を終了する

※ for文、while文と同じ繰り返し命令を書ける

do ～ while文の例

```
int i = 0;
do {
  System.out.println("こんにちは");
  i++;
} while(i < 5);
```

```
int i = 5;
do {
  System.out.println(i);
  i--;
} while(i > 0);
```

ループの処理を中断する「break」

break; でループ処理を強制終了できる

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum += i;
  System.out.println(i + "を加えました");
  if(sum > 20) {
    System.out.println("合計が20を超えた");
    break;
  }
}
System.out.println("合計は" + sum );
```

ループ内の処理をスキップする「continue」

continue; でブロック内の残りの命令文をスキップできる

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  if(i % 2 == 0) {
    continue;
  }
  sum += i;
  System.out.println(i + "を加えました");
}
System.out.println("合計は" + sum );
```

ループ処理のネスト

ループ処理の中にループ処理を入れられる

```
for(int a = 1; a <= 3; a++) {
  System.out.println("a = "+ a); //★
  for(int b = 1; b <= 3; b++) {
    System.out.println("b = "+ b); //☆
  }
}
```

★の命令文は3回実行される

☆の命令文は9回実行される

演習

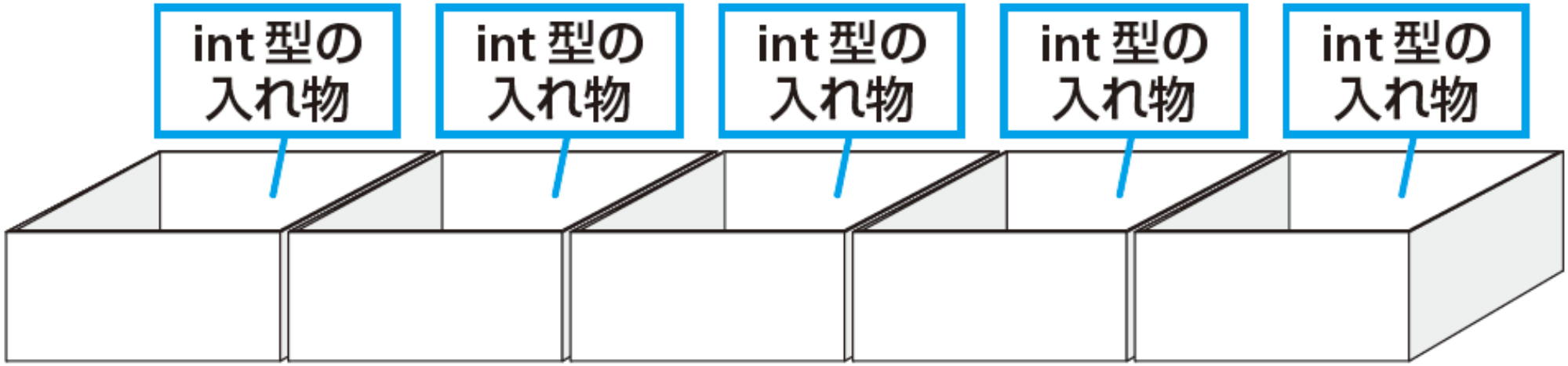
次のような九九表を出力するプログラムを作ってみよう

1×1=1 1×2=2 1×3=3 1×4=4 1×5=5 1×6=6 1×7=7 1×8=8 1×9=9
2×1=2 2×2=4 2×3=6 2×4=8 2×5=10 2×6=12 2×7=14 2×8=16 2×9=18
3×1=3 3×2=6 3×3=9 3×4=12 3×5=15 3×6=18 3×7=21 3×8=24 3×9=27
4×1=4 4×2=8 4×3=12 4×4=16 4×5=20 4×6=24 4×7=28 4×8=32 4×9=36
5×1=5 5×2=10 5×3=15 5×4=20 5×5=25 5×6=30 5×7=35 5×8=40 5×9=45
6×1=6 6×2=12 6×3=18 6×4=24 6×5=30 6×6=36 6×7=42 6×8=48 6×9=54
7×1=7 7×2=14 7×3=21 7×4=28 7×5=35 7×6=42 7×7=49 7×8=56 7×9=63
8×1=8 8×2=16 8×3=24 8×4=32 8×5=40 8×6=48 8×7=56 8×8=64 8×9=72
9×1=9 9×2=18 9×3=27 9×4=36 9×5=45 9×6=54 9×7=63 9×8=72 9×9=81

```
class Example {
  public static void main(String args[]) {
    for(int i = 1; i <= 9; i++) {
      for(int j = 1; j <= 9; j++) {
        命令文
      }
      System.out.println(); // 改行
    }
  }
}
```

配列

- 複数の値の入れ物が並んだもの
  （1次元配列とも呼ぶ）
- 複数の値をまとめて扱うときに便利

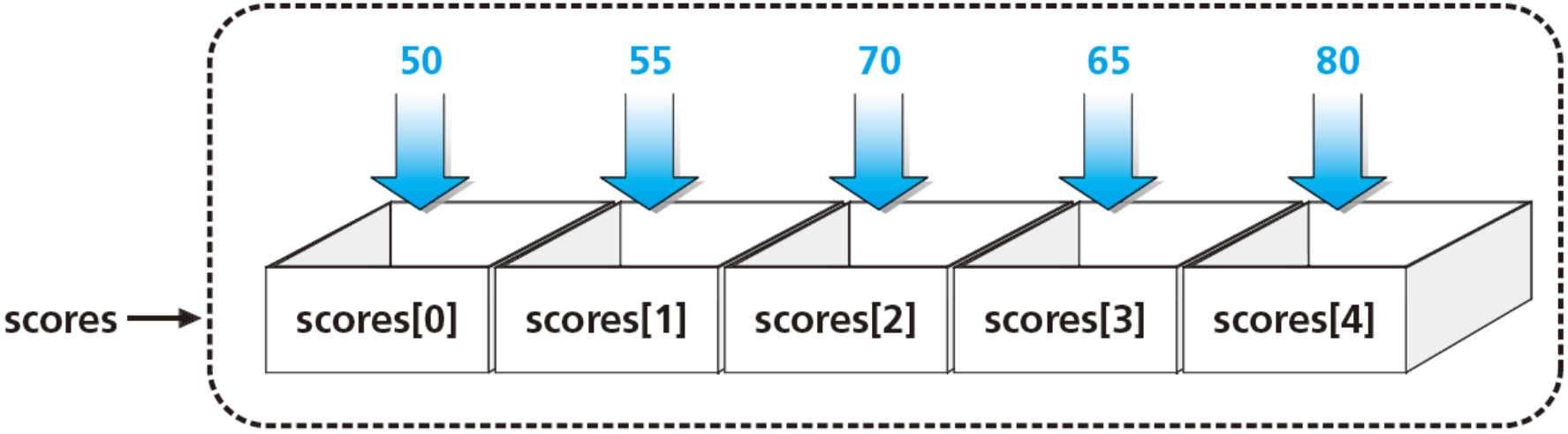


配列の使い方

1. 配列を表す変数を宣言する
   ```
   int[] scores;
   ```
2. 配列の要素（入れ物）を確保する
   ```
   scores = new int[5];
   ```
3. 配列に値を入れる
   ```
   scores[0] = 50;
          ⋮
   scores[4] = 80;
   ```
   []の中の数字はインデックス
   0～(要素の数-1)を指定する
4. 配列に入っている値を参照する。
   例：`System.out.println(scores[2]);`

配列の使用

```
int[] scores;
scores = new int[5];
scores[0] = 50;
scores[1] = 55;
scores[2] = 70;
scores[3] = 65;
scores[4] = 80;

for(int i = 0; i < 5; i++) {
      System.out.println(scores[i]);
}
```



配列の使用

配列は次のようにしても初期化できる

```
int[] scores = {50, 55, 70, 65, 80};
```

配列の大きさ（要素の数）は次のようにして確認できる

```
int n = scores.length;
```

演習

テストの点数の分布に基づいて、右図のような出力を行うプログラムを作ろう

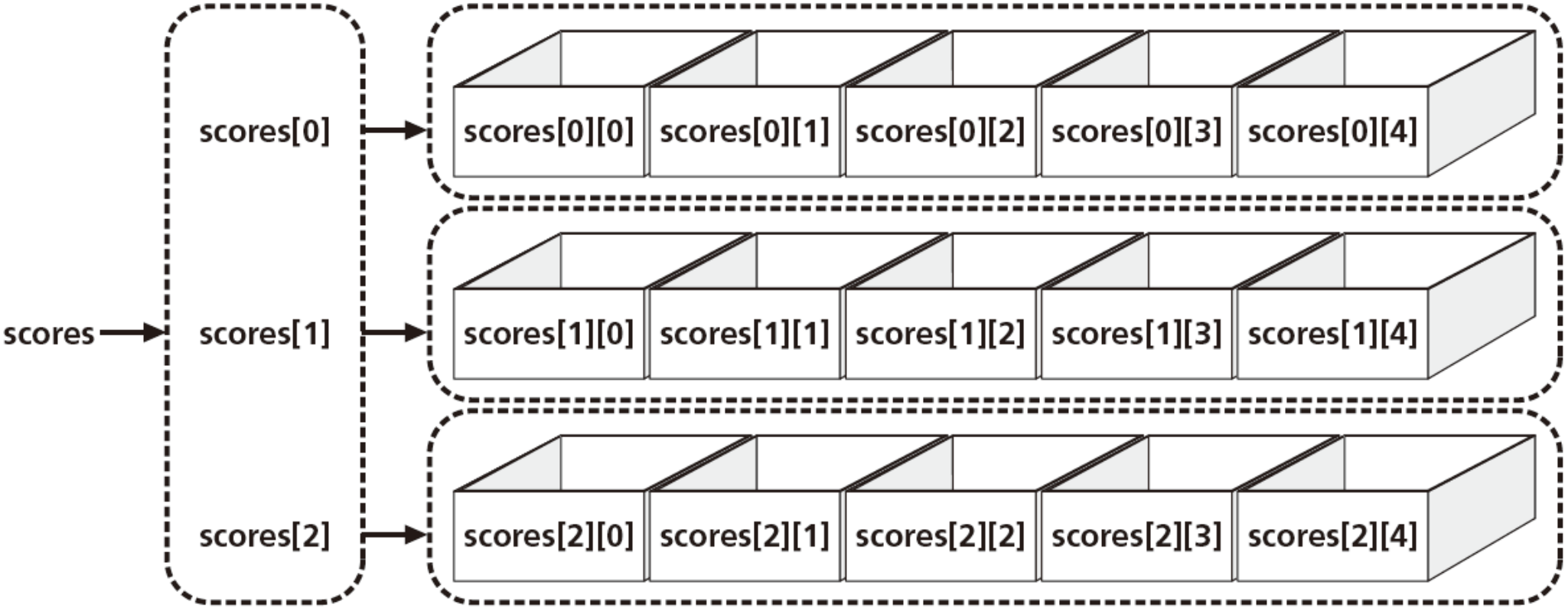
点数と人数の関係
0点：1人、1点：3人、2点：5人、
3点：6人、4点：5人、5点：2人

```
0:*
1:***
2:*****
3:******
4:*****
5:**
```

```
class Example {
  public static void main(String args[]) {
    int[] counts = {1, 3, 5, 6, 5, 2};

    命令文

  }
}
```

多次元配列（配列の配列）

```
int[][] scores = new int[3][5];
scores[0][0] = 50;
scores[2][3] = 65;
```



