JAVA初心者向け研修資料

基本構文からフレームワークまで

```
// Java Learning Path for Beginners
```

2023年

目次

JAVAとは

プログラミング言語の基礎知識

概要

Javaは1995年にSun Microsystems社（現Oracle）によって開発されたオブジェクト指向プログラミング言語です。強力な汎用言語として、さまざまなプラットフォームで実行できるアプリケーションの開発に使用されています。

デスクトップアプリ　モバイルアプリ　サーバーサイド

IoTデバイス　エンタープライズシステム　ゲーム開発

プラットフォーム独立性

「Write Once, Run Anywhere」の理念。一度書いたコードはどのプラットフォームでも動作。

オブジェクト指向

すべてがクラスとオブジェクトで構成され、コードの再利用と保守が容易。

堅牢性と安全性

強力な例外処理、メモリ管理、セキュリティ機能を備えている。

コミュニティとエコシステム

豊富なライブラリ、フレームワーク、活発な開発者コミュニティ。

JAVAの歴史と特徴

進化し続ける25年以上の歴史

1991年

James Goslingらによって「Oak」という名前で開発開始

1995年

正式に「Java」と命名され、Java 1.0がリリース

1998-2000年

Java 2 Platform (J2SE, J2EE, J2ME)の誕生

2006-2011年

Java SE 6, 7がリリース

2014年

Java SE 8リリース - ラムダ式やStream APIの導入

2017年〜現在

6ヶ月ごとの新リリースサイクル開始 (Java 9〜)

Java 9 | Java 10 | **Java 11 (LTS)** | Java 12-16 | **Java 17 (LTS)** | Java 18-21

Write Once, Run Anywhere (WORA)

Javaコードは一度コンパイルすれば、JVMが搭載されているあらゆるプラットフォームで実行可能。プラットフォーム非依存性はJavaの最大の強みの一つ。

Java Virtual Machine (JVM)

バイトコードを実行する仮想マシン。メモリ管理、ガベージコレクション、セキュリティ機能などを提供し、Javaプログラムの安定した実行環境を実現。

エンタープライズ向け機能の充実

マルチスレッディング、分散コンピューティング、ネットワーキングなどの機能が標準で提供され、大規模システムの開発に適している。

豊富なエコシステム

Spring、Hibernate、JavaEEなど多くのフレームワークと巨大なコミュニティがあり、様々な用途のライブラリが利用可能。大企業から長期にわたり採用され続けている。

JAVA開発環境のセットアップ

効率的な開発のための準備

JDKのインストール

JDK（Java Development Kit）は、Javaアプリケーションの開発に必要な開発ツールセット

1. **JDKとJREの違いを理解する**
   JDK = 開発ツール + JRE（実行環境）
   JRE = Javaアプリ実行のみに必要
2. **JDKをダウンロードする**
   公式サイト: `https://www.oracle.com/java/technologies/downloads/`
   または OpenJDK: `https://adoptium.net/`
3. **インストールとPATH設定**
   インストーラに従い、環境変数の設定を確認
   `JAVA_HOME` と `PATH` の設定が必要
4. **インストール確認**
   コマンドプロンプトで `java -version` を実行

> **ヒント:** 初心者の方はJava SE（Standard Edition）の最新LTSバージョン（Java 17 または Java 21 LTS）をお勧めします。

IDEの選択

IDE（統合開発環境）はJava開発の生産性を大幅に向上させます。代表的なIDEを紹介します。

Eclipse

老舗のJava IDE。プラグインが豊富で拡張性が高い

初心者向け

IntelliJ IDEA

高機能で生産性に優れた開発環境。Community版は無料

高機能

VS Code

軽量なエディタ。Java拡張機能をインストールして使用

軽量

プロジェクト作成の基本

1. **新規プロジェクト作成**
   IDEの「新規プロジェクト」オプションを選択
   「Java Project」または「Maven Project」を選択
2. **プロジェクト設定**
   プロジェクト名、保存場所、JDKバージョンを設定
3. **クラスファイル作成**
   src/main/javaディレクトリに新規クラスを作成
4. **Hello World作成**

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

JAVA基本構文：変数と型

データを保存し操作するための基礎

プリミティブ型（基本データ型）

整数型

byte (8ビット)

short (16ビット)

int (32ビット)

long (64ビット)

```
int age = 25;
```

浮動小数点型

float (32ビット)

double (64ビット)

```
double price = 199.99;
```

文字型

char (16ビット、Unicode)

```
char grade = 'A';
```

論理型

boolean (true/false)

```
boolean isActive = true;
```

型	サイズ	デフォルト値	値の範囲
byte	8ビット	0	-128 ～ 127
short	16ビット	0	-32,768 ～ 32,767
int	32ビット	0	-2^31 ～ 2^31-1
long	64ビット	0L	-2^63 ～ 2^63-1

