IT資格道場

Java Bronze SE(1Z0-818-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 Bronze(1Z0-818-JPN) 試験詳細ページ掲載の出題範囲 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・暗記用ノート・頻出度別ノートの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

Java Bronze SE（1Z0-818-JPN）の全7主題・全29章を、細目単位で1章ずつ、出題範囲の順に並べた学習用テキストです。主題1〜7を第1部〜第7部として、部番号を全体で通しています。各章は「本文」「コードで確かめる」「混同しやすい対の概念」「章の試験ポイント」の4点セットで構成しています。

目次

このセクションの使い方

Java Bronze SE（1Z0-818-JPN）の主題1「Java言語のプログラムの流れ」と主題2「データの宣言と使用」を扱う範囲です。章立ては試験の細目そのままで、第1〜3章が主題1、第4〜7章が主題2に対応します。

章	細目	扱う中心
第1章	1.1 Javaプログラムのコンパイルと実行	`javac` と `java` の役割分担、ファイル名とクラス名、`main` の形が崩れたときの着地点
第2章	1.2 Javaテクノロジーの特徴	ソースからバイトコードへ、そしてJVMへという3段の流れ、WORA、プラットフォーム＝JVM＋API、自動メモリ回収
第3章	1.3 Javaプラットフォーム各エディションの特徴	Java SE・Java EE・Java MEの位置づけと包含関係、変わるのはAPIの範囲で言語の文法ではないこと
第4章	2.1 Javaのデータ型（プリミティブ型、参照型）	8つのプリミティブ型とリテラルの既定の型、参照型との違い、既定値、自動で通る変換
第5章	2.2 変数や定数の宣言と初期化、値の有効範囲	識別子の規則、ローカル変数とフィールドの初期化の違い、`final`、ブロックと有効範囲
第6章	2.3 配列（一次元配列）の宣言と作成、使用	宣言と生成の分離、既定値、`length`、添字の境界、配列が参照型であること

章	細目	扱う中心
第7章	2.4 コマンドライン引数の利用	`java` に付けた語と `args` の対応、要素は必ず文字列であること、個数と例外

各章は次の4点セットで構成しています。

1. **本文** …… 「そう決まっている」ではなく「なぜそうなるのか」まで書いています。この試験はコードを読んで結果を答えさせる形が中心なので、書き方を丸暗記するより、**規則が動く順番**（いつコンパイラが見て、いつ実行環境が見るのか）を掴んでおくほうが取り違えが起きにくくなります
2. **混同しやすい対の概念** …… この範囲でいちばん多いひっかけは「似た2つを入れ替える」型です。入れ替えられやすい組を表にしてあります
3. **コードで確かめる** …… 最小のプログラムと、**JDK 17 で実際に動かして得た出力**を並べています。載せている出力・エラーメッセージは、すべて **OpenJDK 17.0.20.1（日本語メッセージ）** で実行して確認したものです。ただし例外のスタックトレースは、**読みやすさのためJDK内部の行を省いた抜粋**にしてあります。頭の中で追った結果と食い違ったら、そこが穴です
4. **章の試験ポイント** …… 実際に壊されやすい箇所と、その見分け方を並べています

この試験の設問はどう問われるか

特徴	内容
出題形式	選択肢から選ぶ形式です。**正答が1つ**のものが主力で、「2つ選択」のように**複数を選ばせる**ものもあります

特徴	内容
設問の立て方	**コードやコマンド行を提示して、結果を選ばせる**形が中心です。行番号が振られていることが多く、「このプログラムを実行したときの出力はどれか」「コンパイルしたときの結果はどれか」「エラーになる行はどれか」を答えさせます
概念だけを問う設問もある	第2章・第3章のように、コードがまったく出ず、**用語と位置づけの理解**だけを問う設問もあります
答えの3つの落ち場所	① **コンパイルエラー**（そもそもクラスファイルにならない）／② **コンパイルは通るが、動かす段階で失敗**（起動できない・実行時に例外が起きる）／③ **正常に動いて何かを出力する**。この3つのどれに落ちるかを毎回仕分けます
誤り肢の作り	「Java にその機能が無い」ではなく、**「別の書き方ならそうなるが、この書き方ではそうならない」**という、近い性質を持ち込む形が多くなっています

