IT資格道場

Java Silver SE 17(1Z0-825-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 17 Programmer I(1Z0-825-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・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 Silver SE17（1Z0-825-JPN）の全8主題・全32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1〜2を第1部・第2部、主題3〜4を第3部・第4部、主題5〜6を第5部・第6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」（章によっては「コンパイルが通る書き方・通らない書き方」も加えた構成）の組で作られています。

目次

    - 第31章 カスタム例外の作成（細目 6.4）
    - 第32章 multi-catch 句の使用（細目 6.5）
- 巻末 —— 全範囲の総まとめ

このセクションの使い方

Java Silver SE 17（1Z0-825-JPN）の主題1「Java の概要と簡単なJavaプログラムの作成」と主題2「Javaの基本データ型と文字列の操作」を扱う範囲です。章立ては細目そのままで、第1〜3章が主題1、第4〜8章が主題2に対応します。

章	細目	扱う中心
第1章	1.1 mainメソッドを持つ実行可能なJavaプログラムの作成	起動される main の4条件、「コンパイルは通るか／起動できるか」の切り分け、`System.out.println` のオーバーロード選択
第2章	1.2 コマンドラインでのJavaプログラムのコンパイルと実行	`javac` と `java` の役割分担、`-d` と `-cp`、ファイル名と public クラス名、単一ファイル実行、`args`
第3章	1.3 パッケージの使用（パッケージ宣言とインポート）	`package` の位置、単一型／オンデマンド・インポート、`import static`、`java.lang` の暗黙インポート、同名衝突、無名パッケージ

章	細目	扱う中心
第4章	2.1 変数の宣言および初期化と変数のスコープ	8つのプリミティブ型とリテラル、フィールドとローカル変数の初期化の違い、明確な代入、スコープ、隠蔽、`final`、型変換
第5章	2.2 ローカル変数型推論（var）	`var` を書ける位置と書けない位置、型が固定されること、推論結果で出力が変わる場面、予約型名
第6章	2.3 文字列の作成と操作（テキスト・ブロックを含む）	不変性、文字列プールと `==` / `equals`、String の主要メソッド、`StringBuilder`、テキスト・ブロック
第7章	2.4 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・使用	宣言の3表記、既定値、初期化子の書ける場所、`length`、実行時例外、二次元とジャグ配列、共変性、`Arrays`
第8章	2.5 ArrayListの使用	`add` / `set` / `get` / `remove` のオーバーロード選択、オートボクシングと `Integer` のキャッシュ、拡張 for と `ConcurrentModificationException`

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。Java Silver はコードを読んで結果を答えさせる形が中心なので、規則を丸暗記するより**規則が動く順番**（いつコンパイラが見て、いつ JVM 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** …… 最小のコードと、**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（Homebrew・日本語ロケール）**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 内部のフレームを省いた抜粋**にしてあります（`java.base/...` の行番号は JDK のパッチ版が変わると動くため、覚える対象ではありません）。`[C@7344699f` のようなハッシュ値も実行のたびに変わります。**覚えるのは、例外の種類・メッセージ本文・どの行で起きるか**の3つです。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

Java Silver の設問はどう問われるか

特徴	内容
出題形式	四肢択一（正答1つ）が主力。ほかに**複数選択**（「2つ選べ」など）がある。**コマンドや綴りを打たせる入力式は無い**
設問の立て方	ほぼ全問が**コードを提示して結果を選ばせる**形。行番号が振られていることが多く、「エラーになる行はどれか」「出力はどれか」「実行時に何が起きるか」を答えさせる

特徴	内容
答えの3値	① **コンパイルエラー**（そもそも .class にならない）／② **コンパイルは通るが実行時に失敗**（例外・起動失敗）／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る
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」**という近接した性質を持ち込む形が多い

この3値の仕分けが、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**（メソッド宣言としては正しいので）。落ちるのは起動の瞬間です。一方 `public void static main` のように修飾子を戻り値の型より後ろに置くと、**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

コードが実行されるまでに、**チェックする主体が3回替わります。**どの主体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	見ている主体	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・スコープ・明確な代入・アクセス可否。**変数に書かれた型**しか見ない	コンパイルエラー（`.class` が作られない）

段階	見ている主体	見ているもの	ここで落ちると
② 起動	`java`（起動ツール）	クラスパス上にクラスがあるか、`public static void main(String[])` があるか	起動失敗（`ClassNotFoundException` / `NoClassDefFoundError` / メイン・メソッドのエラー）
③ 実行	JVM	添字が範囲内か、実体の型が合うか、`null` でないか	実行時例外（`ArrayIndexOutOfBoundsException` など）