第4章
メソッド（クラスメソッド）

メソッドとは

- 長いプログラムが必要になるときは、命令文を分けて管理した方が見通しがよくなる
- メソッドは複数の命令文をまとめたもの

メソッドの宣言のしかた

```
void メソッド名() {
    命令文
}
```

メソッドの例

メソッドを持つクラスの例

countdownという名前のメソッド宣言

```
public class Example {
  public static void countdown(){
    System.out.println(“カウントダウンをします”);
    for(int i = 5; i >= 0; i--){
      System.out.println(i);
    }
  }

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

countdownという名前のメソッドを呼び出す

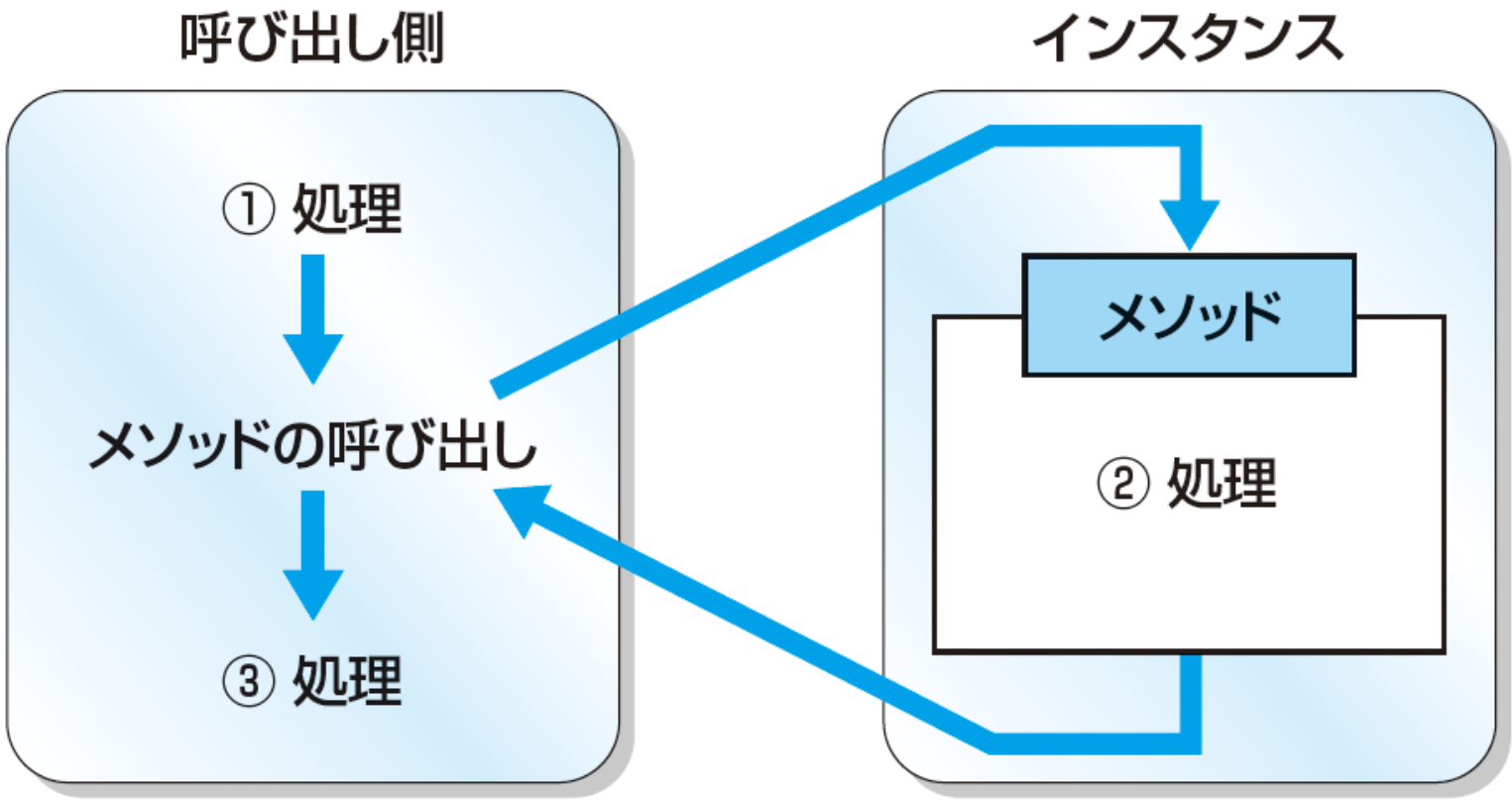
mainメソッド

public static void main(String[] args)

- Javaでは、プログラムが実行されるときに、このmainメソッドがJava仮想マシンから呼び出される

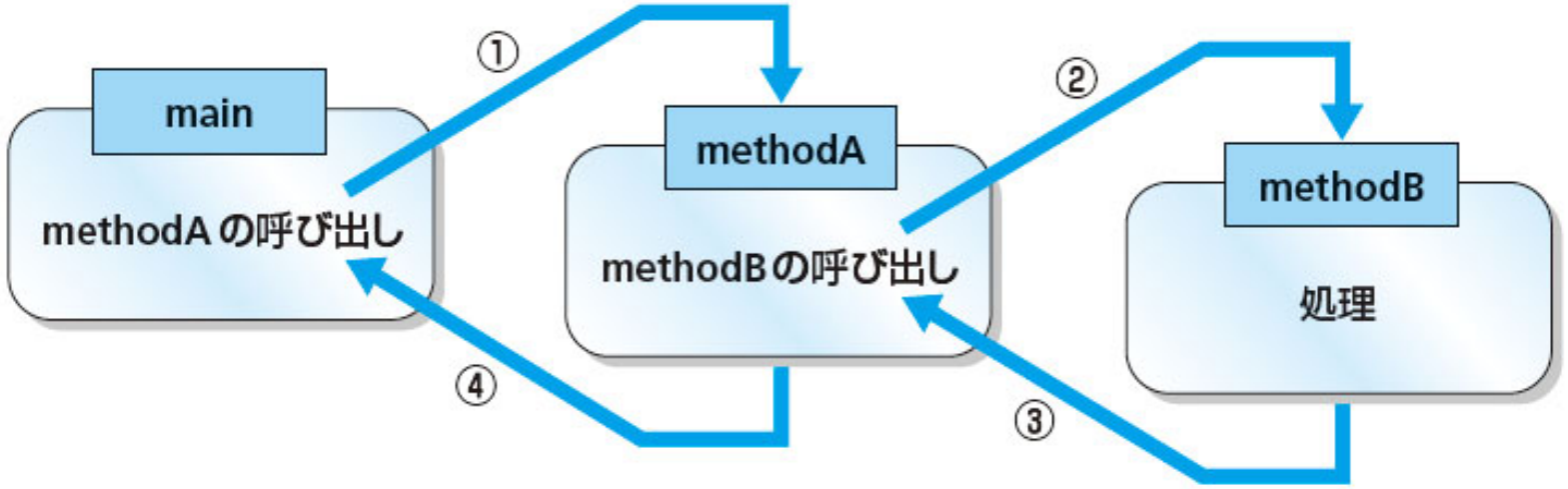
- mainメソッドは、プログラムの開始位置となる特別なメソッド

メソッド呼び出しの処理の流れ



メソッド呼び出しの階層

```
public class Example {
  public static void methodA(){
    methodB();
  }

  public static void methodB(){
  }

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



メソッドの引数と戻り値

メソッドは命令文のセット

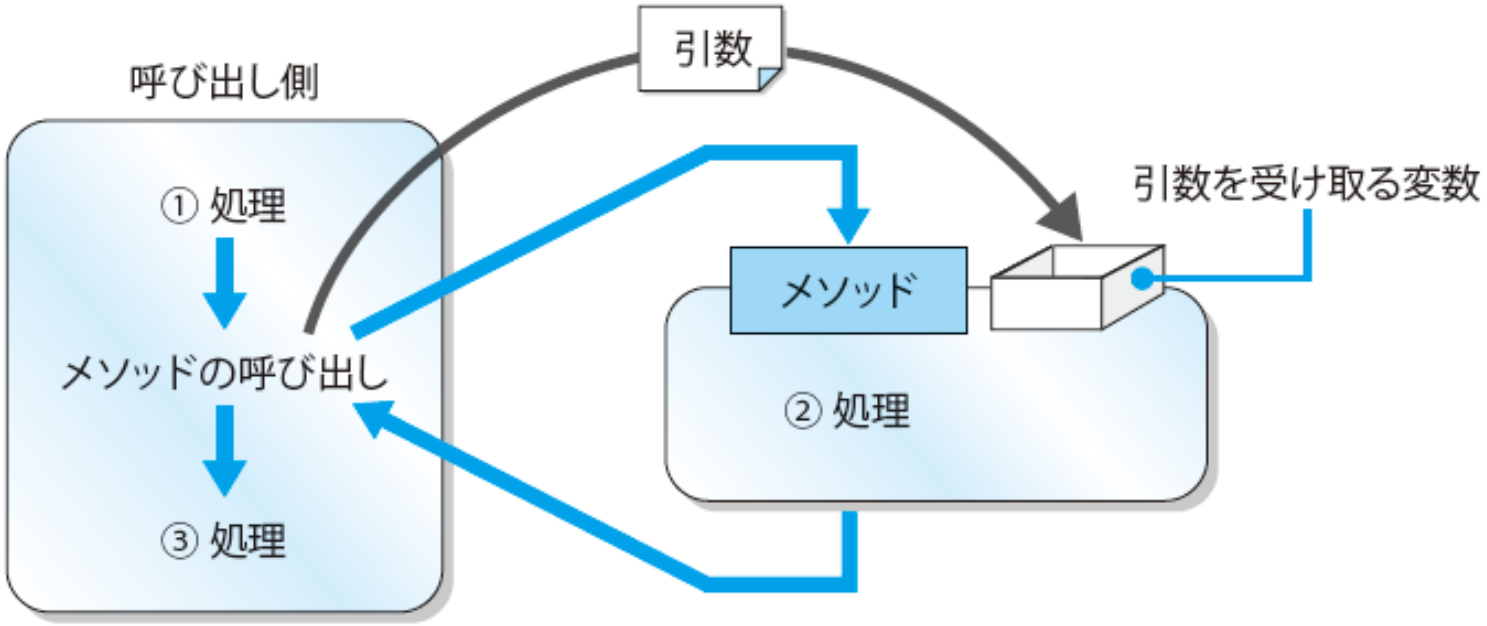
- 引数

  メソッドには、命令を実行するときに値を渡すことができる。この値を「引数」と呼ぶ。

- 戻り値

  メソッドは、命令を実行した結果の値を呼び出し元に戻すことができる。この値を「戻り値」と呼ぶ。

引数のあるメソッド



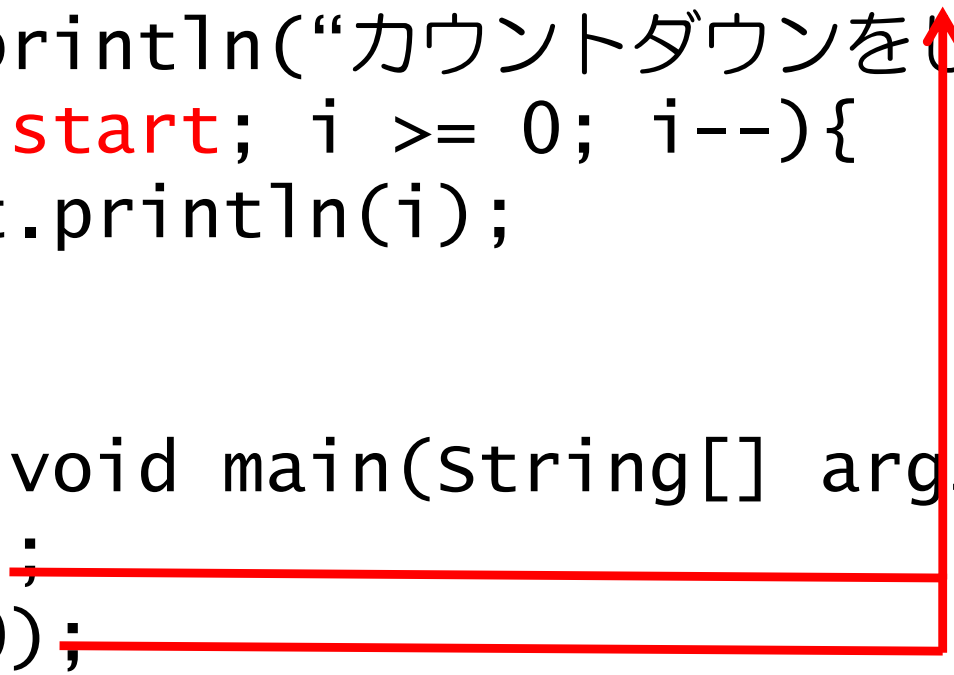

```
void メソッド名(型 変数名) {
        命令文
}
```

引数のあるメソッドの例

引数の受け渡しには、メソッド名の後ろのカッコ()を使用する。

startという名前のint型の変数で値を受け取る

```
class Example {
  public static void countdown(int start){
    System.out.println("カウントダウンをします");
    for(int i = start; i >= 0; i--){
      System.out.println(i);
    }
  }
  public static void main(String[] args) {
    countdown(3);
    countdown(10);
  }
}
```



複数の引数のあるメソッドの例

複数の引数を指定できる

```
class Example {
  public static void countdown(int start, int end){
    System.out.println(“カウントダウンをします”);
    for(int i = start; i >= end; i--){
      System.out.println(i);
    }
  }
  public static void main(String[] args) {
    countdown(7, 3);
  }
}
```

mainメソッドの引数

プログラムの実行時に引数を指定できる

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

※引数は文字列として渡されるため、数値として扱いたい場合は型変換が必要

実行結果

```
> java Example hello 123
  hello
  123
```

Eclipseによる引数指定

ワン・モア・ステップ（キーボード入力）

java.util.Scannerを使う

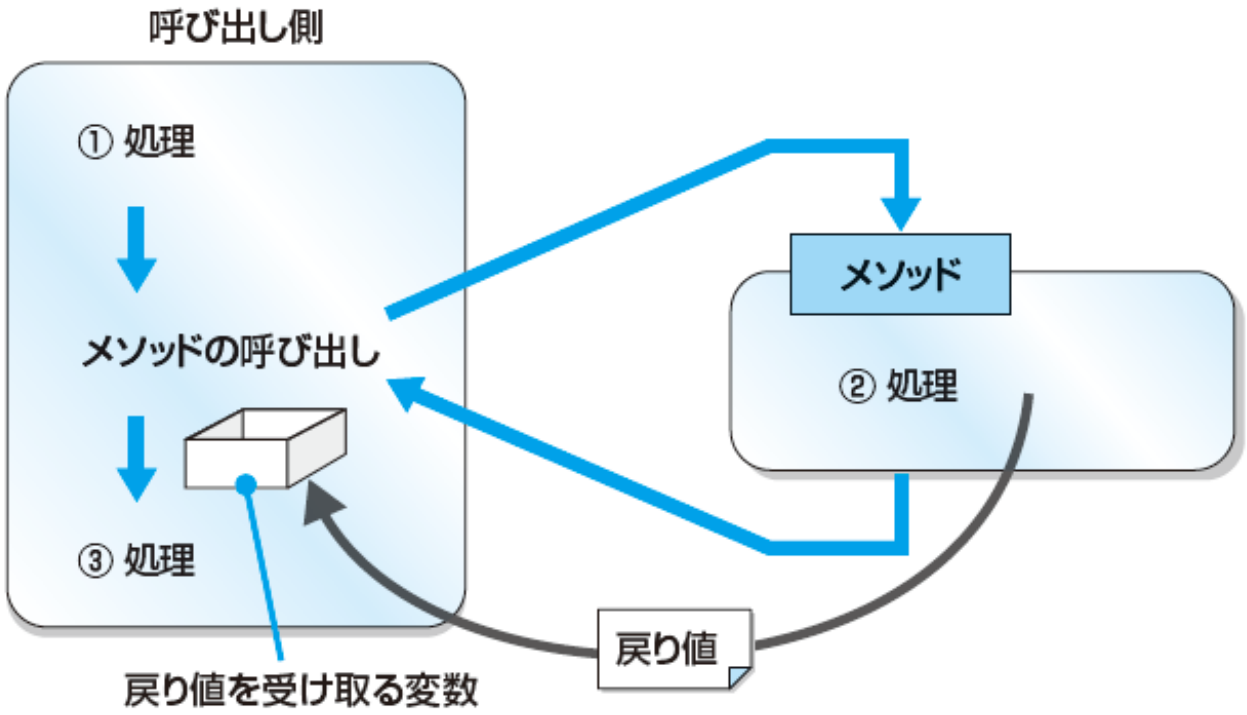
```
import java.util.Scanner;

class Example {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.println(“整数を入力してください。”);
    int i = in.nextInt();
    System.out.println(i + “が入力されました。”);
  }
}
```

