目次

第 1 章：Java とは何か

— なぜ今も Java は使われ続けているのか —

この章のゴール

この章を終えると、次のことを自分の言葉で説明できるようになります。

- Java がどんな目的で作られた言語か
- JVM / JDK / JRE の違い
- Java が今も業務で使われ続けている理由
- Java が向いている分野・向いていない分野

この章ではまだコードを書きません。
まずは「考え方の土台」を作ります。

---

1. Java はどんな言語か（結論）

結論から言います。

> Java は「長く・安全に・大規模に動かす」ための言語です。

流行を追う言語ではありません。
しかし、消えない言語です。

---

2. Java が生まれた背景

Java は 1995 年、Sun Microsystems によって開発されました。

当時の課題は次のようなものでした。

- OS ごとにプログラムを書き直す必要がある
- C/C++ は高速だが、メモリ事故が起きやすい
- 家電・業務システムを安全に動かしたい

これらを解決するために生まれた思想が、
Java の核となっています。

---

3. Write Once, Run Anywhere（WORA）

Java を理解するうえで最も重要な考え方です。

一度書いたプログラムを、どこでも動かす

仕組みのイメージ
Java ソースコード（.java）↓ コンパイル（javac）↓ バイトコード（.class）↓ JVM が実行

ポイント：

- Java は OS 上で直接動かない
- OS ごとに用意された JVM が実行する
- Java プログラムは「JVM 向け」に書かれている

これにより、
Windows / macOS / Linux で同じプログラムが動きます。

---

4. JVM / JDK / JRE の違い（重要）

この 3 つは必ず整理して理解してください。

JVM（Java Virtual Machine）

- Java プログラムを実行する仮想マシン
- OS ごとに存在する
- 実行専門（開発はできない）

-> Java を動かす心臓部

JRE（Java Runtime Environment）

- JVM + 標準ライブラリ
- Java プログラムを「実行するだけ」の環境

-> 利用者向け

JDK（Java Development Kit）

- JRE + 開発ツール
- javac / java / javadoc などを含む

-> 開発者は必ずこれを使う

関係図

[JDK] ├── javac（コンパイル）├── java（実行）└── [JRE] └──[JVM]

---

5. Java はどこで使われているか

Java が強い分野

- 金融システム
- 業務系 Web アプリ
- Android アプリ
- 大規模サーバーシステム

共通点は：

- 止まると困る
- 長期間運用される
- 多人数で開発される

-> 安定性と保守性が最優先

Java があまり向かない分野

- ゲームエンジンのコア
- 超低レイヤ処理
- 一時的なスクリプト

-> Java は堅牢さ重視

---

6. 他言語との位置づけ（経験者向け）

Java vs JavaScript

Java	JavaScript
コンパイル型	インタプリタ
型が厳格	型が柔軟
大規模向き	フロント向き

※ 名前が似ているだけで別言語です。

Java vs Rust

Java	Rust
GC あり	所有権
学習コスト低	学習コスト高
チーム向き	高度な安全性

-> Java は「多人数開発向け」

---

7. Java を学ぶ価値はあるのか

結論は明確です。

> ありません。ただし「目的次第」です。

Java を学ぶことで：

- オブジェクト指向の基礎体力がつく
- 型安全な設計思考が身につく
- 業務コードを読む力がつく

Java は
他言語理解の土台になります。

---

8. この章のまとめ

- Java は長期運用向けの言語
- JVM が最大の特徴
- JDK / JRE / JVM の違いは必須知識
- 流行ではなく「インフラ」

次章予告

次は 第 2 章：開発環境の準備 です。

- なぜ JDK を入れるのか
- javac / java の役割
- 環境構築で詰まらない考え方

を扱います。

第 2 章：開発環境の準備

―「環境構築で詰まらない」ための考え方 ―

この章のゴール

この章を終えると、次のことができるようになります。

- JDK をなぜインストールするのか説明できる
- `javac` と `java` の役割の違いがわかる
- Java が「どこで」実行されているか理解できる
- 環境構築エラーに冷静に対処できる

