Chapter2

Monday, October 6, 2025 3:19 PM

Compile-Time Constants in Java

What Are They?

- **Compile-time constants** refer to **integer literals** that are known at compile time.
- These constants can be assigned to smaller data types **only if** the value fits within the target type's range.

Type Mismatch and Casting

- If you assign a **larger type constant** to a **smaller type variable**, you'll get a **compile-time error** unless you explicitly cast it.

```
int a = 1.0;        // ❌ Compile-time error: 1.0 is a double
int a = (int) 1.0;  // ☑ Correct: explicit cast from double to int
```

Compile-Time Constant Narrowing

- Java allows **narrowing conversions** for **integer literals** if the value fits within the target type's range.

```
short a = 1;        // ☑ Compiles: 1 fits in short range (-32,768 to 32,767)
short b = 128;      // ☑ Compiles: 128 fits in short range
short c = 32768;    // ❌ Error: 32768 exceeds short range
```

- But if the value **exceeds the range**, even though it's a literal, Java will throw a **compile-time error**.

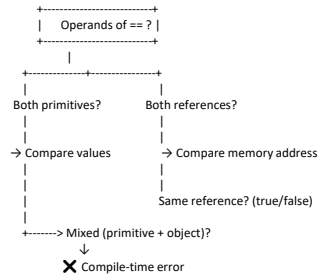
```
short a = 128;      // ☑ OK: 128 is within range
short a = 128000;   // ❌ Error: value too large for short

int i = 1000;
byte b = i; // ❌ compile-time error
```

Let me know if you'd like a visual diagram or a cheat sheet version too!

The big rule

An explicit cast in Java never throws an "overflow" error for primitives.
If the value doesn't fit, Java will **wrap** (for integer→integer casts), or **truncate/clamp** (for float/double→integer casts). No exception is raised.

Equality operator. ("==")



Tip

These don't compile because == and != can't mix primitives and objects.
String is an object type, so Java rejects any attempt to compare it directly with a number or boolean.

This table is very tricky. Be aware of the optimizations that are made by java, on "Conditional Operators"

TABLE 2.11 Conditional operators

Operator	Example	Description
Conditional AND	`a && b`	The value is `true` only if both values are `true`. If the left side is `false`, then the right side will not be evaluated.
Conditional OR	`c || d`	The value is `true` if at least one of the values is `true`. If the left side is `true`, then the right side will not be evaluated.

Java's Rule: Binary Numeric Promotion

In Java, when you perform arithmetic operations (`+`, `-`, `*`, `/`, etc.) on **byte**, **short**, or **char**, the operands are **promoted to** `int` before the operation is executed.

So:

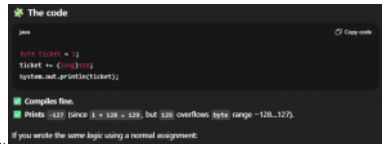
```java
short x = 1, y = 2;
short z = x + y;  // ❌ Error: x + y is promoted to int, can't assign to shor
```

Category / Operators	All operands evaluated?	Order	Short-circuit?	Key notes & "gotchas"
Logical (short-circuit) &&, `		`	**Sometimes**	Left→right
Logical (non-short-circuit) &, `	, ^` (with boolean)	**Yes**	Left→right	**No**
Bitwise &, `	, ^` (with integral types)	**Yes**	Left→right	N/A
Ternary cond ? a : b	**Only one of a/b**	Evaluate cond first	**Conditional**	cond evaluated; if true only a evaluated, else only b. Powerful for avoiding NPE work on the "other" branch. Type of result uses conditional typing rules (numeric promotions / LUB for references / target typing).
Relational <, <=, >, >=	**Yes**	Left→right	No	Numeric promotion applies. Not defined for boolean or reference types (except compareTo, which is a method, not operator).
Equality ==, !=	**Yes**	Left→right	No	Primitives: compares **values** (with promotions). References: compares **references** (same object?). Never mixes primitive with reference (compile-time error). For strings, use .equals() for content.
Arithmetic + - * / %	**Yes**	Left→right	No	**Binary numeric promotion**: byte/short/char → int; then widen to the wider operand among int→long→float→double. / with integers = truncating division. % sign follows dividend. Division by zero (integral) → ArithmeticException.
Unary +, -, ~, !	N/A	Operand evaluated	N/A	~ flips bits; ! negates boolean; unary - negates number (beware Integer.MIN_VALUE negation stays MIN due to overflow).
Inc/Dec ++, --	N/A	Operand evaluated	N/A	**Pre** changes then yields value; **post** yields old value then changes. Side effects happen exactly once.
Shifts <<, >>, >>>	**Yes**	Left→right	No	Shift distance masked (& 0x1F for int, & 0x3F for long). Arithmetic vs logical right shift: >> sign-extends, >>> zero-fills.
String concat + (if either side is String)	**Yes**	Left→right	No	Converts RHS to String (via String.valueOf), effectively builds left→right (like new StringBuilder().append(...)). null + "x" → "nullx". Operator precedence: + is left-assoc; parentheses decide order.
Cast (T)expr	N/A	Operand evaluated	N/A	If reference cast fails → ClassCastException. Primitive casts never throw; they may wrap/truncate.
instanceof	**Yes**	Left→right	No	Evaluates left; if left is null, result is false (no exception). Right is a type token; not "evaluated." Pattern matching obj instanceof Type t binds t on true.
Assignment =	RHS **Yes**; LHS resolved	LHS ref→RHS→store	N/A	LHS (variable/field/array element) chosen first, then RHS evaluated, then assignment. Type conversion may happen (widening ok; narrowing requires cast, except int-constant-fits rule).
Compound assignment +=, -=, *=, ...	RHS **Yes**; LHS resolved	LHS ref→(read LHS)→RHS→op→cast→store	N/A	**LHS evaluated once** (important for arr[i++] += x). Implicit cast to LHS type after the op (may **silently** narrow and wrap).

EXAM

My answers:
1.A,D ❌
2.A,D 👍

1. A,D,G
   !! In java doesn't exist boolean casting!!!



Assignment vs. Compound Assignment

Example	Behavior	Hidden Rule
x = x + y	Normal binary promotion rules apply	Needs explicit cast if narrowing (byte, short, char)

EXAM

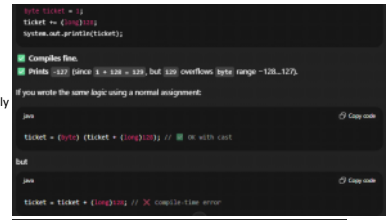
My answers:
1.A,D
2.A,D
3.B,C,D,F
4.B
5.F,E,B,
6.F
7.D
8.A
9.D,E,A
10.A,B,C,D
11.D
12.D
13.F.
14.G,E,B
15.D
16.
17.C,F
18.C
19.A,F
20.D,E

1. A,D,G
   !! In java doesn't exist boolean casting!!!
   G is correct, because syntactically you could cast a type to his type, even is redundant and have not practically in real life. EX: (boolean a= true; boolean b = (boolean)a; =>> this is correct and would compile)
   !!!STUPID answer!!
2. A,D,B (I wasn't pay attention to B response)

Java's Widening Rules (Primitive Types)

From Type	Can Be Widened To
byte	short , int , long , float , double
short	int , long , float , double
char	int , long , float , double
int	long , float , double
long	float , double
float	double

- Widening is **safe** and **automatic** — no cast needed.
- Opposite of narrowing, which requires **explicit casting** and may lose data.



3. B,C,D,F

3. What change, when applied independently, would allow the fo
snippet to compile? (Choose all that apply.)

```
3: long ear = 10;
4: int hearing = 2 * ear;
```

A. No change; it compiles as is.
B. Cast `ear` on line 4 to `int`.
C. Change the data type of `ear` on line 3 to `short`.
D. Cast `2 * ear` on line 4 to `int`.
E. Change the data type of `hearing` on line 4 to `short`.
F. Change the data type of `hearing` on line 4 to `long`.

Attention.! Is about assignment , java promote that constant 2 to a long and knows how to deal with expression.

4.

5. !!Pay Attention!! , I thought that I need to rank from left to right to the biggest priority to the lowest.
   BUT was reversed. From the lowest to the biggest.

10. But it says:"

10. Which is not an output of the following code snippet?

```
short height = 1, weight = 3;
short zebra = (byte) weight * (byte) height;
double ox = 1 + height * 2 + weight;
long giraffe = 1 + 9 % height + 1;
System.out.println(zebra);
System.out.println(ox);
System.out.println(giraffe);
```

A. 2
B. 3
C. 6
D. 6.0
E. The code does not compile.

And a,b,c,d will not be the output because the output will be E. SO they miss ,!!!!!.

11.

16.

16. How many lines of the following code contain compiler errors?

```
int note = 1 * 2 + (long)3;
short melody = (byte)(double)(note *= 2);
double song = melody;
float symphony = (float)((song == 1_000f) ? song * 2L : song);
```

Blue Line will compile, because firstly will be computed the paranthese:"(note *= 2)"
Then from right to left will be processed casting operation.

A. 0
B. 1
C. 2
D. 3
E. 4

19.B,F

19. What is the result of executing the following code snippet? (Choose all that apply.)

```
3: int start = 7;
4: int end = 4;
5: end += ++start;
6: start = (byte)(byte.MAX_VALUE + 1);
```

will result in a negative number, making option B the correct answer. Even if you didn't know the maximum value of `byte`, you should have known the code compiles and runs and looked for the answer for `start` with a negative number.

A. start is 0.
B. start is -128.
C. start is 127.
D. end is 8.
E. end is 11.
F. end is 12.
G. The code does not compile.
H. The code compiles but throws an exception at runtime.

Byte have 8bits length and MAX_VALUE would be :"1111 1111" + 1 => (0001 0000 0000) converted to byte

20.

20. Which of the following statements about unary operators are true? (Choose all that apply.)
   A. Unary operators are always executed before any surrounding numeric binary or ternary operators.
   B. The `-` operator can be used to flip a `boolean` value.
   C. The pre-increment operator (`++`) returns the value of the variable before the increment is applied.
   D. The post-decrement operator (`--`) returns the value of the variable before the decrement is applied.
   E. The `!` operator cannot be used on numeric values.
   F. None of the above.

```
int g = 2;
System.out.println(((g > 2? 4 : 1) > ++g));
```

"Surrounding" means that in the same expression/tree. And yes, unary will be executed firstly. Result will be "false"

Assignment

Example	Behavior	Hidden Rule
x = x + y	Normal binary promotion rules apply	Needs explicit cast if narrowing (byte, short, char)
x += y	Implicit cast back to type of x	Equivalent to x = (T)(x + y)
x *= y	Same rule	Works even if y is double and x is int → result cast to int

🔍 **Pro tip:**
+=, -=, *=, /=, %= perform an *implicit narrowing conversion*.

2️⃣ Numeric Promotion

Binary numeric promotion (used in +, -, *, /, %, &, |, ^, etc.):
1. If either operand is double → both to double
2. Else if either is float → both to float
3. Else if either is long → both to long
4. Else → both to int

That's why:
byte b = 2, c = 3;
// b + c → int
// b = b + c; // ❌ compile error
b += c; // ☑ compiles (implicit cast)

3️⃣ Unary Operators

Operator	Meaning	Special Notes
+	Unary plus (no effect)	Rarely used
-	Negation	-Integer.MIN_VALUE == Integer.MIN_VALUE 😱 overflow
~	Bitwise NOT	Flips every bit; ~0 == -1
!	Logical NOT	Works only on booleans
++, --	Increment/decrement	x++ → use old value, then add 1; ++x → add 1, then use new value

🤔 **Trick:**
++x + ++x is undefined in some languages, but Java defines exact order (left→right).

4️⃣ Relational & Equality Operators

Operator	Works on	Notes
<, <=, >, >=	Numbers + char	Not valid on boolean or objects
==, !=	All types	For objects, compares **references**, not content
equals()	Any object	Override for content equality
compareTo()	Comparable types	For ordering

Trick:
System.out.println(10 == 10.0); // true → int → double promotion
System.out.println('A' == 65); // true → char → int promotion
System.out.println("hi" == "hi"); // true (interned literals)
System.out.println(new String("hi") == "hi"); // false

5️⃣ Logical vs Bitwise

Operator	Type	Description
&, `	, ^`	both numeric and boolean
&&, `	`	

Example:
int x = 0;
boolean res = (x != 0) && (10 / x > 1); // Safe ☑ (short-circuit skips RHS)
boolean res2 = (x != 0) & (10 / x > 1); // ✳ ArithmeticException (div by zero)

6️⃣ Ternary Operator ?:

Syntax:
condition ? expr1 : expr2
- Only **one** branch executes at runtime.
- Both branches must be **type-compatible** after promotion.

Example:
var res = (true) ? 5 : 5.5; // res is double (numeric promotion)
var str = (true) ? "hi" : null; // res is String
⚠ Common trap:
System.out.println(true ? null : "A"); // ❌ compile error — null has no type

7️⃣ Shift Operators

Operator	Description
<<	left shift (multiply by 2^n)
>>	signed right shift (divide by 2^n, keeps sign)
>>>	unsigned right shift (fills with zeros)

Key rule:
Shift count masked:
- For int: shift & 0x1F (mod 32)
- For long: shift & 0x3F (mod 64)

Example:
int x = 1;
System.out.println(x << 32); // still 1 (32 mod 32 == 0)

8️⃣ String Concatenation (+)

- If **either** operand is a String, the result is a String.
- Evaluation order: left → right.
- null converts to "null".

Example:
System.out.println(1 + 2 + "3"); // "33" (left adds first)
System.out.println("1" + 2 + 3); // "123"

9️⃣ Operator Precedence

(simplified)

High → low (left associative unless marked):

Precedence	Operators	Notes
1	++, --, !, ~, casts, unary + -	unary first
2	*, /, %	multiplication
3	+, -	addition / subtraction
4	<<, >>, >>>	shifts
5	<, <=, >, >=, instanceof	relational
6	==, !=	equality
7	&, ^, `	`
8	&&, `	
9	?:	ternary (right-assoc)
10	=, +=, etc.	assignment (right-assoc)

Tip:
Parentheses are your friend! (a + b) * c always clearer.

10 Tricky Compound Examples

Mixed types

```
int x = 5;
x *= 3.5; // implicitly cast to int: x = (int)(x * 3.5) → 17
```

Overflow wrap-around

```
byte b = 127;
b += 1;          // -128 (overflow)
System.out.println(b);
```

Boolean traps

```
boolean a = false;
boolean b = true;
System.out.println(a = b);   // prints true (assignment, not equality)
System.out.println(a == b);  // prints true (after assignment)
```

Bitwise on booleans

```
boolean a = true, b = false;
System.out.println(a & b);   // false
System.out.println(a | b);   // true
System.out.println(a ^ b);   // true (XOR)
```

11 Instanceof patterns (Java 16+)

```
Object o = "Hello";
if (o instanceof String s) { // pattern variable
    System.out.println(s.toUpperCase());
}
```

☑ You can safely use s inside the if block — no cast needed.
❌ Not allowed across control flow jumps (e.g., after else).

12 Null and instanceof

Before Java 14:

```
null instanceof Anything // false
```

After pattern matching:

```
case null -> ...  // explicit match if needed in switch
```

13 Short-circuit in ternaries

Even though both expressions must be type-compatible, only **one** is *evaluated*:

```
boolean flag = false;
System.out.println(flag ? 10/0 : 20); // ☑ prints 20 (no exception)
```

Final Pro Tips

☑ **Compound operators (+=)** always include implicit casts.
☑ **&& and ||** short-circuit; & and | don't.
☑ **Overflow** never throws an exception (use Math.addExact() to detect it).
☑ **Floating-point** comparisons: NaN != NaN, 0.0 == -0.0 is true.
☑ **instanceof** → false for null.
☑ **switch** → uses ==, not .equals().

Chapter3

Tuesday, October 7, 2025 3:35 PM

Switch

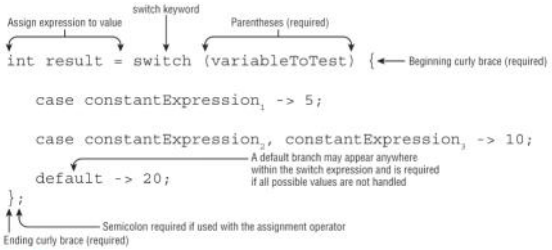


FIGURE 3.5 A `switch` expression

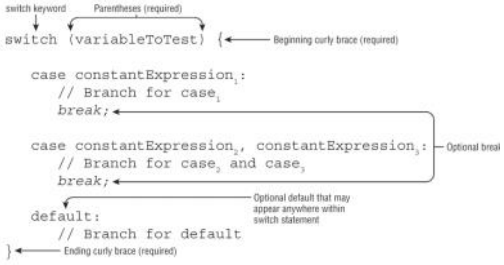


FIGURE 3.4 A `switch` statement

```
final int getCookies() { return 4; }
void feedAnimals() {
   final int bananas = 1;
   int apples = 2;
   int numberOfAnimals = 3;
   final int cookies = getCookies();
   switch (numberOfAnimals) {
      case bananas:
      case apples:       // DOES NOT COMPILE
      case getCookies(): // DOES NOT COMPILE
      case cookies :     // DOES NOT COMPILE
      case 3 * 5 :
   } }
```

When writing `switch` expressions, it may be a good idea to add a `default` branch, even if you cover all possible values. This means that if someone modifies the enum with a new value, your code will still compile.

```
Number variableToTest = ...

int result = switch (variableToTest) {
   case Integer x when x > 0 && x <=20 -> 3;

   case Integer x -> 5;

   case Number n -> 10;
};
```

Type matches switch variable reference type so default clause is not needed

FIGURE 3.7 Pattern matching with `switch`

```
int result = switch (variableToTest) {
   case constantExpression1 -> 5;
   case constantExpression2, constantExpression3 -> {
      yield 10;
   }
   ...
};
```

case expression — case block — A yield is required within a case block unless an exception is thrown — Curly braces required for case blocks

FIGURE 3.6 A `switch` expression with a `case` block and `yield` statement

WATCH SEMICOLONS IN *SWITCH* EXPRESSIONS

When writing a `case` expression, a semicolon is required, but when writing a `case` block, it is prohibited.

```
int fish = 1;
var name = switch (fish) {
   case 1  -> "Goldfish"          // DOES NOT COMPILE (missing s
   case 2  -> { yield "Trout"; };  // DOES NOT COMPILE (extra sem
   default -> "Shark";
}  // DOES NOT COMPILE (missing semicolon)
```

A bit confusing, right? It's just one of those things you have to train yourself to spot on the exam.

Properties:

1. Must to be @Exaustive.
2. Yield is used for return a value from a case block expression. If we will use return, then he whole method will return the value.

WATCH SEMICOLONS IN *SWITCH* EXPRESSIONS

When writing a `case` expression, a semicolon is required, but when writing a `case` block, it is prohibited.

```
int fish = 1;
var name = switch (fish) {
   case 1  -> "Goldfish"          // DOES NOT COMPILE (missing s
   case 2  -> { yield "Trout"; };  // DOES NOT COMPILE (extra sem
   default -> "Shark";
}  // DOES NOT COMPILE (missing semicolon)
```

A bit confusing, right? It's just one of those things you have to train yourself to spot on the exam.

3. Unreachable code will throw an compile time error.
4. Have a null case in Java21. To avoid boilerplate code.

New to Java 21, `switch` now supports `case null` clause when working with object types, allowing us to rewrite our previous example as the following:

```
String fish = null;
System.out.print(switch (fish) {
   case "ClownFish" -> "Hello!";
   case "BlueTang"  -> "Hello again!";
   case null        -> "What type of fish are you?";
   default          -> "Goodbye";
});
```

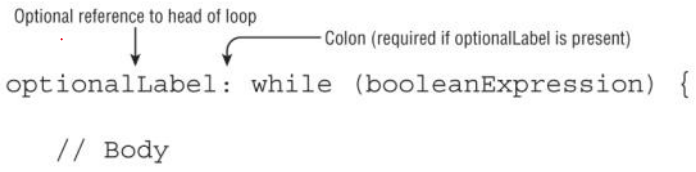
That's a lot less boilerplate code, now that we don't have to handle `null` separately.

5. @Case null is allowed just in pattern matching switch, and the order in case null matter. Needs to be the last and after a default case to ensure will cover all the posibilities.

FOR Loops:

4. Using Incompatible Data Types in the Initialization Block

```
int x = 0;
for (long y = 0, int z = 4; x < 5; x++)  // DOES NOT COMPILE
   System.out.print(y + " ");
```



Summary

Concept	What it affects	Can you change the reference?	Can you change the object?
`String`	object immutability	✅ yes	❌ no
`final String`	reference finality +	❌ no	❌ no

```
// Body

// Somewhere in the loop
break optionalLabel;
}
```

break keyword — Semicolon (required)

FIGURE 3.12 The structure of a `break` statement

`String`	object immutability	✅ yes	❌ no
`final String`	reference finality + object immutability	❌ no	❌ no
`final StringBuilder`	reference finality only	❌ no	✅ yes

EXAM

1.F.
2.A,B,C
3.B.
4.A,D,F
5.C. -> F
6.C. -> E
7.B,D
8.G.
9.E,A
10.E
11.D (Cred ca return typul trebuie sa fie exact cu cel asteptat.
12.C
13.F
14.G
15.B,D,F
16.F.
17.d,B,A
18.e,b
19.F

19. E. The variable `snake` is declared within the body of the `do/while` statement, so it is out of scope on line 7. For this reason, option E is the correct answer. If `snake` were declared before line 3 with a value of `1`, then the output would have been `1 2 3 4 5 -5.0`, and option G would have been the correct answer.

```
2: double iguana = 0;
3: do {
4:    int snake = 1;
5:    System.out.print(snake++ + " ");
6:    iguana--;
7: } while (snake <= 5);
8: System.out.println(iguana);
```

A. `1 2 3 4 -4.0`
B. `1 2 3 4 -5.0`
C. `1 2 3 4 5 -4.0`
D. `0 1 2 3 4 5 -5.0`
E. The code does not compile.
F. The code compiles but produces an infinite loop at runtime.
G. None of the above.

