

# Java 学習教材(2)

筑波大学 コンピュータサイエンス専攻 三谷 純  
最終更新日 2017/3/15

# 本資料の位置づけ

Java 実践編  
アプリケーション作りの基本  
(三谷純 著)

本資料は

『Java 実践編  
アプリケーション作りの基本』

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。

どなたでも自由に使用できます。  
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も構いません

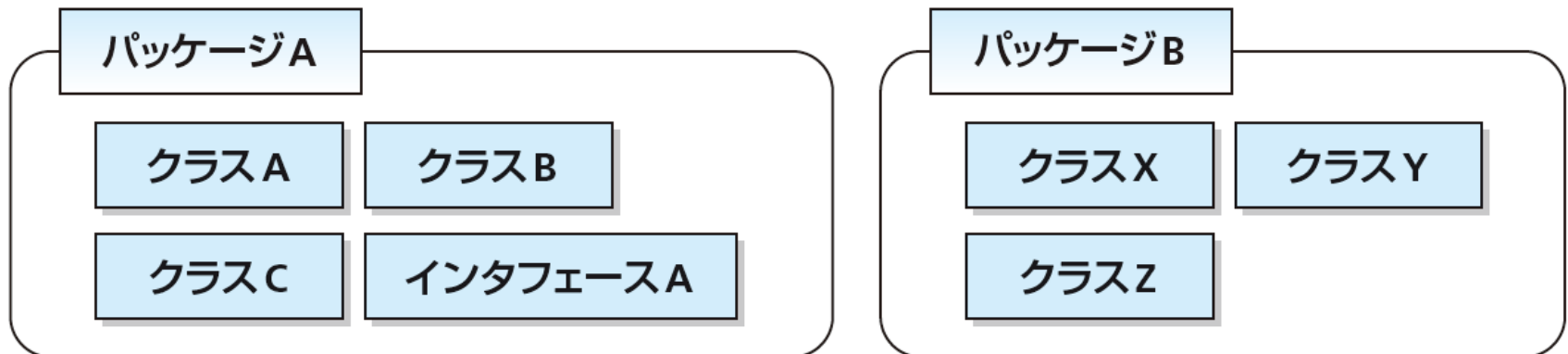


出版社： 翔泳社  
発売日： 2017/4/4  
ISBN- 13: 978- 4798151830

# 第1章 パッケージとJava API

# パッケージとは

- 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あらかじめ準備されている
- 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複数の「パッケージ」に分類されている



# Javaの主なパッケージ

パッケージ名	説明
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）
ジャバエフエックス・シーン・コントロール <code>javafx.scene.control</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）
ジャバエフエックス・シーン・シェイプ <code>javafx.scene.shape</code>	四角や円などの図形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）
ジャバエフエックス・イベント <code>javafx.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）
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）

# 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

---

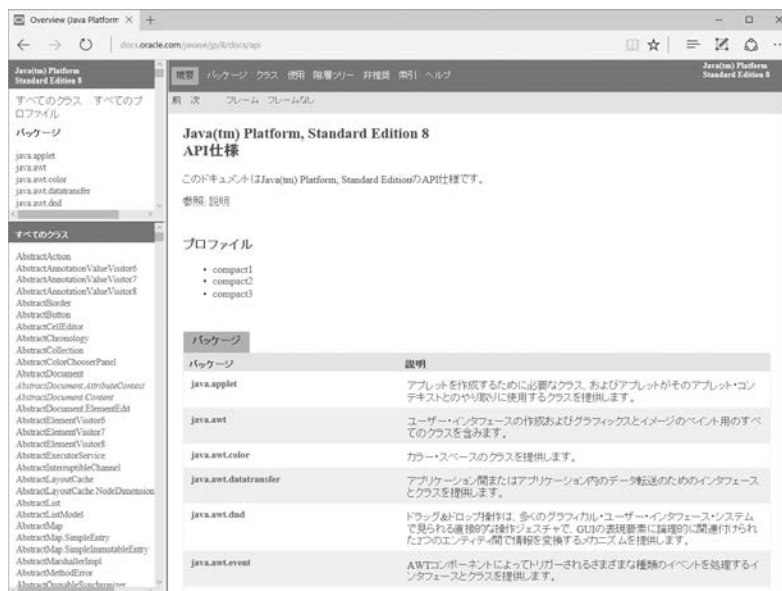
- java.langパッケージにはJavaの基本的な機能を提供するクラスが含まれる。
- 「System.out.println()」のSystemクラスもjava.langパッケージに含まれる。
- 「import java.lang.\*;」という記述は省略できる。

# API 仕様書

Javaにはあらかじめ4000以上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/8/docs/api/>



# API 仕様書で確認できるクラス情報

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

- Randomクラスは  
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クラスが  
ある

# クラスの説明

クラスの説明、  
フィールド、  
コンストラク  
タ、メソッド  
の説明が続く

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

Random クラスのインスタンスは、一連の擬似乱数を生成します。クラスでは『[Java Programming, Volume 3](#)』の 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、それぞれに対して同じ結果を返すことを保証するために、固有のアルゴリズムが Random クラスに指定されます。Java 1.0 以降に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、すべてのメソッドを実装する必要があります。

Random クラスによって実装されるアルゴリズムでは、各呼び出しで擬似乱数を生成します。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。

**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

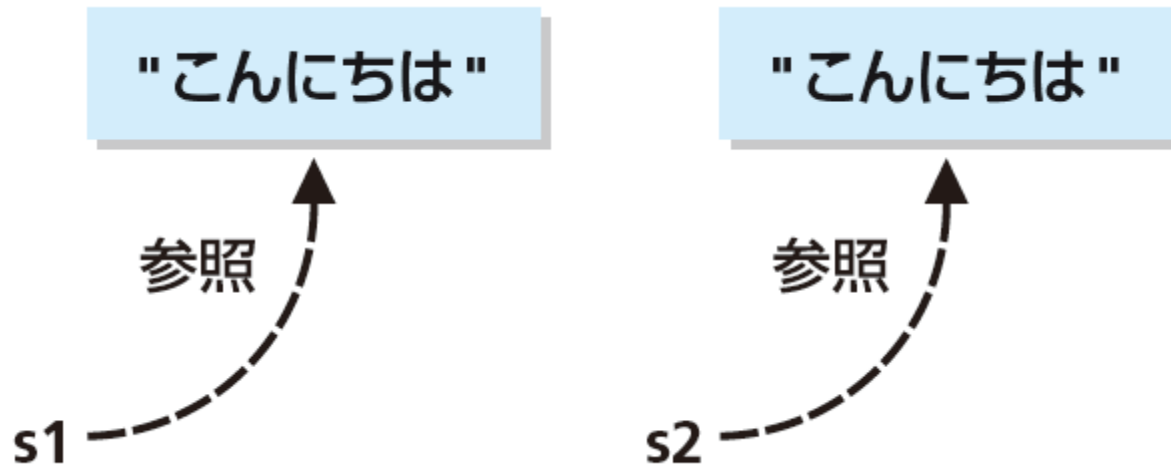
```
String message = "こんにちは";
```

```
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 = "2017/11/22";  
String[] items = str.split("/");  
for(int i = 0; i < items.length; i++) {  
    System.out.println(items[i]);  
}
```

実行結果

```
2017  
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;
```

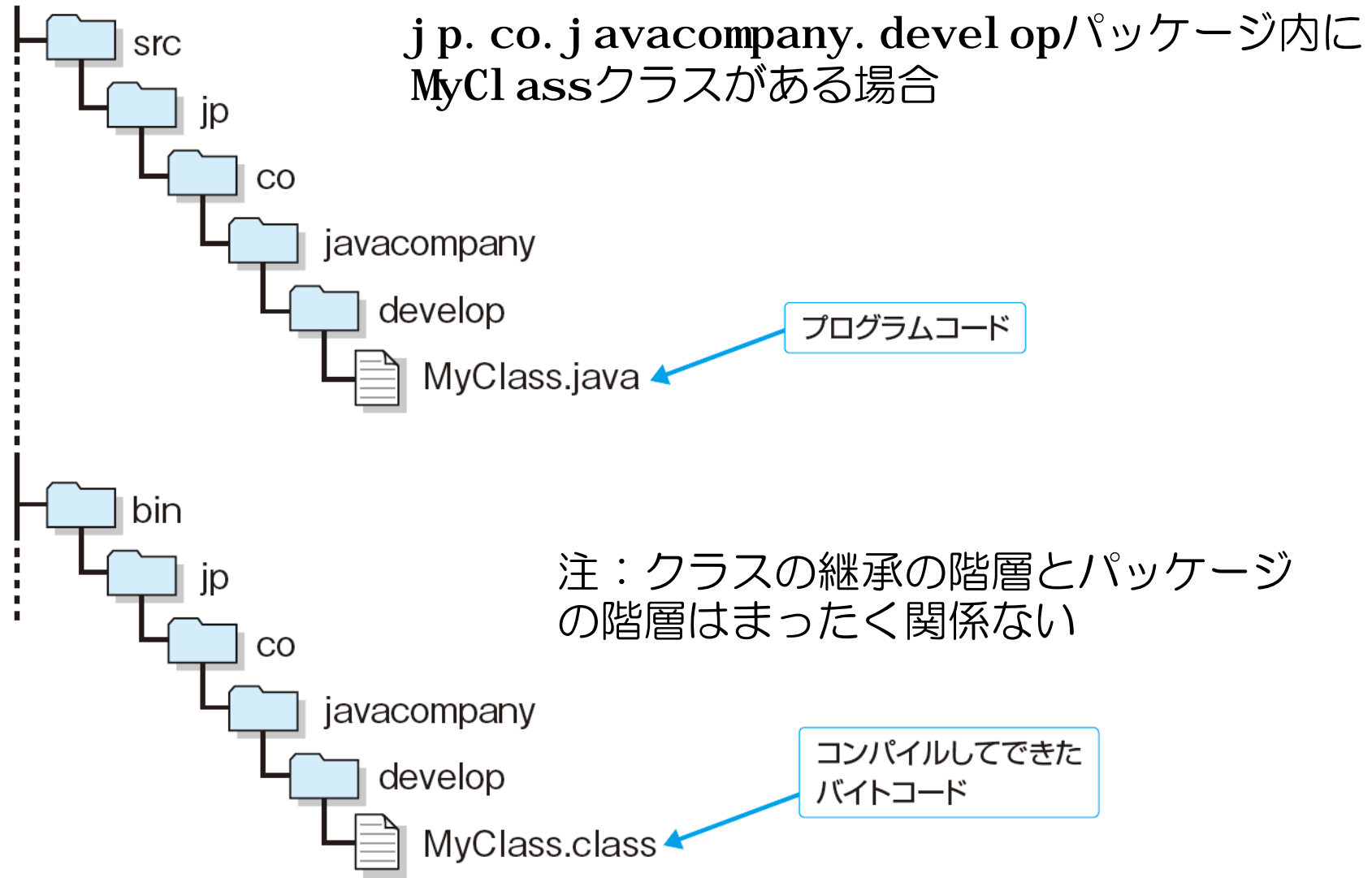
とプログラムコードの先頭に記述する。

```
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：j p. co. j avacompany. devel op



# 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。

パッケージ

```
class A {  
    public static int a;  
    public static int getValue() {  
        return 1;  
    }  
}
```

✗ `A.a = 10;`



クラス A が `public` でないので、  
パッケージの外部からはそのフィールドやメソッドにアクセスできない



✗ `int b = A.getValue();`

# 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

$4 \div 2 \rightarrow 2$

問題が発生する処理

$4 \div 0 \rightarrow ?$

例外

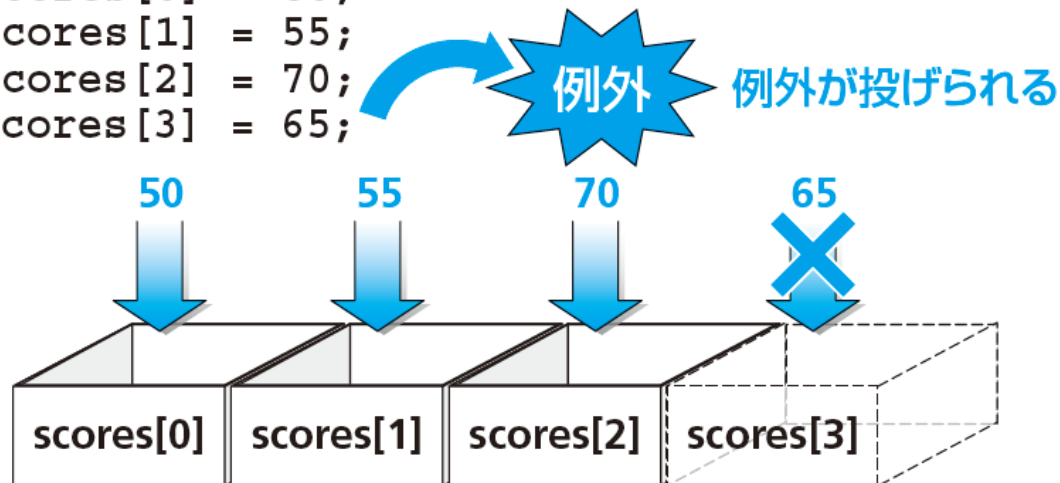
例外が投げられる

# 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。



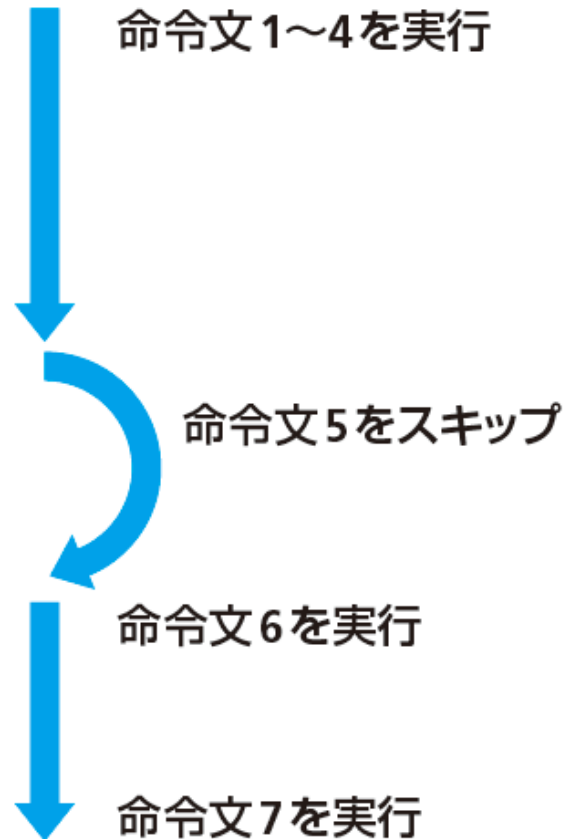
# try～catch文

