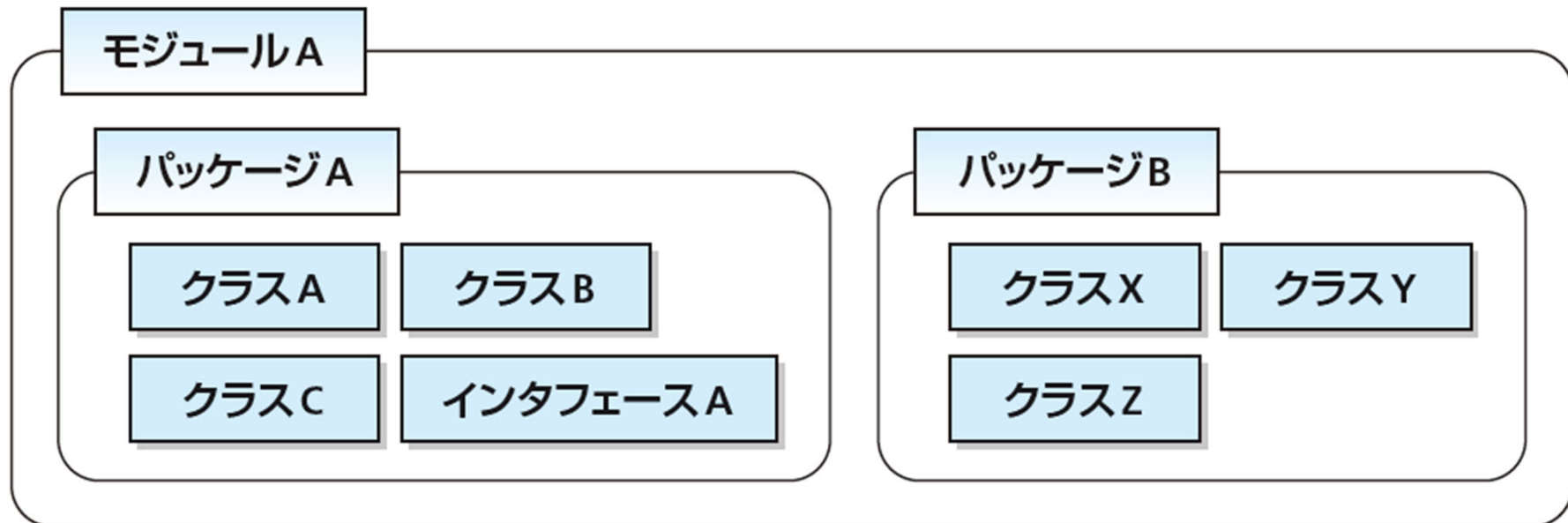


# 第1章 パッケージとJava API

# モジュールとパッケージ

- Javaには「クラスライブラリ」と呼ばれる、便利なクラスやインタフェースが準備されている
- クラスライブラリは必要に応じて自由に使える
- クラスライブラリは「モジュール」と「パッケージ」で管理されている



# Javaの主なモジュールとパッケージ

モジュール名	パッケージ名	説明
ジャバ・ベース <code>java.base</code>	ジャバ・ラング <code>java.lang</code>	Javaの基本的な機能を提供するクラス群（多くのクラスで使用する）
	ジャバ・ユーティル <code>java.util</code>	便利な機能を提供するクラス群（第5章のコレクションフレームワークで扱う）
	ジャバ・アイオー <code>java.io</code>	入出力を扱うクラス群（第7章の入出力処理で扱う）
	ジャバ・ネット <code>java.net</code>	ネットワーク機能を提供するクラス群（第10章のネットワーク接続処理で扱う）
ジャバ・デスクトップ <code>java.desktop</code>	ジャバエックス・スイング <code>javax.swing</code>	グラフィカルなインタフェースを実現するための、ボタンやチェックボックスなどのコントロールを提供するクラス群（第8章のGUIアプリケーションの作成で扱う）
	ジャバ・エーダプリーティ <code>java.awt</code>	四角や円などの図形の描画に関連するクラス群（第9章のグラフィックス描画で扱う）
	ジャバ・エーダプリーティ・イベント <code>java.awt.event</code>	マウス操作やキーボード操作などのイベントを処理するためのクラス群（第8章、第9章のイベント処理で扱う）

# パッケージに含まれるクラスの利用

---

java.utilパッケージに含まれるRandomクラスのインスタンスを生成する例

```
java.util.Random rand = new java.util.Random();
```

パッケージ名    クラス名

（パッケージ名）．（クラス名）という表現をクラスの**完全限定名**という

# import宣言

---

完全限定名を毎回記述するのは大変

とても長い完全限定名の例

```
javax.xml.bind.annotation.adapters.XmlAdapter
```

パッケージ名

クラス名

import宣言をすれば、プログラムの中でパッケージ名の記述を省略できる

```
import パッケージ名.クラス名;
```

# import宣言の効果

---

```
.....  
java.util.Random rand = new java.util.Random();  
.....
```

完全限定名で  
毎回書くのはたいへん

import宣言を使おう!

クラス名を書くだけで  
済むようになった!

```
import java.util.Random;  
.....  
Random rand = new Random();  
.....
```

# import宣言の使用例

---

```
import java.util.Random;

public class ImportExample {
    public static void main(String[] args) {
        Random rand = new Random();
        // 0~1の間のランダムな値を出力する
        System.out.println(rand.nextDouble());
    }
}
```

java.util.Random のimport宣言をしている

# 複数のクラスのimport宣言

---

- 複数のクラスをimport宣言する場合、そのクラスの数だけ宣言する

```
import java.util.ArrayList;  
import java.util.Random;
```



\*（アスタリスク）記号  
を使って省略できる

```
import java.util.*;
```

java.utilパッケージに含まれる全てのクラスを  
import宣言したのと同じ



# パッケージの階層

---

例えば次の二つは異なるパッケージ

java.utilパッケージ

java.util.zipパッケージ

```
import java.util.*;
```

と記述してもjava.util.zipパッケージのクラス  
は使用できない

次のように記述する

```
import java.util.*;  
import java.util.zip.*;
```

# java.langパッケージ

---

- java.langパッケージにはJavaの基本的な機能を提供するクラスが含まれる
- 「`System.out.println()`」のSystemクラスもjava.langパッケージに含まれる
- 「`import java.lang.*;`」という記述は省略できる

# API仕様書

Javaにはあらかじめ膨大な数のクラスやインタフェースが準備されている



API仕様書で使い方を調べられる

<http://docs.oracle.com/javase/jp/11/docs/api/>

※ バージョンが異なる場合は赤字部分を変更

# API仕様書で確認できるクラス情報

---

モジュール `java.base`

パッケージ `java.util`

## クラス **Random**

`java.lang.Object`  
`java.util.Random`

すべての実装されたインタフェース:

`Serializable`

直系の既知のサブクラス:

`SecureRandom`, `ThreadLocalRandom`

- Randomクラスはjava.baseモジュールのjava.utilパッケージに含まれる
- java.lang.Objectクラスを継承している
- Serializableインタフェースを実装している
- サブクラスにSecureRandomクラスとThreadLocalRandomクラスがある

# クラスの情報

- クラスの説明
- フィールドの説明
- コンストラクタの説明
- メソッドの説明

```
public class Random
extends Object
implements Serializable
```

このクラスのインスタンスは、一連の擬似乱数を生成するために使用されます。クラスでは48ビットのシードを使い、このシードは線形合同法で変更されます。詳細はDonald Knuth著『*The Art of Computer Programming, Volume 2*』のセクション3.2.1を参照してください。

2つのRandomインスタンスが同じシードで生成されている場合、それぞれに対して同じシーケンスでメソッド呼出しを行うと、同じシーケンスで数値が生成され返されます。この特性を保証するために、Randomクラスには固有のアルゴリズムが指定されています。Javaコードの絶対的な移植性の保持のために、Javaの実装はここに示されているRandomクラスのアルゴリズムをすべて使用する必要があります。ただし、Randomクラスのサブクラスは、すべてのメソッドの一般規約に準拠したものであればほかのアルゴリズムも使用できます。

Randomクラスによって実装されるアルゴリズムでは、各呼出しで擬似乱数的に生成された最大32ビットを提供できる**protected**ユーティリティ・メソッドが使用されます。

多くのアプリケーションの場合、**Math.random()**メソッドを使うほうが簡単です。

**java.util.Random**のインスタンスはスレッド・セーフです。ただし、複数のスレッドで同じ**java.util.Random**インスタンスを並行して使用すると、競合が発生してパフォーマンスが低下する可能性があります。マルチ・スレッド設計では、代わりに**ThreadLocalRandom**を使用することを検討してください。

**java.util.Random**のインスタンスには安全な暗号化が施されていません。セキュリティ保護を必要とするアプリケーションで使用するために安全な暗号化の施された擬似乱数ジェネレータを取得するには、代わりに**SecureRandom**を使用することを検討してください。

導入されたバージョン:

1.0

関連項目:

直列化された形式

# Stringクラス

---

- String は java.langパッケージに含まれるクラス
- 次の2通りでインスタンスを生成できる

```
String message = "こんにちは";
```

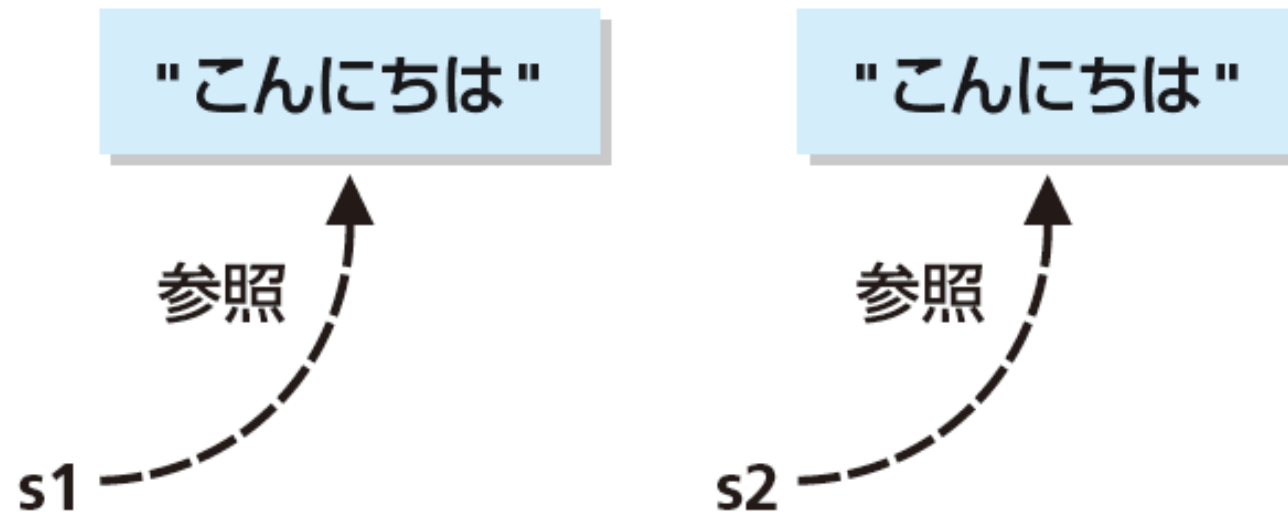
```
String message = new String("こんにちは");
```

new を使わなくてもインスタンスを作れる特殊なクラス

# Stringオブジェクトの生成方法による違い

---

```
String s1 = new String("こんにちは");  
String s2 = new String("こんにちは");  
System.out.println(s1 == s2); // false
```



異なる2つのインスタンスが生成される  
たまたま文字列が同じだけ

# Stringオブジェクトの生成方法による違い

---

```
String s1 = "こんにちは";  
String s2 = "こんにちは";  
System.out.println(s1 == s2); // true
```



1つのインスタンスを参照する



# Stringクラスのメソッド

---

Stringクラスには、文字列を扱うための便利なメソッドがある

```
String str = "Javaの学習";  
System.out.println(str.length()); // 7  
System.out.println(str.indexOf("学習")); // 5  
System.out.println(str.indexOf("Ruby")); // -1  
System.out.println(str.contains("学習")); // true  
System.out.println(str.contains("Ruby")); // false  
  
String str2 = str.replace("Java", "Java言語");  
System.out.println(str2); // Java言語の学習
```

# Stringクラスのメソッド

---

文字列を区切り記号で分割する例

```
String str = "2020/11/22";  
String[] items = str.split("/");  
for(int i = 0; i < items.length; i++) {  
    System.out.println(items[i]);  
}
```

実行結果

```
2020  
11  
22
```

# Mathクラス

java.lang.Mathクラスには数学的な計算を行う便利なクラスメソッドが多数ある

メソッド	説明
<code>static double <b>abs</b>(double d)</code>	dの絶対値を返す
<code>static int <b>abs</b>(int i)</code>	iの絶対値を返す
<code>static double <b>sin</b>(double radians)</code>	radiansの正弦（サイン）を返す
<code>static double <b>cos</b>(double radians)</code>	radiansの余弦（コサイン）を返す
<code>static double <b>tan</b>(double radians)</code>	radiansの正接（タンジェント）を返す
<code>static double <b>sqrt</b>(double d)</code>	dの平方根を返す
<code>static double <b>pow</b>(double x, double y)</code>	xのy乗を返す
<code>static double <b>log</b>(double d)</code>	dの自然対数を返す
<code>static double <b>max</b>(double d, double e)</code>	dとeの大きいほうの値を返す
<code>static int <b>max</b>(int i, int j)</code>	iとjの大きいほうの値を返す
<code>static double <b>min</b>(double d, double e)</code>	dとeの小さいほうの値を返す
<code>static int <b>min</b>(int i, int j)</code>	iとjの小さいほうの値を返す
<code>static double <b>random</b>()</code>	0.0以上で1.0より小さい乱数を返す

# Mathクラスの使用

---

- java.langパッケージはimport文を省略できる
- クラスメソッドの使用方法（復習）  
「Math.メソッド名(引数);」

```
class MathExample {  
    public static void main(String[] args) {  
        System.out.println("-5の絶対値は" + Math.abs(-5));  
        System.out.println("3.0の平方根は" + Math.sqrt(3.0));  
        System.out.println("半径2の円の面積は" + 2*2*Math.PI);  
        System.out.println("sin60° は" +  
                               Math.sin(60.0*Math.PI / 180.0));  
    }  
}
```

# パッケージの作成

---

パッケージは自分で作成できる。

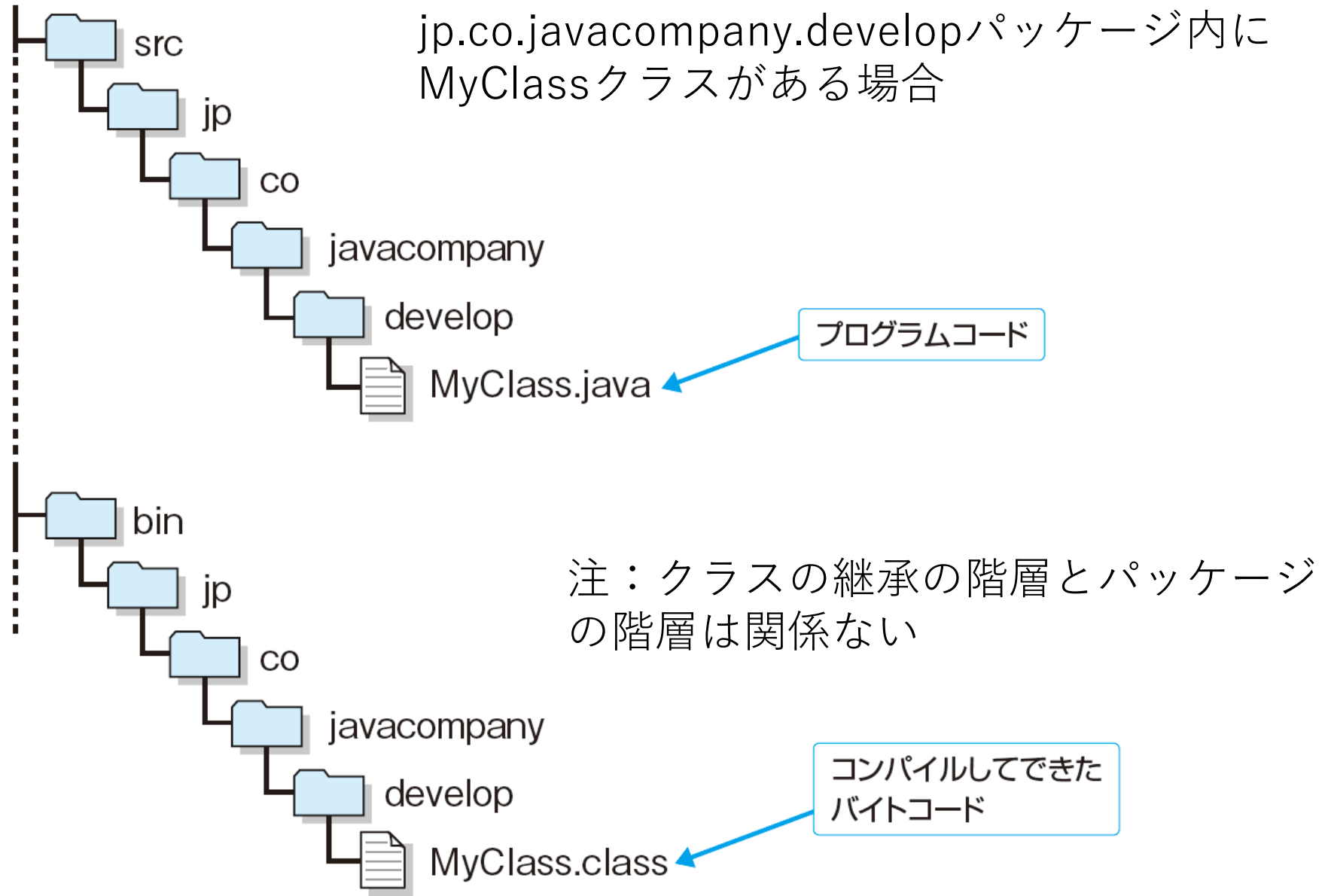
```
package パッケージ名;
```

とプログラムコードの先頭に記述する。

```
package mypackage;  
  
public class MyClass {  
    public void printMessage() {  
        System.out.println("mypackage.MyClassのprintMessageメソッド");  
    }  
}
```

Eclipseでは[ファイル]-[新規]-[パッケージ]でパッケージを新規作成。  
その中にクラスを作成する。

# パッケージの階層構造とフォルダの階層構造



# パッケージ名の設定

---

- パッケージ名は他人が作ったものと同じものではない（**名前の衝突**）
- ドメイン名をパッケージ名に使用することが多い。並び順は逆。  
例：`jp.co.javacompany.develop`

# クラスのアクセス制御

---

アクセス修飾子を使って、パッケージ外部からのアクセスを制御できる

クラスとインタフェースの宣言で利用できるアクセス修飾子

アクセス修飾子	アクセスできる範囲
<code>public</code>	どのパッケージからもアクセスできる
なし	同じパッケージの中からのみ



# メソッドとフィールドのアクセス修飾子

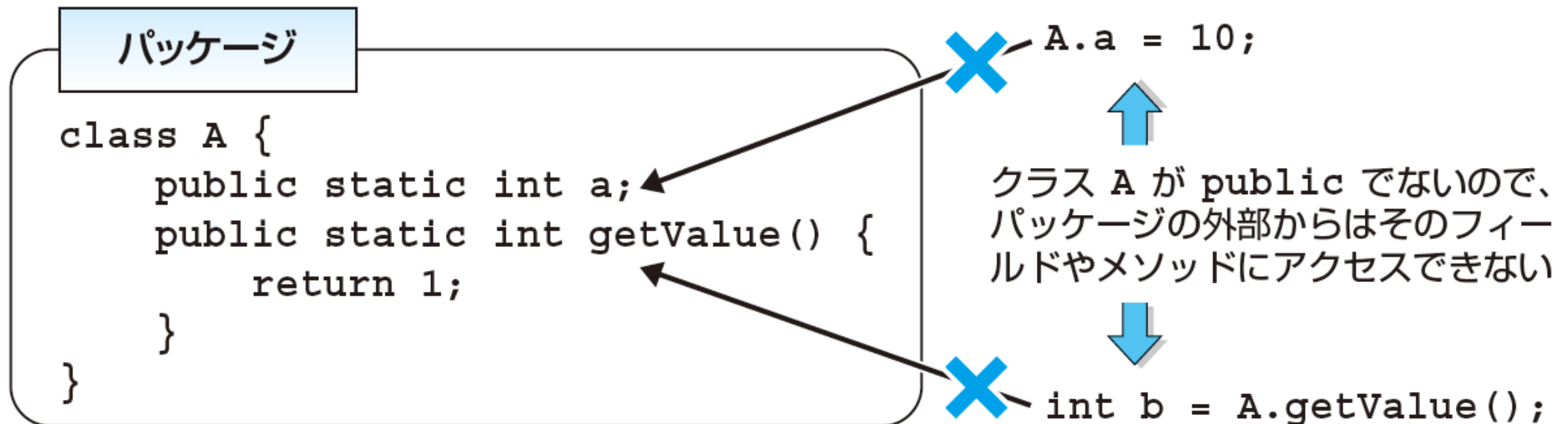
---

メソッドとフィールドの宣言で利用できるアクセス修飾子

アクセス修飾子	アクセスできる範囲
<code>public</code>	どのパッケージのクラスからもアクセスできる
<code>protected</code>	サブクラスまたは同じパッケージ内のクラスからのみ
なし	同じパッケージ内のクラスからのみ
<code>private</code>	同じクラス内からのみ

# アクセス修飾子の優先順位

フィールドやメソッドのアクセス修飾子が public であっても、クラスのアクセス修飾子が public でない場合は、パッケージの外からはアクセスできない



# 複数のクラス宣言を持つプログラムコード

---

- 1つの.javaファイルで複数のクラスを宣言できる
- public修飾子をつけられるのは1つだけ
- public修飾子をつけたクラス名とファイル名は一致する必要がある

MultiClassExample.java

```
class SimpleClass {  
    String str;  
    SimpleClass(String str) {  
        this.str = str;  
    }  
}  
  
public class MultiClassExample {  
    public static void main(String[] args) {  
        SimpleClass sc = new SimpleClass("Hello.");  
        System.out.println(sc.str);  
    }  
}
```