この3つの仕分けが、この範囲でいちばん点差になります。 たとえば `main` から `static` を落としても**コンパイルは通ります**（メソッドの宣言としては正しいからです）。落ちるのは動かそうとした瞬間です。一方で、書く順番を間違えて `public void static main` としてしまうと、**コンパイルの時点で**止まります。「`main` が壊れている」という見た目は同じでも、着地点がまるで違います。

誰がいつ見ているのか —— この範囲を貫く軸

プログラムが動くまでに、**チェックする役目が3回替わります。** どの役目が見ている段階の話なのかを意識すると、選択肢の切り分けが速くなります。

段階	見ている役目	見ているもの	ここで落ちると
① コンパイル	`javac`（コンパイラ）	文法・型・変数が使える範囲・変数に値が入っているか	**コンパイルエラー**（クラスファイルが作られない）
② 起動	`java`（実行を始めるコマンド）	そのクラスがあるか、`public static void main(String[] args)` があるか	**起動できない**（クラスが見つからない・メイン・メソッドが見つからない）
③ 実行	Java 仮想マシン（JVM）	添字が範囲の中か、値が `null` でないか、文字列を数値に直せるか	**実行時の例外**（`ArrayIndexOutOfBoundsException` など）

①のコンパイラは「実際に動かしてみないと分からないこと」を判定しません。配列の添字が範囲を超えるか、コマンドライン引数がいくつ渡されるか——これらは全部③まで持ち越されます。逆に、**値がどうであれ「変数に値が入っていない経路が残っている」**なら、実行すれば問題ない場合でも①で止めます。この非対称が、この範囲のひっかけの土台です。

壊され方（ひっかけ）は7つに整理できる

各章の「章の試験ポイント」では、この**名前**で型を示します。

型の名前	仕掛け
近い言葉の入れ替え	対になる2つの中身を丸ごと入れ替える（`javac` と `java`、`print` と `println`、`length` と `length()`、SE と EE）。**いちばん多い**

型の名前	仕掛け
段階の取り違え	コンパイルエラーになるものを「実行時のエラー」と書く、あるいはその逆（`static` の欠落、未初期化のローカル変数、添字の範囲外）
範囲のすり替え	効く範囲を1段ずらす（ブロックの外から変数が見えると書く、`for` の外から見えると書く）
境界の1つずれ	上限・下限を1つずらす（添字は `length` まで使えると書く、引数は `args[1]` から始まると書く）
型と値の読み違え	型の既定・自動変換・文字列の連結を取り違える（`"1" + "2"` を `3` とする、小数リテラルを `float` とする）
効かない操作を「効いた」と書く	実際には何も起きない書き方を、効いたことにする（コンパイルし直していないのに新しい内容で動くとする）
存在しない決まりの創作	もっともらしいが実在しない決まりを書く（「引数の名前は `args` でなければならない」「定数は大文字で書かないとコンパイルできない」）

最後の型は特に注意してください。この範囲の誤り肢は「日本語として自然で、Java を知らない人には正しく見える」ように書かれます。**その決まりが本当に存在するか**を思い出せるかが分かれ目になります。

第1部 Java言語のプログラムの流れ（主題1）

主題1は「Java というものが何で、書いたプログラムがどうやって動き出すのか」を押さえる範囲です。第1章が**手を動かす側**（`javac` でコンパイルし、`java` で動かす）、第2章が**その裏で何が起きているのか**（バイトコードとJVM）、第3章が**Java というプラットフォームがどう枝分かれしているか**（SE・EE・ME）。

第1章だけがコードとコマンドを読ませる章で、第2章と第3章は用語と位置づけを問う章です。ただし**3章とも中身はつながっています** —— 第1章で打つ `javac` が作るものの正体が第2章のバイトコードであり、第2章で出てくる「プラットフォーム＝JVM＋API」という定義が、そのまま第3章のエディションの違いの説明になります。

第1章 Javaプログラムのコンパイルと実行（1.1）

1-1 Javaは「2つのコマンド」で動く

Java でプログラムを動かすまでには、はっきり分かれた工程が2つあります。

1. **コンパイル**（人が書いた文字のファイルを、機械が読める形へ変換すること）を行う
2. 変換されたものを**実行**する

この2つは、それぞれ別のコマンドが受け持ちます。

工程	コマンド	何をするか
コンパイル	`javac`	ソースファイルを読み、クラスファイルを作る。プログラムは動かさない
実行	`java`	すでにあるクラスファイルを読み込んで動かす。コンパイルはしない

