SpringBoot 基礎

クラス（class）

→ 処理やデータをまとめた設計図。Java プログラムの基本単位。

クラス（class）とは何か

クラスとは、** 処理（動き）とデータ（情報）をひとまとめにした「設計図」** です。
Java のプログラムは基本的にクラスを中心に作られており、クラスは Java プログラムの基本単位といえます。

クラスを用いることで、プログラムの中で扱うもの（例：人、商品、メモ、予約など）を、
現実の対象になぞらえて整理しやすくなります。

クラスに含まれる主な要素

クラスには、主に次のような要素を定義します。

1. フィールド（field：データ）

そのクラスが持つ情報（状態）を表します。
例として「メモ」を表すクラスなら、「本文」「作成日時」などがフィールドになります。

2. メソッド（method：処理）

そのクラスが行う処理（振る舞い）を表します。
例として「メモ」を表すクラスなら、「本文を表示する」「内容を変更する」などがメソッドになります。

3. コンストラクタ（constructor：初期化）

オブジェクトを作成するときに、最初の状態を決める仕組みです。

クラスとオブジェクト（インスタンス）の違い

初学者が最も混乱しやすい点として、クラスと ** オブジェクト（インスタンス）** の違いがあります。
この違いは必ず押さえてください。

- クラス：設計図（型）
- オブジェクト（インスタンス）：設計図から作られた実体（実物）

たとえば、「Person（人）」というクラスは「人の設計図」です。
そこから作られた「田中さんの情報を持つ実体」がオブジェクトです。

クラスの基本例（最小構成）

以下は「人」を表すクラスの例です。

```
public class Person {

  String name;  // フィールド（データ）
  int age;

  void sayHello() {  // メソッド（処理）
    System.out.println(" こんにちは、" + name + " です。");
  }
}
```

この時点では「設計図」が定義されただけで、まだ実体（オブジェクト）は作られていません。

オブジェクトを作成する（new）

クラスから実体を作るには new を使います。

```
Person p = new Person();
p.name = " 田中 ";
p.age = 20;
p.sayHello();
```

- new Person() によって Person のオブジェクトが作られます
- p はそのオブジェクトを指し示す変数です
- p.sayHello() でメソッドを実行します

アクセス修飾子（public / private）とクラス

クラスでは、外部からのアクセスを制御するために アクセス修飾子を使います。

- public：外部から利用できる
- private：外部から直接利用できない（クラス内部だけで利用する）

これは、データを勝手に書き換えられないようにし、安全で壊れにくいプログラムにするための仕組みです。

Spring Boot ではクラスが「部品」になる

Spring Boot では、クラスは単なる設計図ではなく、アプリケーションを構成する
** 部品（コンポーネント）** として扱われます。
たとえば、次のような役割のクラスが登場します。

@Controller：画面や URL の受付を担当するクラス

@Service：業務処理を担当するクラス
@Mapper：DB 操作を担当するクラス（MyBatis）

つまり Spring Boot の学習では、**「クラス＝役割を持った部品」** として捉えると理解が進みます。

まとめ（教科書用）

クラスとは、処理（メソッド）とデータ（フィールド）をまとめて定義する設計図です。
Java ではクラスがプログラムの基本単位となり、クラスから作られた実体をオブジェクト（インスタンス）と呼びます。
Spring Boot では、クラスはアプリケーションを構成する部品として管理され、
役割に応じたクラスを組み合わせてアプリケーションを作成します。

メソッド（method）

メソッド（method）とは何か

メソッドとは、** クラスの中に定義される「処理のまとまり」** です。
プログラムで行いたい処理を、名前を付けてひとつの単位にまとめたものがメソッドです。

メソッドを使うことで、同じ処理を何度も書かずに呼び出せるようになり、プログラムを読みやすく、
修正しやすくできます。

メソッドを使う目的

メソッドには、主に次のような目的があります。

1. 処理を再利用できるようにする
同じ処理を複数回行う場合でも、メソッドにまとめておけば呼び出すだけで実行できます。

2. 処理の内容を分かりやすくする
処理に名前を付けることで、「何をしている処理か」がコードから読み取りやすくなります。

3. 修正がしやすくなる
処理内容を変更したい場合、メソッドの中だけを修正すればよく、変更箇所が少なくなります。

メソッドの基本形（構造）

メソッドは、一般的に次の要素で構成されます。

戻り値の型（return type）　：　処理結果として返す値の型
メソッド名（method name）　：　処理の名前
引数（parameters）　：　呼び出すときに渡す入力
処理本体（body）　：　実際の処理内容

return 文　　：　　戻り値を返す（戻り値がある場合）

例を示します。

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

この例では、「2 つの数を足し算する」処理を add というメソッドとしてまとめています。

戻り値（return）について

メソッドには、戻り値があるメソッドと、戻り値がないメソッドがあります。

1. 戻り値があるメソッド

処理結果を呼び出し元に返すメソッドです。

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

このメソッドは int 型の結果を返します。

2. 戻り値がないメソッド（void）

処理結果を返さないメソッドです。

```
public void sayHello() {
   System.out.println(" こんにちは ");
}
```

void は「戻り値がない」ことを表します。

引数（パラメータ）について

引数とは、メソッドに渡す入力値です。
メソッドは、引数を受け取ることで、同じ処理でも条件や対象を変えて実行できます。

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

このように、引数によって表示する名前を変えることができます。

メソッドの呼び出し方

メソッドは次のように呼び出します。

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

ここで p.sayHello() は、p が持つ sayHello メソッドを実行しています。

static メソッドとインスタンスメソッド

Java のメソッドには、次の 2 種類があります。

1. インスタンスメソッド

オブジェクト（インスタンス）に対して呼び出すメソッドです。

```
p.sayHello();
```

2. static メソッド

クラスに属し、オブジェクトを作らずに呼び出せるメソッドです。

```
Math.abs(-10);
```

Spring Boot の学習では、最初は「インスタンスメソッド」を中心に理解するとよいです。

Spring Boot とメソッドの関係

Spring Boot では、Controller クラスの中のメソッドが、URL と結び付けられます。

例：

```
@GetMapping("/hello")
public String hello() {
  return "hello";
}
```

この場合、/hello にアクセスすると hello() メソッドが実行されます。
つまり Spring Boot では、「どの URL で、どのメソッドを実行するか」を指定していると
考えると理解しやすくなります。

まとめ（教科書用）

メソッドとは、クラスの中に定義される処理のまとまりです。
処理を再利用し、読みやすくし、修正しやすくするために使用します。
メソッドには、戻り値があるものとないものがあり、引数によって入力を受け取れます。
Spring Boot では、Controller のメソッドが URL と対応して実行されるため、メソッドの理解は特に重要です。

パッケージ（package）

→ クラスを整理するためのフォルダ構造。名前空間の役割を持つ。

パッケージ（package）とは何か

パッケージとは、クラスを整理するための仕組みです。
Java では多くのクラスを扱うため、クラスを無秩序に置いてしまうと管理が難しくなります。
そこで、クラスを分類して整理するために パッケージを使用します。

パッケージは、実体としては ** フォルダ構造（ディレクトリ構造）** で表されます。
また、パッケージには ** 名前空間（namespace）** としての役割もあり、同じクラス名があっても、
パッケージが異なれば別のクラスとして扱われます。

パッケージの基本例

Java ファイルの先頭には、次のように package を宣言します。

```
package com.example.springboot_app;
```

この宣言により、そのクラスは com.example.springboot_app というパッケージに属することになります。

パッケージの 2 つの役割

1. クラスを整理する（分類する）

パッケージを使うと、クラスを役割ごとに分けて整理できます。

例：

controller パッケージ　：画面や URL の処理を担当するクラス
service パッケージ　：業務処理を担当するクラス
mapper パッケージ　：データベース操作を担当するクラス

このように分類することで、プロジェクト全体が見通し良くなり、保守や拡張がしやすくなります。

2. 名前空間として衝突を防ぐ

パッケージは「名前空間」の役割を持ちます。
名前空間とは、同じ名前のクラスが存在しても衝突しないようにする仕組みです。

たとえば、次の 2 つはクラス名が同じ User でも、パッケージが違うため別物として扱われます。

- com.example.domain.User
- com.example.admin.User

この仕組みにより、大規模な開発でもクラス名の衝突を避けられます。

パッケージとフォルダ構造の関係

パッケージは、通常、次のようにフォルダ構造と一致させて配置します。

例：
パッケージが com.example.springboot_app の場合、クラスファイルは次の場所に置きます。

```
src/main/java/com/example/springboot_app/（ここに Java ファイル）
```

つまり、パッケージ名の .（ドット）はフォルダの区切りを表します。

import との関係

別のパッケージにあるクラスを使用する場合は、import を使います。

例：

```
import com.example.springboot_app.mapper.NoteMapper;
```

これにより、NoteMapper クラス（またはインターフェース）を現在のクラスで使用できるようになります。

Spring Boot 開発で特に重要な理由

Spring Boot では、パッケージ構成が適切でないと、Spring がクラスを自動検出できず、
次のようなエラーが発生しやすくなります。

Bean が見つからない（Bean not found）
Controller が認識されず 404 になる
Mapper が見つからない

これは Spring Boot が「指定された範囲のパッケージを探索して、
必要なクラスを自動登録する」仕組みを持っているためです。

そのため、一般的に次の方針が推奨されます。

- @SpringBootApplication の付いたメインクラスは、プロジェクトの上位パッケージに置く
- Controller や Mapper などのクラスは、その配下に配置する

まとめ

パッケージとは、クラスを整理するための仕組みであり、フォルダ構造として表されます。
また、名前空間として機能し、同じクラス名が存在してもパッケージが異なれば別のクラスとして扱われます。
Spring Boot ではクラスの自動検出にパッケージ構成が大きく関わるため、
メインクラスを上位パッケージに置き、各クラスを配下に整理することが重要です。

import

→ 他のパッケージにあるクラスを使用するための宣言。

import とは何か

import とは、他のパッケージにあるクラス（またはインターフェース）を使用するための宣言です。
Java ではクラスをパッケージで整理しますが、別のパッケージにあるクラスを利用したい場合、
そのままではクラス名だけでは指定できません。そこで import を使い、どのクラスを使うのかを明示します。

import が必要になる理由

Java では、同じクラス名が複数のパッケージに存在する可能性があります。
そのため、単に User と書いただけでは、どの User を指しているのか曖昧になる場合があります。

import を使うことで、次のような利点があります。

- クラス名だけで簡潔に書けるようになる
- どのクラスを使っているかを明確にできる
- クラス名の衝突（同名クラス）を避けやすくなる

import の基本例

たとえば、別パッケージにある NoteMapper を使用する場合、次のように記述します。

```
import com.example.springboot_app.mapper.NoteMapper;
```

この宣言があることで、プログラム内では NoteMapper とだけ書けば利用できます。

import がない場合の書き方（完全修飾名）

import を書かない場合は、** 完全修飾名（Fully Qualified Name）** で記述すれば使用できます。

例：

```
com.example.springboot_app.mapper.NoteMapper noteMapper;
```

ただし、毎回長い名前を書くことになるため、通常は import を用いて簡潔に記述します。

import の書き方の位置（どこに書くか）

Java ファイルの基本的な順序は次の通りです。

package 宣言
import 宣言
クラス定義（class）

例：

```
package com.example.springboot_app;
import com.example.springboot_app.mapper.NoteMapper;
public class Sample {
  // ...
}
```

