IT資格道場

Java Silver SE 25(1Z0-827-JPN) 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

Oracle公式 Java SE 25 Programmer I(1Z0-827-JPN) 試験内容チェックリスト 準拠
2026年11月版

SAMPLE

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

SAMPLE

目次

目次

第1章 Java の概要と簡単なJavaプログラムの作成 [1.1/1.2/1.3]

main メソッドを持つ実行可能なJavaプログラムの作成 [1.1]

コンパクト・ソースファイルおよびインスタンス main メソッドを持つ Java プログラムの作成、およびコマンドラインでのJavaプログラムのコンパイルと実行 [1.2]

パッケージの使用 (パッケージ宣言とインポート) [1.3]

サンプルはここまでです

SAMPLE

3 冊の使い方は、ダウンロード欄の案内を参照してください。

Java SE 25 Programmer I（1Z0-827-JPN）の全6主題・33細目を、出題範囲の順に並べた学習用テキストです。各章は、規則が動く仕組みの説明と、試験で問われる形のコードで組み立てています。載せているコードと出力は、OpenJDK 25.0.5 の `javac --release 25` と `java` で実際に動かして確かめました。

SAMPLE

目次

- 第1章 Java の概要と簡単なJavaプログラムの作成 [1.1/1.2/1.3]
  - main メソッドを持つ実行可能なJavaプログラムの作成 [1.1]
  - コンパクト・ソースファイルおよびインスタンス main メソッドを持つ Java プログラムの作成、およびコマンドラインでのJavaプログラムのコンパイルと実行 [1.2]
  - パッケージの使用 (パッケージ宣言とインポート) [1.3]
- 第2章 Javaの基本データ型と文字列の操作 [2.1/2.2/2.3/2.4/2.5]
  - 変数の宣言および初期化と変数のスコープ [2.1]
  - ローカル変数の型推論 (var) および無名変数の使用 [2.2]
  - 文字列の作成と操作 (テキスト・ブロックおよび StringBuilder による可変文字列の操作を含む) [2.3]
  - 配列(一次元配列、二次元配列)の宣言、インスタンス化、初期化および使用 [2.4]
  - ArrayList の使用 [2.5]
- 第3章 演算子と制御構造 [3.1/3.2/3.3/3.4]
  - 算術式と boolean 式の評価、演算子の優先順位、Math API [3.1]
  - 分岐文：if、if/else、switch [3.2]
  - 繰り返し文：while、do-while、for、拡張 for [3.3]
  - switch 式・型パターン・レコード・パターン・シール型 [3.4]
- 第4章 クラスの定義とインスタンスの使用 [4.1/4.2/4.3/4.4/4.5/4.6/4.7/4.8]

SAMPLE

- Javaクラスの定義とインスタンス化とオブジェクトのライフサイクル [4.1]
- メソッドとコンストラクタ (柔軟なコンストラクタ本体を含む) の作成 [4.2]
- メソッドのオーバーロードおよび可変長引数 (varargs) の使用 [4.3]
- static変数とstaticメソッドおよび初期化子の使用 [4.4]
- アクセス修飾子の使用 [4.5]
- instanceof のパターン・マッチング [4.6]
- レコード・クラスの作成と使用 [4.7]
- enum 型の使用と作成 (フィールド、メソッドおよびコンストラクタを含む) [4.8]

- 第5章 継承とインタフェースの使用 [5.1/5.2/5.3/5.4/5.5/5.6/5.7/5.8]
  - スーパークラスとサブクラスの作成と使用 [5.1]
  - 抽象クラスの作成と使用 [5.2]
  - メソッドのオーバーライド [5.3]
  - 参照型のキャストとポリモーフィックなメソッド呼び出し [5.4]
  - final クラスの作成と使用 [5.5]
  - インタフェースの作成と実装 [5.6]
  - デフォルト・メソッドとプライベート・メソッドの使用 [5.7]
  - シール・クラスの作成と使用 [5.8]

- 第6章 例外処理 [6.1/6.2/6.3/6.4/6.5]
  - 例外処理の仕組みとチェック例外、非チェック例外、エラーの違い [6.1]

SAMPLE

- try-catch文による例外処理 [6.2]
- try-with-resources 文による例外処理 [6.3]
- カスタム例外の作成 [6.4]
- multi-catch 句の使用 [6.5]

SAMPLE

第1章 Java の概要と簡単なJavaプログラムの作成 [1.1/1.2/1.3]

