IT資格道場

Java Silver SE11 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 11 Programmer I(1Z0-815-JPN) 試験内容チェックリスト 準拠
2026年7月版

本書のご利用にあたって

- 本書は「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。
- 無料サンプルであっても著作物です。本書の転載・改変しての再配布・二次販売を禁じます。
- 本書を AI モデルの学習データとして利用すること(学習目的の入力・データセット化を含む)を禁じます。
- 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。

不正な配布を見つけた場合は info@shikaku-doujou.com までご連絡ください。

目次

3点セットの中の位置づけ

このシリーズは3つのテキストで構成されています。

- **学習用**: これだけで問題が解けるようになる本編
- **頻出度別**: テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡しします。試験直前の最終チェックにも
- **暗記用**: 覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: これだけで問題が解けるようになる本編です

はじめに —— この教材の使い方

このテキストは、**Oracle** が実施する **Java SE 11 Programmer I（試験番号 1Z0-815-JPN）** に合格し、**Oracle Certified Java Programmer, Silver SE 11** 認定を取ることを目的に作られています。

この試験が問うのは、Java の用語を知っていることではありません。**行番号つきのプログラムを読み、「実行するとどう出力されるか」「コンパイルは通るか、通らないなら何行目か」「実行時にどの例外が出るか」を、規則に従って言い当てられること**です。本書は全章を通じて、文法規則・実行順序・API の挙動を「コードと実際の出力の対」で示します。載せているコードの出力・エラーはすべて実機（JDK・`--release 11`）で確かめたものです。

本書は **Oracle が公開している試験内容チェックリスト（12セクション・38細目）を体系化した教材**であり、実際に出題された問題を再現したものではありません。**本書は Oracle の公式教材ではなく、IT資格道場が独自に書き下ろしたオリジナルの教材です。**

プログラミング経験が浅くても読めます。 Bronze 相当の入門知識（変数・分岐・繰り返し・クラス）は本書の中でひとつずつ確かめ直します。

試験の基本情報

項目	内容
試験名	Java SE 11 Programmer I
試験番号	1Z0-815-JPN（日本語試験。認定は日本語試験の合格が対象）
関連資格	Oracle Certified Java Programmer, Silver SE 11
出題形式	選択問題
試験時間	**180分**
出題数	**80問**
合格ライン	**63%**
配信	Oracle 認定資格 ピアソン VUE 配信（監督付き）

受験料・試験時間・出題数・出題範囲は改定されることがあります。 お申し込みの前に、Oracle University 公式サイトで最新の情報をご確認ください。

Oracle はセクション別の出題比率を公表していません。 本書は「このセクションが何問出る」という数字を書きません。本書の分量の配分は、公式チェック

リストの構造と受験者の公開報告からの推定であり、公式値ではありません。

出題範囲の全体像 —— チェックリストの12セクションと本書の章

セクション	内容	本書
1	Javaテクノロジと開発環境についての理解	第1章
2	簡単なJavaプログラムの作成	第2章
3	Javaの基本データ型と文字列の操作	第3章
4	演算子と制御構造	第4章
5	配列の操作	第5章
6	クラスの宣言とインスタンスの使用	第6章
7	メソッドの作成と使用	第7章
8	カプセル化の適用	第8章
9	継承による実装の再利用	第9章
10	インタフェースによる抽象化	第10章
11	例外処理	第11章
12	モジュール・システム	第12章

本書の章の並びは、公式チェックリストの並びとまったく同じです。節見出しの末尾にある `[3.1]` のような番号は、チェックリストの細目の番号です。

時間配分 —— 1問あたり2分15秒

180分で80問ということは、1問あたり約2分15秒です。コードを読んで結末を当てる問題が中心なので、「読んだ瞬間に規則が浮かぶ」状態にしておかないと後半で時間が足りなくなります。本書の各節は、そのための「規則→コード→実測出力→よくある取り違え」の順で書いてあります。

全設問に効く判断軸 ——「3つの問い」

問い	何を切り分けるか
問い1: コンパイル時か、実行時か	エラーが出るなら、それはコンパイラが止めるのか、実行中に例外として出るのか。この試験の誤答肢はここを取り違えさせます
問い2: 何行目で決まるか	宣言・初期化・呼び出し・戻り値、どの行で規則が効くか。「行N でコンパイルエラー」の肢は行を1本ずつ検分します
問い3: 型はどう決まるか	静的型（宣言した型）と実行時の型（生成した型）。どちらが選ばれるかで、オーバーロード／オーバーライド／キャストの結末が決まります

4ステップ学習法

購入者限定テキストは3冊で1セットです。

- **STEP 1（理解する）**: 本書「学習用テキスト」を通読し、12セクションの地図と3つの問いを頭に入れる
- **STEP 2（解き方を掴む）**: 「頻出度別ノート」で頻出度の高い論点を解いて、解き方を掴む

- **STEP 3（腕試し）**：本サイトの問題演習で、コードを読んで結末を当てる形式に慣れる
- **STEP 4（再確認）**：「頻出度別ノート」に戻って総ざらい。試験直前もここへ戻る