この順序は Java の慣習であり、読みやすさのためにも守ることが推奨されます。

ワイルドカード import（*）について

次のように * を使うと、そのパッケージ内のクラスをまとめて読み込めます。

```
import com.example.springboot_app.mapper.*;
```

しかし、教科書や実務では、次の理由で 必要なクラスを明示して import する方法が推奨されます。

- どのクラスを使っているか分かりにくくなる
- 同名クラスが混ざったときに混乱しやすい
- IDE の自動整理に任せたほうが安全

そのため、基本的には次のように 1 つずつ書くほうが良いです。

```
import com.example.springboot_app.mapper.NoteMapper;
```

public / private

→ クラスやメソッドを「外部から使えるかどうか」を制御する修飾子。

public / private とは何か

public と private は、アクセス修飾子と呼ばれるキーワードです。
アクセス修飾子は、クラス・フィールド・メソッドなどに付けることで、
外部から利用できる範囲（アクセスできる範囲）を制御します。

Java では、必要以上に外部から触れないようにすることで、プログラムを安全に保ち、
誤った操作や想定外の変更を防ぎます。

public（公開する）

public を付けた要素は、どこからでもアクセス可能です。

- 別のクラスから呼び出せます
- 別のパッケージからも呼び出せます
- “ 外部に提供する機能 ” として扱います

例：他のクラスからメソッドを呼べる

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

この add は public なので、別のクラスから次のように呼び出せます。

```
Calculator c = new Calculator();
int result = c.add(3, 5);
```

private（非公開にする）

private を付けた要素は、同じクラスの中からしかアクセスできません。
つまり、外部のクラスからは直接触れません。

クラスの外から勝手に変更・実行されないようにできます
内部処理（裏側の仕組み）を隠せます
データの整合性（正しい状態）を守りやすくなります

例：フィールドを private にして保護する

```
public class BankAccount {
    private int balance = 0; // 残高は外から直接変更させない

    public void deposit(int amount) {
        if (amount > 0) {
            balance += amount;
```

```
        }
    }

    public int getBalance() {
        return balance;
    }
}
```

この例では balance が private なので、外部からこういう操作はできません。

```
account.balance = 1000; // ← エラー（直接アクセス不可）
```

代わりに、deposit() のような 用意した手順（ルール付き）を通して残高を変更させます。
これにより、マイナス入金などの不正な操作を防げます。

なぜアクセス制御が必要なのか

アクセス制御をしないと、外部から好き勝手に内部データを書き換えられます。
すると、次のような問題が起きます。

- 本来あり得ない値が入る（例：年齢が -5、残高が -100 など）
- 処理の順番を間違えられる（例：初期化前に利用される）
- どこが原因で壊れたのか追いにくくなる（バグ調査が困難）

つまり private は「事故防止カバー」です。
カバーを外して中のギアを指で触らせたら、だいたい壊れます（そして壊した本人は気づきません）。

典型的な使い分け

・public：外部に提供する入口（呼び出してよい操作）
例：login()、register()、calculate() など

・private：内部の都合（外部に触らせない部品）
例：validate()、encrypt()、計算途中の変数、状態管理のフィールドなど

戻り値（return）

→ メソッドの処理結果として呼び出し元に返す値。

戻り値（もどりち）とは、メソッドが処理した結果を、呼び出し元（そのメソッドを使った側）へ返す値のことです。
Java では、メソッドの最後などで return を使って値を返します。

戻り値があるメソッドとないメソッド
1) 戻り値がある（結果を返す）

メソッド宣言の先頭に 戻り値の型を書きます（例：int、String など）。

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

このメソッドは int 型を返すため、呼び出し側は結果を受け取れます。

```
int result = add(3, 5);  // result は 8 になります
```

2) 戻り値がない（結果を返さない）

戻り値がないメソッドは、戻り値の型に void を指定します。

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

この場合、呼び出しても値は受け取れません

```
greet(); // 表示はされますが、値は返りません
```

return の役割は 2 つあります

役割①：処理結果を返す

戻り値があるメソッドでは、return 値 ; によって 結果を呼び出し元へ渡します。

```
public String makeMessage(String name) {
  return name + " さん、ようこそ ";
}
```

呼び出し元：

```
String msg = makeMessage(" 田中 ");
System.out.println(msg); // 田中さん、ようこそ
```

役割②：メソッドの処理をそこで終了する

return が実行されると、その時点でメソッドの処理は終わり、呼び出し元へ戻ります。
そのため、return の後に書いた処理は実行されません。

```
public int check(int x) {
  if (x >= 0) {
    return 1; // ここで終了
```

```
    }
    return -1;   // x が負のときはこちら
}
```

戻り値の型と return の値は一致させる必要があります

メソッドが int を返すと宣言したなら、return でも int を返さなければいけません。

```
public int getAge() {
    return 20; // OK（int）
}
```

これはエラーになります（型が違うためです）。

```
public int getAge() {
    return "20"; // NG（String は返せません）
}
```

引数（parameter）

→ メソッドに渡す入力データ。

引数（parameter）とは

引数（ひきすう）とは、メソッドに渡す入力データのことです。
メソッドは「処理をまとめた部品」ですので、引数を使うことで 状況に応じて中身を変えて
動かせる部品になります。

引数は「メソッドの入口に渡す値」です

たとえば「2 つの数を足す」メソッドを考えます。毎回決まった数を足すだけでは不便です。
そこで、足したい数を外から渡せるようにします。その " 渡す数 " が引数です。

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

int a と int b が引数（parameter）です
呼び出し側から 3 と 5 を渡すと、その値が a と b に入ります

呼び出し例：

```
int result = add(3, 5); // a=3, b=5 として動く
```

用語の整理：引数（parameter）と実引数（argument）

日本語の学習では混ざりやすいので、教科書では次のように整理すると分かりやすいです。

- 仮引数（parameter）：メソッド定義側に書く「受け取るための変数」
- 実引数（argument）：呼び出し側で実際に渡す「値」

例：

```
public void greet(String name) { // ← name が仮引数（parameter）
  System.out.println(" こんにちは、" + name + " さん ");
}

greet(" 田中 "); // ← " 田中 " が実引数（argument）
```

引数があると「汎用的なメソッド」になります

引数がないメソッドは、毎回同じ動きになりがちです。

```
public void printHello() {
  System.out.println("Hello");
}
```

一方、引数があると「名前を変える」「金額を変える」など、使い回しができます。

```
public void printMessage(String msg) {
  System.out.println(msg);
}

printMessage(" おはようございます ");
printMessage(" お疲れ様です ");
```

引数は複数持てます（0 個も可能です）

- 引数が 0 個：入力不要（毎回同じ処理になりやすい）
- 引数が 1 個以上：入力に応じて処理を変えられる

```
public void noArgs() { }                    // 0 個
public void oneArg(int x) { }               // 1 個
public void twoArgs(String a, int b) { }    // 2 個
```

引数には「型」が必要です（Java のルール）

Java は型が厳格なので、引数にも必ず型を書きます。

```
public void setAge(int age) { ... }
public void setName(String name) { ... }
```

呼び出し側で渡す値も、その型に合っている必要があります。

```
setAge(20);    // OK
setAge("20");  // NG（String を int には渡せません）
```

フレームワーク / Spring の基本概念

フレームワーク
→ アプリケーションの土台を提供し、開発の流れを決める仕組み。

ライブラリ
→ 必要なときに呼び出して使う部品集。

Spring
→ Java アプリケーションを部品単位で管理するフレームワーク。

Spring Boot
→ Spring を「設定最小」で使えるようにした仕組み。

設定より規約（Convention over Configuration）
→ 細かい設定を省略し、決められた書き方に従う考え方。

DI / Bean / コンテナ

DI（依存性注入）

→ クラスが必要とする部品を、自分で作らず外部から渡してもらう仕組み。

DI（依存性注入：Dependency Injection）とは

DI（依存性注入）とは、クラスが処理に必要とする部品（オブジェクト）を、自分自身で作成せず、
外部から与えてもらう設計の考え方です。
日本語では「依存性注入」と呼ばれます。

「依存している」とはどういうことか

あるクラスが、別のクラスを使って処理を行っている場合、
そのクラスは その部品（クラス）に依存している と言います。

```
public class UserService {

    private UserRepository repository = new UserRepository();

    public void register() {
        repository.save();
    }
}
```

この場合、

- UserService は
- UserRepository を 自分で new して使っている

つまり、UserService は UserRepository に 強く依存している状態です。

この書き方の問題点

このように「自分で部品を作る」設計には、次の問題があります。

1. 部品を差し替えにくい
   テスト用の Repository に変更したい場合でも、コードを書き換える必要がある

2. クラス同士が強く結びつく
   変更の影響範囲が広がり、保守が大変になる

3. テストがしづらい
   ダミーやモックを使えない

この「強く結びついた状態」を 密結合 と呼びます。

DI の考え方：部品は「外から渡す」

DI では、クラス自身が部品を作りません。
「必要です」と宣言し、外部から渡してもらいます。

```
public class UserService {

    private UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    public void register() {
        repository.save();
    }
}
```

ここでは、

- UserService は UserRepository を 自分で new していない
- コンストラクタを通して 外部から受け取っている

この「外部から渡す」ことが 注入（Injection）です。

DI を使うと何が良くなるのか

① クラスの役割が明確になります

クラスは「処理の内容」だけに集中でき、
「部品の作り方」を考えなくて済みます。

② 部品を簡単に差し替えられます

同じインターフェースを実装した別クラスに、簡単に切り替えられます。

```
UserRepository repository = new TestUserRepository();
UserService service = new UserService(repository);
```

③ テストがしやすくなります

テスト用のダミークラス（モック）を渡せるため、
データベースなどに依存しないテストが可能になります。

Spring Boot と DI

Spring Boot では、DI は フレームワークが自動で行います。
開発者が new を書く必要はほとんどありません。

```
@Service
public class UserService {

  private final UserRepository repository;

  public UserService(UserRepository repository) {
    this.repository = repository;
  }
}
```

@Service や @Repository を付けると
Spring がクラスを管理し
必要な部品を 自動で注入 します

これにより、部品の組み立てを人がやらなくてよい設計になります。

Bean

→ Spring が管理しているオブジェクト。

Bean（ビーン）とは

Bean（ビーン）とは、**Spring が作成し、管理しているオブジェクト（インスタンス）** のことです。
言い換えると、アプリケーション内で使う部品を Spring が「生成・保管・受け渡し」まで面倒を見ている状態
オブジェクトを Bean と呼びます。

1. そもそも「Spring が管理する」とは何をするのか

Spring は、Bean に対して主に次のことを行います。

- 生成する（new する作業を代わりに行う）
- 必要な部品を注入する（DI：依存性注入）
- 必要な設定を適用する（設定値、プロパティなど）
- 必要があれば破棄する（終了時の後始末）
- 同じ部品の使い回しを制御する（後述のスコープ）