```
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  
}
```

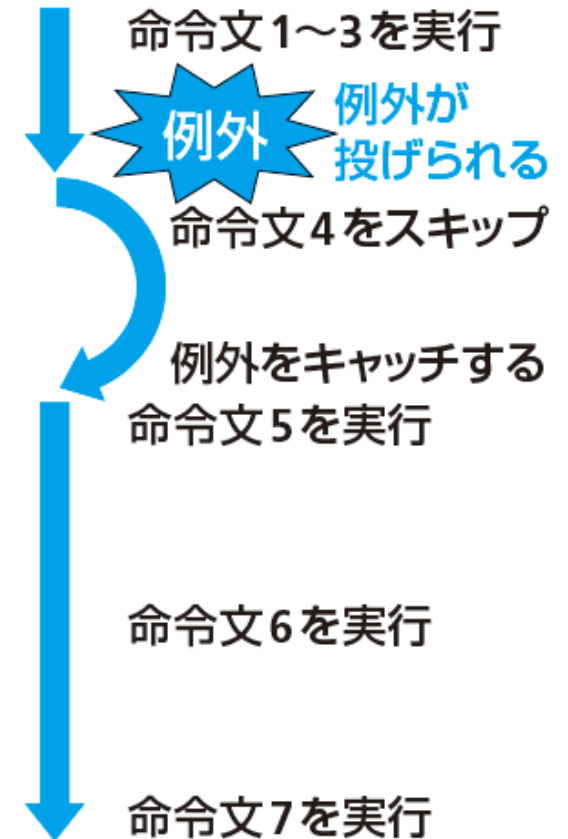
# 処理の流れ

```
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`

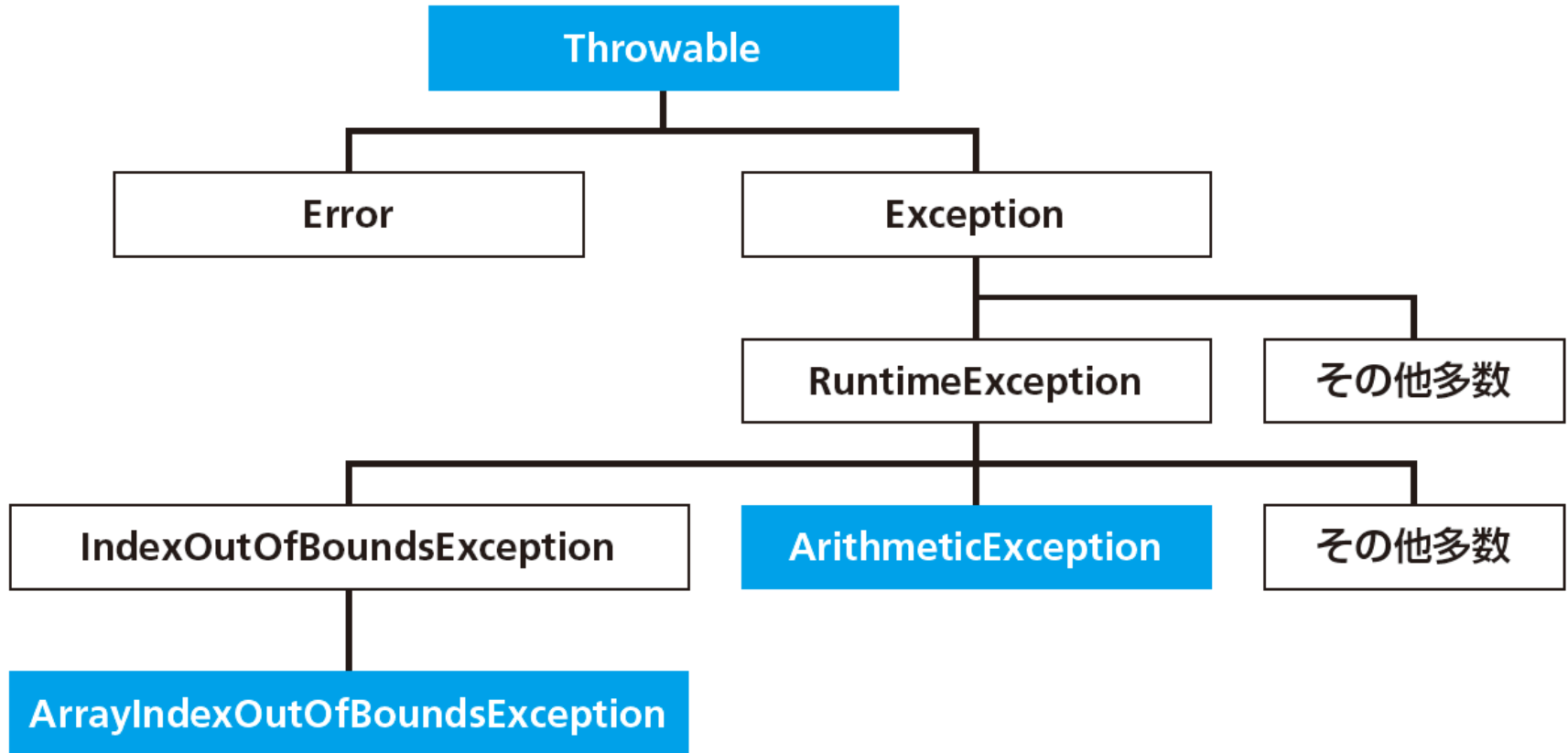
# 例外オブジェクトの種類による場合分け

---

```
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) throws 例外の型 {  
    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。
  - 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

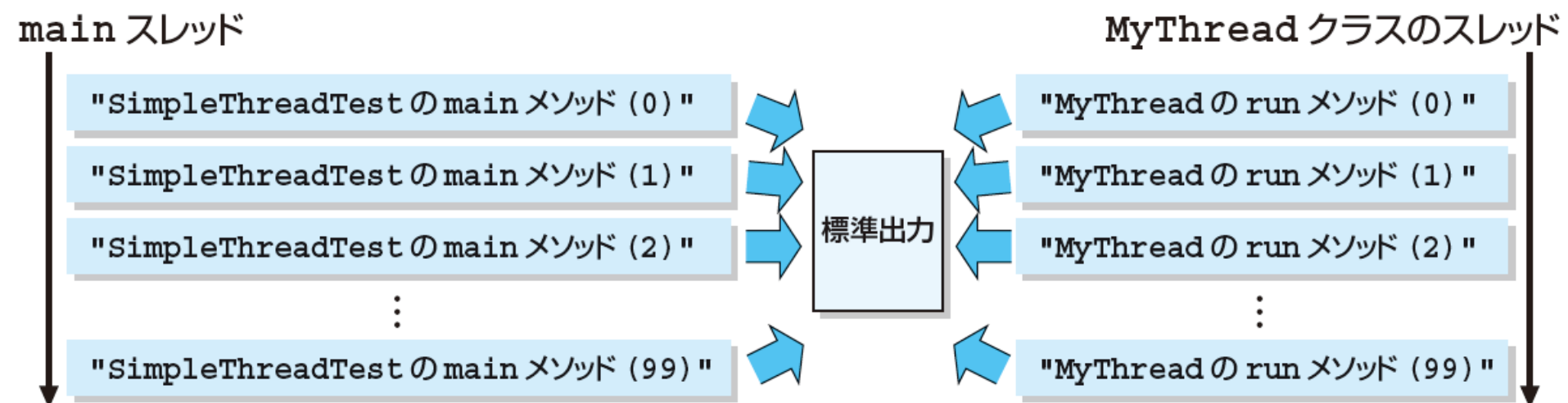


# 方法1のThreadの使用例

```
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("*");  
        }  
    }  
}
```

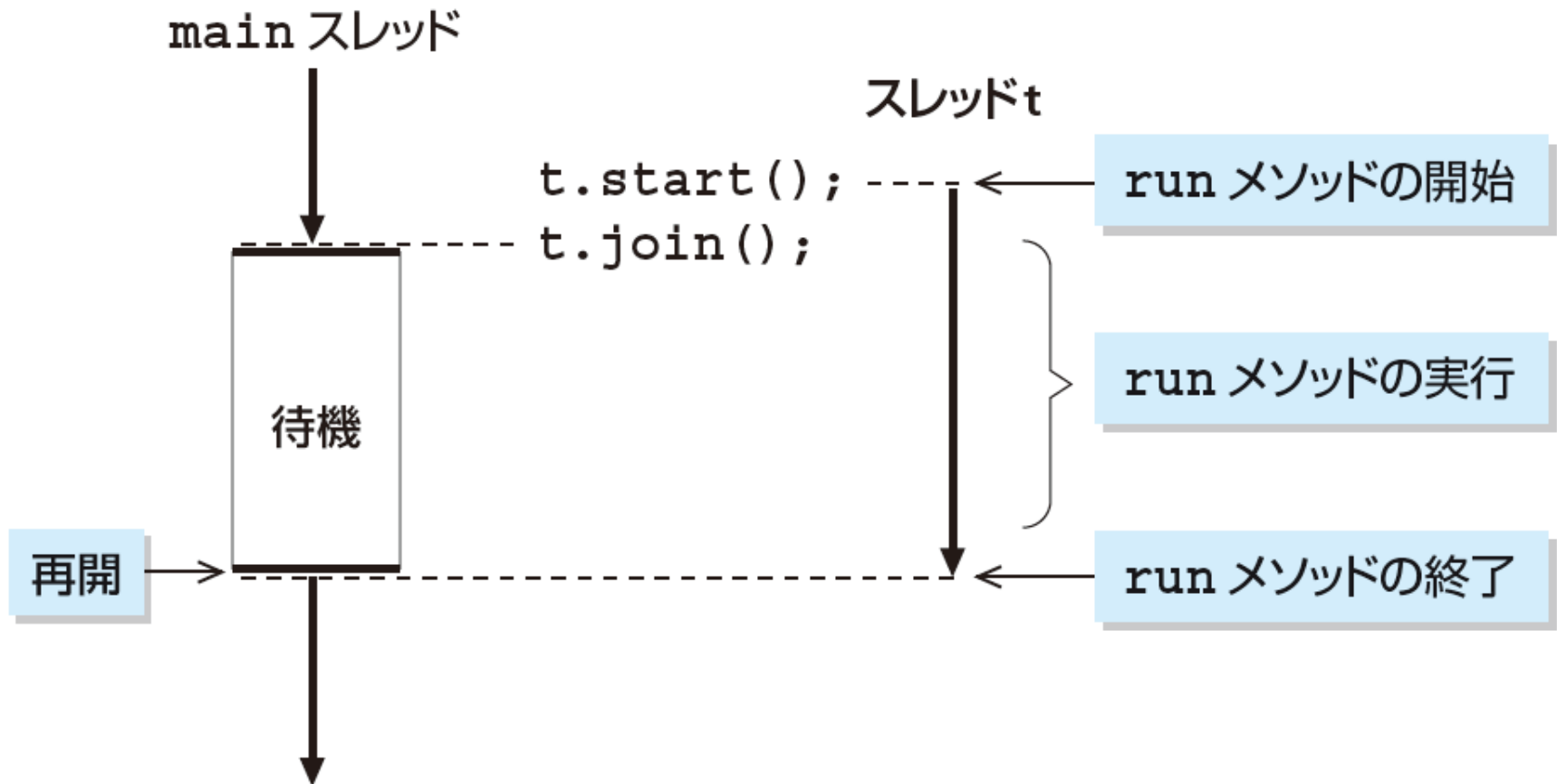
# スレッドの終了を待つ

---

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

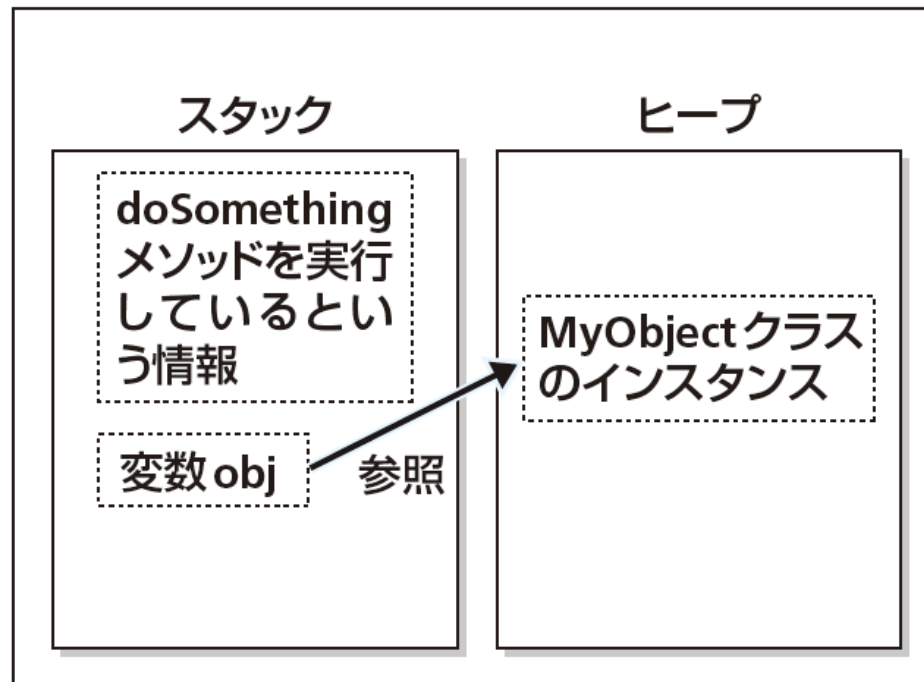
- プログラム実行中に覚えておかななくてはならない情報は**メモリ**に格納される。
- メモリには**スタック**と**ヒープ**と呼ばれる領域がある。

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>• <b>ガーベッジコレクタ</b>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;  
}
```

メモリ



# 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

---

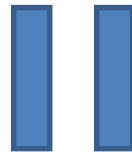
- インスタンスを大量に生成すると、ヒープを消費する。全て使い切ってしまうと実行時エラーが発生する。
- メソッドの呼び出しの階層があまりに深いとスタックも使いきる可能性がある  
(再帰呼び出しなどを行った場合)

# 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

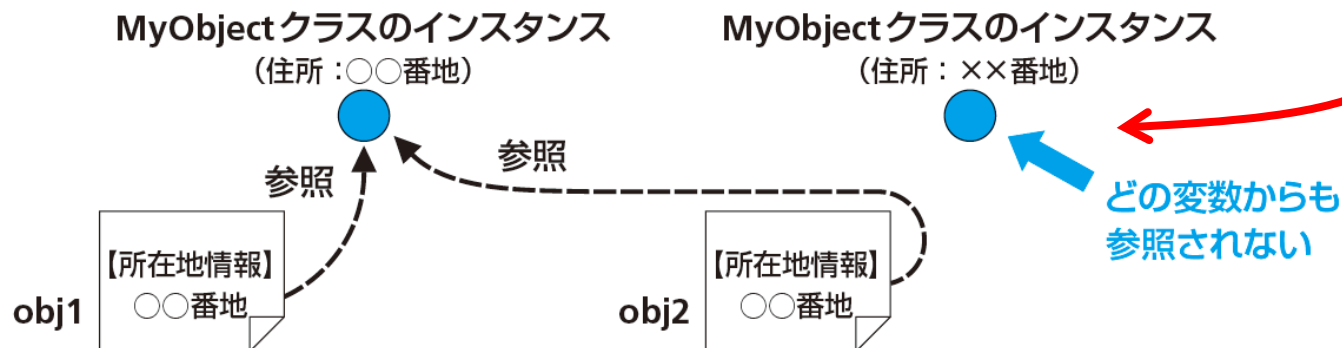
# ガーベッジコレクションの対象になるタイミング

```
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();**」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ArrayLi stオブジェクトの生成