参照型

プリミティブ型以外はすべて参照型です。オブジェクトへの参照（メモリアドレス）を格納します。

String

文字列を扱うクラス

```
String message = "Hello";
```

配列

同じ型の複数の値

```
int[] numbers = {1, 2, 3};
```

クラス

ユーザー定義のデータ型

```
Person person = new Person();
```

インターフェース

メソッドの契約

```
List list = new ArrayList<>();
```

参照型の特徴

nullを持つことができる
ヒープメモリに格納される
ガベージコレクションの対象
メソッドを持つことができる

変数の宣言と初期化

```java
// 変数の宣言
int count;
double temperature;

// 宣言と同時に初期化
int count = 5;
double temperature = 36.5;
String name = "Java";

// 定数の宣言（値の変更不可）
final double PI = 3.14159;
```

命名規則： 変数名はキャメルケース（最初の単語は小文字、以降の単語は大文字から始める）が一般的です。例：firstName, totalAmount

型変換

暗黙的型変換（自動的）： 小さな型から大きな型への変換

```java
byte b = 10;
int i = b; // byteからintへの自動変換
```

byte→short→int→long→float→double

明示的型変換（キャスト）： 大きな型から小さな型への変換

```java
double d = 10.5;
int i = (int) d; // doubleからintへの明示的変換
// i は 10 になる（小数点以下は切り捨て）
```

注意： 明示的型変換では情報が失われる可能性があります。例えば、double から int へ変換すると小数点以下が切り捨てられます。

JAVA基本構文：演算子

値を操作し計算を行うための基礎

算術演算子

演算子	説明	例
+	加算	10 + 5 = 15
-	減算	10 - 5 = 5
*	乗算	10 * 5 = 50
/	除算	10 / 5 = 2
%	剰余（余り）	10 % 3 = 1
++	インクリメント	x++; または ++x;
--	デクリメント	x--; または --x;

```
int a = 10;
int b = 3;
int sum = a + b; // 13
int diff = a - b; // 7
int product = a * b; // 30
int quotient = a / b; // 3（整数の除算）
int remainder = a % b;// 1
```

注意： ++xと x++の違い：前置と後置。前置（++x）は値を増加させてから式を評価し、後置（x++）は式を評価してから値を増加させます。

≠ 比較演算子

等価 ==

左右の値が等しいかどうか

```
5 == 5 // true
5 == 8 // false
```

不等価 !=

左右の値が異なるかどうか

```
5 != 8 // true
5 != 5 // false
```

より大きい >

左の値が右の値より大きいかどうか

```
8 > 5 // true
5 > 8 // false
```

より小さい <

左の値が右の値より小さいかどうか

```
5 < 8 // true
8 < 5 // false
```

以上 >=

左の値が右の値以上かどうか

```
8 >= 5 // true
5 >= 5 // true
```

以下 <=

左の値が右の値以下かどうか

```
5 <= 8 // true
5 <= 5 // true
```

論理演算子

演算子	説明	例
&&	論理AND	(x > 5) && (y < 10) 両方がtrueの場合のみtrue
||	論理OR	(x > 5) || (y < 10) どちらかがtrueならtrue
!	論理NOT	!(x > 5) 条件を反転させる

```
boolean a = true;
boolean b = false;
boolean result1 = a && b; // false
boolean result2 = a || b; // true
boolean result3 = !a; // false
```

ビット演算子

演算子	説明
&	ビットAND
|	ビットOR
^	ビットXOR
~	ビット反転
<<	左シフト
>>	右シフト（符号あり）
>>>	右シフト（符号なし）

```
int a = 5; // 101 (二進数)
int b = 3; // 011 (二進数)
5 & 3 = 1 // 001 (二進数)
5 | 3 = 7 // 111 (二進数)
5 ^ 3 = 6 // 110 (二進数)
```

= 代入演算子

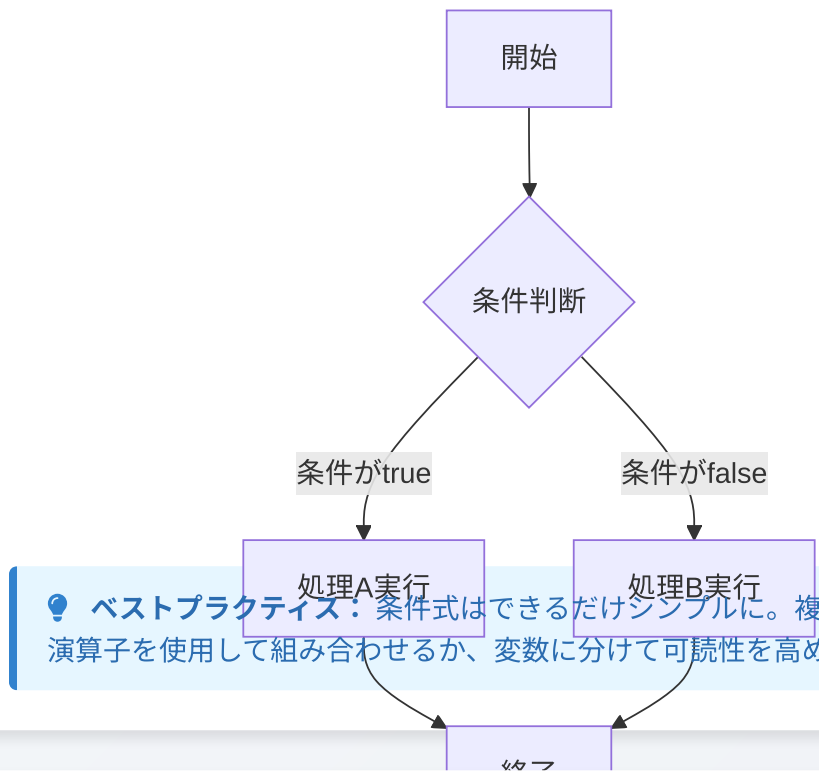
演算子	例	同等の表現
=	x = 10	x = 10
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
&=, |=, ^=, >>=, <<=, >>>=	x &= 5	x = x & 5