## 第2章 例外处理

# 例外の発生

---

- プログラムが動作する時にトラブルが発生することがある。これを「例外」と言う

「例外が発生する」

「例外が投げられる」

「例外がスロー(throw)される」

などと表現する

# 例外が発生する例

---

ゼロでの除算

```
int a = 4;  
int b = 0;  
System.out.println(a / b);
```

通常処理

$4 \div 2 \rightarrow 2$

問題の発生する処理

$4 \div 0 \rightarrow ?$

例外

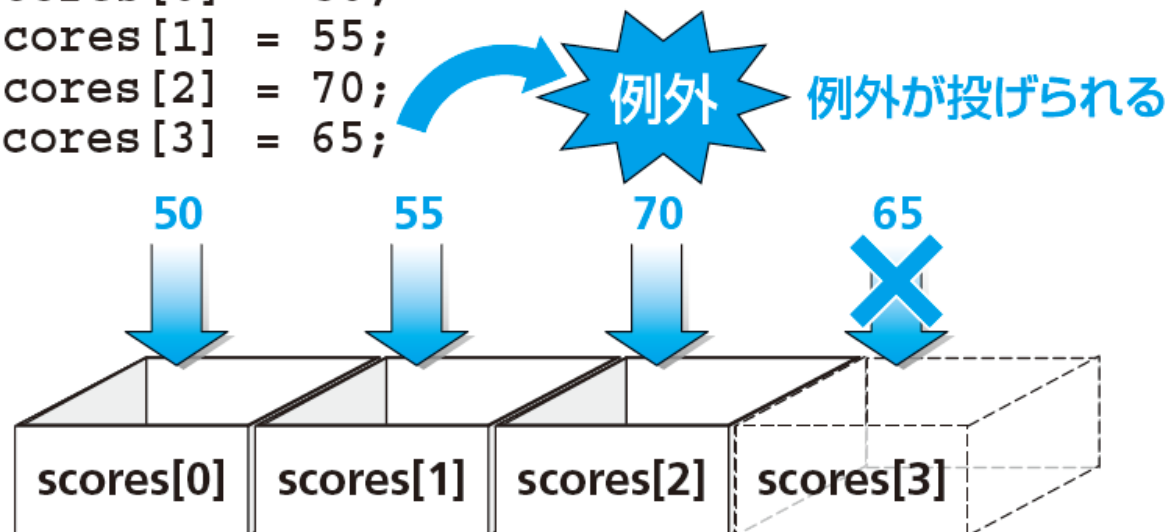
例外が投げられる

# 例外が発生する例

範囲を超えたインデックスの参照

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

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



# 投げられた例外をキャッチする

- 例外が投げられたときにも、その例外をキャッチして処理を続けることができる仕組みがある
- try～catch文を使う
- 「例外処理」と呼ぶ





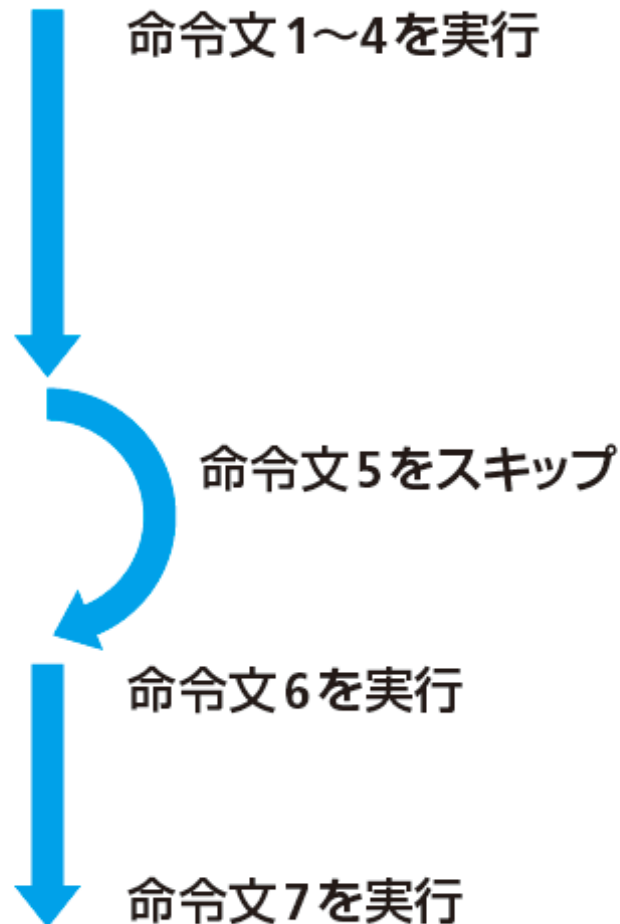
# try～catch文

```
try {  
    tryブロック  
    本来実行したい処理だが、  
    例外が投げられる可能性がある処理  
}  
catch(例外の型 変数名) {  
    catchブロック  
    例外が投げられたときの処理  
}  
finally {  
    finallyブロック  
    最後に必ず行う処理  
}
```

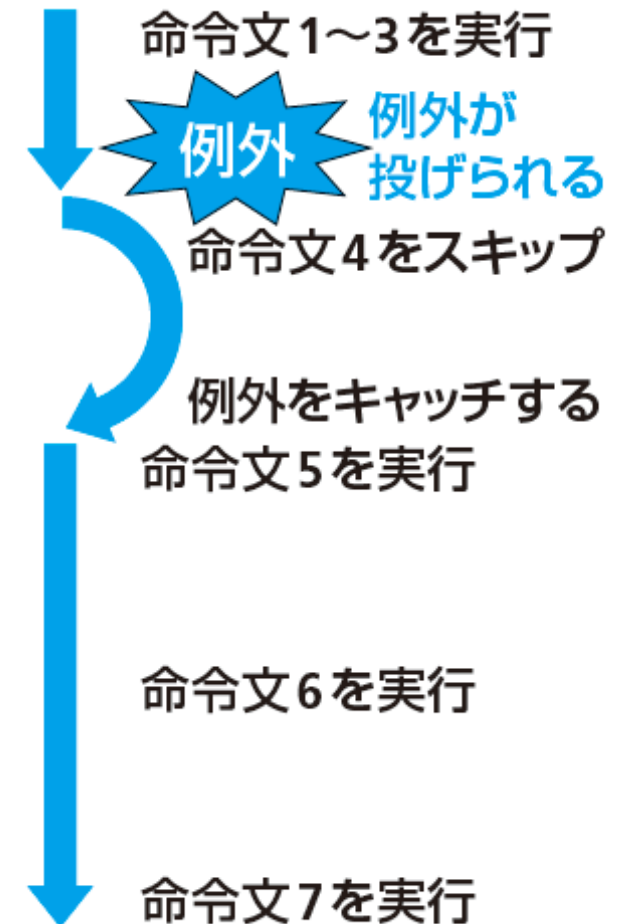
# 処理の流れ

```
try {  
    命令文 1  
    命令文 2  
    命令文 3  
    命令文 4  
}  
catch (Exception e) {  
    命令文 5  
}  
finally {  
    命令文 6  
}  
  
命令文 7
```

## 【通常の処理の流れ】



## 【命令文 3 で例外が投げられた場合の処理の流れ】



# 例外処理の例

---

```
public class ExceptionExample3 {  
    public static void main(String[] args) {  
        int a = 4;  
        int b = 0;  
        try {  
            int c = a / b;  
            System.out.println("cの値は" + c);  
        }  
        catch (ArithmeticException e) {  
            System.out.println("例外をキャッチしました");  
            System.out.println(e);  
        }  
        System.out.println("プログラムを終了します");  
    }  
}
```

# finallyの処理

---

```
public static void main(String[] args) {  
    int a = 4;  
    int b = 0;  
    try {  
        int c = a / b;  
        System.out.println("cの値は" + c);  
    }  
    catch (ArithmeticException e) {  
        System.out.println("例外をキャッチしました");  
        System.out.println(e);  
        return;  
    }  
    finally {  
        System.out.println("finallyブロックの処理です");  
    }  
    System.out.println("プログラムを終了します");  
}
```

# catchブロックの検索

```
class SimpleClass {  
    void doSomething() {  
        int array[] = new int[3];  
        array[10] = 99; // 例外が発生する  
        System.out.println("doSomethingメソッドを終了します");  
    }  
}
```

```
public class ExceptionExample5 {  
    public static void main(String args[]) {  
        SimpleClass obj = new SimpleClass();  
        try {  
            obj.doSomething(); // 例外の発生するメソッドの呼び出し  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("例外をキャッチしました");  
            e.printStackTrace();  
        }  
    }  
}
```

# 例外オブジェクト

---

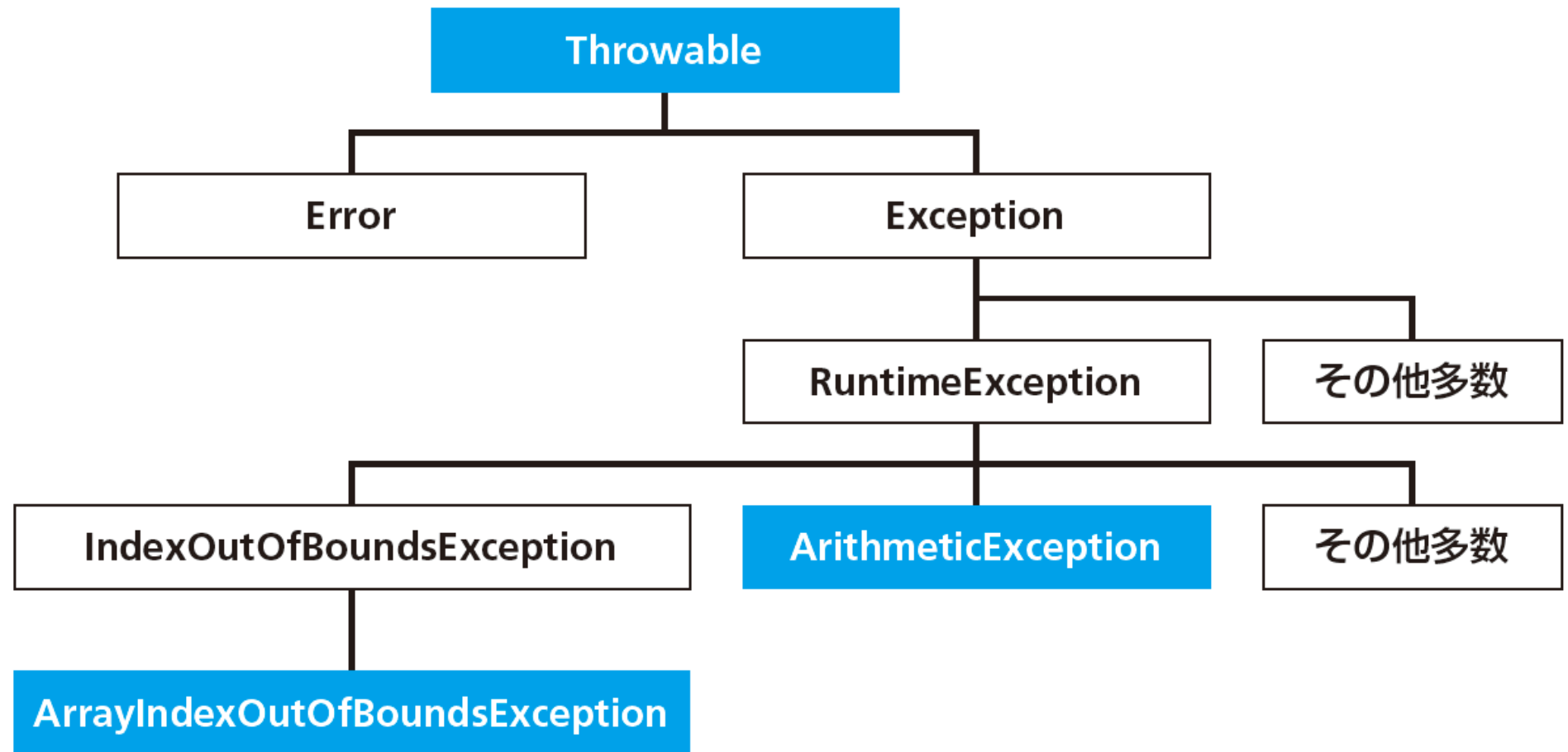
- 例外が発生した時には  
java.lang.Exceptionクラスの  
「例外オブジェクト」が投げられる
- 実際は、Exceptionクラスのサブクラス
- 例外の種類によって異なる  
ゼロ除算：ArithmeticException  
配列の範囲を超えた参照  
ArrayIndexOutOfBoundsException

# 例外オブジェクトの種類による場合分け

---

```
try {  
    例外が投げられる可能性のある処理  
}  
catch(例外の型1 変数名1) {  
    例外の型1の例外が投げられたときの処理  
}  
catch(例外の型2 変数名2) {  
    例外の型2の例外が投げられたときの処理  
}  
finally {  
    最後に必ず行う処理  
}
```

# 例外クラスの階層



Exceptionオブジェクトが投げられる可能性がある場合はtry～catch文を書かなくてはならない。

ただし、RuntimeExceptionだけは、try～catch文が無くてもよい。



# 例外を作成して投げる

---

- 例外オブジェクトを自分で作成して投げる  
ことができる
- 例外オブジェクトの作成

```
Exception e = new Exception("〇〇という例外が発生しました");
```

- 例外オブジェクトを投げる

```
throw e;
```

※通常はExceptionクラスをそのまま使用せず、サブクラスを作って例外を投げる。

# メソッドの外への例外の送出

---

自分で作成した例外オブジェクトをthrowした後の処理

1. try～catch文で囲んで処理する
2. **メソッドの外に投げる**  
(メソッドの呼び出し側で処理する)

2の場合はメソッドの宣言に throws を追加する

戻り値	メソッド名(引数)	<b>throws</b>	<b>例外の型</b>	{
	例外を投げる可能性のあるメソッドの内容			
				}

# メソッドの外への例外の送出の例

```
class Person {
    int age;
    void setAge(int age) throws InvalidAgeException {
        if(age < 0) {
            throw new InvalidAgeException("マイナスの値が指定された");
        }
        this.age = age;
    }
}

public class ExceptionExample7 {
    public static void main(String[] args) {
        Person p = new Person();
        try {
            p.setAge(-5);
        } catch (InvalidAgeException e) {
            System.out.println(e);
        }
    }
}
```

# 第3章 スレッド

# スレッドとは

---

- スレッドは処理の流れ
- これまで見てきたプログラムは命令が1つずつ処理された ← シングルスレッド
- Javaでは複数の処理を同時に行うことができる ← マルチスレッド

例1. ファイルをダウンロードしながら画面の表示を更新する

例2. アニメーション表示しながらユーザのマウス操作を受け付ける

# スレッドの作成

---

- 通常のプログラムは1つのスレッドで実行される
  - 新しいスレッドを追加できる
  - 次の2つの方法がある
1. Threadクラスを継承した新しいクラスを作成する
  2. Runnableインタフェースを実装した新しいクラスを作成する

# 方法1. Threadクラスを拡張する

---

```
class MyThread extends Thread {  
    // Thread クラスのrunメソッドをオーバーライド  
    public void run() {  
        命令文  
    }  
}
```

- Threadクラスを継承したクラスを作成する
- runメソッドをオーバーライドする
- 「命令文」に処理を記述する

# 方法1のThreadの使用例

---

```
class MyThread extends Thread {
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("MyThreadのrunメソッド("+i+)");
        }
    }
}

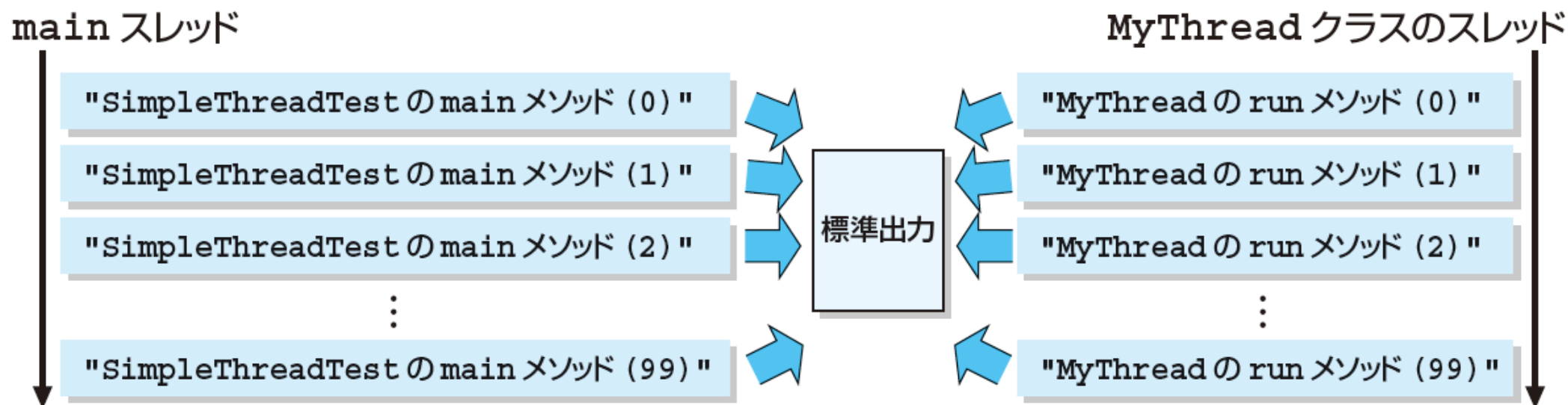
public class SimpleThreadTest {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for(int i = 0; i < 100; i++) {
            System.out.println("SimpleThreadTestのmainメソッド("+i+)");
        }
    }
}
```



# Threadの使用例

- Threadクラスのrunメソッドを直接呼ばない
- startメソッドを呼ぶことで、別スレッドの処理が始まる
- 複数のスレッドが並行して処理を進める



## 方法2. Runnableインタフェースを実装する

---

方法1がいつでも使えるとは限らない。

例：他のクラスのサブクラスは、Threadクラスのサブクラスにならない



### 方法2

- ・ Runnableインタフェースを実装する
- ・ Runnableインタフェースに定義されているrunメソッドを追加する

# 方法2のThread使用例

---

```
class MyThread implements Runnable {
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("MyThreadのrunメソッド("+i+")");
        }
    }
}

public class SimpleThreadTest2 {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        Thread thread = new Thread(t);
        thread.start();
        for(int i = 0; i < 100; i++) {
            System.out.println("SimpleThreadTest2のmainメソッド("+i+")");
        }
    }
}
```

# スレッドを一定時間停止させる

---

`Thread.sleep(停止時間 (ミリ秒));`

```
public class SleepExample {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++) {  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
            System.out.print("*");  
        }  
    }  
}
```

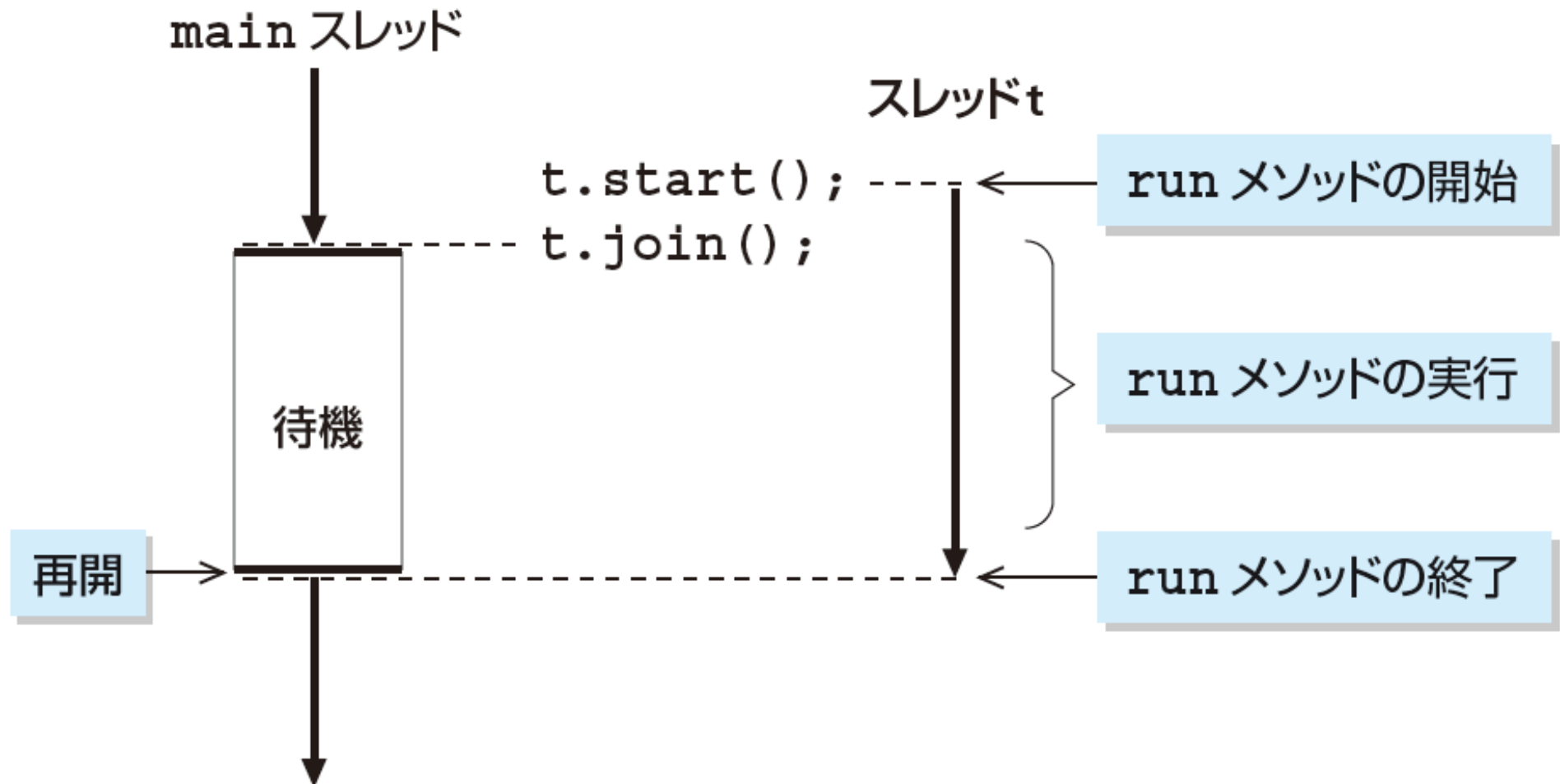
# スレッドの終了を待つ

---

スレッドAからスレッドBの`join`メソッドを呼ぶと、スレッドBの処理が終わるまでスレッドAは次の処理に移らない。

```
MyThread t = new MyThread();  
t.start();  
  
try{  
    t.join();  
} catch (InterruptedException e) {  
    System.out.println(e);  
}
```