20. A,E

```
4:  int height = 1;
5:  L1: while (height++ <10) {
6:     long humidity = 12;
7:     L2: do {
8:        if (humidity-- % 12 == 0) ;
9:        int temperature = 30;
10:       L3: for ( ; ; ) {
11:          temperature++;
12:          if (temperature>50) ;
13:       }
14:    } while (humidity> 4);
15: }
```

A. `break L2` on line 8; `continue L2` on line 12
B. `continue` on line 8; `continue` on line 12
C. `break L3` on line 8; `break L1` on line 12
D. `continue L2` on line 8; `continue L3` on line 12
E. `continue L2` on line 8; `continue L2` on line 12
F. None of the above, as the code contains a compiler error

Option C is incorrect because it contains a compiler error. The label `L3` is not visible outside its loop. Option D is incor-

21.D

21. A minimum of how many lines need to be corrected before the following

21. A minimum of how many lines need to be corrected before the following method will compile?

```
21: void findZookeeper(Integer id) {
22:    System.out.print(switch (id) {
23:       case 10 -> {"Jane";}
24:       case 20 -> {yield "Lisa";};
25:       case 30 -> "Kelly";
26:       case 30 -> "Sarah";
27:       default -> "Unassigned";
28:    });
29: }
```

A. Zero
B. One
C. Two
D. Three
E. Four
F. Five

21. D. Line 23 does not compile because it is missing a `yield` statement. Line 24 does not compile because it contains an extra semicolon at the end. Finally, lines 25 and 26 do not compile because they use the same `case` value. At least one of them would need to be changed for the code to compile. Since three lines need to be corrected, option D is correct.

22.C E

22. What is the output of the following code snippet?

```
2: var tailFeathers = 3;
3: final var one = 1;
4: switch (tailFeathers) {
5:    case one: System.out.print(3 + " ");
6:    default: case 3: System.out.print(5 + " ");
7: }
8: while (tailFeathers> 1) {
9:    System.out.print(--tailFeathers + " "); }
```

A. `3`
B. `5 1`
C. `5 2`
D. `3 5 1`
E. `5 2 1`
F. The code will not compile because of lines 3–5.
G. The code will not compile because of line 6.

23.D F

23. What is the output of the following code snippet?

```
15: int penguin = 50, turtle = 75;
16: boolean older = penguin>= turtle;
17: if (older = true) System.out.println("Success");
18: else System.out.println("Failure");
19: else if (penguin != 50) System.out.println("Other");
```

A. `Success`
B. `Failure`
C. `Other`
D. The code will not compile because of line 17.
E. The code compiles but throws an exception at runtime.
F. None of the above.

24.B

24. What is the output of the following code snippet?

```
22: String zooStatus = "Closed";
23: int visitors = switch (zooStatus) {
24:    case String s when s.equals("Open") -> 10;
25:    case Object s when s != null && !s.equals("") -> 20;
26:    case null -> {yield 30;}
27:    default -> 40;
28: };
29: System.out.print(visitors);
```

A. `10`
B. `20`
C. `30`
D. `40`
E. Exactly one line does not compile.
F. Exactly two lines do not compile.

24. B. Since this is a pattern matching `switch` statement, the `case` branches are evaluated in the order in which they appear. In particular, each branch does not dominate the ones after it, so the code compiles without issue. If either of the `when` clauses were removed from their accompanying `case` clause, then the code would not compile. The first branch is skipped because `Closed` does not match `Open`. The second one matches, resulting in `20` being printed at runtime and making option B correct.

25.D

25. What is the output of the following code snippet?

```
6:  String instrument = "violin";
7:  final String CELLO = "cello";
8:  String viola = "viola";
9:  int p = -1;
10: switch (instrument) {
11:    case "bass" : break;
12:    case CELLO : p++;
13:    default: p++;
14:    case "VIOLIN": p++;
15:    case "viola" : ++p; break;
16: }
17: System.out.print(p);
```

marked `final`. Since `"violin"` and `"VIOLIN"` are not an exact match, the `default` branch of the `switch` statement is executed at runtime. This execution path increments `p` a total of three times, bringing the final value of `p` to `2` and making option D the correct answer.

!!!Attention
If no break, then all below cases will be executed, indifferently of the condition case.

26.F

26. What is the output of the following code snippet?

```
9:  int w = 0, r = 1;
10: String name = "";
11: while (w < 2) {
12:    name += "A";
13:    do {
14:       name += "B";
15:       if (name.length()>0) name += "C";
16:       else break;
17:    } while (r <=1);
18:    r++; w++; }
19: System.out.println(name);
```

A. `ABC`
B. `ABCABC`
C. `ABCABCABC`
D. Line 15 contains a compilation error.
E. Line 18 contains a compilation error.
F. The code compiles but never terminates at runtime.
G. The code compiles but throws a `NullPointerException` at runtime.

27.D

28.F

28. What is the output of calling `getFish("goldie")`?

```
40: void getFish(Object fish) {
41:    if (!(fish instanceof String guppy))
42:       System.out.print("Eat!");
43:    else if (!(fish instanceof String guppy)) {
44:       throw new RuntimeException();
45:    }
46:    System.out.print("Swim!");
47: }
```

A. `Eat!`
B. `Swim!`
C. `Eat!` followed by an exception
D. `Eat!Swim!`
E. An exception is printed
F. None of the above

28. F. Based on flow scoping, `guppy` is in scope after lines 41–42 if the type is not a `String`. In this case, line 43 declares a variable `guppy` that is a duplicate of the previously defined local variable defined on line 41. For this reason, the code does not compile, and option F is correct. If a different variable name was used on line 43, then the code would compile and print `Swim!` at runtime with the specified input.

The problem is the guppy appears 2 times with the same name.

```java
if (!(fish instanceof String guppy)) {
    // guppy is NOT in scope ❌
} else {
    // guppy IS in scope ✅
}
```

```
if (fish instanceof String guppy) {
    // guppy is in scope here ✅
} else {
    // guppy is NOT in scope here ❌
}
```

29. C

29. What is the result of the following code?

```
1: public class PrintIntegers {
2:    public static void main(String[] args) {
3:       int y = -2;
4:       do System.out.print(++y + " ");
5:       while (y <= 5);
6: } }
```

This will compile even we don't have " {}"

A. `-2 -1 0 1 2 3 4 5`
B. `-2 -1 0 1 2 3 4`
C. `-1 0 1 2 3 4 5 6`
D. `-1 0 1 2 3 4 5`
E. The code will not compile because of line 5.
F. The code contains an infinite loop and does not terminate.

30.

30. What is the minimum number of lines that would need to be changed or removed for the following code to compile and return a value when called with `dance(10)`?

```
41: double dance(Object speed) {
42:    return switch (speed) {
43:       case 5 -> {yield 4};
44:       case 10 -> 8;
45:       case 15,20 -> 12;
46:       default -> 20;
47:       case null -> 16;
48:    }
49: }
```

A. Zero, the code compiles and runs without issue
B. One
C. Two
D. Three
E. Four
F. Five
G. Six

30. E. On line 43, the semicolon should be after the `yield` statement, not outside the brace. Line 48 is missing a semicolon after the `return` statement containing the `switch` expression. For these reasons, at least two lines must be corrected. Next, lines 43, 44, and 45 do not compile because the numeric values are not compatible with the reference type for `Object`. We can fix this by changing line 41 to pass `speed` as a compatible type, such as `Integer`. Finally, the `default` clause on line 46 dominates the proceeding `case null` on line 47. Removing line 47 fixes this issue, as `case null` is not required. Since we can get the code to compile by changing or removing four lines, option E is the correct answer.

Chapter4

Wednesday, October 8, 2025 11:47 AM

Strings

Concatenation:

1. If both operands are numeric, `+` means numeric addition.
2. If either operand is a `String`, `+` means concatenation.
3. The expression is evaluated left to right.

!!! Need to know : codePointAt()

Don't worry! You do not need to memorize the ASCII or Unicode values. You just need to know that if you see `codePointAt()` on the exam that it functions similarly to `charAt()` for ASCII characters, returning the numeric value of the character at the location.

```
System.out.println("abc".equals("ABC"));           // false
System.out.println("ABC".equals("ABC"));           // true
System.out.println("ABC".equals(6));               // false
System.out.println("abc".equalsIgnoreCase("ABC")); // true
```

```
public String strip()
public String stripLeading()
public String stripTrailing()
public String trim()
```

The following code shows how to use these methods:

```
System.out.println("abc".strip());           // abc
System.out.println("\n   a b c\n".strip());  // a b c

String text = " abc\t ";
System.out.println(text.trim().length());    // 3
System.out.println(text.strip().length());   // 3
System.out.println(text.stripLeading().length()); // 4
System.out.println(text.stripTrailing().length()); // 4
```

Working with Indentation

Now that Java supports text blocks, it is helpful to have methods that deal with indentation. Both of these are a little tricky, so read carefully!

```
public String indent(int numberSpaces)
public String stripIndent()
```

The `indent()` method adds the same number of blank spaces to the beginning of each line if you pass a positive number. If you pass a negative number, it tries to remove that number of whitespace characters from the beginning of the line. If you pass zero, the information will not change.

Method	Indent change	Normalizes existing line breaks	Adds line break at end if missing
`indent(n)` where `n > 0`	Adds `n` spaces to beginning of each line	Yes	Yes
`indent(n)` where `n == 0`	No change	Yes	Yes
`indent(n)` where `n < 0`	Removes up to `n` spaces from each line where the same number of characters is removed from each nonblank line	Yes	Yes
`stripIndent()`	Removes all leading incidental whitespace	Yes	No

```
var name = "Kate";
var orderId = 5;

// All print: Hello Kate, order 5 is ready
System.out.println("Hello "+name+", order "+orderId+" is ready");
System.out.println(String.format("Hello %s, order %d is ready",
   name, orderId));
System.out.println("Hello %s, order %d is ready"
   .formatted(name, orderId));
```

Symbol	Description
`%s`	Applies to any type, commonly `String` values
`%d`	Applies to integer values like `int` and `long`
`%f`	Applies to floating point values like `float` and `double`
`%n`	Inserts a line break using the system-dependent line separator

```
var x = "Hello World";
var z = " Hello World".trim();
System.out.println(x == z); // false
```

You can also do the opposite and tell Java to use the string pool. The `intern()` method will use an object in the string pool if one is present.

```
public String intern()
```

If the literal is not yet in the string pool, Java will add it at this time.

```
var name = "Hello World";
var name2 = new String("Hello World").intern();
System.out.println(name == name2);     // true
```

Finally, you can type the `[]` before or after the name, and adding a space is optional. This means that all five of these statements do the exact same thing:

```
int[] numAnimals;
int [] numAnimals2;
int []numAnimals3;
int numAnimals4[];
int numAnimals5 [];
```

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder();   // DOES NOT COMPILE
7: objects[0] = new StringBuilder();        // Careful!
```

Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A `StringBuilder` object can clearly go in an `Object[]`. The problem is that we don't actually have an `Object[]`. We have a `String[]` referred to from an `Object[]` variable. At runtime, the code throws an `ArrayStoreException`. You don't need to memorize the name of this exception, but you do need to know that this line will compile and throw an exception.

You can use 7Up, the soda, to help remember the order. Numbers (7) sort first, followed by uppercase (U), and then lowercase (p).

- A **negative** number means the first array is smaller than the second.
- A **zero** means the arrays are equal.
- A **positive** number means the first array is larger than the second.

Here's an example:

```
System.out.println(Arrays.compare(new int[] {1}, new int[] {2}));
```

Example to visualize all

```
int[] a = {1};
int[] b = {1, 2};
int[] c = {};

System.out.println(Arrays.equals(a, b)); // false
System.out.println(Arrays.compare(a, b)); // -1 (a smaller)
System.out.println(Arrays.compare(a, c)); // -1 (1 < 2)
System.out.println(Arrays.compare(b, a)); // 1 (longer wins)
```

Compare Rules:

- If both arrays are the same length and have the same values in each spot in the same order, return zero.
- If all the elements are the same but the second array has extra elements at the end, return a negative number.
- If all the elements are the same, but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

Date and Time

You obtain date and time instances using a `static` method.

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

Each of the four classes has a `static` method called `now()`, which gives the current date and time. Your output is going to depend on the date/time when you run it and where you live. The authors live in the United States, making the output look like the following when run on July 31 at 9:23 a.m.:

```
2025-07-29
09:23:07.760
2025-07-29T09:23:07.760
2025-07-29T09:23:07.760-04:00[America/New_York]
```

First, let's try to figure out how far apart the following moments are in time. Notice how India has a half hour offset, not a full hour. To approach a problem like this, you subtract the time zone from the time. This gives you the GMT equivalent of the time.

```
2025-06-20T06:59+05:30[Asia/Kolkata]     // GMT 2025-06-20 01:29
2025-06-20T07:59-04:00[US/Eastern]       // GMT 2025-06-20 11:59
```

LocalDate

```
var date1 = LocalDate.of(2025, Month.JANUARY, 20);
var date2 = LocalDate.of(2025, 1, 20);
```

LocalTime

```
var time1 = LocalTime.of(6, 15);           // hour and minute
var time2 = LocalTime.of(6, 15, 30);       // + seconds
var time3 = LocalTime.of(6, 15, 30, 200);  // + nanoseconds
```

LocalDataTime

```
var dateTime3 = LocalDateTime.of(2025, Month.JANUARY, 20, 6, 15, 30);
var dateTime2 = LocalDateTime.of(date1, time2);
```

```
public static LocalDateTime of(int year, int month,
    int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
    int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
    int dayOfMonth, int hour, int minute, int second, int nanos)
```

Others take a `Month` reference:

```
public static LocalDateTime of(int year, Month month,
    int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month,
    int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
    int dayOfMonth, int hour, int minute, int second, int nanos)
```

```
public static LocalDateTime of(LocalDate date, LocalTime time)
```

ZonedDataTime

```
var zone = ZoneId.of("US/Eastern");
var zoned1 = ZonedDateTime.of(2025, 1, 20,
    6, 15, 30, 200, zone);
var zoned2 = ZonedDateTime.of(date1, time1, zone);
var zoned3 = ZonedDateTime.of(dateTime1, zone);
```

```
public static ZonedDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second,
  int nanos, ZoneId zone)
public static ZonedDateTime of(LocalDate date, LocalTime time,
  ZoneId zone)
public static ZonedDateTime of(LocalDateTime dateTime, ZoneId zone)
```

!!!Atention

Notice that there isn't an option to pass in the `Month` enum. Also, we did not use a constructor in any of the examples. The date and time classes have private constructors along with `static` methods that return instances. This is known as the factory pattern. The exam creators may throw something like this at you:

```
var d = new LocalDate(); // DOES NOT COMPILE
```

Don't fall for this. You are not allowed to construct a date or time object directly.

Another trick is what happens when you pass invalid numbers to `of()`, for example:

```
var d = LocalDate.of(2025, Month.JANUARY, 32); // RuntimeException
```

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
date = date.plusMinutes(1);      // DOES NOT COMPILE
```

`LocalDate` and `LocalDateTime` have a method to convert themselves into `long` values, equivalent to the number of milliseconds that have passed since January 1, 1970, referred to as the *epoch*. What's special about this date? That's what Unix started using for date standards, so Java reused it.

Manipuating Date and Time

LocalDate

Manipulating Dates and Times

Adding to a date is easy. The date and time classes are immutable. Remember to assign the results of these methods to a reference variable so they are not lost.

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
System.out.println(date);    // 2025-01-20
date = date.plusDays(2);
System.out.println(date);    // 2025-01-22
date = date.plusWeeks(1);
System.out.println(date);    // 2025-01-29
date = date.plusMonths(1);
System.out.println(date);    // 2025-02-28
date = date.plusYears(5);
System.out.println(date);    // 2030-02-28
```

LocalDateTime

There are also nice, easy methods to go backward in time. This time, let's work with `LocalDateTime`:

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
var time = LocalTime.of(5, 15);
var dateTime = LocalDateTime.of(date, time);
System.out.println(dateTime);       // 2025-01-20T05:15
dateTime = dateTime.minusDays(1);
System.out.println(dateTime);       // 2025-01-19T05:15
dateTime = dateTime.minusHours(10);
System.out.println(dateTime);       // 2025-01-18T19:15
dateTime = dateTime.minusSeconds(30);
System.out.println(dateTime);       // 2025-01-18T19:14:30
```

TABLE 4.7 Conversion methods to `LocalDate`, `LocalTime`, and `LocalDateTime`

`LocalDate` to `LocalDateTime`	`atStartOfDay()`
	`atTime(int hour, int minute)`
	`atTime(int hour, int minute, int second)`
	`atTime(int hour, int minute, int second, int nanos)`
	`atTime(LocalTime time)`

`LocalTime` to `LocalDate`	`atDate(LocalDate date)`
`LocalDateTime` to `ZonedDateTime`	`atZone(ZoneId zoneId)`

	Can call on `LocalDate`?	Can call on `LocalTime`?	Can call on `LocalDateTime` or `ZonedDateTime`?
`plusYears()` `minusYears()` `withYear()` `withDayOfYear()`	Yes	No	Yes
`plusMonths()` `minusMonths()` `withMonth()`	Yes	No	Yes
`plusWeeks()` `minusWeeks()`	Yes	No	Yes
`plusDays()` `minusDays()` `withDayOfMonth()`	Yes	No	Yes
`plusHours()` `minusHours()` `withHour()`	No	Yes	Yes
`plusMinutes()` `minusMinutes()` `withMinute()`	No	Yes	Yes
`plusSeconds()` `minusSeconds()` `withSecond()`	No	Yes	Yes
`plusNanos()` `minusNanos()` `withNano()`	No	Yes	Yes

Period & Duration

TABLE 4.8 Where to use `Duration` and `Period`

	Can use with `Period`?	Can use with `Duration`?
`LocalDate`	Yes	No
`LocalDateTime`	Yes	Yes
`LocalTime`	No	Yes
`ZonedDateTime`	Yes	Yes

Instant

Working with Instants

The `Instant` class represents a specific moment in time in the GMT time zone. Suppose that you want to run a timer.

```
var now = Instant.now();
// Do something time consuming
var later = Instant.now();

var duration = Duration.between(now, later);
System.out.println(duration.toMillis()); // Returns number milliseco
```

If you have a `ZonedDateTime`, you can turn it into an `Instant`:

```
var date = LocalDate.of(2025, 5, 25);
var time = LocalTime.of(11, 55, 00);
var zone = ZoneId.of("US/Eastern");
var zonedDateTime = ZonedDateTime.of(date, time, zone);

var instant = zonedDateTime.toInstant(); // 2025-05-25T15:55:00Z
System.out.println(zonedDateTime); // 2025-05-25T11:55-04:00[US/East
System.out.println(instant);       // 2025-05-25T15:55:00Z
```

- If all the elements are the same, but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

First array	Second array	Result	Reason
`new int[] {1, 2}`	`new int[] {1}`	Positive number	The first element is the same, but the first array is longer.
`new int[] {1, 2}`	`new int[] {1, 2}`	Zero	Exact match.
`new String[] {"aa"}`	`new String[] {"aa"}`	Negative number	The first element is a substring of the second.
`new String[] {"a"}`	`new String[] {"A"}`	Positive number	Uppercase is smaller than lowercase.
`new String[] {"a"}`	`new String[] {null}`	Positive number	`null` is smaller than a letter.

TABLE 3.2 Equality vs. comparison vs. mismatch

Method	When arrays contain the same data	When arrays are different
`Arrays.equals()`	`true`	`false`
`Arrays.compare()`	`0`	Positive or negative number
`Arrays.mismatch()`	`-1`	Zero or positive index

Using mismatch()

Now that you are familiar with `compare()`, it is time to learn about `mismatch()`. If the arrays are equal, `mismatch()` returns `-1`. Otherwise, it returns the first index where they differ. Can you figure out what these print?

```
System.out.println(Arrays.mismatch(new int[] {1}, new int[] {1}));
System.out.println(Arrays.mismatch(new String[] {"a"},
    new String[] {"A"}));
System.out.println(Arrays.mismatch(new int[] {1, 2}, new int[] {1}));
```

In the first example, the arrays are the same, so the result is `-1`. In the second example, the entries at element 0 are not equal, so the result is `0`. In the third example, the entries at element 0 are equal, so we keep looking. The element at index 1 is not equal. Or, more specifically, one array has an element at index 1, and the other does not. Therefore, the result is `1`.

```
int[] vars4 [], space [][][];  // 2D and 3D arrays
```

EXAM

1.f

2.c,e,f

2. Which of these array declarations are not legal? (Choose all that apply.)
   - A. `int[][] scores = new int[5][];`
   - B. `Object[][][] cubbies = new Object[3][0][5];`
   - C. `String beans[] = new beans[6];`
   - D. `java.util.Date[] dates[] = new java.util.Date[2][];`
   - E. `int[][] types = new int[];`
   - F. `int[][] java = new int[][];`

2. C, E, F. Option C uses the variable name as if it were a type, which is clearly illegal. Options E and F don't specify any size. Although it is legal to leave out the size for later dimensions of an array, the first one is required. Option A declares a legal 2D array. Option B declares a legal 3D array. Option D declares a legal 2D array. Remember that it is normal to see classes on the exam you might not have learned. You aren't expected to know anything about them.

3.A,C,D

4.A,C,D

5.B

5. What is the result of the following code?

```
7: var sb = new StringBuilder();
8: sb.append("cat").insert(1, "ha").insert(4, "ts");
9: System.out.println(sb);
```

A. `chatcts`
B. `chhatcts`
C. `hhatcts`
D. `hhatscts`
E. An empty line.
F. The code does not compile.

6.C

6. C. Remember to watch return types on math operations. One of the tricks in line 24. The `round()` method returns an `int` when called with a `float`. However, we are calling it with a `double`, so it returns a `long`. The other trick is line 25. The `random()` method returns a `double`. Since two lines have a compiler error, option C is the answer.

7.a,e

8.a,b,f

9.a,e— C F

9. A, C, F. Arrays are zero-indexed, making option A correct and option B incorrect. They are not able to change size, which is option C. The values can be changed, making option D incorrect. Arrays may even not override `equals()`, so it uses object equality. Since two different objects are not equal, option F is correct, and options E and G are incorrect.

10.a

10. How many of these lines contain a compiler error?

```
23: int one = Math.abs(-1, 1);
24: long two = Math.round(5.5);
25: double three = Math.floor(3.0);
26: var doubles = new double[] {one, two, three};
```

A. 0
B. 1
C. 2
D. 3
E. 4

11.a

12.A,D E

12. What is output by the following code ignoring any new lines in the output? (Choose all that apply.)

```
var numbers = "mississippi".indent(1);
numbers = numbers.stripLeading();
System.out.println(numbers.substring(1, 1));
System.out.println(numbers.substring(1, 7));
System.out.println(numbers.substring(7));
```

A. `i`
B. `iii`
C. `r`
D. `ri`
E. a blank line.
F. An exception is thrown.

12. A, D, E. First, notice that the `indent()` call adds a blank space to the beginning of `numbers`, and `stripLeading()` immediately removes it. The `substring()` method has two forms. The first takes the index to start with and the index to stop immediately before. The second takes just the index to start with and goes to the end of the `String`. Remember that indexes are zero-based. The first call starts at index 1 and ends with index 1 since it needs to stop before index 1. This gives us option A. The second call starts at index 7 and ends in the same place, resulting in an empty `String`, which is option E. This prints out a blank line. The final call starts at index 7 and goes to the end of the `String` finishing up with option D.