```
ArrayLi st<Stri ng> array  
    = new ArrayLi st<Stri ng>();
```

- < >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Stri ng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という意味。
- 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

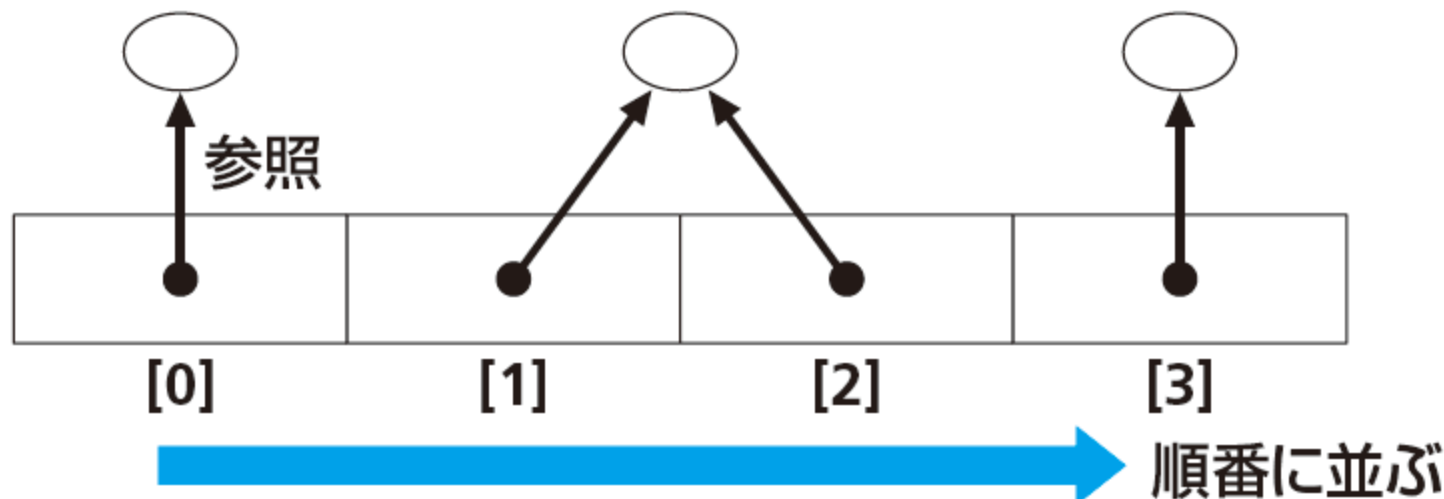
---

- 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

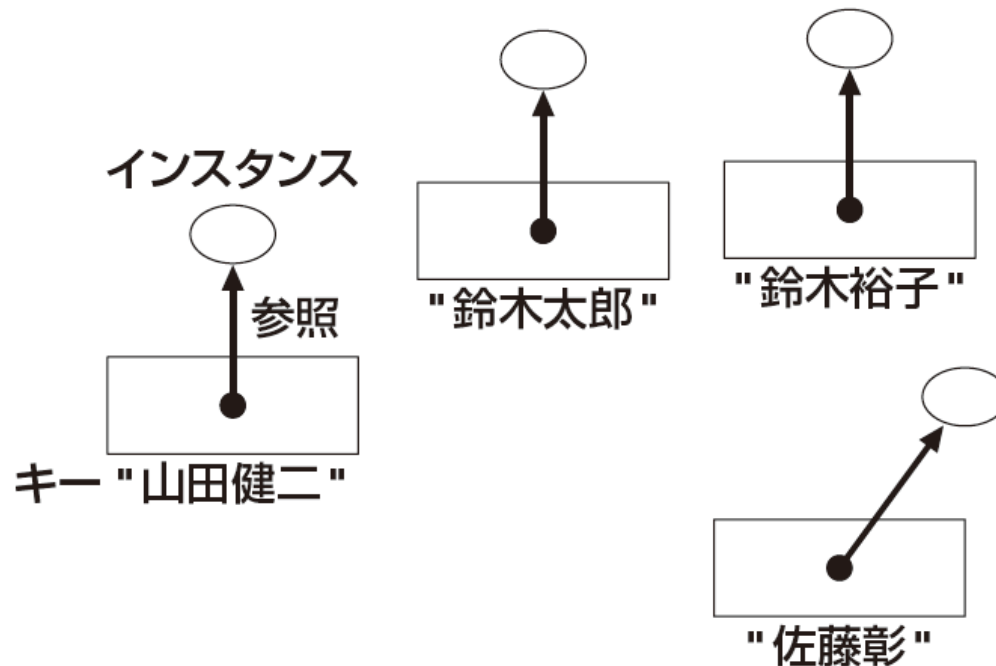
- オブジェクトが順番に並ぶ
- 「先頭から2番目のもの」「最後に追加したもの」を取りだすことができる
- 異なる要素が同一のオブジェクトを参照できる
- `List` インタフェースを実装する

インスタンス



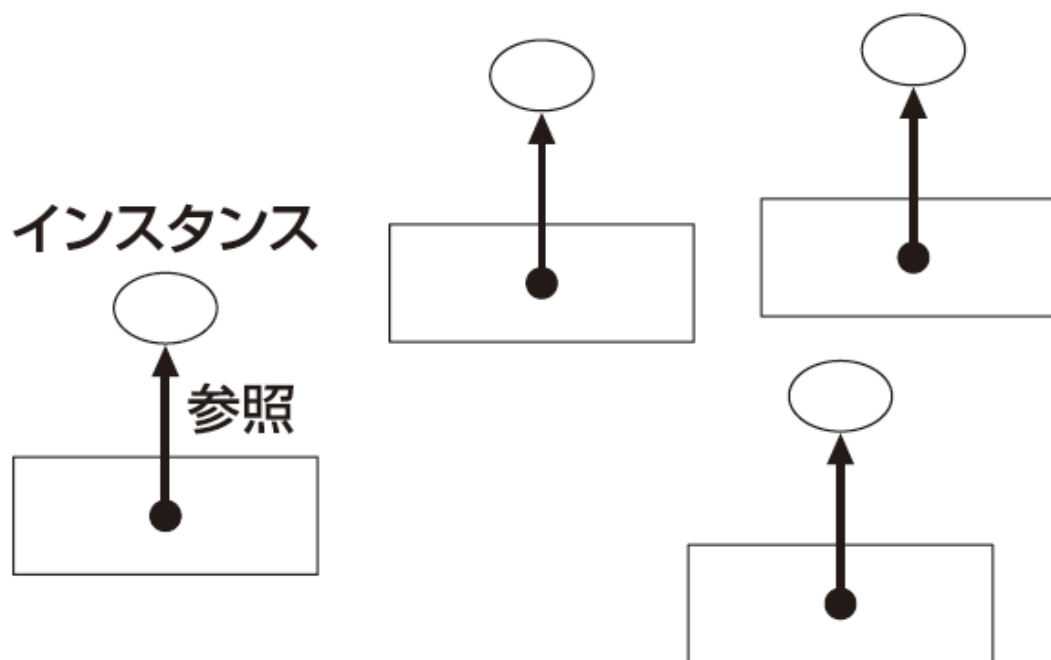
# マップ

- 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



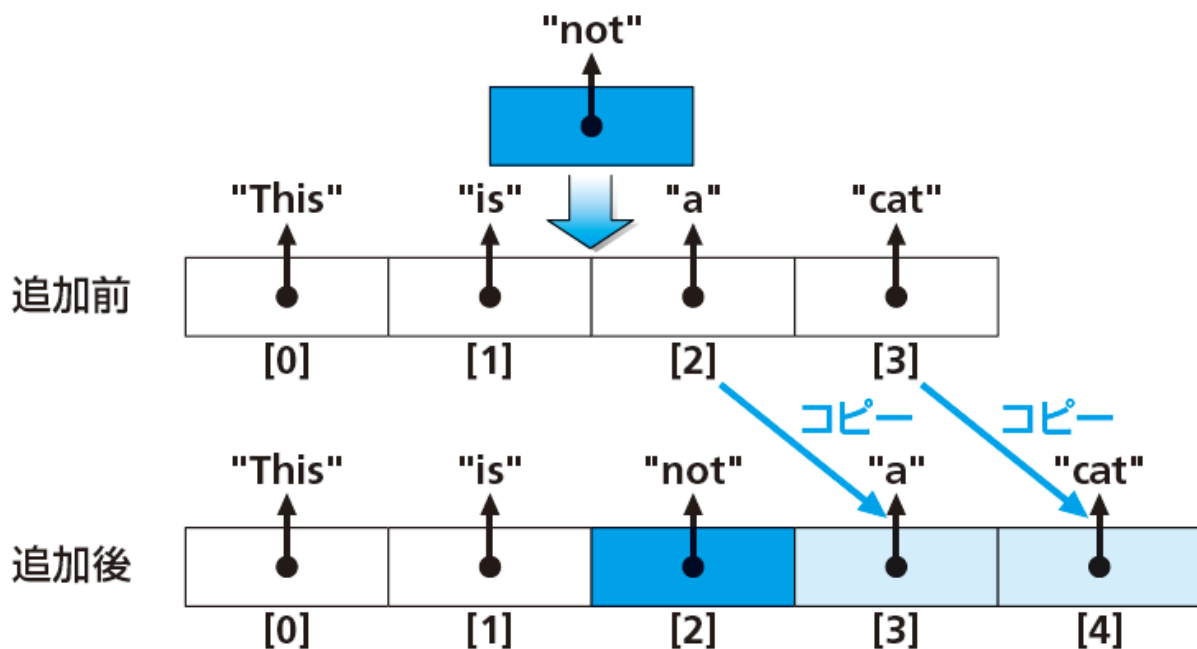
# セット

- 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：ArrayLi st

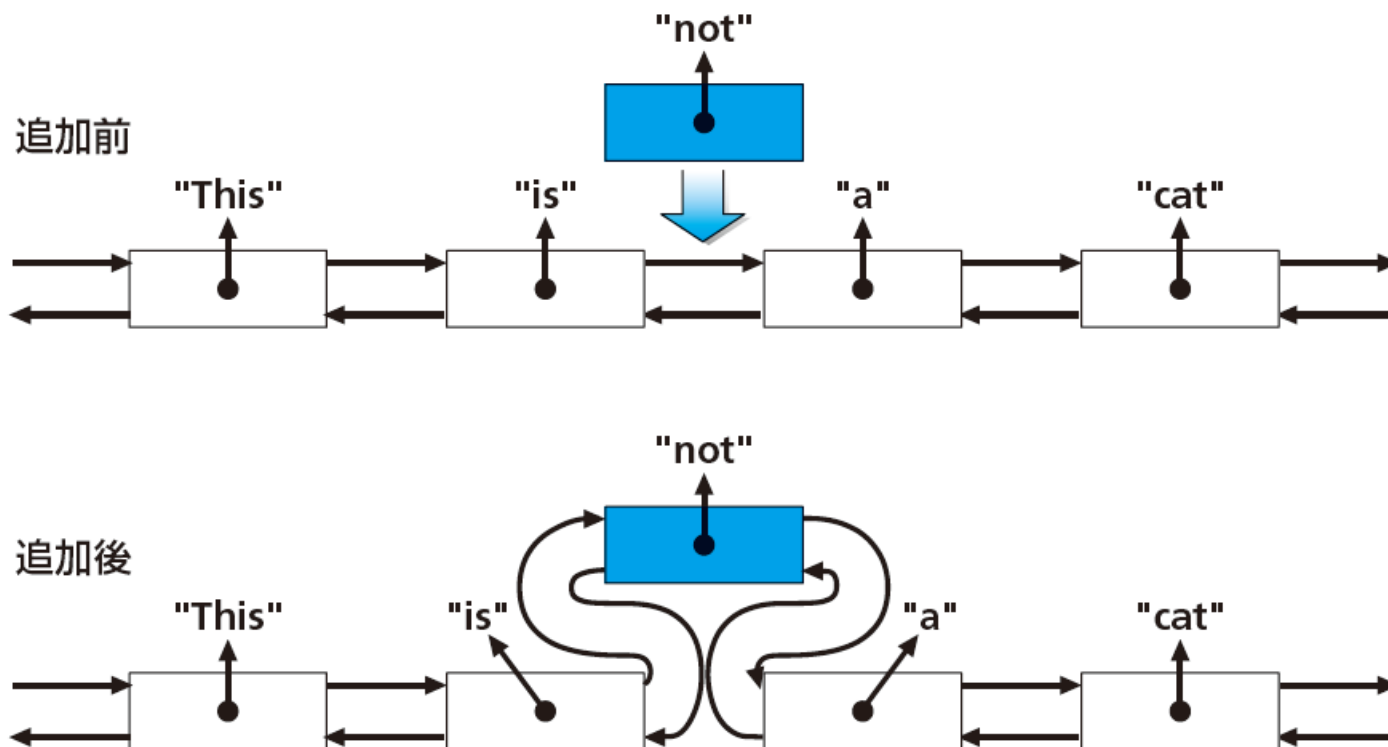
- 配列＋便利な機能
- インデックスで直接要素にアクセスできるので高速
- 要素の追加と削除には内部で要素のコピーが行われるので低速





# リストコレクション：Li nkedLi st

- 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。

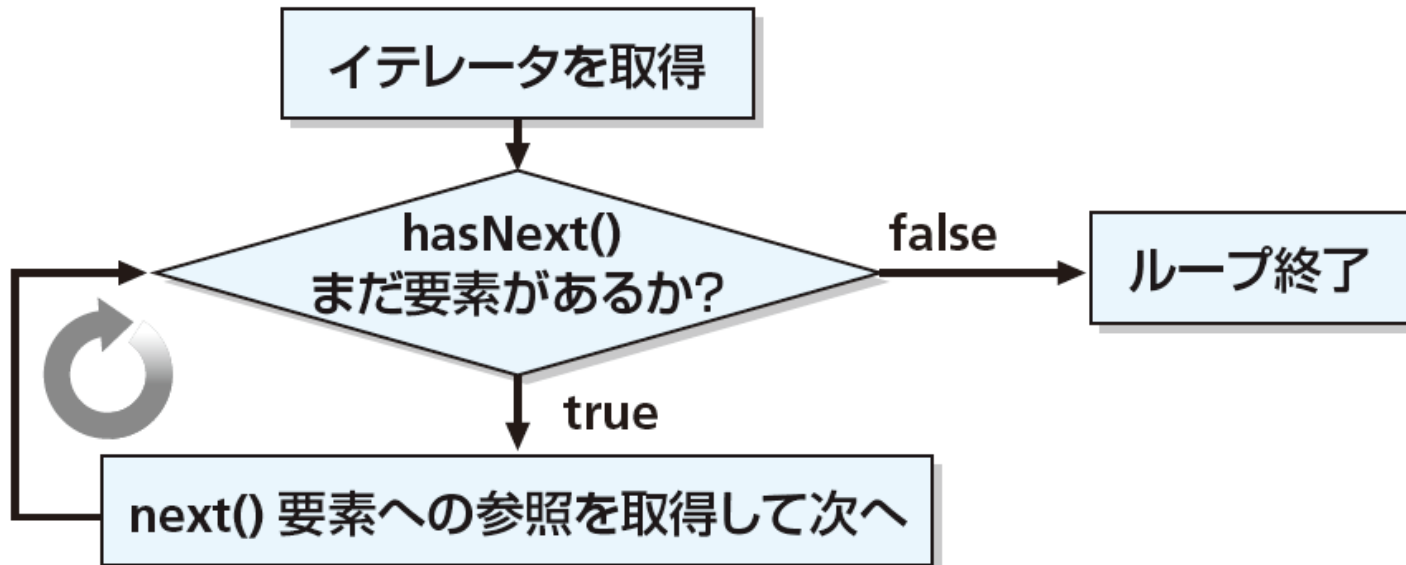
```
ArrayList<String> list = new ArrayList<String>();  
list.add("Good morning. ");  
list.add("Hello. ");  
for(int i = 0; i < list.size(); i++) {  
    System.out.println(list.get(i));  
}
```

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
  - `Element 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

---

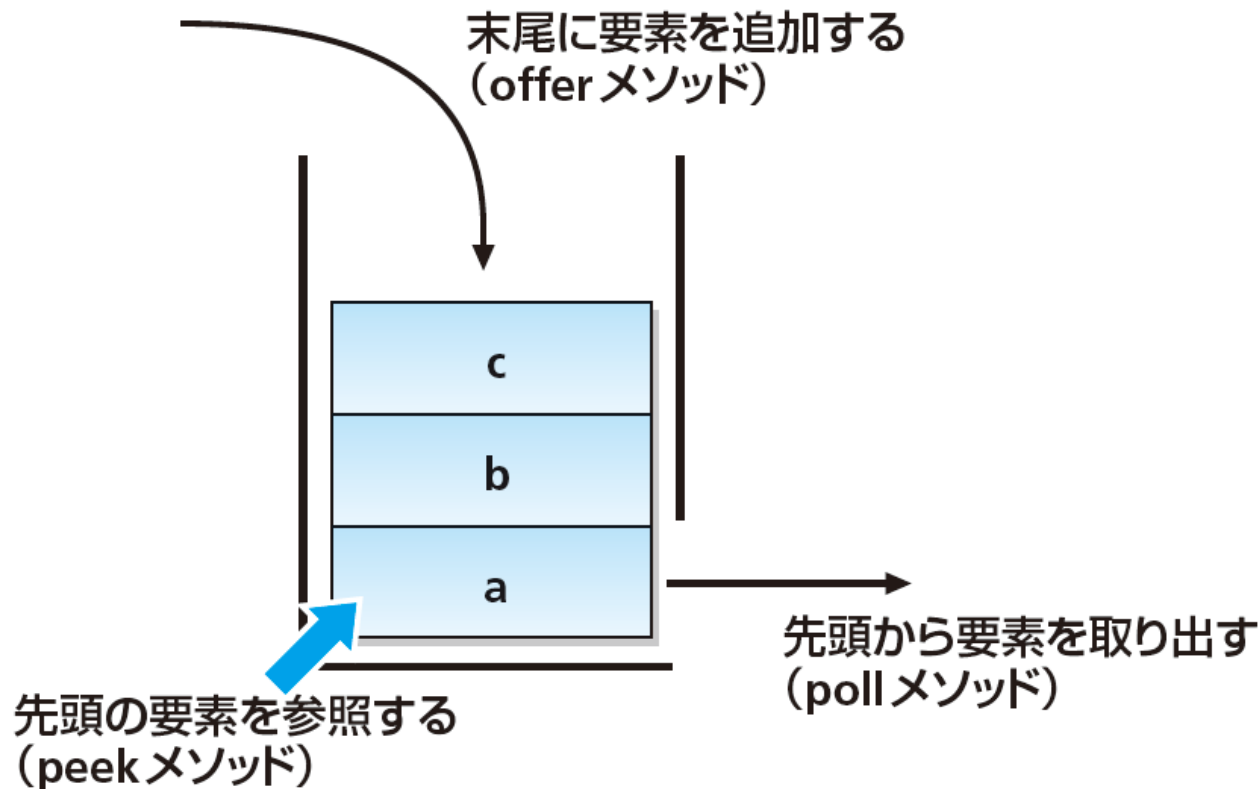
通常のfor文、イテレータを使ったアクセスよりも簡潔に記述できる構文