# joinメソッドの呼び出しによる待機



# スレッドを止める

---

- スレッドはrunメソッドの処理が終了すると、動作が止まる
- whileループの条件を制御してrunメソッドを終了させるのが一般的

```
class MyThread extends Thread {  
    public boolean running = true;  
    public void run() {  
        while(running) {  
            // 命令文  
        }  
        System.out.println("runメソッドを終了");  
    }  
}
```

# マルチスレッドで問題が生じるケース




---

- 複数のスレッドが1つの変数に同時にアクセスすると、不整合が生じる場合がある
- スレッドAが変数*i*の値を1増やす
- スレッドBが変数*i*の値を1増やす
- 上記の処理が同時に行われると*i*の値が1しか増えない場合がある







# 問題が生じないケース

変数moneyの値（現在の値は98）をスレッドAとスレッドBが1ずつ増やす処理

スレッドAの処理	moneyの値	スレッドBの処理
	98	
スレッドAがmoneyの値を参照します。現時点で値は98です。	98	
スレッドAがmoneyに98+1を代入します。moneyの値は99になります。	99	
	99	スレッドBがmoneyの値を参照します。現時点で値は99です。
	100	スレッドBがmoneyに99+1を代入します。moneyの値は100になります。

# 問題が生じるケース

変数moneyの値（現在の値は98）をスレッドAとスレッドBが1ずつ増やす処理。スレッドAの処理にスレッドBが割り込んだ

スレッドAの処理	moneyの値	スレッドBの処理
	98	
スレッドAがmoneyの値を参照します。現時点で値は98です。	98	
	98	スレッドBがmoneyの値を参照します。現時点で値は98です。
	99	スレッドBがmoneyに98+1を代入します。moneyの値は99になります。
スレッドAがmoneyに98+1を代入します。moneyの値は99になります。	99	

# スレッドの同期

---

メソッドに**synchronized**修飾子をつけると、そのメソッドを実行できるスレッドは一度に1つに限定される

```
static synchronized void addOneYen() {  
    money++;  
}
```

あるスレッドによってメソッドが実行されている間は、他のスレッドはその処理が終わるまで待機することになる。  
これをスレッドの**同期**と呼ぶ

## 第4章 ガーベッジコレクションとメモリ

# プログラムの実行とメモリ管理

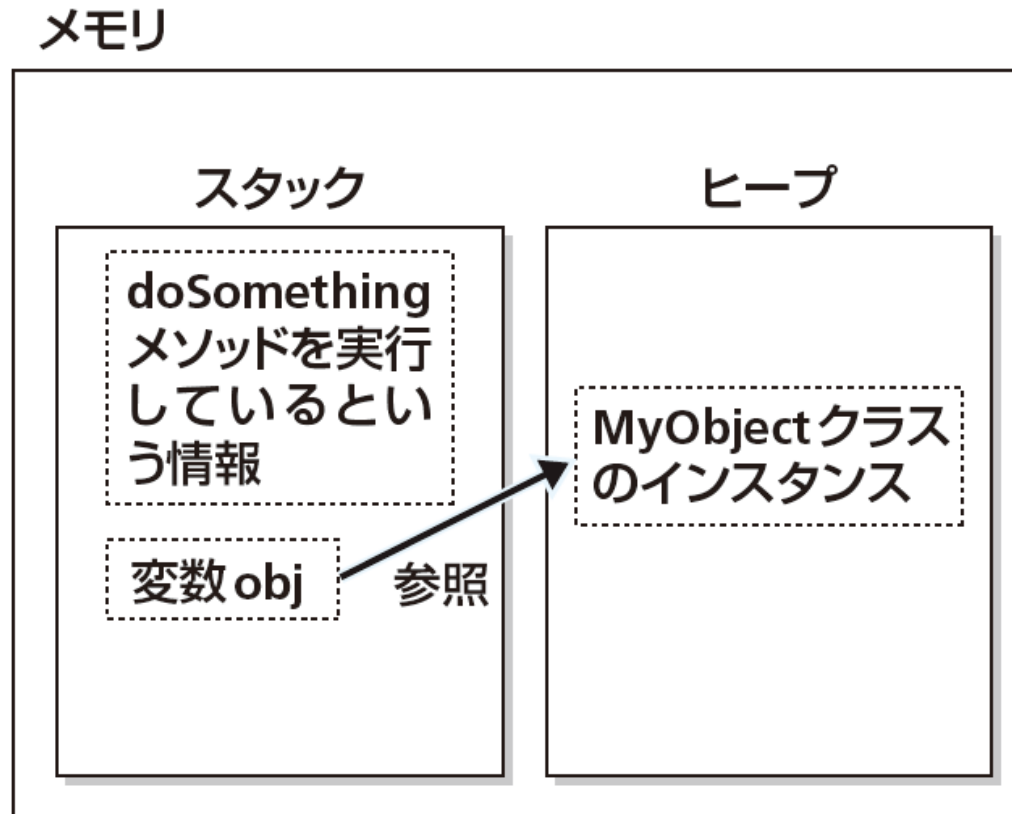
---

- プログラム実行中に覚えておかなくてはならない情報はメモリに格納される。
- メモリにはスタックとヒープと呼ばれる領域がある。

スタック	<ul style="list-style-type: none"><li>• メソッドの呼び出し履歴</li><li>• メソッドの中で宣言されたローカル変数の値</li><li>• メソッドの処理が終わると、そのメソッドに関する情報と変数が削除される</li></ul>
ヒープ	<ul style="list-style-type: none"><li>• 生成されたインスタンス</li><li>• ガーベッジコレクタによって管理される</li></ul>

# スタックとヒープ

```
MyObject doSomething() {  
    MyObject obj = new MyObject();  
    return obj;  
}
```



# 空きメモリサイズの確認

---

```
class DataSet {
    int x;
    int y;
}

public class FreeMemoryTest {
    public static void main(String[] args) {
        System.out.println("空きメモリサイズ:" +
            Runtime.getRuntime().freeMemory());
        DataSet[] data = new DataSet[100];
        for(int i = 0; i < 100; i++) {
            data[i] = new DataSet();
            System.out.println("生成済みインスタンス数:" + (i + 1) +
                " 空きメモリサイズ:" +
                Runtime.getRuntime().freeMemory());
        }
    }
}
```

# 使用できるサイズは有限

---

- インスタンスを大量に生成すると、ヒープを消費する。全て使い切ってしまうと実行時エラーが発生する。
- メソッドの呼び出しの階層があまりに深いとスタックも使いきる可能性がある  
(再帰呼び出しなどを行った場合)



# ガーベッジコレクション

---

- プログラムの中で不要になったインスタンスの情報を削除し、メモリの空き領域を増やす処理
- Java仮想マシンが自動で行う

インスタンスが不要になる



インスタンスがどこからも参照されなくなる

# ガーベッジコレクションの対象になるタイミング

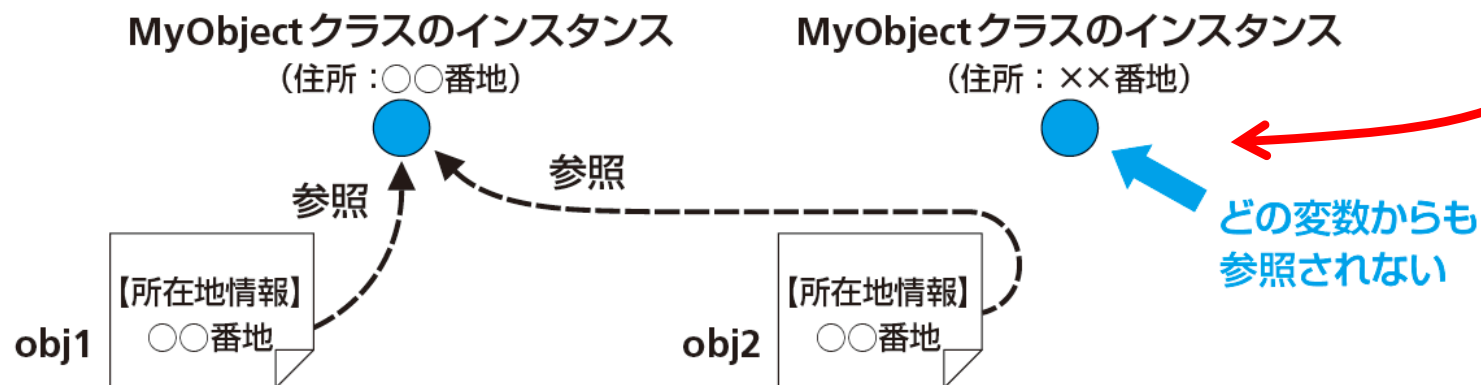
```
MyObject obj1 = new MyObject();  
MyObject obj2 = new MyObject();
```



```
obj2 = obj1;
```

参照先の変更

ガーベッジコレクションの対象



# ガーベッジコレクションの対象になるタイミング

---

```
MyObject obj = new MyObject();  
obj = null;
```

nullの代入

生成されたインスタンスは誰からも参照されなくなる

```
void doSomething() {  
    MyObject obj = new MyObject();  
}
```

メソッドの処理が終わると、

生成されたインスタンスは誰からも参照されなくなる。

# ガーベッジコレクションが実行されるタイミング

---

- Java仮想マシンが適切なタイミングで実行する
- ガーベッジコレクションは時間のかかる処理なので、ある程度不要なインスタンスが貯まってからまとめて行われる
- `Runtime.getRuntime().gc();`  
または  
`System.gc();`  
の命令文で、ガーベッジコレクションを指定したタイミングで行わせることもできる

# 第5章 コレクション

# 大きさの変わる配列

---

## Javaでの配列の使い方

```
MyObject[] objects = new MyObject[100];  
objects[0] = new MyObject();
```

- 要素の数を指定する必要がある
- 格納できる要素の数の上限は後から変更できない



- java.utilパッケージにArrayListという便利なクラスがある
- 要素の数を最初に決める必要が無い
- 便利なメソッドが備わっている

# 型を指定するクラス

---

型パラメータを指定したArrayListオブジェクトの生成

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

- ・ < >の中に配列に格納するオブジェクトの型を指定する ←型パラメータ
- ・ ここで指定した型のオブジェクトだけを格納できるようになる
- ・ 上の例ではString型を指定

# ジェネリクス

---

- API仕様書でArrayListを見るとクラス名が次のようになっている

```
ArrayList<E>
```

- EはArrayListに格納できるオブジェクトのクラスを指定する型パラメータ
- getメソッドは次のように記述されている

```
public E get(int index)
```

- 戻り値の型がEとなっている。型パラメータで指定したクラスの参照が戻り値。
- 特定のクラスに特化したArrayListにできる。



# ラッパークラス

---

- ArrayListに格納できるのは参照型
- 基本型の値は格納できない
- 「ArrayList<int>」は誤り
- ラッパークラスを使用する
- ラッパークラスとは基本型をオブジェクトとして扱うためのクラス

```
ArrayList<Integer> arr =  
    new ArrayList<Integer>();  
arr.add(new Integer(50));  
Integer integer0 = arr.get(0);  
int i = integer0.intValue();
```

# 基本型とそれに対応するラッパークラス

---

基本形	ラッパークラス
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

# コレクションフレームワーク

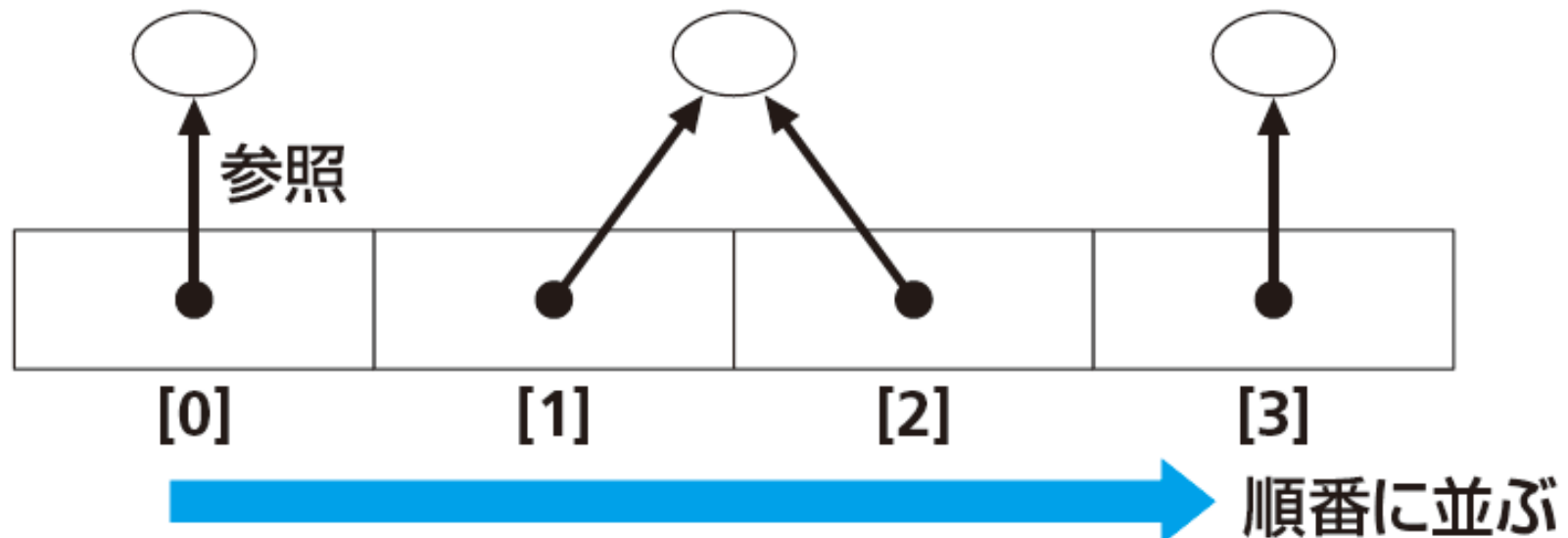
---

- 実際のプログラムでは、複数の（多数の）オブジェクトを扱うことが多い
- 配列よりも便利に使えるクラスがJavaには予め多数準備されている。これらをコレクションフレームワークと呼ぶ（ArrayListもこれに含まれる）
- 目的に応じて使い分ける
  - 膨大な数のオブジェクトから目的のオブジェクトを素早く取り出したい
  - 同じインスタンスへの参照が重複して格納されないようにしたい
  - キーワードを使ってオブジェクトを取りだしたい
  - 特定の値で並び変えたい
- 大きく分けると、リスト・マップ・セットの3種類

# リスト

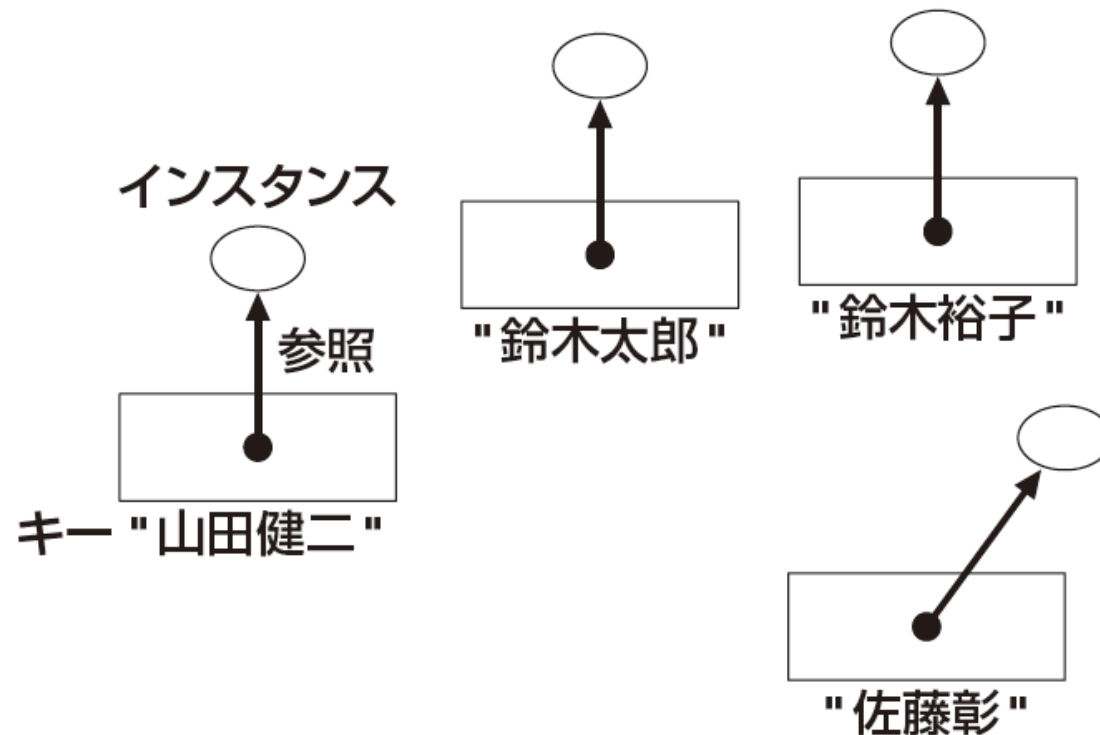
- オブジェクトが順番に並ぶ
- 「先頭から2番目のもの」「最後に追加したもの」を取りだすことができる
- 異なる要素が同一のオブジェクトを参照できる
- Listインタフェースを実装する

インスタンス



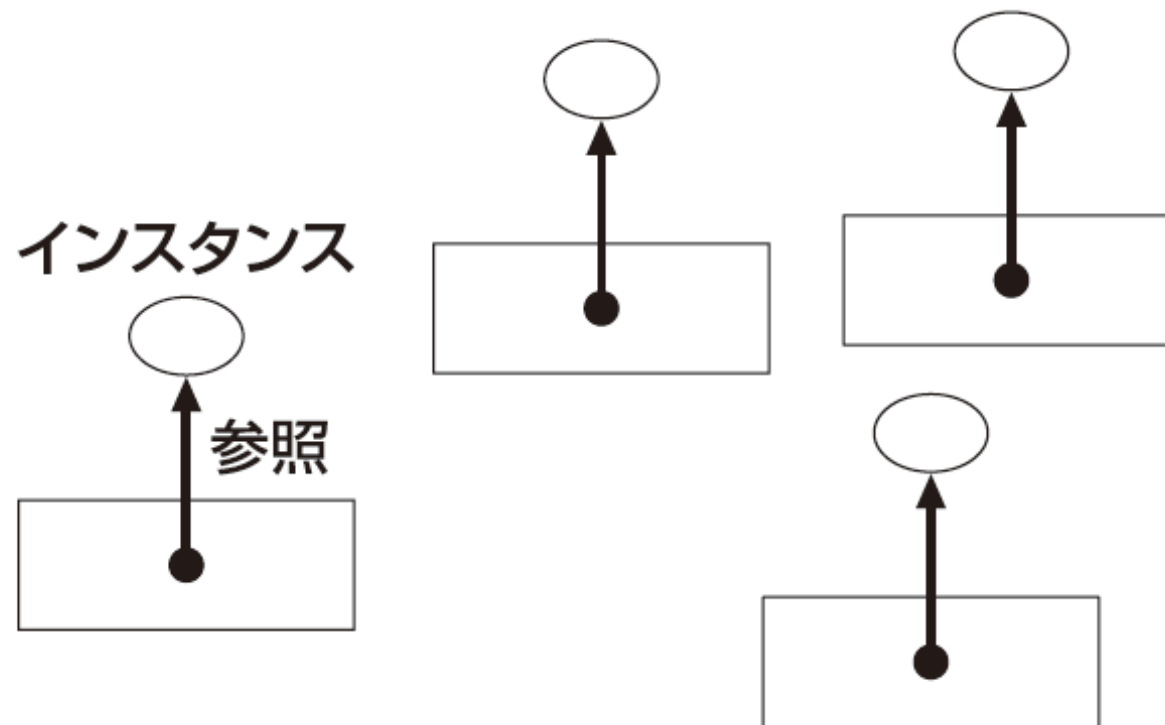
# マップ

- キーと値（オブジェクト）のペアを管理する
- キーでオブジェクトを取りだせる
- キーは重複してはいけない
- Mapインタフェースを実装する



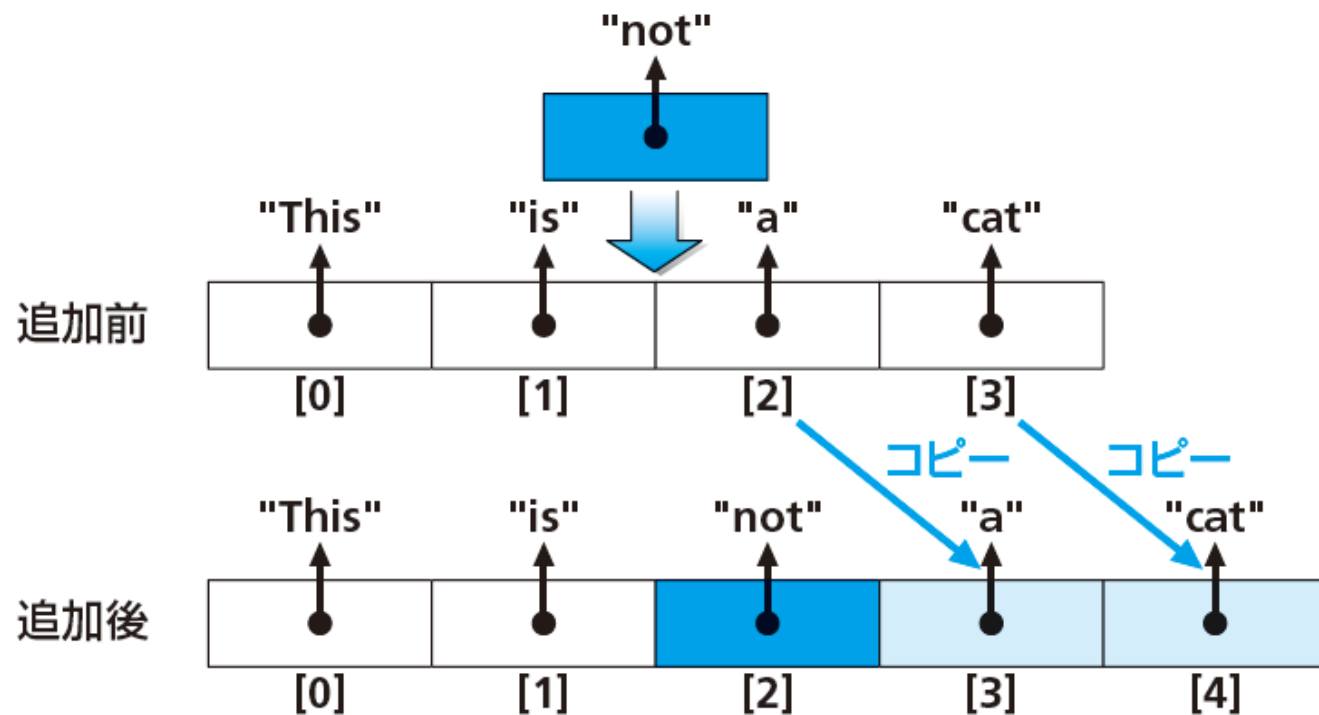
# セット

- 格納されるオブジェクトに重複がないことを保証する
- 個々のオブジェクトを指定して取り出す方法が無い（リストやマップと組み合わせて使用する）
- Setインタフェースを実装する