13.D B

13. What is the result of the following code?

```
public class Lion {
   public void roar(String roar1, StringBuilder roar2) {
      roar1.concat("!!!");
      roar2.append("!!!");
   }
   public static void main(String[] args) {
      var roar1 = "roar";
      var roar2 = new StringBuilder("roar");
      new Lion().roar(roar1, roar2);
      System.out.println(roar1 + " " + roar2);
} }
```

A. `roar roar`
B. `roar roar!!!`
C. `roar!!! roar`
D. `roar!!! roar!!!`
E. An exception is thrown.
F. The code does not compile.

14.F C

14. Given the following, which can correctly fill in the blank allowing the code to compile? (Choose all that apply.)

```
var date = LocalDate.now();
var time = LocalTime.now();
var dateTime = date._____.time);
var zoneId = ZoneId.systemDefault();
var zonedDateTime = ZonedDateTime.of(dateTime, zoneId);
Instant instant = _____.toInstant();
```

A. `atTime()`
B. `atTime()`
C. `withTime()`
D. `dateTime.toInstant()`
E. `new Instant()`
F. `zonedDateTime.toInstant()`

15.C,E

15. What is the output of the following? (Choose all that apply.)

```
var arr = new String[] { "Fin", "Jig", "Git"};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
System.out.println(Arrays.binarySearch(arr, "Pippa"));
```

A. `[Fin, Git, Jig]`
B. `[Fin, Jig, Git]`
C. `[Git, Fin, Jig]`
D. `[Git, Jig, Fin]`
E. `-3`
F. `-2`
G. The results of `binarySearch()` are undefined in this example.

15. C, E. Numbers sort before letters, and uppercase sorts before lowercase. This makes option C one of the answers. The `binarySearch()` method looks at where a value would be inserted, which is before the second element for `Pippa`. It then negates it and subtracts one, which is option E.

16.a,b,d,g

16. Which of these statements are true? Choose all that apply.

```
var letters = new StringBuilder("abcdefg");
```

A. `letters.substring(1, 2);` returns a single-character `String`.
B. `letters.substring(1, 2);` returns a single-character `String`.
C. `letters.substring(1, 1);` returns a single-character `String`.
D. `letters.substring(1, 1);` returns a single-character `String`.
E. `letters.substring(1, 2);` throws an exception.
F. `letters.substring(1, 1);` throws an exception.
G. `letters.substring(1, 1);` throws an exception.

16. A, G. The `substring()` method includes the starting index but not the ending index. When called with 1 and 2, it returns a single-character `String`, making option A correct and option E incorrect. Calling `substring()` with 2 on both parameters is legal. It returns an empty `String`, making options B and F incorrect. Java does not allow the indexes to be specified in reverse order. Option G is correct because it throws a `StringIndexOfBoundsException`. Finally, option H is incorrect because it returns an empty `String`.

17.C

```
10: String s1 = "";
11:   pure"";
12: String s2 = "";
13:
14: s1.toUpperCase();
15: s1.trim();
16: s1.substring(3, 3);
17: s1 += "hue";
18:
19: s1 += 3;
20: s1 += 'c';
21: s1 += false;
22:
23: if ( s1 == "3cfalse") System.out.println("==");
24: if ( s1.equals("3cfalse")) System.out.println("equals");
25: System.out.println(s1.length());
```

A. 2
B. 4
C. 7
D. 10
E. ==
F. equals
G. An exception is thrown.
H. The code does not compile.

17. C, F. This question is tricky because it has several parts. First, you have to know that the text block on lines 23 and 24 is equivalent to a regular `String`. Since there is no line break at the end, this is four characters. Then, you have to know that `String` objects are immutable, which means the results of lines 13–25 are ignored. Finally, on line 26, something happens. We concatenate three new characters to `s1` and now have a `String` of length 7, making option C correct.

Next, `s1 == 3` expands to `s1 = s1 + 3`. A `String` concatenated with any other type gives a `String`. Lines 21, 24, and 26 all append to `s1`, giving a result of `"3cfalse"`. The `if` statement on line 27 returns `true` because the values of the two `String` objects are the same using object equality. For this reason, option F is correct. The `if` statement on line 28 returns `false` because the two `String` objects are not the same in memory. One comes directly from the string pool, and the other comes from building using `String` operations.

18.A,B,D

19.

20A,C
21.e

21. What is the output of the following code?

```
var date = LocalDate.of(2025, Month.APRIL, 30);
date.plusDays(2);
date.plusYears(10);
System.out.println(date.getYear() + " " + date.getMonth()
    + " " + date.getDayOfMonth());
```

A. 2025 APRIL 30
B. 2025 MAY 2
C. 2026 APRIL 2
D. 2028 APRIL 30
E. 2028 MAY 2
F. The code does not compile.
G. A runtime exception is thrown.

33. A. The date starts out as April 30, 2025. Since dates are immutable and the plus methods' return values are ignored, the result is unchanged. Therefore, option A is correct.

22.e

Chapter5 Methods

Thursday, October 9, 2025 10:40 AM

!!Attention

```
public class ParkTrip {
   public void skip1() {}
   default void skip2() {} // DOES NOT COMPILE
   void public skip3() {}  // DOES NOT COMPILE
   void skip4() {}
}
```

```
public class Mike {
   public void hike1() {}
   public void hike2() { return; }
   public String hike3() { return ""; }
   public String hike4() {}            // DOES NOT COMPILE
   public hike5() {}                   // DOES NOT COMPILE
   public String int hike6() { }       // DOES NOT COMPILE
   String hike7(int a) {               // DOES NOT COMPILE
      if (1 < 2) return "orange";
   }
}
```

```
public class Exercise {
   public void bike1() {}
   public final void bike2() {}
   public static final void bike3() {}
   public final static void bike4() {}
   public modifier void bike5() {}       // DOES NOT COMPILE
   public void final bike6() {}          // DOES NOT COMPILE
   final public void bike7() {}
}
```

```
public class BeachTrip {
   public void jog1() {}
   public void 2jog() {}   // DOES NOT COMPILE
   public jog3 void() {}   // DOES NOT COMPILE
   public void Jog_$() {}
   public _() {}           // DOES NOT COMPILE
   public void() {}        // DOES NOT COMPILE
}
```

Accessing a Static Variable or Method

Usually, accessing a `static` member is easy.

```
public class Snake {
   public static long hiss = 2;
}
```

You just put the class name before the method or variable, and you are done. Here's an example:

```
System.out.println(Snake.hiss);
```

Nice and easy. There is one rule that is trickier. You can use an instance of the object to call a `static` method. The compiler checks for the type of the reference and uses that instead of the object—which is sneaky of Java. This code is perfectly legal:

```
5: Snake s = new Snake();
6: System.out.println(s.hiss);  // s is a Snake
7: s = null;
8: System.out.println(s.hiss);  // s is still a Snake
```

Varargs

Which method do you think is called if we pass an `int[]`?

```
public class Toucan {
   public void fly(int[] lengths) {}
   public void fly(int… lengths) {}     // DOES NOT COMPILE
}
```

Trick question! Remember that Java treats varargs as if they were an array. This means the method signature is the same for both methods. Since we are not allowed to overload methods with the same parameter list, this code doesn't compile. Even though the code doesn't look the same, it compiles to the same parameter list.

Method Name:

```
public class Measurement {
   int getHeight1() {
      int temp = 9;
      return temp;
   }
   int getHeight2() {
      int temp = 9L;  // DOES NOT COMPILE
      return temp;
   }
   int getHeight3() {
      long temp = 9L;
      return temp;    // DOES NOT COMPILE
   }
}
```

```
public class Bird {
   public void fly1() {}
   public void fly2()          // DOES NOT COMPILE
   public void fly3(int a) { int name = 5; }
}
```

The `fly1()` method is a valid declaration with an empty method body. The `fly2()` method doesn't compile because it is missing the braces around the empty method body. Methods are required to have a body unless they are declared `abstract`. We cover `abstract` methods in Chapter 6, "Class Design." The `fly3()` method is a valid declaration with one statement in the method body.

```
public void zooAnimalCheckup(boolean isWeekend) {
   final int rest;
   if(isWeekend) rest = 5;
   System.out.print(rest);  // DOES NOT COMPILE
}
```

Does using the `final` modifier mean we can't modify the data? Nope. The `final` attribute refers only to the variable reference; the contents can be freely modified (assuming the object isn't immutable).

The `rest` variable might not have been assigned a value, such as if `isWeekend` is `false`. Since the compiler does not allow the use of local variables that may not have been assigned a value, the code does not compile.

Effectively Final Variables

An *effectively final* local variable is one that is not modified after it is assigned. This means that the value of a variable doesn't change after it is set, regardless of whether it is explicitly marked as `final`. If you aren't sure whether a local variable is effectively final, just add the `final` keyword. If the code still compiles, the variable is effectively final.

Given this definition, which of the following variables are effectively final?

```
11: public String zooFriends() {
12:    String name = "Harry the Hippo";
13:    var size = 10;
14:    boolean wet;
15:    if(size> 100) size++;
16:    name.substring(0);
17:    wet = true;
18:    return name;
19: }
```

Modifier	Description	Chapter Covered
`final`	Specifies that the instance variable must be initialized with each instance of the class exactly once	Chapter 5
`volatile`	Instructs the JVM that the value in this variable may be modified by other threads	Chapter 13
`transient`	Used to indicate that an instance variable should not be serialized with the class	Chapter 14

In Chapter 1, we show that instance variables receive default values based on their type when not set. For example, `int` receives a default value of `0`, while an object reference receives a default value of `null`. The compiler does not apply a default value to `final` variables, though. A `final` instance or `final static` variable must receive a value when it is declared or as part of initialization.

Varargs

Rules for Creating a Method with a Varargs Parameter

1. A method can have at most one varargs parameter.
2. If a method contains a varargs parameter, it must be the last parameter in the list.

```
public class VisitAttractions {
   public void walk1(int… steps) {}
   public void walk2(int start, int… steps) {}
   public void walk3(int… steps, int start) {}     // DOES NOT COMPIL
   public void walk4(int… start, int… steps) {}  // DOES NOT COMPILE
}
```

```
import java.util.List;
import static java.util.Arrays.asList;          // static import
public class ZooParking {
   public static void main(String[] args) {
      List<String> list = asList("one", "two"); // No Arrays. prefix
   }
}
```

Calling varargs methods with varargs.

```
// Pass an array
int[] data = new int[] {1, 2, 3};
walk1(data);

// Pass a list of values
walk1(1,2,3);
```

	private	package	protected	public
the same class	Yes	Yes	Yes	Yes
another class in the same package	No	Yes	Yes	Yes
a subclass in a different package	No	No	Yes	Yes
an unrelated class in a different package	No	No	No	Yes

```
Long badGorilla = 8;  // DOES NOT COMPILE
```

The compiler will automatically cast or autobox the `int` value to `long` or `Integer`, respectively. Neither of these types can be assigned to a `Long` reference variable, though, so the code does not compile. Compare this behavior to the previous example with `ears`, where the unboxed primitive value could be implicitly cast to a larger primitive type.

The types have to be compatible, though, as shown in the following examples.

```
Integer[] winterHours = { 10.5, 17.0 };  // DOES NOT COMPILE
Double[] summerHours  = { 9, 21 };       // DOES NOT COMPILE
```

```
public class Chimpanzee {
   public void climb(long t) {}
   public void swing(Integer u) {}
   public void jump(int v) {}
   public static void main(String[] args) {
      var c = new Chimpanzee();
      c.climb(123);
      c.swing(123);
      c.jump(123L);  // DOES NOT COMPILE
   }
}
```

Overloading

```
public class Eagle {
   public void fly(int numMiles) {}
   public int fly(int numMiles) { return 1; }     // DOES NOT COMPILE
}
```

```
public class Hawk {
   public void fly(int numMiles) {}
   public static void fly(int numMiles) {}     // DOES NOT COMPILE
   public void fly(int numKilometers) {}       // DOES NOT COMPILE
}
```

This method doesn't compile because it differs from the original only by return type. The method signatures are the same, so they are duplicate methods as far as Java is concerned.

```
public class Glider {
   public static String glide(String s) {
      return "1";
   }
   public static String glide(String... s) {
      return "2";
   }
   public static String glide(Object o) {
      return "3";
   }
   public static String glide(String s, String t) {
      return "4";
   }
   public static void main(String[] args) {
      System.out.print(glide("a"));
      System.out.print(glide("a", "b"));
      System.out.print(glide("a", "b", "c"));
   }
}
```

It prints out `142`. The first call matches the signature taking a single `String` because that is the most specific match. The second call matches the signature taking two `String` parameters since that is an exact match. It isn't until the third call that the varargs version is used since there are no better matches.

EXAM

1. A, E
2. b,c,
3. a, d,
4. a,b,c,e
5. a,c,d
6. a,b,d,f

6. Which of the following methods compile? (Choose all that apply.)
   A. `public void violin(int... nums) {}`
   B. `public void viola(String values, int... nums) {}`
   C. `public void cello(int... nums, String values) {}`
   D. `public void bass(String... values, int... nums) {}`
   E. `public void flute(String[] values, ...int nums) {}`
   F. `public void oboe(String[] values, int[] nums) {}`

Option D is incorrect because two varargs parameters are not allowed in the same method. Option E is incorrect because the ... for a varargs must be after

6. A, B, F. Options A and B are correct because the single varargs parameter is the last parameter declared. Option F is correct because it doesn't use any varargs parameters. Option C is incorrect because the varargs parameter is not last. Option D is incorrect because two varargs parameters are not allowed in the same method. Option E is incorrect because the ... for a varargs must be after the type, not before it.

7. ~~b,c,d,e,f~~

7. Given the following method, which of the method calls return `2`? (Choose all that apply.)

```
public int juggle(boolean b, boolean... b2) {
   return b2.length;
}
```

A. `juggle();`
B. `juggle(true);`

```
Java
class Dog {
   String name;
```

```
public int juggle(boolean b, boolean… b2) {
   return b2.length;
}
```

A. `juggle();`
B. `juggle(true);`
C. `juggle(true, true);`
D. `juggle(true, true, true);`
E. `juggle(true, {true, true});`
F. `juggle(true, new boolean[2]);`

7. D, F. Options D and F are correct. Option D passes the initial parameter plus two more to turn into a varargs array of size 2. Option F passes the initial parameter plus an array of size 2. Option A does not compile because it does not pass the initial parameter. Option E does not compile because it does not declare an array properly. It should be `new boolean[] {true, true}`. Option B creates a varargs array of size 0, and option C creates a varargs array of size 1.

8. D.
9. c,b,d,f
10. B
11. b,e
12. B
13. B D

13. What is the output of the following code?

```
// RopeSwing.java
import rope.*;
import static rope.Rope.*;
public class RopeSwing {
   private static Rope rope1 = new Rope();
   private static Rope rope2 = new Rope();
   {
      System.out.println(ropes.length);
   }
   public static void main(String[] args) {
      rope1.length = 1;
      rope2.length = 8;
      System.out.println(ropes.length);
   }
}

// Rope.java
package rope;
public class Rope {
   public static int length = 0;
}
```

A. `0`
B. `00`
C. `2`
D. `8`
E. The code does not compile.
F. An exception is thrown.

13. D. There are two details to notice in this code. First, note that `RopeSwing` has an instance initializer and not a `static` initializer. Since `RopeSwing` is never constructed, the instance initializer does not run. The other detail is that `length` is `static`. Changes from any object update this common `static` variable. The code prints `0`, making option D correct.

14. E
15. a,b

15. Which of the following can replace line 2 to make this code compile?

```
1: import java.util.*;
2: // INSERT CODE HERE
3: public class Imports {
4:    public void method(ArrayList<String> list) {
5:       sort(list);
6:    }
7: }
```

A. `import static java.util.Collections;`
B. `import static java.util.Collections.*;`
C. `import static java.util.Collections.sort(ArrayList<String>);`
D. `static import java.util.Collections;`
E. `static import java.util.Collections.*;`
F. `static import java.util.Collections.sort(ArrayList<String>);`

15. B. The two valid ways to do this are `import static java.util.Collections.*;` and `import static java.util.Collections.sort;`, making option B correct. Option A is incorrect because you can do a static import only on `static` members. Classes such as `Collections` require a regular `import`. Option C is nonsense as method parameters have no business in an import. Options D, E, and F try to trick you into reversing the syntax of `import static`.

16. e
17. b
18. a,d B E

18. Which of the following are output by the following code? (Choose all that apply.)

```
public class StringBuilders {
   public static StringBuilder work(StringBuilder a,
      StringBuilder b) {
      a = new StringBuilder("a");
      b.append("b");
      return a;
   }
   public static void main(String[] args) {
      var s1 = new StringBuilder("s1");
      var s2 = new StringBuilder("s2");
      var s3 = work(s1, s2);
      System.out.println("s1 = " + s1);
      System.out.println("s2 = " + s2);
      System.out.println("s3 = " + s3);
   }
}
```

A. `s1 = a`
B. `s1 = s1`
C. `s2 = s2`
D. `s2 = s2b`
E. `s3 = a`
F. The code does not compile.

19. b,c,e,f

19. Which of the following will compile when independently inserted in the following code? (Choose all that apply.)

```
1:  public class Order3 {
2:     final String value1 = "red";
3:     static String value2 = "blue";
4:     String value3 = "yellow";
5:     {
6:        // CODE SNIPPET 1
7:     }
8:     static {
9:        // CODE SNIPPET 2
10:    } }
```

A. Insert at line 6: `value1 = "green";`
B. Insert at line 6: `value2 = "purple";`
C. Insert at line 6: `value3 = "orange";`
D. Insert at line 9: `value1 = "magenta";`
E. Insert at line 9: `value2 = "cyan";`
F. Insert at line 9: `value3 = "turquoise";`

20. a,e,
21. b,b

21. Which method signatures are valid overloads of the following method signature? (Choose all that apply.)

```
public void moo(int m, int… n)
```

Java

```
class Dog {
    String name;
}

public void rename(Dog d) {
    d.name = "Rex";          // modifies the original object
    d = new Dog();           // creates a new object
    d.name = "Max";          // modifies the new object
}
```

And you call:

Java

```
Dog myDog = new Dog();
myDog.name = "Buddy";
rename(myDog);
System.out.println(myDog.name); // Output: Rex
```

But what happens at runti

Let's say you call:

Java

```
moo(1);          // One int
moo(1, 2);       // Two ints
moo(1, 2, 3);    // Three ints
```

Then:

- `moo()` matches **both**:
  - `moo(int m, int… n)` → `m = 1`, `n = {}` (empty array)
  - `moo(int… x)` → `x = {1}`
- `moo(1, 2)` matches both:
  - `moo(int m, int… n)` → `m = 1`, `n = {2}`
  - `moo(int… x)` → `x = {1, 2}`
- `moo(1, 2, 3)` matches both:
  - `moo(int m, int… n)` → `m = 1`, `n = {2, 3}`
  - `moo(int… x)` → `x = {1, 2, 3}`

So how does the compiler choose?

Java uses **most specific match**:

- `moo(int m, int… n)` is **more specific** than `moo(int… x)` because it requires **at least one fixed argument**.
- So in all cases, the compiler will prefer `moo(int m, int… n)`.

21. Which method signatures are valid overloads of the following method signature? (Choose all that apply.)

```
public void moo(int m, int… n)
```

A. `public void moo(int a, int… b)`
B. `public int moo(char ch)`
C. `public void moooo(int… z)`
D. `private void moo(int… x)`
E. `public void moooo(int y)`
F. `public void moo(int… c, int d)`
G. `public void moo(int… i, int j…)`

21. B, D. Option A is incorrect because it has the same parameter list of types and therefore the same signature as the original method. Options B and D are the correct answers, as they are valid method overloads in which the types of parameters change. When overloading methods, the return type and access modifiers do not need to be the same. Options C and E are incorrect because the method name is different. Options F and G do not compile. There can be at most one varargs parameter, and it must be the last element in the parameter list.

Chapter6 Class Design

Monday, October 13, 2025 11:27 AM

Modifier	Description	Chapter covered
`final`	The class may not be extended.	Chapter 6
`abstract`	The class is abstract, may contain `abstract` methods, and requires a concrete subclass to instantiate.	Chapter 6
`sealed`	The class may only be extended by a specific list of classes.	Chapter 7
`non-sealed`	A subclass of a sealed class permits potentially unnamed subclasses.	Chapter 7
`static`	Used for `static` nested classes defined within a class.	Chapter 7

```
public class Mammal {}

public final class Rhinoceros extends Mammal {}

public class Clara extends Rhinoceros {}  // DOES NOT COMPILE
```

On the exam, pay attention to any class marked `final`. If you see another class extending it, you know immediately the code does not compile.

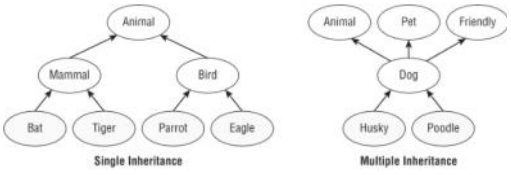
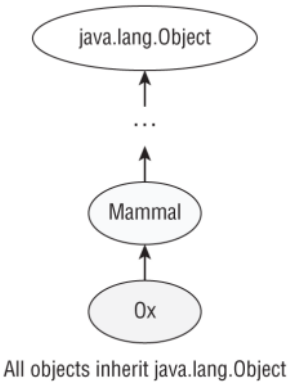


FIGURE 6.2 Types of inheritance



All objects inherit java.lang.Object

Trying to declare a top-level class with `protected` or `private` class will lead to a compiler error, though.

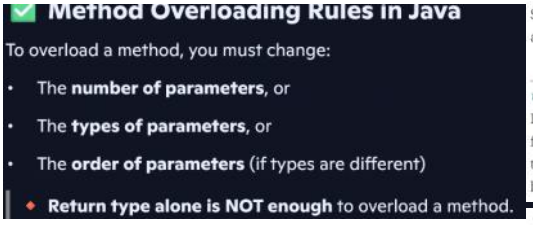
```
// ClownFish.java
protected class ClownFish{} // DOES NOT COMPILE

// BlueTang.java
private class BlueTang {} // DOES NOT COMPILE
```

Does that mean a class can never be declared `protected` or `private`? Not exactly. In Chapter 7, we present nested types and show that when you define a class inside another, it can use any access modifier.