---

1. なぜ JDK が必要なのか

結論から言います。

> Java で開発するなら、JDK 一択です。

理由：

- `.java` → `.class` に変換するには `javac`
- 実行するには `java`
- これらは JDK に含まれている

JRE だけでは「書けない」
JVM だけでは「動かせない」

---

2. JDK に含まれる主なツール

ツール	役割
javac	コンパイル
java	実行
javadoc	ドキュメント生成
jshell	対話実行

---

3. JDK のインストール

推奨

- Oracle JDK
- OpenJDK（Temurin など）
- LTS 版を選択

---

4. インストール確認（必須）

ターミナルで実行：

```
java -version
javac -version
```

表示されれば OK。

---

5. PATH とは何か（超重要）

PATH とは「コマンドを探しに行く道案内」です。

OS は：

1. `java` と入力される
2. PATH に登録された場所を順に探す
3. 見つかれば実行

よくあるエラー

```
command not found
```

- -> Java が無いのではない
- -> 場所が分からないだけ

---

6. 環境構築で詰まったときの思考法

順番で確認：

1. JDK は入っているか
2. version コマンドは動くか
3. 複数の Java が入っていないか

-> 感情ではなく 論理で確認

---

7. この章のまとめ

- Java 開発 ＝ JDK
- `javac` と `java` は別物
- PATH は場所の問題
- 環境構築は思考力

次章予告

次は 第 3 章：Hello World を動かす です。

- 実際にコードを書く
- コンパイルと実行を体験する
- プログラムが動く仕組みを目で見る

を扱います。

第 3 章：Hello World

— Java が動く瞬間を理解する —

この章のゴール

- Java プログラムの実行の流れを説明できる
- Hello World を自分で書いて動かせる
- コンパイルと実行の違いがわかる

---

1. Java プログラムの流れ

```
HelloWorld.java
   ↓ javac
HelloWorld.class
   ↓ java
```

実行結果

Java は 必ずコンパイルが必要 です。

---

2. 最初のプログラムを書く

ファイル名

`HelloWorld.java`

中身

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

```
    }
}
```

---

3. コードの意味（最小限）

- `class`：設計図
- `main`：開始地点
- `println`：画面に表示

---

4. コンパイル

ターミナルで実行：

```
javac HelloWorld.java
```

成功すると `.class` ファイルが生成されます。

---

5. 実行

ターミナルで実行：

```
java HelloWorld
```

-> 拡張子は不要

---

6. よくあるエラー

- クラス名とファイル名が違う
- `java HelloWorld.java` と実行する（拡張子をつけてしまう）
- `main` メソッドが無い

---

7. ハンズオン課題

1. 表示する文字を変える

2. 2 行表示する
3. 引数を表示する

---

8. この章のまとめ

- Java はコンパイル言語
- 実行は `main` から
- まず「動かす」ことが重要

次章予告

次は 第 4 章：基本文法（変数と型）です。

- データを記憶する「箱」を作る
- 数値と文字の違い
- 計算してみる

を扱います。

第 4 章：Java プログラムの構造

―「読めるコード」を書く ―

この章のゴール

- Java コードの構造を理解できる
- `{}` の意味が分かる
- コメントの正しい使い方を知る

---

1. Java はブロック構造

```
{
    // この中が 1 つのブロック
}
```

- クラス
- メソッド
- 制御構文

すべて {} で表現されます。

---

2. クラスのブロック

```
class Sample {
}
```

-> Java は必ずクラスの中 に書きます。

---

3. メソッドのブロック

```
void method() {
}
```

-> 処理のまとまり です。

---

4. インデントの重要性

```
class A {
    void m() {
        System.out.println("OK");
    }
}
```