「暗記用テキスト」は本線には入れません。通勤時間や休憩時間など、**空き時間の反復と用語の再確認**に並走させてください。

それでは、第1章から始めます。

第1章 Javaテクノロジと開発環境についての理解(セクション1)

1-1 Javaテクノロジと開発環境 [1.1]

このセクションは、この試験で唯一「コードが1行も出てこない設問」が中心になる範囲です。問われるのは用語と役割分担で、**誰が何を作り、誰が何を動かし、何がどこに含まれているか**という関係だけが答えを決めます。ここを言い切れる形で持っておくと、選択肢は読むそばから切れていきます。

以下、Javaテクノロジと開発環境の全体像を、①「Java」という語の指すもの、②コンパイルと実行の流れ、③バイトコードとネイティブコードの違い、④JVMの位置、⑤JDK・JRE・JVM の包含関係、⑥開発環境に付属するツール、⑦エディション、の7つに分けて押さえます。

1-1-1 「Java」は言語の名前であり、同時にプラットフォームの名前 [1.1]

「Java」という語は、2つの別のものを指します。

「Java」が指すもの	中身
プログラミング言語としての Java	人がプログラムを書くときの決まり(文法)。人が読み書きする側
プラットフォームとしての Java	書いたプログラムを実際に動かすための土台。機械の中で働く側

プラットフォームとは、プログラムが動くための土台のことです。一般には「機械(ハードウェア)＋基本ソフト(OS)」の組み合わせを指しますが、**Java のプ**

ラットフォームはソフトウェアだけでできています。 Java 専用のハードウェアというものは存在しません。手元の機械が何であっても、そこに Java のプラットフォームが用意されていれば Java のプログラムは動きます。

したがって、次の2つはどちらも正しい説明です。

- 「Java は言語である」…… 正しい。文法があり、人がそれに従って書きます
- 「Java はプラットフォームである」…… これも正しい。書いたものを動かす土台の名前でもあります

「Java は言語であって、プラットフォームではない」という切り分けは誤りです。「Java のプラットフォームには専用のハードウェアが必要だ」も誤りです。

1-1-2 ソースファイルからクラスファイル、そして実行まで [1.1]

Java のプログラムは、書いたその形のままでは動きません。**間に必ず1回、変換の工程が入ります。**

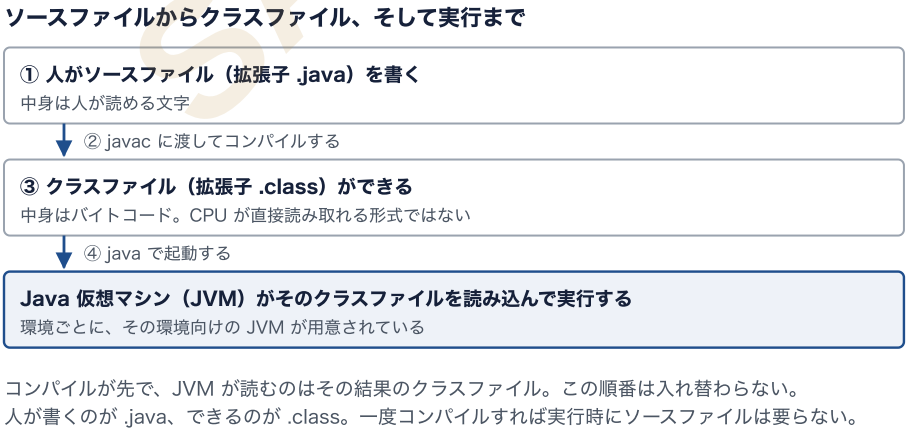

図 1-1　ソースファイルからクラスファイル、そして実行まで

```
Buoy.java ──[ javac ]──▶ Buoy.class ──[ java ]──▶ JVM が読み込んで実行
（人が書く）    コンパイル    （バイトコード）        起動
```

順番は次の4段で固定です。

1. 人が**ソースファイル**（拡張子 `.java`）を書く。中身は人が読める文字
2. `javac` に渡して**コンパイル**する
3. **クラスファイル**（拡張子 `.class`）ができる。中身は**バイトコード**
4. **Java 仮想マシン（JVM）** がそのクラスファイルを読み込んで実行する

実際に確かめると、コンパイルの前後でファイルが1つ増えます（実測）。

```
$ ls
Buoy.java
$ javac Buoy.java
$ ls
Buoy.class      Buoy.java
$ java Buoy
float
```

この順番は入れ替わりません。 取り違えの型は3つです。

よくある取り違え	実際
「JVM が読み込んでから、コンパイルする」	逆です。**コンパイルが先**で、JVM が読むのはその結果のクラスファイルです

よくある取り違え	実際
「実行してから、コンパイルする」	逆です。**実行できるのはコンパイルが終わったあと**です
「クラスファイルを人が書き、コンパイルするとソースファイルができる」	逆です。人が書くのが `.java`、できるのが `.class` です