!!!Attention

Like method parameters, constructor parameters can be any valid class, array, or primitive type, including generics, but may not include `var`. For example, the following does not compile:

```
public class Bonobo {
   public Bonobo(var food) {  // DOES NOT COMPILE
   }
}
```



Calling `this()` has one special rule you need to know. If you choose to call it, the `this()` call must be the first statement in the constructor. The side effect of this is that there can be only one call to `this()` in any constructor.

```
3:    public Hamster(int weight) {
4:       System.out.println("chew");
5:       // Set weight and default color
6:       this(weight, "brown");     // DOES NOT COMPILE
7:    }
```

Even though a print statement on line 4 doesn't change any variables, it is still a Java statement and is not allowed to be inserted before the call to `this()`. The comment on line 5 is just fine. Comments aren't considered statements and are allowed anywhere.

Here we summarize the rules you should know about constructors that we covered in this section. Study them well!

- A class can contain many overloaded constructors, provided the signature for each is distinct.
- The compiler inserts a default no-argument constructor if no constructors are declared.
- If a constructor calls `this()`, then it must be the first line of the constructor.
- Java does not allow cyclic constructor calls.

```
   public Hamster(int weight) {  // Second constructor
      this(weight, "brown");
   }
```

Success! Now Java calls the constructor that takes two parameters, with `weight` and `color` set as expected.

THIS VS. THIS()

Despite using the same keyword, `this` and `this()` are very different. The first, `this`, refers to an instance of the class, while the second, `this()`, refers to a constructor call within the class. The exam may try to trick you by using both together, so make sure you know which one to use and why.

There's one last rule for overloaded constructors that you should be aware of. Consider the following definition of the `Gopher` class:

```
public class Gopher {
   public Gopher(int dugHoles) {
      this(5);  // DOES NOT COMPILE
   }
}
```

The compiler is capable of detecting that this constructor is calling itself infinitely. This is often referred to as a *cycle* and is similar to the infinite loops that we discussed in Chapter 3, "Making Decisions." Since the code can never terminate, the compiler stops and reports this as an error. Likewise, this also does not compile.

```
public class Gopher {
   public Gopher() {
      this(5);  // DOES NOT COMPILE
   }
   public Gopher(int dugHoles) {
      this();   // DOES NOT COMPILE
   }
}
```

Here we summarize the rules you should know about constructors that we covered in this section. Study them well!

- A class can contain many overloaded constructors, provided the signature for each is distinct.
- The compiler inserts a default no-argument constructor if no constructors are declared.
- If a constructor calls `this()`, then it must be the first line of the constructor.
- Java does not allow cyclic constructor calls.

```
    this(5);  // DOES NOT COMPILE
  }
  public Gopher(int dugHoles) {
    this();   // DOES NOT COMPILE
  }
}
```

Like calling `this()`, calling `super()` can only be used as the first statement of the constructor. For example, the following two class definitions will not compile:

```
public class Zoo {
   public Zoo() {
      System.out.println("Zoo created");
      super();     // DOES NOT COMPILE
   }
}
```

```
public class Zoo {
   public Zoo() {
      super();
      System.out.println("Zoo created");
      super();     // DOES NOT COMPILE
   }
}
```

The first class will not compile because the call to the parent constructor must be the first statement of the constructor. In the second code snippet, `super()` is the first statement of the constructor, but it is also used as the third statement. Since `super()` can only be called once as the first statement of the constructor, the code will not compile.

We conclude this section by adding two constructor rules to your skill set.

- If a constructor calls `super()` or `this()`, then it must be the first line of the constructor.
- If the constructor does not contain a `this()` or `super()` reference, then the compiler automatically inserts `super()` with no arguments as the first line of the constructor.

Initialize Class X

1. Initialize the superclass of X.
2. Process all `static` variable declarations in the order in which they appear in the class.
3. Process all `static` initializers in the order in which they appear in the class.

Let's try an example with no inheritance. See if you can figure out what the following application outputs:

```
1:  public class ZooTickets {
2:     private String name = "BestZoo";
3:     { System.out.print(name + "-"); }
4:     private static int COUNT = 0;
5:     static { System.out.print(COUNT + "-"); }
6:     static { COUNT += 10; System.out.print(COUNT + "-"); }
7:
8:     public ZooTickets() {
9:        System.out.print("z-");
10:    }
11:
12:    public static void main(String... patrons) {
13:       new ZooTickets();
14:    } }
```

The output is as follows:

```
0-10-BestZoo-z-
```

Initialize Instance of X

1. Initialize *Class* X if it has not been previously initialized.
2. Initialize the superclass instance of X.
3. Process all instance variable declarations in the order in which they appear in the class.
4. Process all instance initializers in the order in which they appear in the class.
5. Initialize the constructor, including any overloaded constructors referenced with `this()`.

cuted. Remember, constructors are executed from the bottom up, but since the first line of every constructor is a call to another constructor, the flow ends up with the parent constructor executed before the child constructor.

```
1:  class GiraffeFamily {
2:     static { System.out.print("A"); }
3:     { System.out.print("B"); }
4:
5:     public GiraffeFamily(String name) {
6:        this(1);
7:        System.out.print("C");
8:     }
9:
10:    public GiraffeFamily() {
11:       System.out.print("D");
12:    }
13:
14:    public GiraffeFamily(int stripes) {
15:       System.out.print("E");
16:    }
17: }
18: public class Okapi extends GiraffeFamily {
19:    static { System.out.print("F"); }
20:
21:    public Okapi(int stripes) {
22:       super("sugar");
23:       System.out.print("G");
24:    }
25:    { System.out.print("H"); }
26:
27:    public static void main(String[] grass) {
28:       new Okapi(1);
29:       System.out.println();
30:       new Okapi(2);
31:    }
```

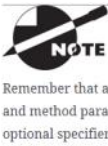
AFBECGH HG
BECGH HG

!! First instantiate all instance fields(including all block initializing) in order, and then execute constructor. This is after the super call from constructor.

To override a method, you must follow a number of rules. The compiler performs the following checks when you override a method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*.

While these rules may seem confusing or arbitrary at first, they are needed for consistency. Without these rules in place, it is possible to create contradictions within the Java language.



Remember that a method signature is composed of the name of the method and method parameters. It does not include the return type, access modifiers, optional specifiers, or any declared exceptions.

METHOD OVERRIDING INFINITE CALLS

You might be wondering whether the use of `super` in the previous example was required. For example, what would the following code output if we removed the `super` keyword?

```
public double getAverageWeight() {
   return getAverageWeight()+20;  // StackOverflowError
}
```

In this example, the compiler would not call the parent `Marsupial` method; it would call the current `Kangaroo` method. The application will attempt to call itself infinitely and produce a `StackOverflowError` at runtime.

Rule #3: Checked Exceptions

The third rule says that overriding a method cannot declare new checked exceptions or checked exceptions broader than the inherited method. This is done for polymorphic reasons similar to limiting access modifiers. In other words, you could end up with an object that is more restrictive than the reference type it is assigned to, resulting in a checked exception that is not handled or declared. One implication of this rule is that overridden methods are free to declare any number of new unchecked exceptions.

Let's try an example:

```
public class Reptile {
   protected void sleep() throws IOException {}

   protected void hide() {}

   protected void exitShell() throws FileNotFoundException {}
}

public class GalapagosTortoise extends Reptile {
   public void sleep() throws FileNotFoundException {}

   public void hide() throws FileNotFoundException {} // DOES NOT CO

   public void exitShell() throws IOException {}      // DOES NOT CO
}
```

A simple test for covariance is the following: given an inherited return type A and an overriding return type B, can you assign an instance of B to a reference variable for A without a cast? If so, then they are covariant. This rule applies to primitive types and object types alike. If one of the return types is `void`, then they both must be `void`, as nothing is covariant with `void` except itself.

!! Does not work for primitive types.
Exp int and short.
Even:
short b= 1;
int a = b
In Overriding this will throw an error.
It works only with reference objects, and for primitive types need to be the exact return type.

```
public class Bear {
   public static void sneeze() {
      System.out.println("Bear is sneezing");
   }
   public void hibernate() {
      System.out.println("Bear is hibernating");
   }
   public static void laugh() {
      System.out.println("Bear is laughing");
   }
}

public class SunBear extends Bear {
   public void sneeze() {           // DOES NOT COMPILE
      System.out.println("Sun Bear sneezes quietly");
   }
   public static void hibernate() { // DOES NOT COMPILE
      System.out.println("Sun Bear is going to sleep");
   }
   protected static void laugh() {  // DOES NOT COMPILE
      System.out.println("Sun Bear is laughing");
   }
}
```

Easy so far. But there are some rules you need to be aware of:

- Only instance methods can be marked `abstract` within a class, not variables, constructors, or `static` methods.
- An abstract class can include zero or more abstract methods, while a non-abstract class cannot contain any.
- A non-abstract class that extends an abstract class must implement all inherited abstract methods.
- Overriding an abstract method follows the existing rules for overriding methods that you learned about earlier in the chapter.

Like the `final` modifier, the `abstract` modifier can be placed before or after the access modifier in class and method declarations, as shown in this `Tiger` class:

```
abstract public class Tiger {
   abstract public int claw();
}
```

The `abstract` modifier cannot be placed after the `class` keyword in a class declaration or after the return type in a method declaration. The following `Bear` and `howl()` declarations do not compile for these reasons:

```
public class abstract Bear {    // DOES NOT COMPILE
   public int abstract howl();  // DOES NOT COMPILE
}
```

```
public abstract class Animal {
   abstract String getName();
}

public abstract class BigCat extends Animal {
   protected abstract void roar();
}

public class Lion extends BigCat {
   public String getName() {
      return "Lion";
   }
   public void roar() {
      System.out.println("The Lion lets out a loud ROAR!");
   }
}
```

While it is not possible to declare a method `abstract` and `private`, it is possible (albeit redundant) to declare a method `final` and `private`.

Creating Constructors in Abstract Classes

Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses. For example, consider the following program:

```
abstract class Mammal {
   abstract CharSequence chew();
   public Mammal() {
      System.out.println(chew());  // Does this line compile?
   }
}

public class Platypus extends Mammal {
   String chew() { return "yummy!"; }
   public static void main(String[] args) {
      new Platypus();
   }
}
```

```
public abstract class Turtle {
   public abstract long eat()      // DOES NOT COMPILE
   public abstract void swim() {}; // DOES NOT COMPILE
   public abstract int getAge() {  // DOES NOT COMPILE
      return 10;
   }
   public abstract void sleep;     // DOES NOT COMPILE
   public void goInShell();        // DOES NOT COMPILE
}
```

```
public abstract final class Tortoise {  // DOES NOT COMPILE
   public abstract final void walk();   // DOES NOT COMPILE
}
```

abstract and *private* Modifiers

A method cannot be marked as both `abstract` and `private`. This rule makes sense if you think about it. How would you define a subclass that implements a required method if the method is not inherited by the subclass? The answer is that you can't, which is why the compiler will complain if you try to do the following:

```
public abstract class Whale {
   private abstract void sing();  // DOES NOT COMPILE
}
```

sible (albeit redundant) to declare a method `final` and `private`.

```
public abstract class Whale {
   private abstract void sing();  // DOES NOT COMPILE
}

public class HumpbackWhale extends Whale {
   private void sing() {
      System.out.println("Humpback whale is singing");
   } }
```

Creating immutable objects.

```
import java.util.*;
public final class Animal {  // Not an immutable object declaration
   private final ArrayList<String> favoriteFoods;

   public Animal() {
      this.favoriteFoods = new ArrayList<String>();
      this.favoriteFoods.add("Apples");
   }

   public List<String> getFavoriteFoods() {
      return favoriteFoods;
   } }
```

```
import java.util.*;
public final class Animal {  // An immutable object declaration
   private final List<String> favoriteFoods;

   public Animal() {
      this.favoriteFoods = new ArrayList<String>();
      this.favoriteFoods.add("Apples");
   }

   public int getFavoriteFoodsCount() {
      return favoriteFoods.size();
   }

   public String getFavoriteFoodsItem(int index) {
      return favoriteFoods.get(index);
   } }
```

We carefully followed the first three rules, but unfortunately, a malicious caller could still modify our data.

```
var zebra = new Animal();
System.out.println(zebra.getFavoriteFoods());  // [Apples]

zebra.getFavoriteFoods().clear();
zebra.getFavoriteFoods().add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoods());  // [Chocolate Chip Coo
```

```
var favorites = new ArrayList<String>();
favorites.add("Apples");

var zebra = new Animal(favorites);  // Caller still has access to fav
System.out.println(zebra.getFavoriteFoodsItem(0));  // [Apples]

favorites.clear();
favorites.add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoodsItem(0));  // [Chocolate Chi
```

Whoops! It seems like `Animal` is not immutable anymore, since its contents can change after it is created. The solution is to make a copy of the list object containing the same elements.

```
public Animal(List<String> favoriteFoods) {
   if (favoriteFoods == null || favoriteFoods.size() == 0)
      throw new RuntimeException("favoriteFoods is required");
   this.favoriteFoods = new ArrayList<String>(favoriteFoods);
}
```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected. It's the same idea as defensive driving: prevent a problem before it exists. With this approach, our `Animal` class is once again immutable.

2. Compile-time type vs. Runtime type

Java separates **what's *visible*** at compile time from **what's *executed*** at runtime.

Stage	Controlled by	Determines
Compile-time	Declared type (`A`)	What methods and fields you *can access*
Runtime	Actual object (`B`)	What *implementation* gets executed

3. Why this rule exists — type safety

Imagine if this were allowed:

```
java

A obj = new A();
obj.showB(); // allowed? (A doesn't have showB)
```

That would blow up at runtime, because not all `A` objects are `B`.

To prevent this, Java's type system enforces:

> You can only call members that are guaranteed to exist in the declared type.

This keeps Java **statically type-safe**.

6. Quick analogy

Think of the declared type as the *contract* and the actual object as the *employee*:

- You hire someone (`obj`) under the contract "A".
- Even if the person *knows more* (`B` skills), you can only *legally* ask them to do tasks written in the contract (A's methods).
- At runtime, they might perform their task in their own unique way (B's override).

!!EXAM

1. E,D -> Doesn't understood deeply the requirements.
2. A,B,F ->
3. B,C
4. F
5. ~~C~~ E

5. Which of the following completes the constructor so that this code prints out `50`?

```
class Speedster {
   int numSpots;
}
public class Cheetah extends Speedster {
   int numSpots;

   public Cheetah(int numSpots) {
      // INSERT CODE HERE
   }

   public static void main(String[] args) {
      Speedster s = new Cheetah(50);
      System.out.print(s.numSpots);
   }
```

5. E. The code compiles, making option F incorrect. An instance variable with the same name as an inherited instance variable is hidden, not overridden. This means that both variables exist, and the one that is used depends on the location and reference type. Because the `main()` method uses a reference type of `Speedster` to access the `numSpots` variable, the variable in the `Speedster` class, not the `Cheetah` class, must be set to `50`. Option A is incorrect, as it reassigns the method parameter to itself. Option B is incorrect, as it assigns the method parameter the value of the instance variable in `Cheetah`, which is `0`. Option C is incorrect, as it assigns the value to the instance variable in `Cheetah`, not `Speedster`. Option D is incorrect, as it assigns the method parameter the value of the instance variable in `Speedster`, which is `0`.

```
    // INSERT CODE HERE
  }

  public static void main(String[] args) {
    Speedster s = new Cheetah(50);
    System.out.print(s.numSpots);
  }
}
```

A. `numSpots = numSpots;`
B. `numSpots = this.numSpots;`
C. `this.numSpots = numSpots;`
D. `numSpots = super.numSpots;`
E. `super.numSpots = numSpots;`
F. The code does not compile regardless of the code inserted into the constructor.
G. None of the above.

correct, as it assigns the method parameter the value of the instance variable in `Cheetah`, which is `0`. Option C is incorrect, as it assigns the value to the instance variable in `Cheetah`, not `Speedster`. Option D is incorrect, as it assigns the method parameter the value of the instance variable in `Speedster`, which is `0`. Options A, B, C, and D all print `0` at runtime. Option E is the correct answer, as it assigns the instance variable `numSpots` in the `Speedster` class a value of `50`. The `numSpots` variable in the `Speedster` class is then correctly referenced in the `main()` method, printing `50` at runtime.

6. D,E
7. A
8. D
9. E B

9. Which of the following statements about overridden methods are true? (Choose all that apply.)
   A. An overridden method must contain method parameters that are the same or covariant with the method parameters in the inherited method.
   B. An overridden method may declare a new exception, provided it is not checked.
   C. An overridden method must be more accessible than the method in the parent class.
   D. An overridden method may declare a broader checked exception than the method in the parent class.
   E. If an inherited method returns `void`, then the overridden version of the method must return `void`.
   F. None of the above.

9. B, E. The signature must match exactly, making option A incorrect. There is no such thing as a covariant signature. An overridden method must not declare any new checked exceptions or a checked exception that is broader than the inherited method. For this reason, option B is correct, and option D is incorrect. Option C is incorrect because an overridden method may have the same access modifier as the version in the parent class. Finally, overridden methods must have covariant return types, and only `void` is covariant with `void`, making option E correct.

10. A,C
11. Tacbf -> C
12. B C

12. How many lines of the following program contain a compilation error?

```
1:  public class Rodent {
2:    public Rodent(Integer x) {}
3:    protected static Integer chew() throws Exception
4:      System.out.println("Rodent is chewing");
5:      return 1;
6:    }
7:  }
8:  class Beaver extends Rodent {
9:    public Number chew() throws RuntimeException {
10:     System.out.println("Beaver is chewing on wood
11:     return 2;
12:   } }
```

A. None
B. 1
C. 2
D. 3
E. 4
F. 5

12. C. The code doesn't compile, so option A is incorrect. The first compilation error is on line 8. Since `Rodent` declares at least one constructor, and it is not a no-argument constructor, `Beaver` must declare a constructor with an explicit call to a `super()` constructor. Line 9 contains two compilation errors. First, the return types are not covariant since `Number` is a supertype, not a subtype, of `Integer`. Second, the inherited method is `static`, but the overridden method is not, making this an invalid override. The code contains three compilation errors, although they are limited to two lines, making option C the correct answer.

13. A,G
14. B,E F

14. Which of the following statements about inheritance are correct? (Choose all that apply.)
   A. A class can directly extend any number of classes.
   B. A class can implement any number of interfaces.
   C. All variables inherit `java.lang.Object`.
   D. If class `A` is extended by `B`, then `B` is a superclass of `A`.
   E. If class `C` implements interface `D`, then `C` is a subtype of `D`.
   F. Multiple inheritance is the property of a class to have multiple direct superclasses.

a subtype of that interface, making option E correct. Finally, option F is correct as it is an accurate description of multiple inheritance, which is not permitted in Java.

15. C,
16. uq uq uqcrcrm -> D
17. C,F,F

17. Which of the following are true? (Choose all that apply.)
   A. `this()` can be called from anywhere in a constructor.
   B. `this()` can be called from anywhere in an instance method.
   C. `this.variableName` can be called from any instance method in the class.
   D. `this.variableName` can be called from any `static` method in the class.
   E. You can call the default constructor written by the compiler using `this()`.
   F. You can access a `private` constructor with the `main()` method in the same class.

17. C, F. Calling an overloaded constructor with `this()` may be used only as the first line of a constructor, making options A and B incorrect. Accessing `this.variableName` can be performed from any instance method, constructor, or instance initializer, but not from a `static` method or `static` initializer. For this reason, option C is correct, and option D is incorrect. Option E is tricky. The default constructor is written by the compiler only if no user-defined constructors were provided. And `this()` can only be called from a constructor in the same class. Since there can be no user-defined constructors in the class if a default constructor was created, it is impossible for option E to be true. Since the `main()` method is in the same class, it can call `private` methods in the class, making option F correct.

18. D,F
19. F

19. What is the output of the following code?

```
1:  class Reptile {
2:    {System.out.print("A");}
3:    public Reptile(int hatch) {}
4:    void layEggs() {
5:      System.out.print("Reptile");
6:    } }
7:  public class Lizard extends Reptile {
8:    static {System.out.print("B");}
9:    public Lizard(int hatch) {}
10:   public final void layEggs() {
11:     System.out.print("Lizard");
12:   }
13:   public static void main(String[] args) {
14:     var reptile = new Lizard(1);
15:     reptile.layEggs();
16:   } }
```

A. `AALizard`
B. `BALizard`
C. `BLizardA`
D. `ALizard`
E. The code will not compile because of line 3.
F. None of the above.

19. F. The `Reptile` class defines a constructor, but it is not a no-argument constructor. Therefore, the `Lizard` constructor must explicitly call `super()`, passing in an `int` value. For this reason, line 9 does not compile, and option F is the correct answer. If the `Lizard` class were corrected to call the appropriate `super()` constructor, then the program would print `BALizard` at runtime, with the `static` initializer running first, followed by the instance initializer, and finally the method call using the overridden method.

20. E A

```
1:  class Bird {
2:    int feathers = 0;
3:    Bird(int x) { this.feathers = x; }
4:    Bird fly() {
5:      return new Bird(1);
6:    } }
7:  class Parrot extends Bird {
```

20. E. The program compiles and runs without issue, making options A through D incorrect. The `fly()` method is correctly overridden in each subclass since the signature is the same, the access modifier is less restrictive, and the return types are covariant. For covari-

20. E A

```
1:  class Bird {
2:     int feathers = 0;
3:     Bird(int x) { this.feathers = x; }
4:     Bird fly() {
5:        return new Bird(1);
6:     } }
7:  class Parrot extends Bird {
8:     protected Parrot(int y) { super(y); }
9:     protected Parrot fly() {
10:       return new Parrot(2);
11:    } }
12: public class Macaw extends Parrot {
13:    public Macaw(int z) { super(3); }
14:    public Macaw fly() {
15:       return new Macaw(3);
16:    }
17:    public static void main(String… sing) {
18:       Bird p = new Macaw(4);
19:       System.out.print(((Parrot)p.fly()).feathers);
20:    } }
```

A. One line contains a compiler error.
B. Two lines contain compiler errors.
C. Three lines contain compiler errors.
D. The code compiles but throws a `ClassCastException` at runtime.
E. The program compiles and prints `3`.
F. The program compiles and prints `0`.