実行結果

```
整数を入力してください。
5　←Enterを押して確定
5が入力されました。
```

※小数を受け取る場合は
nextIntの代わりにnextDouble、
文字列を受け取る場合はnextを使う。

戻り値のあるメソッド




```
戻り値の型 メソッド名(引数列) {
        命令文
        return 戻り値;
}
```

戻り値のあるメソッドの例１

- return を使って値を戻すようにする
- 戻り値は1つだけ
- 戻り値の型をメソッド名の前に記す

```
class Example {
 public static double getAreaOfCircle(double radius) {
  return radius * radius * 3.14;
 }

 public static void main(String[] args) {
  double circleArea = getAreaOfCircle(2.5);
  System.out.println("半径2.5の円の面積は" + circleArea);
 }
}
```

戻り値のあるメソッドの例2

```
class Example {
  public static boolean isPositiveNumber(int i){
    if(i > 0) {
      return true;
    } else {
      return false;
    }
  }
  public static void main(String[] args) {
    int i = -10;
    if(isPositiveNumber(i) == true){
      System.out.println(“iの値は正です”);
    } else {
      System.out.println(“iの値は負またはゼロです”);
    }
  }
}
```

メソッドのまとめ

引数なし、戻り値なし

```
void メソッド名() {
        命令文
}
```

引数あり、戻り値なし

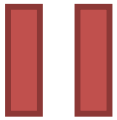
```
void メソッド名(型 変数名) {
        命令文
}
```

引数あり、戻り値あり

```
戻り値の型 メソッド名(型 変数名) {
        命令文
        return 戻り値;
}
```

ワン・モア・ステップ 論理演算式の値

```
if(i > 0) {
  return true;
} else {
  return false;
}
```



```
return (i > 0);
```

```
if(isPositiveNumber(i) == true) { 命令文 }
```

```
if(isPositiveNumber(i)){ 命令文 }
```

オーバーロード

- オーバーロードとは
  同じ名前のメソッドを複数宣言すること
  （ただし、引数は異なる必要がある）

- 同じ名前でも大丈夫？
  呼び出し時に指定される引数のタイプによって実行されるメソッドまたはコンストラクタが区別される

メソッドのオーバーロードの例

```
class Example {
  public static void methodA() {
    System.out.println("引数はありません");
  }
  public static void methodA(int i) {
    System.out.println("int型の値" + i + "を受け取りました");
  }
  public static void methodA(double d) {
    System.out.println("double型の値" + d + "を受け取りました");
  }
  public static void methodA(String s) {
    System.out.println("文字列" + s + "を受け取りました");
  }
  public static void main(String[] args) {
    methodA();
    methodA(1);
    methodA(0.1);
    methodA("Hello");
  }
}
```

オーバーロードができない場合

変数の名前が異なるだけではオーバーロードできない

✕
```
public static void methodA(int i) {略}
public static void methodA(int j) {略}
```

戻り値の型が異なるだけではオーバーロードできない

✕
```
public static void methodA(int i) {略}
public static int methodA(int i) {略}
```

シグネチャ

- 「メソッド名」「引数の型」「引数の数」の3つの要素をシグネチャと呼ぶ

- シグネチャが同じメソッドを宣言することはできない

第5章 クラスの基本

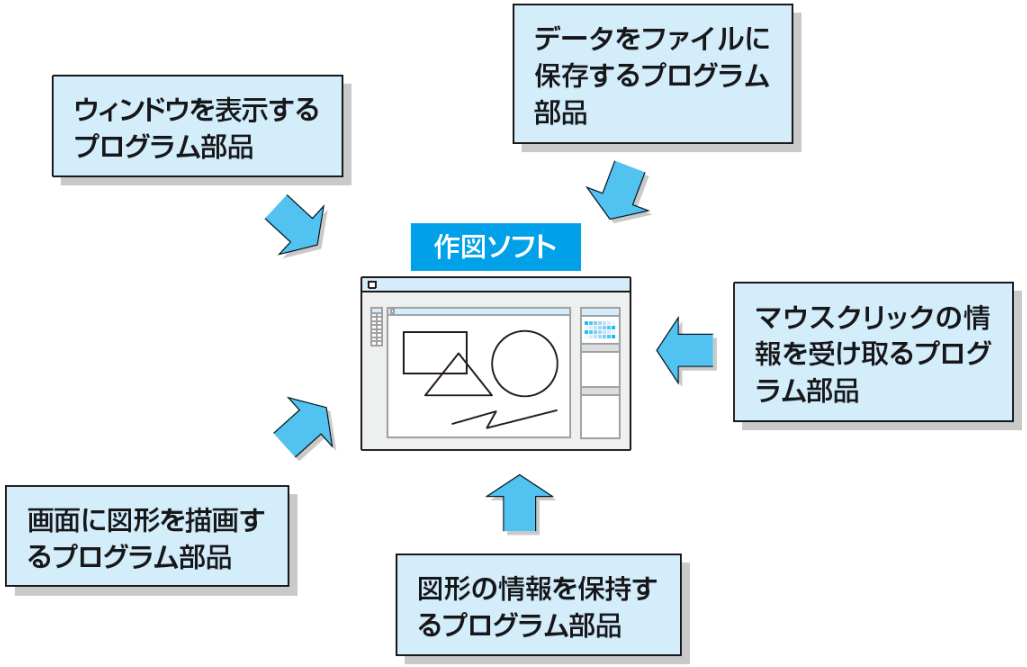
オブジェクト指向とは

- プログラム部品を組み合わせることでプログラム全体を作成する

- プログラムを自動車に例えると・・
  - 自動車は様々な部品から構成される
    車体・エンジン・タイヤ・ヘッドライト
  - 最終製品は部品の組み合わせ
  - それぞれの部品の内部構造を知らなくても、組み合わせ方（使い方）がわかればよい
  - 部品単位でアップデートできる

オブジェクト指向とクラス

- プログラムの部品＝オブジェクト と考える
- オブジェクトがどのようなものか記述したものが「クラス（class）」
- Javaによるプログラミング
  ＝classを定義すること



複雑なプログラムは多くの
プログラム部品から構成される

クラスとインスタンス

- クラス

  オブジェクトに共通する属性（情報・機能）を抽象化したもの
- インスタンス

  具体的な個々のオブジェクト

アプリの例	クラス	インスタンス
学生情報管理ツール	どのような項目を管理するか定めたもの	学生Aの情報、学生Bの情報、学生Cの情報、・・・
RPGゲーム	モンスターが持つ情報や動作を定めたもの	モンスターA、モンスターB、モンスターC、・・・
サッカーゲーム	サッカー選手が持つ情報や動作を定めたもの	選手１，選手２，選手３、・・・

簡単なクラスの宣言とインスタンスの生成

例として学籍番号(id)と氏名(name)を持つ学生証を扱うためのクラスは、次のように宣言する

```
class StudentCard {
  int id;
  String name;
}
```

フィールド（StudenCardクラスがもつ情報）

- クラスの名前は自由に決められる。今回はStudentCardとした
- idとnameという名前のint型とString型の変数をクラスの中に定義した。これをフィールドと呼ぶ
- idとnameという2つの値をセットにして扱える

演習

日常を見まわして、クラスの定義をしてみよう。日本語を使ってかまいません。

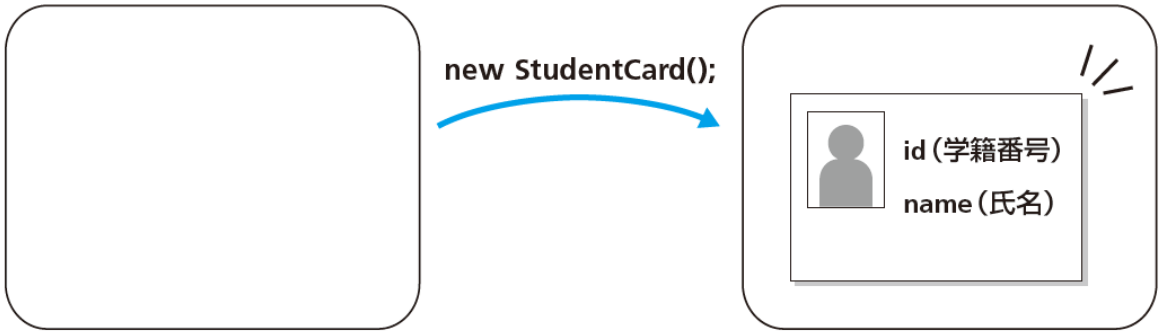
例：

```
class 本 {
  タイトル
  著者
  出版社
}

class 賃貸ルーム {
  広さ
  家賃
  住所
}
```

インスタンスの生成

インスタンスを生成するにはnewを使用する。

```
new StudentCard();
```



生成したインスタンスを変数aに代入できる。

```
StudentCard a = new StudentCard();
```

クラス名

インスタンスが持つ変数に値を代入できる

```
StudentCard a = new StudentCard();
a.id = 1234;
a.name = "鈴木太郎";
```

インスタンス変数へのアクセス

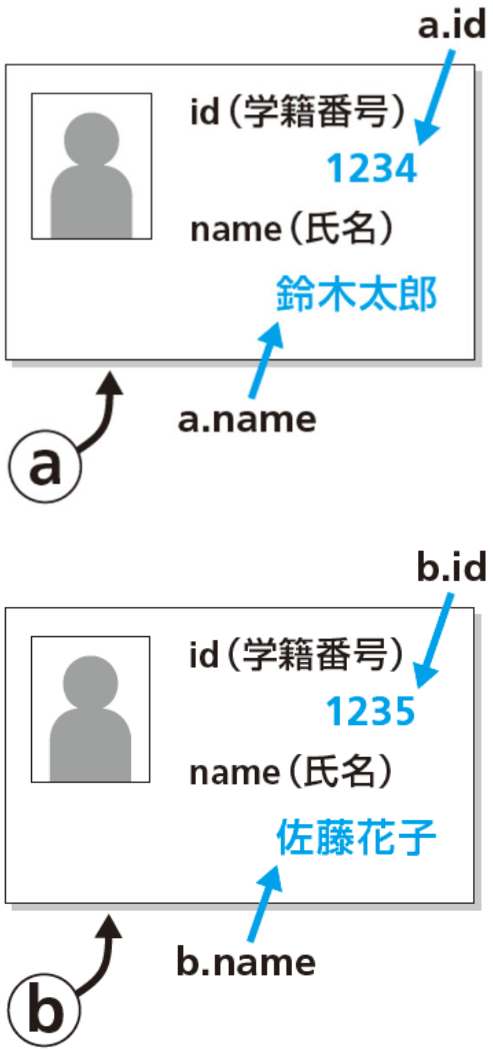
- インスタンスが持つ変数（クラスのフィールドに定義された変数）のことを「インスタンス変数」と呼ぶ
- StudentCardクラスでは変数idとnameがインスタンス変数
- インスタンス変数にアクセスするには
  「インスタンスを代入した変数名＋ドット＋インスタンス変数名」
  とする

例：
```
StudentCard a = new StudentCard();
a.id = 10;
System.out.println("aのidは" + a.id);
```

（ドットを「の」に置き換えて「aのx」と読むとわかりやすい）

StudentCardクラスの使用例

```
class StudentCard {
  int id; // 学籍番号
  String name; // 氏名
}
public class Example {
  public static void main(String[] args) {
    StudentCard a = new StudentCard();
    a.id = 1234;
    a.name = "鈴木太郎";

    StudentCard b = new StudentCard();
    b.id = 1235;
    b.name = "佐藤花子";

    System.out.println("aのidは" + a.id);
    System.out.println("aのnameは" + a.name);
    System.out.println("bのidは" + b.id);
    System.out.println("bのnameは" + b.name);
  }
}
```