ソースファイルは、人が書いたプログラムの文字が入ったファイルで、名前の最後に `.java` が付きます。**クラスファイル**は `javac` が作る変換後のファイルで、名前の最後に `.class` が付きます。

```
Postcard.java  --（javac Postcard.java）-->  Postcard.class  --（java Postcard）-->  画面に出力
```

ここでいちばん大事なのは、**この2つが別々の工程で、順番が決まっている**ということです。`javac` を打たずに `java` だけを打っても、読み込むクラスファイ

ルがまだ存在しないので、プログラムは動きません。実際に試すと、次のように「クラスを見つけられなかった」という内容が表示されて終わります。

```
$ java Sticker
エラー: メイン・クラスStickerを検出およびロードできませんでした
```

このとき `java` が代わりにコンパイルを済ませてくれることはありませんし、クラスファイルが作られることもありません。**クラスファイルを作れるのは `javac` だけ**です。

逆に、`javac` を打っただけではプログラムは動きません。`javac` の仕事はファイルを作るところまでで、画面に文字が出ることもありません。コンパイルが成功したとき、`javac` は**何も表示せずに終了します**（「成功しました」という文言すら出ません）。何も出ないのは失敗ではなく、成功の合図です。

1-2 `javac` に渡すもの、`java` に渡すもの

2つのコマンドは、**渡すものの種類が違います。**ここが最も入れ替えられやすいところです。

コマンド	渡すもの	書き方の例
`javac`	**ソースファイルの名前**（`.java` まで書く）	`javac Postcard.java`
`java`	**クラスの名前**（拡張子は書かない）	`java Postcard`

`javac` はファイルを読む道具なので、読むファイルの名前をそのまま渡します。一方 `java` は「どのクラスから動かし始めるか」を指定する道具なので、**ファイルの名前ではなくクラスの名前**を渡します。

4通りの書き方を実際に打ってみると、正しいのは1つだけだと分かります。以下はすべて `Postcard.java` をコンパイルして `Postcard.class` ができている状態で試した結果です。

```
$ java Postcard
sea breeze
```

```
$ java Postcard.class
エラー: メイン・クラスPostcard.classを検出およびロードできませんでした
```

`java` に `Postcard.class` と書くと、`Postcard.class` という**名前のクラス**を探しに行ってしまい、そんなクラスは無いので見つけられません。拡張子は付けてはいけない、と覚えてください。

```
$ javac Postcard
エラー: クラス名'Postcard'が受け入れられるのは、注釈処理が明示的にリク
エストされた場合のみです
エラー1個
```

`javac` に拡張子を付けずに渡すと、この見慣れない文言が出ます。文言の意味を覚える必要はありません。**「`javac` に拡張子を落とすと叱られる」**という事実だけ押さえておけば十分です。

```
$ javac Postcard.class
エラー: Postcard.classは無効なフラグです
```

`javac` がコンパイルできるのはソースファイル（`.java`）だけです。クラスファイル（`.class`）はすでに変換が終わったものなので、`javac` の材料にはなりません。

そして、どの書き方であっても **`javac` はプログラムを実行しません。**「`javac` を打ったらその場で結果が表示された」という説明を見かけたら、それは誤りです。

クラス名は大文字と小文字まで一致させる

Java はクラス名の**大文字と小文字を区別します。** `Postcard` というクラスを `java postcard` と小文字で起動しようとすると、次のようになります。

```
$ java postcard
エラー: メイン・クラスpostcardを検出およびロードできませんでした
```

同じつづりでも、大文字小文字が違えば別の名前として扱われます。ファイル名についても同じです。

1-3 `javac` が作るのは「クラスごとの `.class`」

`javac` が作るのは、**クラス1つにつき1つのクラスファイル**です。名前は**クラスの名前**から決まり、最後に必ず `.class` が付きます。

`Postcard` というクラスを含む `Postcard.java` をコンパイルすると、同じ場所に `Postcard.class` ができ、ディレクトリには次の2つが並びます。

```
$ javac Postcard.java
$ ls
Postcard.class
Postcard.java
```