コンパイルとは、人が書いた形を機械が読む形へ**作り変える**ことです。「写す」のでも「拡張子を付け替える」のでもありません。実物を16進で覗くと、先頭が `cafe babe` という決まった並びで始まる、人の読む文字ではないデータになっています（実測）。

```
$ xxd Buoy.class | head -2
00000000: cafe babe 0000 0037 001d 0a00 0200 0307  .......7........
00000010: 0004 0c00 0500 0601 0010 6a61 7661 2f6c  ..........java/l

$ file Buoy.class Buoy.java
Buoy.class: compiled Java class data, version 55.0 (Java SE 11)
Buoy.java:  Java source text, ASCII text
```

`version 55.0 (Java SE 11)` という表示は、このクラスファイルが Java SE 11 向けの形式で作られたことを示しています。**拡張子だけが変わった複製ではない**ことが、これで分かります。

そして、**一度コンパイルしてしまえば、実行時にソースファイルは要りません。** クラスファイルだけを別の場所へ持って行っても、そのまま動きます（実測）。

```
$ ls
Buoy.class
$ java Buoy
float
```

「クラスファイルには、実行のたびにソースファイルを読み直すための対応表が入っている」という説明は誤りです。入っているのは**実行する命令そのもの**です。

1-1-3 バイトコードとネイティブコード [1.1]

クラスファイルの中身を **バイトコード** と呼びます。バイトコードは、**Java 仮想マシンが理解する形式の命令**です。ここが Java のいちばんの特徴で、いちばん問われるところでもあります。

> **バイトコードは、CPU が直接読み取れる形式ではありません。**

- CPU が直接理解できるのは、**その CPU 向けに作られた形式の命令**だけです。これを **ネイティブコード**（機械語）と呼びます
- ネイティブコードは、CPU の種類が変われば形式も変わります
- バイトコードはどの CPU 向けでもありません。**Java 仮想マシン向け**です

`javap` というツールでクラスファイルを覗くと、バイトコードの命令列が見えます（実測）。

```
$ javap -c Buoy.class
Compiled from "Buoy.java"
public class Buoy {
  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7    // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13   // String float
       5: invokevirtual #15   // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return
}
```

`getstatic` `ldc` `invokevirtual` `return` —— これらは JVM の命令であって、特定の CPU の命令ではありません。

誤りの選択肢は3方向から作られます。

誤った説明	何が違うか
「CPU が直接読み取れる形式の命令が入ったファイルができる」	それはネイティブコードの説明です。クラスファイルの中身はバイトコードで、読むのは JVM です
「ソースファイルの中身がそのまま複製され、拡張子だけが変わる」	コンパイルは中身を作り変える工程です。同じ中身は残りません
「`javac` が作るのは `.exe` である」	`.exe` は特定の環境向けの実行ファイルの形式で、Java のコンパイルでは作られません

拡張子の対応は、どの環境でも同じです。`.jav` **ではなく** `.java` 、作られるのは `.class` です。

1-1-4 クラスファイルを実行するのは Java 仮想マシン [1.1]

できあがったクラスファイルを読み込んで動かすのは、**Java 仮想マシン**（Java Virtual Machine、略して **JVM**）です。

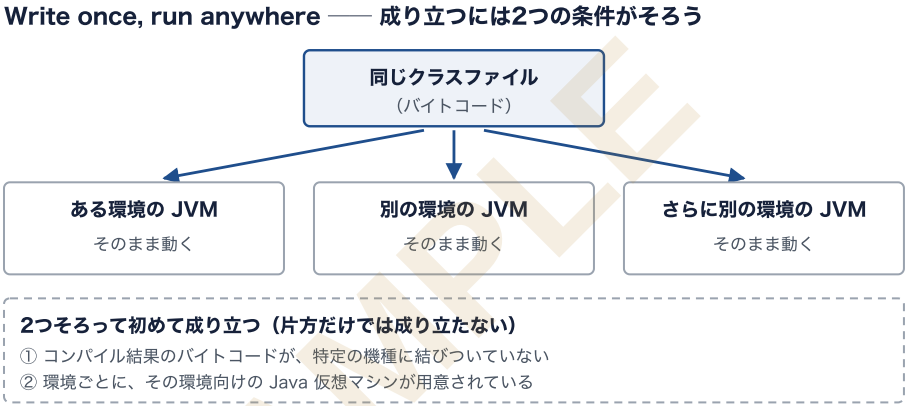

図 1-2　Write once, run anywhere —— 成り立つには2つの条件がそろう

仮想マシンとは、ソフトウェアで作られた「機械のようなもの」です。本物の CPU が機械語を実行するのと同じ立場で、JVM はバイトコードを実行します。実体はソフトウェアなので、環境ごとにその環境向けの JVM が用意されています。

誰が実行するのかを、はっきり区別してください。

候補	実際の役割
JVM	**クラスファイルを読み込んで実行する。これが正解**
OS	OS はクラスファイルの中身を理解しません。JVM を動かしているのが OS です
CPU	CPU が直接理解するのは、その CPU 向けの形式の命令だけです
`javac`	ソースファイルからクラスファイルを**作る**ツールです。動かす役目は持ちません