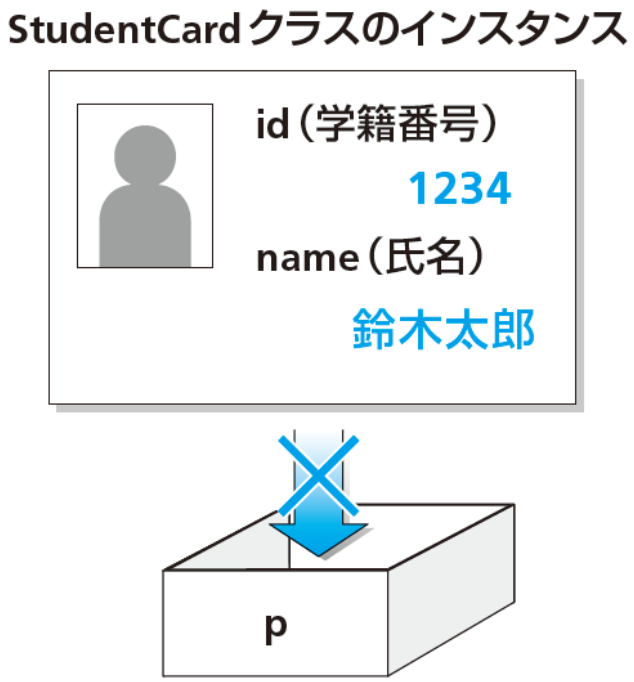


参照型

- Javaで使用できる変数の型

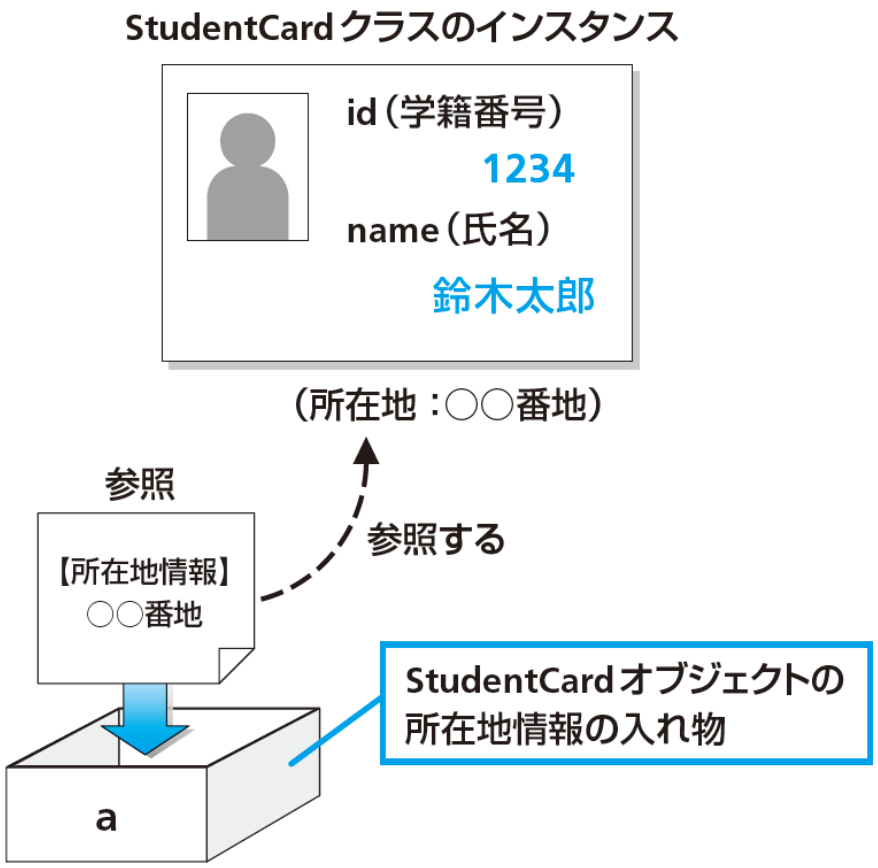
  基本型（int, double, boolean など）

  参照型（インスタンスへの参照）
- 変数にインスタンスそのものは代入されない。



参照の代入

`StudentCard a = new StudentCard();`
としたとき、変数aにはStudentCardクラスのインスタンスの参照が代入される



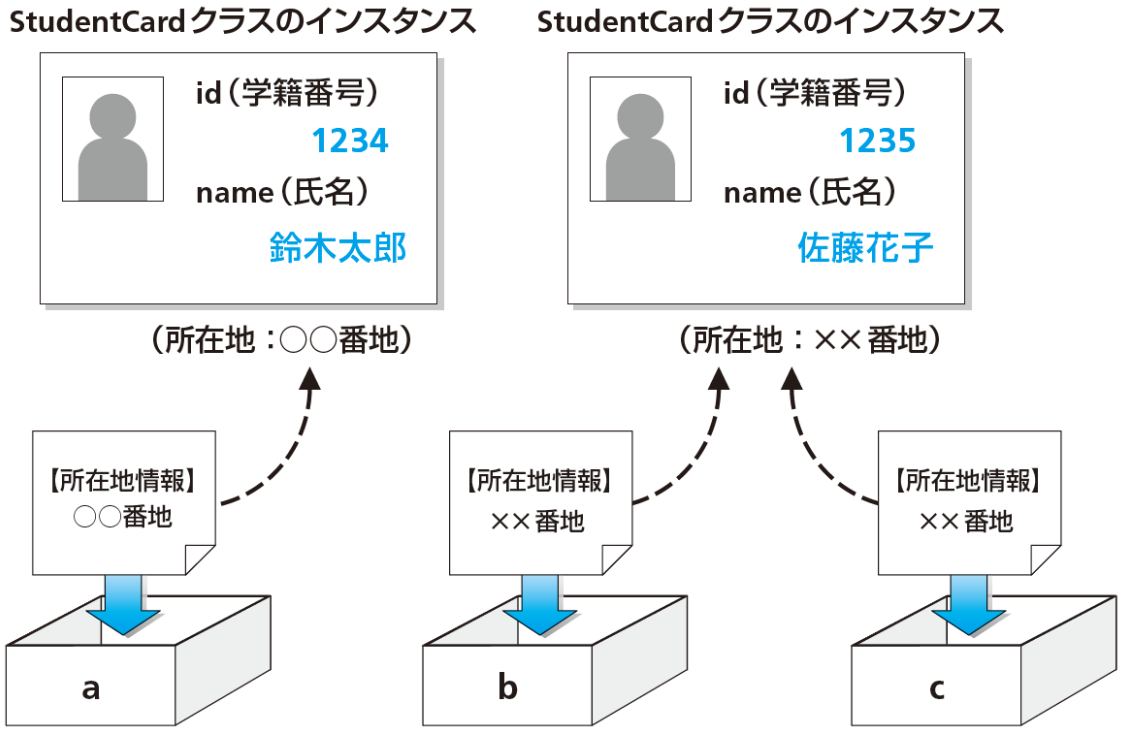
演習

次のプログラムコードの意味を考えてみよう

```
class Dog {
  String name;
}

public class Example {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.name = "Taro";
    Dog dog2 = new Dog();
    dog2.name = "Pochi";
    Dog dog3 = dog2;
    System.out.println(dog3.name);
    dog3.name = "Jiro";
    System.out.println(dog2.name);
  }
}
```

参照の例

```
StudentCard a = new StudentCard();
StudentCard b = new StudentCard();
StudentCard c = b;
a.id = 1234;
a.name = "鈴木太郎";
b.id = 1235;
b.name = "佐藤花子";
```

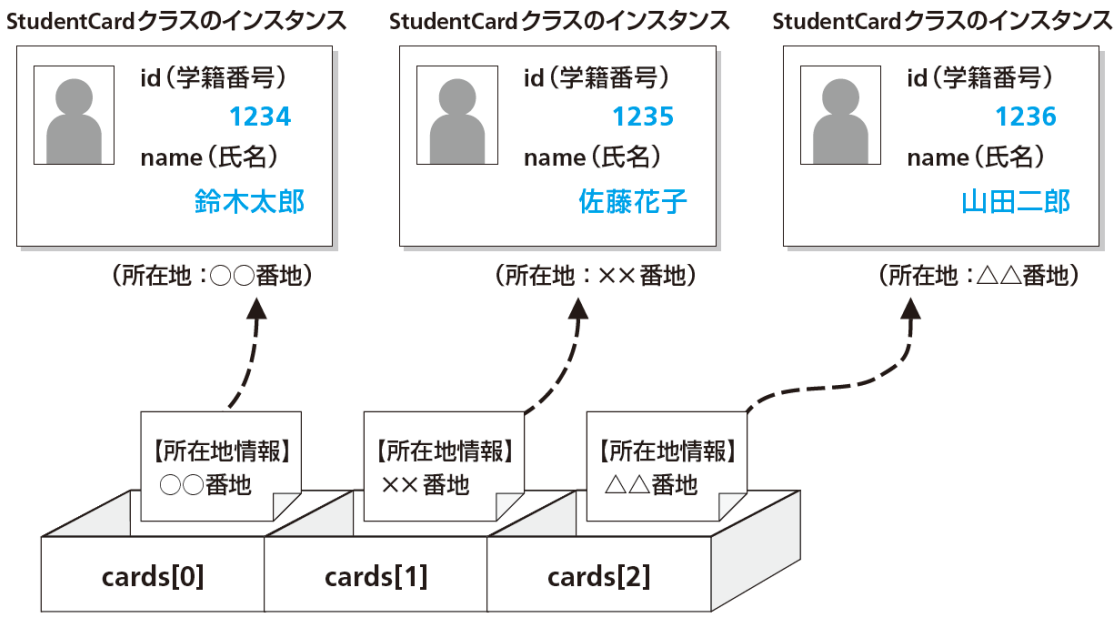


参照の配列

基本型の配列と同じように、参照の配列も作成できる

```
StudentCard[] cards = new StudentCard[3];
cards[0] = new StudentCard();
cards[1] = new StudentCard();
cards[2] = new StudentCard();
cards[0].id = 1234;
cards[0].name = "鈴木太郎";
cards[1].id = 1235;
cards[1].name = "佐藤花子";
cards[2].id = 1236;
cards[2].name = "山田二郎";
```

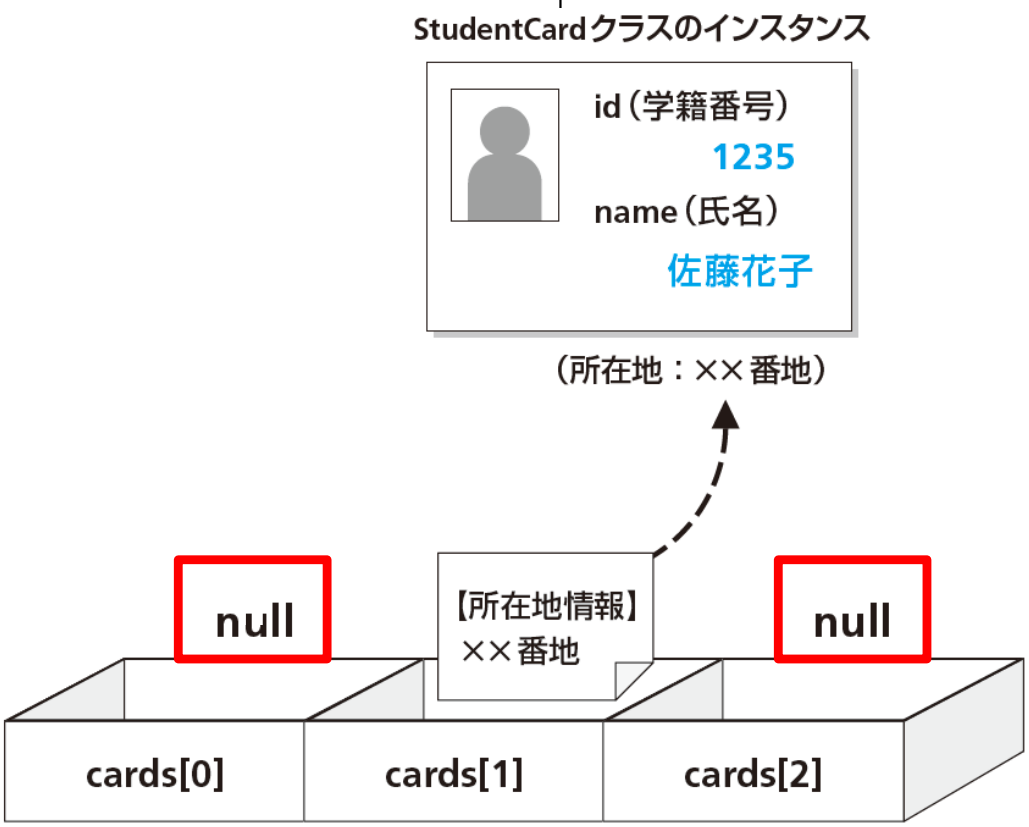
配列を生成



何も参照しないことを表すnull

参照型の変数に、何も参照が入っていない状態をnullという

```
StudentCard[] cards = new StudentCard[3];
cards[1] = new StudentCard();
cards[1].id = 1235;
Cards[1].name = "佐藤花子"
```



nullは参照型の値

nullは、参照型の変数に代入できる

```
StudentCard a;
a = null;
```

nullは参照型の変数の値と比較できる

```
StudentCard a = new StudentCard();
if(a == null) {
  System.out.println("aはnull");
} else {
 System.out.println("aはnullでない");
}
```

参照とメソッド

メソッドには引数としてインスタンスの参照を受け渡しできる

```
static void reset(StudentCard card){
  card.id = 0;
  card.name = "未定";
}
```