```
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よくある。
- `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： $n \rightarrow \{ \text{return } n + 1; \}$ 省略形： $n \rightarrow n + 1;$
------------------------------------------------------------------------------------

# 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。

<code>(Point p0, Point p1) -&gt;</code>	<code>{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}</code>
-----------------------------------------	-----------------------------------------------

# ラムダ式を用いた並べ替えの例

---

Poi nt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

```
poi ntLi st. 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**（Uni code, 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

---

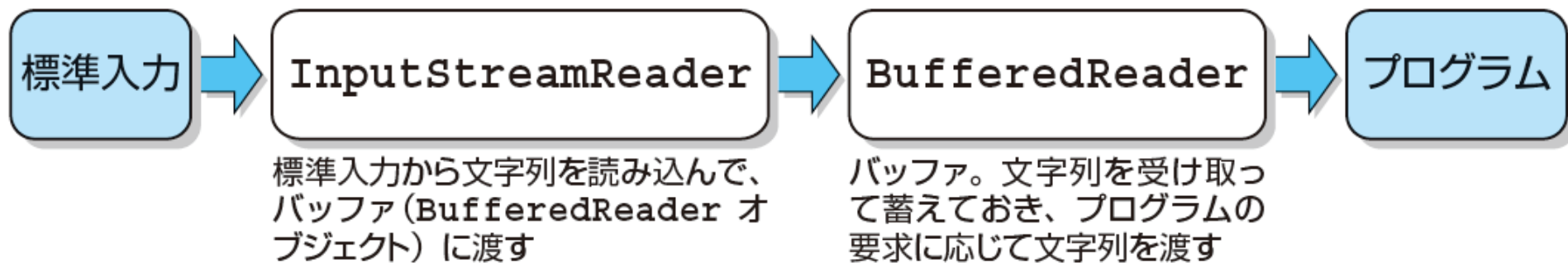
- 入出力先を特に指定しなかった場合に標準的に使用される出力。一般的には「**コンソール**」。

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

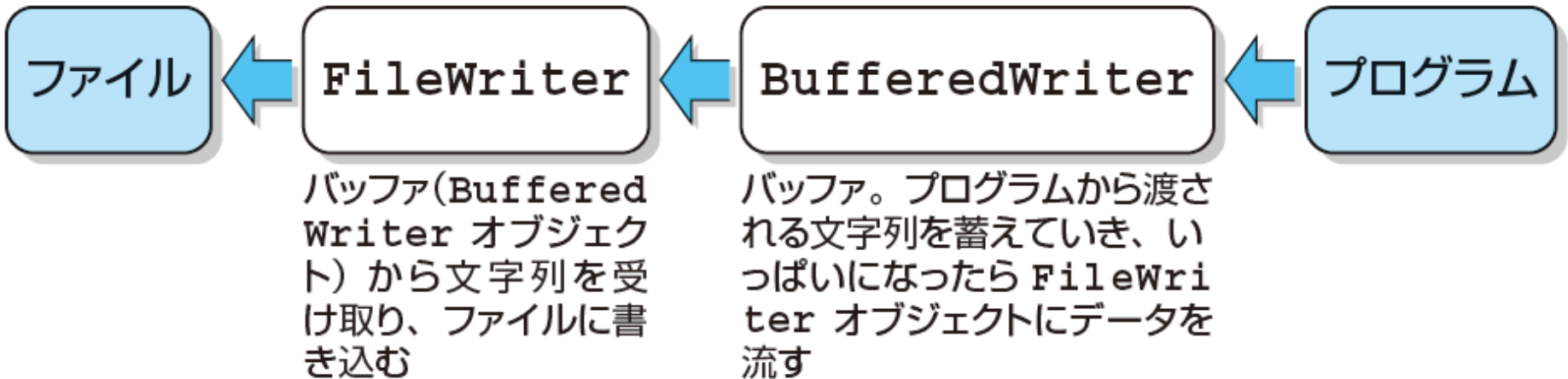
---

- 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);
```



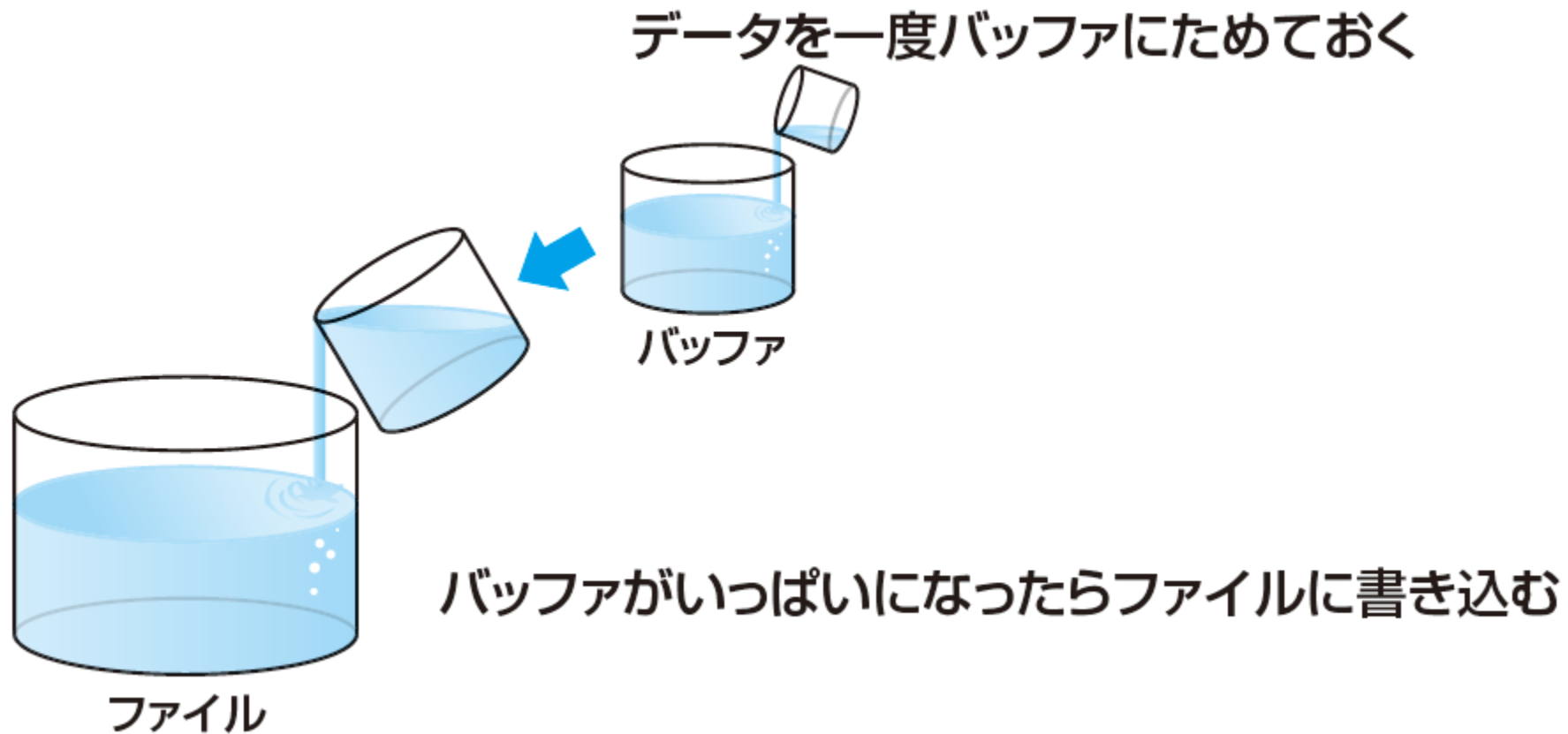
# ファイルへの出力



```
try {  
    File file = new File("C:¥¥j ava¥¥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);  
}
```

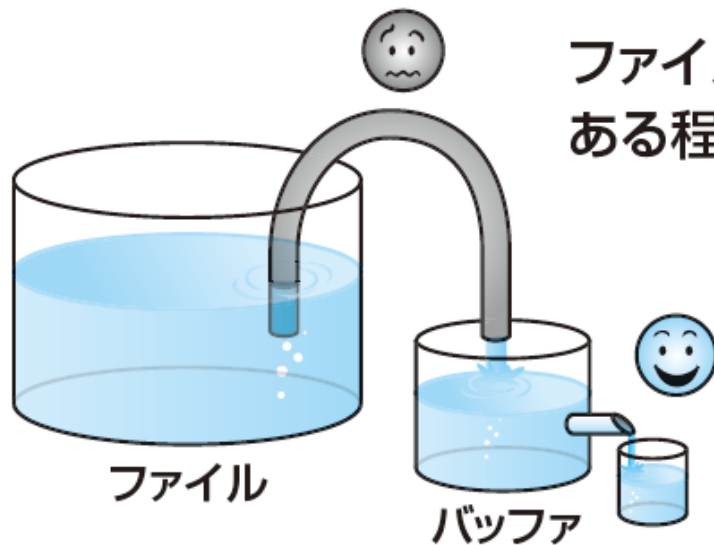
# バッファを用いたデータの書き込み

---



# バッファを用いたデータの読み込み

---

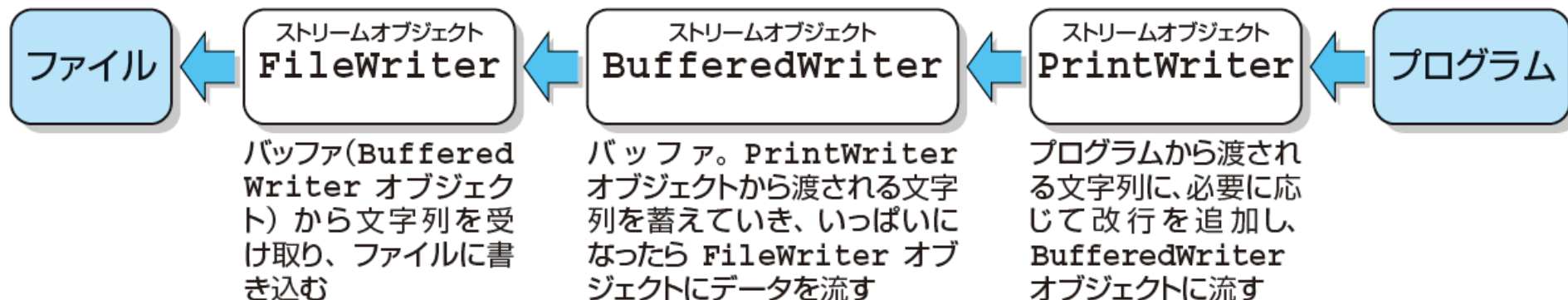
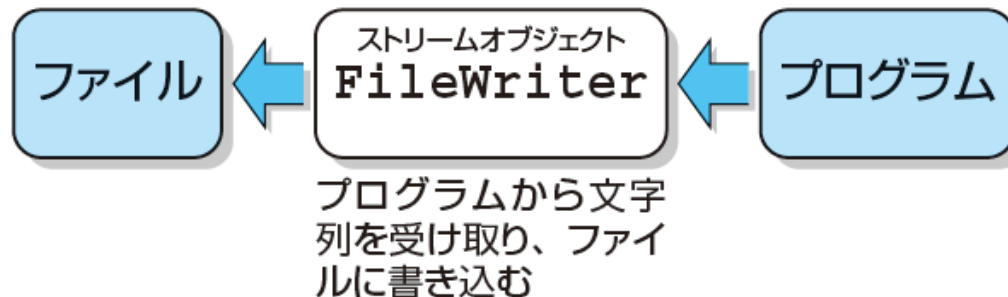