①のコンパイラは「実行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、要素数が負になるか、`Object[]` の実体が本当は `String[]` か ―― これらは全部③まで持ち越されます。逆に、**変数の値がどうであれ経路として代入されない可能性が残る**なら、実行すれば問題ない場合でも①で止めます（明確な代入）。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

本文中の「試験ポイント」では、この番号で型を示します。

型	仕掛け
① 近接概念ペアの入れ替え	対になる2つの中身を丸ごと入れ替える（`length` と `length()`、`trim` と `strip`、`add(int,E)` と `set(int,E)`、`remove(int)` と `remove(Object)`、`toString` と `deepToString`。**最頻**）

型	仕掛け
② 段階の取り違え	本当はコンパイルエラーになるものを「実行時に例外」と書く、あるいはその逆（`static` 欠落／`Object[]` への異種格納／未初期化ローカル変数）
③ 適用範囲のすり替え	効く範囲を1段ずらす（オンデマンド・インポートが**サブパッケージまで**届くと書く、`java.lang` の暗黙インポートが**静的メンバまで**運ぶと書く、`import static X.*` が**型名 X も**運ぶと書く）
④ 境界の付け替え	上限・下限を1つずらす（`substring` の開始位置＝長さは可、`add(int,E)` は `size` まで可だが `get` は `size - 1` まで、`Integer` のキャッシュは 127 まで）
⑤ 数値の読み違え	基数・切り捨て・昇格を取り違える（`017` は8進の15、`int` どうしの除算は切り捨て、`char + int` は `int`）
⑥ 効かない操作を「効いた」と書く	戻り値を捨てているのに効いたことにする（`s.toUpperCase();` だけ書いて `s` が変わったとする）、拡張 for の変数に代入して元が変わったとする
⑦ 存在しない規則の捏造	もっともらしいが実在しない決まりを書く（「`package` 宣言は1行目でなければならない」「`main` は予約語」「引数名は `args` でなければならない」「`var` は予約語だから変数名に使えない」）

⑦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の概要と簡単なJavaプログラムの作成（主題1）

主題1は「1本のプログラムを、書いて・コンパイルして・起動する」までの一本道です。第1章が**プログラムの入口（main）**、第2章が**そこへ至るコマンド操作**、第3章が**クラスをどう置いてどう見つけさせるか（パッケージ）**。3章とも、最後は「クラスの名前と置き場所が合っているか」という同じ話に収束します。

第1章 mainメソッドを持つ実行可能なJavaプログラムの作成（1.1）

1-1 プログラムが動き出すまでに、何が起きているか

Java のソースファイル（`.java`）は、そのままでは実行できません。`javac` がクラスファイル（`.class`）に変換し、`java` がそれを JVM に読み込ませ、**その中の `main` メソッドを呼ぶ**ところから実行が始まります。

```
Hello.java  --(javac)-->  Hello.class  --(java Hello)-->  JVM が
Hello.main(...) を呼ぶ
```

ここで押さえておきたいのは、**`main` は「起動の入口」として使われるだけの、ふつうの静的メソッドだ**ということです。Java の言語仕様には「`main` という名前のメソッドは特別扱いする」という規則がありません。`main` は予約語ですらなく、ただの識別子です。

だから、次の2つはまったく別の話になります。

- **メソッド宣言として正しいか** …… これを判定するのは `javac`。名前が `main` でも、他のメソッドとまったく同じ基準で見られる
- **起動の入口として使えるか** …… これを判定するのは `java`（起動ツール）。**コンパイルがすべて終わったあと**に、条件に合う `main` を探す

この2段構えが分かっていれば、この細目の問題の大半は仕分けだけで解けます。

1-2 起動される main の4条件

Java 言語仕様 SE17 の §12.1.4 は、起動時に呼ばれるメソッドが満たすべき条件を次のように定めています。

#	条件	満たさないと
1	`public` である	「メイン・メソッドが**見つかりません**」
2	`static` である	「メイン・メソッドが**staticではありません**」
3	戻り値の型が `void` である	「メイン・メソッドは**void型の値を返す必要があります**」
4	仮引数が `String` の配列1つである	「メイン・メソッドが**見つかりません**」

仕様は `public static void main(String[] args)` と `public static void main(String... args)` の**どちらも受け付ける**と明記しています。可変長引数の `String...` は、引数の形としては `String[]` と同じものだからです。