つまり、Bean は「ただのオブジェクト」ではなく、
Spring の管理下でライフサイクル（生まれてから消えるまで）を制御される部品です。

2. Bean があると何がうれしいのか（目的）

Bean を使う最大のメリットは、次の 2 点です。

- new を書かなくてよい（部品を自分で作らない）
- 部品の組み立て（依存関係の配線）を Spring に任せられる

結果として、

- 差し替えがしやすい
- テストがしやすい
- 設計が疎結合になり、保守性が上がる

という利点が得られます。

3. Bean はどうやって作られるのか（代表的な方法）

方法①：アノテーションで登録する（最も一般的）

```
@Service
public class UserService {
}
```

@Service が付いたクラスは、Spring が見つけて Bean として登録します。
同様に、次も Bean 登録に使われます。

- @Component（汎用）
- @Service（サービス層の部品）
- @Repository（データアクセス層の部品）
- @Controller / @RestController（Web 層の部品）

方法②：設定クラス（@Configuration）で登録する

「このクラスは自分で作りたい」「生成方法を細かく指定したい」場合に使います。

```
@Configuration
public class AppConfig {

  @Bean
  public UserService userService(UserRepository repo) {
    return new UserService(repo);
  }
}
```

4. Bean が DI にどう関係するのか

例として、次のようなコードを考えます。

```
@Service
public class UserService {

  private final UserRepository repository;

  public UserService(UserRepository repository) {
    this.repository = repository;
  }
}
```

ここで起きていることは次の通りです。

1. Spring が UserRepository（Bean）を用意する
2. Spring が UserService（Bean）を作るときに
3. コンストラクタ引数として UserRepository を渡す（注入する）

開発者は new UserRepository() を書いていませんが、
Spring が Bean を用意して注入してくれるため、動作します。

5. Bean の「スコープ」（使い回し方）

Bean は「いつ作られ、どれくらい使い回すか」を設定できます。
代表的には次の 2 つが重要です。

- singleton（シングルトン）：基本設定。アプリ全体で 1 つを使い回す
- request（リクエスト）：Web の 1 リクエストごとに新しく作る

通常、サービスやリポジトリは singleton で使われます。
つまり、毎回 new されるのではなく、同じインスタンスが再利用されます。

コンテナ

→ Bean を作成・管理・連携する Spring の中核機能。

コンテナ（Spring コンテナ）とは

Spring における「コンテナ」とは、Bean（Spring が管理するオブジェクト）を作成し、管理し、
必要な場所へ受け渡す仕組みのことです。
正式には IoC コンテナ（Inversion of Control：制御の反転）と呼ばれ、Spring の中核機能になっています。

1. コンテナが行うこと（何をしてくれるのか）

Spring コンテナは、主に次の仕事を行います。

① Bean を作成する（生成）
開発者が new で作る代わりに、コンテナがオブジェクトを生成します。

② 依存関係を解決して注入する（DI）
ある Bean が別の Bean を必要とする場合、コンテナがそれを用意し、適切に渡します。

③ Bean を保存して使い回す（スコープ管理）
通常の Bean はアプリ全体で 1 つだけ作って使い回します（singleton が基本です）。

④ 初期化・破棄などライフサイクルを管理する
起動時に初期化し、終了時に後片付けを行います。

2.「Bean が入っている箱」というイメージで理解できます

コンテナはよく **「部品箱」** に例えられます。
部品（Bean）を箱に入れて管理しておき
必要になったら取り出して渡す

ただし、実際は「保管」だけではありません。
部品を作り、配線し、正しい順番で準備するところまで行うため、単なる箱よりも高機能です。

3. コンテナがない世界（自分で全部組み立てる）

コンテナがないと、開発者が部品をすべて new で作り、手でつなぎます。

```
UserRepository repo = new UserRepository();
UserService service = new UserService(repo);
UserController controller = new UserController(service);
```

この方法でも動きますが、次の問題が起こりやすくなります。

- 部品の数が増えるほど組み立てが複雑になる
- 差し替えやテストが難しくなる
- どこで何を作っているか追いにくくなる

4. コンテナがある世界（Spring が組み立てる）

Spring では、クラスにアノテーションを付けておくと、コンテナが部品を自動で登録・組み立てします。

```
@Repository
public class UserRepository { }

@Service
public class UserService {
  private final UserRepository repo;
  public UserService(UserRepository repo) { this.repo = repo; }
}

@Controller
public class UserController {
  private final UserService service;
  public UserController(UserService service) { this.service = service; }
}
```

ここで開発者は new を書かずに済みます。
どの部品が必要か（依存関係）だけを書けば、組み立てはコンテナが担当します。

- pring のコンテナとは、Bean を作成し管理する中核機能である
- コンテナは Bean の生成、依存関係の解決（DI）、スコープ管理、ライフサイクル管理を行う
- 開発者は部品の組み立てを意識せず、必要な依存関係を宣言するだけでよい
- Spring Boot ではコンテナの準備が自動化され、開発者は直接扱う場面が少ない

アノテーション

アノテーション

→ クラスやメソッドに役割を指定するための目印。

```
@SpringBootApplication
```

→ Spring Boot アプリケーションの開始点を示す。

@SpringBootApplication とは

@SpringBootApplication は、Spring Boot アプリケーションの開始点（エントリーポイント）であることを示すアノテーションです。
このアノテーションを付けたクラスを起点として、Spring Boot はアプリケーションを起動し、
必要な設定や部品（Bean）の準備を自動で行います。

一般的には、次のように main メソッドを持つクラスに付けて使います。

```
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

1.「開始点」とは具体的に何を指すのか

@SpringBootApplication が付いたクラスは、Spring Boot にとって次の意味を持ちます。

- このクラスを基準に、アプリ全体の設定と起動を開始します
- このクラスが属するパッケージを起点に、部品（Bean）を探して登録します
- Web アプリであれば、サーバー（内蔵 Tomcat など）を起動します

つまり、アプリ起動時の「司令塔」の位置を Spring Boot に教える役割があります。

2. @SpringBootApplication の中身（実は 3 つの機能のまとめ）

@SpringBootApplication は、実際には次の 3 つをまとめた「セット商品」です。

1.@SpringBootConfiguration
　このクラスが「設定クラスである」ことを示します（Spring の設定の中心になります）

2.@EnableAutoConfiguration
　Spring Boot が状況を判断して、必要な設定を自動で有効化します
　例：Web の依存関係があれば Web 用設定、DB があれば DataSource 設定など

3.@ComponentScan
　指定した範囲のクラスを探し、@Component / @Service / @Repository / @Controller などを Bean として登録します

この 3 点が重要です。
特に「自動設定」と「部品探し（スキャン）」が一緒に有効になることが、Spring Boot を " 書いたら動く " 状態にしています。

3. どこに置くべきか（重要）

@SpringBootApplication を付けたクラスは、通常 プロジェクトの最上位パッケージに置きます。
理由は @ComponentScan の探索範囲が「このクラスのパッケージ配下」になるためです。

例：

jp.co.example.app.Application（最上位）
jp.co.example.app.controller
jp.co.example.app.service
jp.co.example.app.repository

この構成にすると、配下の @Controller や @Service が自動的に見つかり、Bean として登録されます。

逆に、開始点クラスを深い階層に置くと、上位パッケージ側のクラスがスキャン対象外になり、
Bean が見つからずにエラーになることがあります。

4. @SpringBootApplication があると何が起きるか（起動の流れ）

起動時に大まかに次のことが行われます。

1. SpringApplication.run(...) が実行される
2. Spring の コンテナ（ApplicationContext）が作られる
3. @ComponentScan により、対象クラスを探して Bean 登録する
4. @EnableAutoConfiguration により、必要な設定が自動で有効になる
5. Web アプリの場合、内蔵サーバーが起動する
6. アプリがリクエスト受付状態になる

@SpringBootApplication は Spring Boot アプリケーションの開始点を示すアノテーションである
アプリ起動時の設定読み込み、部品（Bean）の探索・登録、自動設定の有効化をまとめて行う
一般的に、最上位パッケージに置くことで、アプリ全体のクラスが正しくスキャンされる

@Controller

→ 画面遷移やリクエストを処理するクラスであることを示す。

@Controller とは

@Controller は、このクラスが Web アプリケーションの「コントローラ」であることを Spring に伝えるためのアノテーションです。
コントローラは、ブラウザなどから届くリクエスト（要求）を受け取り、必要な処理を行い、どの画面

（テンプレート）を表示するかを決定します。

1. コントローラの役割（何を担当するのか）

Spring MVC の基本構造（MVC）では、役割分担があります。

- Controller（コントローラ）：受付係（リクエストを受け、処理の指示を出し、画面を決める）
- Service（サービス）：業務処理の中心（登録、計算、判定など）
- Repository / Mapper（リポジトリ／マッパー）：データ取得・保存（DB とのやりとり）
- View（ビュー）：画面（Thymeleaf などのテンプレート）

このうち @Controller は「受付係」にあたります。
画面表示や画面遷移を含む Web の入口を担当します。

2. @Controller を付けると何が起きるのか

@Controller を付けると、Spring は次のように扱います。

1. そのクラスを Bean（管理対象）として登録します
2. @GetMapping や @PostMapping が付いたメソッドを探し、URL と処理を結び付けます
3. リクエストが来たときに、対応するメソッドを呼び出します
4. そのメソッドの戻り値などを使って、表示する画面や遷移先を決めます

3. 画面を返す

```
@Controller
public class HomeController {

  @GetMapping("/")
  public String home(Model model) {
    model.addAttribute("message", " ようこそ ");
    return "index";
  }
}
```

このコードの意味は次の通りです。

- ブラウザが / にアクセスすると
- home() メソッドが呼ばれる
- Model に message を入れる（画面へ渡すデータ）
- return "index"; により index.html（テンプレート）を表示する

※ Thymeleaf を使う場合、通常は templates/index.html が表示対象になります。

4. 画面遷移（redirect）もコントローラの仕事

コントローラは、処理後に別の URL へ移動させることもよく行います。

```
@PostMapping("/save")
public String save() {
  // 登録処理（省略）
  return "redirect:/";
}
```

redirect:/ は「/ に移動してください」という指示です

画面の二重送信（リロードで再登録）を防ぐためにもよく使われます

5. @Controller と @RestController の違い（混同注意）

ここは授業でつまずきやすいので、はっきり整理します。

@Controller
主に 画面（HTML）を表示するために使います
return "index"; のように テンプレート名を返すことが多いです

@RestController
主に API（JSON など）を返すために使います
return "Hello"; が 文字列や JSON として返るのが基本です
同じ return でも意味が変わるため、ここは重要です。

@GetMapping

→ GET リクエストとメソッドを結びつける。

@GetMapping は、ブラウザなどから送られてくる「GET リクエスト」と、Java のメソッドを結び付ける（関連付ける）ためのアノテーションです。
Spring MVC では、どの URL にアクセスされたときに、どのメソッドを実行するかを決める必要があります。
その対応付けを行うのが @GetMapping です。

1. GET リクエストとは何か

GET リクエストは、主に次の目的で使われます。

- 画面を表示する
- データを取得する
- 検索結果を表示する

例：ブラウザで URL を開く操作は、基本的に GET です。

- https://example.com/
- https://example.com/users
- https://example.com/search?keyword=java

GET は「取得（読み取り）」が目的であり、登録・更新・削除のような変更処理には通常使いません。

2. @GetMapping の基本的な書き方

次の例では、/hello に GET アクセスされたときに hello() が実行されます。