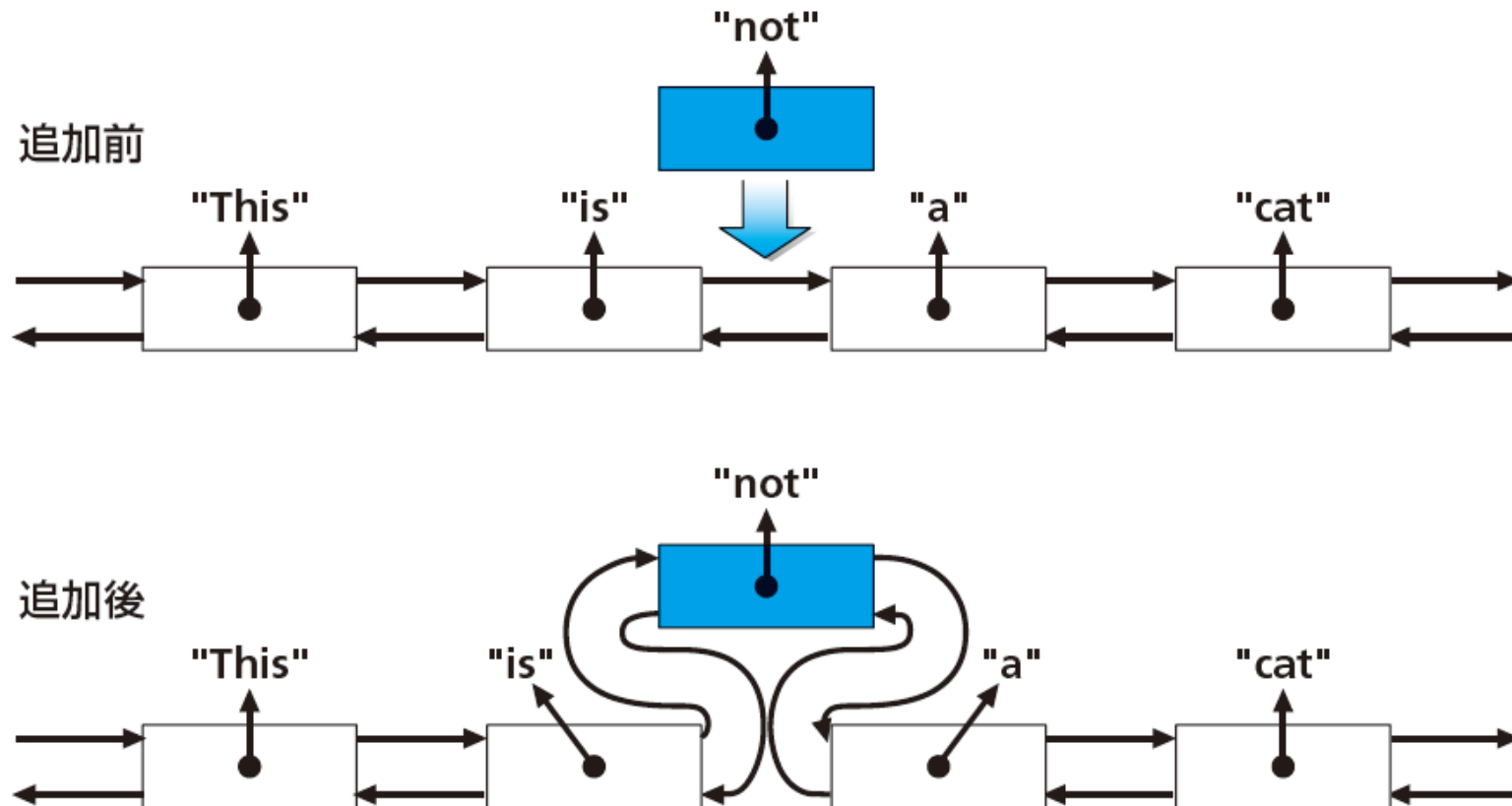
# リストコレクション：ArrayList

- 配列 + 便利な機能
- インデックスで直接要素にアクセスできるので高速
- 要素の追加と削除には内部で要素のコピーが行われるので低速



# リストコレクション：LinkedList

- 要素の追加と削除は高速
- インデックスを指定しての要素へのアクセスは低速





# マップコレクション：HashMap

---

キーと値に使用する型を型パラメータで指定する

```
HashMap<String, String> map =  
    new HashMap<String, String>();  
map.put("住所", "茨城県つくば市");  
map.put("氏名", "Java 太郎");  
  
System.out.println(map.get("住所"));  
System.out.println(map.get("氏名"));
```

# セットコレクション：HashSet

---

キーと値に使用する型を型パラメータで指定する

```
HashSet<String> set =  
    new HashSet<String>();  
map.add("Jan");  
map.add("Feb");  
  
System.out.println(set); // 要素を全て出力  
System.out.println(set.contains("Jan"));
```

## コレクションに含まれる全要素へのアクセス

---

これまでに学習したfor文を使うと、次のようにしてArrayListの全要素にアクセスできる

```
ArrayList<String> list = new ArrayList<String>();  
list.add("Good morning.");  
list.add("Hello.");  
for(int i = 0; i < list.size(); i++) {  
    System.out.println(list.get(i));  
}
```

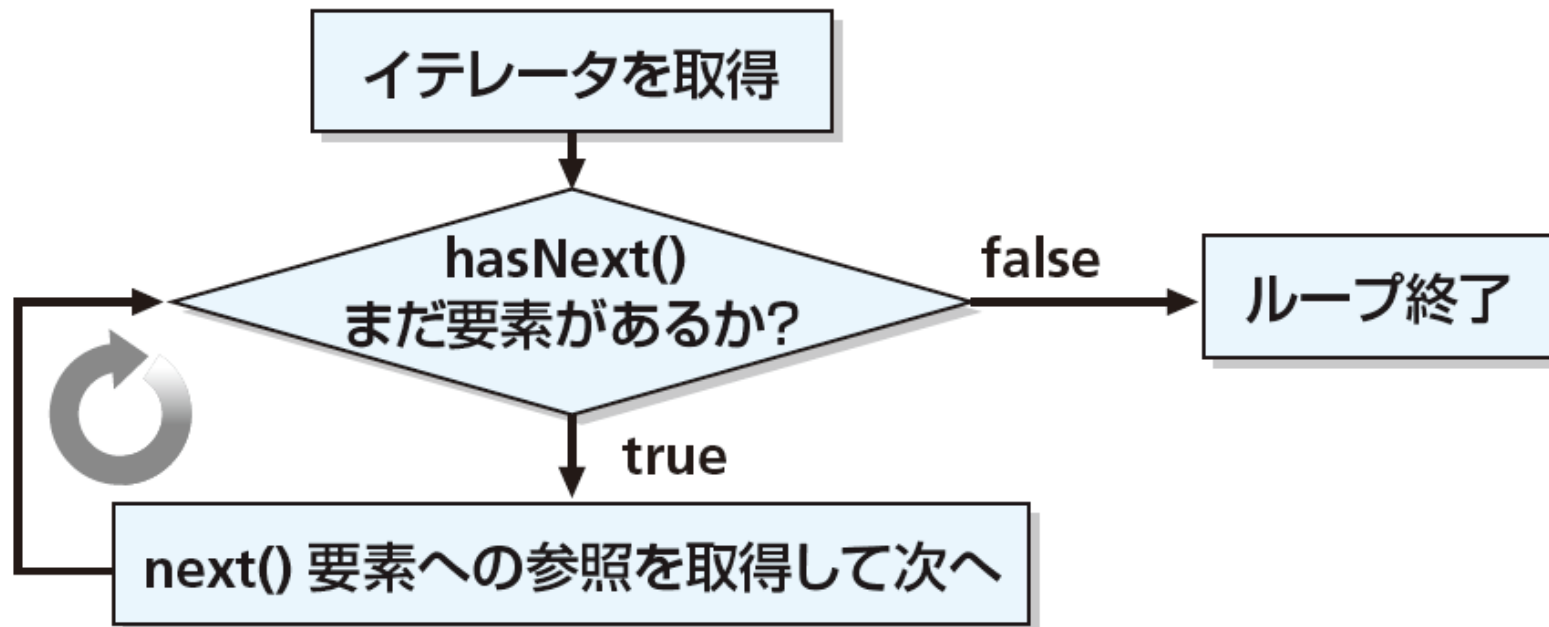
- この方法は、インデックスを指定して要素にアクセスできるものにし  
か使えない
- インデックスを指定したアクセスが低速なコレクションには適さない
- 他にも全要素にアクセスする方法がある

# イテレータ (Iterator)

- イテレータは、コレクションの中の要素を1つずつ順番に参照する能力をもつオブジェクト
- 主に次の2つのメソッドがある

`boolean hasNext()` まだ要素があるか

`E next()` 現在参照している要素を返して、次の要素に移動する



# イテレータ (Iterator) の使用例

---

```
HashSet<String> set = new HashSet<String>();  
set.add("A");  
set.add("B");  
set.add("C");  
set.add("D");  
Iterator<String> it = set.iterator();  
while(it.hasNext()) {  
    String str = it.next();  
    System.out.println(str);  
}
```

セットコレクションに対しても、1つずつ要素を参照できる。

# 拡張for文

---

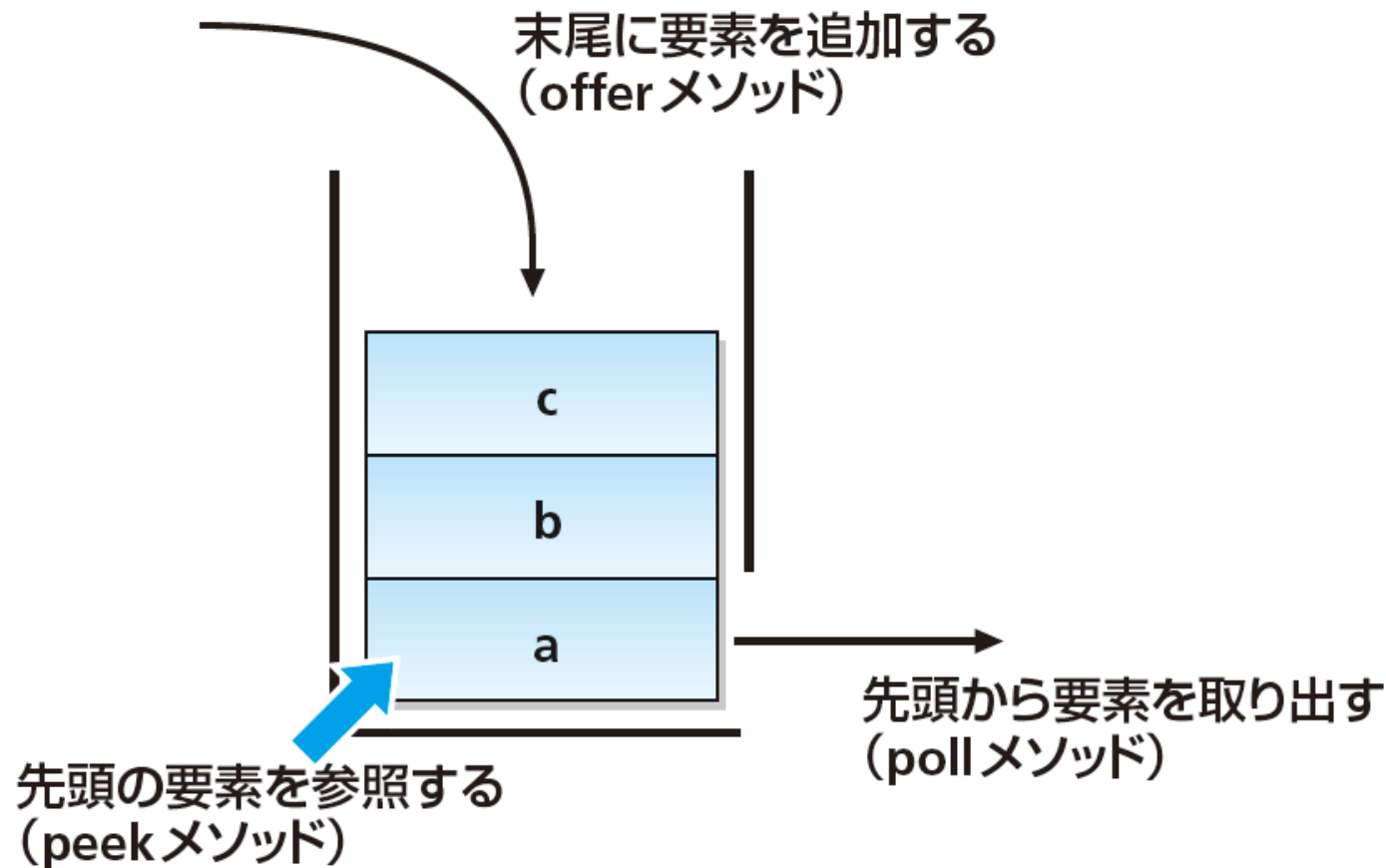
通常のfor文、イテレータを使ったアクセスよりも  
簡潔に記述できる構文

```
for(型名 変数名 : コレクション) {  
    forループ内の処理  
}
```

```
ArrayList<String> list = new ArrayList<String>();  
list.add("Good morning.");  
list.add("Hello.");  
for(String str : list) {  
    System.out.println(str);  
}
```

# LinkedListクラスによるキュー

キュー：先入れ先出し（First In First Out:FIFO）  
によるオブジェクト管理



# Queueインタフェースの使用

---

```
Queue<String> queue = new LinkedList<String>();

queue.offer("(1)");
System.out.println("キューの状態:" + queue);
queue.offer("(2)");
System.out.println("キューの状態:" + queue);
queue.offer("(3)");
System.out.println("キューの状態:" + queue);
queue.offer("(4)");
System.out.println("キューの状態:" + queue);

while(!queue.isEmpty()) {
    System.out.println("要素の取り出し:" + queue.poll());
    System.out.println("キューの状態" + queue);
}
```



# LinkedListクラスによるスタック

---

スタック：後入れ先出し（Last In First Out:LIFO）によるオブジェクト管理

# LinkedListのスタックとしての利用

---

```
LinkedList<String> stack = new LinkedList<String>();

stack.push("(1)");
System.out.println("スタックの状態:" + stack);
stack.push("(2)");
System.out.println("スタックの状態:" + stack);
stack.push("(3)");
System.out.println("スタックの状態:" + stack);
stack.push("(4)");
System.out.println("スタックの状態:" + stack);

while(!stack.isEmpty()) {
    System.out.println("要素の取り出し:"+stack.pop());
    System.out.println("スタックの状態" + stack);
}
```

# sortメソッドによる並び替え

---

- コレクションに格納された要素を値の大小で並び替えたいことがある
- Collectionsクラスのsortメソッドで、Listインタフェースを実装したコレクション（ArrayList, LinkedListなど）の要素の並び替えを行える

```
ArrayList<String> list = new ArrayList<String>();  
// listに要素を追加する処理  
collections.sort(list); // 並び替え実行
```

# 自作クラスのインスタンスの並び替え

---

自分で作ったクラスを順番に並び替える場合は、  
大小をどのように決定するのか明らかにしておく  
必要がある。



Comparable インタフェースを実装する



```
public int comapareTo(クラス名 変数名)
```

メソッドを実装する

# Comparableインタフェースの実装例

---

```
class Point implements Comparable<Point> {  
    int x;  
    int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int compareTo(Point p) {  
        return (this.x + this.y) - (p.x + p.y);  
    }  
}
```

```
ArrayList<Point> list = new ArrayList<Point>();  
// listに要素を追加する処理  
Collections.sort(list); // 並び替え実行
```

# 第6章 ラムダ式

# 内部クラス

---

クラスの宣言を別のクラスの内部で行える

```
class Outer {  
    class Inner {  
    }  
}
```

内部クラスは外部クラスのプライベート宣言された変数にもアクセスできる

# 匿名クラス

---

- メソッドの引数を指定するカッコの中で、「クラスを定義すること」と「そのインスタンスを生成すること」の両方を行える → クラスの名前が不要
- その場だけで必要なクラスの作成に使える



# 匿名クラスの例

---

```
interface SayHello {
    public void hello();
}
class Greeting {
    static void greet(SayHello s) {
        s.hello();
    }
}
public class AnonymousClassExample {
    public static void main(String[] args) {
        Greeting.greet(new SayHello() {
            public void hello() {
                System.out.println("こんにちは");
            }
        });
    }
}
```

# 関数型インタフェース

---

抽象メソッドを1つしか含まないインタフェースのこと

```
interface SayHello {  
    public void hello();  
}
```

関数型インタフェースを引数の型に持つメソッド

```
class Greeting {  
    static void greet(SayHello s) {  
        s.hello();  
    }  
}
```

# ラムダ式

---

関数型インタフェースを実装したクラスの宣言を短く記述するための構文。

```
Greeting.greet(new SayHello() {  
    public void hello() {  
        System.out.println("こんにちは");  
    }  
});
```



```
Greeting.greet( () -> {System.out.println("こんにちは");} );
```

# ラムダ式の記述方法

---

(引数列) -> {処理内容}

引数がない例

```
() -> {System.out.println("こんにちは");}
```

引数が1つの例

```
(int n) -> { return n + 1;}
```

引数が2つの例

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

# ラムダ式の省略形

---

引数列の型は省略できる

元のラムダ式： <code>(int n) -&gt; { return n + 1; }</code> 省略形： <code>(n) -&gt; { return n + 1; }</code>
---

引数が一つだけの場合、引数を囲む( )を省略できる

元のラムダ式： <code>(n) -&gt; { return n + 1; }</code> 省略形： <code>n -&gt; { return n + 1; }</code>
---

# ラムダ式の省略形

---

処理の中身に命令文が1つしかない場合、処理を囲む{ }とreturnキーワード、セミコロン (;) を省略できる。

元のラムダ式： <code>n -&gt; { return n + 1; }</code> 省略形： <code>n -&gt; n + 1;</code>
--

# ラムダ式の省略形を使った例

---

```
interface SimpleInterface{
    public int doSomething(int n);
}

public class LambdaExample {
    static void printout(SimpleInterface i) {
        System.out.println(i.doSomething(2));
    }

    public static void main(String[] args) {
        printout(n -> n + 1);
    }
}
```

# forEachメソッドとラムダ式

---

- ArrayListの要素にアクセスするために拡張for文を用いる以外にもforEachメソッドを使用できる
- ラムダ式と組み合わせることで各要素に対する処理を簡潔に記述できる

```
for(Point p : pointList) {  
    p.x *= 2;  
    p.y *= 2;  
}  
  
for(Point p : pointList) {  
    p.printInfo()  
}
```



```
pointList.forEach( p -> {p.x *= 2; p.y *= 2;} );  
pointList.forEach( p -> p.printInfo() );
```



# ラムダ式を用いた並べ替え

---

- 各コレクションクラスに備わっているsortメソッドを使用すると、Comparableインタフェースを実装していないオブジェクトも並べ替えできる
- sortメソッドの引数に「どのように順序付けするか」を記述したラムダ式を渡す

(Point p0, Point p1) -> {2つのPointオブジェクトに対して順序付けを行うための処理}

# ラムダ式を用いた並べ替えの例

---

Pointクラスのインスタンス変数xとyの値を足し合わせた値で順序付けし、昇順に並べる

```
pointList.sort((p0, p1) -> (p0.x + p0.y) - (p1.x + p1.y));
```

# 第7章 入出力

# 入出力

---

**入力**：プログラムにデータが入ってくること

- ファイルからのデータの読み込み
- キーボードで入力されたデータの読み込み
- ネットワーク通信によるデータの読み込み

**出力**：プログラムからデータを送りだすこと

- 画面への表示
- ファイルへの保存
- ネットワーク通信による送信
- プリントアウト

- **ストリームオブジェクト**がデータの橋渡しをする
- 扱うデータは「**文字列データ**（Unicode, 16ビット単位）」と「**バイナリデータ**（8ビット単位）」に分けられる