JAVA基本構文：制御構文（条件分岐）

プログラムの流れを制御する基本構造

条件分岐の基本

条件に基づいてプログラムの流れを変更するための制御構文です。



ベストプラクティス： 条件式はできるだけシンプルに。複雑な場合は論理演算子を使用して組み合わせるか、変数に分けて可読性を高めましょう。

if文とif-else文

```
// 基本的なif文
if (条件式) {
    // 条件がtrueの時に実行する処理
}

// if-else文
if (条件式) {
    // 条件がtrueの時に実行する処理
} else {
    // 条件がfalseの時に実行する処理
}
```

単一文の場合、波括弧{}は省略可能ですが、可読性と将来的な修正のしやすさのために常に記述することをお勧めします。

```
int age = 18;

if (age >= 20) {
    System.out.println("成人です");
} else {
    System.out.println("未成年です");
}
```

if-else if-else文

```
if (条件式1) {
    // 条件1がtrueの時に実行
} else if (条件式2) {
    // 条件1がfalseで条件2がtrueの時に実行
} else if (条件式3) {
    // 条件1, 2がfalseで条件3がtrueの時に実行
} else {
    // すべての条件がfalseの時に実行
}
```

```
int score = 85;
String grade;

if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else if (score >= 60) {
    grade = "D";
} else {
    grade = "F";
}
```

switch文

```
switch (式) {
    case 値1:
        // 値1に一致した場合の処理
        break;
    case 値2:
        // 値2に一致した場合の処理
        break;
    default:
        // どの値にも一致しない場合の処理
}
```

```
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "月曜日";
        break;
    case 2:
        dayName = "火曜日";
        break;
    case 3:
        dayName = "水曜日";
        break;
    default:
        dayName = "不明";
}
```

注意： breakを忘れるとフォールスルー(fall-through)が発生し、次のcaseも実行されます。これは意図的に使うこともありますが、多くの場合はバグの原因となります。

使い分け

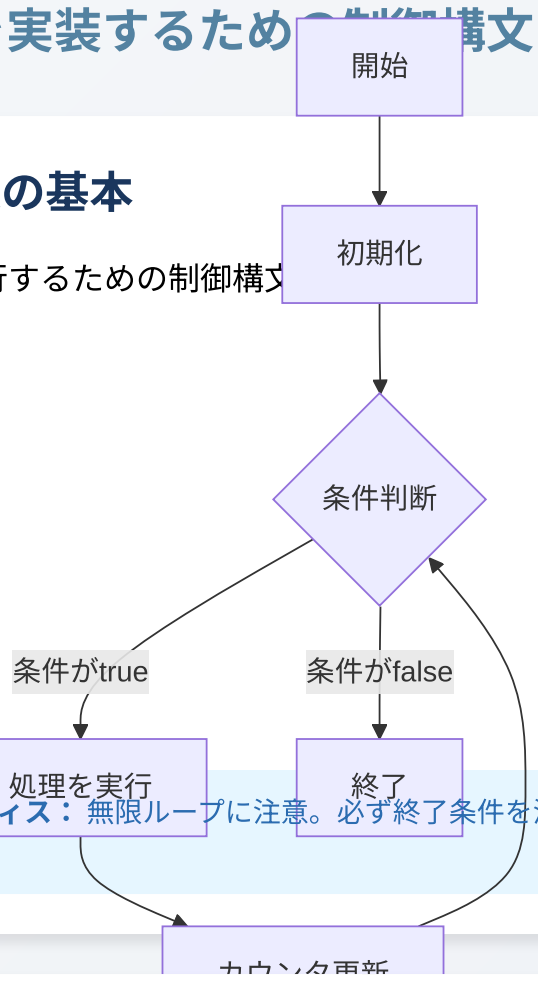
構文	使用場面
if文	単一の条件チェック
if-else文	二者択一の条件分岐
if-else if-else文	複数の条件を順番にチェック
switch文	一つの変数に対して複数の値と比較する場合