ここで1つ、行き過ぎた言い方に注意します。**「Java のプログラムは OS の機能をまったく使わない」というのは誤り**です。画面に文字を出す、ファイルを読み書きするといった働きでは OS の機能が使われます。JVM がその環境の OS を使って仕事をしているのであって、「OS を使わないから環境を選ばない」という理屈ではありません。

Write once, run anywhere（一度書けば、どこでも動く。略して **WORA**）という言葉は、この仕組みを一言で表したものです。意味はそのままで、**一度書いたプログラムを、Java プラットフォームが用意されているどの環境でも動かせる**ということです。

「once（一度）」という語に引きずられた誤り肢が定番です。

誤った読み方	実際
「1つのソースファイルに書けるクラスは1つだけ」という制限のこと	クラスの個数の話ではありません。1ファイルに複数のクラスを書けます（第2章 2-2 で実測します）
「一度書いたソースファイルは、あとから変更できない」という性質のこと	ソースファイルは何度でも書き直せます

誤った読み方	実際
「1つのプログラムを実行できるのは1回だけ」という制限のこと	同じプログラムは何度でも実行できます

once が掛かっているのは「書くこと」であって、実行の回数でもクラスの個数でもありません。

WORA が成り立つ理由は、**2つの条件がそろっているから**です。片方だけでは成り立ちません。

1. **コンパイル結果のバイトコードが、特定の機種に結びついていない**（機種に依存しない）
2. **環境ごとに、その環境向けの Java 仮想マシンが用意されている**

1 だけではバイトコードを読める相手がいません。2 だけでは環境ごとに別のものを作る必要が残ります。2つそろって初めて、**同じクラスファイルを別の環境へ持って行ってもそのまま動く**という結果になります。だから「環境が変わるたびにソースファイルを書き直す」「コンパイルし直して配り直す」必要はありません。

1-1-5 JDK・JRE・JVM の包含関係 [1.1]

開発環境まわりの用語は、**大きい方から小さい方への入れ子**で覚えます。

JDK・JRE・JVM の包含関係 ―― 大きい方から小さい方への入れ子

JDK は JRE を含み、JRE は JVM を含む。向きが逆の選択肢（JRE が JDK を含む）は誤り。
javac は JDK にしか入っていない。Java プラットフォーム ＝ JVM ＋ API。

図 1-3　JDK・JRE・JVM の包含関係 ―― 大きい方から小さい方への入れ子

```
JDK（開発キット）
 ├── 開発ツール（javac / javadoc / jar / javap / jshell / jdeps / jlink など）
 └── JRE（実行環境）
      ├── JVM（Java 仮想マシン）
      └── API（標準クラスライブラリ）
```

略語	正式名称	何を含むか	何ができるか
JVM	Java Virtual Machine	—	バイトコードを**実行する**中核
JRE	Java Runtime Environment	JVM ＋ API	Java プログラムを**動かす**ことだけができる
JDK	Java Development Kit	JRE ＋ 開発ツール	**作ることも動かすこともできる**

押さえどころは3つです。

- **JDK は JRE を含み、JRE は JVM を含みます。** 向きが逆になっている選択肢（「JRE が JDK を含む」「JVM が JRE を含む」）はすべて誤りです
- `javac` **は JDK にしか入っていません。** JRE だけではコンパイルできず、実行しかできません
- **Java プラットフォーム ＝ JVM ＋ API** です。これが JRE の中身にあたります

API（Application Programming Interface）は、**Java があらかじめ用意している部品の集まり**です。

- 中身は**クラス**と**インタフェース**です
- 関係するものどうしが **パッケージ** という単位にまとめられ、分類されています
- 標準で提供されているので、開発者が自分で用意するものではありません

たとえば計算に使う `Math`、文字を扱う `String` は、どちらも `java.lang` というパッケージに入っている API の部品です。誰も書いていないのに、いきなり使えます。

API を別のものと取り違える誤り肢が、そのまま出題されます。

誤った説明	実際
「API は Java 仮想マシンの別名で、プログラムを実行する役目を持つ」	JVM は**実行する側**、API は**使われる部品の側**です
「API はソースファイルをクラスファイルに変換するツールの名前」	変換を行うのは `javac` です

誤った説明	実際
「API は開発者が自分で作る部品を指し、標準では何も提供されていない」	逆です。**標準で提供されている**のが API です

なお、Java SE 11 以降は JRE だけを単体で受け取る配布形態が用意されておらず、実行環境も JDK を導入して使うのが標準的です。**それでも「JDK ⊃ JRE ⊃ JVM」という包含関係の用語そのものは変わりません。**

Java SE 9 からは、この JDK 自身がモジュールという単位に分割されています。実際に問い合わせると、`java.base` を筆頭にモジュールが並んでいることが確認できます（実測）。

```
$ java --list-modules
java.base@17.0.20.1
java.compiler@17.0.20.1
java.datatransfer@17.0.20.1
...
```