Java のプログラムは、ソースファイル（`.java`）を `javac` でコンパイルして、クラスファイル（`.class`）にします。`java` コマンドが Java 仮想マシン（JVM）を起動し、指定されたクラスの `main` メソッドを呼んで実行を始めます。[1]

Java SE 25 では、この入口の決まりが大きく変わりました。`main` の形が広がり、クラス宣言の無いファイルも書けます。第1章は、この新しい決まりと、パッケージと import の読み方を問います。[2]

main メソッドを持つ実行可能なJavaプログラムの作成 [1.1]

main メソッドの形（public static void main(String[] args)・可変長 String... の形・修飾子の順）

起動の入口になれる `main` を、候補（candidate main method）と呼びます。候補は次の2形のどちらかです。[1]

- 引数が `String` の配列 1 つで、戻り値が `void`
- 引数が無く、戻り値が `void`

どちらの形でも、アクセス修飾子は `public`・`protected`・修飾子なし（パッケージアクセス）のどれかです。`private` は候補になりません。`static` は付けても付けなくてもよく、`throws` 節も付けられます。

Java SE 25 より前は、`public static void main(String[] args)` だけが入口でした。この形は今も候補の 1 つで、最もよく見る書き方です。

SAMPLE

引数の型は `String[]` でも可変長の `String...` でも構いません。どちらも同じ型を表します。引数名は `args` でなくても構いません。`final` も付けられます。`String args[]` の書き方も通ります。

修飾子の順は、戻り値の型 `void` より前に置けば自由です。`static public void main` は有効で、`public void static main` はコンパイルエラーになります。

起動できるかどうかは、コンパイルでは決まりません。次の表の宣言は、どれも `javac` を通ります。`java` で起動した時に、候補かどうかが決まります。

宣言	起動
`public static void main(String... args)`	できる
`static public void main(final String[] args)`	できる
`protected static void main(String[] args)`	できる
`static void main(String[] args)`	できる
`public void main(String[] args)`	できる（インスタンス main）
`void main()`	できる（インスタンス main）
`public static void main(String[] args) throws Exception`	できる
`private static void main(String[] args)`	できない
`public static int main(String[] args)`	できない（戻り値が void でない）

SAMPLE

宣言	起動
`public static void main(String args)`	できない（String の配列でない）
`public static void main(String[][] args)`	できない
`public static void main(Integer[] args)`	できない
`public static void main(String[] args, int x)`	できない（引数が 2 つ）

起動できない時の表示は次の形です。`main` は予約語ではないので、こうした宣言そのものはエラーになりません。[3]

> エラー: メイン・メソッドがクラスE2で見つかりません。次のようにメイン・メソッドを定義してください。
>
> `public static void main(String[] args)`
>
> またはJavaFXアプリケーション・クラスはjavafx.application.Applicationを拡張する必要があります

同じクラスに `main(String[])` と `main(String...)` を両方書くと、同じシグネチャなのでコンパイルエラーです。一方、候補が複数あること自体はエラーではありません。

起動できる main の選ばれ方（String[] 引数の main が優先・引数なしの main・継承した main）

クラスに候補が複数ある時、ランチャーは次の順で 1 つだけ選びます。[1]

SAMPLE

1. 引数が `String[]` の候補があれば、それを選ぶ。
2. なければ、引数なしの候補を選ぶ。
3. どちらも無ければ、エラーで終わる。

選ぶ時に、`static` かどうかは見ません。引数の有無だけで優先が決まります。`static void main()` と `void main(String[] args)` があれば、後者が選ばれます。

```
// M2.java
class M2 {
    static void main() { System.out.println("static no-arg"); }
    void main(String[] args) { System.out.println("instance with-
args"); }
}
```

```
instance with-args
```

組み合わせごとの結果を表にします。

クラスにある main	実行される main
`static main(String[])` と `static main()`	`static main(String[])`
インスタンス `main(String[])` とインスタンス `main()`	`main(String[])`
`private static main(String[])` と `static main()`	`static main()`（private は候補でない）

SAMPLE

クラスにある main	実行される main
`static main(String[])` と、同じ引数のインスタンス `main(String[])`	コンパイルエラー（同じシグネチャは宣言できない）

候補は、クラス自身が宣言していなくても、継承した物で構いません。次のクラス `M6` は何も宣言していませんが、`Base` の `static main` が実行されます。

```
// M6.java
class Base { static void main(String[] args) { System.out.println("Base
main"); } }
class M6 extends Base { }
```

```
Base main
```