JAVA基本構文：ループ構文

繰り返し処理を実装するための制御構文

ループ構文の基本

処理を繰り返し実行するための制御構文



ベストプラクティス： 無限ループに注意。必ず終了条件を満たす仕組みを持たせましょう。

for文

```
// 基本的なfor文
for (初期化; 条件式; 更新式) {
    // 繰り返し実行する処理
}
```

```
// 1から10までの数値を表示する例
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

拡張for文（for-each）

```
// 拡張for文の基本形
for (要素の型 変数名 : 配列またはコレクション) {
    // 各要素に対して実行する処理
}
```

```
// 配列の全要素を表示する例
String[] fruits = {"りんご", "バナナ", "オレンジ"};
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

while文

```
// while文の基本形
while (条件式) {
    // 条件がtrueの間、繰り返し実行する処理
}
```

```
// 1から5までの数値を表示する例
int count = 1;
while (count <= 5) {
    System.out.println(count);
    count++;
}
```

条件を最初にチェックするため、条件がfalseの場合は1度も実行されません。

do-while文

```
// do-while文の基本形
do {
    // 少なくとも1回は実行される処理
} while (条件式);
```

```
// 少なくとも1回は実行される例
int num = 10;
do {
    System.out.println("現在の値: " + num);
    num--;
} while (num > 0);
```

条件を後でチェックするため、最低1回は処理が実行されます。

使い分けとフロー制御

構文	使用場面
for文	繰り返し回数が事前に分かっている場合
拡張for文	配列やコレクションのすべての要素を処理する場合
while文	条件が満たされている間繰り返す場合
do-while文	少なくとも1回は処理を実行したい場合

ループ制御文

```
break;    // ループを即座に終了
continue; // 現在の繰り返しをスキップし、次の繰り返しへ
```

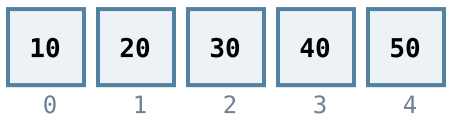
複雑なループ制御よりも、シンプルな条件設計を心がけましょう。

JAVA基本構文：配列

同じ型のデータをまとめて扱う仕組み

配列の基本

配列とは同じデータ型の値を複数格納できるデータ構造です。



重要： Javaの配列はインデックスが0から始まります。

一次元配列の宣言と初期化

```
// 宣言のみ
int[] numbers;
String[] names;

// 宣言と同時にサイズを指定
int[] scores = new int[5]; // 5つの要素を持つ配列

// 初期化と同時に値を代入
int[] points = {10, 20, 30, 40, 50};
String[] fruits = {"りんご", "バナナ", "オレンジ"};
```

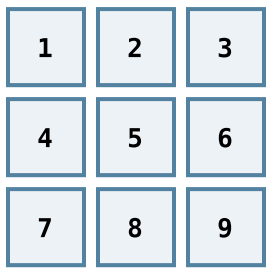
配列へのアクセスと値の変更

```
int[] numbers = {5, 10, 15, 20, 25};

// 値の取得
int secondNumber = numbers[1]; // 10が取得される

// 値の変更
numbers[2] = 100; // 3番目の要素が15から100に変わる
// 配列の長さを取得
int length = numbers.length; // 5が返される
```

ベストプラクティス： 配列の要素にアクセスする前に、必ずインデックスが有効な範囲内であることを確認しましょう。範囲外のインデックスを参照すると `ArrayIndexOutOfBoundsException` が発生します。

多次元配列

配列の要素として配列を持つことで、多次元配列を表現できます。




```
// 2次元配列の宣言と初期化
int[][] matrix = new int[3][3];

// 初期値を与えて初期化
int[][] grid = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};

// 2次元配列の要素へのアクセス
int centerValue = grid[1][1]; // 5が取得される
```

配列のループ処理

```
// 通常のfor文を使ったループ
int[] values = {10, 20, 30, 40, 50};
for(int i = 0; i < values.length; i++) {
  System.out.println(values[i]);
}

// 拡張for文（for-each）を使ったループ
for(int value : values) {
  System.out.println(value);
}
```

配列の操作・ユーティリティ

```
import java.util.Arrays;

int[] numbers = {5, 2, 9, 1, 5};

// 配列のソート
Arrays.sort(numbers); // {1, 2, 5, 5, 9}

// 文字列表現に変換
String str = Arrays.toString(numbers); // "[1, 2, 5, 5, 9]"

// 配列のコピー
int[] copied = Arrays.copyOf(numbers, numbers.length);
```

注意： 配列のサイズは作成後に変更できません。サイズ可変の配列が必要な場合は、ArrayListなどのコレクションを使用しましょう。

JAVA基本構文：メソッド

プログラムを構造化し再利用可能にする仕組み

メソッドの基本

メソッドは特定の処理をまとめて名前を付けた再利用可能なコードブロックです。

```
public int calculateSum(int a, int b)

int result = a + b;

return result;
```

```
// メソッドの基本構文
アクセス修飾子 戻り値の型 メソッド名(引数リスト) {
  // メソッドの処理
  return 戻り値; // 戻り値がある場合
}
```