モジュール・システムそのものはセクション12で改めて扱います。ここでは「**JDK はモジュールに分割された構成になっている**」という事実だけ押さえてください。

1-1-6 開発環境に付いてくるツール [1.1]

Java の開発環境には、開発のための道具が付いてきます。名前と働きを取り違えた選択肢が、そのまま誤り肢になります。

ツール	働き	入力	出力
`javac`	ソースファイルを**コンパイル**する	`.java`	`.class`
`java`	できあがったプログラムを**実行**する	クラス名	実行結果
`javadoc`	ソース内の説明から**HTML のドキュメントを作る**	`.java`	HTML ファイル
`jar`	複数のクラスファイルを1つの書庫にまとめる	`.class` ほか	`.jar`
`javap`	クラスファイルの中身（構成・バイトコード）を表示する	`.class`	テキスト
`jshell`	対話的に断片を実行して試す	入力した文	その場の結果

実際に開発環境の `bin` を見ると、これらが並んでいます（実測。`jar` `java` `javac` `javadoc` `javap` `jshell` `jdeps` `jlink` `jmod` ほか）。`jshell` に1行渡すと、その場で答えが返ります。

```
$ echo "System.out.println(1+2)" | jshell -q -
3
```

名前の形が似ているので、働きで覚えてください。コンパイルするのが `javac`（c は compile の c）、実行するのが `java`、文書（document）を作るのが `javadoc` です。「`javac` が実行し、`java` がコンパイルする」「`javadoc` がコンパイルする」はいずれも入れ替えの誤り肢です。

1-1-7 Java SE・Java EE・Java ME の位置づけ [1.1]

Java SE・Java EE・Java ME という3つの名前は、**言語の種類ではなくエディションの名前**です。エディションが違っても、**Java という言語そのものは1つ**です。条件分岐の書き方も、繰り返しの書き方も、クラスの定義のしかたも、どのエディションでも同じです。

違うのは、**同梱されている API の範囲**です。

エディション	正式な呼び方	一言でいうと	主な用途
Java SE	Standard Edition（**標準版**）	中核。ほかの**土台になる**	デスクトップ・サーバ・汎用アプリケーション
Java EE	Enterprise Edition（企業向け）	SE の**上に積んだ**もの	企業の業務システムを**サーバ側**で動かす
Java ME	Micro Edition（小型機器向け）	SE より**絞った**もの	携帯電話・組込み機器など小型のデバイス

「中核 / 上に積む / 絞る」の3語が骨格です。包含関係には向きがあります。

```
[上に積む]   Java EE ―― 企業向けのAPIを追加した層
                ↑
[中核・土台]  Java SE ―― 標準版。ここが基準になる
                ↓
[絞る]       Java ME ―― 標準版より機能の範囲を狭めたもの
```

- **Java SE** は「3つのうちの1つ」というより、**残りの2つが位置を測るための原点**です。手元の機械で `javac` や `java` を使うとき、動いているのは Java

SE です

- **Java EE** は **Java SE を基盤として、その上に企業向けの API を追加したもの**です。単独では成り立たず、別の言語でもありません。**SE で身に付けた書き方はそのまま通用します**
- **Java ME** は標準版より機能の範囲を**絞った**ものです。**「足した側」ではなく「絞った側」**で、方向が EE と逆です。「ME は SE の古い版だ」は誤りで、小さいのは**小さい機器に載せるため**です

「SE」の2文字は短いので、読み違えが起こります。**SE は「標準版」であって、資格の等級でも「サーバ専用」でも「小型機器向け」でもありません。** 試験名にも SE の文字が入るため引っかかりやすいところです。

さらに2点。**Java EE は Oracle から Eclipse Foundation へ移管され、現在は Jakarta EE という名前で仕様が管理されています。** 変わったのは名前と管理団体で、「企業向け・サーバ側」という位置づけは続いています。また **Java Card** は IC カードのような極小デバイス向けの別のプラットフォームで、組込み機器全般向け（＝Java ME）ともサーバ側（＝Java EE）とも別物です。

エディションで**変わるもの・変わらないもの**を分けておきます。

	中身
変わらないもの（言語そのものの決まり）	制御構文の**文法**／変数の宣言のしかた／クラスの定義のしかた／**予約語の種類**／オブジェクト指向であるという**考え方**
変わるもの（プラットフォームの中身）	**含まれている API の範囲**／想定している**機器やシステムの規模**／仕様を**管理している団体**

この節は1文にまとめられます。**「エディションの違いは、含まれている API の範囲の違いである。」**

1-1-8 混同しやすい対の概念 [1.1]