メソッドの戻り値にすることもできる

```
static StudentCard compare(StudentCard card0, StudentCard card1){
  if (card0.id < card1.id) {
    return card0;
  } else {
    return card1;
  }
}
```

クラスの定義とファイル

複数のクラス宣言を1つのファイルに記述せず、複数のファイルに分けて記述できる

StudentCard.java

```
public class StudentCard {
  (中略)
}
```

Example.java

```
public class Example {
  public void main (String args) {
    //StudentCardクラスを使った処理を行う
    (中略)
  }
}
```

ワン・モア・ステップ(インスタンス変数の初期値)

インスタンス変数は、インスタンスが生成されるときに自動的に初期化される

```
class DataSet {
  int i;
  double d;
  boolean b;
  String s;
  Dataset data;
}
```

0で初期化される
0.0で初期化される
falseで初期化される
nullで初期化される
nullで初期化される

第6章 クラスの一歩進んだ使い方

コンストラクタ

コンストラクタとは、インスタンスが生成されるときに自動的に実行される特別なメソッド

コンストラクタの構文

```
クラス名(引数列) {
    命令文
}
```

- クラス名と同じ名前のメソッド
- 引数を渡せる（初期化に使用できる）
- 戻り値を定義できない

コンストラクタの例

コンストラクタをもつStudentCardクラス

```
class StudentCard {
  int id;
  String name;

  // コンストラクタ
  StudentCard(int id, String name) {
    this.id = id;
    this.name = name;
  }
}
```

インスタンスの生成

```
StudentCard a = new StudentCard(1234, "鈴木太郎");
System.out.println(a.id);
System.out.println(a.name);
```

自分自身を表す this

インスタンス変数を参照する

```
this.変数名
```

自分のメソッドを実行する

```
this.メソッド名(引数)
```

自分のコンストラクタを実行する

```
this(引数)
```

※ この記述が行えるのはコンストラクタの先頭行だけ

演習

```
class Rectangle {
  double width;  // 幅
  double height; // 高さ
}
```

1. Rectangleクラスに面積を戻り値とする getAreaメソッドを追加しよう
2. 幅と高さを指定できるコンストラクタを追加しよう
3. 引数で渡されたRectangleクラスのインスタンスと比較して、自分の方が面積が大きければtrue、そうでなければfalseを戻り値とするisLargerメソッドを追加しよう

コンストラクタのオーバーロードの例

```
class StudentCard {
  int id;
  String name;
  StudentCard() {
    this.id = 0;
    this.name = "未定";
  }
  StudentCard(String name) {
    this.id = 0;
    this.name = name;
  }
  StudentCard(int id, String name) {
    this.id = id;
    this.name = name;
  }
}
```

インスタンスの生成

```
StudentCard a = new StudentCard();
StudentCard b = new StudentCard("鈴木太郎");
StudentCard c = new StudentCard(1235, "佐藤花子");
```

thisの省略

参照しているものが自分自身（インスタンス）の変数またはメソッドであることが明らかな場合、thisを省略できる

省略できない場合

```
StudentCard(int id, String name) {
  this.id = id;
  this.name = name;
}
```

省略できる場合

```
StudentCard(int i, String s) {
  this.id = i;
  this.name = s;
}
```

クラス変数

- インスタンス変数はインスタンスごとに保持される情報
- クラス変数はクラスに保持される情報

例：「犬」クラスについて考えてみる
インスタンス変数：名前、性別、毛色
クラス変数：足の本数、尻尾の有無

- インスタンス変数は個別のオブジェクトの属性を表す
- クラス変数はクラスとして持っている属性を表す

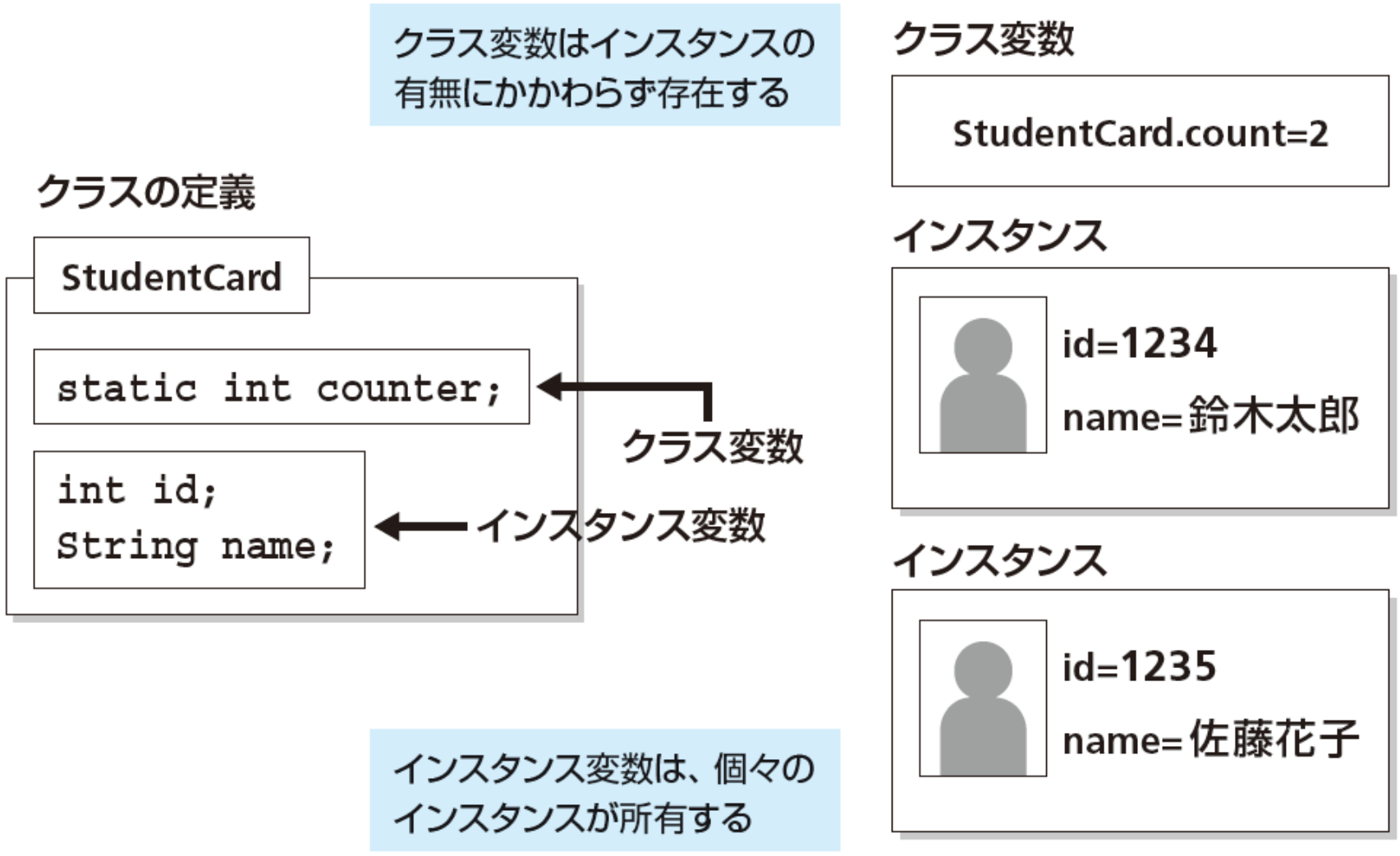
クラス変数の例

- クラス変数を宣言するときには、static 修飾子をつける
- クラス変数は宣言の時に初期化しておく

StudentCardクラスに、counterというint型のクラス変数を追加した例

```
class StudentCard {
  static int counter;
  int id;
  String name;
}
```

クラス変数とインスタンス変数



クラス変数の利用例

```
class StudentCard {
  static int counter = 0;
  int id;
  int name;

  StudentCard(int id, String name) {
    this.id = id;
    this.name = name;
    StudentCard.counter++;
  }
}
```

クラス変数の利用例(続き)

```
System.out.println("StudentCard.counter=" + StudentCard.counter);

StudentCard a = new StudentCard(12345, "鈴木太郎");
System.out.println("StudentCard.counter=" + StudentCard.counter);

StudentCard a = new StudentCard(12346, "佐藤花子");
System.out.println("StudentCard.counter=" + StudentCard.counter);
```

※クラス変数には「クラス名.クラス変数名」でアクセスできる
※クラス変数は、インスタンスを1つも生成しなくても参照できる

クラス名の省略

インスタンス変数を参照することが明らかな場合はthisを省略できた

クラス変数を参照することが明らかな場合はクラス名を省略できる

```
class StudentCard {
  static int counter = 0;
  int id;
  String name;

  Point(int id, String name) {
    this.id = id;
    this.name = name;
    StudentCard.counter++;
  }
}
```

クラスメソッド

- クラスに対して呼び出される「クラスメソッド」というメソッドがある

- インスタンスを生成しなくても
「クラス名.メソッド名」
で呼び出すことができる

- メソッドの宣言にstatic修飾子をつける

クラスメソッドの例

```
class SimpleCalc {
  // 引数で渡された底辺と高さの値から三角形の面積を返す
  static double getTriangleArea(double base, double height) {
    return base * height / 2.0;
  }
}
```

クラスメソッドの使用例

```
System.out.println("底辺が10、高さが5の三角形の面積は"
+ SimpleCalc.getTriangleArea(10, 5) + "です");
```

インスタンスを生成しなくても使用できる
単純な計算処理のように、インスタンス変数を使用しない処理を行うのに便利

クラスの構造の復習

```
class クラス名 {
   インスタンス変数の宣言
   インスタンス変数の宣言
   ・・・
   クラス変数の宣言
   クラス変数の宣言
   ・・・
   コンストラクタの宣言
   コンストラクタの宣言
   ・・・
   インスタンスメソッドの宣言
   インスタンスメソッドの宣言
   ・・・
   クラスメソッドの宣言
   クラスメソッドの宣言
}
```

フィールド

メソッド

演習

空欄に当てはまる用語を選ぼう

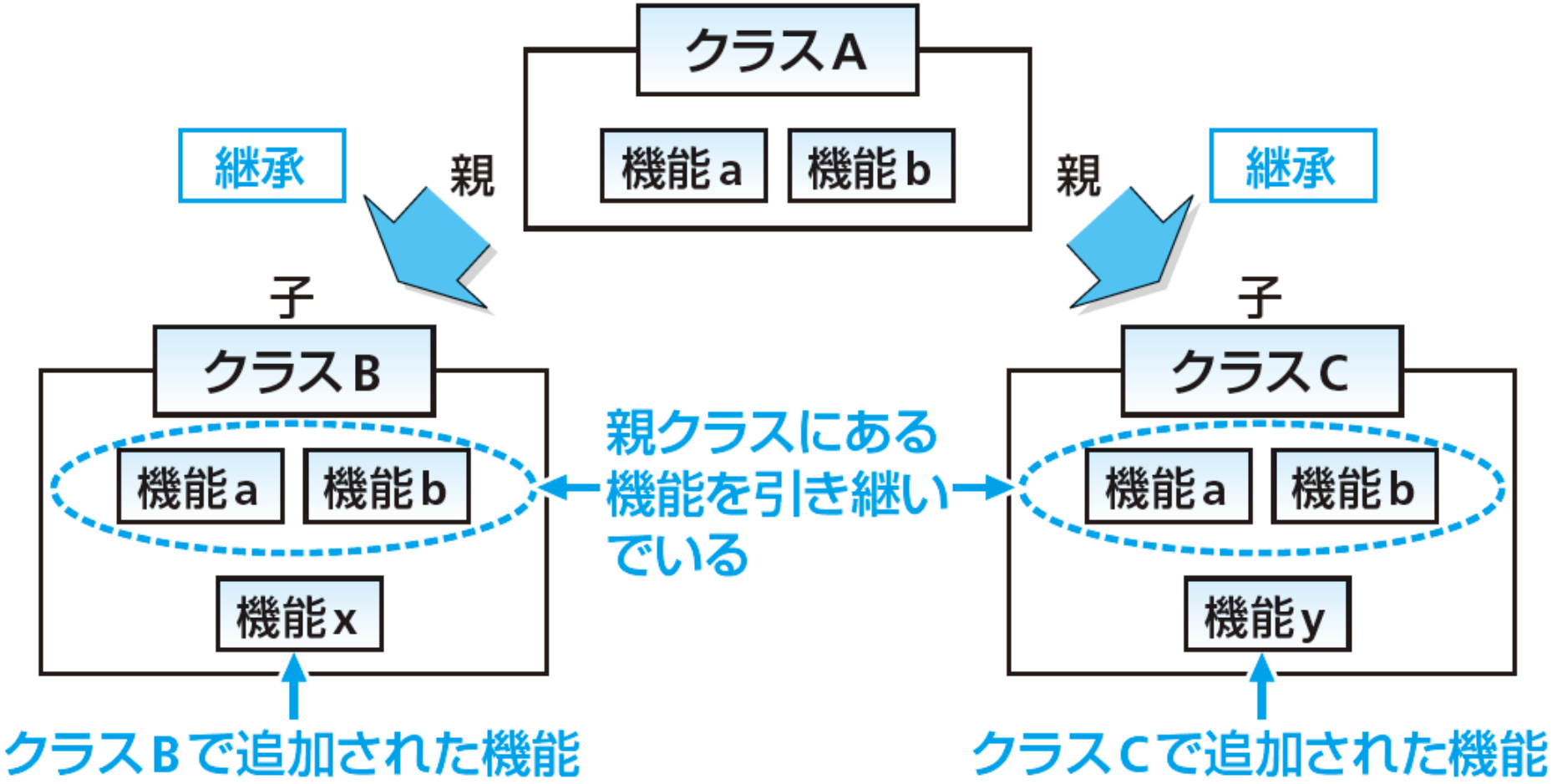
- Java言語は[(1)]指向型の言語であり、クラスを組み合わせてプログラムを作りあげる。クラスは[(1)]の属性や機能を定義したものである。
- クラス定義の中で[(1)]の持つ情報を定義したものを[(2)]とよび、機能を定義したものを[(3)]とよぶ。
- プログラムコードの中でnewを使って、クラスの[(4)]を生成する。
- 変数に格納できるもの（[(3)]の引数の型に指定できるもの）は、intやdoubleなどの[(5)]型と、[(4)] の所在地情報を表わす[(6)]型のどちらかである。
- [(6)]型の変数に、何の所在地情報も入っていない状態を[(7)]というキーワードで表現する。