引数と戻り値

```
// 引数あり・戻り値あり
public int multiply(int x, int y) {
  return x * y;
}

// 引数あり・戻り値なし
public void greet(String name) {
  System.out.println("こんにちは、" + name +
"さん！");
}

// 引数なし・戻り値あり
public String getCurrentDate() {
  return
java.time.LocalDate.now().toString();
}

// 引数なし・戻り値なし
public void showMessage() {
  System.out.println("処理が完了しました。");
}
```

重要： void は「戻り値がない」ことを示す特別な戻り値の型です。この場合、return 文は省略できます。

メソッドのオーバーロード

同じ名前で異なる引数を持つ複数のメソッドを定義できます。

sum(int a, int b)
整数2つの合計

sum(double a, double b)
小数2つの合計

sum(int... values)
可変長引数の合計

↓

引数の**型**または**数**が異なる必要があります

```
// 整数2つの加算
public int add(int a, int b) {
  return a + b;
}

// 小数2つの加算
public double add(double a, double b) {
  return a + b;
}

// 整数3つの加算
public int add(int a, int b, int c) {
  return a + b + c;
}
```

メソッドの呼び出し

```
// メソッドの定義
public class Calculator {
  public int square(int num) {
    return num * num;
  }

  public static void main(String[] args) {
    Calculator calc = new Calculator();
    int result = calc.square(5);
    System.out.println("5の2乗は: " + result);
  }
}
```

可変長引数

```
// 任意の数の整数を受け取るメソッド
public int sum(int... numbers) {
  int total = 0;
  for (int num : numbers) {
    total += num;
  }
  return total;
}

// 呼び出し例
sum(1, 2); // 3
sum(1, 2, 3); // 6
sum(1, 2, 3, 4, 5); // 15
```

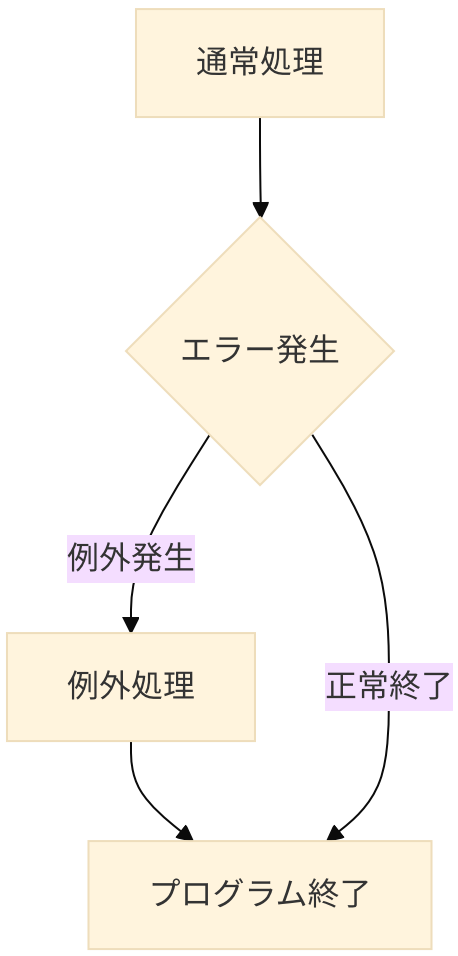
ベストプラクティス： メソッド名は動詞または動詞句で始め、その機能を明確に表す名前にしましょう。例：`calculateTotal()`, `sendEmail()`, `isValid()`

JAVA基本構文：例外処理

エラーを適切に処理して安全なプログラムを作るための仕組み

例外の基本概念

例外とは実行時に発生する予期しない状況や異常を表すオブジェクトです。




```
// 基本的な例外発生の例
int[] array = new int[3];
array[5] = 10; // ArrayIndexOutOfBoundsException発生
```

try-catch-finally構文

```
try {
  // 例外が発生する可能性のあるコード
  int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
  // 例外を処理するコード
  System.out.println("0で割ることはできません");
  e.printStackTrace(); // スタックトレースを出力
} finally {
  // 例外発生の有無にかかわらず実行されるコード
  System.out.println("処理完了");
}
```