混同しやすい組	見分け方
バイトコード ⇔ ネイティブコード	バイトコードは **JVM** が読む形式で機種に依存しない。ネイティブコードは **CPU** が直接読む形式で CPU ごとに違う
`.java` ⇔ `.class`	人が書くのが `.java`、`javac` が作るのが `.class`
`.class` ⇔ `.exe`	`.exe` は特定環境向けの実行ファイル形式。Java のコンパイルでは作られない
`javac` ⇔ `java`	作る側が `javac`、動かす側が `java`
`javac` ⇔ `javadoc`	コンパイルが `javac`、HTML ドキュメント生成が `javadoc`
JVM ⇔ API	JVM は動かす側、API は使われる部品の側。両方あわせて Java プラットフォーム
JDK ⊃ JRE ⊃ JVM ⇔ 逆向き	大きい方から JDK・JRE・JVM。`javac` は JDK にしか入っていない
JVM ⇔ OS / CPU	実行するのは JVM。OS は中身を理解せず、CPU はその形式の命令を直接は実行しない
「OS の機能を使わない」⇔「違いを JVM が吸収する」	画面表示やファイル入出力では OS の機能が使われる
WORA の once ＝書く回数 ⇔ 実行回数・クラスの個数	一度**書けば**どこでも動く、という意味

混同しやすい組	見分け方
コンパイル＝作り変える ⇔ 複製して拡張子を変える	中身は作り変わる。同じ中身は残らない
Java SE ⇔ Java EE ⇔ Java ME	中核 ／ 上に積む ／ 絞る。EE は SE を基盤にし、ME は SE より狭い
エディション＝API の範囲の違い ⇔ 文法の違い	文法・予約語は言語側の決まりで、エディションでは変わらない

1-1-9 章の試験ポイント [1.1]

1. **「Java」は言語の名前でもプラットフォームの名前でもある。** 片方だけが正しいという切り分けは誤りです。Java のプラットフォームは**ソフトウェアだけ**でできていて、専用のハードウェアは要りません。
2. **順番は「人が `.java` を書く → `javac` が `.class` を作る → JVM が読み込んで実行」で固定。** 逆順の選択肢はすべて誤りです。
3. **クラスファイルの中身はバイトコードで、読むのは JVM。** CPU が直接読む形式（ネイティブコード）ではありません。`.exe` も作られません。
4. **一度コンパイルすれば、実行時にソースファイルは不要。** クラスファイルだけを持って行っても動きます。
5. **JDK ⊃ JRE ⊃ JVM。** `javac` は JDK にだけ入っています。Java プラットフォーム ＝ JVM ＋ API です。
6. **API は標準で提供されるクラスとインタフェースの集まり。** JVM の別名でも、変換ツールの名前でも、開発者が自作するものでもありません。
7. **ツールの働きを入れ替えない。** `javac` ＝コンパイル、`java` ＝実行、`javadoc` ＝HTML ドキュメント、`jar` ＝書庫、`javap` ＝クラスファイルの表示、`jshell` ＝対話実行。

8. **WORA の「once」は「書くこと」に掛かる。** 実行回数の制限でも、1ファイル1クラスの制限でもありません。成立の条件は「機種非依存のバイトコード」と「環境ごとの JVM」の2つがそろうことです。
9. **エディションは API の範囲の違い。** SE が中核、EE は SE の上に積む、ME は SE より絞る。文法と予約語はエディションで変わりません。
10. **Java EE は Jakarta EE へ移管された。** 変わったのは名前と管理団体で、企業向け・サーバ側という位置づけは変わっていません。

1-2 Java 言語の主要機能 [1.2]

前節が「開発環境まわりの用語」だったのに対し、ここは **Java 言語の主要機能**、つまり言語そのものが持っている性質です。オブジェクト指向・プラットフォーム非依存・ガベージ・コレクション・強い型付け・例外処理 —— この5つが柱で、そこに「Java が意図的に持たなかったもの」が加わります。

1-2-1 オブジェクト指向であること [1.2]

Java は **オブジェクト指向**（object-oriented）のプログラミング言語です。

- データと、その扱い方をひとまとめにしたものを **クラス** として書きます
- クラスをもとにして **オブジェクト**（インスタンス）を作ります
- **オブジェクトを組み合わせて、プログラムを組み立てます**

オブジェクト指向の3本柱は、この試験の後半（セクション8・9・10）でそれぞれ扱われます。ここでは名前と一言の対応だけ持っておきます。

柱	一言でいうと	扱うセクション
カプセル化	データと操作をまとめ、外から直接触らせない	セクション8
継承	既にあるクラスの実装を受け継いで再利用する	セクション9
ポリモーフィズム	同じ呼び出しが、実際の型に応じて別の実装につながる	セクション9・10

次の説明はいずれも誤りです。

- 「クラスという仕組みを持たず、手続きを順番に並べるだけで書く」…… Java はクラスを土台にした言語です
- 「画面に部品を並べる操作だけで、文法を書かずにプログラムを作る」…… Java のプログラムは**文法に従って文字で書きます**
- 「1つのプログラムには1つの処理しか書けない」…… 処理の数に決まった上限はありません

もう1つ、**Java はクラスの多重継承を持ちません。** 1つのクラスが直接 `extends` できるスーパークラスは1つだけです（セクション9）。複数の型を同時に持たせたいときはインタフェースを使い、こちらは `implements` **で複数指定できます**（セクション10）。「Java はクラスの多重継承ができる」は誤り、「インタフェースは1つしか実装できない」も誤りです。