ここで、次の3つは**いずれも作られません。**誤り選択肢としてよく置かれるので、はっきり否定できるようにしておいてください。

作られないもの	理由
その環境専用の実行ファイル（`Postcard.exe` など）	Java は環境を選ばずに動かすため、環境ごとの実行ファイルではなく共通の形式であるクラスファイルを作ります（この仕組みは第2章で扱います）
メソッドごとのファイル（`main.class` など）	作られる単位は**クラス**であって、メソッドではありません。`main` はメソッドの名前なので `main.class` はできません
拡張子の付かないファイル（`Postcard`）	実行時に `java Postcard` と書くので同じ名前のファイルができそうに見えますが、実際にできるのは `Postcard.class` です

1つのソースファイルに複数のクラスを書いたとき

1つのソースファイルには、クラスを2つ以上書くことができます。その場合、**クラスの数だけクラスファイルができます。**

```
public class Wharf {
    public static void main(String[] args) {
        System.out.println(Anchor.depth());
    }
}
class Anchor {
    static int depth() {
        return 24;
    }
}
```

これを `Wharf.java` という名前で保存してコンパイルすると、次の2つができます。

```
$ javac Wharf.java
$ ls
Anchor.class
Wharf.class
Wharf.java
```

`public` が付いているかどうかは、クラスファイルができるかどうかとは**関係ありません。** `public` の付いていない `Anchor` も、同じようにクラスファイルになります。また、**クラスファイルが他のクラスファイルの中にまとめられることもありません。** どのクラスも対等に、それぞれ別のファイルになります。

エラーが1つでもあれば、クラスファイルはできない

`javac` は文法の誤りを見つけると、その場でコンパイルをやめてエラーを表示します。たとえば、文の終わりに付けるセミコロン（`;`）を落とすと、次のようになります。

```
1: public class Bicycle {
2:     public static void main(String[] args) {
3:         int wheels = 2
4:         System.out.println(wheels);
5:     }
6: }
```

```
$ javac Bicycle.java
Bicycle.java:3: エラー: ';'がありません
        int wheels = 2
                     ^
エラー1個
$ ls
Bicycle.java
```

`Bicycle.java:3:` の `3` は、誤りが見つかった**行の番号**です。そして注目すべきは `ls` の結果で、**`Bicycle.class` はできていません。**

- これは「気を付けてください」という**警告ではなく、エラー**です。エラーが出た場合、クラスファイルは作られません
- クラスファイルが無いので、`java` で実行する段階まで進めません

つまり「コンパイルは成功するが、実行しても何も出ない」という結末にはなりません。**そもそもコンパイルが成功していない**からです。

1-4 `public` を付けたクラスは、ファイル名と同じ名前にする

Java には、**`public` を付けたクラスは、クラス名と同じ名前のソースファイルに書かなければならない**という決まりがあります。

`Sunlight` という `public` クラスを `Almanac.java` というファイルに書くと、コンパイルは通りません。

```
$ javac Almanac.java
Almanac.java:1: エラー: クラス Sunlightはpublicであり、ファイル
Sunlight.javaで宣言する必要があります
public class Sunlight {
       ^
エラー1個
```

エラー文言が「ファイル `Sunlight.java` で宣言する必要があります」と、**あるべきファイル名を名指ししてくれる**のが特徴です。この食い違いは**コンパイルの段階で見つかる**ので、実行まで進むことはありませんし、クラスファイルは1つもできません。

「ファイル名から `Almanac.class` ができる」という説明も誤りです。**クラスファイルの名前はファイル名ではなくクラス名から決まる**ので、`Almanac.class` という名前になることはありません。

`public` は1つのファイルに1つまで

同じ理由から、1つのソースファイルに `public` クラスを2つ書くこともできません。両方をファイル名と一致させることは不可能だからです。

```
$ javac Duet.java
Duet.java:6: エラー: クラス Twinはpublicであり、ファイルTwin.javaで宣言する必要があります
public class Twin {
       ^
エラー1個
```

`public` が付いていなければ、名前は自由

この決まりが効くのは **`public` を付けたクラスだけ**です。`public` の付かないクラスは、ファイル名と違う名前でも問題なくコンパイルできます。

```
class Yardstick {
    public static void main(String[] args) {
        System.out.println("30 cm");
    }
}
```

これを `Toolbox.java` という名前で保存しても、コンパイルは成功します。

```
$ javac Toolbox.java
$ ls
Yardstick.class
Toolbox.java
```