- 動作は変わらない
- 可読性が大きく変わる

---

5. コメントの種類

1 行コメント

```
// コメント
```

複数行コメント

```
/*
 コメント
*/
```

ドキュメントコメント

```
/**
 * 説明
 */
```

---

6. 良いコメント・悪いコメント

- [NG] コードの説明
- [OK] 意図の説明

---

7. 命名規則（最低限）

種類	例
クラス	`HelloWorld`
メソッド	`sayHello`
変数	`userName`

---

8. ハンズオン課題

1. インデントを整える
2. コメントを追加する
3. 読みやすく書き直す

---

9. この章のまとめ

- Java は構造で読む

- {} が設計を表す
- 可読性はスキル

次章予告

次は 第 5 章：変数とデータ型 です。

- データを記憶する「箱」を作る
- 数値と文字の違い
- 計算してみる

を扱います。

第 5 章：変数とデータ型

―「型がある」ことの意味を理解する ―

この章のゴール

この章を終えると、次のことができるようになります。

- 変数とは何かを説明できる
- Java の基本的なデータ型を使える
- プリミティブ型と参照型の違いがわかる
- 型エラーを恐れずにコードが書ける

---

1. 変数とは何か

変数とは、値に名前を付けること

```
int age = 20;
```

- int：型
- age：名前
- 20：値

-> 名前を付けることで「意味」を持ちます。

2. なぜ型が必要なのか

Java は 型が厳格な言語 です。

型があることで：

- 実行前にミスを検出できる
- コードの意図が明確になる
- チーム開発で事故が減る

-> 型は制約ではなく保険

---

3. プリミティブ型（基本型）

よく使う基本型：

型	意味
`int`	整数
`long`	大きな整数
`double`	小数
`boolean`	`true` / `false`
`char`	1 文字

```
int count = 10;
double price = 12.5;
boolean active = true;
```

---

4. 参照型とは何か

```
String name = "Java";
```

参照型の特徴：

- 値そのものではなく「場所」を持つ
- `null` を代入できる
- メソッドを持つ

---

5. String が特別な理由

```
String s = "Hello";
```

これは実は省略記法：

```
String s = new String("Hello");
```

- 不変（immutable）
- 業務で最重要な型

---

6. var（型推論）

```
var message = "Hello";
```

注意点：

- ローカル変数のみ
- 可読性を下げる場合がある

-> 使いすぎない

---

7. 型変換（キャスト）

```
int x = 10;
double y = x;      // 暗黙変換

double a = 9.8;
int b = (int) a;   // 明示的変換
```

-> 情報が失われる可能性あり。

---

8. よくあるエラー

```
int x = "10"; // コンパイルエラー
```

-> 型が違う。

---

9. ハンズオン課題

1. `int` / `double` / `boolean` の変数を宣言する
2. `String` で自己紹介文を作る
3. `double` → `int` キャストを試す

---

10. この章のまとめ

- 変数は意味のある名前
- 型は安全装置
- `String` は特別
- 型を味方にする

次章予告

次は 第 6 章：演算子 です。

- 計算を行う
- 条件を比較する
- 「インクリメント」の正体

を扱います。

第 6 章：演算子

— プログラムは「計算」と「判断」でできている —

この章のゴール

この章を終えると、次のことができるようになります。

- 基本的な演算子を使える
- 条件判断の仕組みがわかる
- `==` と `equals` の違いを意識できる

1. 算術演算子

演算子	意味
+	加算
-	減算
*	乗算
/	除算
%	余り

```
int a = 10;
int b = 3;
System.out.println(a / b); // 3
```

-> 整数同士の割り算は整数。

---

2. 代入演算子

```
int x = 5;
x += 3; // x = x + 3 と同じ
```

---

3. 比較演算子

```
int age = 20;
System.out.println(age >= 18); // true
```

---

4. 論理演算子

```
boolean adult = true;
boolean ticket = false;

System.out.println(adult && ticket); // false
```

- `&&` : かつ (AND)

- || : または (OR)
- ! : 否定 (NOT)

---

5. == と equals の違い

プリミティブ型

```
int a = 10;
int b = 10;
System.out.println(a == b); // true
```

参照型（String）

```
String s1 = "Java";
String s2 = "Java";

System.out.println(s1 == s2);       // 参照比較（同じ場所か？）
System.out.println(s1.equals(s2)); // 中身比較（同じ文字か？）
```

-> 文字列比較は必ず equals を使う

---

6. よくある落とし穴

- 複雑すぎる条件式
- カッコを付けない

-> 読みやすさ優先

---

7. ハンズオン課題

1. 2 つの数の四則演算を行う
2. 年齢判定の条件式を書く
3. equals と == の比較を試す

---

8. この章のまとめ

- 演算子は判断の道具
- 整数除算に注意
- equals を忘れない

次章予告

次は 第 7 章：条件分岐 です。

- if 文で処理を分ける
- switch 文の使いどころ
- 条件が複雑になったらどうするか

を扱います。

第 7 章：制御構文

— プログラムに「流れ」を与える —

この章のゴール

この章を終えると、次のことができるようになります。

- if / switch を使える
- for / while で繰り返し処理が書ける
- ロジックをコードに落とせる

---

1. if 文（条件分岐）

```
int age = 20;

if (age >= 18) {
    System.out.println("成人");
} else {
```

```
    System.out.println("未成年");
}
```