1-2-2 プラットフォームに依存しないこと [1.2]

1-1-4 で見た WORA が、言語側から見た主要機能としてもう一度問われます。要点だけ繰り返します。

- コンパイル結果の**バイトコードが特定の機種に結びついていない**

- **環境ごとの JVM** が用意されている
- だから**同じクラスファイルを別の環境へ持って行ってもそのまま動く**

ここで「プラットフォームに依存しないのは**ソースファイルだけ**で、クラスファイルは環境ごとに作り直す」という選択肢が出ますが、**誤り**です。作り直さなくてよいのがこの仕組みの要点です。

1-2-3 ガベージ・コレクション [1.2]

Java には、**使われなくなったオブジェクトの領域を自動的に回収するしくみ**があります。これを **ガベージ・コレクション**（garbage collection、ごみ集め）と呼びます。押さえるのは次の1点です。

> **開発者は、メモリを解放するための命令を書きません。回収は実行環境が自動で行います。**

解放を書き忘れて不具合になることが起きにくいのが利点です。「そもそも解放の命令が用意されていない」ことは、実際に書いてみると確かめられます（実測）。

```
public class FreeTry {
    public static void main(String[] args) {
        Object o = new Object();
        free(o);
    }
}
```

```
FreeTry.java:4: エラー: シンボルを見つけられません
        free(o);
        ^
  シンボル:   メソッド free(Object)
  場所: クラス FreeTry
エラー1個
```

`delete o;` と書いても同じで、`delete` という語は Java には無いので「クラス delete が見つかりません」という趣旨のコンパイルエラーになります（実測でエラー2個）。**`free` も `delete` も Java の予約語ではなく、解放用の構文でもありません。**

誤り肢は4方向から作られます。

誤った説明	実際
「使い終わったオブジェクトは、専用の命令を書いて解放する」	**そのための命令が用意されていません**
「使い終わったオブジェクトは、コンパイルの段階で取り除かれる」	どれが使われなくなるかは**動かしてみないと分かりません**
「使い終わったオブジェクトは、OS を起動しなおすまで残り続ける」	**実行中に自動で回収されます**
「回収されるのは、プログラムが終わったあとだけ」	同じ理由で誤りです

なお `System.gc()` という呼び出しは存在しますが、これは**回収を「依頼する」だけ**で、その場で回収されることを保証しません。書いても書かなくてもプログラムは普通に動きます（実測で `done` と出力されます）。「`System.gc()` を呼べば必ず即座に解放される」という選択肢は誤りです。

どのオブジェクトが対象になるかは、セクション6で「参照が無くなったオブジェクト」として改めて扱います。ここでは**「使われなくなったオブジェクトは、実行環境が自動的に回収する」**——ここまでを言い切れれば十分です。

1-2-4 強い型付け（静的型付け）[1.2]

Java は **型の扱いが厳しい**言語です。

決まり	意味
変数は**使う前に型を決めて宣言する**	宣言せずにいきなり使うことはできません
宣言した型に**合わない値は代入できない**	入れようとするとコンパイルの段階で止まります
型は**あとから入れ替わらない**	代入した値に合わせて型が変わることはありません
型は**コンパイルの段階で調べられる**	実行してみるまで分からない、ということはありません

3つとも実測で確かめられます。

```
public class TypeGuard {
    public static void main(String[] args) {
        int count = 5;
        count = "five";
        System.out.println(count);
    }
}
```

```
TypeGuard.java:4: エラー: 不適合な型: Stringをintに変換できません:
        count = "five";
                ^
エラー1個
```

宣言せずに使えば、**代入した行と読んだ行の両方**でエラーになります（実測でエラー2個）。

```
public class NoDecl {
    public static void main(String[] args) {
        count = 5;
        System.out.println(count);
    }
}
```

```
NoDecl.java:3: エラー: シンボルを見つけられません
        count = 5;
        ^
  シンボル:   変数 count
  場所: クラス NoDecl
NoDecl.java:4: エラー: シンボルを見つけられません
（エラー2個）
```

`boolean` **は他の型と行き来できません。**条件式の位置に整数を書くと、そこで止まります（実測）。

```
int n = 1;
if (n) {          // ← エラー
    System.out.println("yes");
}
```

```
IfInt.java:4: エラー: 不適合な型: intをbooleanに変換できません:
        if (n) {
            ^
エラー1個
```

この厳しさのおかげで、**動かす前に誤りを見つけられます**。「変数は宣言せずに使えて、代入した値に合わせて型が入れ替わる」「型は実行してみるまで決まらない」―― これらはいずれも Java に当てはまりません。

なお、Java 10 で入った `var` は**この厳しさを緩めるものではありません。**型を書く手間を省くだけで、型はコンパイル時に1つに確定します（3-3 で詳しく扱います）。

1-2-5 言語に組み込まれた例外処理 [1.2]