継承で候補になる物は、次の 2 つです。

- 親クラスの `main`（上の `M6`）
- 実装したインタフェースの `default void main()`

`default` メソッドはインスタンスメソッドですが、実装したクラスでは候補になります。`interface Runner { default void main() { ... } }` を `class M7 implements Runner { }` が実装すれば、`java M7` で `default` の `main` が動きます。インタフェースに `static void main(String[] args)` を書いて、そのインタフェースを直接起動しても構いません。

インスタンスの `main` を呼ぶ手順は、1.2 のインスタンス main で扱います。

SAMPLE

コマンドライン引数（args の要素数・添字・空の配列）

`java` コマンドでクラス名の後ろに並べた語が、`main` の引数の配列になります。語の区切りは空白で、空白を含めたい語は引用符で囲みます。

```
// E1.java
public class E1 {
    public static void main(String[] args) {
        System.out.println(args.length);
        if (args.length > 0) System.out.println(args[0]);
        System.out.println(args[0]);
    }
}
```

- `java E1 a b` の出力：`2`、`a`、`a` の 3 行
- 引数を渡さない `java E1` の出力：`0` の後で例外

```
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 0 out of bounds for length 0
        at E1.main(E1.java:6)
```

論点	内容
要素数	渡した語の個数ぴったり。`java E1 "hello world" 3` は 2 個で、`args[0]` は `hello world`
引数が無い時	長さ 0 の空の配列。`null` にはならない

SAMPLE

論点	内容
添字	0 から数える。範囲の外を読むと `ArrayIndexOutOfBoundsException`
要素の型	すべて `String`。`3` と渡しても `args[1]` は文字列 `"3"`
空の引数 `""`	長さ 0 の文字列が 1 個の要素になる

`main` も普通の `static` メソッドなので、プログラムの中から `main(new String[]{"x", "y"});` のように呼べます。自分の `main` を呼ぶ再帰も書けます。

プログラムは、JVM が `main` を呼ぶ所から始まります。[4] 終わるのは、次のどちらかの時です。

- デーモンでないスレッドがすべて終わった時
- `System.exit` が呼ばれた時

`main` の中で捕まえない例外が出ると、メッセージを表示して終了コード 1 で終わります。`System.exit(3)` なら、終了コードは 3 です。`main` が先に終わっても、デーモンでないスレッドが動き続けていれば、プログラムはそのスレッドが終わるまで終わりません。

SAMPLE

コンパクト・ソースファイルおよびインスタンス main メソッドを持つ Java プログラムの作成、およびコマンドラインでのJavaプログラムのコンパイルと実行 [1.2]

コンパクト・ソースファイル（クラス宣言の無いファイル・暗黙に宣言される final のトップレベルクラス）

メソッドやフィールドを、クラスの外に直接書いたファイルが、コンパクト・ソースファイル（コンパクトなコンパイル単位）です。JDK 21 から 24 までのプレビューを経て、JDK 25 で正式な機能になっています。直前の JDK 24 のプレビュー（JEP 495）での名前は「簡易ソースファイル（simple source files）」で、正式化の時に改名されました。[5]

```
// Hello.java
void main() {
    System.out.println("Hello");
}
```

コンパイラは、このファイルが「暗黙に宣言されるクラス」を持つと見なします。そのクラスの性質を表にします。[6]

項目	内容
種類	`final` のトップレベルクラス。無名パッケージに属し、アクセスはパッケージアクセス
継承と実装	`Object` を継承する。インタフェースは実装しない
コンストラクタ	引数なしのデフォルト・コンストラクタだけ

SAMPLE

項目	内容
名前	実装が決める。`javac` はファイル名から拡張子を除いた名前（ここでは `Hello`）を付ける
メンバー	ファイルに書いたメソッド、フィールド、クラス、インタフェース

`javap -p Hello.class` は、`final class Hello { Hello(); void main(); }` と表示します。

ファイルの構成は、先頭に `import`、その後にメンバー宣言の並びです。メンバーの順番は自由で、メソッドは少なくとも 1 つ必要です。`import` は、メンバー宣言の後には書けません。

`class A { }` と `void main() { ... }` が同居するファイルも、コンパクト・ソースファイルです。`A` は暗黙のクラスのメンバークラスになり、`Cmp$A.class` のような名前で出力されます。クラス宣言だけのファイルは、これまでと同じ普通のファイルです。