# 標準出力

---

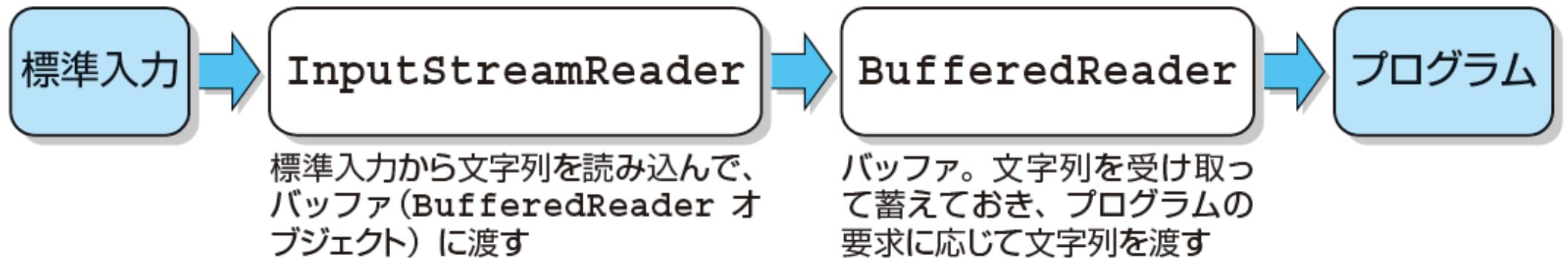
入出力先を特に指定しなかった場合に標準的に使用される出力。一般的には「**コンソール**」。

標準出力への文字列の出力

```
System.out.println("こんにちは");
```

# 標準入力

標準入力からの文字列の受け取り



```
InputStreamReader in =  
    new InputStreamReader(System.in);  
BufferedReader reader = new BufferedReader(in);  
try {  
    String str = reader.readLine();  
} catch(IOException e) {}
```

# 文字列と数値の変換

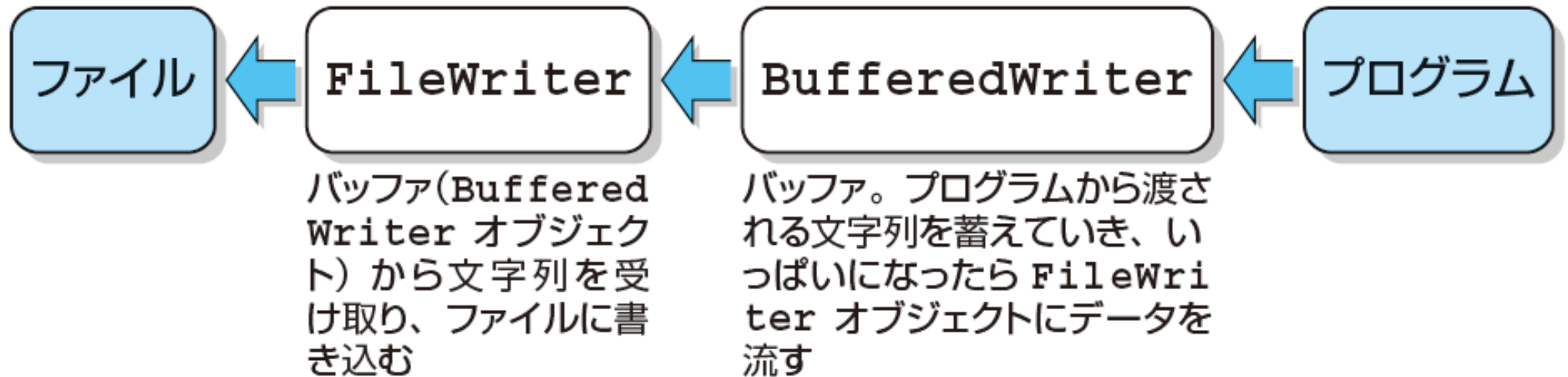
---

- 標準入力から渡されるデータは文字列
- 「10」という文字列を数値として処理できるようにするには、次のような変換が必要

```
String str0 = "10";  
int i = Integer.parseInt(str0);
```

```
String str1 = "0.5";  
double d = Double.parseDouble(str1);
```

# ファイルへの出力

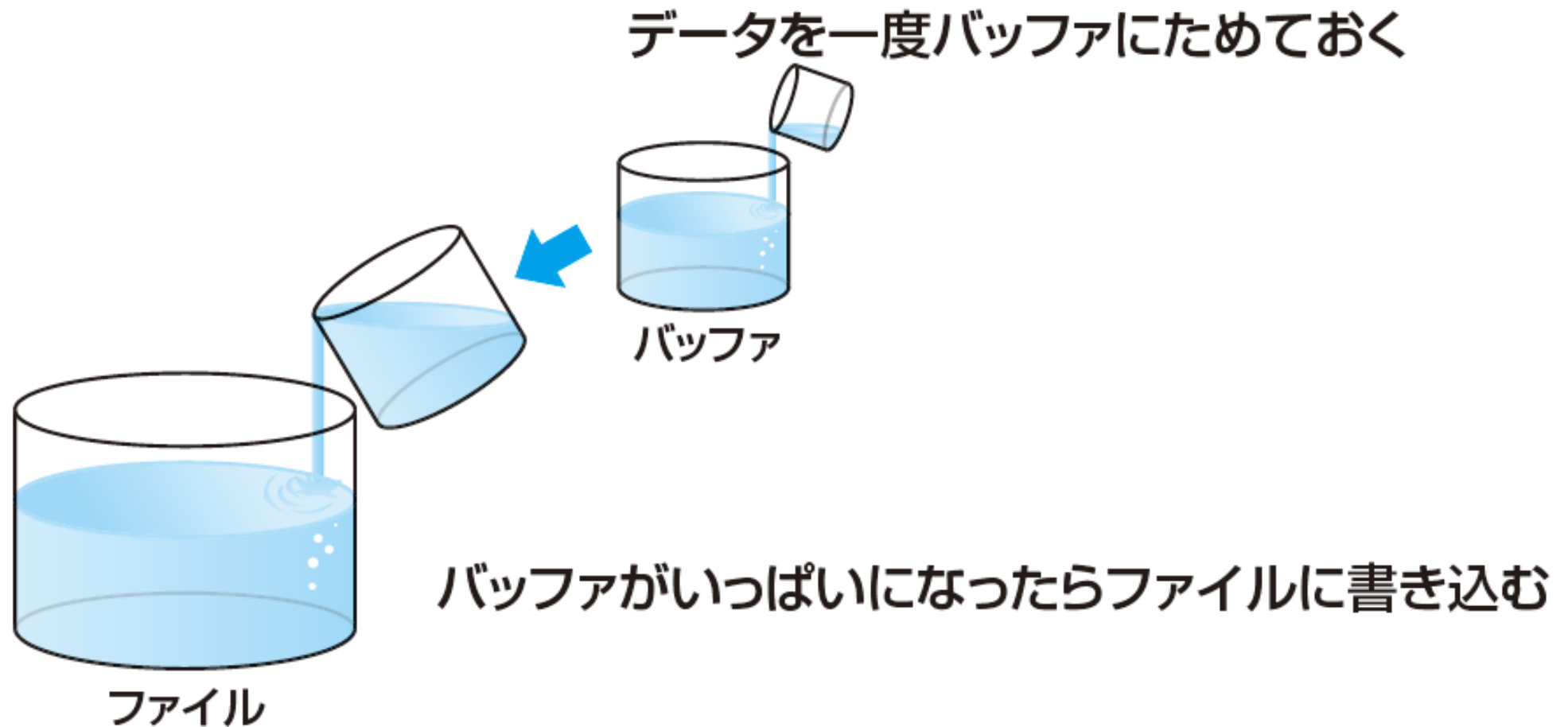


```
try {  
    File file = new File("C:¥¥java¥¥test.txt");  
    FileWriter fw = new FileWriter(file);  
    BufferedWriter bw = new BufferedWriter(fw);  
    for(int i = 0; i < 5; i++) {  
        bw.write "[" + i + "]"¥r¥n);  
    }  
    bw.close();  
} catch (IOException e) {  
    System.out.println(e);  
}
```



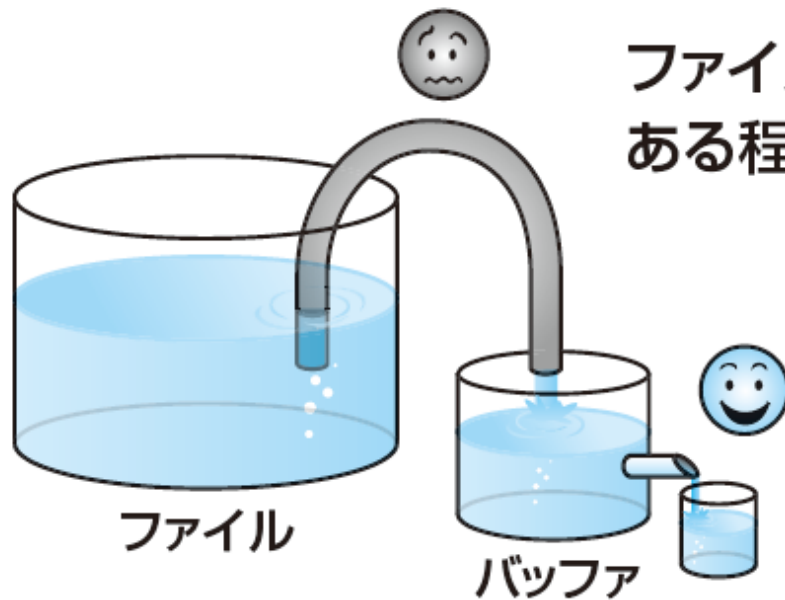
# バッファを用いたデータの書き込み

---



# バッファを用いたデータの読み込み

---

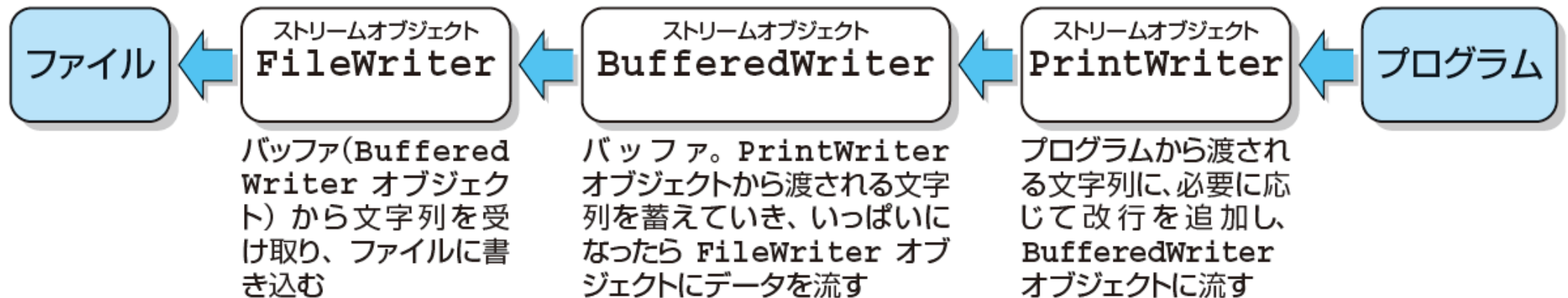
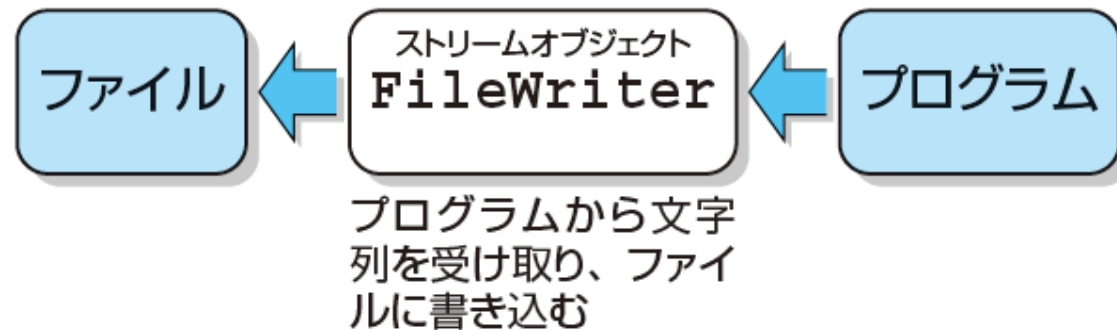


ファイルからデータを受け取るのは手間がかかるので、  
ある程度まとめて受け取って、バッファに蓄えておく

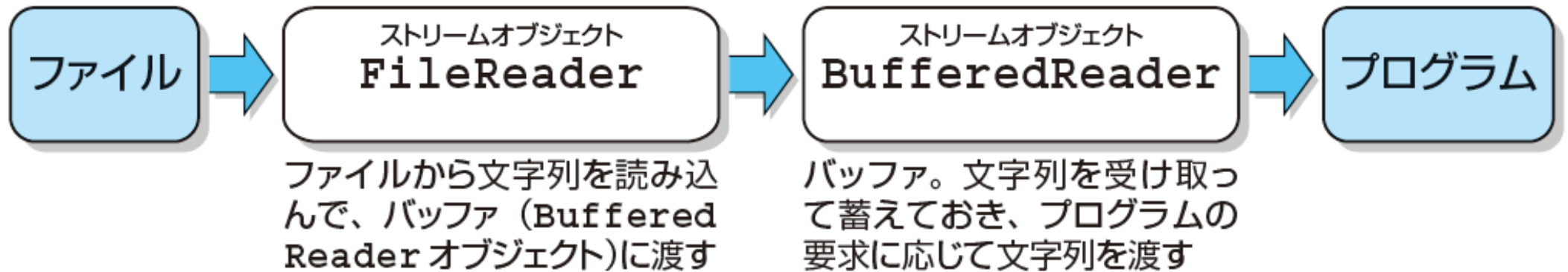
バッファからデータを受け取るのは簡単なので、  
必要に応じてバッファからデータを読み取る

# ストリームの連結

- ストリームは最低1つ必要
- 目的に応じてストリームを複数連結できる



# ファイルからの入力



```
try {  
    File file = new File("C:¥¥java¥¥test.txt");  
    FileReader fr = new FileReader(file);  
    BufferedReader br = new BufferedReader(fr);  
    String s;  
    while((s = br.readLine()) != null) {  
        System.out.println(s + "を読み込みました");  
    }  
    br.close();  
} catch (IOException e) {  
    System.out.println(e);  
}
```

# シリアライゼーションとオブジェクトの保存

---

プログラムの状態をファイルに保存する

(一度プログラムを終了させて、後から続きを行うときなどに必要)

方法1. テキストファイルに文字列で情報を書き込む

方法2. シリアライゼーションによってオブジェクト  
そのものをファイルに保存する

# シリアライゼーション

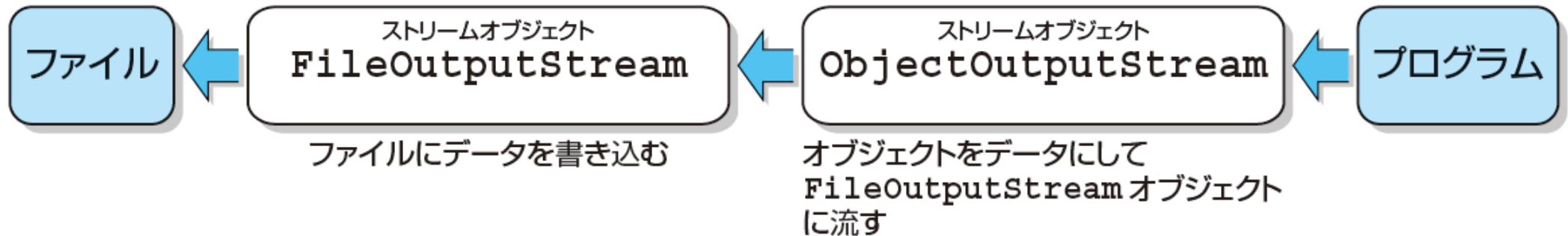
---

- オブジェクトをその状態を保持したままファイルに書き出すことができる
- このことをシリアライゼーションと言う
- ファイルに書き出せるオブジェクトは、  
Serializableインタフェースを実装している必要がある。実装すべきメソッドは何もないので、宣言するだけ

```
class Point implements Serializable {  
    int x;  
    int y;  
}
```

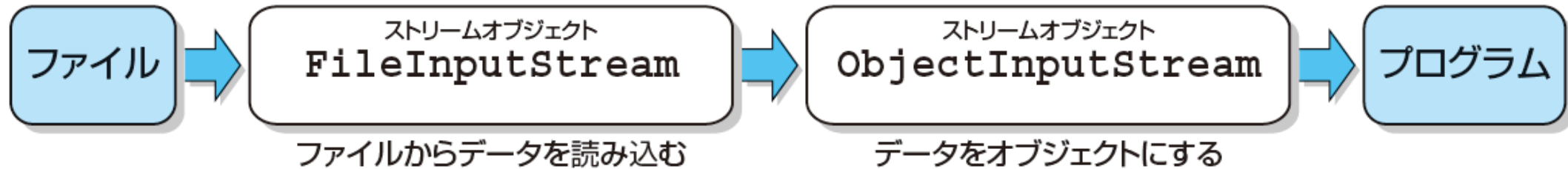
# シリアライゼーションを使用したオブジェクトの保存

---



```
try {
    FileOutputStream fs =
        new FileOutputStream("C:¥¥java¥¥triangle.ser");
    ObjectOutputStream os = new ObjectOutputStream(fs);
    os.writeObject(/* 出力するオブジェクトへの参照 */);
    os.close();
} catch(IOException e) {
    System.out.println(e);
}
```

# 保存したオブジェクトの再現



```
try {
    FileInputStream fs = new
        FileInputStream("C:¥¥java¥¥triangle.ser");
    ObjectInputStream os = new ObjectInputStream(fs);
    MyObject obj = (MyObject)os.readObject();
    os.close();
} catch(IOException e) {
    System.out.println(e);
} catch (ClassNotFoundException e) {
    System.out.println(e);
}
```



# ファイルとフォルダの操作

---

- java.io.Fileクラスを使って、ファイルとフォルダの操作を行える。
- Fileオブジェクトの作成

```
File file = new File("C:¥¥java¥¥test.txt");
```

Fileクラスのファイル操作のための主なメソッド

<code>boolean exists()</code>	ファイルが存在するかどうか確認する
<code>boolean delete()</code>	ファイルを削除する
<code>boolean renameTo(File dest)</code>	名前を変更する

# フォルダ操作

---

- フォルダもファイルと同じように `java.io.File` クラスを使用する
- `File` オブジェクトの作成

```
File file = new File("C:¥¥java");
```

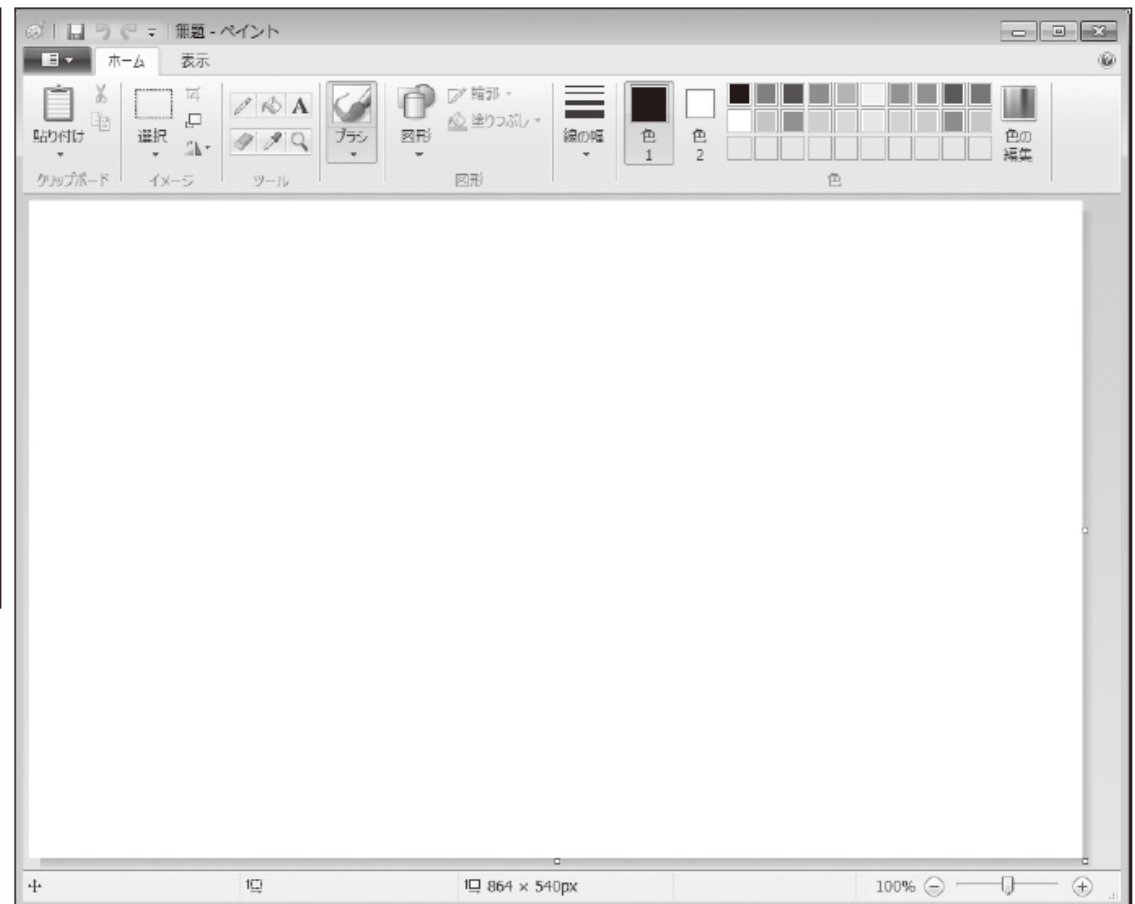
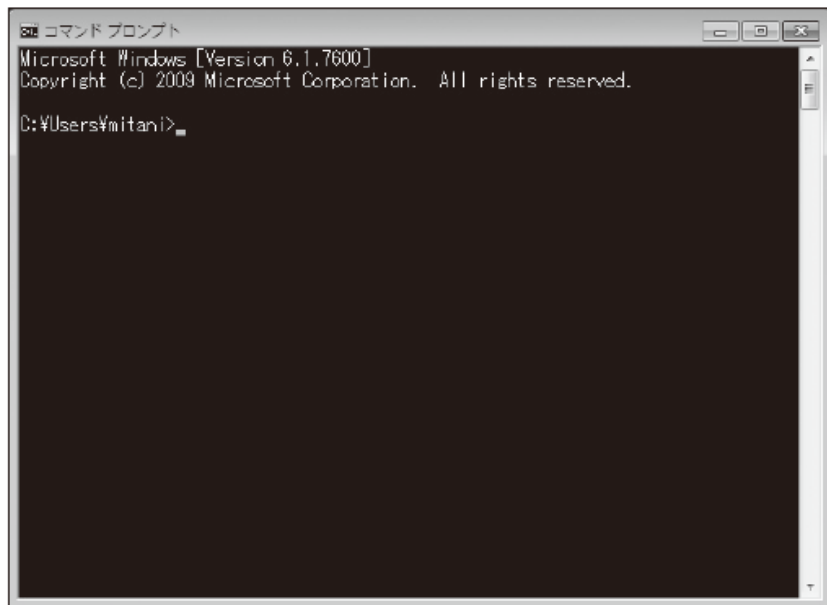
`File` クラスのフォルダ操作のための主なメソッド