この4つ以外は、起動の判定に関係しません。具体的には次のものはすべて自由です。

自由に変えてよいもの	例
修飾子どうしの並び順	`static public void main(String[] args)` でよい
引数の名前	`args` である必要はまったくない。`argv` でも `signals` でもよい
`String[]` と `String...` の別	どちらでもよい

自由に変えてよいもの	例
`throws` 節	`public static void main(String[] args) throws Exception` は起動できる
`final` 修飾子	`public static final void main(String[] args)` も条件を満たす
メソッド本体の書き方	`var` によるローカル変数宣言なども当然使える

`throws` 節が付いていても起動できるのは、`throws` **が「例外を投げる可能性がある」という宣言にすぎず、それ自体が例外を起こすわけではない**からです。宣言と発生を混ぜて読まないでください。

1-3 「コンパイルエラー」と「コンパイルは通るが起動できない」

ここがこの章の中心です。`main` を壊す方向は何通りもありますが、**着地点は2つに分かれます。**

壊し方	コンパイル	起動	実測されるメッセージ
`static` を落とす	**通る**	失敗	`エラー: メイン・メソッドがクラスXのstaticではありません。`
戻り値を `int` / `String` にする	**通る**	失敗	`エラー: メイン・メソッドはクラスXのvoid型の値を返す必要があります。`

壊し方	コンパイル	起動	実測されるメッセージ
`private` / 修飾子なしにする	**通る**	失敗	`エラー: メイン・メソッドがクラスXで見つかりません。`
引数を `String[]` 以外にする（`main(int)` だけ）	**通る**	失敗	`エラー: メイン・メソッドがクラスXで見つかりません。`
修飾子を戻り値の型より後ろに置く（`public void static main`）	**エラー**	—	`<identifier>がありません` ほか

上4つは全部「コンパイルは成功する」側です。メソッド宣言としてはどれも文法どおりなので、`javac` には止める理由がありません。止まるのは `java` が「条件に合う `main` を探す」段階です。

最後の1つだけが違います。Java のメソッド宣言は「修飾子 → 戻り値の型 → メソッド名 → 引数」の順と決まっていて、**修飾子どうしの順序は自由でも、修飾子は全部、戻り値の型より前**でなければなりません。`public void static main` は `void` のあとに `static` が来ているので、コンパイラはこれをメソッド宣言として読み取れません。**`static` が「書かれているのに効かない」のではなく、そもそも宣言として成立していない**わけです。

実測したエラーは次のとおりです。`static` の欠落なら実行時エラー、置き場所の誤りならコンパイルエラー —— この対比がそのまま出題されます。

```
Misplaced.java:2: エラー: <identifier>がありません
    public void static main(String[] args) {
                ^
Misplaced.java:2: エラー: '('がありません
Misplaced.java:2: エラー: 無効なメソッド宣言です。戻り値の型が必要です。
エラー3個
```

なぜ `public` でないと「見つかりません」なのか

`static` 欠落や戻り値の不一致には専用のメッセージが出るのに、`private` にすると「見つかりません」になります。これは起動ツールが**「`public` な `main` を探す」という探し方**をしているためです。`private` な `main` はそもそも探索の網に掛からないので、「条件に合うものが1つも無かった」という結末になります。

同じ理由で、`main(int level)` **しか**持たないクラスも「見つかりません」になります。引数の形が違うものは、名前が `main` でも候補になりません。

1-4 main が特別なのは「起動の瞬間」だけ

条件に合う `main` が複数の意味を持つように見える場面がありますが、規則は単純です。

起動対象になるのは、`java` コマンドに渡したクラスの `main` だけです。それ以外の `main` は、ただの静的メソッドとして扱われます。

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

public class Second {
    public static void main(String[] args) {
        System.out.println("second");
        Helper.main(args);
    }
}
```

`java Second` の出力（実測）：

```
second
helper
```

`main` を持つクラスがファイル内にいくつあっても構いません。**どれが起動されるかは、起動時に指定したクラスで決まります。** 上のコードは `java Helper` と起動すれば `helper` の1行だけになります。

同じことがオーバーロードにも言えます。

```
public class Dispatch {
    public static void main(String[] args) {
        System.out.println("array " + args.length);
        main(7);
    }

    public static void main(int depth) {
        System.out.println("depth " + depth);
    }
}
```

`java Dispatch` の出力（実測）：