20. E. The program compiles and runs without issue, making options A through D incorrect. The `fly()` method is correctly overridden in each subclass since the signature is the same, the access modifier is less restrictive, and the return types are covariant. For covariance, `Macaw` is a subtype of `Parrot`, which is a subtype of `Bird`, so overridden return types are valid. Likewise, the constructors are all implemented properly, with explicit calls to the parent constructors as needed. Line 19 calls the overridden version of `fly()` defined in the `Macaw` class, as overriding replaces the method regardless of the reference type. This results in `feathers` being assigned a value of `3`. The `Macaw` object is then cast to `Parrot`, which is allowed because `Macaw` inherits `Parrot`. The `feathers` variable is visible since it is defined in the `Bird` class, and line 19 prints `3`, making option E the correct answer.

21. E,F,G

21. Which of the following are properties of immutable classes? (Choose all that apply.)

A. The class can contain setter methods, provided they are marked `final`.
B. The class must not be able to be extended outside the class declaration.
C. The class may not contain any instance variables.
D. The class must be marked `static`.
E. The class may not contain any `static` variables.
F. The class may only contain `private` constructors.
G. The data for mutable instance variables may be read, provided they cannot be modified by the caller.

21. B, G. Immutable objects do not include setter methods, making option A incorrect. An immutable class must be marked `final` or contain only `private` constructors, so no subclass can extend it and make it mutable, making option B correct. Options C and E are incorrect, as immutable classes can contain both instance and `static` variables. Option D is incorrect, as marking a class `static` is not a property of immutable objects. Option F is incorrect. While an immutable class may contain only `private` constructors, this is not a requirement. Finally, option G is correct. It is allowed for the caller to access data in mutable elements of an immutable object, provided they have no ability to modify these elements.

22. E D

22. What does the following program print?

```
1:  class Person {
2:     static String name;
3:     void setName(String q) { name = q; } }
4:  public class Child extends Person {
5:     static String name;
6:     void setName(String w) { name = w; }
7:     public static void main(String[] p) {
8:        final Child m = new Child();
9:        final Person t = m;
10:       m.name = "Elysia";
11:       t.name = "Sophia";
12:       m.setName("Webby");
13:       t.setName("Olivia");
14:       System.out.println(m.name + " " + t.name);
15:    } }
```

A. `Elysia Sophia`
B. `Webby Olivia`
C. `Olivia Olivia`
D. `Olivia Sophia`
E. The code does not compile.
F. None of the above.

22. D. The code compiles and runs without issue, making option E incorrect. The `Child` class overrides the `setName()` method and hides the `static name` variable defined in the inherited `Person` class. Since variables are only hidden, not overridden, there are two distinct `name` variables accessible, depending on the location and reference type. Line 8 creates a `Child` instance, which is implicitly cast to a `Person` reference type on line 9. Line 10 uses the `Child` reference type, updating `Child.name` to `Elysia`. Line 11 uses the `Person` reference type, updating `Person.name` to `Sophia`. Lines 12 and 13 both call the overridden `setName()` instance method declared on line 6. This sets `Child.name` to `Webby` on line 12 and then to `Olivia` on line 13. The final values of `Child.name` and `Person.name` are `Olivia` and `Sophia`, respectively, making option D the correct answer.

!!!! In instance methods you could use static variables but not reversed.

23. QPZj -> B

24. 182943 -> B

24. What is printed by the following program?

```
1:  class Antelope {
2:     public Antelope(int p) {
3:        System.out.print("4");
4:     }
5:     { System.out.print("2"); }
6:     static { System.out.print("1"); }
7:  }
8:  public class Gazelle extends Antelope {
9:     public Gazelle(int p) {
10:       super(6);
11:       System.out.print("3");
12:    }
13:    public static void main(String hopping[]) {
14:       new Gazelle(0);
15:    }
16:    static { System.out.print("8"); }
17:    { System.out.print("9"); }
18: }
```

A. `182640`
B. `182943`
C. `182493`
D. `421389`
E. The code does not compile.
F. The output cannot be determined until runtime.

24. C. The code compiles and runs without issue, making options E and F incorrect. First, the class is initialized, starting with the superclass `Antelope` and then the subclass `Gazelle`. This involves invoking the `static` variable declarations and `static` initializers. The program first prints `1`, followed by `8`. Then we follow the constructor pathway from the object created on line 14 upward, initializing each class instance using a top-down approach. Within each class, the instance initializers are run, followed by the referenced constructors. The `Antelope` instance is initialized, printing `24`, followed by the `Gazelle` instance, printing `93`. The final output is `182493`, making option C the correct answer.

25. B,C

26. D, G

26. What is the output of the following code?

```
4:  public abstract class Whale {
5:     public abstract void dive();
6:     public static void main(String[] args) {
7:        Whale whale = new Orca();
8:        whale.dive(3);
9:     }
10: }
11: class Orca extends Whale {
12:    static public int MAX = 3;
```

26. What is the output of the following code?

```
4:  public abstract class Whale {
5:     public abstract void dive();
6:     public static void main(String[] args) {
7:        Whale whale = new Orca();
8:        whale.dive(3);
9:     }
10: }
11: class Orca extends Whale {
12:    static public int MAX = 3;
13:    public void dive() {
14:       System.out.println("Orca diving");
15:    }
16:    public void dive(int… depth) {
17:       System.out.println("Orca diving deeper "+MAX)
18: } }
```

A. `Orca diving`
B. `Orca diving deeper 3`
C. The code will not compile because of line 4.
D. The code will not compile because of line 8.
E. The code will not compile because of line 11.
F. The code will not compile because of line 12.
G. The code will not compile because of line 17.
H. None of the above.

14/26

Chapter7 Beyond Classes

Tuesday, October 14, 2025 12:01 PM

Declaring and Using an Interface

In Java, an interface is defined with the `interface` keyword, analogous to the `class` keyword used when defining a class. Refer to Figure 7.1 for a proper interface declaration.

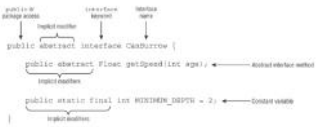


FIGURE 7.1 Defining an interface

Inheriting an Interface

Like an abstract class, when a concrete class inherits an interface, all of the inherited abstract methods must be implemented. We illustrate this principle in Figure 7.3. How many abstract methods does the concrete `Swan` class inherit?

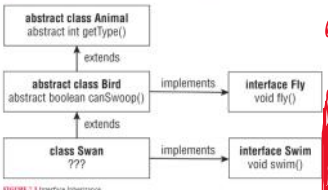


TABLE 7.1 Interface member types

	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	Class	—	`public static final`	Yes
`abstract` method	Instance	—	`public abstract`	No
`default` method	Instance	`default`	`public`	Yes
`static` method	Class	`static`	`public`	Yes
`private` method	Instance	`private`	—	Yes
`private static` method	Class	`private static`	—	Yes

```
public interface Schedule {
   default void wakeUp()        { checkTime(7);  }
   private void haveBreakFast() { checkTime(9);  }
   static void workOut()        { checkTime(18); }
   private static void checkTime(int hour) {
      if (hour > 17) {
         System.out.println("You're late!");
      } else {
         System.out.println("You have "+(17-hour)+" hours left "
               + "to make the appointment");
      }
   }
}
```

Using these rules, which of the following methods do not compile?

```
public interface ZooTrainTour {
   abstract int getTrainName();
   private static void ride() {}
   default void playHorn() { getTrainName(); ride(); }
   public static void slowDown() { playHorn(); }
   static void speedUp() { ride(); }
}
```

The `ride()` method is `private` and `static`, so it can be accessed by any default or static method within the interface declaration. The `getTrainName()` is `abstract`, so it can be accessed by a `default` method associated with the instance. The `slowDown()` method is `static`, though, and cannot call a `default` or `private` method, such as `playHorn()`, without an explicit reference object. Therefore, the `slowDown()` method does not compile.

Unlike a class, which can extend only one class, an interface can extend multipl interfaces.

```
public interface Nocturnal {
   public int hunt();
}

public interface CanFly {
   public void flap();
}

public interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements HasBigEyes {
   public int hunt() { return 5; }
   public void flap() { System.out.println("Flap!"); }
}
```

```
public interface CanRun {}
public class Cheetah extends CanRun {}   // DOES NOT COMPILE

public class Hyena {}
public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

Be wary of exam questions that mix class and interface declarations.

The following is an example of an incompatible declaration:

```
public interface Herbivore { public void eatPlants(int plantsLeft); }

public interface Omnivore  { public int eatPlants(int foodRemaining) }

public class Tiger implements Herbivore, Omnivore { // DOES NOT COMP
   // Doesn't matter!
}
```

The implementation of `Tiger` doesn't matter in this case since it's impossible to write a version of `Tiger` that satisfies both inherited `abstract` methods. The code does not compile, regardless of what is declared inside the `Tiger` class.

```
public interface Walk {
   public default int getSpeed() { return 5; }
}

public interface Run {
   public default int getSpeed() { return 10; }
}

public class Cat implements Walk, Run {}  // DOES NOT COMPILE
```

```
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   } }
```

This is an area where a `default` method `getSpeed()` exhibits properties of both an instance and `static` method. We use the interface name to indicate which method we want to call, but we use the `super` keyword to show that we are following instance inheritance, not class inheritance. Note that calling `Walk.this.getSpeed()` would not have worked. A bit confusing, we know, *but you need to be familiar with this syntax for the exam.*

`abstract int getTrainName();`
is an **abstract instance method that must be implemented by any class** implementing `ZooTrainTour`.

Even though it has no body, **it's still a perfectly valid method call** in code that will later be executed on an actual object that does implement it.

So in the line:

```
getTrainName();
```

inside `playHorn()`, we're saying:

> "When an object that implements this interface calls `playHorn()`, execute `getTrainName()` on that same object."

That's 100 % fine, because by the time `playHorn()` actually runs, the object is guaranteed to have a concrete implementation of `getTrainName()`.

ENUMS

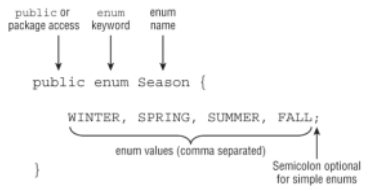


FIGURE 7.4 Defining a simple enum

```
System.out.print("begin,");
var firstCall = SeasonWithVisitors.SUMMER;   // Prints 4 times
System.out.print("middle,");
var secondCall = SeasonWithVisitors.SUMMER;  // Doesn't print anythi
System.out.print("end");
```

One thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season {} // DOES NOT COMPILE
```

```
for(var season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

The `ordinal()` method returns an `int` value, which denotes the order in which the value is declared in the enum:

```
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

```
if (Season.SUMMER == 2) {} // DOES NOT COMPILE
```

An enum provides a useful `valueOf()` method for converting from a `String` to

RULES!

The following list includes the implicit modifiers for interfaces that you need to know for the exam:

- Interfaces are implicitly `abstract`.
- Interface variables are implicitly `public`, `static`, and `final`.
- Interface methods without a body are implicitly `abstract`.
- Interface methods without the `private` modifier are implicitly `public`.

The last rule applies to `abstract`, `default`, and `static` interface methods, which we cover in the next section.

Default Interface Method Definition Rules

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is implicitly `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is implicitly `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

While Table 7.2 might seem like a lot to remember, here are some quick tips for the exam:

- Treat `abstract`, `default`, and non-`static` `private` methods as belonging to an instance of the interface.
- Treat `static` methods and variables as belonging to the interface class object.
- All `private` interface method types are only accessible within the interface declaration.

ENUMS

There are a few things to notice here. On line 23, the list of enum values ends with a semicolon (`;`). While this is optional for a simple enum, it is required if there is anything in the enum besides the values. Lines 25–33 are regular Java

Sealed classes

Specifying the Subclass Modifier

While some types, like interfaces, have a certain number of implicit modifiers, sealed classes do not. *Every class that directly extends a sealed class must specify exactly one of the following three modifiers:* `final`, `sealed`, or `non-sealed`. Remember this rule for the exam!

Location of direct subclasses	`permits` clause
In a different file from the sealed class	Required
In the same file as the sealed class	Permitted, but not required
Nested inside of the sealed class	Permitted, but not required

For this reason, interfaces that extend a sealed interface can only be marked `sealed` or `non-sealed`. They cannot be marked `final`.

Sealed Class Rules

- Sealed classes are declared with the `sealed` and `permits` modifiers.
- Sealed classes must be declared in the same package or named module as their direct subclasses.
- Direct subclasses of sealed classes must be marked `final`, `sealed`, or `non-sealed`. For interfaces that extend a sealed interface, only `sealed` and `non-sealed` modifiers are permitted.
- The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
- Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them.

Records

Members Automatically Added to Records

- **Constructor**: A constructor with the parameters in the same order as the record declaration
- **Accessor method**: One accessor for each field
- `equals()`: A method to compare two elements that returns `true` if each field is equal in terms of `equals()`
- `hashCode()`: A consistent `hashCode()` method using all of the fields
- `toString()`: A `toString()` implementation that prints each field of the record in a convenient, easy-to-read format

The first line of an overloaded constructor must be an explicit call to another constructor via `this()`. If there are no other constructors, the long constructor must be called. Contrast this with what you learned about in Chapter 6, where calling `super()` or `this()` was often optional in constructor declarations. Also, unlike compact constructors, you can only transform the data on the first line. After the first line, all of the fields will already be assigned, and the object is immutable.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
```

Tips

Another way to think of it is that a `private` interface method is only accessible to non-`static` methods defined within the interface. A `private static` interface method, on the other hand, can be accessed by any method in the interface. For both types of `private` methods, a class inheriting the interface cannot directly invoke them.

```
public sealed class Snake permits Cobra {  // DOES NOT COMPILE
   final class Cobra extends Snake {}
}
```

This code does not compile because `Cobra` requires a reference to the `Snake` namespace. The following fixes this issue:

```
public sealed class Snake permits Snake.Cobra {
   final class Cobra extends Snake {}
}
```

Records

Fun fact: it is legal to have a record without any fields. It is simply declared with the `record` keyword and parentheses:

```
public record Crane() {}
```

This is not the kind of thing you'd use in your own code, but it could come up on the exam.

```
System.out.print("begin,");
var firstCall = SeasonWithVisitors.SUMMER;   // Prints 4 times
System.out.print("middle,");
var secondCall = SeasonWithVisitors.SUMMER;  // Doesn't print anythi
System.out.print("end");
```

```
public enum SeasonWithTimes {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   FALL {
      public String getHours() { return "9am-5pm"; }
   };
   public abstract String getHours();
}
```

What's going on here? It looks like we created an `abstract` class and a bunch of tiny subclasses. In a way, we are. The enum itself has an `abstract` method. This means that each and every enum value is required to implement this method. If we forget to implement the method for one of the values, we get a compiler error:

```
The enum constant WINTER must implement the abstract method getHours
```

```
SPRING 1
SUMMER 2
FALL 3

if (Season.SUMMER == 2) {} // DOES NOT COMPILE
```

An enum provides a useful `valueOf()` method for converting from a `String` to an enum value. This is helpful when working with older code or parsing user input. The `String` passed in must match the enum value exactly, though.

```
Season s = Season.valueOf("SUMMER"); // SUMMER
Season t = Season.valueOf("summer"); // IllegalArgumentException
```

compact constructors, you can only transform the data on the first line. After the first line, all of the fields will already be assigned, and the object is immutable.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
      this(numberEggs + 1, firstName + " " + lastName);
      numberEggs = 10; // NO EFFECT (applies to parameter, not instar
      this.numberEggs = 20; // DOES NOT COMPILE
   }
}
```

Only the long constructor, with fields that match the record declaration, supports setting field values with a `this` reference. Compact and overloaded constructors do not.

For the exam, you should be aware of the following rules when working with pattern matching and records:

- If any field declared in the record is included, then all fields must be included.
- The order of fields must be the same as in the record.
- The names of the fields do not have to match.
- At compile time, the type of the field must be compatible with the type declared in the record.
- The pattern may not match at runtime if the record supports elements of various types.

- Overloaded and compact constructors
- Instance methods including overriding any provided methods (accessors, `equals()`, `hashCode()`, `toString()`)
- Nested classes, interfaces, annotations, enums, and records

SEALD CLASSES

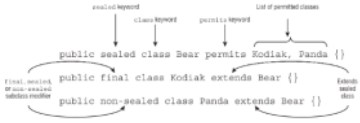


FIGURE 7.8 Defining a sealed class

Compiling Sealed Classes

Let's say we create a `Penguin` class and compile it in a new package without any other source code. With that in mind, does the following compile?

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}
```

No, it does not! Why? The answer is that a sealed class needs to be declared (and compiled) in the same package as its direct subclasses. But what about the subclasses themselves? They must each extend the sealed class. For example, the following two declarations do not compile:

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}  // DOES NOT COMPILE

// Emperor.java
package zoo;
public final class Emperor {}
```

```
// Sealed interface
public sealed interface Swims permits Duck, Swan, Floats {}

// Classes permitted to implement sealed interface
public final class Duck implements Swims {}
public final class Swan implements Swims {}

// Interface permitted to extend sealed interface
public non-sealed interface Floats extends Swims {}
```

Nested Classes

A *nested class* is a class that is defined within another class. A nested class can come in one of four flavors, with all supporting instance and `static` variables/members.

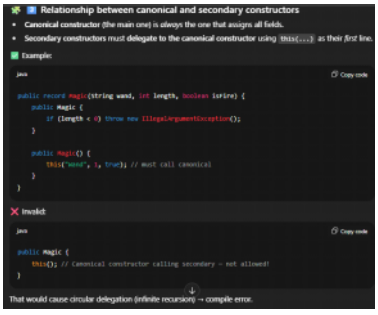
- *Inner class*: A non-`static` type defined at the member level of a class
- *Static nested class*: A `static` type defined at the member level of a class
- *Local class*: A class defined within a method body
- *Anonymous class*: A special case of a local class that does not have a name

Inner classes have the following properties:

- Can be declared `public`, `protected`, package, or `private`
- Can extend a class and implement interfaces
- Can be marked `abstract` or `final`
- Can access members of the outer class, including `private` members

Records

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String name) {}  // DOES NOT COMPILE
}
```



Local Classes

Local classes have the following properties:

- Do not have an access modifier.
- Can be declared `final` or `abstract`.
- Can include instance and `static` members.
- Have access to all fields and methods of the enclosing class (when defined in an instance method).
- Can access `final` and effectively final local variables.

Compact Constructors

Luckily, the authors of Java added the ability to define a compact constructor for records. A *compact constructor* is a special type of constructor used for records to process validation and transformations succinctly. It takes no parameters and implicitly sets all fields. Figure 7.7 shows an example of a compact constructor.

FIGURE 7.7 Declaring a compact constructor

While compact constructors can modify the constructor parameters, *they cannot modify the fields of the record.* For example, this does not compile:

```
public record Crane(int numberEggs, String name) {
   public Crane {
      this.numberEggs = 10;  // DOES NOT COMPILE
   }
}
```

Like enums, that means you can't extend or inherit a record.

```
public record BlueCrane() extends Crane {}  // DOES NOT COMPILE
```

Permitted modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`final`	Yes	Yes	Yes	No

```
if(c instanceof Couple(Bear a, Bear b)) {
   System.out.print(a.name() + " " + b.name());
}
if(c instanceof Couple(Bear(String firstName, List<String> f),
      Bear b)) {
   System.out.print(firstName + " " + b.name());
}
if(c instanceof Couple(Bear(String name, List<String> f1),
      Bear(String name, List<String> f2))) {
   System.out.print(name + " " + name);
}
```

```
public record Crane(int numberEggs, String name) {
   private static int TYPE = 10;
   public int size;                          // DOES NOT COMPILE
   private final boolean friendly = true;  // DOES NOT COMPILE
}
```

Records also do not support instance initializers. All initialization for the fields of a record must happen in a constructor. They do support `static` initializers, though.

```
public record Crane(int numberEggs, String name) {
  static { System.out.print("Hello Bird!"); }
  { System.out.print("Goodbye Bird!"); }  // DOES NOT COMPILE
  { this.name = "Big"; }                  // DOES NOT COMPILE
}
```

	Inner class	`static` nested class	Local class	Anonymous class
Can include instance and `static` members?	Yes	Yes	Yes	Yes
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of	N/A	N/A	Yes (if `final` or effectively	Yes (if `final` or effectively final)

Nested Classes

FIGURE 7.7 Declaring a compact constructor

```
if(c instanceof Couple(Bear a, Bear b)) {
   System.out.print(a.name() + " " + b.name());
}
if(c instanceof Couple(Bear(String firstName, List<String> f),
      Bear b)) {
   System.out.print(firstName + " " + b.name());
}
if(c instanceof Couple(Bear(String name, List<String> f1),
      Bear(String name, List<String> f2))) {
   System.out.print(name + " " + name);
}
```

```
public record Crane(int numberEggs, String name) {
   private static int TYPE = 10;
   public int size;                        // DOES NOT COMPILE
   private final boolean friendly = true;  // DOES NOT COMPILE
}
```

Records also do not support instance initializers. All initialization for the fields of a record must happen in a constructor. They do support `static` initializers, though.

```
public record Crane(int numberEggs, String name) {
  static { System.out.print("Hello Bird!"); }
  { System.out.print("Goodbye Bird!"); }  // DOES NOT COMPILE
  { this.name = "Big"; }                  // DOES NOT COMPILE
}
```

Nested Classes

Instantiating an Instance of an Inner Class

There is another way to instantiate `Room` that looks odd at first. OK, well, maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:    public static void main(String[] args) {
21:       var home = new Home();
22:       Room room = home.new Room();  // Create the inner class ins
23:       room.enter();
24:    }
```

Let's take a closer look at lines 21 and 22. We need an instance of `Home` to create a `Room`. We can't just call `new Room()` inside the `static main()` method, because Java won't know which instance of `Home` it is associated with. Java solves this by calling `new` as if it were a method on the `home` variable. We can shorten lines 21–23 to a single line:

```
21:       new Home().new Room().enter();  // Sorry, it looks ugly to
```

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);        // 30
9:              System.out.println(this.x);   // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

```
public class Fox {
   private class Den {}
   public void goHome() {
      new Den();
   }
   public static void visitFriend() {
      new Den();  // DOES NOT COMPILE
   }
}

public class Squirrel {
   public void visitFox() {
      new Den();  // DOES NOT COMPILE
   }
}
```

Let's take a look at an example:

```
1: public class Park {
2:    static class Ride {
3:       private int price = 6;
4:    }
5:    public static void main(String[] args) {
6:       var ride = new Ride();
7:       System.out.println(ride.price);
8: } }
```

Line 6 instantiates the nested class. Since the class is `static`, you do not need an instance of `Park` to use it. You are allowed to access `private` instance variables, as shown on line 7.

```
11: class Emu1 {
12:    String name = "Emmy";
13:    static Feathers createFeathers() {
14:       return new Feathers("grey");
15:    }
16:    record Feathers(String color) {
17:       void fly() {
18:          System.out.print(name + "  is flying");  // DOES NOT
19:       } } }
20:
21: class Emu2 {
22:    String name = "Emmy";
23:    static Feathers createFeathers() {
24:       return new Feathers("grey");  // DOES NOT COMPILE
25:    }
26:    class Feathers {
27:       void fly() {
28:          System.out.print(name + "  is flying");
29:       } } }
```

Line 14 compiles without issue because the record is implicitly `static`. Line 24 does not compile, though, as the class version of `Feathers` is not `static` and would require an instance of `Emu2` to create. Likewise, the outer variable, `name`, is only visible to the nested class if it is not `static`, as shown by line 28 compiling and line 18 not compiling.