暗黙のクラスには、次の制約があります。

ファイルに書いた物	javac の結果
起動できる main が無い	コンパイルエラー（`void main()` か `void main(String[] args)` の形が無い）
コンストラクタ `Hello() { }`	コンパイルエラー（戻り値の型が無いメソッドと読まれる）
`static { }` や `{ }` の初期化子	コンパイルエラー
`package` 宣言	コンパイルエラー

SAMPLE

ファイルに書いた物	javac の結果
`new Hello()`	コンパイルエラー（名前をソースの中で使えない）

暗黙のクラスの名前はソースに書けないので、`new` で自分自身を作れません。`this` は使えます。`this.getClass().getName()` は `Hello` を返します。

インスタンス main メソッド（static でない main・引数なしのコンストラクタで生成してから呼ぶ）

`static` でない `main` が選ばれた時、ランチャーはまず、そのクラスを引数なしのコンストラクタで生成します。次に、できたインスタンスで `main` を呼びます。[1]

```
// M11.java
class M11 {
    int n = 5;
    M11() { System.out.println("ctor n=" + n); }
    void main(String[] args) { System.out.println("main n=" + n + "
args=" + args.length); }
}
```

```
ctor n=5
main n=5 args=0
```

コンストラクタの出力が先で、フィールドの初期値はその前に入っています。必要なコンストラクタは、`private` でない引数なしの物です。次のどれかだと、起動できません。

SAMPLE

- 引数付きのコンストラクタ `M9(int x)` だけがある。
- 引数なしのコンストラクタが `private` である。
- 抽象クラスなので、インスタンスを作れない。

最初の 2 つは「非privateのゼロ引数コンストラクタがクラスM10に見つかりません」と表示されます。抽象クラスは「抽象クラスAはインスタンス化できません」です。コンストラクタを何も書かなければ、デフォルト・コンストラクタが付くので問題ありません。

コンパクト・ソースファイルのクラスは、デフォルト・コンストラクタだけを持ちます。インスタンス `main` を書けば、そのまま動きます。継承した `protected` のインスタンス `main` も、子クラスのコンストラクタが働いてから呼ばれます。

`static` の `main` が選ばれた時は、インスタンスを作りません。`static void main()` を持つコンパクト・ソースファイルも書けます。引数付きの候補があれば、`static` かどうかに関係なく、そちらが優先です。

java.lang.IO（println・print・readln）と java.base の自動インポート（import module java.base 相当）

コンソールの入出力用に、`java.lang.IO`（Java 25 で追加）があります。`final` クラスで、メソッドはすべて `static` です。[7]

メソッド	働き
`print(Object obj)`	標準出力へ書く（改行なし）
`println(Object obj)`	標準出力へ書いて改行する
`println()`	改行だけを書く
`readln()`	標準入力から 1 行読む。末尾に達していたら `null`

SAMPLE

メソッド	働き
`readln(String prompt)`	先にプロンプトを書いてから 1 行読む

引数なしの `print()` はありません。`IO.print()` はコンパイルエラーです。`print` と `println` は `Object` を受けるので、`IO.println(null)` は `null` と出ます。

`IO` は `java.lang` にあるので、どのファイルでも `import` なしで `IO.println("...")` と書けます。ただし、`IO` の `static` メソッドは、コンパクト・ソースファイルにも自動では入りません。`println("x")` だけだと、「シンボルを見つけられません」のコンパイルエラーです。JDK 24 までのプレビューにあった自動インポートは、正式版で外れました。[5] 単純名で書きたければ、`import static java.lang.IO.*;` を自分で書きます。

```
// Rd.java
void main() {
    String name = IO.readln("name? ");
    IO.println("hello " + name);
    String again = IO.readln();
    IO.println(again);
}
```

標準入力に `Ken` と 1 行だけ与えると、出力は `name? hello Ken` の後に `null` が続きます。プロンプトの後ろは改行されません。2 回目の `readln()` は、入力の末尾で `null` を返します。

もう 1 つの自動インポートが、java.base です。コンパクト・ソースファイルは、先頭に `import module java.base;` があるかのように、java.base が公開する