```
@Controller
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "hello";
  }
}
```

この意味は次の通りです。

ブラウザが /hello にアクセスする（GET）

- Spring が @GetMapping("/hello") を見て、対応するメソッドを探す
- hello() を実行する
- return "hello"; により、テンプレート（例：hello.html）を表示する

3. URL に何も指定しない場合

@GetMapping のパスが / の場合は、トップページなどで使われます。

```
@GetMapping("/")
public String home() {
  return "index";
}
```

4. パラメータ（クエリ文字列）を受け取る例

GET では、URL に ?key=value の形で値を付けて送ることがよくあります。

例：/search?keyword=java

これを受け取るには @RequestParam を使います。

```
@GetMapping("/search")
public String search(@RequestParam("keyword") String keyword, Model model) {
  model.addAttribute("keyword", keyword);
  return "search";
```

```
    }
```

このときの動作は次の通りです。

- /search?keyword=java にアクセスすると
- keyword というパラメータの値 "java" が keyword 変数に入る
- その値を画面に渡して表示できる

@PostMapping

→ POST リクエストとメソッドを結びつける。

1. POST リクエストとは何か

POST リクエストは、主に次の目的で使われます。

- 登録（作成）
- 更新
- 削除
- ログイン処理
- フォーム送信

サーバー側のデータや状態を変更する処理に使います。

2. @PostMapping の基本的な書き方

次の例では、/save に POST が送られたときに save() が実行されます。

```
@Controller
public class UserController {

  @PostMapping("/save")
  public String save(@RequestParam("name") String name) {
    // ここで登録処理などを行う（省略）
    return "result";
  }
}
```

この意味は次の通りです。

- フォームなどから /save に POST 送信される
- Spring が @PostMapping("/save") を見て、対応するメソッドを探す
- save() が実行され、送信されたデータ（name）を受け取る
- 処理結果として表示する画面（例：result.html）を返す

3. HTML フォーム送信と @PostMapping

HTML のフォームで method="post" を指定すると POST になります。

```
<form action="/save" method="post">
 <input type="text" name="name">
 <button type="submit"> 送信 </button>
</form>
```

- name という入力欄の値が POST で送られます
- Spring 側では @RequestParam("name") で受け取れます

4. POST の後は redirect を使うことが多い（重要）

POST で登録処理を行ったあとに、次のように redirect: を返すことがよくあります。

```
@PostMapping("/save")
public String save(@RequestParam("name") String name) {
  // 登録処理（省略）
  return "redirect:/";
}
```

これは 二重送信を防ぐためです。
POST の結果画面のままブラウザで再読み込みすると、同じ POST が再送信されることがあります
redirect:/ によって GET の画面に移動させると、その問題が起きにくくなります
この流れは PRG パターン（Post/Redirect/Get）と呼ばれ、Web 開発でよく使われます。

5. @PostMapping と @GetMapping の使い分け

基本の考え方は次の通りです。

- @GetMapping：表示・取得（読む処理）
- @PostMapping：登録・更新など（書く処理）

例として、フォームを扱う場合は次のように分けます。

- GET：入力フォームを表示する
- POST：入力内容を送信して登録する

@PostMapping は、HTTP の POST リクエストとメソッドを結び付けるアノテーションである
POST は主に、登録・更新・削除などサーバー側の状態を変更する処理に用いられる
フォーム送信では method="post" により POST が送られ、@RequestParam などで受け取れる
POST 後は二重送信防止のため redirect: を用いることが多い

```
@Autowired
```

→ 必要な Bean を自動的に注入する。

@Autowired は、Spring が管理している Bean（オブジェクト）を、自動的に注入（DI）するためのアノテーションです。
開発者が new で部品を作って渡す代わりに、Spring コンテナが「必要な部品」を探して用意し、指定した場所へ渡してくれます。

1.「注入する」とは何をすることか

「注入（DI）」とは、簡単に言うと次の意味です。

- あるクラスが処理のために別の部品（Bean）を必要とするとき
- その部品を 自分で作らず
- Spring が 外部から渡す（セットする）

@Autowired は、この「外部から渡す」を Spring に依頼するための目印です。

2. @Autowired の代表的な使い方

（A）コンストラクタ注入（推奨）
現在の Spring では、最も推奨される書き方です。

```
@Service
public class UserService {

    private final UserRepository repository;

    @Autowired
    public UserService(UserRepository repository) {
        this.repository = repository;
    }
}
```

UserService を作るとき、Spring が UserRepository を探して渡します
UserRepository も Bean として登録されている必要があります

※実際には コンストラクタが 1 つだけの場合、@Autowired は省略できます（後述します）。

3. Spring はどうやって「必要な Bean」を見つけるのか

@Autowired が付いた場所に対して、Spring は次の順序で探します。

1. ** 型（クラスやインターフェース）** が一致する Bean を探す
2. 候補が 1 つなら、それを注入する
3. 候補が複数ある場合は、区別が必要になる（例：@Qualifier を使う）

例：同じ型の Bean が 2 つあると、どちらを入れるか判断できずエラーになります。

4. @Autowired は「省略できる」ことがある（重要）

コンストラクタが 1 つだけの場合、Spring は「ここに注入すればよい」と判断できるため、
次のように @Autowired を省略できます。

```
@Service
public class UserService {

  private final UserRepository repository;

  public UserService(UserRepository repository) { // @Autowired なしでも注入される
    this.repository = repository;
  }
}
```

@Autowired は、必要な Bean を Spring が自動的に注入するためのアノテーションである
注入とは、クラスが必要とする部品を自分で作らず、外部（Spring コンテナ）から渡してもらうことである
代表的な注入方法には、コンストラクタ注入、フィールド注入、セッター注入がある
コンストラクタが 1 つだけの場合は、@Autowired を省略できることが多い

@Autowired は「new を消す魔法」ではなく、
new を Spring コンテナに担当させるための合図です。

MVC（画面構成）

MVC モデル

→ アプリケーションを役割ごとに分ける設計思想。

MVC モデルとは

MVC モデルとは、アプリケーションを ** 役割ごとに分けて設計する考え方（設計思想）** です。
MVC は次の 3 つの頭文字を取ったものです。

- M：Model（モデル）
- V：View（ビュー）
- C：Controller（コントローラ）

この 3 つに役割を分けることで、プログラムが整理され、変更に強くなります。

1. なぜ役割分担が必要なのか

Web アプリには、次のように性質の違う処理が混在します。

- 画面を表示する処理（HTML、テンプレート）
- ユーザー操作を受け取る処理（ボタン押下、入力値）
- データを扱う処理（登録、検索、更新、DB 操作）
- 業務ルールの処理（判定、計算、チェック）

これらを 1 つのクラスや 1 つのファイルにまとめてしまうと、次の問題が起きやすくなります。
- どこに何が書いてあるか分かりにくい
- 修正の影響範囲が広がる
- テストしにくい
- 変更が増えるほど壊れやすくなる

MVC は、こうした問題を避けるために生まれた考え方です。

Model

→ データを扱う部分。

2. Model（モデル）の役割

Model は、データや業務処理（ビジネスロジック）を担当します。

- データの入れ物（例：User、Product など）
- 業務ルール（例：在庫が 0 なら購入不可、金額計算、入力チェック）
- データ操作（例：DB から取得、DB へ保存）※プロジェクトによって分割します

Spring の現場では、Model をさらに分けることが多く、たとえば次のように役割を分割します。

- エンティティ（データ）
- サービス（業務処理）
- リポジトリ／マッパー（DB アクセス）

View

→ 画面表示を担当する部分。

3. View（ビュー）の役割

View は、画面表示を担当します。

- HTML（テンプレート）
- 画面の見た目（CSS）
- 表示のための構造

View は「データをどう見せるか」が役割であり、原則として 複雑な業務処理は書きません。
（例：Thymeleaf のテンプレート、JSP、React の画面など）

Controller

→ リクエストを受け取り、処理と画面を制御する部分。

4. Controller（コントローラ）の役割

Controller は、ユーザーからのリクエストを受け取り、必要な処理を呼び出し、表示する画面（View）を決める役割です。

- URL へのアクセスを受け取る
- 入力値を受け取る
- Service を呼び出す
- 結果を View に渡す
- 画面を返す、またはリダイレクトする

Spring MVC では、@Controller や @GetMapping / @PostMapping を使って Controller を作ります。

5. MVC の処理の流れ（全体像）

Web アプリでの典型的な流れは次の通りです。

1. ユーザーがブラウザで操作する（リクエスト送信）
2.Controller がリクエストを受け取る
3.Model（Service 等）が業務処理やデータ処理を行う
4.Controller が結果を View に渡す
5.View が画面を表示する

- MVC モデルは、アプリケーションを Model・View・Controller の 3 つの役割に分ける設計思想である
- Model はデータと業務処理、View は画面表示、Controller はリクエスト受付と処理の振り分けを担当する
- 役割分担により、プログラムが整理され、修正や保守がしやすくなる

画面表示（Thymeleaf）

テンプレートエンジン

→ HTML に動的なデータを埋め込む仕組み。

テンプレートエンジンとは

テンプレートエンジンとは、あらかじめ用意した HTML（ひな形）に、プログラム側のデータを埋め込み、完成した HTML を作る仕組みです。
Web アプリでは、ユーザーごと・状況ごとに表示内容が変わることが多いため、テンプレートエンジンを使って動的なページを生成します。

1.「HTML にデータを埋め込む」とはどういうことか

HTML をそのまま用意すると、内容は固定になります。たとえば次の HTML は、誰が見ても同じ表示です。

```
<h1> ようこそ </h1>
<p> 本日の予約はありません。</p>
```

しかし実際の業務では、

- ログインしている人の名前を表示したい
- 一覧を DB から取得して表示したい
- 条件によって表示を変えたい（空なら「データがありません」など）

といった要望が出ます。

そこで、テンプレートエンジンは HTML の中に「差し込み場所」や「繰り返し」「条件分岐」を
書けるようにし、最終的な HTML を組み立てます。

2. テンプレートエンジンでできる代表的なこと

テンプレートエンジンには次のような機能があります。

① 変数の埋め込み（値の表示）
例：ユーザー名を表示する

② 繰り返し（一覧表示）
例：商品リスト、予約一覧、顧客一覧など

③ 条件分岐（表示の出し分け）
例：データが空なら「データがありません」を表示

④ レイアウトの共通化
例：ヘッダー・フッターを共通部品として使う

3. Spring Boot でよく使うテンプレートエンジン（Thymeleaf）

Spring Boot では、代表的に **Thymeleaf（タイムリーフ）** がよく使われます。
Thymeleaf は HTML に近い形で書けるため、学習教材でも扱いやすい利点があります。

例：値の埋め込み

Controller 側で Model に値を入れます。