<code>String[] list()</code>	フォルダに含まれるファイルの一覧を返す
<code>boolean mkdir()</code>	フォルダを作成する
<code>boolean mkdirs()</code>	階層を持ったフォルダを一度に作成する
<code>boolean delete()</code>	フォルダを削除する

# 第8章 GUIアプリケーション

# GUIアプリケーションとは

- GUI: Graphical User Interface
- CUI: Character-based User Interface



# Swingライブラリ

---

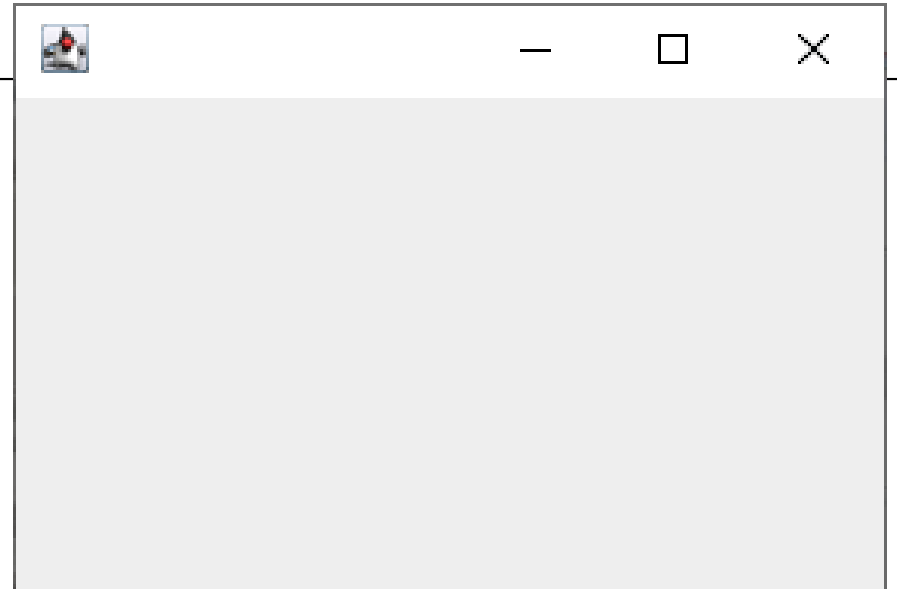
- ウィンドウ、ボタン、メニューなどを扱うクラス群を集めたライブラリ
- javax.swingパッケージに含まれる
- GUIアプリケーションを扱うには、ユーザの操作に応答するためのイベント処理が必要

# フレームの作成

---

```
import javax.swing.*;

class SimpleFrameExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```



# 自分自身のインスタンスを生成するクラス

---

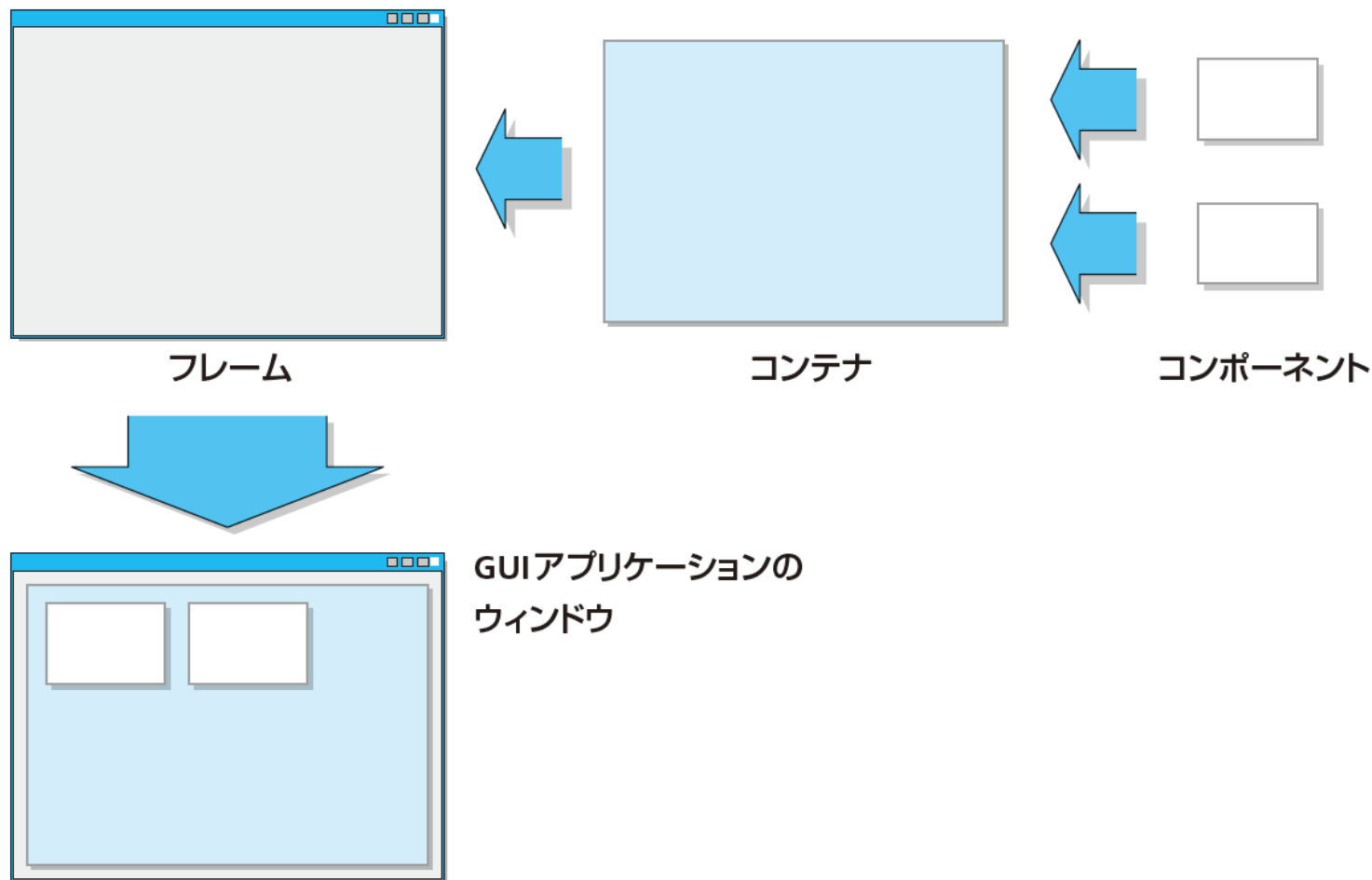
```
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) {
        new MyFrame();
    }

    MyFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);
        setVisible(true);
    }
}
```

# コンポーネントの配置

- ボタンやチェックボックスなどのGUI部品は「**コンポーネント**」と呼ばれる
- コンポーネントは、**コンテナ**の上に配置される。





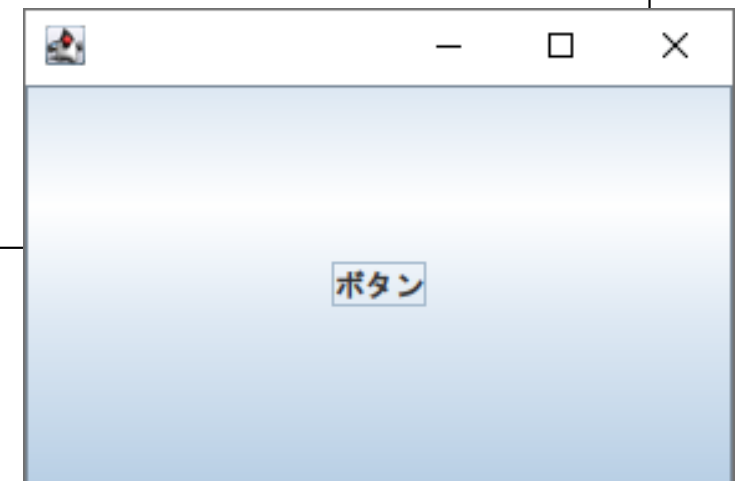
# JButton (ボタン) コンポーネントの配置

---

```
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) {
        new MyFrame();
    }

    MyFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().add(new JButton("ボタン"));
        setSize(300, 200);
        setVisible(true);
    }
}
```



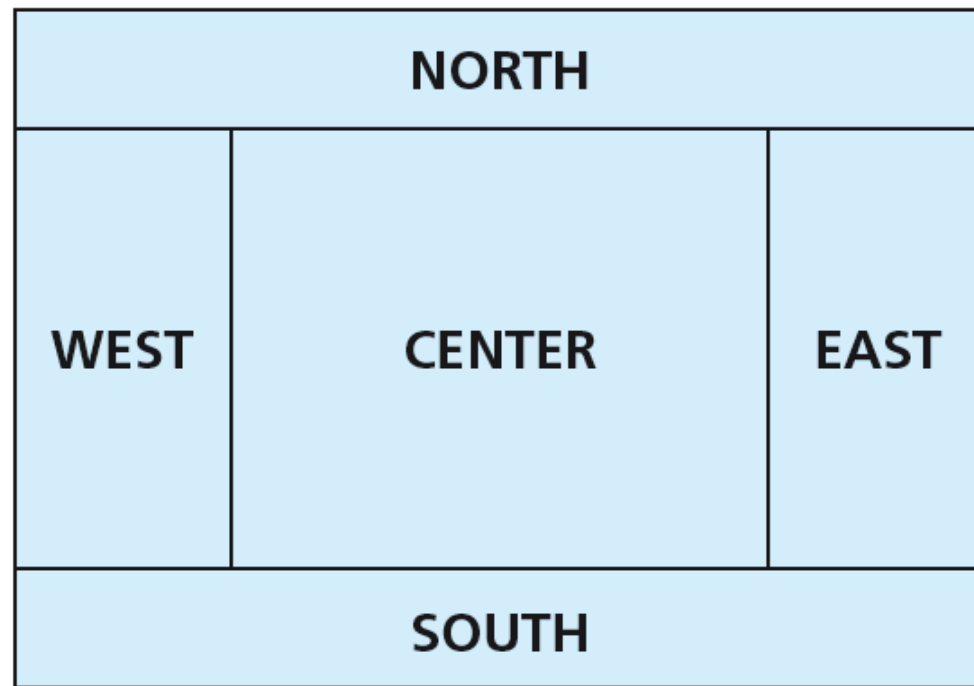
ボタンを配置しただけなので、クリックしても何も起こらない

# ボーダーレイアウト

---

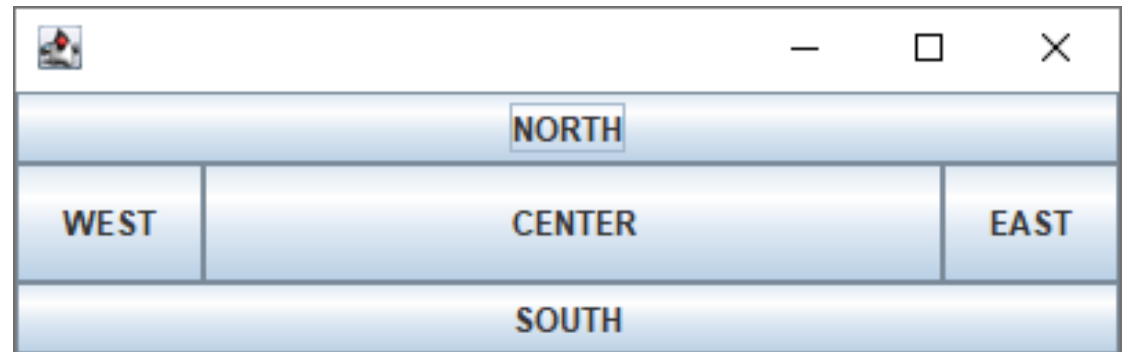
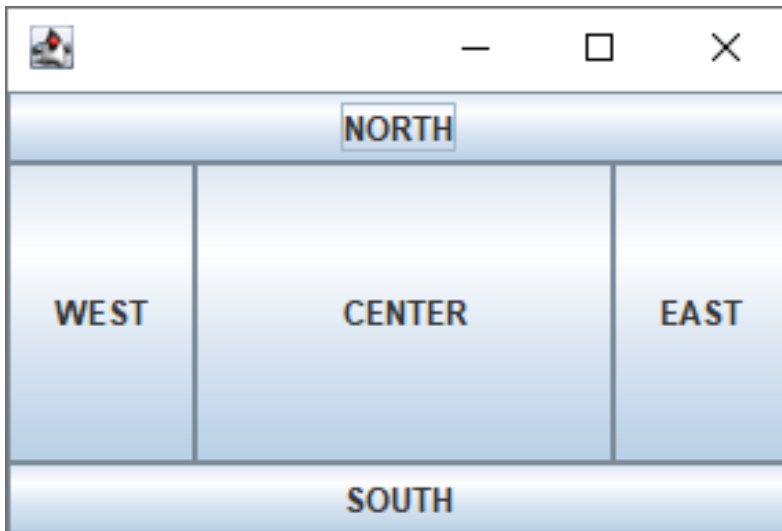
- JFrameクラスのgetContentPaneメソッドで取得できるコンテナはBorderLayoutと呼ばれる方法でコンポーネントが配置される
- 配置場所を5つから選択して指定できる

```
add(BorderLayout.WEST, new JButton("ボタン"));
```



# ボーダーレイアウトの配置例

```
getContentPane().add(BorderLayout.CENTER, new JButton("CENTER"));  
getContentPane().add(BorderLayout.SOUTH, new JButton("SOUTH"));  
getContentPane().add(BorderLayout.WEST, new JButton("WEST"));  
getContentPane().add(BorderLayout.EAST, new JButton("EAST"));  
getContentPane().add(BorderLayout.NORTH, new JButton("NORTH"));  
setSize(300, 200);  
setVisible(true);
```

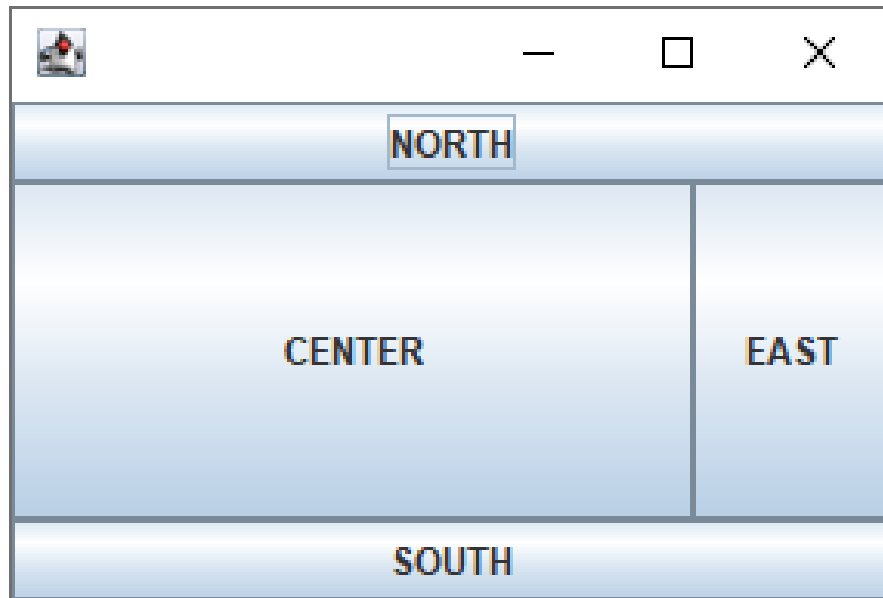


ウィンドウサイズを変えると、ボタンのサイズが自動調整される

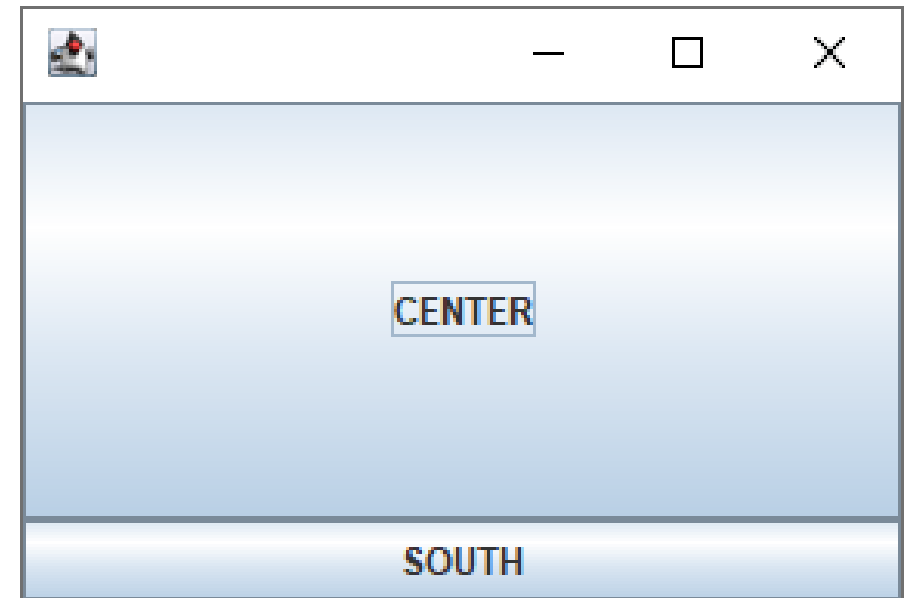
# ボーダーレイアウトの配置例

---

コンポーネントの数に応じて、自動的に大きさが調整される



WESTに配置しなかった場合



CENTER,SOUTHのみ  
配置した場合

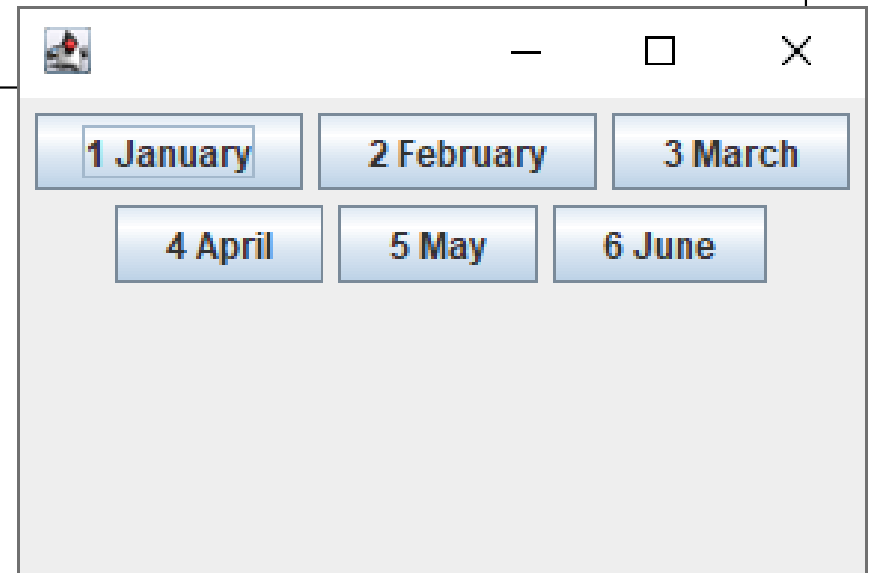
# レイアウトマネージャ

---

- レイアウトマネージャがコンポーネントをどこに、どの大きさに配置するかを決定する。
- 各種のレイアウトマネージャが準備されている。それぞれ、レイアウト方法が異なる。
  - BorderLayout
  - FlowLayout
  - BoxLayout
  - GridLayout
- レイアウトマネージャを変更するには、コンテナのsetLayoutメソッドを使用する

# フローレイアウト

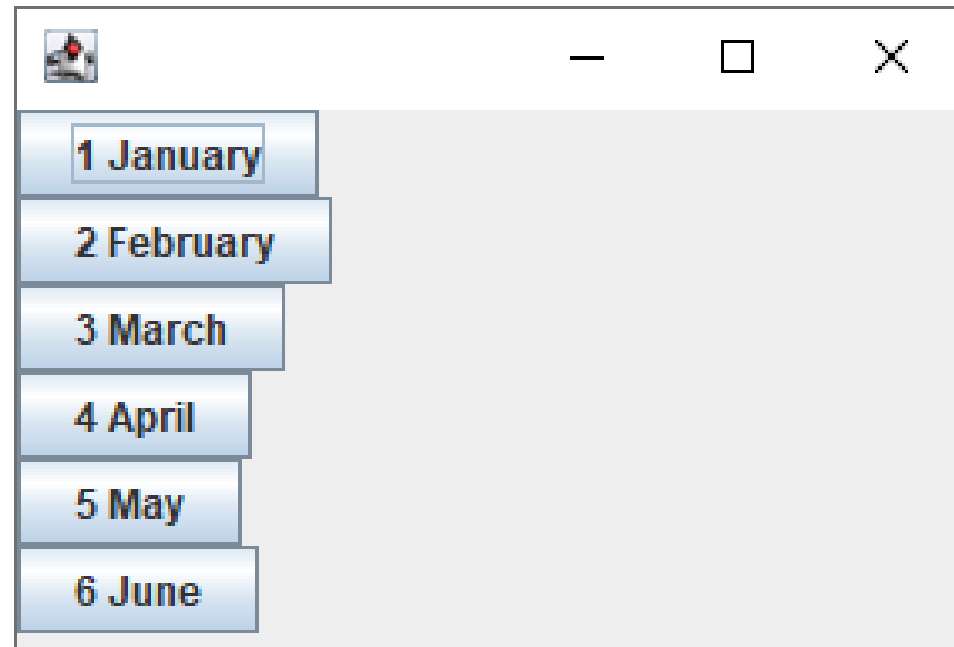
```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
getContentPane().setLayout(new FlowLayout());  
getContentPane().add(new JButton("1 January"));  
getContentPane().add(new JButton("2 February"));  
getContentPane().add(new JButton("3 March"));  
getContentPane().add(new JButton("4 April"));  
getContentPane().add(new JButton("5 May"));  
getContentPane().add(new JButton("6 June"));  
setSize(300, 200);  
setVisible(true);
```



# ボックスレイアウト

---