---

2. else if

```
int score = 75;

if (score >= 80) {
    System.out.println("A");
} else if (score >= 60) {
    System.out.println("B");
} else {
    System.out.println("C");
}
```

---

3. switch 文

```
int n = 2;

switch (n) {
    case 1:
        System.out.println("1");
        break;
    case 2:
        System.out.println("2");
        break;
    default:
        System.out.println("その他");
}
```

---

4. for 文（回数が決まっている）

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

5. while 文（条件が重要）

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

---

6. break / continue

```
for (int i = 0; i < 5; i++) {
    if (i == 2) continue; // スキップ
    if (i == 4) break;    // 中断
    System.out.println(i);
}
```

---

7. よくあるミス

- 無限ループ
- 条件の書き間違い

-> 紙に書いて考える。

---

8. ハンズオン課題

1. 1〜10 を表示する
2. 偶数のみ表示する
3. 点数評価プログラムを作る

---

9. この章のまとめ

- 制御構文は思考の翻訳

- `if` / `for` / `while` を使い分ける
- 読めるロジックを書く

次章予告

次は 第 8 章：配列 です。

- たくさんのデータをまとめて扱う
- 配列の仕組みと注意点
- `for` 文との組み合わせ

を扱います。

第 8 章：配列とコレクション

―「複数のデータ」を正しく扱う ―

この章のゴール

この章を終えると、次のことができるようになります。

- 配列の基本を理解できる
- 配列の限界を説明できる
- `List`（コレクション）を使える
- `for` 文と組み合わせて処理できる

---

1. なぜ「複数の値」を扱うのか

今まで：

```
int a = 10;
int b = 20;
int c = 30;
```

問題点：

- 数が増えると管理不能
- 動的に増減できない

->「まとめて扱う」必要がある

---

2. 配列とは何か

配列とは、同じ型の値を並べて管理する仕組み

```
int[] numbers = new int[3];
```

- 要素数は固定
- インデックスは 0 から始まる

---

3. 配列の操作

代入

```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

取得

```
System.out.println(numbers[1]); // 20
```

---

4. 配列の注意点

```
numbers[3] = 40; // 実行時エラー
```

- 範囲外アクセスは例外
- サイズ変更不可

-> これが配列の限界

---

5. for 文と配列

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

- length はプロパティ
- ハードコードしない

---

6. 拡張 for 文（for-each）

```
for (int n : numbers) {
    System.out.println(n);
}
```

- 読みやすい
- インデックス不要

---

7. コレクションとは何か

サイズが可変なデータ構造

よく使うのは：

- List：順序あり
- Set：重複なし
- Map：キーと値

---

8. List を使ってみる

```
import java.util.ArrayList;
import java.util.List;

List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
```

取得

```
System.out.println(names.get(0));
```

---

9. 配列と List の違い

	配列	List
サイズ	固定	可変
速度	高速	柔軟
機能	単純	高機能

-> 実務では List が主役

---

10. ハンズオン課題

1. `int` 配列を作り表示する
2. `String` の `List` を作成する
3. 合計・平均を計算する

---

11. この章のまとめ

- 配列は固定長
- `List` は柔軟
- 使い分けが重要

次章予告

次は 第 9 章：メソッド です。

- 処理をまとめて再利用する
- 引数と戻り値の仕組み
- 「スコープ」とは何か

を扱います。

第 9 章：メソッド

— 処理を「名前付きの部品」にする —

この章のゴール

この章を終えると、次のことができるようになります。

- メソッドの定義と呼び出しができる
- 引数と戻り値を理解する
- 再利用できるコードを書ける

---

1. なぜメソッドが必要か

すべて `main` に書くと：

- 長くなる
- 読みにくい
- 再利用できない

-> 処理を分ける必要がある

---

2. メソッドの基本形

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

---

3. 例：足し算メソッド

```
static int add(int a, int b) {
    return a + b;
}
```

呼び出し：

```
int result = add(3, 5);
```

---

4. void メソッド

```
static void greet(String name) {
    System.out.println("こんにちは " + name);
}
```

- 表示などに使う（値を返さない）

---

5. 引数の考え方

- 外から値を渡す
- 汎用性が上がる