Local Classes

	Inner class	`static` nested class	Local class	Anonymous class
Can include instance and `static` members?	Yes	Yes	Yes	Yes
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method?	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Polymorphism

- A reference with the same type as the object
- A reference that is a superclass of the object
- A reference of an interface the object implements or inherits

We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

We summarize these concepts into a set of rules for you to memorize for the exam:

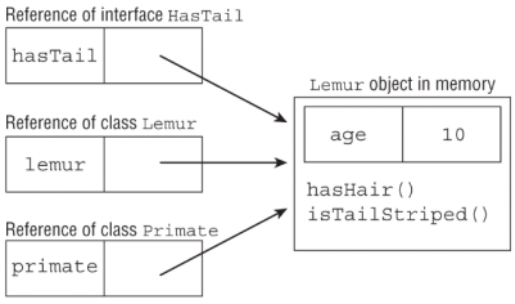
1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown.
4. The compiler disallows casts to unrelated types.

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. As an illustrative example, consider the following:

```
public void processData() {
   final int length = 5;
   int width = 10;
   int height = 2;
   class VolumeCalculator {
      public int multiply() {
         return length * width * height; // DOES NOT COMPILE
      }
   }
   width = 2;
}
```

The `length` and `height` variables are `final` and effectively final, respectively, so neither causes a compilation issue. On the other hand, the `width` variable is reassigned during the method, so it cannot be effectively final. For this reason, the local class declaration does not compile.

Polymorphism



Disallowed Casting

Let's try an example. Do you think the following program compiles?

```
1: interface Canine {}
2: interface Dog {}
3: class Wolf implements Canine {}
4:
5: public class BadCasts {
6:    public static void main(String[] args) {
7:       Wolf wolfy = new Wolf();
8:       Dog badWolf = (Dog)wolfy;
9:    } }
```

This limitation aside, the compiler can enforce one rule around interface casting. The compiler does not allow a cast from an interface reference to an object reference if the object type cannot possibly implement the interface, such as if the class is marked `final`. For example, what if we changed line 3 of our previous code?

```
3: final class Wolf implements Canine {}
```

Line 8 no longer compiles. The compiler recognizes that there are no possible subclasses of `Wolf` capable of implementing the `Dog` interface.

EXAM

1. ~~A~~B,E, D

1. Which of the following are valid record declarations? (Choose all that apply)

```
public record Iguana(int age) {
   private static final int age = 10; }

public final record Gecko() {}

public abstract record Chameleon()  {
   private static String name; }

public record BeardedDragon(boolean fun) {
   @Override public boolean fun() { return false; } }

public record Reptile(long size) {
   public Reptile {
      if(size == 1) throw new IllegalArgumentException();
   } }

public record Newt(double age) extends Reptile {
   public Newt(double age) {
      age = this.age % 2 == 0 ? 5 : 10;
   } }
```

A. Iguana
B. Gecko
C. Chameleon
D. BeardedDragon
E. Reptile
F. Newt
G. None of the above

1. B, D, E. `Iguana` does not compile, as it declares a `static` field with the same name as an instance field. Records are implicitly `final` and cannot be marked `abstract`, which is why `Gecko` compiles and `Chameleon` does not, making option B correct. Notice in `Gecko` that records are not required to declare any fields. `BeardedDragon` also compiles, as records may override any accessor methods, making option D correct. `Reptile` compiles as it contains a valid compact constructor, making option E correct. `Newt` does not compile because it cannot extend another record. It also does not compile because the compact constructor tries to read `this.age`, which is not permitted.

2. A,B,D,E
3. ~~C~~ D

3. What is the result of the following program?

```
11: public class Favorites {
```

!!ENUM construtors have implicit constructors private.

F. Next
G. None of the above

2. A,B,D,E
3. ~~C~~, D

3. What is the result of the following program?

```
11: public class Favorites {
12:    enum Flavors {
13:       VANILLA, CHOCOLATE, STRAWBERRY
14:       public Flavors() {}
15:    }
16:    public static void main(String[] args) {
17:       for(final var e : Flavors.values())
18:          System.out.print((e.ordinal() % 2) + " ");
19:    } }
```

!!ENUM construtors have implicit constructors private.

Line 14 is incorrect

A. `0 1 0`
B. `1 0 1`
C. Exactly one line of code does not compile.
D. More than one line of code does not compile.
E. The code compiles but produces an exception at runtime.
F. None of the above.

4. ~~B~~, C

4. What is the output of the following program?

```
public sealed class ArmoredAnimal permits Armadillo {
   public ArmoredAnimal(int size) {}
   @Override public String toString() { return "Strong"; }
   public static void main(String[] a) {
      var c = new Armadillo(10, null);
      System.out.println(c);
   }
}
class Armadillo extends ArmoredAnimal {
   @Override public String toString() { return "Cute"; }
   public Armadillo(int size, String name) {
      super(size);
   }
}
```

!!A class which extends seald must be : final or sealed or non-sealed.

A. `Strong`
B. `Cute`
C. The program does not compile.
D. The code compiles but produces an exception at runtime.
E. None of the above.

5. ~~A~~, E

5. Which statement about the following program is correct?

```
1:  interface HasExoskeleton {
2:     double size = 2.0f;
3:     abstract int getNumberOfSections();
4:  }
5:  abstract class Insect implements HasExoskeleton {
6:     abstract int getNumberOfLegs();
7:  }
8:  public class Beetle extends Insect {
9:     int getNumberOfLegs() { return 6; }
10:    int getNumberOfSections(int count) { return 1; }
11: }
```

!! If subclasses don't implement abstract methods, the error will occur at declaration class.
(Line 8)

A. It compiles without issue.
B. The code will produce a `ClassCastException` if called at runtime.
C. The code will not compile because of line 2.
D. The code will not compile because of line 5.
E. The code will not compile because of line 8.
F. The code will not compile because of line 10.

6. D,E
7. ~~B~~ E

7. What is the output of the following program?

```
1: interface Aquatic {
2:    int getNumOfGills(int p);
3: }
4: public class ClownFish implements Aquatic {
5:    String getNumOfGills() { return "14"; }
6:    int getNumOfGills(int input) { return 15; }
7:    public static void main(String[] args) {
8:       System.out.println(new ClownFish().getNumOfGills(-1));
9: } }
```

7. E. The inherited interface method `getNumOfGills(int)` is implicitly `public`; therefore, it must be declared `public` in any concrete class that implements the interface. Since the method uses the package (default) modifier in the `ClownFish` class, line 6 does not compile, making option E the correct answer. If the method declaration were corrected to include `public` on line 6, then the program would compile and print `15` at runtime, and option B would be the correct answer.

A. `14`
B. `15`
C. The code will not compile because of line 4.
D. The code will not compile because of line 5.
E. The code will not compile because of line 6.
F. None of the above.

8. ~~C~~,D,E ~~B~~ G

8. Given the following, select the statements that can be inserted into the blank line so that the code will compile and print `true` at runtime? (Choose all that apply.)

```
record Walrus(List<String> diet) {}
record Exhibit(Walrus animal, String location) {}

var e = new Exhibit(new Walrus(List.of("Wally")), "Artic");
System.out.print(e instanceof _____);
```

8. B, E, G. Options A and F do not compile because they are not compatible with `List<String>`. Option C does not compile because the reference type of `w` is `Object`, which doesn't have an `animal()` method. Option D does not compile because the variable `i` is used twice in the same pattern matching statement. Option H does not compile because you can't use `null` in a pattern matching statement. Options B, E, and G correctly compile and print `true` at runtime.

A. `Exhibit(Walrus(List<Integer> z), Object a)`
B. `Exhibit(Walrus(List m), Object n)`
C. `Object w && w.animal().diet().size() == 0`
D. `Exhibit(Walrus(var i), var i)`
E. `Exhibit(var p, var q)`
F. `Exhibit(List<?> g, var h)`
G. `Exhibit(var x, CharSequence y)`
H. `Exhibit(Walrus(null), var v)`
I. None of the above

9. A,E,F
10. G-> A,B,C,E (The grill is wrong)

10. What types can be inserted in the blanks on the lines marked `X` and `Z` that allow the code to compile? (Choose all that apply.)

```
interface Walk { private static List move() { return null; } }
interface Run extends Walk { public ArrayList move(); }
class Leopard implements Walk {
   public _____ move() {  // X
      return null;
   }
}
class Panther implements Run {
   public  move(_____) {  // Z
      return null;
   }
}
```

A. `Integer` on the line marked `X`
B. `ArrayList` on the line marked `X`
C. `List` on the line marked `X`
D. `List` on the line marked `Z`
E. `ArrayList` on the line marked `Z`
F. None of the above, since the `Run` interface does not compile.
G. Does not compile for a different reason.

11. B

12. A,F,E

E. `ArrayList` on the line marked `2`
F. None of the above, since the `Run` interface does not compile.
G. Does not compile for a different reason.

11. B

12. A,F,E 0

12. Which variables or members are accessible from within the `hiss()` method? (Choose all that apply.)

```
13: public class BoaConstrictor {
14:    private Body body;
15:    BoaConstrictor(Body b) { this.body = b; }
16:    private long tail = 10;
17:    record Body(int stripes) {
18:       private static int counter = 0;
19:       int counter() { return counter; }
20:       Body {
21:          stripes = stripes + counter++;
22:       }
23:       private void hiss() {} } }
```

A. `counter()`
B. `tail`
C. `body`
D. `stripes()`
E. `stripes`
F. `counter`
G. Line 15 does not compile.
H. Line 17 does not compile.
I. Lines 20–22 do not compile.

Two key facts

1. **Record accessors are auto-generated.**
   For `record Body(int stripes)`, the compiler generates a public method `int stripes()` automatically. That's where `stripes()` comes from— you don't see it written, but it exists.
2. **Nested records are implicitly `static`.**
   Since `Body` is a nested record, it is implicitly `static`. A `static` nested type **cannot access instance members of the enclosing class** (like `tail` and `body`) without an explicit `BoaConstrictor` instance.

13. E G

13. What is the result of the following program?

```
public class Weather {
   enum Seasons {
      WINTER, SPRING, SUMMER, FALL
   }

   public static void main(String[] args) {
      Seasons w = null;
      switch (w) {
         case Seasons.SPRING -> System.out.print("s");
         case Seasons.WINTER -> System.out.print("w");
         case Seasons.SUMMER -> System.out.print("m");
         default -> System.out.println("missing data"); }
   } }
```

It prints an NullPointerException!!

A. `s`
B. `w`
C. `m`
D. `missing data`
E. Exactly one line of code does not compile.
F. More than one line of code does not compile.
G. The code compiles but produces an exception at runtime.

14. A,C,D,E

14. Which statements about sealed classes are correct? (Choose all that apply.)
A. A sealed interface restricts which subinterfaces may extend it.
B. A sealed class cannot be indirectly extended by a class that is not listed in its `permits` clause.
C. A sealed class can be extended by an `abstract` class.
D. A sealed class can be extended by a subclass that uses the `non-sealed` modifier.
E. A sealed interface restricts which subclasses may implement it.
F. A sealed class cannot contain any nested subclasses.
G. None of the above.

14. A, C, E. A sealed interface restricts which interfaces may extend it, or which classes may implement it, making options A and E correct. Option B is incorrect. For example, a `non-sealed` subclass allows classes not listed in the `permits` clause to indirectly extend the sealed class. Option C is correct. While a sealed class is commonly extended by a subclass marked `final`, it can also be extended by a sealed or `non-sealed` subclass marked `abstract`. Option D is incorrect, as the modifier is `non-sealed`, not `unsealed`. Finally, option F is incorrect, as sealed classes can contain nested subclasses.

15. B,C

15. Which line allows the code to print `Not scared` at runtime?

```
public class Ghost {
   public static void boo() {
      System.out.println("Not scared");
   }
   protected final class Spirit {
      public void boo() {
         System.out.println("Booo!!!");
      }
   }
   public static void main(String… haunt) {
      var g = new Ghost().new Spirit();
      ____________________________;
   } }
```

A. `g.boo()`
B. `g.super.boo()`
C. `new Ghost().boo()`
D. `g.Ghost.boo()`
E. `new Spirit().boo()`
F. None of the above

15. F. Trick question—the code does not compile! For this reason, option F is correct. The `Spirit` class is marked `final`, so it cannot be extended. The `main()` method uses an anonymous class that inherits from `Spirit`, which is not allowed. If `Spirit` were not marked `final`, then option C would be correct. Option A would print `Booo!!!`, while options B, D, and E would not compile for various reasons.

16. C E

16. The following code appears in a file named `Ostrich.java`. What is the result of compiling the source file?

```
1: public class Ostrich {
2:    private int count;
3:    static class OstrichWrangler {
4:       public int stampede() {
5:          return count;
6:       } } }
```

A. The code compiles successfully, and one bytecode file is generated: `Ostrich.class`.
B. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `OstrichWrangler.class`.
C. The code compiles successfully, and two bytecode files are generated: `Ostrich.class` and `Ostrich$OstrichWrangler.class`.
D. A compiler error occurs on line 3.
E. A compiler error occurs on line 5.

16. E. The `OstrichWrangler` class is a `static` nested class; therefore, it cannot access the instance member `count`. For this reason, line 5 does not compile, and option E is correct.

17. E,G
18. E
19. F
20. D, H

20. What is the output of this code?

```
13: record Gorilla(int x, Double y) {
14:    Gorilla {}
15:    Gorilla() { this(1,2.0); }
16: }
17: record Family(Gorilla parent1, Gorilla parent2) {}
18:
19: var family = new Family(
20:    new Gorilla(1, null), new Gorilla(0, 1.2));
21: System.out.print(switch (family) {
22:    case Family(var a, var b) -> "1";
23:    case Family(Gorilla c, Gorilla (int d, double e)) -> "2";
24:    case Family(Gorilla (int f, Double g), var h) -> "3";
25:    case Family(Gorilla i, Gorilla (int j, Double k)) -> "4";
26:    case Family(Object m, Object n) -> "5";
27:    case null -> "6";
28:    default -> "7";
29: });
```

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. None of the above

20. H. The record declarations compile but the `switch` expression does not, making option H correct. First, the second `case` statement does not compile, as `double` is not compatible with `Double`. Next, the pattern matching `case` statement on line 23 dominates the ones on lines 23–25. If three of them were to be removed (including the second one), then the code would compile and print the value associated with the remaining one.

21. F
22. C,D

22. Which of the following can be inserted in the `rest()` method? (Choose all that apply.)

```
public class Lion {
   class Cub {}
   static class Den {}
   static void rest() {
      _____;
   } }
```

A. `Cub a = Lion.new Cub()`
B. `Lion.Cub b = new Lion().Cub()`
C. `Lion.Cub c = new Lion().new Cub()`
D. `var d = new Den()`
E. `var e = Lion.new Cub()`
F. `Lion.Den f = Lion.new Den()`
G. `Lion.Den g = new Lion.Den()`
H. `var h = new Cub()`

23. D
24. B,D,E -> Why D is not correct?
25. F

25. What does the following program print?

```
1:  public class Zebra {
2:     private int x = 24;
3:     public int hunt() {
4:        String message = "x is ";
5:        abstract class Stripes {
6:           private int x = 0;
7:           public void print() {
8:              System.out.print(message + Zebra.this.x);
9:           }
10:       }
11:       var s = new Stripes() {};
12:       s.print();
13:       return x;
14:    }
15:    public static void main(String[] args) {
16:       new Zebra().hunt();
17:    } }
```

A. `x is 0`
B. `x is 24`
C. Line 6 generates a compiler error.
D. Line 8 generates a compiler error.
E. Line 11 generates a compiler error.
F. None of the above.

25. B. `Zebra.this.x` is the correct way to refer to `x` in the `Zebra` class. Line 5 defines an abstract local class within a method. While line 11 defines a concrete anonymous class that extends the `Stripes` class. The code compiles without issue and prints `x is 24` at runtime, making option B the correct answer.

26. C
27. B,C,D,G
28. A,B,D
29. F

29. How many lines of the following program contain a compilation error?

```
1:  class Primate {
2:     protected int age = 2;
3:     { age = 1; }
4:     public Primate() {
5:        this().age = 3;
6:     }
7:  }
8:  public class Orangutan {
9:     protected int age = 4;
10:    { age = 5; }
11:    public Orangutan() {
12:       this().age = 6;
13:    }
14:    public static void main(String[] bananas) {
15:       final Primate x = (Primate)new Orangutan();
16:       System.out.println(x.age);
17:    }
18: }
```

A. None, and the program prints `1` at runtime.
B. None, and the program prints `3` at runtime.
C. None, but it causes a `ClassCastException` at runtime.
D. 1
E. 2
F. 3

30. X E, C

30. Assuming the following classes are declared as top-level types in the same file, which classes contain compiler errors? (Choose all that apply.)

```
sealed class Bird {
   public final class Flamingo extends Bird {}
}

sealed class Monkey {}

class EmperorTamarin extends Monkey {}

non-sealed class Mandrill extends Monkey {}

sealed class Friendly extends Mandrill permits Silly {}

final class Silly {}
```

A. `Bird`
B. `Monkey`
C. `EmperorTamarin`
D. `Mandrill`
E. `Friendly`
F. `Silly`
G. All of the classes compile without issue.

30. C, E. `Bird` and its nested `Flamingo` subclass compile without issue. The `permits` clause is optional if the subclass is nested or declared in the same file. For this reason, `Monkey` and its subclass `Mandrill` also compile without issue. `EmperorTamarin` does not compile, as it is missing a `non-sealed`, `sealed`, or `final` modifier, making option C correct. `Friendly` also does not compile since it lists a subclass `Silly` that does not extend it, making option E correct. While the `permits` clause is optional, the `extends` clause is not. `Silly` compiles just fine. Even though it does not extend `Friendly`, the compiler error is in the sealed class.

Chapter8

Friday, October 17, 2025 9:21 AM

Functional Interfaces

```
@FunctionalInterface  // DOES NOT COMPILE
public interface Dance {
   void move();
   void rest();
}
```

Java includes `@FunctionalInterface` on some, but not all, functional interfaces. This annotation means the authors of the interface promise it will be safe to use in a lambda in the future. However, just because you don't see the annotation doesn't mean it's not a functional interface. Remember that having exactly one abstract method is what makes it a functional interface, not the annotation.

Let's take a look at an example. Is the `Soar` class a functional interface?

```
public interface Soar {
   abstract String toString();
}
```

It is not. Since `toString()` is a `public` method implemented in `Object`, it does not count toward the single abstract method test. On the other hand, the following implementation of `Dive` is a functional interface:

```
public interface Dive {
   String toString();
   public boolean equals(Object o);
   public abstract int hashCode();
   public void dive();
}
```

The `dive()` method is the single abstract method, while the others are not counted since they are `public` methods defined in the `Object` class.

While all method references can be turned into lambdas, the opposite is not always true. For example, consider this code:

```
var str = "";
StringChecker lambda = () -> str.startsWith("Zoo");
```

How might we write this as a method reference? You might try one of the following:

```
StringChecker methodReference = str::startsWith;        // DOES NOT
StringChecker methodReference = str::startsWith("Zoo"); // DOES NOT
```

Neither of these works! While we can pass the `str` as part of the method reference, there's no way to pass the `"Zoo"` parameter with it. Therefore, it is not possible to write this lambda as a method reference.

Calling Instance Methods on a Parameter

This time, we are going to call the same instance method that doesn't take any parameters. The trick is that we will do so without knowing the instance in advance. We need a different functional interface this time since it needs to know about the `String`.

```
interface StringParameterChecker {
   boolean check(String text);
}
```

We can implement this functional interface as follows:

```
23: StringParameterChecker methodRef = String::isEmpty;
24: StringParameterChecker lambda = s -> s.isEmpty();
25:
26: System.out.println(methodRef.check("Zoo"));  // false
```

You can even combine the two types of instance method references. Again, we need a new functional interface that takes two parameters.

```
interface StringTwoParameterChecker {
   boolean check(String text, String prefix);
}
```

Pay attention to the parameter order when reading the implementation.

```
26: StringTwoParameterChecker methodRef = String::startsWith;
27: StringTwoParameterChecker lambda = (s, p) -> s.startsWith(p);
28:
29: System.out.println(methodRef.check("Zoo", "A"));  // false
```

Since the functional interface takes two parameters, Java has to figure out what they represent. The first one will always be the instance of the object for instance methods. Any others are to be method parameters.

Calling Constructors

A *constructor reference* is a special type of method reference that uses `new` instead of a method and instantiates an object. For this example, our functional interface will not take any parameters but will return a `String`.

```
interface EmptyStringCreator {
   String create();
}
```

To call this, we use `new` as if it were a method name.

```
30: EmptyStringCreator methodRef = String::new;
31: EmptyStringCreator lambda = () -> new String();
32:
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
```

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;
```

RULES

A method reference and a lambda behave the same way at runtime. You can pretend the compiler turns your method references into lambdas for you.

There are four formats for method references.

- `static` methods
- Instance methods on a particular object
- Instance methods on a parameter to be determined at runtime
- Constructors

TABLE 8.3 Method references

Type	Before colon	After colon	Example
`static` methods	Class name	Method name	`Math::random`
Instance methods on a particular object	Instance variable name	Method name	`str::startsWith`
Instance methods on a parameter	Class name	Method name	`String::isEmpty`
Constructor	Class name	`new`	`String::new`

TABLE 8.4 Common functional interfaces

Functional interface	Return type	Method name	# of parameters
`Supplier<T>`	`T`	`get()`	0
`Consumer<T>`	`void`	`accept(T)`	1 (T)
`BiConsumer<T, U>`	`void`	`accept(T,U)`	2 (T, U)
`Predicate<T>`	`boolean`	`test(T)`	1 (T)
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 (T, U)
`Function<T, R>`	`R`	`apply(T)`	1 (T)
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 (T, U)
`UnaryOperator<T>`	`T`	`apply(T)`	1 (T)
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 (T, T)

TABLE 8.5 Rules for accessing a variable from a lambda body inside a method

Variable type	Rule
Instance variable	Allowed
Static variable	Allowed
Local variable	Allowed if `final` or effectively final
Method parameter	Allowed if `final` or effectively final
Lambda parameter	Allowed

Tips

Now let's try another one. Do you see what's wrong here?