```
array 0
depth 7
```

`main(int)` は起動対象の候補になりませんが、**プログラムが動き出したあとは普通のメソッド呼び出しの規則で選ばれます。**ここで「同名のメソッドが2つあるから起動対象が決まらずエラー」と書かれた選択肢が出ますが、そうはなりません。起動対象の選定は**引数の型で一意に決まる**からです。

1-5 args に何が入るか

コマンドライン引数は、**クラス名より後ろに並べた語だけ**が入ります。クラス名自体は入りません。

- 引数を1つも渡さなかった場合 …… **長さ 0 の配列**が渡される。`null` にはならない

- 添字は 0 から始まる。`args[0]` が最初の引数
- 渡した個数ぴったりの長さで作られる。足りない分が `null` で埋まる仕組みは**無い**

```
public class Vararg {
    static public void main(String... items) throws Exception {
        System.out.println(items.length);
        System.out.println(items == null);
    }
}
```

`java Vararg` の出力（実測）：

```
0
false
```

このクラスは、修飾子が `static public` の順、引数名が `items`、型が `String...`、さらに `throws Exception` 付き —— 4条件から外れる要素を一切含んでいないので、問題なく起動します。**「どれかが `args` と違うから起動できない」という選択肢は、いつも誤りです。**

`args` の要素数より大きい添字を読むと `ArrayIndexOutOfBoundsException` になりますが、これはコンパイルでは検出できません（実行してみないと個数が分からないため）。第2章の 2-10 と第7章で改めて扱います。

1-6 System.out.println のオーバーロード選択

この細目の出力問題は、ほぼ `System.out.println` / `System.out.print` の**どのオーバーロードが選ばれるか**で決まります。`System.out` は `PrintStream` 型で、`println` は引数の型ごとに別々のメソッドを持っています。

渡すもの	選ばれるもの	出力
`char`	`println(char)`	**文字**として出る
`int`（`char` どうし／`char + int` の計算結果を含む）	`println(int)`	**数値**として出る
`char[]`	`println(char[])`	**要素を文字として並べて**出る
`char[]` 以外の配列	`println(Object)`	`[I@8bcc55f` のような**型と参照の表記**
`String`	`println(String)`	そのまま

`char[]` だけが専用のオーバーロードを持っているのが要点です。他の型の配列（`int[]`・`String[]` など）を渡すと `println(Object)` が選ばれ、配列の既定の文字列表現が出ます。

さらに、**同じ `char[]` でも「直接渡す」のと「文字列連結する」のとで結果が変わります。**

```
public class Chars {
    public static void main(String[] args) {
        char[] word = {'s', 'k', 'y'};
        System.out.print(word);
        System.out.println("?");
        System.out.println("" + word);
        char c = 'M';
        System.out.println(c);
        System.out.println(c + 1);
        System.out.println("" + c + 1);
        int[] nums = {1, 2};
        System.out.println(nums);
    }
}
```

出力（実測）：

```
sky?
[C@7344699f
M
78
M1
[I@c387f44
```

1行ずつ読み解きます。

- `print(word)` は `print(char[])` が選ばれ、`sky` を改行なしで出力。続く `println("?")` が `?` を足して改行するので、1行目は `sky?`
- `"" + word` は**文字列連結**なので、`char[]` は `Object` として扱われ `toString()` の結果 `[C@…` になる。`[C` は「`char` の配列」を表す型の表記で、`@` の後ろは実行のたびに変わる
- `println(c)` は `println(char)` → 文字 `M`
- `println(c + 1)` は `char + int` の加算。**`char` は数値として扱われ、結果は `int`** になるので `println(int)` が選ばれ、`'M'` のコード値 77 に 1 を足した `78`
- `"" + c + 1` は左から順に連結されるので `"M"` に `"1"` が付いて `M1`
- `println(nums)` は `int[]` なので `println(Object)` → `[I@…`

「`+` が計算なのか連結なのか」は、両辺のどちらかが `String` かどうかだけで決まります。片方が `String` なら連結、そうでなければ数値の計算です。

混同しやすい対の概念

混同しやすい組	見分け方
`static` を落とした main ⇔ 修飾子の位置を誤った main	前者はコンパイルが**通り**、起動で失敗。後者はコンパイルで**止まる**。「`static` が無い」と「`static` の置き場所が違う」は着地点が別
`public` でない main ⇔ 戻り値が `void` でない main	どちらもコンパイルは通るが、メッセージが違う。`public` でない＝「**見つかりません**」／`void` でない＝「**void型の値を返す必要があります**」

混同しやすい組	見分け方
起動対象の main の選定 ⇔ 通常のオーバーロード解決	起動対象は `String[]`（＝`String...`）を取るものに**一意に決まる**。`main(int)` は起動されないが、プログラム内からは普通に呼べる
`println(char[])` ⇔ `println(Object)`	`char` の配列**だけ**が専用オーバーロードを持つ。`int[]` や `String[]` を渡すと `[I@…` 形式
`char[]` **を直接渡す** ⇔ `char[]` **を文字列連結する**	直接渡せば `sky`、`"" +` で連結すれば `[C@…` 。**渡し方**でオーバーロードが変わる
`println(char)` ⇔ `println(int)`	`c` は文字、`c + 1` は数値。`char` が式の中で `int` に昇格した時点で、選ばれるオーバーロードが変わる
`String[] args` **が長さ0** ⇔ `args` **が null**	引数を渡さなくても**長さ 0 の配列**が渡される。`null` になることはない
`main` **は予約語** ⇔ `main` **はただの識別子**	予約語ではない。だから `public static int main(...)` という宣言も**コンパイルは通る**

コードで確かめる

① static を落とすと、どこで止まるか