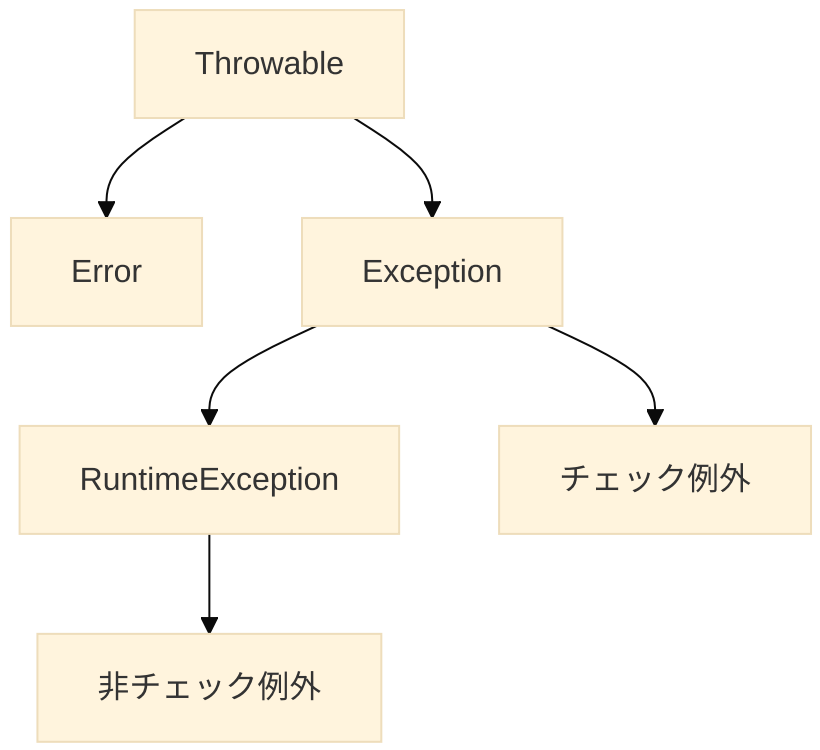
```
// 複数のcatchブロック
try {
  // 処理
} catch (ArrayIndexOutOfBoundsException e) {
  // 配列の範囲外アクセス処理
} catch (ArithmeticException e) {
  // 算術例外処理
} catch (Exception e) {
  // その他の例外処理
}
```

throws句

```
// メソッドから例外をスロー
public void readFile(String fileName) throws IOException {
  FileReader file = new FileReader(fileName);
  // ファイル処理
}
```

```
// 例外をスローしたメソッドの呼び出し
try {
  readFile("data.txt");
} catch (IOException e) {
  System.out.println("ファイル読み込みエラー");
}
```

チェック例外と非チェック例外



チェック例外

- コンパイル時に例外処理（try-catchまたはthrows）が必要
- 例: IOException, SQLException, FileNotFoundException

IOException **SQLException** **ClassNotFoundException**

非チェック例外

- RuntimeExceptionクラスとそのサブクラス
- 例外処理は任意（必須ではない）
- 例: NullPointerException, ArrayIndexOutOfBoundsException

NullPointerException **ArithmeticException** **IllegalArgumentException**

カスタム例外の作成

```
// カスタムのチェック例外
public class InsufficientFundsException extends Exception {
  public InsufficientFundsException(String message) {
    super(message);
  }
}
```

```
// カスタムの非チェック例外
public class InvalidDataFormatException extends RuntimeException {
  public InvalidDataFormatException(String message) {
    super(message);
  }
}
```

```
// カスタム例外の使用例
public void withdraw(double amount) throws InsufficientFundsException {
  if (amount > balance) {
    throw new InsufficientFundsException("残高不足");
  }
  balance -= amount;
}
```

ベストプラクティス： 例外処理で例外を隠さず、適切にログに記録しましょう。また、具体的な例外をキャッチし、一般的な例外（Exception）のみをキャッチするのは避けましょう。

JAVA基本構文：文字列操作

テキストデータを効率的に扱うための基本技術

A Stringクラスの基本

Javaでは、文字列はStringクラスのオブジェクトとして扱われます。Stringは不変（immutable）であり、一度作成された文字列の内容を変更することはできません。

```
// 文字列の宣言と初期化
String str1 = "Hello"; // リテラル形式
String str2 = new String("Hello"); // オブジェクト作成

// 文字列連結
String greeting = str1 + ", World!"; // "Hello, World!"
String greeting2 = str1.concat(", World!"); // "Hello, World!"
```

注意： Stringリテラル（"Hello"など）は文字列プール内に保存され、同じ値のリテラルは同じオブジェクトを参照します。一方、new演算子で作成されたStringは異なるオブジェクトになります。

主要なStringメソッド

メソッド	説明	例
length()	文字数を取得	"Java".length() → 4
charAt(int)	指定位置の文字を取得	"Java".charAt(0) → 'J'
substring(int, int)	部分文字列を取得	"JavaWorld".substring(0, 4) → "Java"
indexOf(String)	検索（前から）	"Java".indexOf("a") → 1
lastIndexOf(String)	検索（後ろから）	"Java".lastIndexOf("a") → 3
replace(char, char)	文字置換	"Java".replace('a', 'o') → "Jovo"
toUpperCase()/toLowerCase()	大文字/小文字変換	"Java".toUpperCase() → "JAVA"
trim()	前後の空白を削除	" Java ".trim() → "Java"
split(String)	区切り文字で分割	"A,B,C".split(",") → ["A","B","C"]

StringBuilderとStringBuffer

Stringは不変であるため、頻繁に変更が必要な場合はStringBuilderやStringBufferを使います。

StringBuilder

- 非同期（スレッドセーフではない）
- 単一スレッドで使用する場合に最適
- StringBufferより高速

StringBuffer

- 同期（スレッドセーフ）
- 複数スレッドで共有する場合に安全
- StringBuilderより低速

```
// StringBuilder の使用例
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(", ");
sb.append("World");
String result = sb.toString(); // "Hello, World"

// メソッドチェーン
String result2 = new StringBuilder()
    .append("Hello")
    .append(", ")
    .append("World")
    .toString();
```