Java は、**異常が起きたときの扱いを言語の仕組みとして持っています。**戻り値でエラーコードを返して呼び出し側が確かめる、という方式ではありません。

- 異常が起きると**例外オブジェクト**が投げられ、通常の処理の流れから外れます
- `try` と `catch` で受け止め、`finally` で後始末をします
- 受け止めなければ、そのメソッドの呼び出し元へ伝わっていきます

さらに Java には、**受け止めるか、投げると宣言するかを、コンパイラが強制する例外**（チェック例外）があります。この強制があるおかげで、「異常の扱いを書き忘れたまま動いてしまう」ことが起きにくくなっています。詳しい規則はセクション11で扱います。ここでは**「例外処理は言語の機能として最初から用意されている」**という位置づけだけ押さえてください。

実行時に起きる例外は、コンパイルでは止まりません。たとえば整数の 0 除算は、**手前の出力は画面に出てから**落ちます（実測）。

```
System.out.println("before");
System.out.println(5 / 0);
```

```
before
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivZero.main(DivZero.java:4)
```

「コンパイルエラーになるか、実行時例外になるか」は、この試験の全セクションを貫く分かれ道です。ここでその感覚をつかんでおいてください。

1-2-6 Java が意図的に持たなかったもの [1.2]

言語の主要機能は「あるもの」だけでなく「無いもの」でも問われます。

Java に無いもの	代わりに
ポインタを直接操作する演算	参照は持つが、アドレス計算はできない
メモリの明示的な解放（`free` / `delete`）	ガベージ・コレクションが自動で回収する
クラスの多重継承	インタフェースを複数実装する

Java に無いもの	代わりに
演算子の定義を書き換える仕組み	用意されていない（`+` の文字列連結は言語仕様側の規定）
`goto` による任意の飛び先への移動	`goto` は**予約語として確保されているが、使えない**。ラベル付きの `break` / `continue` を使う
グローバル変数	すべての変数はクラスかメソッドに属する

最後から2番目に注意してください。**`goto` は Java の予約語です。**予約語なので識別子として使えませんが、文としての機能は用意されていません。同じ立場の語に `const` があります。「`goto` は Java の予約語ではない」という選択肢は誤りです。

こうした「持たなかった」判断は、**安全側に倒すため**です。アドレスを直接いじれない、解放し忘れが起きない、継承の経路が1本に定まる —— いずれも、書き手が誤りを起こしにくい方向へ寄せた設計になっています。

1-2-7 混同しやすい対の概念 [1.2]

混同しやすい組	見分け方
クラスの多重継承 ⇔ インタフェースの複数実装	前者はできない（`extends` は1つ）。後者はできる（`implements` は複数）
ガベージ・コレクションは実行中 ⇔ コンパイル時／再起動時	使われなくなったオブジェクトは**実行中に**回収されます
`System.gc()` は依頼 ⇔ 即座の解放を保証	依頼するだけです。保証はありません
`free` / `delete`	Java には無い。書くとコンパイルエラー

混同しやすい組	見分け方
型は宣言時に決まる ⇔ 代入した値で入れ替わる	宣言した型が固定され、合わない値は代入できません
`if (n)` (整数) ⇔ `if (b)` (`boolean`)	整数は条件式に書けません。`boolean` だけです
コンパイルエラー ⇔ 実行時例外	型が合わない＝前者。0 除算・添字超過＝後者（手前の出力は画面に出る）
`goto` は予約語 ⇔ `goto` は使える文	予約語だが**文としては使えない**。`const` も同じ立場
`var` は型推論 ⇔ `var` は動的型付け	コンパイル時に型が1つに確定します。厳しさは緩みません

1-2-8 章の試験ポイント [1.2]

1. **Java はオブジェクト指向言語で、クラスを土台にオブジェクトを組み合わせて作る。** カプセル化・継承・ポリモーフィズムが3本柱です。
2. **クラスの多重継承は無く、インタフェースは複数実装できる。** ここを入れ替えた選択肢が出ます。
3. **プラットフォームに依存しないのはクラスファイルの方。** 環境ごとに作り直す必要はありません。
4. **ガベージ・コレクションは実行中に自動で行われ、解放の命令は言語に用意されていない。** `free` も `delete` も書けません。`System.gc()` は依頼にすぎません。
5. **強い型付け（静的型付け）。** 宣言なしでは使えず、合わない値は入らず、型は入れ替わらず、検査はコンパイル時に行われます。
6. **`boolean` は他の型と行き来できない。** `if (1)` は成立しません。

7. **例外処理は言語の機能として最初からある。**受け止めるか投げると宣言するかを強制される例外があります(セクション11)。
8. `goto` と `const` **は予約語だが、文としては使えない。**「予約語ではない」という選択肢は誤りです。
9. **コンパイルエラーと実行時例外の区別が、この試験を貫く軸。**実行時例外なら、その手前の出力は画面に出ます。

サンプルはここまでです

続き(第2章〜巻末)は、Java Silver SE11 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。

Java Silver SE11 学習用テキスト(無料サンプル)

版

2026年7月版(Oracle公式 Java SE 11 Programmer I(1Z0-815-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