```
@GetMapping("/hello")
public String hello(Model model) {
```

```
    model.addAttribute("name", " 田中 ");
    return "hello";
}
```

テンプレート（hello.html）側では、埋め込みを記述します。

```
<p th:text="${name}"> ここに名前が表示されます </p>
```

この結果、ブラウザには次のような完成 HTML が表示されます。

```
<p> 田中 </p>
```

ポイントは、** ブラウザが受け取るのは「完成した HTML」** であり、
th:text のような記述は最終的に消えることです。

4. テンプレートエンジンを使う流れ（全体像）

Web アプリでの流れは次の通りです。

1. ブラウザが URL にアクセスする
1.Controller がリクエストを受け取る
1.Controller が必要なデータを用意し、Model に入れる
1. テンプレートエンジンが HTML テンプレートにデータを埋め込む
1. 完成した HTML がブラウザに返され、表示される

つまりテンプレートエンジンは、Controller が用意したデータを、画面として見える形（HTML）に変換する
役割を担っています。

5. テンプレートエンジンと SPA（React/Vue）との違い（補足）

テンプレートエンジンは、サーバー側で HTML を完成させて返す方式です。
一方、Vue/React のような SPA は、ブラウザ側（JavaScript）で画面を組み立てる方式です。

どちらが正しいという話ではなく、目的や構成に応じて使い分けます。

Thymeleaf

→ Spring Boot で標準的に使われるテンプレートエンジン。

Thymeleaf（タイムリーフ）とは

Thymeleaf は、Spring Boot でよく利用されるテンプレートエンジンです。

テンプレートエンジンとは、HTML に対して 動的なデータ（変数や一覧データなど）を埋め込み、
最終的な HTML を生成する仕組みのことです。

Thymeleaf を使うことで、ログインユーザー名の表示、一覧の表示、条件による表示切り替えなどを、
HTML テンプレート内で行えるようになります。

1. Thymeleaf の特徴

① HTML に近い書き方ができる
Thymeleaf は「HTML をベースに拡張する」方式のため、見た目が HTML と非常に近いです。
そのため、学習者にとっても理解しやすく、実務でも扱いやすい利点があります。

② ブラウザで見ても崩れにくい（自然なテンプレート）
Thymeleaf は「自然なテンプレート（Natural Template）」と呼ばれる考え方を持ちます。
Thymeleaf の属性（例：th:text）を使っていても、HTML として成立しやすい構造になっています。

③ Spring MVC と相性が良い
Spring の Controller から Model に入れた値を、テンプレート側で簡単に表示できます。

2. Thymeleaf でできる代表的なこと

Thymeleaf では、主に次のような処理ができます。

- 値の表示（変数の埋め込み）
- 繰り返し（一覧表示）
- 条件分岐（ある場合だけ表示）
- リンク生成（URL を安全に生成）
- フォーム連携（入力値の受け渡し、エラー表示など）
- レイアウト共通化（ヘッダー・フッターの部品化）

3. Thymeleaf のテンプレートはどこに置くのか

Spring Boot の標準設定では、Thymeleaf の HTML ファイルは通常次の場所に置きます。
- src/main/resources/templates/

例：
- src/main/resources/templates/hello.html

このとき、Controller の return "hello"; は、hello.html を探して表示することを意味します。

model.addAttribute

→ Controller から View にデータを渡す方法。

model.addAttribute とは

model.addAttribute は、Controller から View（画面テンプレート）へデータを渡すための方法です。

Spring MVC では、Controller が処理の結果として「画面に表示したい値」を用意し、その値を Model に入れて View に引き渡します。
その「Model に値を入れる」操作が model.addAttribute です。

1. Model（モデル）とは何か（ここでの意味）

ここでの Model は、MVC の「Model（業務データ）」という広い意味ではなく、
「画面に渡すデータを一時的に入れておく入れ物」 です。

Controller が View に渡したい情報を入れる箱
View（Thymeleaf など）は、その箱から値を取り出して表示する

と考えると分かりやすいです。

2. addAttribute の基本形

```
model.addAttribute(" 名前 ", 値 );
```

- 第 1 引数：View 側で参照するための キー（名前）
- 第 2 引数：渡したい 値（文字列、数値、オブジェクト、リストなど）

3. 具体例：文字列を渡す

Controller

```
@GetMapping("/hello")
public String hello(Model model) {
  model.addAttribute("message", " こんにちは ");
  return "hello";
}
```

Thymeleaf（hello.html）

```
<p th:text="${message}"> ここに表示 </p>
```

この結果、画面には「こんにちは」と表示されます。

4. 具体例：オブジェクトを渡す

Controller

```
@GetMapping("/user")
public String user(Model model) {
  User u = new User(" 田中 ", 30);
```

```
    model.addAttribute("user", u);
    return "user";
}
```

Thymeleaf（user.html）

```
<p th:text="${user.name}"> 名前 </p>
<p th:text="${user.age}"> 年齢 </p>
```

このように、オブジェクトの中身も参照できます。

5. 具体例：一覧（List）を渡す（繰り返し表示）

Controller

```
@GetMapping("/users")
public String users(Model model) {
    List<String> users = List.of(" 田中 ", " 佐藤 ", " 鈴木 ");
    model.addAttribute("users", users);
    return "users";
}
```

Thymeleaf（users.html）

```
<ul>
  <li th:each="u : ${users}" th:text="${u}"> 名前 </li>
</ul>
```

リストの要素数だけ <li> が繰り返され、一覧表示ができます。

6. addAttribute はどこまで有効か

model.addAttribute で入れた値は、基本的に そのリクエスト内で View を表示するために使われます。
別のページへ redirect: で移動した場合、通常は引き継がれません。

```
return "redirect:/";
```

この場合は「別のリクエスト」になるため、Model の中身はそのままでは渡りません。
（※別の仕組みとして RedirectAttributes などがあります）

Spring Boot × DB

JDBC

→ Java からデータベースに接続するための仕組み。

JDBC とは

JDBC（ジェイディービーシー）とは、Java からデータベース（DB）に接続し、SQL を実行して、データの取得や更新を行うための仕組みです。
正式には Java Database Connectivity の略で、Java がさまざまな種類のデータベースを共通の方法で扱えるようにするための標準仕様です。

1. JDBC でできること

JDBC を使うと、主に次のことができます。

- データベースへ接続する
- SQL（SELECT / INSERT / UPDATE / DELETE）を実行する
- 実行結果（検索結果）を受け取って Java のデータとして扱う
- 接続を終了する（後片付け）

つまり、**「Java と DB の橋渡し」** が JDBC の役割です

2. JDBC は「ドライバ」を通して DB と会話する

JDBC では、データベースごとに用意された JDBC ドライバを使って通信します。
- MySQL 用ドライバ
- PostgreSQL 用ドライバ
- Oracle 用ドライバ など

Java から見ると「JDBC という共通ルールで操作」し、
内部では「ドライバが DB の言葉に翻訳して通信」します。

3. JDBC を使った基本的な流れ

JDBC の処理は、次の順番で進みます。

1. DB に接続する（Connection を取得する）
2. SQL を準備する（Statement / PreparedStatement）
3. SQL を実行する（検索 or 更新）
4. 結果を受け取る（ResultSet）
5. 後片付けをする（接続を閉じる）

4. 例：検索（SELECT）の基本形

```
String sql = "SELECT id, name FROM users WHERE id = ?";
```

```
try (Connection con = DriverManager.getConnection(url, user, pass);
    PreparedStatement ps = con.prepareStatement(sql)) {

  ps.setInt(1, 10);          // ? の場所に値を入れる
  ResultSet rs = ps.executeQuery(); // SELECT 実行

  if (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
  }
}
```

ここで登場する主な部品は次の通りです。

- Connection：DB への接続（通路）
- PreparedStatement：SQL を安全に実行するための仕組み
- ResultSet：検索結果（表データ）を受け取る入れ物

5. 例：更新（INSERT / UPDATE / DELETE）の基本形

更新系の SQL では executeUpdate() を使います。

```
String sql = "INSERT INTO users(name) VALUES (?)";

try (Connection con = DriverManager.getConnection(url, user, pass);
    PreparedStatement ps = con.prepareStatement(sql)) {

  ps.setString(1, " 田中 ");
  int count = ps.executeUpdate(); // 変更された行数が返る
}
```

6. Spring Boot では JDBC をどう扱うのか（位置づけ）

Spring Boot でも、DB 接続の土台として JDBC が使われています。
ただし、実務では JDBC を直接書くことは少なく、次のような仕組みを利用することが多いです。

- JdbcTemplate（Spring の JDBC ラッパー）
- MyBatis（SQL を書きつつ、Java への変換を支援）
- JPA（ORM）

JDBC は「DB 接続の基礎技術」であり、上位の仕組み（MyBatis など）も内部では JDBC を利用しています。

- JDBC は、Java からデータベースに接続し、SQL を実行するための標準的な仕組みである
- JDBC はデータベースごとの JDBC ドライバを通して通信を行う
- 基本の流れは、接続（Connection）→ SQL 準備（PreparedStatement）→ 実行 → 結果取得（ResultSet）→ 終了処理である
- Spring Boot では JDBC を土台として、JdbcTemplate や MyBatis などの仕組みが利用されることが多い

DataSource

→ DB 接続情報を管理する設定オブジェクト。

DataSource（データソース）とは

DataSource（データソース）とは、データベースへ接続するための情報（URL・ユーザー名・パスワードなど）と、接続（Connection）を取得する仕組みをまとめて管理するオブジェクトです。
Spring や一般的な Java の DB 開発では、JDBC の接続処理を直接書く代わりに、DataSource を通して接続を扱うのが基本になります。

1. DataSource が必要になる理由

JDBC を直接使う場合、次のように毎回接続情報を指定して接続します。

```
Connection con = DriverManager.getConnection(url, user, pass);
```

この方法は動きますが、次の問題が起こりやすくなります。

- 接続情報があちこちに散らばる
- 設定変更（DB サーバー変更など）が大変になる
- 接続を毎回作ると遅くなる（負荷が高い）
- 接続の管理（再利用・後片付け）が難しい

そこで DataSource を使うと、接続情報と接続の管理を一か所に集約でき、保守性と性能が向上します。

2. DataSource が行う主な役割

DataSource は主に次の役割を持ちます。

① DB 接続情報を保持する
- 接続先 URL
- ユーザー名
- パスワード
- ドライバ設定 など

② Connection を提供する（必要なときに渡す）

```
Connection con = dataSource.getConnection();
```

3. Spring Boot では DataSource をどう扱うのか

Spring Boot では、設定ファイル（application.yml / application.properties）に DB 接続情報を書くと、
DataSource が自動で作成（Bean 登録） されます。

例（application.yml）：

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/sample
    username: user
    password: pass
```