(a)参照 (b)フィールド (c)変数 (d)関数 (e)オブジェクト (f)メソッド (g)null (h)基本 (i)インスタンス

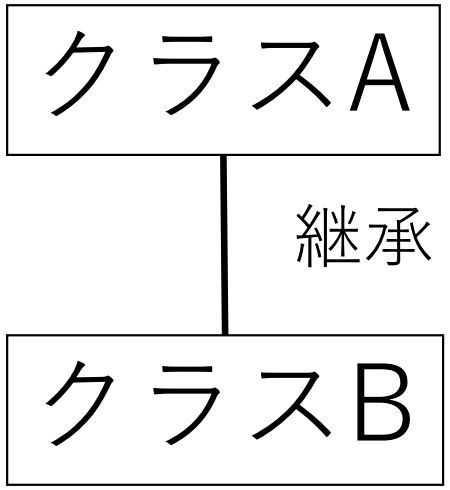
第7章 継承

継承とは

- すでにあるクラスの機能を新しいクラスが引き継ぐこと。機能の拡張が容易にできる。



Javaの継承



クラスAとクラスBの関係

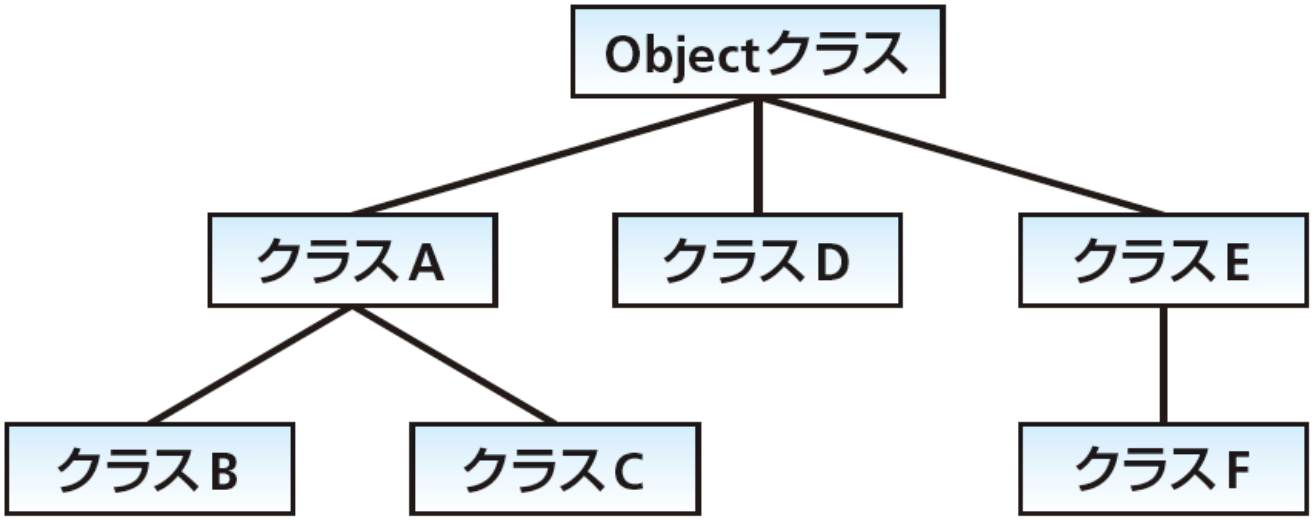
AはBのスーパークラス（親クラス）である

BはAのサブクラス（子クラス）である

BはAを継承したクラスである

BはAから派生したクラスである

Javaの継承

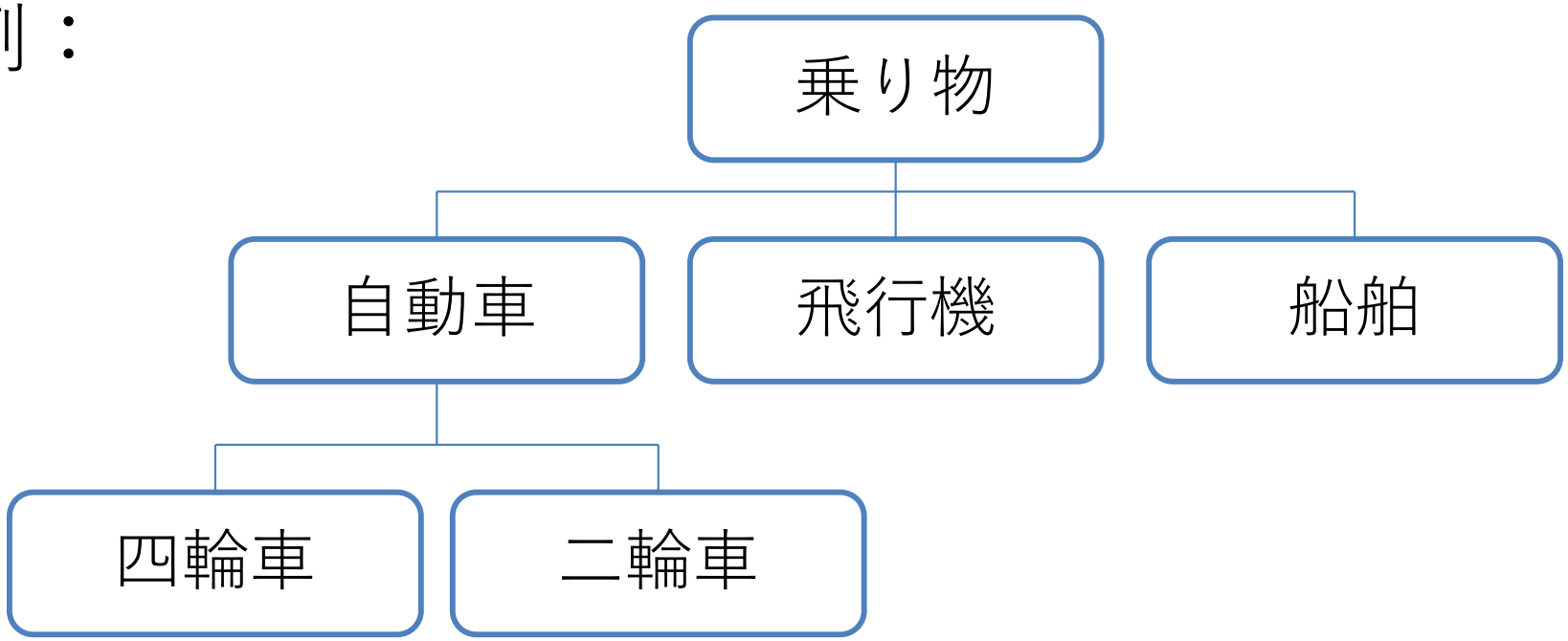


- あるクラスのスーパークラスは1つだけ
- あるクラスのサブクラスは複数可
- 継承の関係を図にすると樹形図になる
- 最も上位のクラスはObjectクラス。すべてのクラスが、このクラスを直接的または間接的に継承する

演習

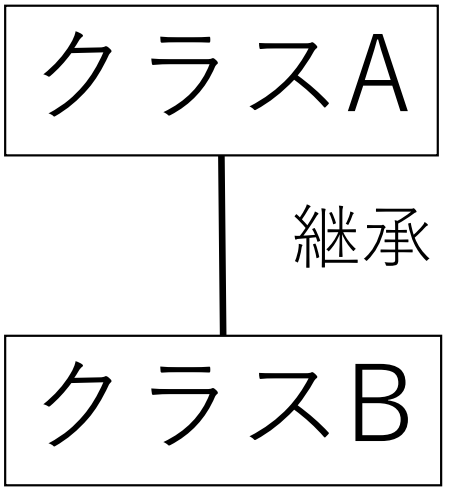
日常を見まわして、クラスの継承関係で表現できそうなものを探してみよう

例：



継承を行うための extends

```
class A {
   クラスAの内容
}

class B extends A {
   追加する新しいフィールドとメソッド
}
```



クラスBがクラスAを継承する場合、クラスBの宣言に「extends A」と記す

Objectクラスの継承

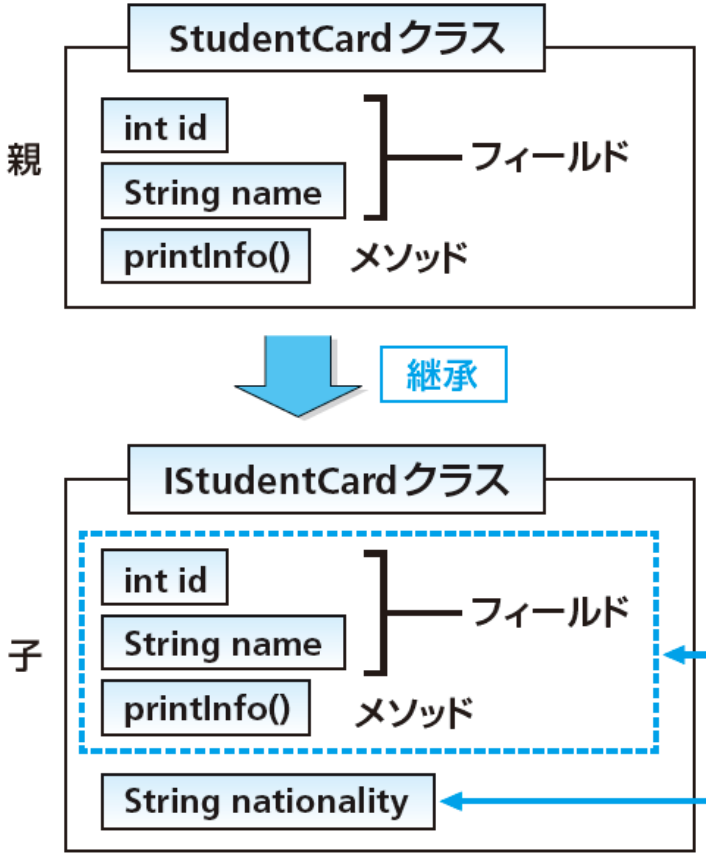
すべてのクラスがObjectクラスを継承するので、次のように書くのが本来の書き方。ただし、`extends Object` は省略できる。

```
class A extends Object {
   クラスAの内容
}

class B extends A {
   追加する新しいフィールドとメソッド
}
```

フィールドとメソッドの継承

```
class StudentCard {
  int id;
  String name;

  void printInfo() {
    System.out.println(this.id);
    System.out.println(this.name);
  }
}
```

```
class IStudentCard extends StudentCard {
  String nationality; //国籍
}
```




```
IStudentCard a = new IStudentCard();
a.id = 2345;
a.name = "John Smith";
a.nationality  = "イギリス";
```