ベストプラクティス： ループ内で多くの文字列連結を行う場合は、常にStringBuilderを使用してパフォーマンスを向上させましょう。

≠ 文字列の比較

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

// == 演算子（参照比較）
System.out.println(s1 == s2); // true（同じ文字列プール内のオブジェクト）
System.out.println(s1 == s3); // false（異なるオブジェクト）

// equals メソッド（内容比較）
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // true

// compareTo メソッド（辞書順比較）
String a = "Apple";
String b = "Banana";
System.out.println(a.compareTo(b)); // 負の値（aはbより前）
```

注意： 文字列の内容を比較する場合は常にequals()メソッドを使用しましょう。==演算子は参照が同じかどうかを比較するため、異なるオブジェクトの同じ内容を比較するとfalseになります。

JAVA基本構文：入出力処理

データを読み書きするための基本技術

標準入出力

Javaでは、System.inで標準入力、System.outで標準出力を扱うことができます。


```
// 標準出力の使用例
System.out.println("Hello, World!"); // 改行あり
System.out.print("こんにちは"); // 改行なし
System.out.printf("%s, %d", "Java", 17); // 書式指定
```

標準エラー出力： エラーメッセージには System.err を使用します。

Scannerクラス

java.utilパッケージのScannerクラスを使用すると、様々な形式の入力を簡単に読み取ることができます。

```
import java.util.Scanner;

// キーボードからの入力
Scanner scanner = new Scanner(System.in);

System.out.print("名前を入力: ");
String name = scanner.nextLine(); // 1行読み込み

System.out.print("年齢を入力: ");
int age = scanner.nextInt(); // 整数を読み込み

System.out.println("こんにちは " + name + ", あなたは " + age + " 歳です。");

scanner.close(); // 使用後は必ずクローズ
```

注意： nextInt()の後にnextLine()を呼び出すと、改行文字が残っているため空行が読み込まれます。対策として、nextInt()の後にnextLine()を余分に呼び出して改行文字を消費します。

Fileクラスの基本操作

java.ioパッケージのFileクラスを使用すると、ファイルやディレクトリを操作できます。

```
import java.io.File;

// ファイルの作成と情報取得
File file = new File("example.txt");

boolean exists = file.exists(); // ファイルの存在確認
long size = file.length(); // ファイルのサイズ（バイト）
boolean isDirectory = file.isDirectory(); // ディレクトリかどうか

// ディレクトリの作成
File dir = new File("myFolder");
dir.mkdir(); // ディレクトリの作成

// ディレクトリ内のファイル一覧取得
File[] files = dir.listFiles();
```

メソッド	説明
createNewFile()	新しい空のファイルを作成
delete()	ファイル/空のディレクトリを削除
renameTo(File)	ファイル名変更/移動
getAbsolutePath()	絶対パスを取得

ファイル入出力の基本

テキストファイルの読み書きにはFileReader/FileWriterクラスが便利です。

```
import java.io.*;

// ファイルへの書き込み
try (FileWriter writer = new
FileWriter("output.txt")) {
  writer.write("Hello, Java ファイル入出力\n");
  writer.write("2行目の内容");
} catch (IOException e) {
  e.printStackTrace();
}

// ファイルからの読み込み
try (BufferedReader reader = new
BufferedReader(new FileReader("output.txt"))) {
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

- **BufferedReader/BufferedWriter**: バッファリングによりI/O処理が効率化
- **try-with-resources**: リソースの自動クローズが保証される
- **Java NIO**: より高度なI/O操作には、java.nioパッケージを使用

オブジェクト指向プログラミング：概要

Javaの中心的な概念を理解する

オブジェクト指向とは

オブジェクト指向プログラミング（OOP）は、データと処理を一体化した「オブジェクト」を基本単位としてプログラムを構成する手法です。

オブジェクト指向の4つの柱

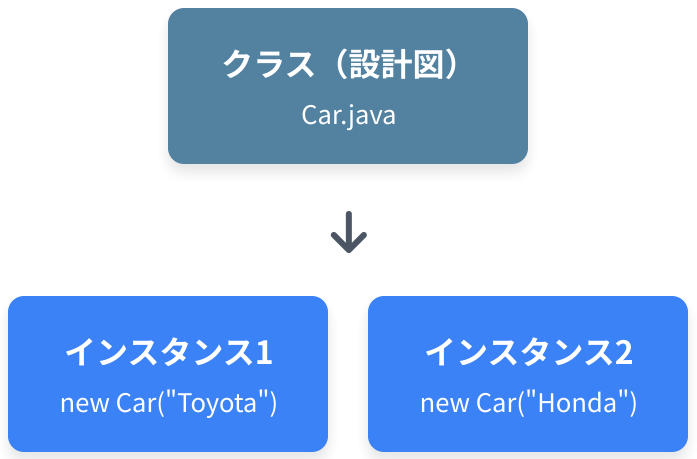
- **カプセル化**：データと処理をひとまとめにし、内部詳細を隠蔽
- **継承**：既存クラスの特性を引き継いで新しいクラスを作成
- **ポリモーフィズム**：同じインターフェースで異なる実装を提供
- **抽象化**：複雑な実装を単純な形で表現