4. DataSource と JDBC の関係（位置づけ）

この設定があると、Spring Boot が起動時に

- DataSource を作る
- Spring のコンテナに Bean として登録する

という処理を自動で行います。
そのため開発者は、DataSource を自分で new する必要がほとんどありません。

5. MyBatis / JdbcTemplate との関係

Spring Boot + MyBatis では、MyBatis は内部で DataSource を利用して DB 接続を行います。
- 開発者：SQL や Mapper を書く
- MyBatis：DataSource から Connection をもらって SQL を実行する

同様に JdbcTemplate も DataSource を利用します。

MyBatis / Flyway

MyBatis

→ SQL を直接書いて DB 操作を行うためのフレームワーク。

MyBatis とは

MyBatis（マイバティス）とは、SQL を自分で直接書き、その SQL を使ってデータベース操作
（検索・登録・更新・削除）を行うためのフレームワークです。
Java のコードと SQL をうまく分担し、SQL の実行結果を Java のオブジェクトへ変換する作業を支援してくれます。

1. MyBatis を使う目的（なぜ使うのか）

JDBC をそのまま書くと、次のようなコードが多くなります。
- Connection の取得
- PreparedStatement の作成
- ResultSet から値を取り出す
- 例外処理や後片付け（close）

このような「定型作業」が増えると、コードが長くなり、保守が大変になります。
MyBatis を使うと、これらの定型作業を減らしつつ、SQL は自分で書けるという特徴があります。
そのため、「SQL をしっかりコントロールしたい」場合に向いています。

2. MyBatis の特徴

① SQL をそのまま書ける
SQL の細かいチューニングや複雑な結合（JOIN）を、SQL として管理できます。

② SQL と Java の役割分担が明確
SQL：XML（またはアノテーション）側に書く
Java：Mapper（インターフェース）として呼び出す

③ SQL の結果を自動で Java に詰め替える
ResultSet から値を取り出してオブジェクトへ詰める作業を、MyBatis が支援します。

3. MyBatis の基本構造（登場人物）

MyBatis では、主に次の 3 つが出てきます。

1. Mapper（マッパー）
Java 側のインターフェース
「この SQL を実行する」という入り口になる

2. Mapper XML（SQL 定義ファイル）
実行する SQL を書く場所（select / insert / update / delete）

3. Entity / DTO（結果を受け取るクラス）
DB から取得したデータを入れる Java オブジェクト

4. 例：検索（SELECT）の流れ

（A）Mapper インターフェース（Java）

```
@Mapper
public interface UserMapper {
   User findById(int id);
}
```

（B）Mapper XML（SQL）

```
<select id="findById" resultType="User">
 SELECT id, name
 FROM users
 WHERE id = #{id}
</select>
```

（C）呼び出し側（Service など）

```
User user = userMapper.findById(10);
```

このとき MyBatis は内部で次のことを行います。

- #{id} に値を安全に埋め込む（PreparedStatement 相当）
- SQL を実行する
- 結果を User オブジェクトに詰め替える

5. #{} と ${} の違い（重要）

MyBatis では SQL に値を埋め込む書き方が 2 種類あります。

#{}（基本はこちら）

- PreparedStatement の仕組みで安全に値を渡します
- SQL インジェクション対策として推奨されます

${}（原則注意）

- 文字列として SQL に直接埋め込まれます
- 使い方を誤ると SQL インジェクションの原因になります
- ORDER BY のカラム名切替など、限定用途で使います

授業では **「基本は #{}」** と覚えてもらうのが安全です。

6. Spring Boot で MyBatis を使うときの位置づけ

Spring Boot と組み合わせると、次の構成が一般的です。

- Controller　：リクエスト受付
- Service　：業務処理
- Mapper（MyBatis）　：DB 操作
- DataSource　：DB 接続情報
- MyBatis　：DataSource から Connection を取得して SQL 実行

MyBatis は「DB と SQL 実行担当」で、内部では JDBC を利用しています。

Mapper

→ SQL と Java メソッドを結びつける仕組み。

Mapper（マッパー）とは

Mapper（マッパー）とは、MyBatis において **「SQL と Java のメソッドを結び付けるための仕組み」** です。
開発者は Java 側では通常のメソッド呼び出しを行い、MyBatis が内部で対応する SQL を実行して結果を返します。

> 「Mapper は " 自分で書いたクラス " ではなく、
> MyBatis があとから作ってくれるクラスを " 呼び出すための設計図 " です。

1. Mapper の役割（何をしているのか）

Mapper は、次の 3 つをつなぐ「橋渡し役」です。

- Java のメソッド呼び出し（例：findById(10)）
- 実行する SQL（例：SELECT ... WHERE id = ?）
- SQL の実行結果を Java の型に変換（例：User オブジェクトに詰め替え）

そのため Mapper を使うと、開発者は JDBC の細かな処理（Connection、PreparedStatement、ResultSet など）を意識せずに、SQL を実行できます。

2. Mapper は通常「インターフェース」として作成します

MyBatis では、Mapper を次のような インターフェースとして書くのが一般的です。

```
@Mapper
public interface UserMapper {
  User findById(int id);
}
```

ここで重要なのは、Mapper インターフェースのメソッドには 実装（中身）がない点です。
実装がないのに呼び出せるのは、MyBatis が実行時に「代理のクラス（プロキシ）」を自動生成し、SQL 実行処理を肩代わりするためです。

> Mapper インターフェースに実装がなくても呼び出せる理由

MyBatis では、Mapper は通常 インターフェースとして定義します。
このインターフェースには、メソッドの宣言だけが書かれており、処理の中身（実装）は記述しません。

```
@Mapper
public interface UserMapper {
  User findById(int id);
}
```

一見すると、

- メソッドの中身がない
- クラスでもない
- new していない

にもかかわらず、findById() を呼び出せるため、不思議に感じられます。

1. 実装は「開発者が書かない」だけで、存在しないわけではない

重要な点は、実装が存在しないのではなく、開発者が書いていないだけという点です。

MyBatis では、アプリケーションの起動時に次の処理が行われます。
① Mapper インターフェースを読み取る
②対応する Mapper XML（SQL 定義）を探す
③それらをもとに、実際に動作するクラスを MyBatis が自動で生成する

この自動生成されたクラスが、SQL の実行処理を担当します。

2. MyBatis が自動生成する「代理クラス」の役割

MyBatis が内部で生成するクラスは、Mapper インターフェースの代わりに実際の処理を行うクラスです。
このように「本来の処理を代行するクラス」を、一般に 代理（プロキシ）クラスと呼びます。

概念的には、次のようなクラスが内部で作られていると考えることができます。

```
class UserMapperProxy implements UserMapper {

  @Override
  public User findById(int id) {
    // Mapper XML から対応する SQL を取得
    // JDBC を使って SQL を実行
    // 実行結果を User オブジェクトに変換
    return user;
  }
}
```

※このクラスは MyBatis が自動生成するものであり、開発者が直接書くことはありません。

3. 開発者から見た Mapper の使い方

開発者は、この内部の仕組みを意識する必要はありません。
Service クラスなどからは、次のように 通常のメソッド呼び出しとして Mapper を利用できます。

```
User user = userMapper.findById(10);
```

このとき実際に呼び出されているのは、
MyBatis が生成した 代理クラスのメソッドです。

4. なぜ Mapper をインターフェースにするのか

Mapper をインターフェースとして定義する理由は、役割を明確に分けるためです。

- Java（Mapper インターフェース）
  →「どのような操作ができるか」を定義する
- Mapper XML
  →「どの SQL を実行するか」を定義する
- MyBatis
  → 両者を結び付け、実行処理を担当する

この分業により、Java のコードと SQL をきれいに分離でき、保守性が高まります。

- Mapper は通常、インターフェースとして定義され、メソッドの宣言のみを持つ
- メソッドの実装は、開発者ではなく MyBatis が実行時に自動生成する
- MyBatis が生成する代理クラスが、SQL の実行と結果の変換を担当する
- 開発者は、実装の存在を意識せず、通常のメソッド呼び出しとして Mapper を利用できる

3. SQL は Mapper XML（またはアノテーション）に書きます

（A）Mapper XML で SQL を書く例

```
<select id="findById" resultType="User">
 SELECT id, name
 FROM users
 WHERE id = #{id}
</select>
```

- id="findById" が Java のメソッド名に対応します
- #{id} は メソッドの引数を受け取る場所です（安全に値が渡されます）

以下のように結びつく仕組みです

Java：findById(int id)
XML：<select id="findById"> ... #{id} ... </select>

が結び付く仕組みです。

4. Mapper を呼び出す側から見ると「普通のメソッド」に見える

Service などで Mapper を呼ぶときは、次のように書けます。

```
User user = userMapper.findById(10);
```

しかし内部では MyBatis が次を自動で行っています。

- SQL を組み立てる（#{} に値を入れる）
- PreparedStatement を使って実行する
- 結果を User オブジェクトに詰め替える

開発者が JDBC の定型作業を書かずに済むのが、Mapper の利点です。

5.「SQL と Java メソッドが結び付く」条件（重要）

Mapper が正しく動くためには、次の対応関係が成立している必要があります。

① メソッド名と SQL の id が一致している
- Java：findById
- XML：<select id="findById">

② 引数名（または @Param）と #{} の名前が一致している
- 引数が 1 つの場合は、MyBatis が扱いやすく、多くの場合そのまま書けます。
- 引数が複数になる場合は、@Param を使って名前を明示するのが一般的です。

```
User findByNameAndStatus(@Param("name") String name, @Param("status") String status);
```

XML 側：

```
WHERE name = #{name} AND status = #{status}
```

Mapper XML

→ SQL を定義する XML ファイル。

Mapper XML とは

Mapper XML（マッパー・エックスエムエル）とは、MyBatis で使用する **「SQL を定義するための XML ファイル」** です。
Java のコードから呼び出される SQL（SELECT / INSERT / UPDATE / DELETE）を、この XML にまとめて記述し、MyBatis が実行できる形で管理します。

1. なぜ Mapper XML を使うのか

Mapper XML を使う主な目的は、次の通りです。

- SQL を Java コードから分離して管理できる
  → Java の処理と SQL を混ぜないため、見通しが良くなります。
- SQL を直接書ける
  → 複雑な JOIN や集計、チューニングも SQL として表現できます。
- 条件分岐や繰り返しを使って SQL を動的に組み立てられる
  → 検索条件が可変の画面などで特に有効です。

2. Mapper XML の基本構造

Mapper XML では、最初に <mapper> タグを書き、どの Mapper インターフェースに対応するかを namespace で指定します。

```
<mapper namespace="com.example.mapper.UserMapper">
 <!-- SQL 定義（select / insert / update / delete）-->
</mapper>
```

namespace は通常、対応する Mapper インターフェースの ** 完全修飾名（パッケージ＋クラス名）** にします。これにより「この XML はこの Mapper 用である」と MyBatis が判断できます。

3. SQL を定義するタグ（代表例）

Mapper XML では、SQL の種類ごとに次のタグを使います。

- <select>：検索（SELECT）
- <insert>：登録（INSERT）
- <update>：更新（UPDATE）
- <delete>：削除（DELETE）

4. 例：SELECT（検索）を定義する

```
<mapper namespace="com.example.mapper.UserMapper">

 <select id="findById" resultType="com.example.domain.User">
  SELECT id, name
  FROM users
  WHERE id = #{id}
 </select>

</mapper>
```