スーパークラスのフィールドを**引き継いで**いる

メソッドのオーバーライド

- スーパークラスにあるメソッドと同じ名前、同じ引数のメソッドをサブクラスでも宣言すること
- サブクラスのメソッドが優先される

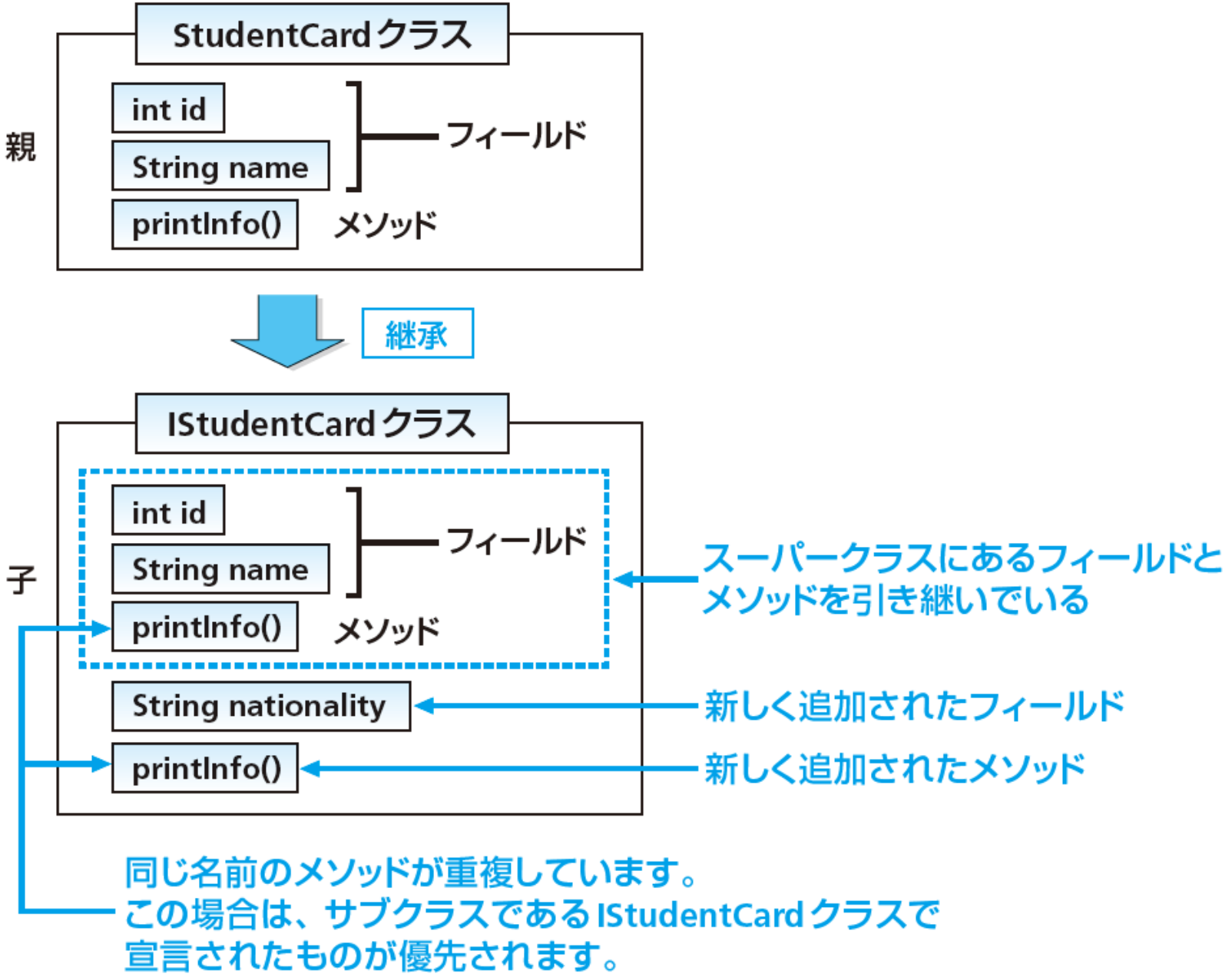
```
class StudentCard {
  int id;
  String name;

  void printInfo() {
    // 略
  }
}
```

```
class IStudentCard extends StudentCard {
  String nationality;

  void printInfo() {
    // 略
  }
}
```

```
IStudentCard a = new IStudentCard();
a.printInfo();
```

オーバーロード（引数が異なり名前が同じメソッドを宣言すること）と単語が似ているので注意。

メソッドのオーバーライド



演習

次のうちクラスの継承について誤っているものを選ぼう

(1) クラスAがクラスBを継承するとき、クラスAをクラスBのサブクラスと呼ぶ

(2) あるクラスを継承するサブクラスが複数存在することもある

(3) あるクラスのスーパークラスが複数存在することもある

(4) サブクラスは、スーパークラスに定義されている変数やメソッドを引き継ぐ

スーパークラスのメソッドの呼び出し

サブクラスからスーパークラスのメソッドを実行するには次のように記述する

```
super.メソッド名(引数);
```

```
class StudentCard {
  int id;
  String name;

  void printInfo() {
    // 略
  }
}
```

```
class IStudentCard extends StudentCard {
  String nationality; // 国籍

  void printInfo() {
    super.printInfo();
  }
}
```

継承関係とコンストラクタの動き

- コンストラクタは継承されない
- コンストラクタが存在しない場合、デフォルトコンストラクタが仮想的に追加される（ただし実際のプログラムコードは変化しない）

```
class B extends A {
}
```

```
class B extends A {
  B() {
    super();
  }
}
```

継承関係とコンストラクタの動き

子クラスのコンストラクタの先頭で、親クラスのコンストラクタを明示的に呼び出さない場合、引数無しのコンストラクタの呼び出しが、仮想的に追加される。

```
class B extends A {
  B() { abc(); }
  B(int i) { def();}
}
```

```
class B extends A {
  B() {super();abc();}
  B(int i) {super();
                  def();}
}
```

スーパークラスのコンストラクタの呼び出し

サブクラスからスーパークラスのコンストラクタを明示的に呼び出すこともできる

```
super(引数);
```

```
class B extends A {
  B(int x) {
    super(x);
  }
}
```

演習

```
class X {
  X() { System.out.println("[X]");}
  void a() { System.out.println("[x.a]");}
  void b() { System.out.println("[x.b]");}
}
class Y extends X {
  Y() { System.out.println("[Y]"); }
  void a() { System.out.println("[y.a]"); }
}
```

上記のようにクラスX,Yが宣言されている場合の、次のプログラムコードを実行した結果を予測しよう

```
X x = new X();        Y y = new Y();
x.a();                y.a();
x.b();                y.b();
```

演習（発展）

```
class X {
  X() { System.out.println("[X()]");}
  X(int i) { System.out.println("[X(int i)]");}
}
class Y extends X {
  Y() { System.out.println("[Y()]"); }
  Y(int i) { System.out.println("[Y(int i)]"); }
}
class Z extends Y {}
```

上記のようにクラスX,Yが宣言されている場合の、次のプログラムコードを実行した結果を確認しよう

```
Y y0 = new Y();
Y y1 = new Y(10);
Z z = new Z();
```

継承関係と代入の可否

スーパークラスの型の変数に、サブクラスのインスタンスを代入できる

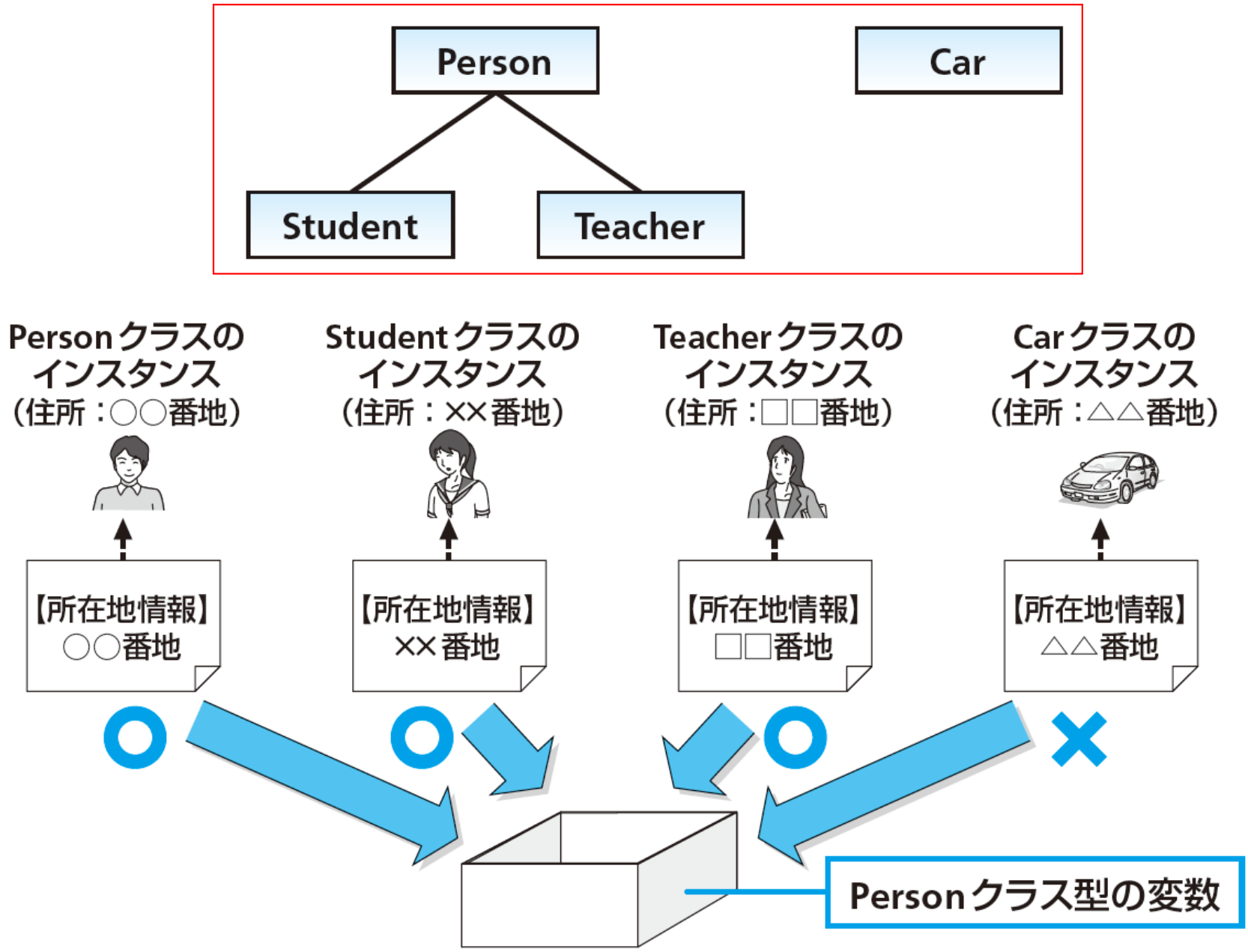
これまでに学習したインスタンスの生成と代入

```
StudentCard a = new StudentCard();
IStudentCard b = new IStudentCard();
```

StudentCard型の変数にIStudentCardクラスのインスタンスを代入できる

```
StudentCard a = new IStudentCard();
```

継承関係と代入の可否


Person
Car
Student
Teacher
Personクラスの
インスタンス
（住所：○○番地）
Studentクラスの
インスタンス
（住所：××番地）
Teacherクラスの
インスタンス
（住所：□□番地）
Carクラスの
インスタンス
（住所：△△番地）
【所在地情報】
○○番地
【所在地情報】
××番地
【所在地情報】
□□番地
【所在地情報】
△△番地
Personクラス型の変数

ポリモーフィズム（多態性）

```
class Person {
  void work() {
    // "人です。仕事します。"
  }
}

class Student extends Person {
  void work() {
   // "学生です。勉強します。"
  }
}

class Teacher extends Person {
  void work() {
    // "教員です。授業します。"
  }
  void makeTest() {
  }
}
```

```
Person[] persons = new Person[3];
persons[0] = new Person();
persons[1] = new Student();
persons[2] = new Teacher();

for(int i = 0; i < 3; i++) {
        persons[i].work();
}
```

同じ型の変数に入っていても、そのインスタンスによって動作が異なる。

メソッドの引数とポリモーフィズム

```
// 通常の3倍働いてもらう
void workThreeTimes(Person p) {
  p.work();
  p.work();
  p.work();
}
```

上のようなメソッドには、引数として、Personクラスのサブクラスのインスタンス（new Teacher(), new Student()）を渡すことができる。実際の処理は、インスタンスに定義されているworkメソッドが実行される。

演習

次のようにクラスA,B,Cが定義されています

```
class A { }
class B extends A { }
class C { }
```

次の変数の宣言と代入で誤りがあるものを選ぼう

```
(1) A a = new A();
(2) A a = new B();
(3) A a = new C();
(4) B b = new A();
(5) B b = new B();
(6) B b = new C();
```

型変換（キャスト）

スーパクラスの型に代入されたサブクラスの参照を、サブクラスの型にキャストできる

```
Person p = new Teacher();
Teacher t = (Teacher)p;
t.makeTest();
```

```
Person p = new Teacher();
((Teacher)p).makeTest();
```

第8章 抽象クラスとインタフェース

修飾子とアクセス制御

- 修飾子とは、クラス、フィールド、メソッドの性質を指定するもの
- アクセスを制御するためのものをアクセス修飾子と呼ぶ

アクセス修飾子	意味・機能
`public`	ほかのどのクラスからもアクセスできる。
`protected`	サブクラスまたは同じパッケージ内のクラスからしかアクセスできない（注❼-11）。
なし	同一パッケージのクラスからしかアクセスできない。
`private`	同じクラス内からしかアクセスできない。

private修飾詞とカプセル化

- private 修飾詞を使用すると、他のクラスからアクセスできない（不可視）になる
- このように、他のクラスからインスタンス変数を隠すことを「カプセル化」という
- 「カプセル化」はオブジェクト指向プログラミングで大事な役割を果たす

private修飾子の使用例

```
class Car {
  private int speed; // 速度(Km/h)

  // speedの値を1増やす。ただし最大でも80までとする。
  public void speedUp() {
    if(speed < 80) {
      speed++;
    }
  }

  // speedの値を1減らす。ただし0以下にはならない。
  public void speedDown() {
    if(speed > 0) {
      speed--;
    }
  }
}
```

（発展）アクセッサを経由したアクセス

```
class Example {
  private int valueA;
  private int valueB;

  public int getValueA() {
      return valueA;
  }
  public void setValueA(int a) {
      valueA = a;
  }
  public int getValueB() {
      return valueB;
  }
  public void setValueA(int b) {
      valueB = b;
  }
}
```

final修飾子

- 後から変更してはいけないものにfinal修飾子を付ける
- クラス、メソッド、フィールドにつけると、それぞれ次のような意味を持つ

クラス：サブクラスを作れない

メソッド：サブクラスでオーバーライドできない

フィールド：値を変更できない（定数）

定数の使用例

```
public final static double PI =
                              3.14159265358979793;
```

```
public final static int
ADULT_AGE = 20;
```