ファイルからデータを受け取るのは手間がかかるので、  
ある程度まとめて受け取って、バッファに蓄えておく

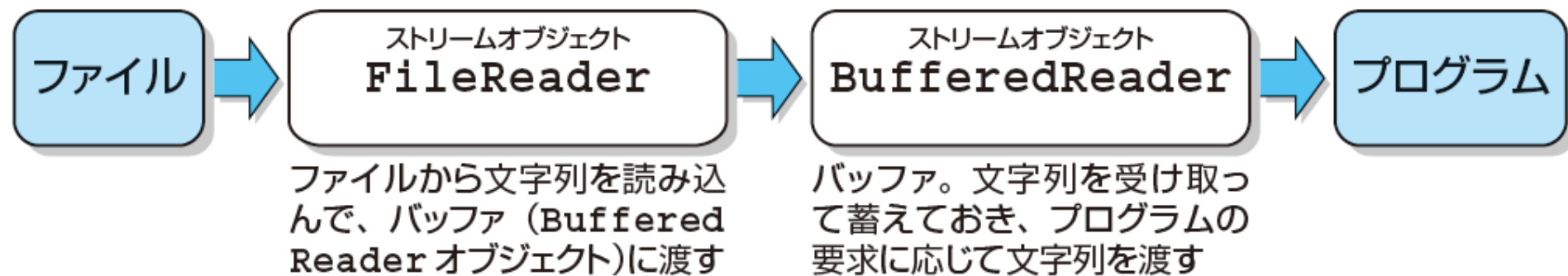
バッファからデータを受け取るのは簡単なので、  
必要に応じてバッファからデータを読み取る

# 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

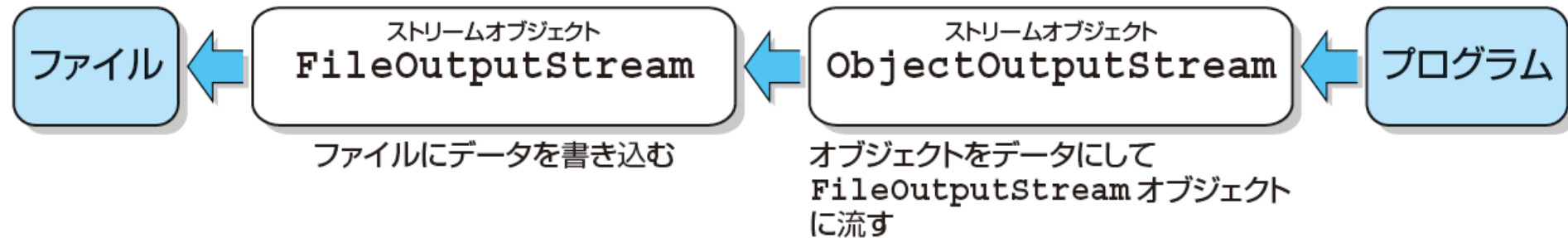
方法2. シリアライゼーションによってオブ  
ジェクトそのものをファイルに保存する

# 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;  
}
```

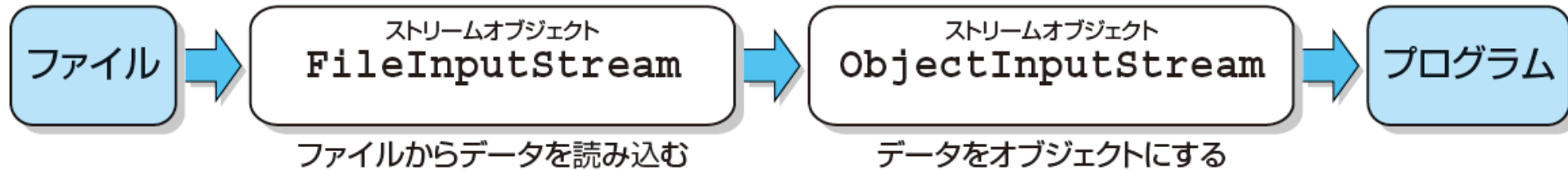
# シリアライゼーションを使用したオブジェクトの保存



```
try {
    FileOutputStream fs =
        new FileOutputStream("C: ¥¥j ava¥¥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: ¥¥j ava¥¥tri angl e. 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");
```

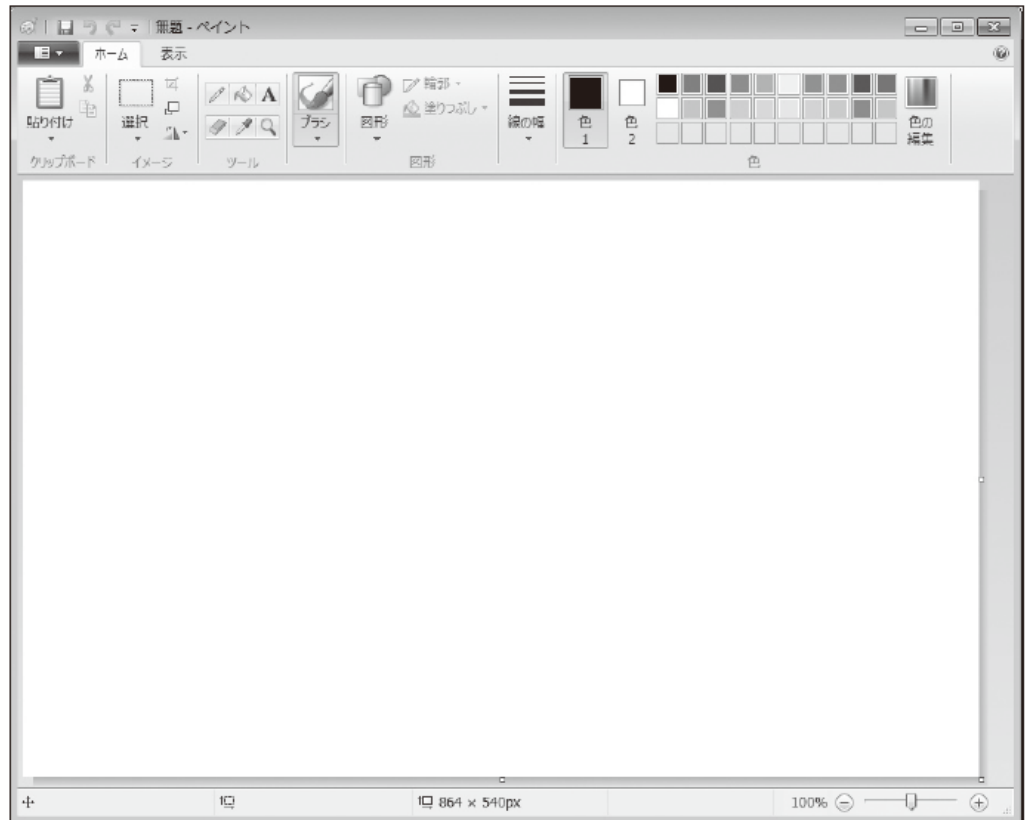
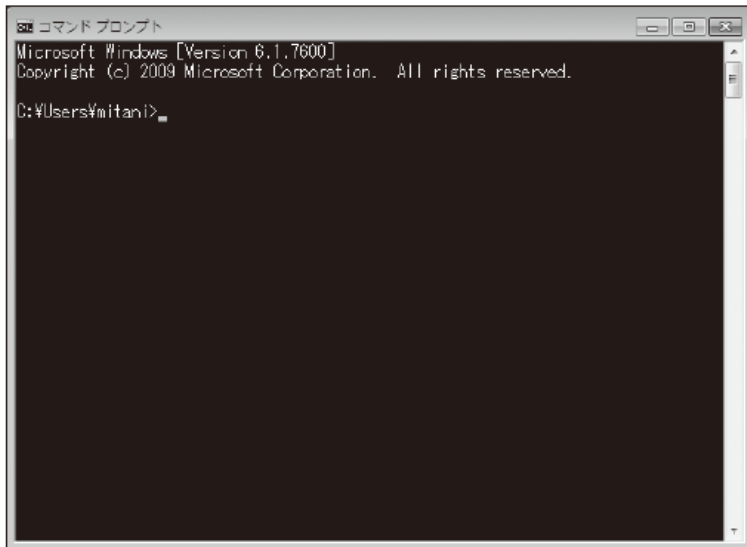
`File` クラスのフォルダ操作のための主なメソッド

<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 JavaFXによる 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



# JavaFXライブラリ

---

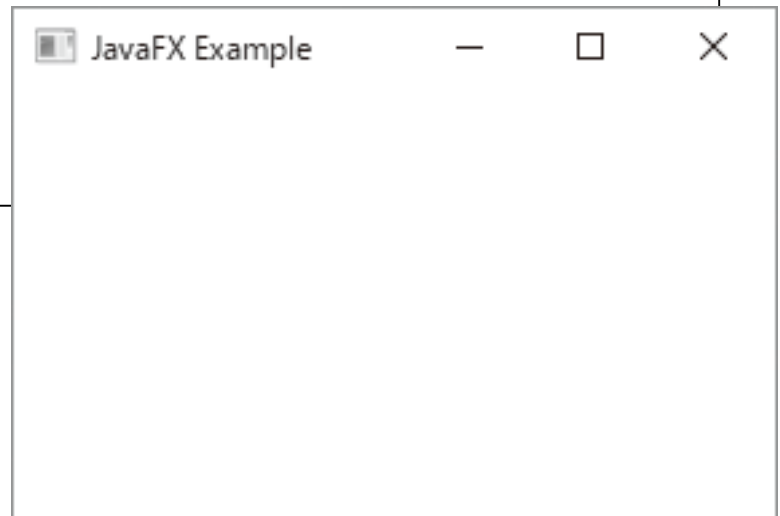
ウィンドウ、ボタン、テキスト入力ボックスなどを扱うクラス群を集めたライブラリ

各部品をコントロールと呼ぶ。

GUI アプリケーションを簡単に作ることができる。

# はじめてのアプリケーションの作成

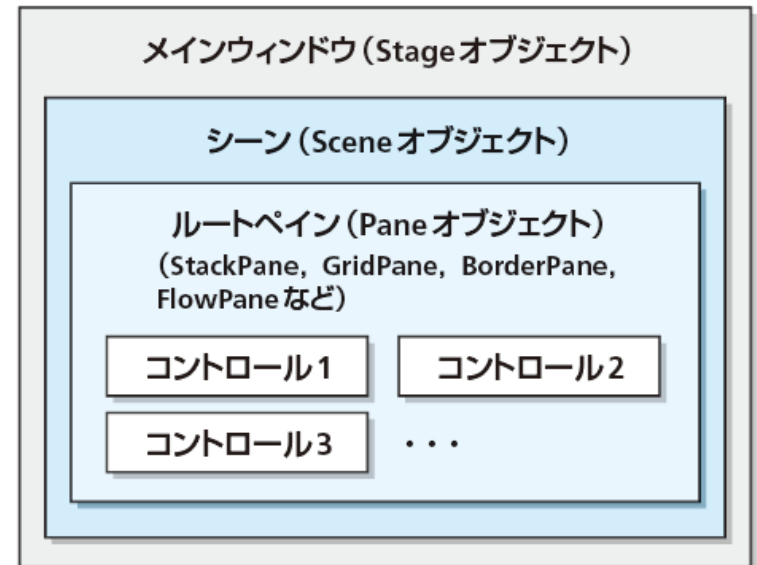
```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleApplication extends Application {
    public void start(Stage stage) {
        stage.setTitle("JavaFX Example");
        stage.setWidth(300);
        stage.setHeight(200);
        stage.show();
    }
    public static void main(String[] args)
        launch();
    }
}
```



Applicationを継承したクラスでは、start メソッドをオーバーライドする。引数で受け取るステージに対して、アプリケーションの表示に関する設定を行う。

# コントロールの配置

- GUI アプリケーションを構成する部品をコントロールと呼ぶ。
- コントロールはペインと呼ばれるオブジェクトの上に配置できる。
- ペインはシーンを介してメインウィンドウに配置される。





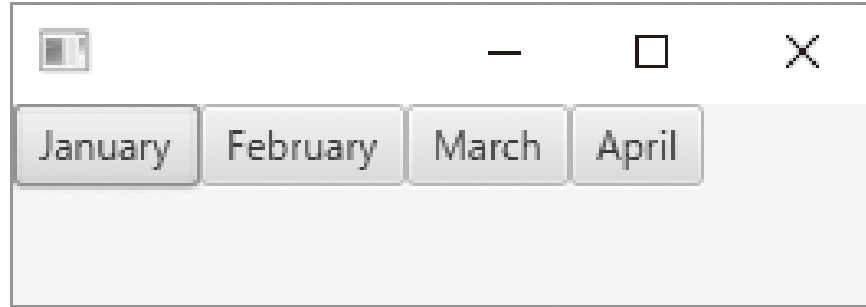
# コントロール生成の流れ

---

**start** メソッドにおいて次の処理をする。

1. ステージに対して、ウィンドウのタイトル、サイズなどの設定を行う
2. ボタン、チェックボックスなどのコントロールを生成する
3. ペインを生成し、2. で生成したコントロールを配置する
4. 3. で生成したペインをもとにシーンを生成する
5. ステージに4. で生成したシーンを配置する
6. ステージの**show**メソッドでアプリケーションウィンドウを表示する

# コントロール生成の例



```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.stage.Stage;

public class LayoutExample extends Application {
    public void start(Stage stage) {
        // (1) メインウィンドウの設定
        stage.setWidth(280);
        stage.setHeight(100);
        // (2) コントロールの生成
        Button button1 = new Button("January");
        (他のボタンについては省略)
```

# コントロール生成の例(続き)

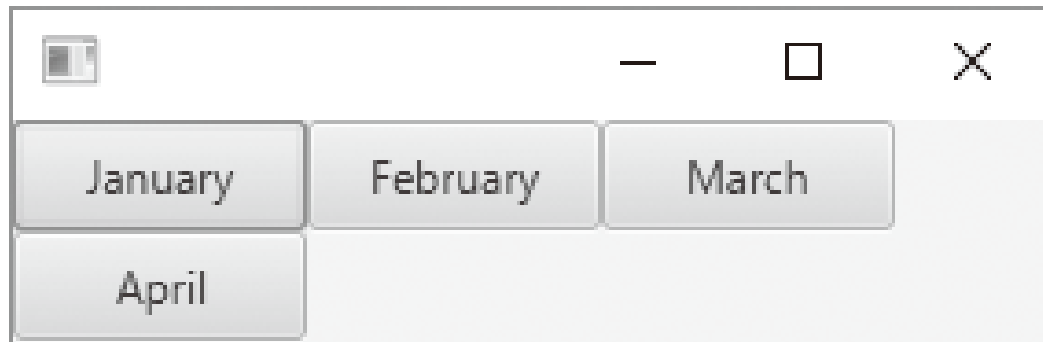
---

```
// (3) ルートペインを生成し、コントロールを配置
FlowPane root = new FlowPane();
root.getChildren().add(button1);
(他のボタンについては省略)
// (4) ルートペインを元にシーンを生成
Scene scene = new Scene(root);
// (5) ステージにシーンを配置
stage.setScene(scene);
// (6) アプリケーションウィンドウを表示
stage.show();
}
public static void main(String[] args) {
    launch();
}
}
```

# コントロールのサイズの指定

setSizeメソッドを使う。