できあがったクラスファイルの名前は、**ファイル名ではなくクラス名から決まる**ので `Yardstick.class` です。したがって実行するときも、`java` にはクラス名の `Yardstick` を渡します。

```
$ java Yardstick
30 cm
```

```
$ java Toolbox
エラー: メイン・クラスToolboxを検出およびロードできませんでした
```

`Toolbox` はファイルの名前であってクラスの名前ではないので、`java Toolbox` では起動できません。**`java` に渡すのは、いつでもクラスの名前です。**

1-5 参照している別のクラスも、一緒にコンパイルされる

`javac` に渡すソースファイルは、必ずしも全部並べる必要はありません。`javac` は、指定されたファイルが使っているクラスを同じ場所から探し、まだクラスファイルが無ければ**そのソースファイルも一緒にコンパイルします。**

```
// Greenhouse.java
public class Greenhouse {
    public static void main(String[] args) {
        System.out.println(Blossom.color());
    }
}
```

```
// Blossom.java
public class Blossom {
    static String color() {
        return "violet";
    }
}
```

`Greenhouse` は `Blossom` を使っています。ここで `javac Greenhouse.java` とだけ打っても、両方のクラスファイルができます。

```
$ javac Greenhouse.java
$ ls
Blossom.class
Blossom.java
Greenhouse.class
Greenhouse.java
$ java Greenhouse
violet
```

「使っているクラスのファイル名を指定していないからコンパイルに失敗する」ということはありません。また `Blossom.class` も作られているので、実行時にクラスが見つからずに失敗することもありません。

1-6 実行の入口 —— `main` メソッド

`java` コマンドは、指定されたクラスの中から**決まった形をした入口**を探し、そこから実行を始めます。その入口が次の形です。

```
public static void main(String[] args) {
}
```

`java` が入口として認めるのは、次の4つをすべて満たすものです。

決まっている部分	内容
`public`	付いていること
`static`	付いていること
`void`	戻り値の型が `void`（値を返さない）であること
`String[]`	受け取る引数が、文字列の配列1つであること

反対に、**決まっていない部分もあります。**

- **引数の名前は自由です。** `args` でも `items` でも `parcels` でも構いません。`String[]` という型さえ合っていれば、入口として選ばれます

実際に、引数名を変えたものを動かすと問題なく実行できます。

```
public class Pushcart {
    public static void main(String[] parcels) {
        System.out.println("rolling");
    }
}
```

```
$ javac Pushcart.java
$ java Pushcart
rolling
```

「引数の名前が `args` でなければならない」という決まりは**存在しません。**名前が違うことがコンパイルエラーの理由になることも、実行できない理由になることもありません。また、表示されるのは `System.out.println` に書いた文字列であって、引数の名前が画面に出るわけでもありません。

なお、`java` コマンドのクラス名の後ろに語を並べると、その語がプログラムに渡されます。`java Pushcart spring autumn` のように書いても、エラーにはなりません。渡された語を**取り出して使う方法**は第7章で扱います。

`main` が無いクラスもコンパイルはできる

`main` を持たないクラスも、**コンパイルは問題なくできます。**クラスファイルを作るだけなら入口は要らないからです。

```
public class Crayon {
    static String tip() {
        return "sharp";
    }
}
```

```
$ javac Crayon.java
$ ls
Crayon.class
Crayon.java
$ java Crayon
エラー: メイン・メソッドがクラスCrayon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)
```

コンパイルは成功してクラスファイルもできましたが、`java` で起動しようとすると入口が見つからず失敗しました。このとき、`tip` の中の `"sharp"` が表示されることはありません。**実行が始まっていない**ので、メソッドの中身が動く機会がそもそも無いからです。また、これはエラーで終了しており、正常終了ではありません（終了コードは `1` でした）。

1-7 `main` が1か所崩れたとき、どこで止まるか

この範囲でいちばん問われるのが、**「コンパイルで止まるのか、実行で止まるのか」**の見分けです。

`main` の形が決まりから外れていても、**メソッドの書き方としては文法どおり**であることがほとんどです。文法どおりなら `javac` に止める理由はないので、**コンパイルは成功します。** 止まるのは、`java` が入口を探す段階です。