SAMPLE

パッケージの `public` な型をすべて暗黙にインポートします。`java.util` の `List` や `java.time` の `LocalDate` が、`import` なしで使えます。[6]

```
// Auto.java
void main() {
    List<Integer> xs = List.of(3, 1, 2);
    System.out.println(xs + " " + new ArrayList<>(xs).size() + " " +
LocalDate.of(2025, 1, 2));
}
```

```
[3, 1, 2] 3 2025-01-02
```

同じ `List` を、普通のクラスの中で `import` なしに使うことはできません。「シンボルを見つけられません」のコンパイルエラーになります。普通のファイルは、`java.lang` の型しか自動では取り込みません。すべてのファイルに `import java.lang.*;` が暗黙にあるので、`String`・`Math`・`IO` は単純名で使えます。

javac と java でのコンパイルと実行（クラスパス・.class の置き場所）

`javac` はファイルを、`java` はクラスを指定します。`javac Foo.java` は `Foo.class` を、ソースと同じディレクトリに作ります。`-d` を付けると出力先を指定でき、ディレクトリが無くても、エラーにはなりません。クラスがパッケージに属していれば、`-d` の下にパッケージ名のディレクトリができます。[8]

SAMPLE

```
javac -d out src/com/acme/App.java      → out/com/acme/App.class
java -cp out com.acme.App               → 実行できる
java -cp out App                        → メイン・クラスAppを検出およびロードできませんでした
java -cp out com.acme.App.class         → メイン・クラスcom.acme.App.classを検出およびロードできませんでした
java com.acme.App                       → メイン・クラスcom.acme.Appを検出およびロードできませんでした
```

読み方を表にします。

項目	決まり
`java` に渡す物	完全修飾名（パッケージ名付きのクラス名）。`.class` や `.java` は付けない
`-cp`（`--class-path`）	クラスを探す場所を指定する。`CLASSPATH` 環境変数より優先する
既定のクラスパス	`-cp` も `CLASSPATH` も無い時は、カレントディレクトリ（`.`）だけ
区切り文字	Linux・macOS はコロン `:`、Windows はセミコロン `;`
`-d out` で作った物	`-cp out` で探す。`java Foo` は `.` の `Foo.class` を探すので、`out` の中は見つからない

`javac` がクラスを置く場所と、`java` が探す場所は別の話です。`package a.b;` を書いたファイルを、ディレクトリ `a/b` に置かずに `javac Foo.java` でコンパイルしても、コンパイルは通ります。しかし、実行では止まります。

SAMPLE

実行	結果
`java Foo`	`NoClassDefFoundError: Foo (wrong name: a/b/Foo)`
`java a.b.Foo`	`ClassNotFoundException: a.b.Foo`

クラスファイルが `a/b/Foo.class` の位置にあって、`-cp` がその根を指していなければ、見つかりません。

依存するクラスが別のディレクトリにある時は、クラスパスに足します。`javac -cp libout -d appout app/UseDep.java` でコンパイルし、`java -cp appout:libout UseDep` で実行します。`-cp appout` だけでは、依存クラスを見つけられません。実行時に `NoClassDefFoundError: dep/Dep` が出ます。

`javac` のメッセージには、エラーと警告があります。エラーが 1 つでもあればコンパイルは失敗し、警告だけならコンパイルは止まりません（`-Werror` を付けた時だけ、警告でも止まります）。[8]

`public` なトップレベルクラスの名前は、ファイル名と一致させます。`Wrong.java` に `public class M19` を書くと、「クラス M19はpublicであり、ファイルM19.javaで宣言する必要があります」のコンパイルエラーです。1 つのファイルに `public` なトップレベルクラスを 2 つ書くことはできません。

ソースファイル・モードの起動（java Foo.java・複数ファイルのプログラム・ファイル内の最初のクラス）

`java` に `.java` ファイルを渡すと、ソースファイル・モードになります。メモリ上でコンパイルしてから実行するので、`.class` は作られません。ファイル名の後ろの語は、`main` の引数です。[9]

SAMPLE

```
java Hello.java a b      → コンパイルして実行（Hello.class は作られない）
```

コンパクト・ソースファイルも、このモードで動きます。`void main(String[] args) { IO.println("hi " + args.length); }` を `Hello.java` に書き、`java Hello.java a b` と実行すれば `hi 2` です。コンパイルエラーがあれば「エラー: コンパイルが失敗しました」と出て、終了コードは 1 になります。

JDK 25 の java は、次の順で起動するクラスを決めます。[10]