定数を使った方が可読性があがる。保守がしやすくなる。

```
if(age == 20) {    }
```

```
if(age == ADULT_AGE) {    }
```

static 修飾子

クラス変数、クラスメソッドを宣言するときに使用する

```
static int counter = 0;
```

```
static double getSum(int x, int y) {
    return x + y;
}
```

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

抽象クラス

抽象クラスはインスタンスを作れないクラス

abstract修飾子をつけて宣言する

```
abstract class Shape {
}
```

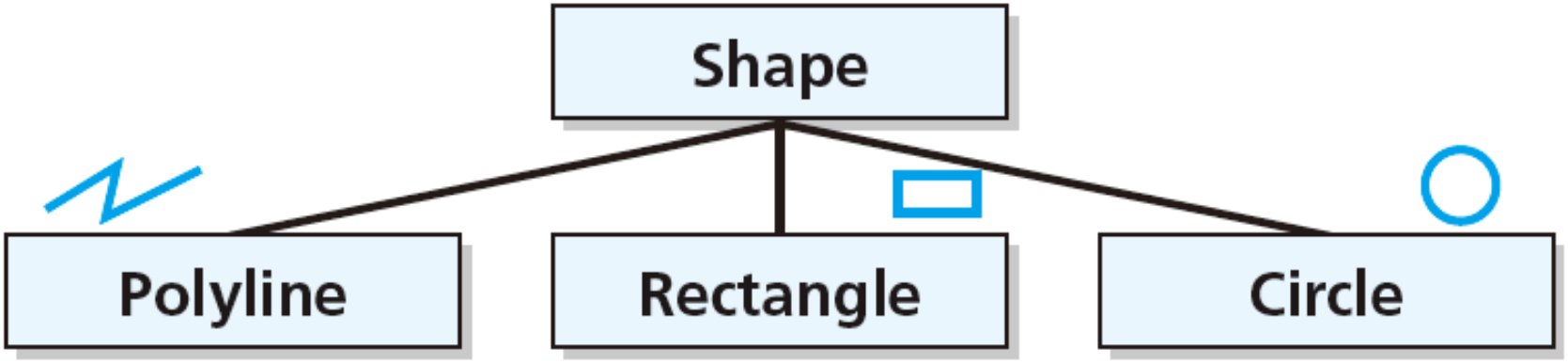
上のShapeクラスは抽象クラスとして宣言されているのでインスタンスを作れない

```
Shape s = new Shape();
```

抽象クラス

どのような時に抽象クラスが必要なのか？

ポリモーフィズムを使いたい＆スーパークラスのインスタンスは作らせたくない（作っても意味がない）



抽象メソッド

- 抽象クラスにしか作れない
- abstract修飾子をつけて宣言する
- 実体が無い

```
abstract class Shape {
  abstract void draw();
}
```

サブクラスは必ず抽象メソッドをオーバーライトしなくてはならない。

注）サブクラスも抽象クラスならこの限りではない

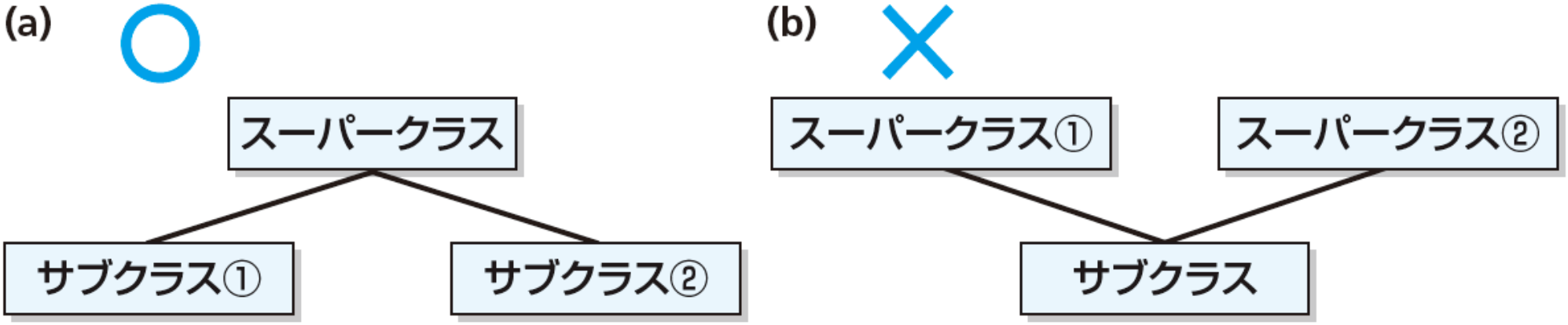
```
abstract class Shape {
  abstract void draw();
}

class Polyline extends Shape
{
  void draw() {
    // 折れ線を描画
  }
}
class Rectangle extends Shape
{
  void draw() {
    // 長方形を描画
  }
}
class Circle extends Shape {
  void draw() {
    // 円を描画
  }
}
```

```
Shape[] shapes = new Shape[3];
shapes[0] = new Polyline();
shapes[1] = new Rectangle();
shapes[2] = new Circle();

for(int i = 0; i < 3; i++) {
        shapes[i].draw();
}
```

Javaの継承

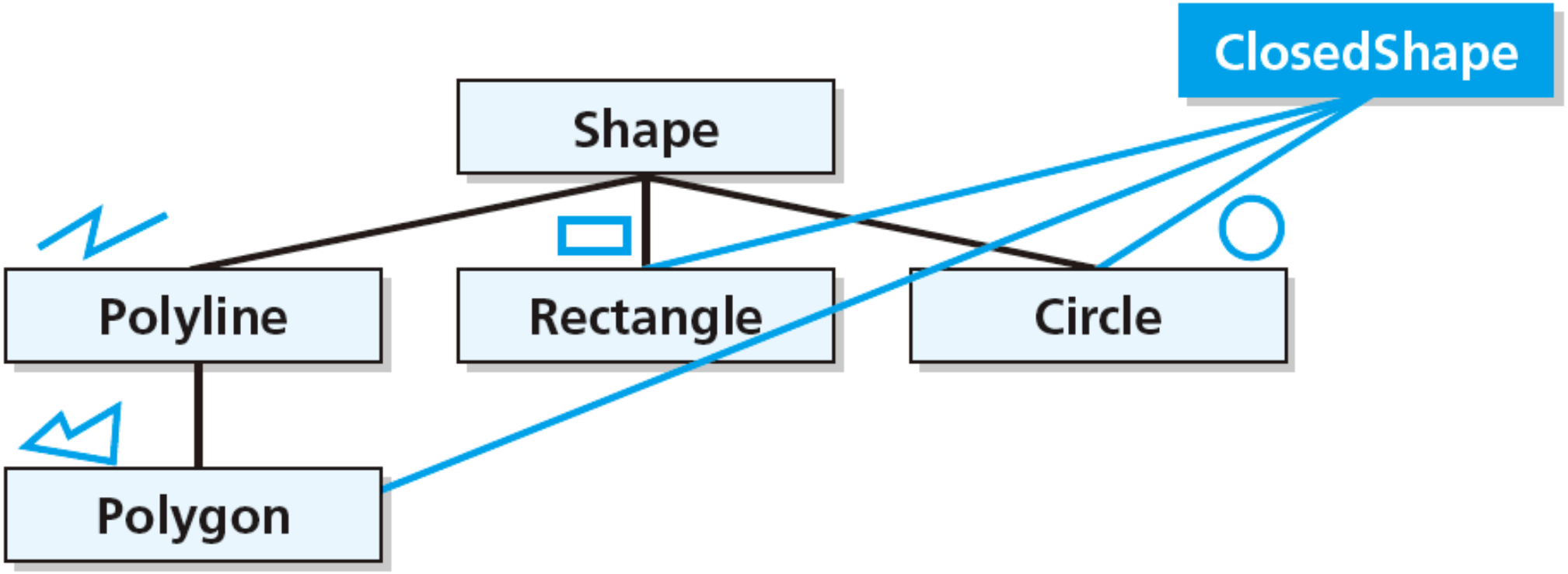
スーパークラスは1つだけ
（多重継承ができない）

(a) スーパークラスを継承するサブクラスはいくつでも作れます。
(b) サブクラスは複数のスーパークラスを持つことはできません。



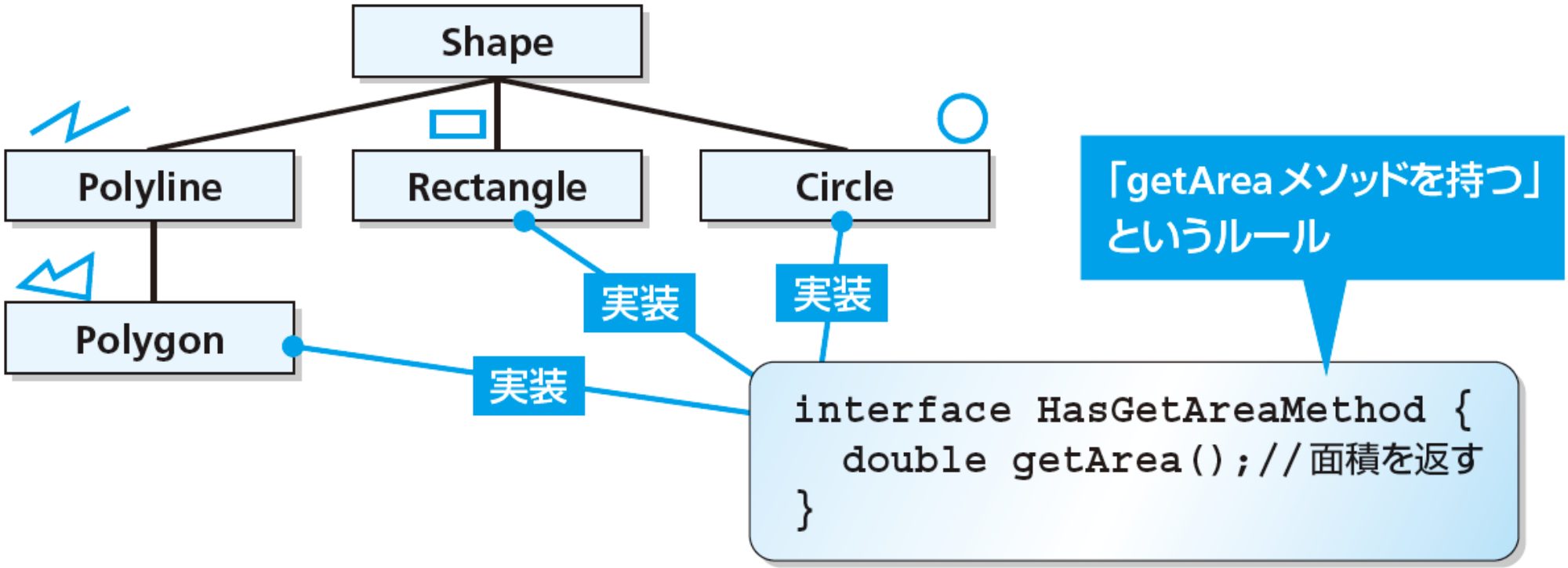
多重継承をしたい場合もある

継承関係にないクラス間で、ポリモーフィズムを活用したいときに、複数のスーパークラスを持たせたい



インタフェースとは

- クラスが持つべきメソッドを記したルールブック
- 「そのルールブックに記されたメソッドを持っているよ」と宣言する（「インタフェースを実装する」という）ことで、継承関係にないクラス間でポリモーフィズムを使用できる



インタフェースの使い方

インタフェースの宣言（クラスの宣言と似ている。メソッドの中身は定義しない。「このようなメソッドを持つ」というルールだけ定める）

```
interface インタフェース名 {
  メソッドの宣言
}
```

```
interface HasGetAreaMethod {
  double getArea();
}
```

インタフェースの実装（「このクラスはルールに従って、決められたメソッドを持っている」と宣言する）

```
class クラス名 implements インタフェース名 {
  クラスの内容
}
```

インタフェースの使用

```
interface HasGetAreaMethod {
  double getArea();
}
```

```
class Rectangle implements HasGetAreaMethod {
  double getArea() { return width*height;}
}
```

```
class Circle implements HasGetAreaMethod {
  double getArea() { return r*r*3.14;}
}
```

```
HasGetAreaMethod r = new Rectangle();
HasGetAreaMethod c = new Circle();
```

※ インタフェースの参照型の変数に、インタフェースを実装したクラスのインスタンスを代入できる。

複数インタフェースの実装

カンマ(,)で区切って、複数のインタフェースを実装できる

```
class A implements Interface A, Interface B {
    クラスの内容
}
```

※ 複数のインタフェースを実装する場合には、それぞれのインタフェースで宣言されているメソッド全ての実装が必要

定数の宣言

インタフェースのフィールドで宣言された変数は、暗黙的にpublic static final という3つの修飾子がついているものとして扱われ、値が変更できない。

宣言例

```
interface MoveDirection {
  int UP = 0;
  int DOWN = 1;
}
```

上手く使えばコードが読みやすくなる。

使用例

```
void move(int direction) {
  switch(direction) {
    case MoveDirection.UP:
      // 上に移動する処理
    break;
    case MoveDirection.DOWN:
      // 下に移動する処理
    break;
  }
}
```

演習

```
interface I {}
abstract class A {}
class B extends A {}
class C implements I {}
```

上記のように宣言されている場合、次の中で誤っているものはどれでしょう

```
1. A a = new A();
2. B b = new B();
3. C c = new C();
4. I i = new I();
5. A b = new B();
6. B a = new A();
7. I b = new B();
8. I c = new C();
```