実際に4通り崩して測った結果が次の表です。

崩し方	コンパイル	実行	表示される内容
`static` を落とす	**成功**	失敗	`エラー: メイン・メソッドがクラスCabinのstaticではありません。`
戻り値を `int` にする	**成功**	失敗	`エラー: メイン・メソッドはクラスCargoのvoid型の値を返す必要があります。`
引数を `String` にする（配列にしない）	**成功**	失敗	`エラー: メイン・メソッドがクラスTeapotで見つかりません。`
`public` を外す	**成功**	失敗	`エラー: メイン・メソッドがクラスHedgeで見つかりません。`

4つとも、コンパイルは成功しています。 クラスファイルもできています。たとえば引数を `String` にした場合、それは「文字列を1つ受け取るメソッド」として文法的に正しいので、`javac` は何も言いません。ただし `java` が探しているの

は `String[]` を受け取るものなので、別のメソッドとして扱われ、入口としては選ばれません。

その結果、`System.out.println` が書かれていても**1文字も表示されません。**実行が始まっていないからです。同じ理由で、「引数の個数だけ繰り返して表示される」といったこともありません。

一方、**コンパイルで止まるのは、文法そのものが壊れている場合**です。1-3 で見たセミコロンの欠落がその例で、この場合はクラスファイルが作られず、実行の段階に進めません。

> コンパイルで止まる　　… セミコロンが無い、かっこが閉じていない、`public` クラス名とファイル名が違う
> 実行で止まる（起動失敗）… `main` の `public` / `static` / `void` / `String[]` のどれかが欠けている、`main` が無い

1-8 画面に出す —— print と println

画面に文字を出すには `System.out.print` と `System.out.println` を使います。違いは**改行するかどうか**の1点だけです。

書き方	動き
`System.out.print(...)`	渡された文字をそのまま出す。**改行しない**
`System.out.println(...)`	渡された文字を出したあとに、**改行する**

`println` の末尾の `ln` は「行（line）」の意味だと考えると、覚えやすくなります。

```
public class Weathervane {
    public static void main(String[] args) {
        System.out.print("north");
        System.out.println("south");
        System.out.print("east");
        System.out.print("west");
        System.out.println();
    }
}
```

```
$ javac Weathervane.java
$ java Weathervane
northsouth
eastwest
```

読み解くと次のようになります。

- `print("north")` は改行しないので、次に出る `south` が同じ行に続きます。`println("south")` が改行するので、1行目は `northsouth`
- `print("east")` と `print("west")` はどちらも改行しないので `eastwest` と続きます
- 最後の `println()` は、渡すものが無くても**改行だけを出します**。これで2行目が終わります

ここで押さえるべきなのは、**「1つの文につき1行」ではない**ということです。行が変わるのは `println` を実行したときだけで、`print` をいくつ並べても行は

変わりません。逆に、`println` が2回あれば必ず2回改行されるので、表示が1行にまとまることもありません。

1-9 ソースを直したら、コンパイルからやり直す

`java` が読むのは**クラスファイルだけ**で、ソースファイルは見ていません。ここから、初心者がとても多く踏む落とし穴が生まれます。

```
$ javac Window.java
$ java Window
open
```

この状態で、ソースファイルの中の `"open"` を `"closed"` に書き換えて保存し、**`javac` を打ち直さずに** `java` だけを打つと、こうなります。

```
$ java Window
open
```

書き換える前の `open` が表示されました。ソースファイルを書き換えても、**クラスファイルは前回コンパイルしたときの内容のまま**だからです。`java` はソースファイルの中身を確認しないので、食い違っていても文句を言いませんし、実行できなくなることもありません。

書き換えた内容を動かすには、**もう一度 `javac` を実行してクラスファイルを作り直します。**

```
$ javac Window.java
$ java Window
closed
```

また、実行されるのはクラスファイル1つ分の内容だけなので、書き換える前と後の両方が動いて2行表示される、といったことも起きません。

「直したのに変わらない」と思ったときは、コンパイルし忘れを疑う。これがこの節の要点です。

混同しやすい対の概念

混同しやすい組	見分け方
`javac` ⇔ `java`	`javac` はコンパイル**だけ**（プログラムは動かない）。`java` は実行**だけ**（クラスファイルは作らない）。名前が似ているので入れ替えられやすい
`javac` **に渡すもの** ⇔ `java` **に渡すもの**	`javac` は**ソースファイル名**（`javac Postcard.java`）。`java` は**クラス名**（`java Postcard`）。`java Postcard.class` も `javac Postcard` も動かない
ファイル名 ⇔ **クラス名**	クラスファイルの名前も、`java` に渡す名前も、**クラス名**から決まる。ファイル名から決まるのではない
`public` **クラス** ⇔ `public` **でないクラス**	ファイル名と一致させる決まりが効くのは `public` を付けたクラスだけ。`public` でなければファイル名と違ってよい