```
public class NoStatic {
    public void main(String[] args) {
        System.out.println("hello");
    }
}
```

```
$ javac NoStatic.java          （エラーも警告も出ずに成功）
$ java NoStatic
エラー: メイン・メソッドがクラスNoStaticのstaticではありません。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)
（終了コード 1・"hello" は出力されない）
```

② 戻り値の型を変えると

```
public class Returns {
    public static int main(String[] args) {
        System.out.println("hello");
        return 3;
    }
}
```

```
$ javac Returns.java            （成功）
$ java Returns
```

エラー: メイン・メソッドはクラスReturnsのvoid型の値を返す必要があります。

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)
```

`return 3;` を書いても、それが終了コードになる仕組みはありません。終了コードを指定するのは `System.exit(...)` です。

③ private にすると

```
public class Hidden {
    private static void main(String[] args) {
        System.out.println("hello");
    }
}
```

```
$ javac Hidden.java             （成功）
$ java Hidden
```

エラー: メイン・メソッドがクラスHidden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

「アクセス権が無いという例外」は発生しません。**探索に掛からなかった**という扱いです。

④ 修飾子の位置を誤ると

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

```
$ javac Misplaced.java
Misplaced.java:2: エラー: <identifier>がありません
Misplaced.java:2: エラー: '('がありません
Misplaced.java:2: エラー: 無効なメソッド宣言です。戻り値の型が必要です。
エラー3個
```

ここだけコンパイルで止まります。①〜③との違いを、必ずセットで覚えてください。

章の試験ポイント

1. **【型②】`main` の壊れ方は「コンパイルエラー」と「起動失敗」に二分される。**`static` 欠落・`public` でない・戻り値が `void` でない・引数が `String[]` でない ―― これらは**全部コンパイルが通ります。**コンパイルで

止まるのは、修飾子を戻り値の型より後ろに置くなど、**メソッド宣言の文法自体が壊れている**場合だけです。

2. **【型⑦】起動条件は4つだけ。** `public` / `static` / 戻り値 `void` / 引数が `String` の配列。**引数名・修飾子の順序・ `throws` 節・ `String...` か `String[]` かは、いずれも判定に関係しません。**「引数名が `args` でないから起動できない」型の選択肢は常に誤りです。
3. **【型①】 `public` でない場合と `static` でない場合で、メッセージが違う。** `public` でない＝「見つかりません」／ `static` でない＝「staticではありません」／ `void` でない＝「void型の値を返す必要があります」。どのメッセージが出るかを問う設問があります。
4. **【型①】 `main` は起動の瞬間だけ特別。** 同じファイルに `main` を持つクラスがいくつあってもコンパイルは通り、起動されるのは指定した1つだけ。他の `main` は通常の静的メソッドとして呼び出せます。
5. **【型①】 `println` は `char[]` にだけ専用のオーバーロードを持つ。** `char[]` を直接渡せば文字が並び、`int[]` や `String[]` を渡せば `[I@…` / `[Ljava.lang.String;@…` になります。
6. **【型⑤】 `char` は式の中で `int` に昇格する。** `println(c)` は文字、`println(c + 1)` は数値。`"" + c + 1` は連結なので `M1` 。3つを区別してください。
7. **【型⑦】 `args` は長さ 0 になることはあっても `null` にはならない。**また要素数は渡した引数の個数ぴったりで、余りが `null` で埋まることもありません。
8. **【型②】 `main` は予約語ではない。**だから `int` や `String` を返す `main` 、 `private` な `main` は**宣言できます。**宣言できることと起動できることを混ぜないでください。

サンプルはここまでです

続き(第2章〜巻末)は、Java Silver SE17 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 Silver SE 17(1Z0-825-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 17 Programmer I(1Z0-825-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