```
button1.setSize(80, 30);  
button2.setSize(80, 30);  
button3.setSize(80, 30);  
Button4.setSize(80, 30);
```



※ウィンドウサイズに対して横1列で収まらない場合は、配置場所を下げて左端からまた並べる。

# 様々なペインクラスとコントロールの配置

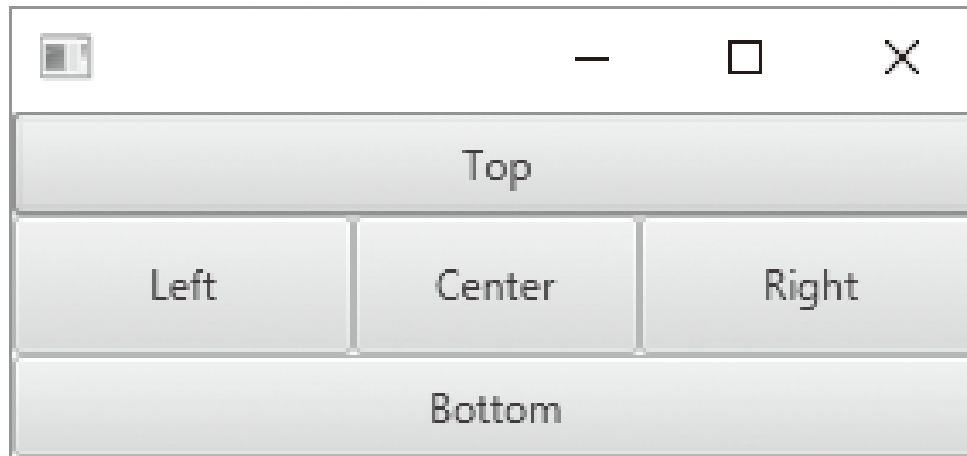
---

- ボーダーペイン
- グリッドペイン
- ボックスペイン
- Etc.

# ボーダーペイン

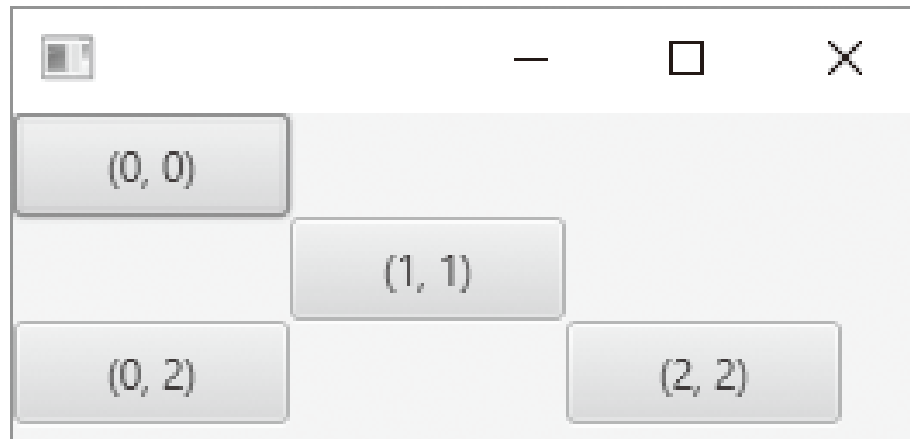
---

```
BorderPane root = new BorderPane();  
root.setTop(button1);  
root.setLeft(button2);  
root.setCenter(button3);  
root.setRight(button4);  
root.setBottom(button5);
```



# グリッドペイン

```
GridPane root = new GridPane();  
GridPane.setConstraints(button1, 0, 0);  
GridPane.setConstraints(button2, 1, 1);  
GridPane.setConstraints(button3, 2, 2);  
GridPane.setConstraints(button4, 0, 2);  
root.getChildren().addAll(button1, button2, button3, button4);
```

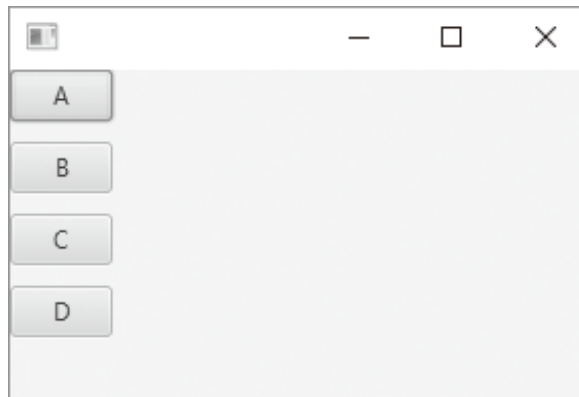


# ボックススペイン

```
HBox root = new HBox(10);  
root.getChildren().addAll(button1, button2, button3, button4);
```



```
VBox root = new VBox(10);  
root.getChildren().addAll(button1, button2, button3, button4);
```





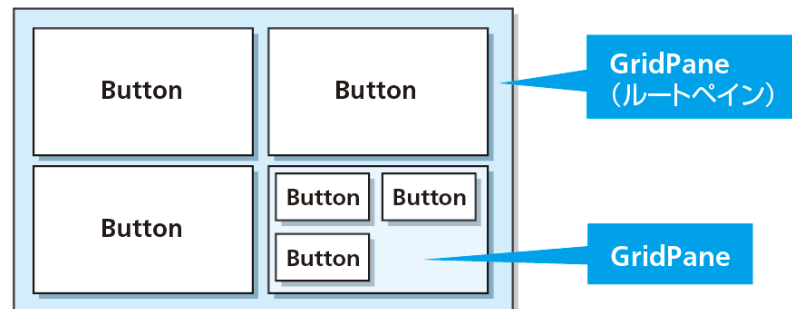
# 複数のペインの組み合わせ

```
GridPane subPane = new GridPane();
GridPane.setConstraints(button4, 0, 0);
GridPane.setConstraints(button5, 1, 0);
GridPane.setConstraints(button6, 0, 1);

subPane.getChildren().addAll(button4, button5, button6);

GridPane root = new GridPane();
GridPane.setConstraints(button1, 0, 0);
GridPane.setConstraints(button2, 1, 0);
GridPane.setConstraints(button3, 0, 1);
GridPane.setConstraints(subPane, 1, 1);

root.getChildren().addAll(button1, button2, button3, subPane);
```



# 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.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。



# イベント処理の例

```
Button button = new Button("ボタン");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("ボタンが押されました");
    }
});
```

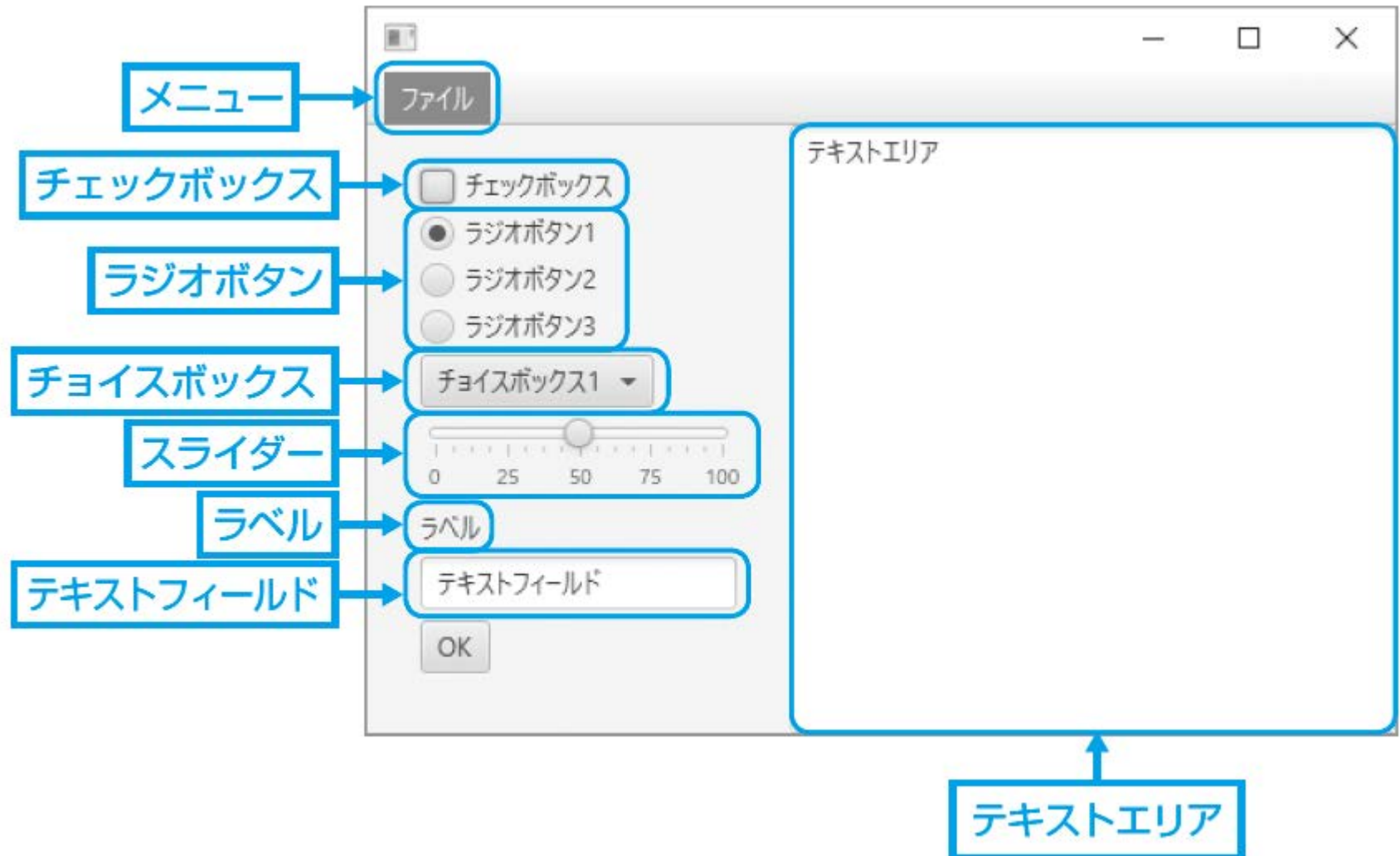
↓ ラムダ式による簡略化も可能

```
Button button = new Button("ボタン");
button.setOnAction(event -> System.out.println("ボタンが押されました"));
```



クリックすると「ボタンが押されました」とコンソールに出力される。

# JavaFXのコントロール



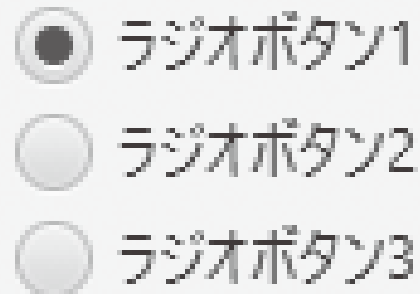
# JavaFXのコントロール

## チェックボックス



```
CheckBox checkBox = new CheckBox("チェックボックス");
```

## ラジオボタン



```
RadioButton radioButton1 = new RadioButton("ラジオボタン1");  
RadioButton radioButton2 = new RadioButton("ラジオボタン2");  
RadioButton radioButton3 = new RadioButton("ラジオボタン3");  
ToggleGroup toggleGroup = new ToggleGroup();  
radioButton1.setToggleGroup(toggleGroup);  
radioButton2.setToggleGroup(toggleGroup);  
radioButton3.setToggleGroup(toggleGroup);
```

# JavaFXのコントロール

## チョイスボックス

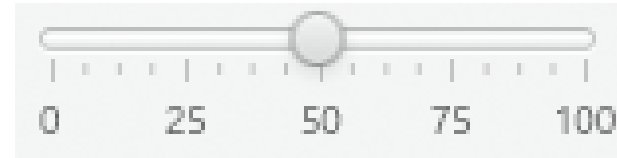
✓ チョイスボックス1

チョイスボックス2

チョイスボックス3

```
ChoiceBox<String> choiceBox = new ChoiceBox<>();  
choiceBox.getItems().addAll("項目1", "項目2", "項目3");  
choiceBox.setValue("項目1");
```

## スライダー



```
Slider slider = new Slider(0, 100, 50);  
slider.setShowTickMarks(true);  
slider.setShowTickLabels(true);
```

## スライダー操作時のイベント処理

```
slider.setOnMouseClicked(event -> System.out.println(slider.getValue()));
```

# JavaFXのコントロール


---

ラベル



```
Label label = new Label("ラベル");
```

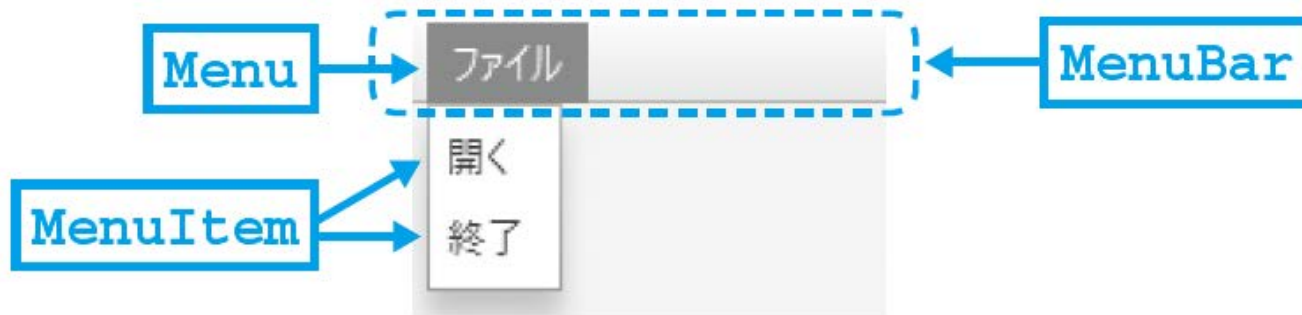
テキストフィールドとテキストエリア



```
TextField textField = new TextField("テキストフィールド");  
TextArea textArea = new TextArea("テキストエリア");
```

# JavaFXのコントロール

## メニュー



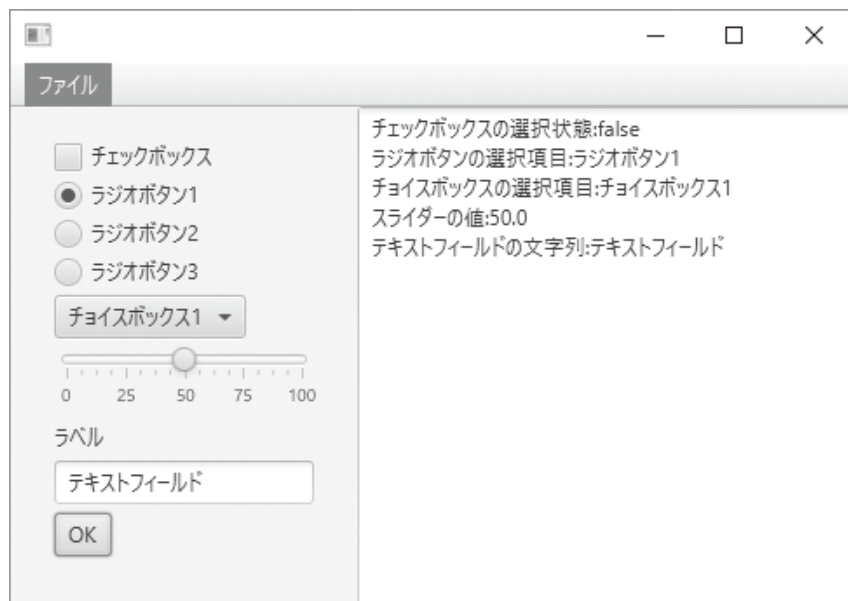
```
MenuBar menuBar = new MenuBar();  
Menu fileMenu = new Menu("ファイル");  
MenuItem menuOpen = new MenuItem("開く");  
MenuItem menuExit = new MenuItem("終了");  
fileMenu.getItems().add(menuOpen);  
fileMenu.getItems().add(menuExit);  
menuBar.getMenus().add(fileMenu);
```

### クリック時のイベント処理