```
static int square(int x) {
    return x * x;
}
```

---

6. 1 メソッド 1 責務

[NG] 悪い例：

```
void process() {
    // 入力・計算・表示
}
```

[OK] 良い例：

```
int calc() { }
void print() { }
```

---

7. オーバーロード

```
int add(int a, int b) { }
int add(int a, int b, int c) { }
```

---

8. ハンズオン課題

1. 合計を返すメソッドを作る
2. 判定を返すメソッドを作る
3. 配列の平均を返すメソッドを作る

---

9. この章のまとめ

- メソッドは部品
- 再利用が価値
- 読める設計を意識

次章予告

次は 第 10 章：クラスとオブジェクト です。

- クラス＝設計図
- オブジェクト＝実体
- `new` キーワードの意味

を扱います。

第 10 章：クラスとオブジェクト

― Java は「設計図」と「実体」で考える ―

この章のゴール

この章を終えると、次のことができるようになります。

- クラスとオブジェクトの違いを理解する
- フィールドとメソッドを定義できる
- `new` の意味を説明できる

---

1. なぜクラスが必要か

```
String name;
int age;
```

人が増えると管理不能。

-> まとめて扱う必要がある

---

2. クラスとは何か

クラスは オブジェクトの設計図

```
class Person {
    String name;
    int age;
}
```

---

3. オブジェクトを作る

```
Person p = new Person();
```

- new：実体を作る

---

4. フィールド操作

```
p.name = "Alice";
p.age = 20;
```

---

5. メソッドを持たせる

```
class Person {
    String name;
    int age;
```

```
    void introduce() {
        System.out.println(name + "です");
    }
}
```

---

6. コンストラクタ

```
Person(String name, int age) {
    this.name = name;
    this.age = age;
}
```

---

7. this の意味

- このオブジェクト自身
- フィールドと引数の区別

---

8. カプセル化（入口）

- フィールドを直接触らせない
- メソッド経由で操作

-> 次章につながる

---

9. ハンズオン課題

1. Book クラス作成
2. コンストラクタ追加
3. 情報表示メソッドを作る

---

10. この章のまとめ

- クラス＝設計図
- オブジェクト＝実体
- Java の核心

次章予告

次は 第 11 章：カプセル化 です。

- データを守る仕組み
- `private` と `public`
- Getter / Setter の役割

を扱います。

第 11 章：継承とポリモーフィズム

―「共通点」と「違い」を正しく扱う ―

この章のゴール

この章を終えると、次のことができるようになります。

- 継承の目的を説明できる
- `extends` の意味を理解する
- メソッドのオーバーライドができる
- ポリモーフィズム（多態性）を体感する

---

1. なぜ継承が必要なのか

次のコードを考えます。

```
class Dog {
    void speak() {
```

```
        System.out.println("ワン");
    }
}

class Cat {
    void speak() {
        System.out.println("ニャー");
    }
}
```

問題点：

- 構造が同じ
- クラスが増えるほどコピペが増える

-> 共通部分をまとめたい

---

2. 継承とは何か

継承とは、既存クラスの性質を引き継ぐこと

```
class Animal {
    void speak() {
        System.out.println("鳴く");
    }
}

class Dog extends Animal {
}
```

- Animal：親クラス（スーパークラス）
- Dog：子クラス（サブクラス）

---

3. メソッドのオーバーライド

```
class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("ワン");
```

```
    }
}
```

- 親のメソッドを上書き
- `@Override` を付けるのが推奨

---

4. ポリモーフィズムとは何か（重要）

```
Animal a1 = new Dog();
Animal a2 = new Cat();

a1.speak(); // ワン
a2.speak(); // ニャー
```

- 型：`Animal`
- 実体：`Dog` / `Cat`

-> 同じ型で違う振る舞い

---

5. ポリモーフィズムの強み

```
void makeSpeak(Animal animal) {
    animal.speak();
}
```

- クラスが増えても修正不要
- 拡張に強い設計

---

6. super キーワード