```
(a, b) -> { int a = 0; return 5; }    // DOES NOT COMPILE
```

EXAM

1.A
2.c
3.A,C
4.A,F
5.A,C,D

5. Which of the following functional interfaces contain an abstract method that returns a primitive value? (Choose all that apply.)
   A. `BooleanSupplier`
   B. `CharSupplier`
   C. `DoubleSupplier`
   D. `FloatSupplier`
   E. `IntSupplier`
   F. `StringSupplier`

5. A, C, E. Java includes support for three primitive streams, along with numerous functional interfaces to go with them: `int`, `double`, and `long`. For this reason, options C and E are correct. Additionally, there is a `BooleanSupplier` functional interface, making option A correct. Java does not include primitive streams or related functional interfaces for other numeric data types, making options B and D incorrect. Option F is incorrect because `String` is not a primitive but an object. Only primitives have custom suppliers.

6.A,C
7.E
8.E
9.A,C,F

9. Which statements are true? (Choose all that apply.)
   A. The `Consumer` interface is good for printing out an existing value.
   B. The `Supplier` interface is good for printing out an existing value.
   C. The `IntegerSupplier` interface returns an `int`.
   D. The `Predicate` interface returns an `int`.
   E. The `Function` interface has a method named `test()`.
   F. The `Predicate` interface has a method named `test()`.

10.C

10. Which of the following can be inserted without causing a compilation error? (Choose all that apply.)

```
public void remove(List<Character> chars) {
   char end = 'z';
   Predicate<Character> predicate = c -> {
      char start = 'a'; return start <= c && c <= end; };

   // INSERT LINE HERE
}
```

A `char start = 'a';`

10. A, B, C. Since the scope of `start` and `c` is within the lambda, the variables can be declared or updated after it without issue, making options A, B, and C correct. Option D is incorrect because setting `end` prevents it from being effectively final.

```
33: var myString = methodRef.create();
34: System.out.println(myString.equals("Snake"));  // false
```

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3));   // 8
```

PARAMETER LIST FORMATS

You have three formats for specifying parameter types within a lambda: without types, with types, and with `var`. The compiler requires all parameters in the lambda to use the same format. Can you see why the following are not valid?

```
5: (var x, y) -> "Hello"                // DOES NOT COMPILE
6: (var x, Integer y) -> true           // DOES NOT COMPILE
7: (String x, var y, Integer z) -> true  // DOES NOT COMPILE
8: (Integer x, y) -> "goodbye"          // DOES NOT COMPILE
```

Lines 5 needs to remove `var` from `x` or add it to `y`. Next, lines 6 and 7 need to use the type or `var` consistently. Finally, line 8 needs to remove `Integer` from `x` or add a type to `y`.

It gets even more interesting when you look at where the compiler errors occur when the variables are not effectively final.

```
2:  public class Crow {
3:     private String color;
4:     public void caw(String name) {
5:        String volume = "loudly";
6:        name = "Caty";
7:        color = "black";
8:
9:        Consumer<String> consumer = s ->
10:          System.out.println(name + " says "       // DOES NOT COM
11:             + volume + " that she is " + color);  // DOES NOT COM
12:       volume = "softly";
13:    }
14: }
```

```
   Predicate<Character> predicate = c -> {
      char start = 'a'; return start <= c && c <= end; };

   // INSERT LINE HERE
}
```

A. `char start = 'a';`
B. `char c = 'x';`
C. `chars = null;`
D. `end = 'i';`
E. None of the above

11.D

12.a

13. E

13. Which is true of the following code?

```
int length = 3;

for (int i = 0; i<3; i++) {
   if (i%2 == 0) {
      Supplier<Integer> supplier = () -> length; // A
      System.out.println(supplier.get());        // B
   } else {
      int j = i;
      Supplier<Integer> supplier = () -> j;      // C
      System.out.println(supplier.get());        // D
   }
}
```

A. The first compiler error is on line `A`.
B. The first compiler error is on line `B`.
C. The first compiler error is on line `C`.
D. The first compiler error is on line `D`.
E. The code compiles successfully.

13. E. Lambdas are only allowed to reference `final` or effectively final variables. You can tell the variable `j` is effectively final because adding a `final` keyword before it wouldn't introduce a compiler error. Each time the `else` statement is executed, the variable is redeclared and goes out of scope. Therefore, it is not reassigned. Similarly, `length` is effectively final. There are no compiler errors, and option E is correct.

14.B,D,E

14. Which of the following are valid lambda expressions? (Choose all that apply.)

A. `(Wolf w, var c) -> 39`
B. `(final Camel c) -> {}`
C. `(a,b,c) -> {int b = 3; return 2;}`
D. `(x,y) -> new RuntimeException()`
E. `(var y) -> return 0;`
F. `() -> {float f}`
G. `(Cat a, b) -> {}`

14. B, D. Option B is a valid functional interface, one that could be assigned to a `Consumer<Camel>` reference. Notice that the `final` modifier is permitted on variables in the parameter list. Option D is correct, as the exception is being returned as an object and not thrown. This would be compatible with a `BiFunction` that included `RuntimeException` as its return type.
Options A and G are incorrect because they mix format types for the parameters. Option C is invalid because the variable `b` is used twice. Option E is incorrect, as a `return` statement is permitted only inside braces (`{}`). Option F is incorrect because the variable declaration requires a semicolon (`;`) after it.

15.F

15. Which lambda expression, when entered into the blank line in the following code, causes the program to print `hahaha`? (Choose all that apply.)

```
import java.util.function.Predicate;
public class Hyena {
   private int age = 1;
   public static void main(String[] args) {
      var p = new Hyena();
      double height = 10;
      int age = 1;
      testLaugh(p, _____________);
      age = 2;
   }
   static void testLaugh(Hyena panda, Predicate<Hyena> joke) {
      var r = joke.test(panda) ? "hahaha" : "silence";
      System.out.print(r);
   }
}
```

A. `p -> p.age <= 10`
B. `shenzi -> age==1`
C. `p -> true`
D. `age==1`
E. `shenzi -> age==2`
F. `h -> h.age < 5`
G. None of the above, as the code does not compile

15. A, F. Option A is a valid lambda expression. While `main()` is a `static` method, it can access `age` since it is using a reference to an instance of `Hyena`, which is effectively final in this method. Since `var` is not a reserved word, it may be used for variable names. Option F is also correct, with the lambda variable being a reference to a `Hyena` object. The variable is processed using deferred execution in the `testLaugh()` method.
Options B and E are incorrect; since the local variable `age` is not effectively final, this would lead to a compilation error. Option C would also cause a compilation error, since the expression uses the variable name `p`, which is already declared within the method. Finally, option D is incorrect, as this is not even a lambda expression.

16.c

17.c

18.b,f,g

19.a F

19. Which of the following compiles and prints out the entire set?

```
Set<?> set = Set.of("lion", "tiger", "bear");
var s = Set.copyOf(set);
Consumer<Object> consumer = _____________;
s.forEach(consumer);
```

A. `() -> System.out.println(s)`
B. `s -> System.out.println(s)`
C. `(s) -> System.out.println(s)`
D. `System.out.println(s)`
E. `System::out::println`
F. `System.out::println`

19. F. While there is a lot in this question trying to confuse you, note that there are no options about the code not compiling. This allows you to focus on the lambdas and method references. Option A is incorrect because a `Consumer` requires one parameter. Options B and C are close. The syntax for the lambda is correct. However, `s` is already defined as a local variable, and therefore the lambda can't redefine it. Options D and E use incorrect syntax for a method reference. Option F is correct.

20.g E

20. Which lambda can replace the `new Sloth()` call in the `main()` method and produce the same output at runtime?

```
import java.util.List;
interface Yawn {
   String yawn(double d, List<Integer> time);
}
class Sloth implements Yawn {
   public String yawn(double zzz, List<Integer> time) {
      return "Sleep: " + zzz;
   } }
public class Vet {
   public static String takeNap(Yawn y) {
      return y.yawn(10, null);
   }
   public static void main(String… unused) {
      System.out.print(takeNap(new Sloth()));
   } }
```

A. `(z,f) -> { String x = ""; return "Sleep: " + x }`
B. `(t,s) -> { String t = ""; return "Sleep: " + t; }`
C. `(w,q) -> {"Sleep: " + w}`
D. `(e,u) -> { String g = ""; "Sleep: " + e }`
E. `(a,b) -> "Sleep: " + (double)(b==null ? a : a)`
F. `(r,k) -> { String g = ""; return "Sleep:"; }`
G. None of the above, as the program does not compile

20. E. Option A does not compile because the second statement within the block is missing a semicolon (`;`) at the end. Option B is an invalid lambda expression because `t` is defined twice: in the parameter list and within the lambda expression. Options C and D are both missing a `return` statement and semicolon. Options E and F are both valid lambda expressions, although only option E matches the behavior of the `Sloth` class. In particular, option F only prints `Sleep:`, not `Sleep: 10.0`.

21e,d,f A

21. Which of the following are valid functional interfaces? (Choose all that apply.)

```
public interface Transport {
   public int go();
   public boolean equals(Object o);
}

public abstract class Car {
   public abstract Object swim(double speed, int duration);
}

public interface Locomotive extends Train {
   public int getSpeed();
}
```

21. A, E, F. A valid functional interface is one that contains a single abstract method, excluding any `public` methods that are already defined in the `java.lang.Object` class. `Transport` and `Boat` are valid functional interfaces, as they each contain a single abstract method: `go()` and `hashCode(String)`, respectively. This gives us options A and E. Since the other methods are part of `Object`, they do not count as abstract methods. `Train` is also a functional interface since it extends `Transport` and does not define any additional abstract methods. This adds option F as the final correct answer.
`Car` is not a functional interface because it is an abstract class. `Locomotive` is not a functional interface because it includes two abstract methods, one of which is inherited. Finally, `Spaceship` is not a valid interface, let alone a functional interface, because a `default` method must provide a body. A quick way

```
public abstract class Car {
   public abstract Object swim(double speed, int duration);
}

public interface Locomotive extends Train {
   public int getSpeed();
}

public interface Train extends Transport {}

abstract interface Spaceship extends Transport {
   default int blastOff();
}

public interface Boat {
   int hashCode();
   int hashCode(String input);
}
```

A. `Boat`
B. `Car`
C. `Locomotive`
D. `Spaceship`
E. `Transport`
F. `Train`
G. None of these is a valid functional interface.

also a functional interface since it extends `Transport` and does not define any additional abstract methods. This adds option F as the final correct answer. `Car` is not a functional interface because it is an abstract class. `Locomotive` is not a functional interface because it includes two abstract methods, one of which is inherited. Finally, `Spaceship` is not a valid interface, let alone a functional interface, because a `default` method must provide a body. A quick way to test whether an interface is a functional interface is to apply the `@FunctionalInterface` annotation and check if the code still compiles.

Chapter9

Friday, October 17, 2025 3:53 PM

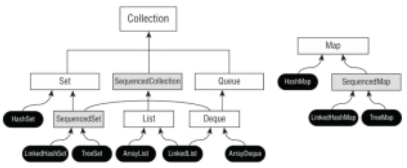


FIGURE 9.1 Java Collections Framework

```
16: String[] array = new String[] {"a", "b", "c"};
17: List<String> asList = Arrays.asList(array);   // [a, b, c]
18: List<String> of = List.of(array);             // [a, b, c]
19: List<String> copy = List.copyOf(asList);      // [a, b, c]
20:
21: array[0] = "c";
22:
23: System.out.println(asList);                   // [c, b, c]
24: System.out.println(of);                       // [a, b, c]
25: System.out.println(copy);                     // [a, b, c]
26:
27: asList.set(0, "a");
28: System.out.println(Arrays.toString(array)); // [a, b, c]
```

Line 17 creates a `List` that is backed by an array. Line 21 changes the array, and line 23 reflects that change. Lines 27 and 28 show the other direction where changing the `List` updates the underlying array. Lines 18 and 19 each create an immutable `List`.

When run independently, the following shows both types are immutable by throwing an exception when trying to set a value.

```
of.set(0, "y");   // UnsupportedOperationException
copy.set(0, "y");  // UnsupportedOperationException
```

Going back to our `Rabbit` that does not implement `Comparable`, we try to add it to a `TreeSet`:

```
2:  public class UseTreeSet {
3:     record Rabbit(int id) {}
4:     public static void main(String[] args) {
5:        Set<Duck> ducks = new TreeSet<>();
6:        ducks.add(new Duck("Puddles"));
7:
8:        Set<Rabbit> rabbits = new TreeSet<>();
9:        rabbits.add(new Rabbit(1));  // ClassCastException
10: } }
```

Line 6 is fine. `Duck` does implement `Comparable`. `TreeSet` is able to sort it into the proper position in the set. Line 9 is a problem. When `TreeSet` tries to sort it, Java discovers the fact that `Rabbit` does not implement `Comparable`. Java throws an exception that looks like this:

```
Exception in thread "main" java.lang.ClassCastException:
   class UseTreeSet$Rabbit cannot be cast to class java.lang.Comparable
```

```
var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>(list1);
var list3 = new ArrayList<String>(10);
```

The first two are the common constructors you need to know for all `Collection`s. The final example says to create an `ArrayList` containing a specific number of slots, but again not to assign any. You can think of this as the size of the underlying array.

This gets tricky when you have an `Integer` type. What do you think the following prints?

```
31: var list = new LinkedList<Integer>();
32: list.add(3);
33: list.add(2);
34: list.add(1);
35: list.remove(2);
36: list.remove(Integer.valueOf(2));
37: System.out.println(list);
```

The correct answer is `[3]`. Let's look at how we got there. At the end of line 34, we have `[3, 2, 1]`. Line 35 passes a primitive, which means we are requesting deletion of the element at index `2`. This leaves us with `[3, 2]`. Then line 36 passes an `Integer` object, which means we are deleting the value 2. That brings us to `[3]`.

Solution

```
8: Set<Rabbit> rabbits = new TreeSet<>((r1, r2) -> r1.id - r2.id);
9: rabbits.add(new Rabbit(1));
```

Rules

- `List` : A `list` is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an `int` index.
- `Set` : A `set` is a collection that does not allow duplicate entries.
- `Queue` : A `queue` is a collection that orders its elements in a specific order for processing. A `Deque` is a subinterface of `Queue` that allows access at both ends.
- `Map` : A `map` is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

If the requested key _____	And mapping function returns _____	Then:
Has a `null` value in map	N/A (mapping function not called)	Update key's value in map with value parameter.
Has a non-`null` value in map	`null`	Remove key from map.
Has a non-`null` value in map	A non-`null` value	Set value to mapping function result.
Is not in map	N/A (mapping function not called)	Add key with value parameter to map directly without calling mapping function.

Comparing Comparable and Comparator

There are several differences between `Comparable` and `Comparator`. We've listed them for you in Table 9.8.

TABLE 9.8 Comparison of `Comparable` and `Comparator`

Difference	`Comparable`	`Comparator`
Package name	`java.lang`	`java.util`
Interface must be implemented by class comparing?	Yes	No
Method name in interface	`compareTo()`	`compare()`
Number of parameters	1	2
Common to declare using a lambda	No	Yes

Here are the things that you can't do with generics (and by "can't," we mean without resorting to conversions like passing in a class object):

- **Call a constructor**: Writing `new T()` is not allowed because at runtime, it would be `new Object()`.
- **Create an array of that generic type**: This one is the most annoying, but it makes sense because you'd be creating an array of `Object` values.
- **Call `instanceof`**: This is not allowed because at runtime `List<Integer>` and `List<String>` look the same to Java, thanks to type erasure.
- **Use a primitive type as a generic type parameter**: This isn't a big deal because you can use the wrapper class instead. If you want a type of `int`, just use `Integer`.
- **Create a `static` variable as a generic type parameter**: This is not allowed because the type is linked to the instance of the class.
- **Catch an exception of type `T`**: Even if `T extends Exception`, it cannot be used in a catch block since the precise type is not known.

Tips

Without generics, we'd have to write a lot of code like the following:

```
List numbers = new ArrayList(List.of(1,2,3));
Integer element = (Integer)numbers.get(0);  // Required cast to compi
numbers.add("Welcome to the zoo!");         // Unrelated types allowe
```

USING BOTH SHORTENERS

What happens if you use both `var` and the diamond operator?

```
var map = new HashMap<>();
```

Believe it or not, this does compile! If you try to have them both infer, there isn't enough information and you get `Object` as the generic type. This is equivalent to the following:

```
HashMap<Object, Object> map = new HashMap<Object, Object>();
```

The final thing to know about `merge()` is what happens when the mapping function is called and returns `null`. The key is removed from the map when this happens.

```
BiFunction<String, String, String> mapper = (v1, v2) -> null;
Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus Tour");
favorites.put("Tom", "Bus Tour");

favorites.merge("Jenny", "Skyride", mapper);
favorites.merge("Sam", "Skyride", mapper);
System.out.println(favorites);  // {Tom=Bus Tour, Sam=Skyride}
```

Tom was left alone since there was no `merge()` call for that key. Sam was added since that key was not in the original list. Jenny was removed because the mapping function returned `null`.

```
Comparator<Squirrel> c = Comparator.comparing(Squirrel::species)
   .thenComparingInt(Squirrel::weight);
```

REVERSING BINARYSEARCH()

The `binarySearch()` method requires a sorted `List`.

```
11: List<Integer> list = Arrays.asList(6,9,1,8);
12: Collections.sort(list); // [1, 6, 8, 9]
13: System.out.println(Collections.binarySearch(list, 6)); // 1
14: System.out.println(Collections.binarySearch(list, 3)); // -2
```

There is a trick in working with `binarySearch()`. What do you think the following outputs?

```
3: var names = Arrays.asList("Fluffy", "Hoppy");
4: Comparator<String> c = Comparator.reverseOrder();
5: var index = Collections.binarySearch(names, "Hoppy", c);
6: System.out.println(index);
```

The answer happens to be `-1`. Before you panic, you don't need to know that the answer is `-1`. You do need to know that the answer is not defined. Line 3 creates a list, `[Fluffy, Hoppy]`. This list happens to be sorted in ascending order. Line 4 creates a `Comparator` that reverses the natural order. Line 5 requests a binary search in descending order. Since the list is not in that order, we don't meet the precondition for doing a search.

Java Collections Cheat Sheet: Underlying Objects & Hidden Behaviors

Arrays.asList(...)

- **Creates a fixed-size list** backed by the original array.
- You **can modify elements** (set()), but **cannot add/remove** — throws UnsupportedOperationException.
- **Backed by array**: changes in array reflect in list and vice versa.

☑ **Make it mutable:**

List<Integer> mutableList = new ArrayList<>(Arrays.asList(1, 2, 3));
mutableList.add(4); // Works fine

List.of(...), Set.of(...), Map.of(...) (Java 9+)

- Creates **immutable collections**.
- Any attempt to add, remove, or modify throws UnsupportedOperationException.

Example:

List<String> list = List.of("a", "b"); // Immutable

Collections.unmodifiableList(...)

- Wraps a mutable list in an **unmodifiable view**.
- Changes to the original list **do reflect**, but you **can't modify via the wrapper**.

Example:

List<String> base = new ArrayList<>(List.of("a", "b"));
List<String> unmod = Collections.unmodifiableList(base);
base.add("c"); // unmod now also shows "c"
unmod.add("d"); // throws exception

Map quirks

- HashMap allows null keys and values.
- TreeMap does **not** allow null keys (throws NullPointerException).
- LinkedHashMap preserves insertion order.
- ConcurrentHashMap disallows null keys/values for thread safety.

Set quirks

- HashSet uses HashMap internally.
- LinkedHashSet preserves insertion order.
- TreeSet sorts elements (must be Comparable or use a Comparator).

Common Pitfalls

Scenario	What Happens	Fix
Arrays.asList(...).add()	✗ Exception	Wrap in new ArrayList<>(...)
List.of(...).set(0, ...)	✗ Exception	Use mutable list
Map.of("a", null)	✗ Exception	Use new HashMap<>()
TreeSet with mixed types	✗ ClassCastException	Use consistent types or custom comparator

Generics

A type parameter can be named anything you want. The convention is to use single uppercase letters to make it obvious that they aren't real class names. The following are common letters to use:

- `E` for an element
- `K` for a map key
- `V` for a map value
- `N` for a number
- `T` for a generic data type
- `S`, `U`, `V` and so forth for multiple generic types

Understanding Type Erasure

Specifying a generic type allows the compiler to enforce proper use of the generic type. For example, specifying the generic type of `Crate` as `Robot` is like replacing the `T` in the `Crate` class with `Robot`. However, this is just for compile-time.

Behind the scenes, the compiler replaces all references to `T` in `Crate` with `Object`. In other words, after the code compiles, your generics are just `Object` types. The `Crate` class looks like the following at runtime:

Overloading a Generic Method

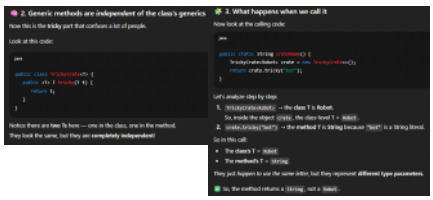
Only one of these two methods is allowed in a class because type erasure will reduce both sets of arguments to `(List list)`.