1. ファイルの最初のトップレベルクラスに起動できる `main` があれば、そのクラス。
2. なければ、ファイル名と同じ名前で `main` を持つ別のトップレベルクラス。
3. どちらも無ければ、エラーで終わる。

`A.java` の中身ごとの結果を表にします。

A.java のクラス	結果
`First`（main 無し）、`A`（main あり）	`A` が動く（ファイル名と同じ）
`First`（main あり）、`A`（main あり）	`First` が動く（最初のクラス）
`First`（main 無し）、`Second`（main あり）	「クラスが見つかりません: A」
どのクラスにも main が無い	「クラスにmain(String[])またはmain()メソッドが見つかりません: A」

ソースファイル・モードでは、`public` クラスの名前がファイル名と違っても構いません。`script.java` に `public class Whatever` を書いても、`java script.java` で動きます。

SAMPLE

JDK 22 から、複数ファイルのプログラムも書けます。起動するファイルが、同じディレクトリの別の `.java` のクラスを使うと、それもメモリ上でコンパイルされます。`Main.java` が `Util.java` の `Util` を呼ぶ構成なら、`java Main.java` だけで動きます。サブディレクトリの `.java` は、無名パッケージのクラスとしては探されません。`lib/Tool.java` の `Tool` を使うと、「シンボルを見つけられません」のコンパイルエラーです。[10]

ソースの根は、起動するファイルの `package` 宣言から決まります。`package a.b;` のファイルは、`a/b/` の下に置く必要があります。置き場所が合わないと、「ソース・ファイルへのパスの終わりがパッケージ名a.bと一致しません」と出ます。

`--source 25` を付けると、ソースの版を指定できます。拡張子が `.java` でないスクリプトは、`--source` が無ければ動かせません。

パッケージの使用 (パッケージ宣言とインポート) [1.3]

パッケージ宣言 (ファイル先頭の 1 つ・ディレクトリ構成・無名パッケージ)

`package` 宣言は、そのファイルのクラスが属するパッケージを決めます。ファイルに 1 つだけ書け、`import` やクラス宣言より前に置きます。コメントは前に置いても構いません。[11]

書き方	結果
`package p;` が 1 つ、先頭	正しい
`import` の後に `package`	コンパイルエラー
`package` が 2 つ	コンパイルエラー
クラスの後に `package`	コンパイルエラー

SAMPLE

```
// p4/A.java
package p;
public class A { public static void main(String[] a) {
System.out.println("p.A"); } }
```

パッケージ名は、ディレクトリの構成に対応します。`com.acme` のクラスは `com/acme` に置かれ、`.class` も `com/acme` の下に出ます。`javac` はソースの置き場所を確かめません。`java` はクラスパスの根から `com/acme/App.class` を探します。

`package` 宣言が無いファイルのクラスは、無名パッケージに属します。コンパクト・ソースファイルも、常に無名パッケージです。無名パッケージは、小さなプログラムや試しのために用意された仕組みです。無名パッケージのクラスは、`import` できません。`import Plain;` のように、パッケージ名の無い宣言は構文エラーです。

パッケージの間に、特別な上下関係はありません。`oliver` と `oliver.twist` は別のパッケージで、互いに特別なアクセス権を持ちません。`import java.util.*;` は `java.util` の型だけを入れます。サブパッケージの `java.util.function.Function` は入りません。

他のパッケージのクラスは、`public` でなければ使えません。パッケージアクセスの `lib.Hidden` を別パッケージから `lib.Hidden` と書くと、「libのHiddenはpublicではありません。パッケージ外からはアクセスできません」のエラーです。

SAMPLE

import 宣言（単一型・オンデマンド・java.lang の自動インポート・同名の型の衝突）

`import` 宣言は、他のパッケージの型を単純名で使えるようにします。書かなくても、完全修飾名（`java.util.List`）なら使えます。形は 2 つです。[1][2]

形	例	入る物
単一型インポート	`import java.util.List;`	その 1 つの型だけ
オンデマンド・インポート	`import java.util.*;`	そのパッケージの、使える型すべて（サブパッケージは入らない）

`java.lang` は自動でインポートされます。`import java.lang.*;` や `import java.lang.String;` を書いても構いませんが、同じ物が重なるだけです。同じ `import` を 2 回書いても、2 回目は無視されます。使っていない `import` は、エラーになりません。

次の `import` は、コンパイルエラーです。

書いた import	理由
`import java.util.Nope;`	存在しない型
`import java.nope.*;`	存在しないパッケージ
`import java.util;`、`import java.*;`	パッケージそのものは入れられない