```
class Dog extends Animal {
    @Override
    void speak() {
        super.speak();
        System.out.println("ワン");
    }
}
```

- 親クラスの処理を呼び出す

---

7. 継承の注意点

- is-a 関係か？
- 乱用しない

-> 次章の interface に続く（※本教材では次章は例外処理ですが、概念的にはインターフェースと密接です）

---

8. ハンズオン課題

1. `Animal` クラス作成
2. `Dog` / `Cat` クラス作成
3. `List<Animal>` で `speak` を呼ぶ

---

9. この章のまとめ

- 継承は共通化
- `override` で振る舞い変更
- ポリモーフィズムが設計を楽にする

次章予告

次は 第 12 章：例外処理 です。

- エラーと例外の違い
- `try-catch` の使い方
- 落ちないプログラムを作る

を扱います。

第 12 章：例外処理

—「エラーで落ちない」プログラムを書く —

この章のゴール

この章を終えると、次のことができるようになります。

- エラーと例外の違いを理解する
- try / catch を使える
- 例外を設計として扱える

---

1. エラーと例外の違い

- Error：致命的（基本対処しない）
- Exception：対処可能

-> 扱うのは例外

---

2. 例外が起きる例

```
int[] arr = {1, 2, 3};
System.out.println(arr[5]);
```

-> 実行時エラー。

---

3. try / catch の基本

```
try {
    // 処理
} catch (Exception e) {
    // 例外処理
}
```

---

4. 複数の catch

```
try {
} catch (NullPointerException e) {
} catch (Exception e) {
}
```

- 広い例外は最後

---

5. finally

```
finally {
    // 必ず実行
}
```

---

6. checked / unchecked 例外

- checked：コンパイル時に強制
- unchecked：実行時

---

7. 例外を投げる

```
if (age < 0) {
    throw new IllegalArgumentException();
}
```

---

8. よくある誤解

- [NG] 例外を無視する
- [NG] 全部 catch する

-> バグを隠すな

---

9. ハンズオン課題

1. 配列アクセスを try-catch する
2. null の例外捕捉
3. 自作例外を投げる

---

10. この章のまとめ

- 例外は設計の一部
- 落ちないコードを書く

次章予告

次は 第 13 章：パッケージとモジュール です。

- クラスを整理する方法
- import の仕組み
- 大規模開発への備え

を扱います。

第 13 章：パッケージとモジュール

— コードを「迷子」にしない —

この章のゴール

この章を終えると、次のことができるようになります。

- package の役割を理解する
- import の意味がわかる
- クラスを整理できる

---

1. なぜパッケージが必要か

クラスが増えると：

- 名前衝突
- 見通し悪化

-> 整理が必要

---

2. パッケージとは何か

```
package com.example.app;
```

- フォルダ構造と一致
- 名前空間

---

3. import の役割

```
import java.util.List;
```

- クラス名を省略できる
- コピーではない

---

4. 同名クラスの衝突

```
java.util.Date date;
```

-> 完全修飾名を使う。

---

5. パッケージ設計の考え方

- model
- service
- controller
- util

-> 役割で分ける

---

6. モジュール（Java9+）

```
module my.app {
    requires java.base;
}
```

- 大規模向け
- 今は存在を知る程度で OK

---

7. ハンズオン課題

1. 自分用パッケージ作成
2. クラスを分割
3. `import` して使用

---

8. この章のまとめ

- パッケージは整理術
- `import` は可読性
- 大規模ほど重要

次章予告

次は 第 14 章：実践編（予定）です。

これまでの知識を活かして、より実践的な開発手法やライブラリの活用について学びます。

第 14 章：ジェネリクス

―「型安全」を仕組みで実現する ―

この章のゴール

この章を終えると、次のことができるようになります。

- ジェネリクスの目的を説明できる
- <T> の意味がわかる
- List<String> が安全な理由を理解できる
- キャスト不要な設計ができる

---

1. ジェネリクスが必要な理由

ジェネリクスが無いと：

```
List list = new ArrayList();
list.add("Java");
list.add(10);

String s = (String) list.get(0); // 実行時エラーの可能性
```

問題点：

- 何が入っているか分からない
- キャストが必要
- バグが実行時まで分からない

-> 安全ではない

---

2. ジェネリクスとは何か

型を後から指定できる仕組み

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

- String 以外は入れられない
- 間違いは コンパイル時 に検出

---

3. ジェネリクスの基本構文

クラス名<型引数>

例：

```
List<Integer> numbers = new ArrayList<>();
```

---

4. の意味

```
class Box<T> {
    private T value;

    void set(T value) {
        this.value = value;
    }

    T get() {
        return value;
    }
}
```

- T は型の変数
- 使う側が型を決める

---

5. ジェネリクスクラスの利用