MEMO： Javaはオブジェクト指向言語として設計されており、すべてのプログラムはクラスとオブジェクトで構成されています（プリミティブ型を除く）。

オブジェクト指向のメリット

- **再利用性**：一度作成したクラスを別のプログラムで再利用可能
- **保守性**：プログラムの一部を変更しても他の部分に影響が少ない
- **拡張性**：既存のコードを変更せずに新機能を追加可能
- **モジュール化**：複雑なシステムを小さな単位で管理
- **チーム開発**：役割分担が明確になり、並行開発が容易

クラス・オブジェクト・インスタンスの関係



- **クラス**：オブジェクトの設計図（属性とメソッドを定義）
- **オブジェクト**：クラスの定義に基づいて作成された実体
- **インスタンス**：メモリ上に実体化されたオブジェクト
- **インスタンス化**：クラスからオブジェクトを生成する過程

クラスとオブジェクトの例え

クラスとオブジェクトの関係は「設計図」と「製品」の関係に似ています：

- 家の設計図（クラス）から、実際の家（オブジェクト）を建てる
- 同じ設計図から複数の家を建てることができる
- 同じ設計でも、色や内装などが異なる家ができる（状態の違い）

Javaにおけるクラスの基本構造

```java
// ファイル名：Car.java
public class Car {
    // フィールド（属性）
    private String brand;
    private String model;
    private int year;

    // コンストラクタ
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // メソッド（振る舞い）
    public void start() {
        System.out.println(brand + " " + model + " エンジン始動");
    }

    public void stop() {
        System.out.println(brand + " " + model + " エンジン停止");
    }
}
```

```java
// 使用例
Car myCar = new Car("Toyota", "Prius", 2023);
myCar.start();  // 出力: Toyota Prius エンジン始動
myCar.stop();   // 出力: Toyota Prius エンジン停止
```

オブジェクト指向プログラミング：クラスとオブジェクト

クラスの定義とオブジェクトの生成

クラスの定義

Javaでクラスを定義する際の基本的な構文と要素について説明します。

```
public class Student { // フィールド（インスタンス変数） private String name; private int id; private double gpa; // コンストラクタ public Student(String name, int id) { this.name = name; this.id = id; this.gpa = 0.0; } // メソッド public void study() { System.out.println(name + "が勉強しています"); } public void updateGPA(double newGPA) { this.gpa = newGPA; } // ゲッター public String getName() { return name; } public double getGPA() { return gpa; } }
```

MEMO： クラス名は慣習的に大文字で始め、ファイル名もクラス名と同じにします（Student.java）。

Student クラス

フィールド（データ）

- private String name
- private int id
- private double gpa

コンストラクタ

+ Student(String name, int id)

メソッド（振る舞い）

+ void study()
+ void updateGPA(double newGPA)
+ String getName()
+ double getGPA()

オブジェクトの生成と利用

クラスからオブジェクト（インスタンス）を作成し、使用する方法です。

```
// Studentクラスから2つのオブジェクトを生成 Student student1 = new Student("鈴木太郎", 101); Student student2 = new Student("佐藤花子", 102); // 各オブジェクトのメソッドを呼び出し student1.study(); // "鈴木太郎が勉強しています" と出力 student1.updateGPA(3.5); student2.study(); // "佐藤花子が勉強しています" と出力 student2.updateGPA(3.8); // ゲッターを使用してデータにアクセス System.out.println(student1.getName() + "のGPA: " + student1.getGPA()); System.out.println(student2.getName() + "のGPA: " + student2.getGPA());
```

student1		student2
name: "鈴木太郎" id: 101 gpa: 3.5	↔	name: "佐藤花子" id: 102 gpa: 3.8

クラスとオブジェクトの重要なポイント

1 コンストラクタの役割

オブジェクトの初期化を担当し、クラス名と同じ名前で、戻り値を指定しません。複数のコンストラクタを定義できます（オーバーロード）。

```
public Student() { } // デフォルトコンストラクタ public Student(String name) { this.name = name; } // 別のコンストラクタ
```

2 this キーワード

現在のオブジェクト（インスタンス）を参照するために使用します。特にフィールド名とパラメータ名が同じ場合に便利です。

3 アクセス修飾子

クラスのメンバー（フィールドやメソッド）のアクセス範囲を制限します。

- `private`：同じクラス内からのみアクセス可能
- `public`：どこからでもアクセス可能
- `protected`：同じパッケージ内とサブクラスからアクセス可能
- 修飾子なし（default）：同じパッケージ内からのみアクセス可能

4 カプセル化の実践

フィールドを`private`にして直接アクセスを制限し、ゲッター・セッターメソッドを通じてアクセスを提供することが一般的です。