ここで重要な点は次の通りです。

- id="findById"
  → Java の Mapper インターフェースにある メソッド名と対応します。
- resultType="..."
  → SQL の結果を詰め替える 戻り値の型を指定します。
- #{id}
  → Java 側から渡された引数（id）を安全に SQL に渡します（PreparedStatement 相当）。

5. Java 側との対応関係

Java 側（Mapper インターフェース）：

```
@Mapper
public interface UserMapper {
  User findById(int id);
}
```

- メソッド名 findById と、XML の <select id="findById"> が対応します。
- 引数 id と、SQL 内の #{id} が対応します。

この対応が取れていると、Java 側からは次のように「普通のメソッド」として呼び出せます。

```
User user = userMapper.findById(10);
```

6. 条件分岐・動的 SQL（概要）

Mapper XML では、検索条件の有無に応じて SQL を変えることもできます。

- <if>：条件が真のときだけ SQL 断片を追加
- <where>：WHERE 句を賢く組み立てる
- <foreach>：IN 句などで繰り返し展開する

これにより、検索画面の「条件あり・条件なし」を 1 つの SQL 定義で扱いやすくなります。

7. Mapper XML を置く場所（Spring Boot の一般例）

プロジェクトによって異なりますが、Spring Boot では次のように配置することが多いです。

- src/main/resources/mapper/ 配下
  例：src/main/resources/mapper/UserMapper.xml

そして application.yml などで、MyBatis に XML の場所を伝えます（設定方法はプロジェクトにより異なります）。

- Mapper XML は、MyBatis で SQL を定義する XML ファイルである
- Java の Mapper インターフェースと対応付けて使用し、SQL を Java コードから分離できる
- <select> / <insert> / <update> / <delete> により SQL を定義し、id はメソッド名と対応する
- #{} を使うことで、メソッド引数の値を安全に SQL に渡すことができる
- 条件分岐や繰り返しにより、動的な SQL も記述できる

Flyway

→ データベース構造をバージョン管理するツール。

Flyway（フライウェイ）とは、データベースの構造（テーブルやカラムなど）をバージョン管理し、変更を順番どおりに適用するためのツールです。
一般に「DB マイグレーションツール」と呼ばれ、開発環境・テスト環境・本番環境のデータベースを 同じ手順で同じ状態にそろえる目的で利用されます。

1. なぜ DB のバージョン管理が必要なのか

アプリ開発では、機能追加や修正に伴って次のような変更が発生します。

- 新しいテーブルを作る
- 既存テーブルにカラムを追加する
- インデックスを追加する
- 制約を変更する など

これらを人が手作業で行うと、次の問題が起きやすくなります。

- 適用漏れ（変更を忘れる）
- 手順ミス（順番を間違える）
- 環境ごとの差異（開発は動くが本番は動かない）
- 誰がいつ何を変更したか追えない

Flyway を使うと、これらの問題を避けやすくなります。

2. Flyway の基本的な考え方

Flyway は、DB 変更を **SQL ファイル（マイグレーションファイル）** として保存し、
それを バージョン順に適用していきます。

例：次のようなファイル名を用意します。

```
V1__create_users_table.sql
V2__add_email_column.sql
V3__create_orders_table.sql
```

ポイントは以下の通りです。

- V1, V2, V3 が バージョン番号です
- __（アンダースコア 2 つ）の後ろは 説明文です（読みやすさのため）

3. Flyway が行う処理の流れ

Flyway は起動時（またはコマンド実行時）に、概ね次を行います。

1. マイグレーションファイル（SQL）を指定フォルダから探す
2. すでに適用済みかどうかを確認する
3. 未適用のものを バージョン順に実行する
4. 「どのバージョンまで適用したか」を DB 内に記録する

この「適用履歴の記録」を行うために、Flyway は DB 内に専用テーブルを作ります。
一般的に flyway_schema_history という名前で作成されます。

4. Flyway を使うメリット

① DB 変更の履歴が残る
「いつ」「何を」「どの順で」変えたかがファイルとして残るため、追跡できます。

② 環境差分が起きにくい
開発・テスト・本番で同じ SQL を同じ順に適用できるため、DB 構造が揃います。

③ 変更手順が自動化できる
人の手作業が減り、ミスの発生確率が下がります。

④ チーム開発に強い
「誰かが勝手に DB をいじっていた」という状態を防ぎやすくなります。

5. Spring Boot での位置づけ（概要）

Spring Boot では Flyway を導入すると、一般的に

1. アプリ起動時に Flyway が動作し
2. 未適用のマイグレーション SQL を自動で適用する

という運用ができます。
そのため、アプリを起動するだけで DB 構造を最新状態に整えられるようになります。

> Flyway は、データベース構造の変更をバージョン管理するためのツールである
> SQL ファイルとして変更内容を保存し、バージョン順に適用する
> 適用履歴を DB 内の専用テーブルに記録し、未適用分のみを実行する
> 環境差分の防止、作業ミスの削減、履歴管理の明確化に役立つ

マイグレーション

→ DB の構造変更を履歴として管理する仕組み。

マイグレーションとは

マイグレーションとは、データベース（DB）の構造変更（テーブル作成・カラム追加など）を、履歴として記録し、順番どおりに適用できるように管理する仕組みです。
ここでいう「DB の構造」とは、主に次のようなものを指します。

- テーブル（作成・削除）
- カラム（追加・変更・削除）

- 制約（NOT NULL、外部キーなど）
- インデックス（追加・削除）
- ビューやシーケンス など

1. なぜマイグレーションが必要なのか

アプリケーション開発では、機能追加や仕様変更に伴って DB の構造も変化します。
たとえば次のような変更が頻繁に発生します。

- 「メールアドレス」カラムを追加する
- 新しいテーブルを作る
- 既存カラムの型や制約を変更する

これを人が手作業で行うと、次の問題が起こりやすくなります。

- 実行忘れ（変更を適用し忘れる）
- 手順ミス（順番を間違える、SQL を間違える）
- 環境差分（開発では動くのに本番では動かない）
- 履歴不明（誰がいつ何を変えたか分からない）

マイグレーションは、これらの問題を防ぐために導入されます。

2. マイグレーションで行うこと（基本の考え方）

マイグレーションでは、DB の変更を「ファイル（またはコード）」として残し、
その変更を 順番どおりに適用します。

たとえば、次のように段階的に DB を進化させていきます。

- 1 回目：ユーザーテーブルを作る
- 2 回目：メールアドレスカラムを追加する
- 3 回目：注文テーブルを追加する

このように「DB の成長の履歴」を残すことで、いつでも同じ手順で同じ状態を再現できます。

3.「DB のバージョン管理」としてのマイグレーション

マイグレーションは、言い換えると DB のバージョン管理です。

- アプリのソースコードは Git で履歴管理する
- DB の構造もマイグレーションで履歴管理する

これにより、チーム開発でも「同じ DB 構造」を維持しやすくなります。

4. マイグレーションの一般的な運用

マイグレーションは、次のような運用で使われることが多いです。

1. 開発者が DB 変更の SQL（または定義）を作る
2. 変更ファイルをリポジトリに追加する（履歴として残す）
3. 開発・テスト・本番で、そのファイルを 同じ順番で適用する
4. 適用済みかどうかを記録し、二重適用を防ぐ

Flyway のようなツールを使うと、この運用を自動化できます。

- マイグレーションとは、DB の構造変更を履歴として管理し、順番どおりに適用する仕組みである
- テーブルやカラム、制約、インデックスなどの変更を対象とする
- 手作業による変更はミスや環境差分を生みやすいため、履歴管理が重要となる
- マイグレーションにより、開発・テスト・本番の DB 構造を同じ状態にそろえやすくなる

設定・起動まわり

pom.xml

→ Maven の設定ファイル。依存関係を管理する。

pom.xml とは

pom.xml は、Maven（メイヴン）で使用する 設定ファイルです。
正式には Project Object Model（POM）と呼ばれ、プロジェクトの情報や、ビルド方法、使用するライブラリ（依存関係）などをまとめて管理します。

Spring Boot のプロジェクトでは、この pom.xml が「プロジェクトの設計図」のような役割を持ちます。

1. Maven とは何をするツールか（前提）

Maven は、主に次の作業を自動化するためのツールです。

- ライブラリ（依存関係）のダウンロードと管理
- コンパイル（Java を class にする）
- テストの実行
- パッケージ作成（jar / war を作る）
- ビルド手順の統一

そして、Maven は pom.xml の内容にもとづいてこれらを実行します。

2. pom.xml に書く主な内容

pom.xml には、主に次の情報を記述します。

① プロジェクト情報
プロジェクトを識別するための情報です。

- groupId：所属（会社名や組織名など）
- artifactId：プロジェクト名
- version：バージョン

例：

```
<groupId>jp.co.example</groupId>
<artifactId>sample-app</artifactId>
<version>1.0.0</version>
```

② 依存関係（dependencies）

プロジェクトで使うライブラリを指定します。
Maven はここに書かれたライブラリをインターネット（Maven Central など）から自動で取得します。

例：

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
```

この例では、Web アプリに必要な Spring のライブラリ一式が導入されます。

③ ビルド設定（build / plugins）
ビルド時の動作を指定します。
たとえば、Spring Boot では jar を起動可能な形式にまとめるためにプラグインを利用します。

④ Java のバージョンなどの設定（properties）
プロジェクトで利用する Java のバージョンなどを指定できます。

3.「依存関係を管理する」とはどういう意味か

依存関係とは、アプリが動くために必要な 外部ライブラリのことです。
たとえば Spring Boot で Web アプリを作るには、Servlet や JSON 変換など多数のライブラリが必要になります。

依存関係を pom.xml に書いておくと、

- 必要なライブラリを自動で集めてくれる
- バージョンの整合性を取りやすい
- チーム全員が同じ環境で開発できる

という利点があります。

4. Spring Boot では pom.xml が特に重要になる理由

Spring Boot では「starter」という仕組みにより、必要なライブラリ一式をまとめて導入できます。

例：spring-boot-starter-web を入れると

- Spring MVC
- 内蔵 Tomcat
- JSON 変換機能
- ログ関連

などがまとめて導入されます。
これにより、pom.xml を少ない記述で済ませやすくなります。

> pom.xml は、Maven の設定ファイルであり、プロジェクトの設計図の役割を持つ
> プロジェクト情報、依存関係（ライブラリ）、ビルド方法などを記述する
> 依存関係を pom.xml で管理すると、必要なライブラリを自動取得でき、環境差分を防ぎやすい
> Spring Boot では starter 依存関係により、少ない記述で必要な機能一式を導入できる

dependency

→ プロジェクトで使用する外部ライブラリ。

dependency（依存関係）とは

dependency（ディペンデンシー）とは、** プロジェクトが動作するために必要な外部ライブラリ（部品）** のことです。
Java のプログラムは、自分で書いたコードだけで完結することは少なく、多くの場合は外部の便利な機能（ライブラリ）を組み合わせて作ります。
その「使う外部ライブラリ」を Maven では pom.xml の <dependencies> に記述し、管理します。

1. 依存関係が必要な理由

たとえば、Web アプリを作るには次のような機能が必要になります。

- HTTP リクエストの処理
- JSON の変換
- サーバー（Tomcat など）の動作
- ログ出力
- DB 接続 など

これらをすべて自分で一から作るのは現実的ではありません。

そこで、すでに用意されている高品質なライブラリを利用します。
つまり dependency とは、
「自分のプロジェクトが頼って使う、外部の部品」
という意味です。

2. Maven における dependency の書き方

Maven では pom.xml に次のように記述します。

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
```