```
getContentPane().setLayout(  
    new BorderLayout(getContentPane(), BorderLayout.Y_AXIS);
```



# グリッドレイアウト

---

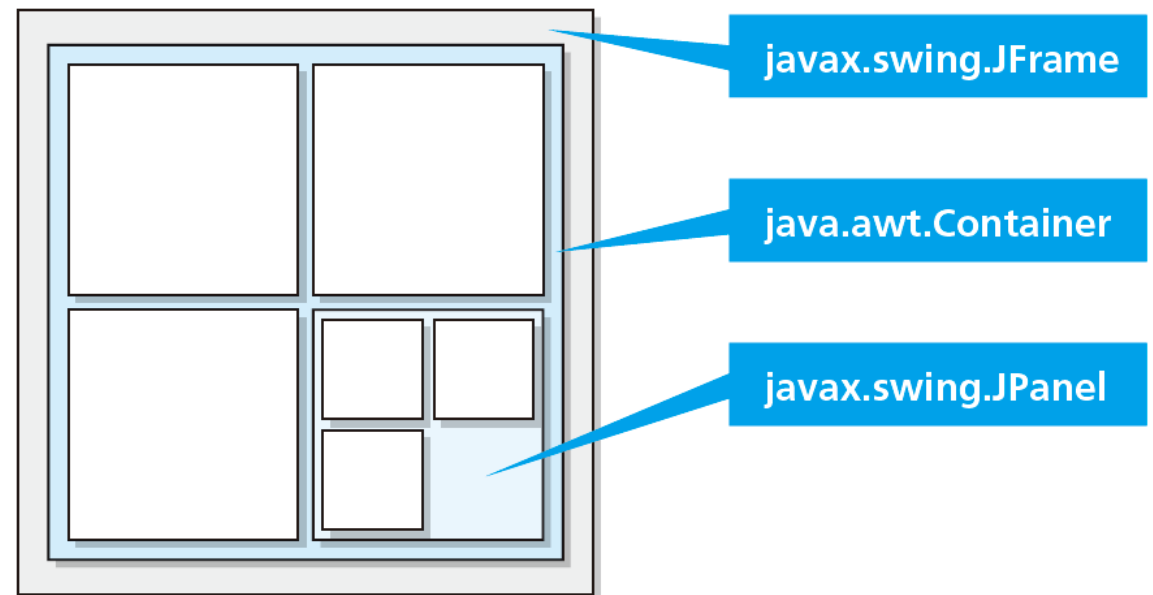
```
getContentPane().setLayout(  
    new GridLayout(2, 3));
```





# パネルを活用したレイアウト

- JPanelはコンポーネントをのせることができるコンポーネント
- JPanelを活用することで、複雑なコンポーネントの配置が可能になる



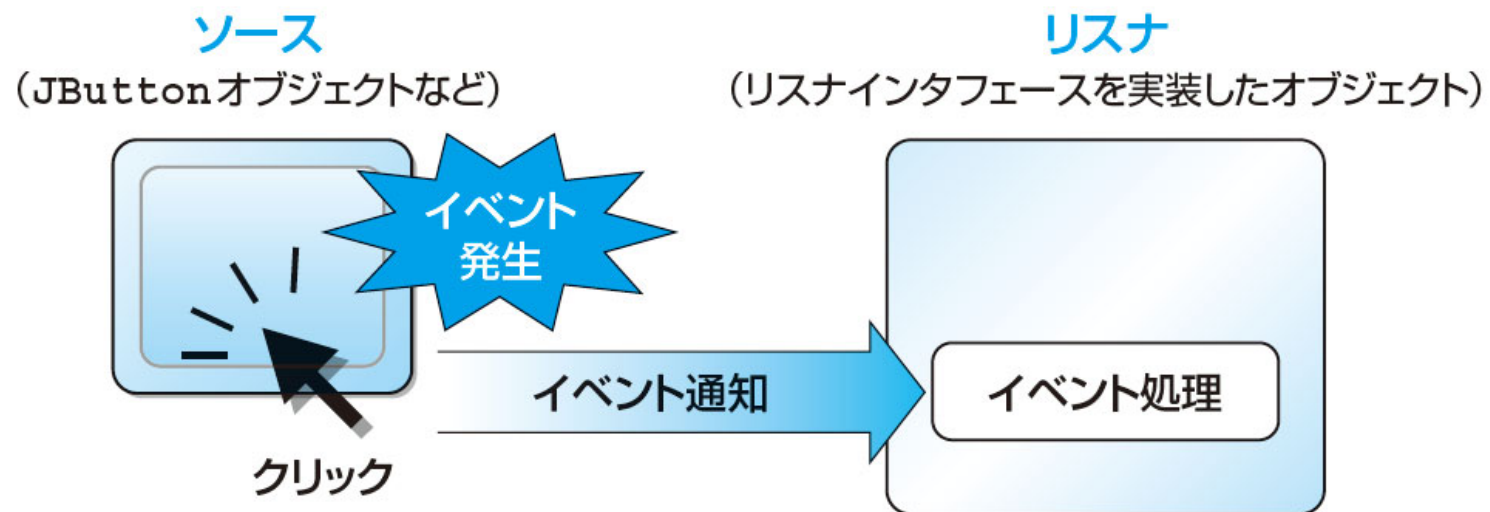
# イベント処理

---

- 「ボタンがクリックされた」などのように、ユーザによって何か操作が行われたことを「**イベントが発生した**」という
- Javaでは、イベントオブジェクト (java.awt.Event)によって、イベントが表現される
- イベントが発生したときの処理を**イベント処理**という

# ソースとリスナ

- ソース (source)
  - イベントを発生するオブジェクトのこと
  - JButtonなどのコンポーネントに限られる
- リスナ (listener)
  - イベントを受け取るオブジェクトのこと
  - 必要なインタフェース（リスナインタフェース）を実装しているクラスがリスナになれる



# イベント処理（ボタン押下時の処理）を行うプログラムの作成

---

## 1. リスナを作成する

- イベント通知を受け取るクラスを作成する
- **ActionListener**インタフェースを実装する
- イベントが発生すると、actionPerformedメソッドが呼び出される

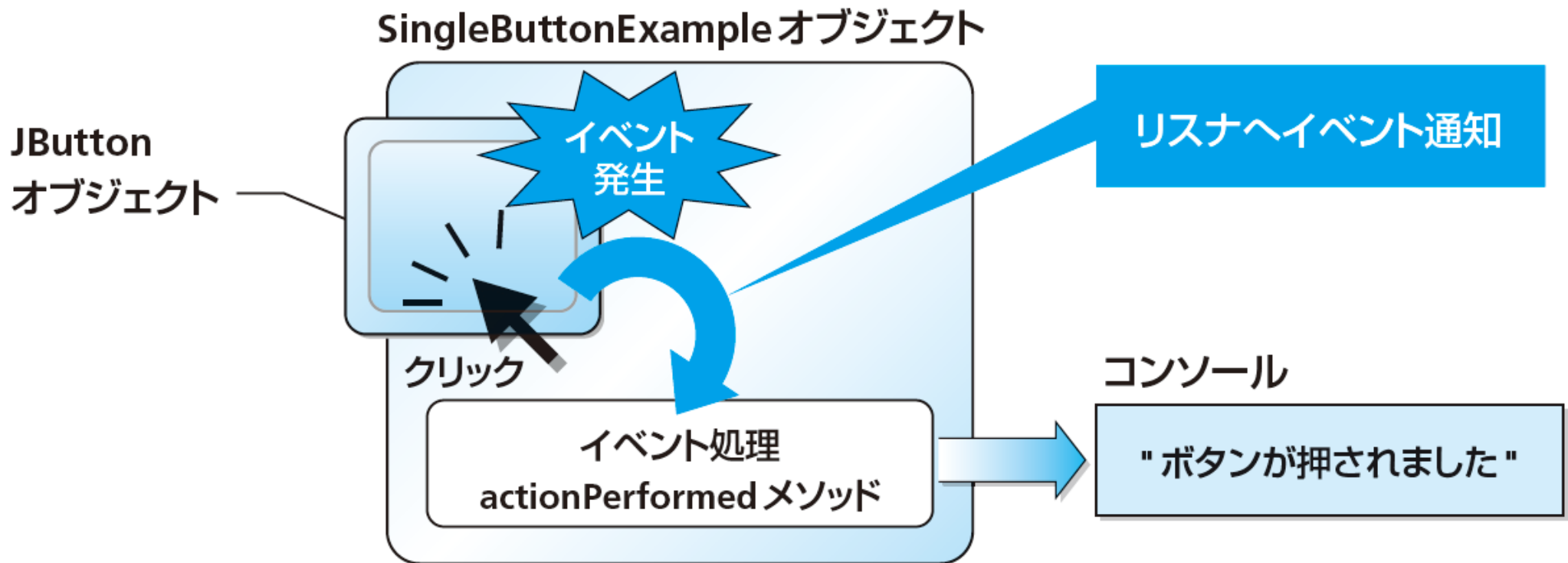
## 2. ソースにリスナを登録する

- イベントが発生した時に、その通知を誰に送るのかをソースに登録する←**リスナ登録**。
- リスナ登録はaddActionListenerメソッドで行う

```
1: import javax.swing.*;
2: import java.awt.event.*; ← イベント処理に関するクラスが含まれます
3:
4: public class SingleButtonExample extends JFrame implements ActionListener →
   ActionListener {
5:     public static void main(String[] args) {
6:         new SingleButtonExample();
7:     }
8:
9:     SingleButtonExample() {
10:        JButton button = new JButton("ボタン"); ← ボタンオブジェクトを生成します
11:        button.addActionListener(this); ← ボタンに対して自分自身をリスナ登録します
12:        getContentPane().add(button);
13:        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:        setSize(200, 100);
15:        setVisible(true);
16:    }
17:
18:    public void actionPerformed(ActionEvent ae) {
19:        System.out.println("ボタンが押されました"); ← ボタンが押されたときの処理です
20:    }
21: }
```

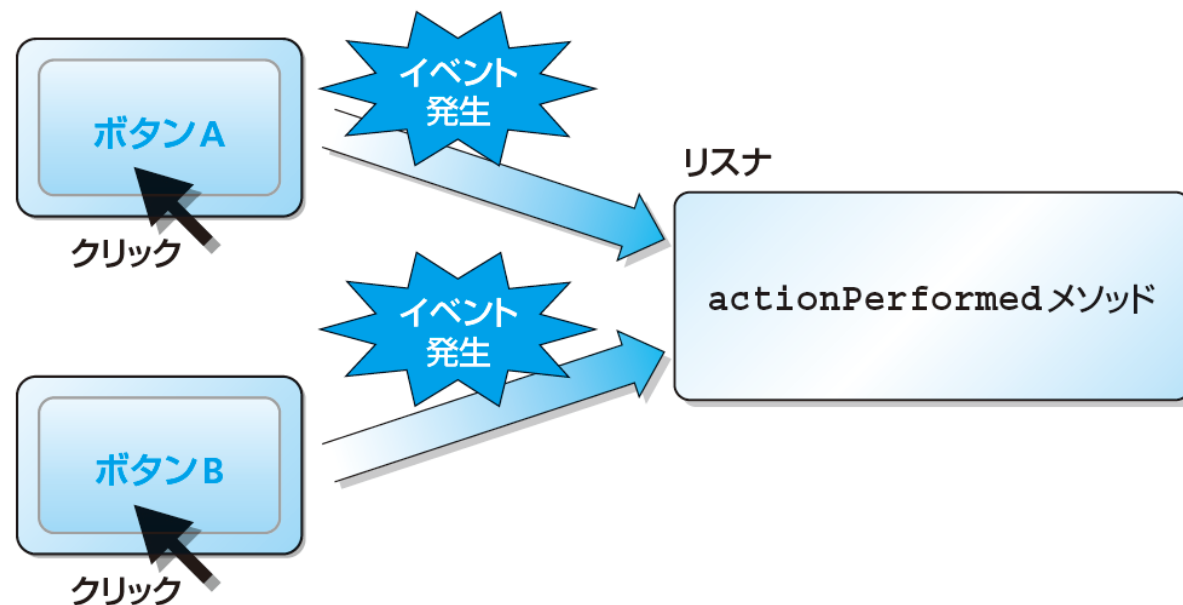
# イベント処理の様子

- リスナ登録したオブジェクト  
(SimpleButtonExampleオブジェクト) へイベントが通知される (actionPerformedメソッドが呼び出される)



# 複数のコンポーネントがある場合

- 異なるボタンに対して、同じオブジェクトをリスナ登録している場合は、同じように `actionPerformed` メソッドが呼ばれる
- 引数で渡される `ActionEvent` オブジェクトの `getSource` メソッドでイベント通知元のオブジェクトを判別できる





```

1: import java.awt.*;
2: import javax.swing.*;
3: import java.awt.event.*;
4:
5: public class MultiButtonsExample extends JFrame implements →
   ActionListener {
6:     public static void main(String[] args) {
7:         new MultiButtonsExample();
8:     }
9:
10:    JButton button1;
11:    JButton button2;
12:
13:    MultiButtonsExample() {
14:        button1 = new JButton("ボタン1");
15:        button1.addActionListener(this);
16:        getContentPane().add(BorderLayout.WEST, button1);
17:
18:        button2 = new JButton("ボタン2");
19:        button2.addActionListener(this);
20:        getContentPane().add(BorderLayout.EAST, button2);
21:
22:        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23:        setSize(200, 100);
24:        setVisible(true);
25:    }
26:
27:    public void actionPerformed(ActionEvent ae) {
28:        if (ae.getSource() == button1) {
29:            System.out.println("ボタン1が押されました");
30:        } else if (ae.getSource() == button2) {
31:            System.out.println("ボタン2が押されました");
32:        }
33:    }
34: }

```

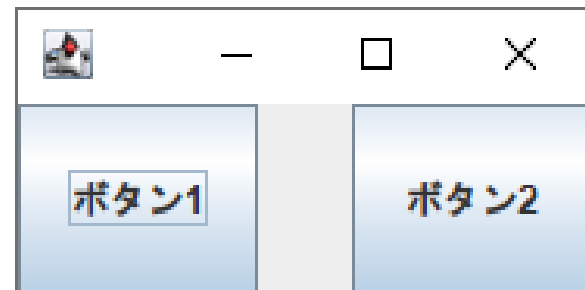
↑  
ActionListener  
インタフェースを  
実装します

} actionPerformedメソッドの中で参照できるように、  
コンストラクタの中ではなくフィールドで宣言します

} ボタンを作成し、リスナ  
を登録します

} ボタンを作成し、リスナ  
を登録します

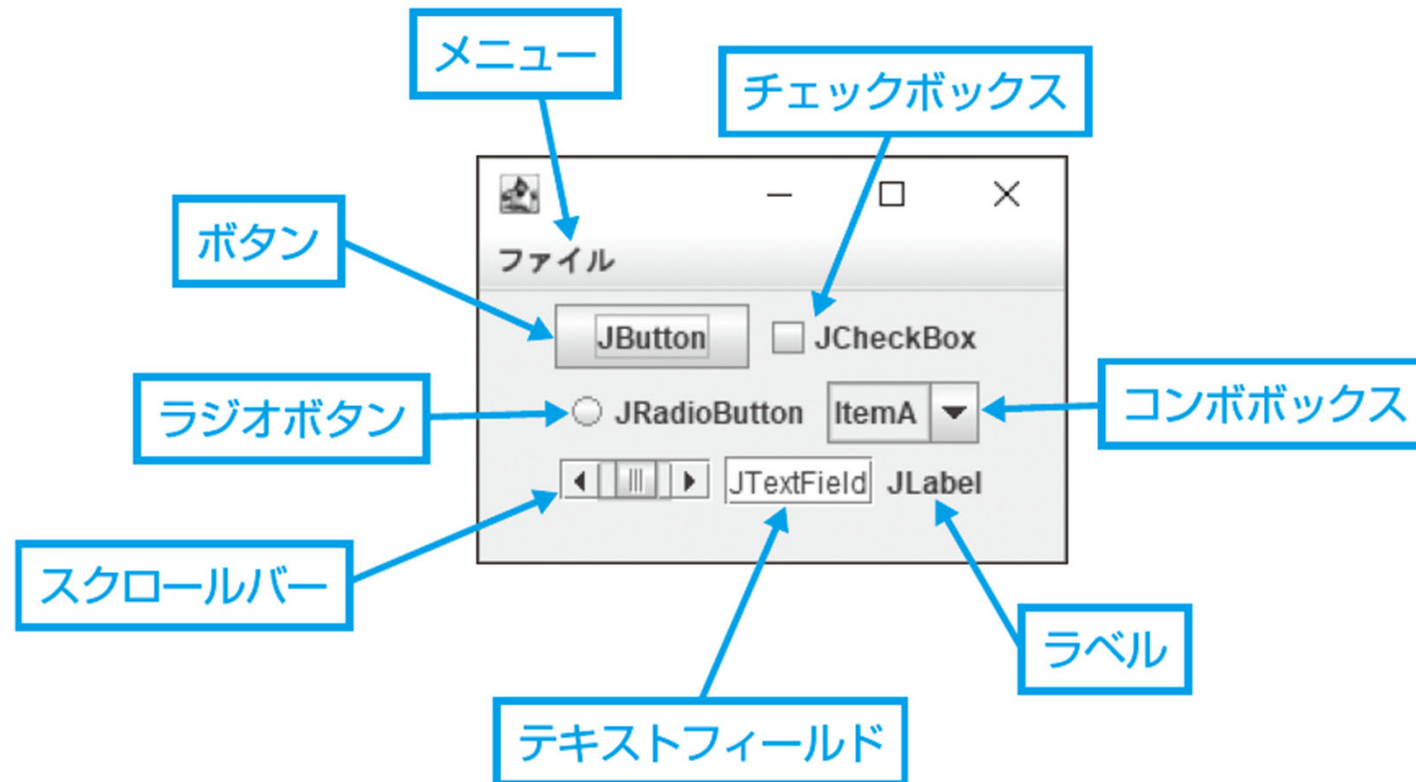
← イベントソースと  
ボタンオブジェク  
トを比較します





# さまざまなコンポーネント

- Swingライブラリには、さまざまなコンポーネントがある
- 必要に応じてAPI仕様書を調べてみよう



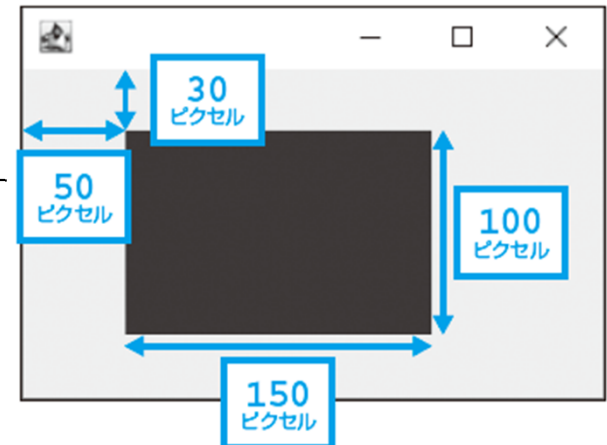
## 第9章

# グラフィックスとマウスイベント

# JPanelを使ったグラフィックス描画

- ・ 描画を行うためのパネルクラスの作成 (JPanelを継承)
- ・ paintComponentメソッドをオーバーライドする
- ・ 引数のGraphicsオブジェクトを使って描画

```
import java.awt.*;  
import javax.swing.*;  
  
public class MyPanel extends JPanel {  
    public void paintComponent(Graphics g) {  
        g.fillRect(50, 30, 150, 100);  
    }  
}
```



- ・ 作成したパネルクラスを配置する

```
getContentPane().add(new MyPanel());
```

# Graphiscオブジェクトの座標系

---

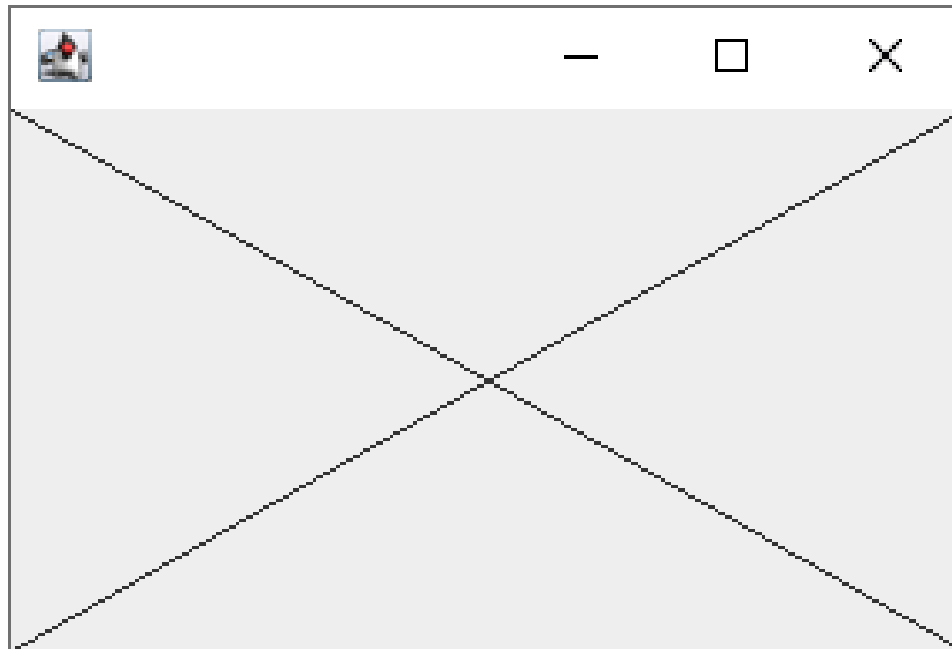
描画領域の大きさはコンポーネントの  
getSizeメソッドで取得できる



# 直線の描画

---