同じ単純名の型が複数あると、衝突が起きます。

SAMPLE

状況	結果
`import java.util.*;` と `import java.sql.*;` の後で `Date` を使う	「Dateの参照はあいまいです」。使わなければエラーにならない
上の 2 つに加えて `import java.util.Date;`	`java.util.Date`（単一型がオンデマンドに勝つ）
`import java.util.Date;` と `import java.sql.Date;`	`import` の行でエラー（同じ単純名の単一型が 2 つ）
パッケージ `q` に `q.Date` があり、`import java.util.*;` もある	`q.Date`（同じパッケージの型がオンデマンドに勝つ）
同じ `q` で `import java.util.Date;` を書く	`java.util.Date`（単一型が同じパッケージの型に勝つ）
同じファイルで宣言した `Vector` と `import java.util.Vector;`	エラー

`import java.util.Map.*;` のように、型の名前でオンデマンドにすると、その型のメンバー型（`Entry` など）が入ります。

static import とモジュールのインポート宣言（import module）

`import static` は、クラスの `static` メンバー（フィールド、メソッド、メンバー型）を、単純名で使えるようにします。[1][2]

SAMPLE

```
// Si.java
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
import static java.lang.Integer.*;
class Si { public static void main(String[] x) {
System.out.println(sqrt(16) + " " + PI + " " + MAX_VALUE + " " +
parseInt("12")); System.out.println(Math.abs(-2)); } }
```

```
4.0 3.141592653589793 2147483647 12
2
```

単一の static インポート（`import static java.lang.Math.sqrt;`）は、その名前の `static` メンバーをすべて入れます。オンデマンド（`import static java.lang.Integer.*;`）は、使える `static` メンバーをすべて入れます。

書いた物	結果
`import static java.util.Collections.sort;` の後に `Collections.max(l)`	「シンボルを見つけられません」。クラス名そのものは入らないので、別に `import java.util.*;` が要る
`import java.lang.Math.sqrt;`（`static` 忘れ）	コンパイルエラー。`Math` の中の型 `sqrt` を探す
`import java.lang.Math.*;` の後に `sqrt(4)`	コンパイルエラー。入るのはメンバー型だけ
`import static java.lang.String.length;`	「シンボルを見つけられません」。`static` でないメンバーは入らない
`private` やパッケージアクセスのメンバー	入らない

SAMPLE

書いた物	結果
`import static java.lang.Math.max;` があり、自分のクラスに `static int max(int a, int b)` がある。`max(3.0, 4.0)` を呼ぶ	コンパイルエラー。自分の `max` だけが見え、`double` を `int` にできない
`Math.abs` と `StrictMath.abs` を両方 `static` インポートして `abs(-3)`	「absの参照はあいまいです」
`Integer.MAX_VALUE` と `Long.MAX_VALUE` を両方 `static` インポートして `MAX_VALUE` を使う	「MAX_VALUEの参照はあいまいです」（使わなければエラーにならない）
`import static java.lang.Integer.MAX_VALUE;` と `import static java.lang.Long.*;`	`MAX_VALUE` は `Integer` の値（単一がオンデマンドに勝つ）

モジュールのインポート宣言 `import module モジュール名;` は、JDK 25 で正式な機能になっています。[13] 指定したモジュールが公開するパッケージの `public` なトップレベル型を、まとめてオンデマンドで取り込みます。そのモジュールを読むことで読めるようになるモジュール（`requires transitive` の先）が公開するパッケージも入ります（`import module java.sql;` だけで `java.util.logging.Logger` も単純名で使える）。`import module` は、モジュールを宣言したファイルだけの機能ではありません。

```
// Mod.java
import module java.base;
class Mod { public static void main(String[] x) { List<Integer> l = new
ArrayList<>(List.of(2,1)); Collections.sort(l); Path p = Path.of("a");
System.out.println(l + " " + p); } }
```

SAMPLE

```
[1, 2] a
```

`java.util` の `List` と、`java.nio.file` の `Path` が、`import` なしで使えます。モジュールの名前が存在しなければ、「インポートされたモジュールが見つかりません: java.nope」のエラーです。`package` 宣言の前には書けません。