```
menuExit.setOnAction(event -> System.exit(0));
```



# 複数のコントロールの組み合わせ



ほかにも数多くのコントロールがある。基本的に以下の手順で扱う。

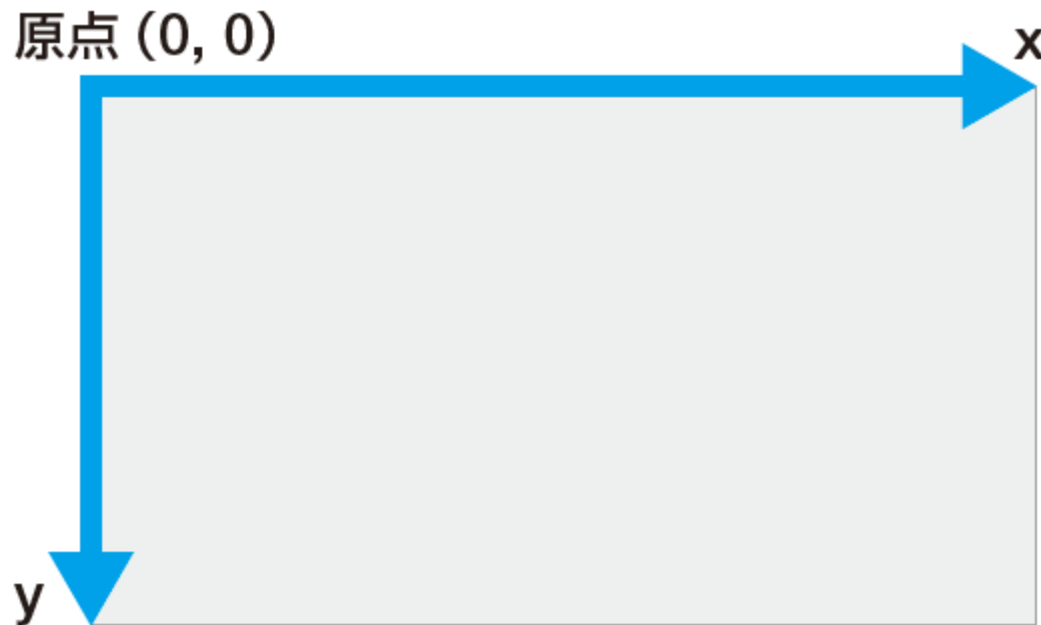
1. コントロールを生成。
2. 必要に応じて、コントロールにイベントハンドラを登録。
3. コントロールをペインの上に配置。

## 第9章

# グラフィックスとマウスイベント

# 座標系

JavaFXでは、原点(0, 0)が描画できる範囲の左上隅で、**x**座標軸は右方向、**y**座標軸が下方向になる。

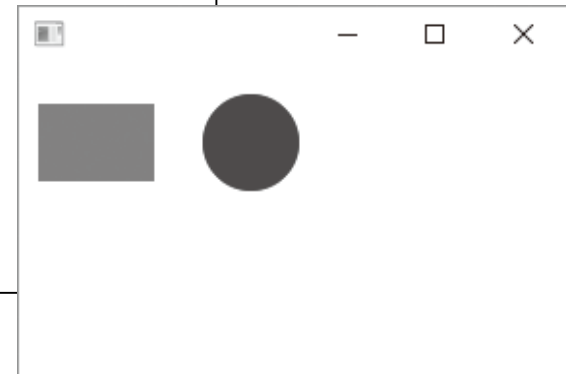


# シェイプ

- `javafx.scene.shape`パッケージには、円や四角、直線など、さまざまな図形を表すクラスが含まれている。
- 色の指定には`javafx.scene.paint.Color`オブジェクトを使える。

```
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
```

```
Rectangle rect = new Rectangle(10, 20, 60, 40);
rect.setFill(Color.RED);
Circle circle = new Circle(120, 40, 25);
circle.setFill(Color.BLUE);
Group root = new Group();
root.getChildren().addAll(rect, circle);
```



# キャンバス

---

- JavaFXでは、シェイプを生成する方法とは別に、キャンバス（Canvas）を用いて図形を描画する方法もある。
- たくさんの図形を描画するときに、生成するオブジェクトはキャンバス一つで済む。

キャンバスの生成

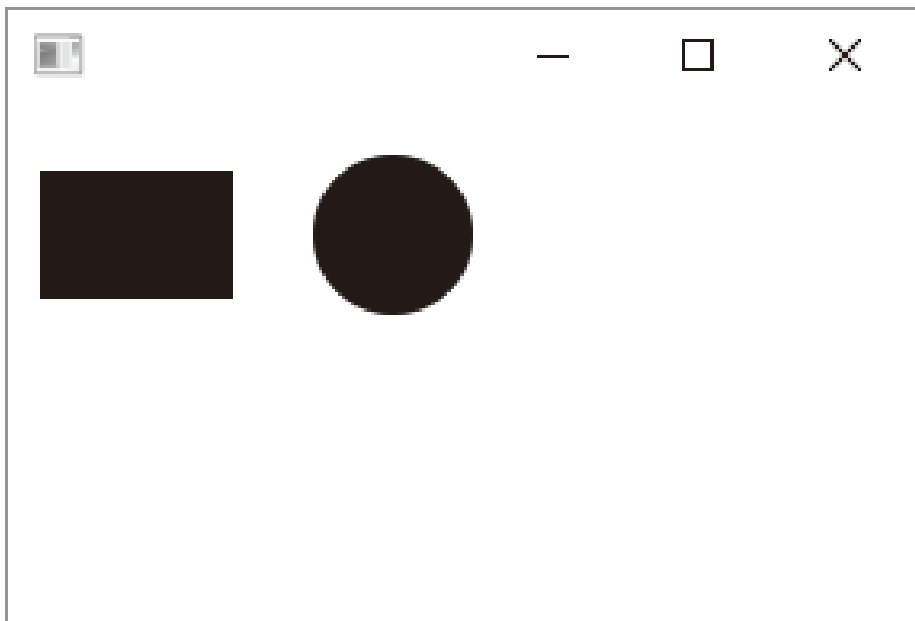
```
Canvas canvas = new Canvas(300, 200);
```

- 図形を描画するにはグラフィックスコンテキストを取得し、その描画メソッドを使う。

```
GraphicsContext gc = canvas.getGraphicsContext2D();
```

# キャンバスを用いた図形の描画例

```
Canvas canvas = new Canvas(300, 200);  
GraphicsContext gc =  
    canvas.getGraphicsContext2D();  
gc.fillRect(10, 20, 60, 40);  
gc.fillOval(95, 15, 50, 50);  
Group root = new Group();  
root.getChildren().add(canvas);
```



`fillRect`や`fillOval`の引数に  
左上隅の位置と図形の幅と高さを  
指定する。

図形を塗りつぶす色を変更するに  
はグラフィックスコンテキストの  
`setFill`メソッド、線の色を変更  
するには`setStroke`メソッドを使  
用する。

# さまざまな描画メソッド

メソッド名	説明
<code>fill</code>	現在のパスを塗りつぶす
<code>stroke</code>	現在のパスを描く
<code>strokeLine</code>	直線を描く
<code>strokePolyline</code>	折れ線を描く
<code>fillArc</code>	塗りつぶされた「円弧」を描く
<code>strokeArc</code>	線による「円弧」を描く
<code>fillOval</code>	塗りつぶされた「楕円」を描く
<code>strokeOval</code>	線による「楕円」を描く
<code>fillPolygon</code>	塗りつぶされた「多角形」を描く
<code>strokePolygon</code>	線による「多角形」を描く
<code>fillRect</code>	塗りつぶされた「長方形」を描く
<code>strokeRect</code>	線による「長方形」を描く
<code>fillText</code>	文字列を描く
<code>strokeText</code>	文字列の輪郭を描く

# パス(Path) の活用

---

- グラフィックスコンテキストの `strokePolyline` よりも柔軟な折れ線の描画が可能
- グラフィックスコンテキストの `moveTo` メソッドで始点を指定し、`lineTo` メソッドで次の点を繰り返し指定することで折れ線のパスを構築する。



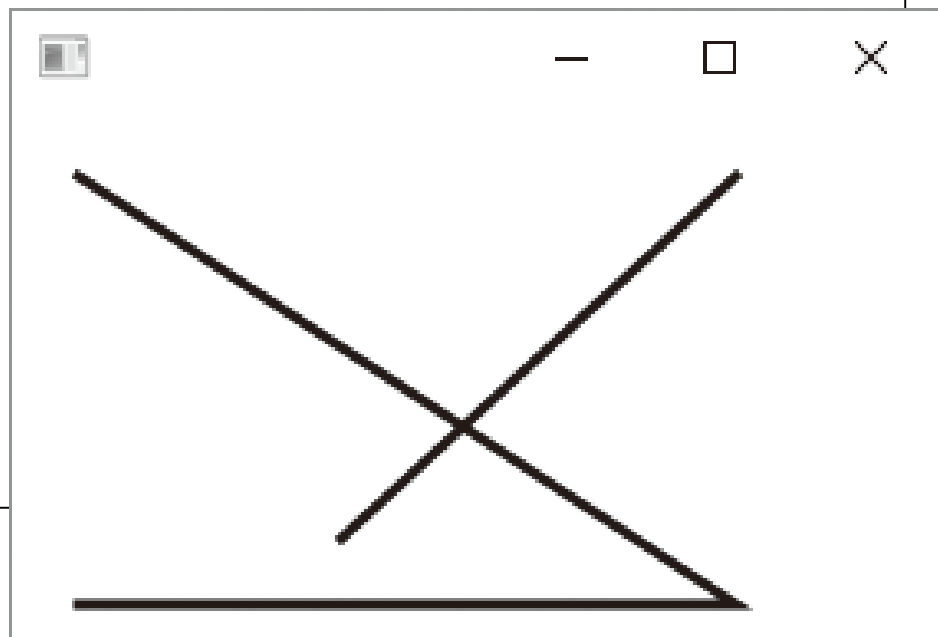
# パス(Path) の例

```
Canvas canvas = new Canvas(300, 200);  
GraphicsContext gc = canvas.getGraphicsContext2D();
```

```
gc.moveTo(20, 20);  
gc.lineTo(220, 150);  
gc.lineTo(20, 150);
```

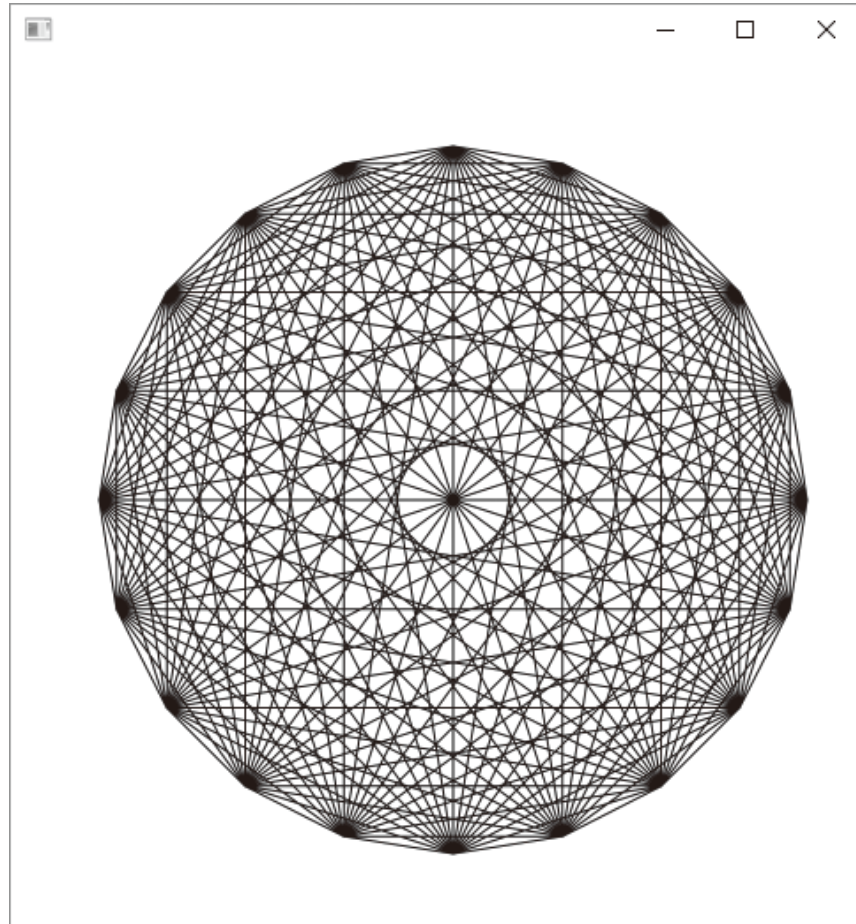
```
gc.moveTo(220, 20);  
gc.lineTo(100, 130);
```