```
public void paintComponent(Graphics g) {  
    Dimension d = getSize();  
    g.drawLine(0, 0, d.width, d.height);  
    g.drawLine(0, d.height, d.width, 0);  
}
```



# 様々な描画メソッド

---

- `drawArc` 円弧を描く
- `drawLine` 直線を描く
- `drawOval` 楕円を描く
- `drawPolygon` 多角形を描く
- `drawPolyline` 折れ線を描く
- `drawRect` 長方形を描く
- `drawString` 文字列を描く
- `setColor` 描画色を指定する

# Graphics2Dクラス

---

- Graphicsクラスよりも高機能なグラフィックス機能を提供する  
(グラデーション、アンチエイリアシング、線の太さ、線のスタイルの指定など)
- paintComponentメソッドの引数のGraphicsオブジェクトをキャストして使用できる

```
public void paintComponent(Graphics g) {  
    Graphics2D g2d = (Graphics2D)g;  
    // Graphics2Dのメソッドを使った描画  
}
```

# マウスイベント

---

- マウスクリック、カーソルの移動、ドラッグ操作などもイベントの1つ
- イベント処理によって、マウスで操作できるプログラムを作成できる
- 以下ものが必要
  1. リスナ登録
  2. リスナインタフェースの実装
  3. イベントが通知された時の処理



# 2つのイベントリスナ

---

- MouseListenerインタフェース
  - `void mouseClicked(MouseEvent e)`
  - `void mouseEntered(MouseEvent e)`
  - `void mouseExited(MouseEvent e)`
  - `void mousePressed(MouseEvent e)`
  - `void mouseReleased(MouseEvent e)`
- MouseMotionListenerインタフェース
  - `void mouseDragged(MouseEvent e)`
  - `void mouseMoved(MouseEvent e)`
- それぞれ次のメソッドでリスナ登録する
  - `addMouseListener`
  - `addMouseMotionListener`

```
import java.awt.event.*;
import javax.swing.*;

class MyPanel extends JPanel
implements MouseListener, MouseMotionListener {

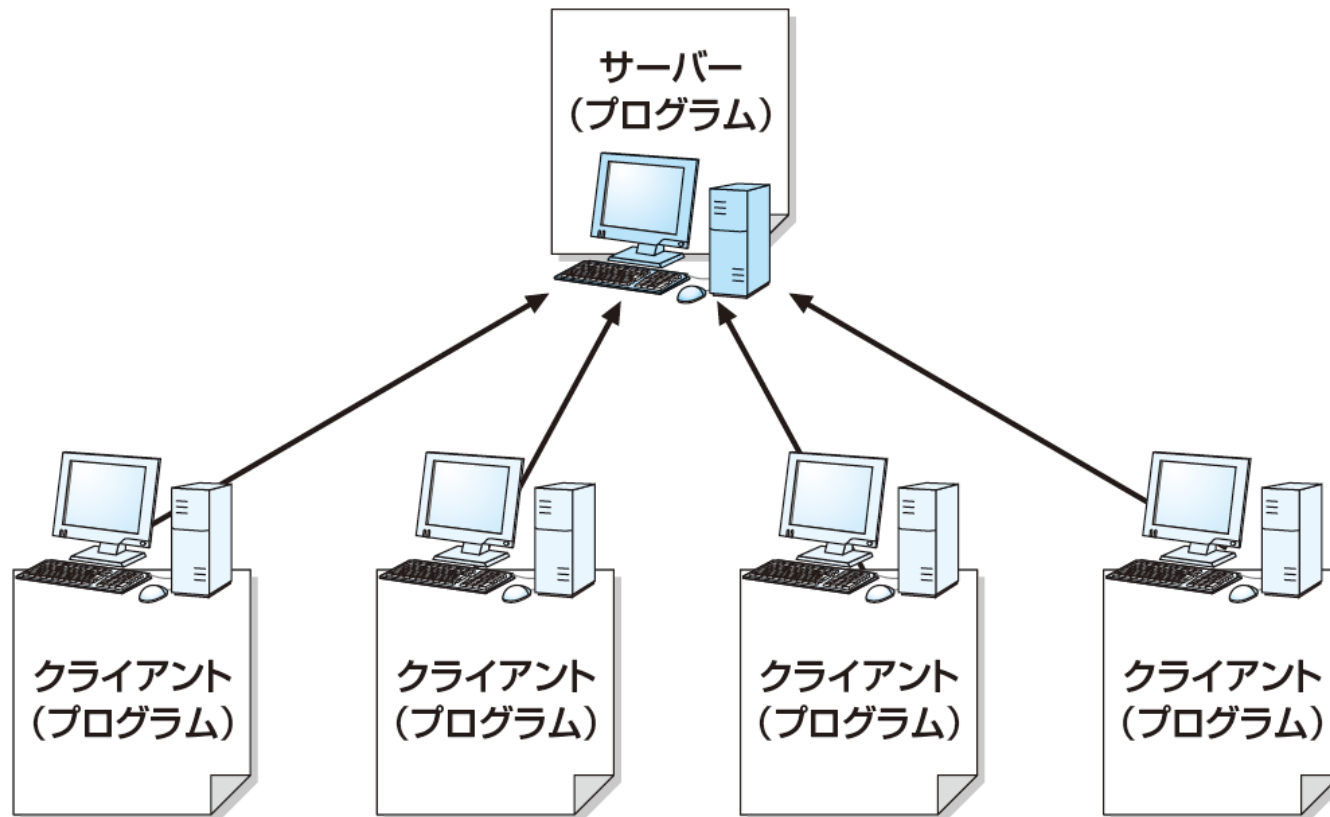
    public MyPanel() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
}
```

# 第10章 ネットワーク

# ネットワーク接続

- 一般的にTCP/IPというプロトコルが用いられることが多い
- 一方をサーバー、他方をクライアントと呼ぶ



# IPアドレスとポート番号

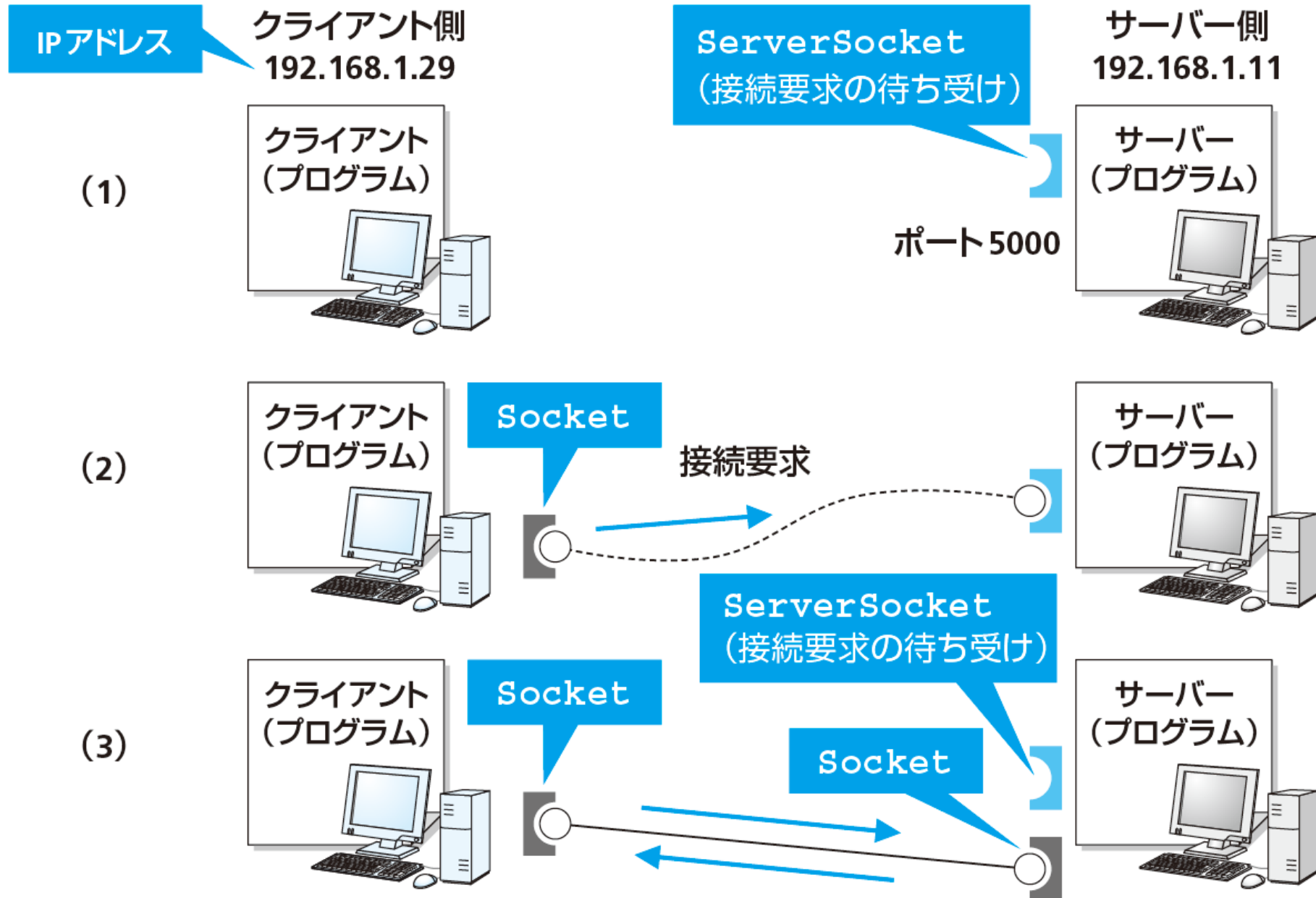
---

- クライアントがサーバーに接続要求を出す
- サーバーのIPアドレスと、サーバー側のプログラムが使用するポート番号を知っている必要がある

IPアドレスの例：192.168.1.11

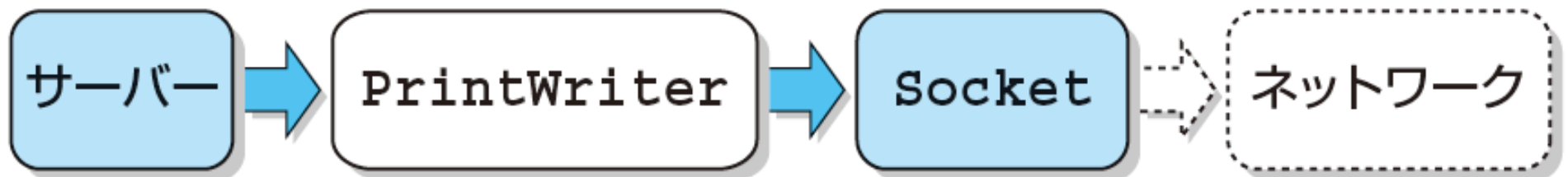
ポート番号の例：5000番

# ServerSocketとSocket



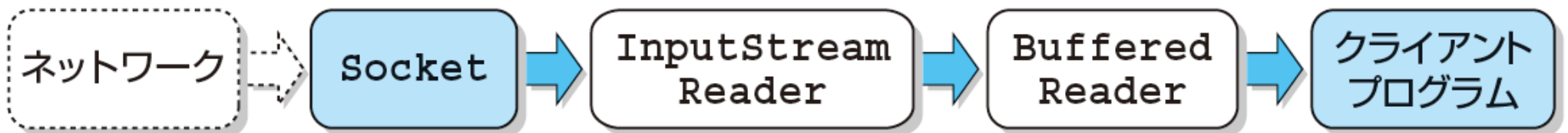
# サーバー側のプログラム例

```
try {  
    ServerSocket serverSocket = new ServerSocket(5000);  
    while(true) {  
        Socket socket = serverSocket.accept();  
        PrintWriter writer =  
            new PrintWriter(socket.getOutputStream());  
        writer.println("こんにちは。私はサーバです。");  
        writer.close();  
    }  
} catch(IOException e) {  
    System.out.println(e);  
}
```








# クライアント側のプログラム例

```
try {  
    Socket socket = new Socket("127.0.0.1", 5000);  
    BufferedReader reader = new BufferedReader(  
        new InputStreamReader(socket.getInputStream()));  
    String message = reader.readLine();  
    System.out.println("サーバーからの文字列:" + message);  
    reader.close();  
} catch(IOException e) {  
    System.out.println(e);  
}
```







クライアント	サーバー
<p data-bbox="152 402 1115 683">② <code>Socket socket = new Socket("127.0.0.1", 5000);</code> サーバーの5000番ポートに接続要求を出す。接続できたタイミングでSocketオブジェクトが生成される</p>  <p data-bbox="152 1145 1115 1481">④ <code>BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));</code> 確立したソケット通信に基づいて入力ストリームを作成する</p> 	<p data-bbox="1169 140 2101 306">① <code>ServerSocket serverSocket = new ServerSocket(5000);</code> 5000番ポートで接続要求を待ち受ける</p>  <p data-bbox="1169 769 2101 1002">③ <code>Socket socket = serverSocket.accept();</code> クライアントからの接続を受け付け、ソケット通信が確立する</p>  <p data-bbox="1169 1145 2101 1423">④ <code>PrintWriter writer = new PrintWriter(socket.getOutputStream());</code> 確立したソケット通信に基づいて出力ストリームを作成する</p> 

## クライアント


## サーバー




⑥ `String message = reader.readLine();`  
ストリームから文字を受け取る



⑧ `reader.close();`  
ストリームを閉じる



⑤ `writer.println`  
`("こんにちは。私はサーバーです。");`  
ストリームに文字列を出力する



⑦ `writer.close();`  
ストリームを閉じる

# 第11章 一歩進んだ Javaプログラミング

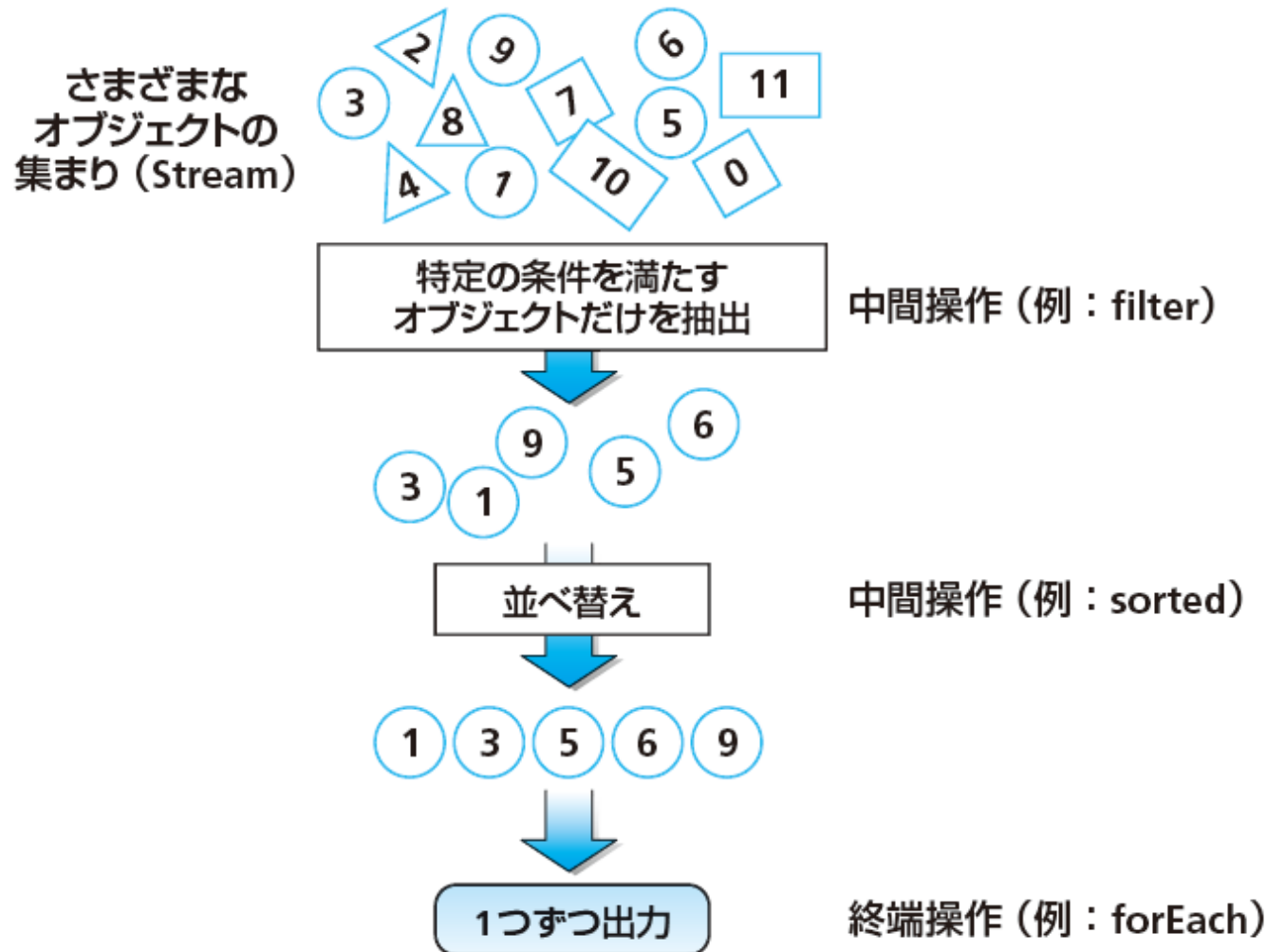
# コレクションとストリーム

---

- コレクションに格納されたオブジェクトを扱うために拡張for文やforEachメソッドを使って一つ一つ取り出す方法がある
- ストリームという概念を使えば、たくさんのプログラムコードが必要だった処理を短く直感的に記述できる

# ストリーム

オブジェクトのまとまりに対して、なにかしらの処理を行って、最後に出力を行う



# ストリームの生成

---

List<String>型のコレクションに対するstreamメソッドを利用

```
List<String> list = Arrays.asList("January", "February", "March");  
Stream<String> stream = list.stream();
```

Streamインタフェースのofメソッドを利用

```
Stream<String> steam = Stream.of("January", "February", "March");
```

# ストリームに対する終端操作

---

## ストリームに対する一連の操作の最後に行う

作成したストリームからオブジェクトを1つずつ取り出してコンソールに出力

```
List<String> list = Arrays.asList("January", "February", "March");  
list.stream().forEach(s -> System.out.println(a));
```

listに含まれる要素数をコンソールに出力

```
Long n = list.stream().count();  
System.out.println(n);
```

# ストリームに対する中間操作

---

- filterメソッドは、ある条件を満たすオブジェクトだけを取り出して次の操作に渡すことができる

条件に合うオブジェクトの数を取得

```
long l = list.stream().  
filter(s -> s.length() > 5).  
count();
```

※戻り値がStreamオブジェクトなので別のメソッドをドット (.)で連結できる。

- mapメソッドは、個々のデータを別のオブジェクトにマッピングできる

前後を[]で囲った文字列を出力

```
list.stream().  
map(s -> "[" + s + "]").  
forEach(s -> System.out.println(s));
```



# ストリームに対する中間操作

---

- Sortedメソッドはデータを並べ替えることができる。

文字列の長さで並べ替える

```
sorted((s0, s1) -> s0.length() - s1.length())
```

# ストリーム処理の例

---

```
List<String> list = Arrays.asList("January",  
    "February", "March", "April", "May",  
    "June", "July", "August", "September",  
    "October", "November", "December");
```

listに格納された文字列の文字数が5以下のものに対して、アルファベット順で並べ替え、文字列の両端に "[" と "]" 記号をつけて、最後に1つ1つ出力

```
list.stream().  
    filter(s -> s.length() <= 5).  
    sorted().  
    map(s -> "[" + s + "]").  
    forEach(s-> System.out.println(s));
```

# スタティックインポート

---

パッケージ名とクラス名を省略して、クラス変数やクラスメソッドを直接記述できる

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class StaticImportExample {
    public static void main(String[] args) {
        System.out.println("PI=" + PI);
        System.out.println("abs(-2)=" + abs(-2));
    }
}
```

# インタフェースのデフォルトメソッド

---

- インタフェースのメソッドにdefault(デフォルト)修飾子をつけるとメソッドの中身を記述できる
- デフォルトメソッドを宣言すれば、メソッドをオーバーライドせずに済ませられる
- 通常のインタフェースと同様にオーバーライドすることもできる

```
interface SayHello {  
    default void hello() {  
        System.out.println("Hello");  
    }  
}
```

# デフォルトメソッドの例

---

```
interface SayHello {  
    default void hello() {  
        System.out.println("Hello");  
    }  
}  
  
class EnglishGreet implements SayHello {  
}  
  
class JapaneseGreet implements SayHello {  
    public void hello() {  
        System.out.println("こんにちは");  
    }  
}
```

```
SayHello a = new EnglishGreet();  
SayHello b = new JapaneseGreet();  
a.hello(); →Helloと出力  
b.hello(); →こんにちはと出力
```

# アノテーション

---

- メソッド名の前の`@Override`など、記号(@)を先頭に持つ表記
- 「このメソッドはスーパークラスまたはインタフェースのメソッドをオーバーライドしたものである」ということをコンパイラに伝える役割を持つ
- 適切にオーバーライドができていないときに、コンパイラがエラーを出す

# System.out.printfメソッド

---

文字列の中の特別な記号（%d, %fなど）を、引数で指定した変数の値に置き換えることができる

```
int x = 5;  
int y = 10;  
int z = 15;
```

```
System.out.printf("x=%d%n", x);  
System.out.printf("(x,y)=(%d,%d)%n", x,y);  
System.out.printf("(x,y,z)=(%d,%d,%d)%n", x,y,z);
```

%d 整数、 %f 小数を含む値、 %s 文字列、 %n 改行

# enum宣言

---

特定の定数の値だけを取れる独自の型を定義できる

```
class Student {  
    enum Gender { MALE, FEMALE };  
    String name;  
    Gender gender;  
}
```

```
Student s = new Student();  
s.name = "山田太郎";  
s.gender = Student.Gender.MALE;
```



# == 演算子とequalsメソッド

---

- ==演算子を使って2つのオブジェクト参照を比較した場合、両方が同一のインスタンスを参照している場合はtrue, そうでない場合はfalse
- equalsメソッドはObjectクラスに備わっているメソッド。オーバーライドして自分で定義できる
- コレクションクラスでは、オブジェクトの比較にequalsメソッドが使用される