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
   protected void chew(List<Double> input) {}  // DOES NOT COMPILE
}
```

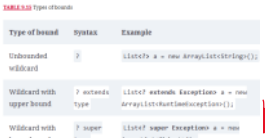
Making sure the generic types match for overridden methods. And remember, generic methods cannot be overloaded by changing the generic parameter type only.

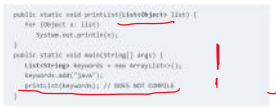
Example

```
List<String> strings = new ArrayList<>();
List<Integer> integers = new ArrayList<>();

System.out.println(strings.getClass() == integers.getClass()); // true
```

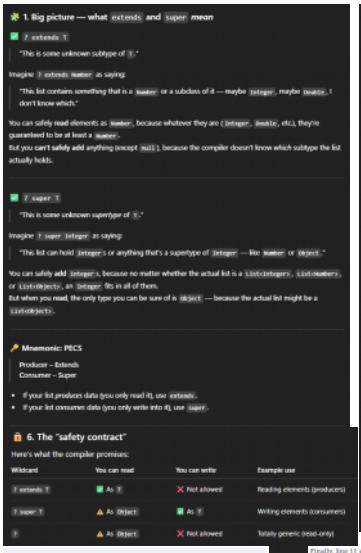


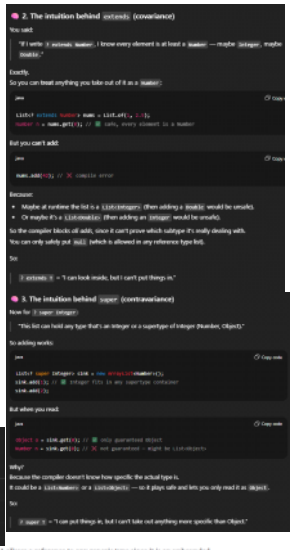
Bounding Generic Types

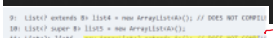




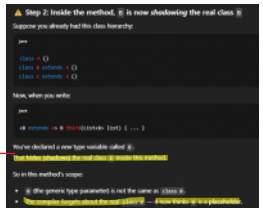
Solution

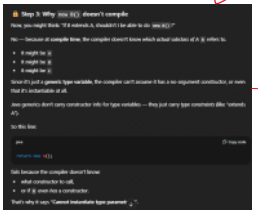






Finally, line 11 allows a reference to any generic type since it is an unbounded wildcard. The problem is that you need to know what that type will be when instantiating the ArrayList. It wouldn't be useful anyway, because you can't add any elements to that ArrayList.

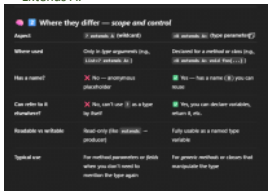




Instead we could read.

Possible fix

Difference between B extends A and ? Extends A.

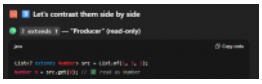


Finally, can you figure out why this example does not compile?

```
<X> void fifth(List<X super B> list) { } // DOES NOT COMPILE
```

This last method, `fifth()`, does not compile because it tries to mix a method-specific type parameter with a wildcard. A wildcard must have a `?` in it.

class. A compiler error results from code that attempts to add an item in a list with an unbounded or upper-bounded wildcard.



cific type parameter with a wildcard. A wildcard must have a `?` in it.

class. A compiler error results from code that attempts to add an item in a list with an unbounded or upper-bounded wildcard.

EXAM

1. A, F !! When says "Additionally" means 2 scenarios and 2 ans.
2. G, C

2. Which of the following are true? (Choose all that apply)

```
12: List<?> q = List.of("mouse", "parrot");
13: var v = List.of("mouse", "parrot");
14:
15: q.removeIf(String::isEmpty);
16: q.removeIf(s -> s.length() == 4);
17: v.removeIf(String::isEmpty);
18: v.removeIf(s -> s.length() == 4);
```

A. This code compiles and runs without error.
B. Exactly one of these lines contains a compiler error.
C. Exactly two of these lines contain a compiler error.
D. Exactly three of these lines contain a compiler error.
E. Exactly four of these lines contain a compiler error.
F. If any lines with compiler errors are removed, this code runs without throwing an exception.
G. If any lines with compiler errors are removed, this code throws an exception.

2. C, G. Line 12 creates a `List<?>`, which means it is treated as if all the elements are of type `Object` rather than `String`. Lines 15 and 16 do not compile since they call the `String` methods `isEmpty()` and `length()`, which are not defined on `Object`. Line 13 creates a `List<String>` because `var` uses the type that it deduces from the context. Lines 17 and 18 do compile. However, `List.of()` creates an immutable list, so both of those lines would throw an `UnsupportedOperationException` if run. Therefore, options C and G are correct.

3. b

3. What is the result of the following statements?

```
1:  var greetings = new ArrayDeque<String>();
4:  greetings.offerLast("hello");
5:  greetings.offerLast("hi");
6:  greetings.offerFirst("ola");
7:  greetings.pop();
8:  greetings.peek();
9:  while (greetings.peek() != null)
10:     System.out.print(greetings.pop());
```

A. `hello`
B. `hellohi`
C. `hellohiola`
D. `hiola`
E. The code does not compile.
F. An exception is thrown.

3. B. This is a double-ended queue. On lines 4 and 5, we add to the back, giving us `[hello, hi]`. On line 6, we add to the front and have `[ola, hello, hi]`. On line 7, we remove the first element, which is `"ola"`. On line 8, we look at the new first element (`"hello"`) but don't remove it. On lines 9 and 10, we remove each element in turn until no elements are left, printing `hello` and `hi` together, which makes option B the answer.

4. B,F

4. Which of these statements compile? (Choose all that apply)

```
A. HashSet<Number> hs = new HashSet<Integer>();
B. HashSet<? super ClassCastException> set = new HashSet<Exception>
   ();
C. List<> list = new ArrayList<String>();
D. List<Object> values = new HashSet<Object>();
E. List<Object> objects = new ArrayList<? extends Object>();
F. Map<String, ? extends Number> hm = new HashMap<String, Integer>
   ();
```

4. B, F. Option A does not compile because the generic types are not compatible. We could say `HashSet<? extends Number> hs2 = new HashSet<Integer>();`. Option B uses a lower bound, so it allows superclass generic types. Option C does not compile because the diamond operator is allowed only on the right side. Option D does not compile because a `Set` is not a `List`. Option E does not compile because upper bounds are not allowed when instantiating the type. Finally, option F does compile because the upper bound is on the correct side of the `=`.

5. E => B

5. What is the result of the following code?

```
1:  public record Hello<T>(T t) {
2:     public Hello(T t) { this.t = t; }
3:     private void println(T message) {
4:        System.out.print(t + "-" + message);
5:     }
6:     public static void main(String[] args) {
7:        new Hello<String>("hi").println(1);
8:        new Hello("hola").println(true);
9:     } }
```

A. `hi` followed by a runtime exception.
B. `hi-1hola-true`
C. The first compiler error is on line 1.
D. The first compiler error is on line 3.
E. The first compiler error is on line 8.
F. The first compiler error is on another line.

5. B. The record compiles and runs without issue. Line 8 gives a compiler warning for not using generics but not a compiler error. Line 1 creates the `Hello` class with the generic type `String`. It also passes an `int` to the `println()` method, which gets autoboxed into an `Integer`. While the `println()` method takes a generic parameter of type `T`, it is not the same `<T>` defined for the class on line 1. Instead, it is a different `T` defined as part of the method declaration on line 3. Therefore, the `String` argument on line 7 applies only to the class. The method can take any object as a parameter, including autoboxed primitives. Line 8 creates the `Hello` class with the generic type `Object` since no type is specified for that instance. It passes a `boolean` to `println()`, which gets autoboxed into a `Boolean`. The result is that `hi-1hola-true` is printed, making option B correct.

6. B,F

6. Which of the following can fill in the blank to print `[7, 5, 3]`? (Choose all that apply)

```
public record Platypus(String name, int beakLength) {
   @Override public String toString() {return "" + beakLength;}

   public static void main(String[] args) {
      Platypus p1 = new Platypus("Paula", 3);
      Platypus p2 = new Platypus("Peter", 5);
      Platypus p3 = new Platypus("Peter", 7);

      List<Platypus> list = Arrays.asList(p1, p2, p3);

      Collections.sort(list, Comparator.comparing(_____________));

      System.out.println(list);
   }
}
```

6. B, F. We're looking for a `Comparator` definition that sorts in descending order by `beakLength`. Option A is incorrect because it sorts in ascending order by `beakLength`. Similarly, option C is incorrect because it sorts by `beakLength` in ascending order within those matches that have the same `name`. Option E is incorrect because there is no `thenComparingNumber()` method.
Option B is a correct answer, as it sorts by `beakLength` in descending order. Options D and F are trickier. First, notice that we can call either `thenComparing()` or `thenComparingInt()` because the former will simply autobox the `int` into an `Integer`. Then observe what `reversed()` applies to. Option D is incorrect because it sorts by name in ascending order and only reverses the beak length of those with the same name. Option F creates a comparator that sorts by name in ascending order and then by beak size in descending order. Finally, it reverses the result. This is just what we want, so option F is correct.

7. F,B

7. Which of the following method signatures are valid overrides of the `halry()` method in the `Alpaca` class? (Choose all that apply)

```
import java.util.*;

public class Alpaca {
   public List<String> halry(List<String> list) { return null; }
}
```

7. B, F. A valid override of a method with generic arguments must have a return type that is covariant, with matching generic type parameters. Options D and E are incorrect because the return type is too broad. Additionally, the generic arguments must have the same signature with the same generic types. This eliminates options A and C. The remaining options are correct, making the answer options B and F.

8. B -> E

8. Which of the following fills in the blank, allowing the code to compile and run without issue?

```
11: SequencedCollection<String> animals = new _____________<>();
12: animals.addFirst("lions");
13: animals.addLast("tigers");
14: for(var a : animals)
15:    System.out.println(a);
16: System.out.println(animals.get(0));
```

A. `HashSet`
B. `LinkedList`
C. `TreeSetMap`
D. `HashMap`
E. None of the above

8. E. There is no `getInt()` method defined in `SequencedCollection`, meaning line 16 does not compile, regardless of what is placed in the blank. For this reason, option E is correct. If line 16 was removed or corrected to use `getFirst()`, then `LinkedList` would be the correct answer.

9. A

9. What is the result of the following program?

```
3:  public class MyComparator implements Comparator<String> {
4:     public int compare(String a, String b) {
5:        return b.toLowerCase().compareTo(a.toLowerCase());
6:     }
7:     public static void main(String[] args) {
8:        String[] values = { "123", "Abb", "aab" };
9:        Arrays.sort(values, new MyComparator());
10:       for (var s: values)
11:          System.out.print(s + " ");
12:    }
13: }
```

A. `Abb aab 123`
B. `aab Abb 123`
C. `123 Abb aab`
D. `123 aab Abb`
E. The code does not compile.
F. A runtime exception is thrown.

Because is b before a, is in a reversed order.

10. A,B,D

10. Which of these statements can fill in the blank so that the `Helper` class compiles successfully? (Choose all that apply)

```
2:  public class Helper {
3:     public static <U extends Exception>
4:        void printException(U u) {
5:
6:        System.out.println(u.getMessage());
7:     }
8:     public static void main(String[] args) {
9:        Helper._____________________________;
10:    } }
```

A. `printException(new FileNotFoundException("A"))`
B. `printException(new Exception("B"))`
C. `<Throwable>printException(new Exception("C"))`

10. A, B, D. The generic type must be `Exception` or a subclass of `Exception` since this is an upper bound, making options A and B correct. Options C and E are wrong because `Throwable` is a superclass of `Exception`. Additionally, option D is correct despite the odd syntax by explicitly listing the type. You should still be able to recognize it as acceptable.

10. Which of these statements can fill in the blank so that the `Helper` class compiles successfully? (Choose all that apply.)

```
2:  public class Helper {
3:      public static <E extends Exception>
4:          void printException(E e) {
5:
6:          System.out.println(e.getMessage());
7:      }
8:      public static void main(String[] args) {
9:          Helper._____________________________;
10:     } }
```

A. `printException(new FileNotFoundException("A"))`
B. `printException(new Exception("B"))`
C. `<Throwable>printException(new Exception("C"))`
D. `<NullPointerException>printException(new NullPointerException("D"))`
E. `printException(new Throwable("E"))`

`Exception` since this is an upper bound, making options A and B correct. Options C and E are wrong because `Throwable` is a super-class of `Exception`. Additionally, option D is correct despite the odd syntax by explicitly listing the type. You should still be able to recognize it as acceptable.

11. A,B,D,F E

12. Which of the following will compile when filling in the blank? (Choose all that apply.)

```
var list = List.of(1, 2, 3);
var set = Set.of(3, 2, 3);
var map = Map.of(1, 2, 2, 4);

_____________.forEach(System.out::println);
```

A. `list`
B. `set`
C. `map`
D. `map.keys()`
E. `map.keySet()`
F. `map.values()`
G. `map.valueSet()`

11. A, B, E, F. The `forEach()` method works with a `Collection`, such as `List` or a `Set`. Therefore, options A and B are correct. Additionally, options E and F return a `Set` and `Collection`, respectively, and can be used as well. Options D and G refer to methods that do not exist. Option C is tricky because a `Map` does have a `forEach()` method. However, it uses two lambda parameters rather than one. Since there is no matching `System.out.println` method, it does not compile.

12. F B E

12. What do these statements can fill in the blank so that the `Wildcard` class compiles successfully? (Choose all that apply.)

```
3:  public class Wildcard {
4:      public void showSize(List<?> list) {
5:          System.out.println(list.size());
6:      }
7:      public static void main(String[] args) {
8:          Wildcard card = new Wildcard();
9:          _____________________________;
10:         card.showSize(list);
11:     } }
```

A. `List<?> list = new HashSet <String>()`
B. `ArrayList<? super Date> list = new ArrayList<Date>()`
C. `List<?> list = new ArrayList<?>()`
D. `List<Exception> list = new LinkedList<java.io.IOException>()`
E. `ArrayList <? extends Number> list = new ArrayList <Integer>()`
F. None of the above

12. B, E. The `showSize()` method can take any type of `List` since it uses an unbounded wildcard. Option A is incorrect because it is a `Set` and not a `List`. Option C is incorrect because the wildcard is not allowed to be on the right side of an assignment. Option D is incorrect because the generic types are not compatible. Option B is correct because a lower-bounded wildcard allows that same type to be the generic. Option E is correct because `Integer` is a subclass of `Number`.

13. E C

13. What is the result of the following program?

```
1:  public record Sorted(int num, String text)
4:      implements Comparable<Sorted>, Comparator<Sorted> {
5:
6:      public String toString() { return "" + num; }
7:      public int compare(Sorted s1, Sorted s2) {
8:          return s1.text.compareTo(s2.text);
9:      }
10:     public int compareTo(Sorted s2, Sorted s1) {
11:         return s1.num - s2.num;
12:     }
13:     public static void main(String[] args) {
14:         var s1 = new Sorted(88, "a");
15:         var s2 = new Sorted(55, "b");
16:         SequencedSet<Sorted> ss = new TreeSet<Sorted>();
17:         ss.add(s1); ss.add(s2);
18:         var s3 = new Sorted(22, "c");
19:         ss.add(s3); ss.add(s2);
20:         System.out.println(ss + " " + ss);
21:     } }
```

A. `[55, 88] [55, 88]`
B. `[55, 88] [88, 55]`
C. `[88, 55] [55, 88]`
D. `[88, 55] [88, 55]`
E. The code does not compile.
F. A runtime exception is thrown.

13. C. This question is difficult because it defines both `Comparable` and `Comparator` on the same object. The `t1` object doesn't specify a `Comparator`, so it uses the `Comparable` object's `compareTo()` method. This sorts by the `text` instance variable. The `t2` object does specify a `Comparator` when calling the constructor, so it uses the `compare()` method, which sorts by the `int`. This gives us option C as the answer. Note that the `SequencedSet` reference on line 16 does not change the ordering, as the underlying object is still a `TreeSet`.

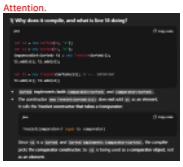
14. A,

14. What is the result of the following code?

```
Comparator<Integer> c1 = (n1, n2) -> n2 - n1;
Comparator<Integer> c2 = Comparator.naturalOrder();
Comparator<Integer> c3 = Comparator.reverseOrder();

var list = Arrays.asList(5, 4, 7, 2);
Collections.sort(list,_____________);
Collections.reverse(list);
Collections.reverse(list);
System.out.println(Collections.binarySearch(list, 2));
```

A. One or more of the comparators can fill in the blank so that the code prints `0`.
B. One or more of the comparators can fill in the blank so that the code prints `1`.
C. One or more of the comparators can fill in the blank so that the code prints `2`.
B. The result is undefined regardless of which comparator is used.
E. A runtime exception is thrown regardless of which comparator is used.
F. The code does not compile.

Attention.



!!!Precondition for binarySearch,
- needs to be ordered ascending.

15. A,B,

15. Which of the following lines can be inserted to make the code compile? (Choose all that apply.)

```
class W {}
class X extends W {}
class Y extends X {}
class Demo {
   // INSERT CODE HERE
}
```

A. `W w5 = new W();`
B. `W w2 = new X();`
C. `W w3 = new Y();`
D. `Y y5 = new W();`
E. `Y y2 = new X();`
F. `Y y3 = new Y();`

15. A, B. `Y` is both a class and a type parameter. This means that within the class `Y`, when we refer to `Y`, it uses the type parameter. All of the choices that mention class `Y` are incorrect because it no longer means the class `Y`. Only options A and B are correct.

16. A,B <

16. Which options are true of the following code? (Choose all that apply.)

```
                    q = new LinkedList<>();
var u = Collections.unmodifiableCollection(q);
q.add(10);
q.add(12);
q.remove(1);
System.out.print(u);
```

A. If we fill in the blank with `List<Integer>`, the output is `[10]`.
B. If we fill in the blank with `Queue<Integer>`, the output is `[10]`.
C. If we fill in the blank with `var`, the output is `[10]`.
D. One or more of the scenarios does not compile.
E. One or more of the scenarios throws a runtime exception.

16. A, C. A `LinkedList` implements both `List` and `Queue`. The `List` interface has a method to remove by index. Since this method exists, Java does not autobox to call the other method, making the output `[10]` and option A correct. Similarly, option C is correct because the method to remove an element by index is available on a `LinkedList<Object>` (which is what `var` represents here). By contrast, `Queue` has only the remove by object method, so Java does autobox there. Since the number `1` is not in the list, Java does not remove anything for the `Queue`, and the output is `[10, 12]`. The `unmodifiableCollection()` call is a distractor as it is an unmodifiable view and the underlying connection can be changed.

17. C,E

17. What is the result of the following code?

```
4:  Map m = new HashMap();
5:  m.put(123, "456");
6:  m.put("abc", "def");
7:  System.out.println(m.contains("123"));
```

A. `false`
B. `true`
C. Compiler error on line 4.
D. Compiler error on line 5.
E. Compiler error on line 7.
F. A runtime exception is thrown.

17. E. This question looks like it is about generics, but it's not. It is trying to see whether you noticed that `Map` does not have a `contains()` method. It has `containsKey()` and `containsValue()` instead, making option E the answer. If `containsKey()` were called, the answer would be `false` because `123` is an `Integer` key in the `Map`, rather than a `String`.

18. B,F ?

18. What is the result of the following code? (Choose all that apply)

```
48: var map = Map.of(1,2, 3, 4);
49: var list = list.copyOf(map.entrySet());
50:
51: List<Integer> one = List.of(1, 10, 2);
52: var copy = list.copyOf(one);
53: var copyCopy = List.copyOf(copy);
54: var thirdCopy = new ArrayList<>(copyCopy);
55:
56: list.replaceAll(x -> x * 2);
57: one.replaceAll(x -> x * 2);
58: thirdCopy.replaceAll(x -> x * 2);
59:
60: System.out.println(thirdCopy);
```

A. One line fails to compile.
B. Two lines fail to compile.
C. Three lines fail to compile.
D. The code compiles but throws an exception at runtime.
E. If any lines with compiler errors are removed, the code throws an exception at runtime.
F. If any lines with compiler errors are removed, the code prints `[10, 12, 4]`.
G. The code compiles and prints `[40, 12, 4]` without any changes.

18. A, E. The key to this question is keeping track of the types. Line 48 is a `Map<Integer, Integer>`. Line 49 builds a `List` out of a `Set` of `Entry` objects, giving us `List<Entry<Integer, Integer>>`. This causes a compiler error on line 56 since we can't multiply an `Entry` object by two.
Lines 51–54 are all of type `List<Integer>`. The first three are immutable, and the one on line 54 is mutable. This means line 57 throws an `UnsupportedOperationException` since we attempt to modify the list. Line 58 would work if we could get to it. Since there is one compiler error and one runtime error, options A and E are correct.

19. B, sau A ? B

19. What code change is needed to make the method compile, assuming there is no class named `T`?

```
public static T identity(T t) {
  return t;
}
```

A. Add `<T>` after the `public` keyword.
B. Add `<T>` after the `static` keyword.
C. Add `<T>` after `T`.
D. Add `<T>` after the `public` keyword.
E. Add `<T>` after the `static` keyword.
F. No change is required. The code already compiles.

20. F.
21. B,D,F
22. ~~E~~ B

22. What is the output of the following code snippet?

```
21: SequencedMap<Integer, String> cats = new TreeMap<>();
22: cats.put(3, "Snowball");
23: cats.put(2, "Sugar");
24: cats.put(1, "Minnie Mouse");
25: cats.pollFirstEntry();
26: var id = cats.lastEntry().getKey();
27: cats.pollFirstEntry();
28: System.out.print(cats.firstEntry().getValue());
```

A. `Minnie Mouse`
B. `Snowball`
C. `Sugar`
D. The code does not compile.
E. The code compiles, but an exception is thrown at runtime.

22. **B.** The code compiles and runs without issue, so options D and E are incorrect. A `TreeMap` sorts its items in the natural order of keys (not the values). Therefore, lines 24 and 27 remove `[1, Minnie Mouse]` and `[2, Sugar]`, respectively. Line 26 has no impact on the map. On line 28, `Snowball` is printed, making option B correct. If line 26 were changed to use `pollLastEntry()`, then the map would be empty and line 28 would throw a `NullPointerException` trying to call `getValue()`.

23. ~~A~~ H

23. What is the output of the following code snippet?

```
var fishes = new TreeSet<String>();
fishes.add("Koi");
fishes.addFirst("Clown");
fishes.add("Carp");
for(var fish : fishes)
   System.out.print(fish + ", ");
```

A. `carp, clown, koi,`
B. `carp, koi, clown,`
C. `clown, carp, koi,`
D. `clown, koi, carp,`
E. `koi, carp, clown,`
F. `koi, clown, carp,`
G. The code does not compile.
H. The code compiles but throws an exception at runtime.

23. **H.** `TreeSet` is a `SequencedSet`, so it does have an `addFirst()` method. For this reason, the code does compile. Unfortunately, `addFirst()` is not supported at runtime, as inserting an element at the front of the `TreeSet` could violate the `Comparator` of the `TreeSet`. For this reason, the code program throws an `UnsupportedOperationException` on the third line.