混同しやすい組	見分け方
コンパイルで止まる ⇔ 実行で止まる	文法が壊れている（セミコロン欠落・`public` クラス名とファイル名の不一致）＝コンパイルで止まる。`main` の形が決まりから外れている＝コンパイルは通り、実行で止まる
エラー ⇔ 警告	エラーが出たらクラスファイルは**できない**。セミコロンの欠落は警告ではなくエラー
`main` が無い ⇔ コンパイルできない	`main` が無くてもコンパイルはできる。できないのは**起動**のほう
`print` ⇔ `println`	`print` は改行しない。`println` は改行する。行が変わるのは `println` のときだけ
引数の型 `String[]` ⇔ 引数の名前 `args`	型は決まっているが、名前は自由。「名前が `args` でないとだめ」という決まりは無い
ソースファイルを書き換えた ⇔ クラスファイルが変わった	書き換えただけでは変わらない。`javac` を打ち直して初めて反映される

コードで確かめる

① 2つのコマンドを順に打つ

```
public class Postcard {
    public static void main(String[] args) {
        System.out.println("sea breeze");
    }
}
```

```
$ javac Postcard.java
$ ls
Postcard.class
Postcard.java
$ java Postcard
sea breeze
```

ここで確かめたこと: `javac` は成功しても何も表示せず、`Postcard.class` を作るだけ。実行して文字が出るのは `java` を打ったとき。もし `javac` を飛ばして `java Postcard` だけを打つと、`エラー: メイン・クラスPostcardを検出およびロードできませんでした` となって動きません。

② `public` クラス名とファイル名が食い違う

ファイル名 `Almanac.java`、中身は次のとおりです。

```
1: public class Sunlight {
2:     public static void main(String[] args) {
3:         System.out.println("clear sky");
4:     }
5: }
```

```
$ javac Almanac.java
Almanac.java:1: エラー: クラス Sunlightはpublicであり、ファイル
Sunlight.javaで宣言する必要があります
public class Sunlight {
       ^
エラー1個
$ ls
Almanac.java
```

ここで確かめたこと: コンパイルの段階で止まる。クラスファイルは1つもできず、`Almanac.class` も `Sunlight.class` も残らない。実行時に見つかる誤りではない。

③ 引数を `String[]` から `String` に変える

```
public class Teapot {
    public static void main(String label) {
        System.out.println("hot water");
    }
}
```

```
$ javac Teapot.java
$ java Teapot
エラー: メイン・メソッドがクラスTeapot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)
```

ここで確かめたこと: `javac` は何も言わずに終了し、`Teapot.class` はできる。止まるのは実行のとき。`"hot water"` は表示されない。

④ `print` と `println` の並び

```
public class Weathervane {
    public static void main(String[] args) {
        System.out.print("north");
        System.out.println("south");
        System.out.print("east");
        System.out.print("west");
        System.out.println();
    }
}
```

```
$ javac Weathervane.java
$ java Weathervane
northsouth
eastwest
```

ここで確かめたこと: 表示は2行。`print` では行が変わらず、`println` のところだけで行が変わる。文の数（5つ）と行の数（2つ）は一致しない。

⑤ 別ファイルのクラスを使っている場合

```
public class Greenhouse {
    public static void main(String[] args) {
        System.out.println(Blossom.color());
    }
}
```

```
public class Blossom {
    static String color() {
        return "violet";
    }
}
```

```
$ javac Greenhouse.java
$ ls
Blossom.class
Blossom.java
Greenhouse.class
Greenhouse.java
$ java Greenhouse
violet
```

ここで確かめたこと: `Blossom.java` を指定していないのに `Blossom.class` もできている。使っているクラスのソースファイルが同じ場所にあれば、`javac` がまとめてコンパイルする。