```
Box<String> box = new Box<>();
box.set("Hello");

String s = box.get();
```

-> キャスト不要・安全

---

6. ジェネリクスメソッド

```
static <T> void print(T value) {
    System.out.println(value);
}
```

---

7. 制限付きジェネリクス

```
static <T extends Number> void printNumber(T n) {
    System.out.println(n);
}
```

- Number のサブクラスのみ許可

---

8. ワイルドカード

```
List<?> list;
List<? extends Number> nums;
```

- API 設計で重要
- 読み取り中心

---

9. ハンズオン課題

1. raw 型とジェネリクスの違いを確認
2. Box<T> を実装して使う
3. Number 制約のメソッド作成

---

10. この章のまとめ

- ジェネリクスは型安全の要
- 実行前にバグを防ぐ
- Java の信頼性の核心

次章予告

次は 第 15 章：Stream API です。

- `for` 文を書かずにデータを処理する
- `map` / `filter` の使い方
- 読みやすいコードを書く技術

を扱います。

第 15 章：Stream API

— for 文を書かずに「流れ」で処理する —

この章のゴール

この章を終えると、次のことができるようになります。

- Stream API の考え方を理解する
- `map` / `filter` / `reduce` を使える
- `for` 文との使い分けができる

---

1. 従来の for 文

```
int sum = 0;
for (int n : numbers) {
    if (n % 2 == 0) {
        sum += n;
    }
}
```

問題：

- 目的が見えにくい
- 手続きが混ざる

---

2. Stream API とは

データの流れに処理をつなげる仕組み

```
int sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(n -> n)
    .sum();
```

-> 意図がそのまま読める

---

3. Stream の構造

```
collection.stream()
    . 中間操作 ()
    . 終端操作 ();
```

- 中間：map / filter
- 終端：forEach / sum

---

4. filter（絞り込み）

```
numbers.stream()
    .filter(n -> n > 5)
    .forEach(System.out::println);
```

---

5. map（変換）

```
names.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

---

6. 集計（reduce / sum）

```
int total = numbers.stream().mapToInt(n -> n).sum();
```

---

7. forEach の注意点

- 副作用を避ける
- 状態変更しない

---

8. Stream と for の使い分け

- 変換・集計 → Stream
- 複雑な制御 → for

---

9. ハンズオン課題

1. 偶数抽出
2. 文字列変換
3. 平均計算

---

10. この章のまとめ

- Stream は可読性重視
- モダン Java 必須

次章予告

次は 第 16 章：ファイル入出力 です。

- ファイルを安全に読み書きする
- `try-with-resources` の仕組み

- 実務でのデータの扱い方

を扱います。

第 16 章：ファイル入出力

— ファイルを「安全に」読み書きする —

この章のゴール

この章を終えると、次のことができるようになります。

- Java でファイルを読む・書く
- `try-with-resources` を使える
- 例外を考慮した実装ができる

---

1. なぜファイル入出力が重要か

実務では：

- 設定
- ログ
- CSV / JSON

-> 避けて通れない

---

2. 推奨 API（java.nio.file）

```
import java.nio.file.Files;
import java.nio.file.Path;
```

---

3. ファイルを読む

```
try {
    List<String> lines = Files.readAllLines(Path.of("sample.txt"));
```

```
    lines.forEach(System.out::println);
} catch (IOException e) {
    System.out.println("読み込み失敗");
}
```

---

4. ファイルに書く

```
try {
    Files.writeString(Path.of("output.txt"), "Hello File");
} catch (IOException e) {
    System.out.println("書き込み失敗");
}
```

---

5. try-with-resources

```
try (BufferedReader br = Files.newBufferedReader(Path.of("sample.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

- 自動クローズ
- finally 不要

---

6. パスの注意点

- 相対パスは実行位置基準
- 絶対パスで切り分け

---

7. ハンズオン課題

1. ファイル表示
2. 複数行書き込み

3. 例外メッセージ表示

---

8. この章のまとめ

- NIO を使う
- 例外前提で書く
- 実務必須スキル

次章予告

次は 第 17 章：実践的な応用へ（予定）です。

これまでの知識を活かして、より高度な機能やアプリケーション開発の手法について学びます。

第 17 章：標準ライブラリ活用

―「車輪の再発明」をしない ―

この章のゴール

この章を終えると、次のことができるようになります。

- Java 標準ライブラリの役割を理解する
- 日付と時間を安全に扱える
- `Optional` を使って `null` を回避できる
- 便利なユーティリティクラスを活用できる

---

1. 標準ライブラリとは何か

> Java に最初から用意されている、信頼できる部品集

- テスト済み
- 多くの現場で使用実績あり
- 自作より安全

-> まず標準を疑う

---

2. 日付と時間（java.time）

現在日時