モジュールのインポートは、取り込む型の数が多いので、同じ単純名の衝突が起きやすい形です。

書いた物	結果
`import module java.base;` と `import module java.desktop;` の後で `List`	`java.util.List` と `java.awt.List` があいまいで、エラー
`import module java.base;` と `import module java.sql;` の後で `Date`	`java.util.Date` と `java.sql.Date` であいまいで、エラー
上の 2 組に `import java.util.List;` を足す	単一型インポートが勝つので、`List` は `java.util.List`

同じ名前の型が複数ある時の優先順を、強い物から並べます。[14]

順	宣言
1	同じファイルで宣言した型と、単一型インポートした型
2	同じパッケージの別ファイルの型
3	オンデマンド・インポート（`java.lang` の自動インポートを含む）
4	モジュールのインポート

SAMPLE

3 と 4 の関係は、次のとおりです。`import module java.base; import java.awt.*;` の後の `List` は `java.awt.List` で、エラーになりません。`java.lang` の型（`Object` や `Record`）は、モジュールのインポートの同名より常に優先されます。同じパッケージの `q.List` も、`import module java.base;` で入る `java.util.List` に負けません。

SAMPLE

サンプルはここまでです

続き(第2章〜巻末)は、Java Silver SE25 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

Footnotes

1. Java 言語仕様 SE 25 §12.1 と §12.1.4（候補の main・起動の手順）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-12.html#jls-12.1.4 ↩ ↩2 ↩3 ↩4

2. Oracle 日本の Java SE 25 Programmer I（1Z0-827-JPN）試験ページ「試験内容チェックリスト」。https://www.oracle.com/jp/education/certification/javase-25-programmer-1/ ↩

3. `main` が起動できない時の表示は、JDK 25 の `java` コマンド（日本語ロケール）の出力。↩

4. Java 言語仕様 SE 25 §12.8（プログラムの終わり）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-12.html#jls-12.8 ↩

5. JEP 512: Compact Source Files and Instance Main Methods。https://openjdk.org/jeps/512 ↩ ↩2

6. Java 言語仕様 SE 25 §7.3（コンパクトなコンパイル単位）と §8.1.8（暗黙に宣言されるクラス）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-7.html#jls-

SAMPLE

7.3 ・ https://docs.oracle.com/javase/specs/jls/se25/html/jls-8.html#jls-8.1.8 ↩ ↩2

7. Java SE 25 API: java.lang.IO（Since 25）。https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/IO.html ↩

8. javac コマンド（`-d`・`--class-path`）と java コマンド（クラスパス）。https://docs.oracle.com/en/java/javase/25/docs/specs/man/javac.html ・ https://docs.oracle.com/en/java/javase/25/docs/specs/man/java.html ↩ ↩2

9. java コマンド「Using Source-File Mode to Launch Source-Code Programs」。https://docs.oracle.com/en/java/javase/25/docs/specs/man/java.html ↩

10. JEP 458: Launch Multi-File Source-Code Programs（起動するクラスの決め方）。https://openjdk.org/jeps/458 ↩ ↩2

11. Java 言語仕様 SE 25 §7.4（パッケージ宣言・無名パッケージ）と §7.1。https://docs.oracle.com/javase/specs/jls/se25/html/jls-7.html#jls-7.4 ↩

12. Java 言語仕様 SE 25 §7.5（import 宣言）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-7.html#jls-7.5 ↩ ↩2

13. JEP 511: Module Import Declarations と Java 言語仕様 SE 25 §7.5.5。https://openjdk.org/jeps/511 ・

SAMPLE

https://docs.oracle.com/javase/specs/jls/se25/html/jls-7.html#jls-7.5.5 ↩

14. Java 言語仕様 SE 25 §6.4.1（シャドウイング：shadowing）と §7.5.2 の例 7.5.2-1（同じパッケージの型がオンデマンド・インポートの型を隠す）。https://docs.oracle.com/javase/specs/jls/se25/html/jls-6.html#jls-6.4.1 ・ https://docs.oracle.com/javase/specs/jls/se25/html/jls-7.html#jls-7.5.2 ↩

SAMPLE

Java Silver SE 25(1Z0-827-JPN) 学習用テキスト(無料サンプル)

版

2026年11月版(Oracle公式 Java SE 25 Programmer I(1Z0-827-JPN) 試験内容チェックリスト 準拠)

発行

IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ

info@shikaku-doujou.com

Oracle、Java、および MySQL は、Oracle Corporation およびその子会社、関連会社の米国およびその他の国における登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、Oracle との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。

SAMPLE