章の試験ポイント

1. **`javac` は作るだけ、`java` は動かすだけ**（ひっかけの型: 近い言葉の入れ替え）。`javac` はプログラムを実行せず、`java` はクラスファイルを作りません。「`javac` を打ったら結果が表示された」「`java` を打ったらコンパイルもされた」は、どちらも誤りです。

2. **`javac` にはソースファイル名、`java` にはクラス名**（ひっかけの型: 近い言葉の入れ替え）。正しいのは `javac Postcard.java` と `java Postcard` の組み合わせだけです。`java Postcard.class` は「その名前のクラス」を探して見つけられず、`javac Postcard` は拡張子が無いことを叱られ、`javac Postcard.class` はクラスファイルを材料にできません。

3. **`javac` を飛ばして `java` だけ打つと動かない**（ひっかけの型: 段階の取り違え）。クラスファイルがまだ無いので「クラスを検出およびロードできませんでした」となります。`java` がその場でコンパイルしてくれることも、クラスファイルが作られることもありません。

4. **できるのは `.class`、単位はクラス**（ひっかけの型: 効かない操作を「効いた」と書く）。その環境専用の実行ファイル、拡張子の付かないファイル、メソッドごとのファイルは作られません。1つのソースファイルに2つのクラスを書けば、`public` の有無に関係なくクラスファイルは2つできます。片方が片方の中にまとめられることもありません。

5. **`public` クラス名とファイル名の一致はコンパイルの条件**（ひっかけの型: 段階の取り違え）。食い違えばコンパイルの段階で止まり、クラスファイルは1つもできません。実行時に見つかる誤りではありません。`public` は1つのファイルに1つまでで、大文字小文字まで一致させる必要があります。

6. `public` **でないクラスは、ファイル名と違う名前でよい**（ひっかけの型: 範囲のすり替え）。この決まりが効くのは `public` を付けたクラスだけです。この場合、クラスファイルの名前も `java` に渡す名前もクラス名から決まるので、ファイル名では起動できません。

7. `main` **の形が崩れても、たいていコンパイルは通る**（ひっかけの型: 段階の取り違え）。`static` を落とす、戻り値を `int` にする、引数を `String` にする、`public` を外す —— どれもメソッドの書き方としては正しいので `javac` は通し、`java` が入口を探す段階で失敗します。このとき出力は1文字も出ません。

8. **引数の名前は自由**（ひっかけの型: 存在しない決まりの創作）。決まっているのは `public static void main` と、引数が `String[]` であることまでです。「名前が `args` でないとコンパイルできない」「実行できない」はどちらも誤りで、引数の名前が画面に表示されることもありません。

9. `main` **が無くてもコンパイルはできる**（ひっかけの型: 段階の取り違え）。クラスファイルは作られます。できないのは起動のほうで、エラーになって終了します（正常終了ではありません）。クラスの中のメソッドが勝手に動くこともないので、その中の文字列が表示されることもありません。

10. **文法の誤りはエラーであって警告ではない**（ひっかけの型: 効かない操作を「効いた」と書く）。セミコロンの欠落などがあると `javac` はエラーを表示して止まり、クラスファイルは作られません。「警告は出るがクラスファイルはできる」は誤りです。

11. `print` **は改行しない、**`println` **は改行する**（ひっかけの型: 近い言葉の入れ替え）。行が変わるのは `println` のときだけです。文を1つ書くたびに行が変わるわけでもなく、`println` があるのに1行にまとまることもありません。

12. **ソースを書き換えたら `javac` からやり直す**（ひっかけの型: 効かない操作を「効いた」と書く）。`java` はクラスファイルだけを読むので、書き換えただけでは古い内容が動きます。食い違っていてもエラーにはならず、前と後の両方が動くこともありません。

13. **使っているクラスも一緒にコンパイルされる**（ひっかけの型: 存在しない決まりの創作）。同じ場所にソースファイルがあれば、`javac` に名前を並べなくてもまとめてコンパイルされます。「全部のファイル名を指定しないとコンパイルできない」という決まりはありません。

サンプルはここまでです

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

Java Bronze SE(1Z0-818-JPN) 学習用テキスト(無料サンプル)

版

2026年7月版(Oracle公式 Java SE Bronze(1Z0-818-JPN) 試験詳細ページ掲載の出題範囲準拠)

発行

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

お問い合わせ

info@shikaku-doujou.com