```
LocalDateTime now = LocalDateTime.now();
```

日付の計算

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
```

---

3. フォーマット

```
DateTimeFormatter f =
    DateTimeFormatter.ofPattern("yyyy-MM-dd");

String s = LocalDate.now().format(f);
```

---

4. Optional とは何か

「値があるかもしれない」を型で表現する

```
Optional<String> opt = Optional.of("Java");
```

---

5. Optional の使い方

```
opt.ifPresent(System.out::println);

String v = opt.orElse("default");
```

- null チェック不要
- NPE 防止

---

6. Math / Objects

```
int max = Math.max(10, 20);

Objects.requireNonNull(name);
```

---

7. ハンズオン課題

1. 日付を取得して表示
2. Optional でデフォルト値設定
3. Math を使った計算

---

8. この章のまとめ

- 標準ライブラリは最強
- Optional で安全設計
- 再発明しない

次章予告

次は 第 18 章：ミニアプリ制作 です。

- 学んだ知識を総動員する
- 仕様から設計・実装への流れ
- 小さな成果物を作る

を扱います。

第 18 章：ミニアプリ制作

— 学んだ知識を「形」にする —

この章のゴール

この章を終えると、次のことができるようになります。

- 仕様をコードに落とせる
- クラス設計を実践できる
- Java でアプリを完成させる

---

1. 作るもの：成績管理ミニアプリ

仕様

- 名前と点数を管理
- 平均点を計算
- 結果をファイルに保存

---

2. 設計を文章で書く

1. 学生は名前と点数を持つ
2. 複数の学生を管理する
3. 平均点を計算する

---

3. Student クラス

```
class Student {
    private String name;
    private int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    int getScore() {
        return score;
```

```
    }

    String getName() {
        return name;
    }
}
```

---

4. 管理クラス

```
class ScoreManager {
    private List<Student> students = new ArrayList<>();

    void add(Student s) {
        students.add(s);
    }

    double average() {
        return students.stream()
            .mapToInt(Student::getScore)
            .average()
            .orElse(0.0);
    }
}
```

---

5. main クラス

```
public class Main {
    public static void main(String[] args) {
        ScoreManager m = new ScoreManager();
        m.add(new Student("Alice", 80));
        m.add(new Student("Bob", 70));

        System.out.println(m.average());
    }
}
```

---

6. 改良アイデア

- 標準入力対応
- CSV 入出力
- 合否判定追加

---

7. この章のまとめ

- 仕様 → 設計 → 実装
- Java で作れることを証明

次章予告

次は 第 19 章：次のステップ（最終章）です。

- これからの学習方針
- Web 系・業務系の道
- 「作り続ける」ことの重要性

を扱います。

第 19 章：次のステップ

— Java 学習の「終わり」と「始まり」—

この章のゴール

この章を終えると、次のことができるようになります。

- 今後の学習方針を明確にする
- Java を実務につなげる
- 学習を止めない

---

1. ここまででできるようになったこと

- Java の基本文法を理解
- クラス設計ができる
- 例外を考慮した実装ができる
- 小さなアプリを完成させた

-> 立派なスタートライン

---

2. 次に学ぶ分野

Web 系

- Spring Boot
- REST API
- DB / SQL

業務系

- JUnit
- Maven / Gradle
- ログ

---

3. 作ることが最重要

おすすめ：

- TODO アプリ
- 家計簿
- CSV 加工ツール

-> 小さくていい

---

4. 良い学習ループ

作る → 詰まる → 調べる → 直す → 成長

---

5. Java は古いのか？

いいえ。安定しているだけです。

- 業務・金融・インフラ
- 長期運用前提

---

6. 最後に

ここまでやり切ったことは
大きな成果です。

Java は「目的」ではなく「道具」。

これからは
-> 何を作るか が主役です。

Java 完全ハンズオン教材
〜 完 〜