このように書くと、Maven が自動的にライブラリをダウンロードし、プロジェクトで使えるように準備します。

3. dependency の各要素の意味

① groupId
ライブラリの提供元（組織・グループ）を表します。

② artifactId
ライブラリ名（製品名）を表します。

③ version（必要な場合）
利用するバージョンを指定します。

Spring Boot の場合は、親設定（starter parent）によりバージョンが管理されるため、依存関係側で version を省略できることが多いです。

4. dependency を追加すると何が起きるのか

dependency を追加すると、次のことが起こります。

- Maven が必要な jar ファイルを自動取得する
- そのライブラリが内部で必要とする別ライブラリも一緒に取得する（依存関係の連鎖）
- Java のコードから import して使えるようになる

たとえば spring-boot-starter-web を追加すると、Web 開発に必要な複数のライブラリがまとめて導入されます。

> dependency（依存関係）とは、プロジェクトが利用する外部ライブラリである
> Maven では pom.xml に依存関係を記述し、必要なライブラリを自動取得して利用できる
> dependency には groupId / artifactId / version などを指定し、ライブラリを一意に特定する
> 依存関係を追加すると、そのライブラリが必要とする関連ライブラリも合わせて導入されることが多い

Maven

→ ビルド・依存関係管理ツール。

Maven（メイヴン）とは

Maven（メイヴン）とは、Java 開発において ** ビルド（コンパイルやテスト、成果物作成）** と、
依存関係（外部ライブラリ）の管理を自動化するためのツールです。
プロジェクトの設定は pom.xml にまとめて記述し、Maven はその内容に従って必要な処理を実行します。

1. Maven が担当する主な役割

> ① ビルド（Build）

ビルドとは、アプリケーションを動かせる形に整える作業のことです。
Maven は次の作業をまとめて実行できます。

- Java ソースコードの コンパイル（.java → .class）
- テストの実行（JUnit など）
- 実行可能な形式への パッケージ作成（jar / war）
- 不要ファイルの削除（クリーン）

これらを手作業で行うと手順が複雑になりますが、Maven ではコマンド一つで実行できます。

> ② 依存関係管理（Dependency Management）

依存関係とは、プロジェクトが利用する外部ライブラリのことです。
Maven は pom.xml に書かれた依存関係をもとに、必要なライブラリを自動的にダウンロードし、プロジェクトで利用できるようにします。

- どのライブラリが必要か
- どのバージョンを使うか
- そのライブラリがさらに必要とするライブラリ（依存の連鎖）

これらを Maven がまとめて管理します。

2. Maven を使うと何が良くなるのか

① 手順が統一されます

チーム開発では、各自が勝手な手順でビルドすると、環境差分やトラブルの原因になります。
Maven を使うと「pom.xml に従う」というルールで手順が統一されます。

② 環境差分が起きにくくなります

依存関係を pom.xml に書いておけば、別のパソコンでも同じライブラリを自動で取得できるため、開発環境をそろえやすくなります。

③ 追加・変更が管理しやすくなります

ライブラリを追加したい場合は pom.xml に追記すればよく、ファイルを手作業でコピーする必要がありません。

3. Maven の基本コマンド（代表例）

mvn compile	:	コンパイルする
mvn test	:	テストを実行する
mvn package	:	jar / war を作成する
mvn clean	:	生成物を削除する
mvn clean package	:	削除してから jar / war を作る

Spring Boot では、次のように実行することもあります。

mvn spring-boot:run	:	アプリケーションを起動する

4. Maven と pom.xml の関係

Maven の動作は pom.xml によって決まります。
pom.xml には、主に次の情報が書かれます。

- プロジェクト情報（groupId / artifactId / version）
- 依存関係（dependencies）
- ビルド設定（plugins）
- Java バージョンなどの設定（properties）

Maven は、pom.xml を読み取り、プロジェクトを正しい手順で組み立てます。

Maven は、Java 開発におけるビルドと依存関係管理を行うツールである
記述し、その内容に従ってコンパイル、テスト、パッケージ作成などを自動化する
依存関係を定義することで、必要な外部ライブラリを自動的に取得し、環境差分を防ぎやすくなる
チーム開発において手順を統一し、保守性を高めることができる

application.properties / yml

→ Spring Boot の設定ファイル。

application.properties / application.yml とは

application.properties と application.yml は、Spring Boot における 設定ファイルです。
アプリケーションの動作に関する設定（例：サーバーのポート番号、データベース接続情報、ログ設定など）を、ソースコードとは別の場所にまとめて記述できます。

両者は役割が同じで、書き方（形式）が異なるだけです。

- application.properties：キー = 値 の形式で書きます
- application.yml：階層構造（インデント）で書きます

1. どこに置くのか（基本）

通常は次の場所に配置します。

- src/main/resources/application.properties
- /application.yml

Spring Boot は起動時にこれらを読み込み、設定内容を自動的に反映します。

2. 何を書くのか（代表例）

> ① サーバー設定（ポート番号）

Web アプリの待ち受けポートを変更できます。

properties

```
server.port=8081
```

yml

```
server:
  port: 8081
```

> ② データベース接続設定（DataSource）

DB の接続先や認証情報を設定します。

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/sample
spring.datasource.username=user
spring.datasource.password=pass
```

yml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/sample
    username: user
    password: pass
```

> ③ ログ設定（例：ログレベル）

開発中にログを詳しくしたい場合などに利用します。

properties

```
logging.level.org.springframework=INFO
```

yml

```
logging:
  level:
    org.springframework: INFO
```

3. なぜ設定ファイルが重要なのか

設定ファイルを使うことで、次のメリットがあります。

> ① ソースコードを変更せずに動作を変えられる

たとえば「DB の接続先を変更する」「ポート番号を変える」といった調整を、コードを書き換えずに行えます。

> ② 環境ごとに設定を分けやすい

開発環境と本番環境では、DB やログ設定が異なることが一般的です。
設定ファイルを使うと、環境に応じた切り替えが容易になります。

4. properties と yml の違い（使い分け）

どちらを使っても動作は同じです。違いは「書きやすさ」です。

application.properties の特徴

- 1 行ずつ キー = 値 で書ける
- 小規模では分かりやすい
- 階層が深くなると行が長くなりがちです

application.yml の特徴

- 階層構造で書けるため見通しが良い
- 設定項目が多いプロジェクトで整理しやすい
- インデント（空白）が重要で、間違えると読み取りに失敗します

授業では、どちらか一方に統一すると混乱が減ります（プロジェクトで統一することが大切です）。

5. 設定値は Spring Boot の機能に自動で反映される

Spring Boot の特徴として、設定ファイルに書いた値は、次のように自動的に反映されます。

- DataSource の自動作成
- サーバー起動設定
- ログ出力設定
- MyBatis や Flyway の設定（導入していれば）

設定ファイルは、アプリの動作そのものを決める重要な部品です。

> application.properties と application.yml は、Spring Boot の設定ファイルである
> サーバー設定、DB 接続設定、ログ設定など、アプリの動作に関する設定を記述できる
> ソースコードを変更せずに設定を変更できるため、環境ごとの切り替えに適している
> properties は キー = 値 形式、yml は階層構造で記述し、役割は同じである

classpath

→ Java がクラスやリソースを探す場所。

classpath（クラスパス）とは

classpath（クラスパス）とは、**Java がプログラム実行時に「クラス（.class）やリソースファイル」を探しに行く場所（検索範囲）** のことです。
Java は、必要なクラスや設定ファイルを自動的に見つけて読み込む必要があります。そのときに参照する「探す場所の一覧」が classpath です。

1. なぜ classpath が必要なのか

Java のプログラムは、1 つのファイルだけで動くことは少なく、次のように多くの部品を使って動きます。

- 自分で作成したクラス（Controller、Service など）
- 外部ライブラリのクラス（Spring、MyBatis など）
- 設定ファイルやテンプレート（application.yml、HTML など）

Java はこれらを必要に応じて読み込みますが、どこにあるかを探す範囲が決まっていないと見つけられません。
その検索範囲を指定するのが classpath です。

2. classpath に含まれるもの

classpath には主に次が含まれます。

① コンパイルされたクラス（.class）

通常、次のような場所が classpath に入ります。

- Maven：target/classes
- Gradle：build/classes

ここに Controller や Service などの class ファイルが置かれ、Java はそれを読み込みます。

② 依存ライブラリ（jar）

Maven の dependency として追加したライブラリ（Spring など）は jar ファイルとして classpath に追加されます。
その結果、import して利用できるようになります。

③ リソースファイル

src/main/resources に置いたファイル（例：設定ファイル、テンプレート、画像など）も、ビルド後に classpath 上へ配置されます。

例：

- application.properties / application.yml
- templates/ 配下の Thymeleaf テンプレート
- mapper/ 配下の MyBatis Mapper XML

3.「import」と「classpath」の関係（混同注意）

授業で混乱しやすい点として、次を区別する必要があります。

- classpath：Java が探しに行ける範囲（実行環境の設定）
- import：ソースコード上で「このクラス名を使います」と宣言する記述

つまり、import を書いても classpath に存在しなければクラスは見つかりませんし、
逆に classpath に存在しても import を書かなければ（または完全修飾名を書かなければ）使えません。

4. Spring Boot でよく出る classpath の例

① application.yml が見つかる理由

src/main/resources/application.yml は、ビルド後に classpath に含まれる場所へコピーされます。
そのため Spring Boot は起動時に自動で読み取れます。

② Thymeleaf テンプレートが見つかる理由

src/main/resources/templates/ の HTML も classpath に配置されるため、return "hello" のような指定で見つけられます。

③ MyBatis の Mapper XML が読み込める理由

Mapper XML も resources 配下に置けば classpath 上に配置され、MyBatis が参照できるようになります。

5. classpath の典型的なエラー

classpath が正しくないと、次のようなエラーが起きます。

- ClassNotFoundException（クラスが見つからない）
- NoClassDefFoundError（実行時に必要なクラスがない）
- 設定ファイルが読めない（application.yml が見つからない）
- テンプレートが見つからない（Thymeleaf の HTML が見つからない）

これらは「コードが間違っている」というより、Java が探す場所（classpath）に必要なものが無いことが原因のケースが多いです。

- classpath とは、Java がクラスやリソースファイルを探す場所（検索範囲）のことである
- classpath には、自作クラス（.class）、依存ライブラリ（jar）、resources 配下の設定ファイルなどが含まれる
- Spring Boot では application.yml やテンプレート、Mapper XML なども classpath 上から読み込まれる
- classpath が正しくないと、クラスやファイルが見つからないエラーが発生する