```
gc.setLineWidth(3);  
gc.stroke();
```



# ワン・モア・ステップ

**sin**、**cos**を使った簡単な計算とパスの生成を組み合わせて、綺麗な模様を描くことができる。



# 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。
- キャンバスは `javafx.scene.Node` クラスを継承していて、この `Node` クラスには、マウス操作に対するイベントハンドラを設定するメソッドがある。

メソッド名	イベントが発生するタイミング
<code>setOnMouseClicked</code>	マウスクリックされたとき
<code>setOnMouseEntered</code>	マウスカーソルがクライアント領域に入ったとき
<code>setOnMouseExited</code>	マウスカーソルがクライアント領域から出たとき
<code>setOnMouseDragged</code>	マウスドラッグされたとき
<code>setOnMouseMoved</code>	マウスカーソルが動いたとき
<code>setOnMousePressed</code>	マウスボタンが押されたとき
<code>setOnMouseReleased</code>	押された状態だったマウスボタンが解放されたとき

# マウスイベントの例

---

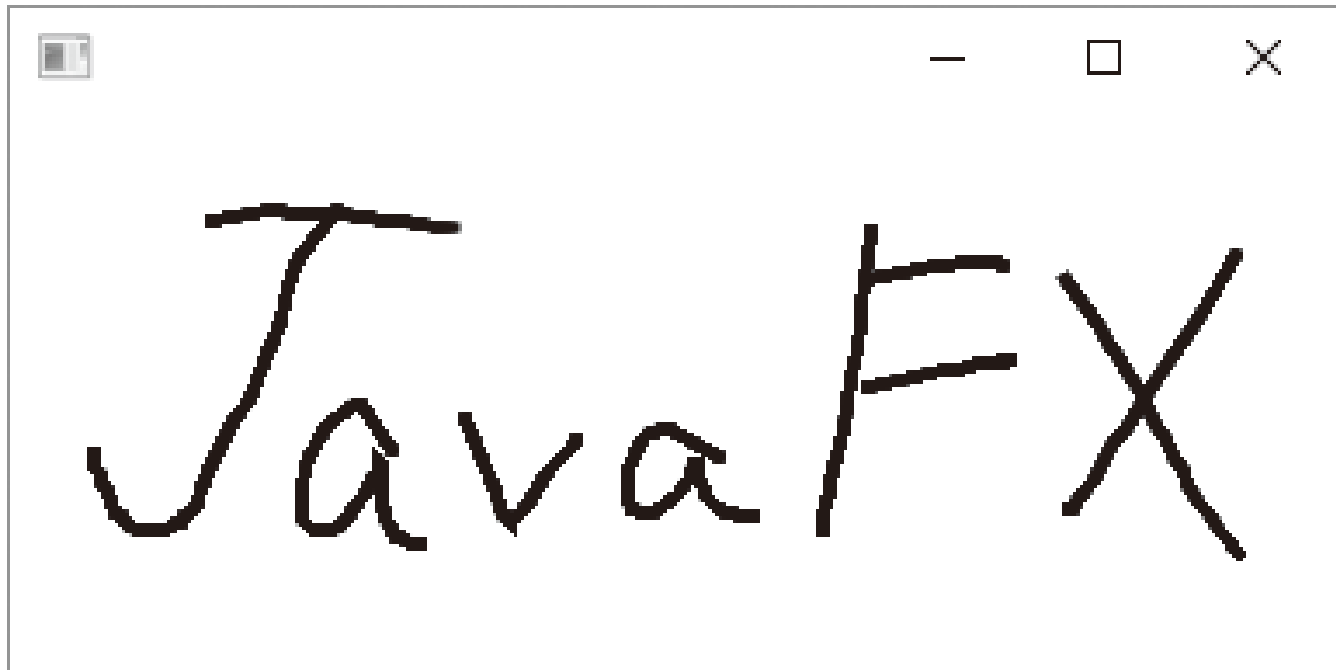
クリックされた座標値を出力するコード

```
Canvas canvas = new Canvas(300, 200);
canvas.setOnMouseClicked(new EventHandler<MouseEvent>()
{
    public void handle(MouseEvent event) {
        System.out.println(event.getX() + ", " + event.getY());
    }
});
```

# マウスイベントの例

---

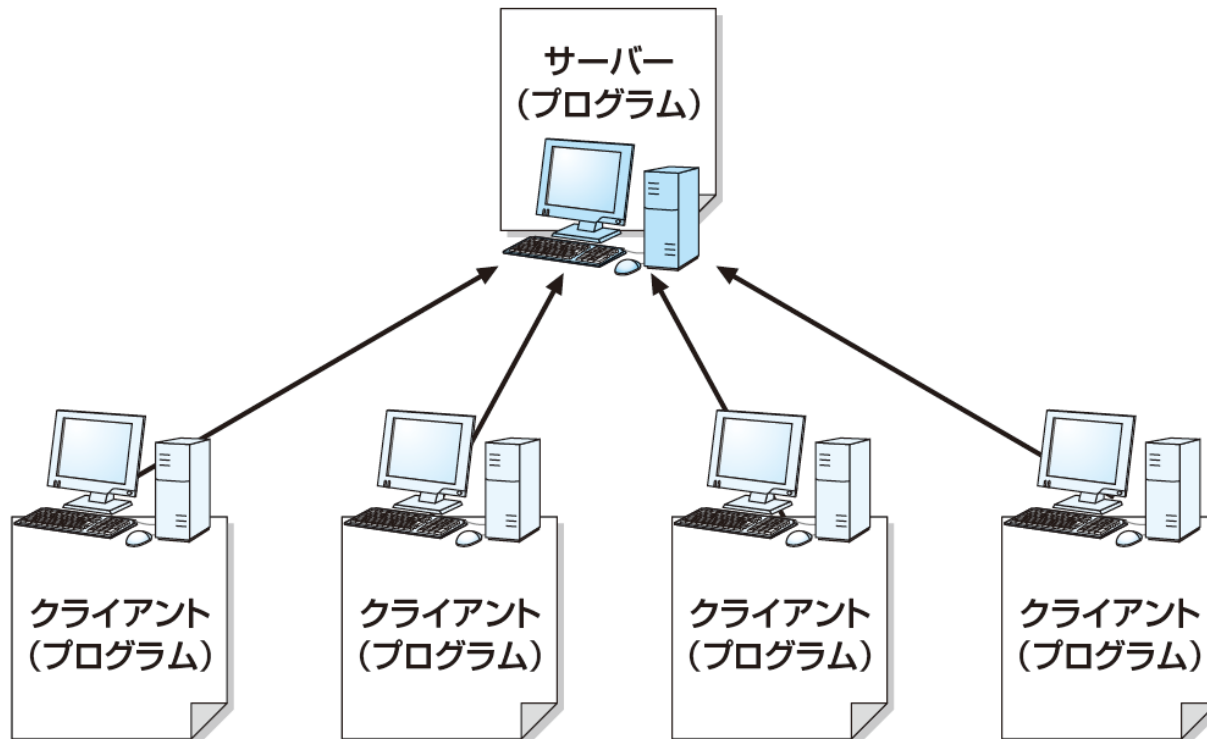
マウスイベント処理と、グラフィックス描画を組み合わせることで、簡単な「お絵かきアプリケーション」を作ることができる。



# 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



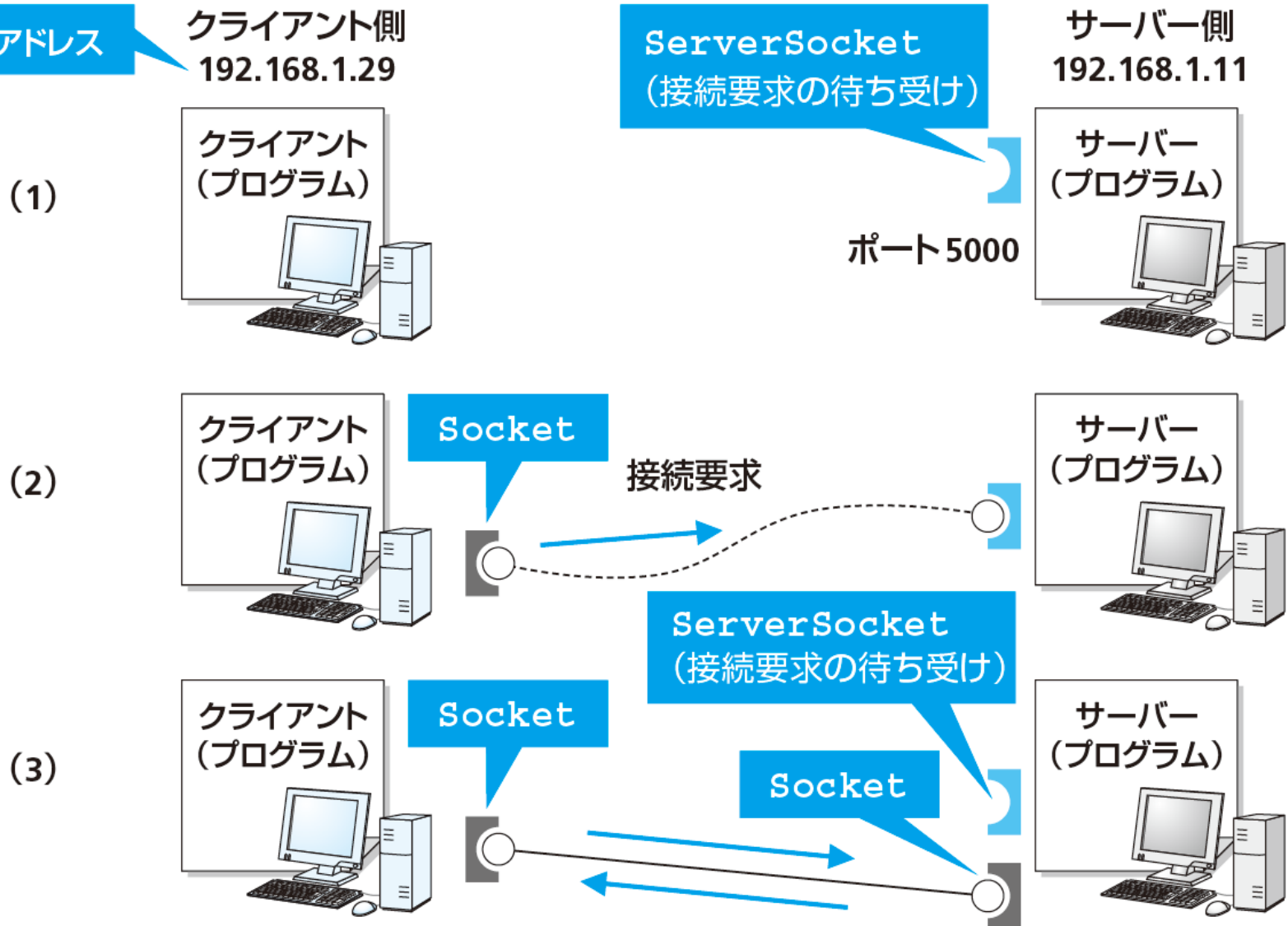
# IPアドレスとポート番号

---

- 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

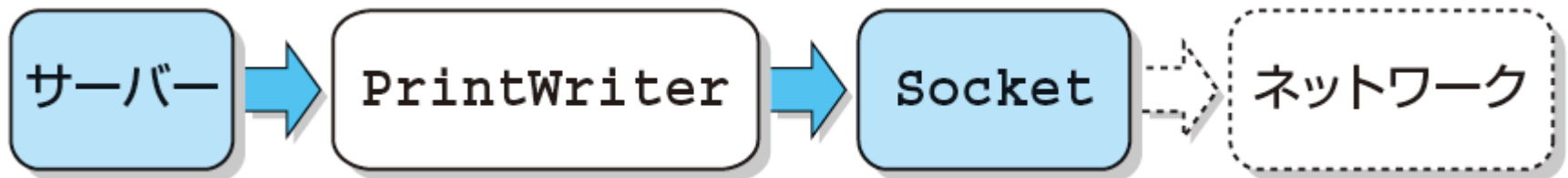


# ServerSocket と Socket



# 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

② `Socket socket = new Socket("127.0.0.1", 5000);`  
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





④ `BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));`  
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

## サーバー

① `ServerSocket serverSocket = new ServerSocket(5000);`  
5000番ポートで接続要求を待ち受ける

③ `Socket socket = serverSocket.accept();`  
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

④ `PrintWriter writer = new PrintWriter(socket.getOutputStream());`  
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

クライアント	サーバー
<div data-bbox="537 254 562 554"></div> <div data-bbox="92 572 680 715"><p>⑥ <code>String message = reader.readLine();</code> ストリームから文字を受け取る</p></div> <div data-bbox="537 729 562 953"></div> <div data-bbox="92 972 562 1068"><p>⑧ <code>reader.close();</code> ストリームを閉じる</p></div>	<div data-bbox="1416 254 1441 329"></div> <div data-bbox="1010 344 1798 486"><p>⑤ <code>writer.println("こんにちは。私はサーバーです。");</code> ストリームに文字列を出力する</p></div> <div data-bbox="1416 501 1441 782"></div> <div data-bbox="1010 796 1479 892"><p>⑦ <code>writer.close();</code> ストリームを閉じる</p></div>

# 第11章 一歩進んだ Javaプログラミング

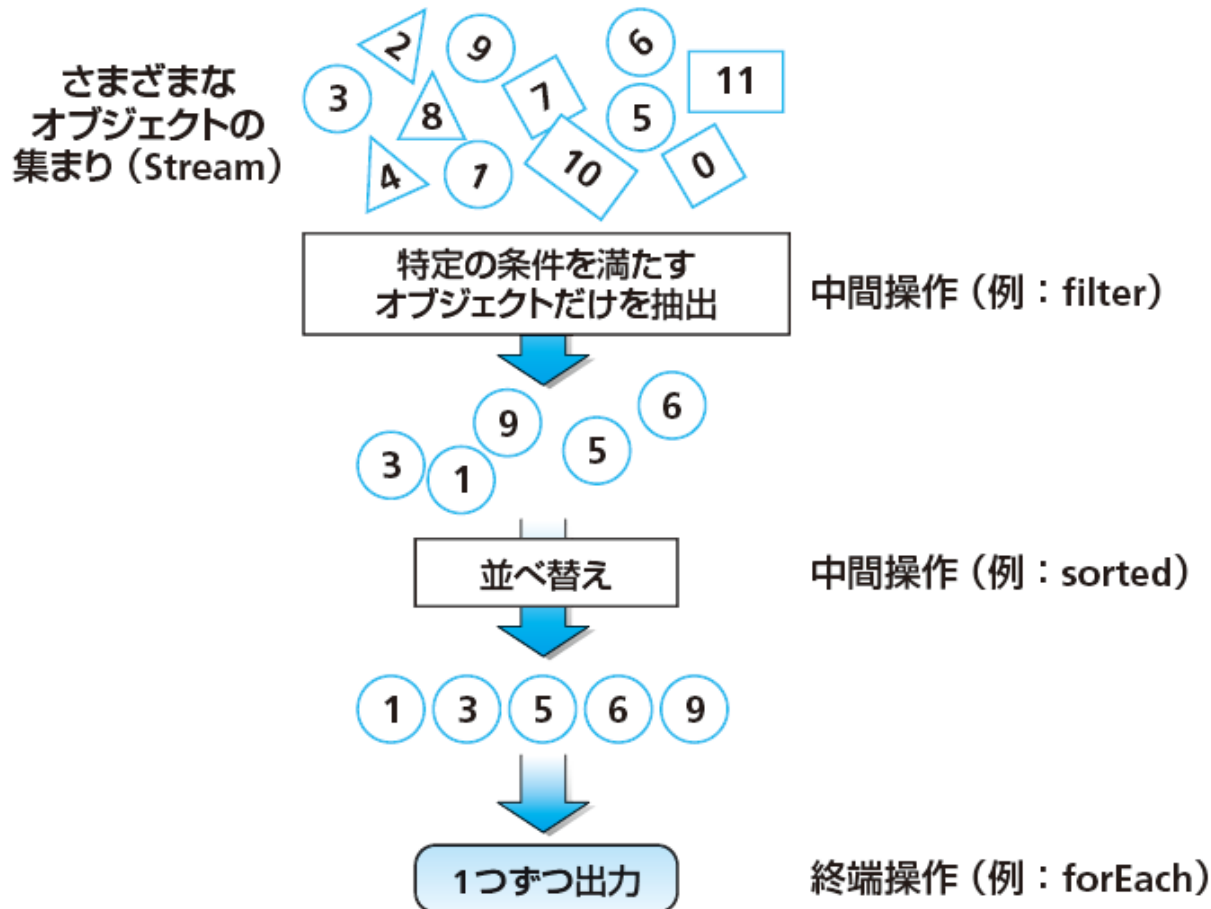
# コレクションとストリーム

---

- 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することができる。

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