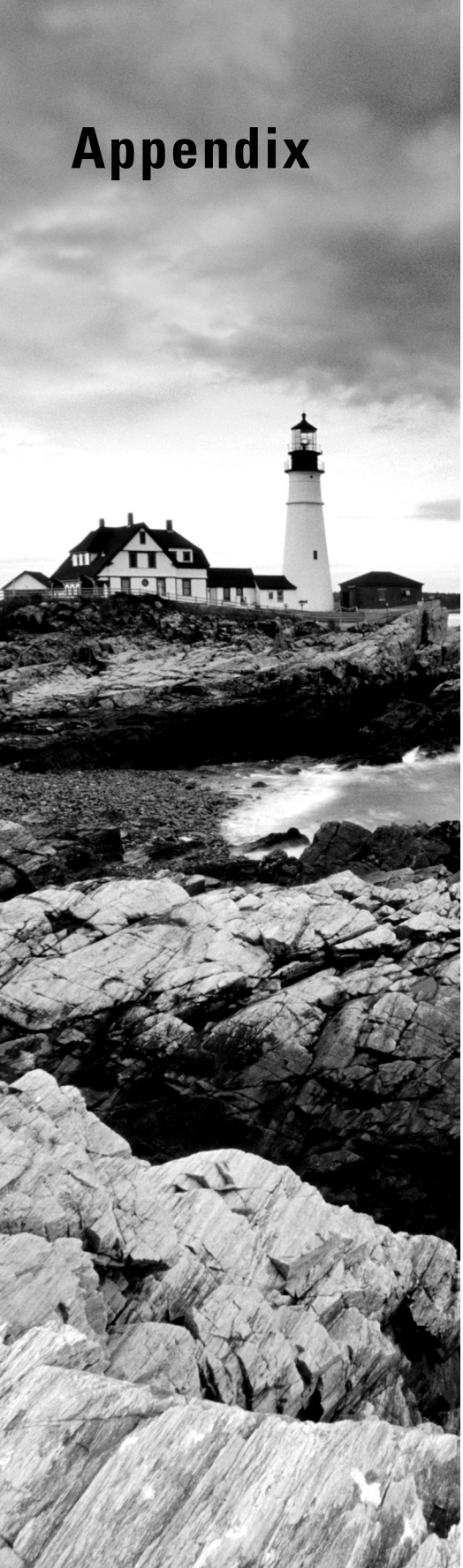

Appendix

Answers to the Review Questions

Chapter 1: Building Blocks

1. D, E. Option E is the canonical `main()` method signature. You need to memorize it. Option D is an alternate form with the redundant `final`. Option A is incorrect because the `main()` method must be public. Options B and F are incorrect because the `main()` method must have a `void` return type. Option C is incorrect because the `main()` method must be `static`.

2. C, D, E. The `package` and `import` statements are both optional. If both are present, the order must be `package`, then `import`, and then `class`. Option A is incorrect because `class` is before `package` and `import`. Option B is incorrect because `import` is before `package`. Option F is incorrect because `class` is before `package`.

3. A, E. `Bunny` is a class, which can be seen from the declaration: `public class Bunny`. The variable `bun` is a reference to an object. The method `main()` is the standard entry point to a program. Option G is incorrect because the parameter type matters, not the parameter name.

4. B, E, G. Option A is invalid because a single underscore is not allowed. Option C is not a valid identifier because `true` is a Java reserved word. Option D is not valid because a period (`.`) is not allowed in identifiers. Option F is not valid because the first character is not a letter, dollar sign ($), or underscore (_). Options B, E, and G are valid because they contain only valid characters.

5. A, D, F. Garbage collection is never guaranteed to run, making option F correct and option E incorrect. Next, the class compiles and runs without issue, so option G is incorrect. The `Bear` object created on line 9 is accessible until line 13 via the `brownBear` reference variable, which is option A. The `Bear` object created on line 10 is accessible via both the `polarBear` reference and the `brownBear.pandaBear` reference. After line 12, the object is still accessible via `brownBear.pandaBear`. After line 13, though, it is no longer accessible since `brownBear` is no longer accessible, which makes option D the final answer.

6. F. To solve this problem, you need to trace the braces `{}` and see when variables go in and out of scope. The variables on lines 2 and 7 are only in scope for a single line block. The variable on line 12 is only in scope for the `for` loop. None of these are in scope on line 14. By contrast, the three instance variables on lines 3 and 4 are available in all instance methods. Additionally, the variables on lines 6, 9, and 10 are available since the method and `while` loop are still in scope. This is a total of 7 variables, which is option F.

7. C, E. The first thing to recognize is that this is a text block and the code inside the `"""` is just text. Options A and B are incorrect because the `numForks` and `numKnives` variables are not used. This is convenient since `numKnives` is not initialized and would not compile if it were referenced. Option C is correct as it is matching text. Option D is incorrect because the text block does not have a trailing blank line. Finally, option E is also an answer since `" # knives` is indented.

8. B, D, E, H. A `var` cannot be initialized with a `null` value without a type, but it can be assigned a `null` value later if the underlying type is not a primitive. For these reasons, option H is correct, but options A and C are incorrect. Options B and D are correct as the underlying types are `String` and `Integer`, respectively. Option E is correct as this is a valid numeric expression. You might know that dividing by zero produces a runtime exception, but the question was only about whether the code compiled. Finally, options F and G are incorrect as `var` cannot be used in a multiple-variable assignment.

9. E. Options C and D are incorrect because local variables don't have default values. Option A is incorrect because `float` should have a decimal point. Option B is incorrect because primitives do not default to `null`. Option E is correct and option F incorrect because reference types in class variables default to `null`.

10. A, E, F. An underscore (`_`) can be placed in any numeric literal, as long as it is not at the beginning, at the end, or next to a decimal point (`.`). Underscores can even be placed next to each other. For these reasons, options A, E, and F are correct. Options B and D are incorrect as the underscore (`_`) is next to a decimal point (`.`). Options C and G are incorrect because an underscore (`_`) cannot be placed at the beginning or end of the literal.

11. E. The first two imports can be removed because `java.lang` is automatically imported. The following two imports can be removed because `Tank` and `Water` are in the same package, making the correct option E. If `Tank` and `Water` were in different packages, exactly one of these two imports could be removed. In that case, the answer would be option D.

12. A, C, D. Line 2 does not compile as only one type should be specified, making option A correct. Line 3 compiles without issue as it declares a local variable inside an instance initializer that is never used. Line 4 does not compile because Java does not support setting default method parameter values, making option C correct. Finally, line 7 does not compile because `fins` is in scope and accessible only inside the instance initializer on line 3, making option D correct.

13. A, B, C. Option A is correct because it imports all the classes in the `aquarium` package including `aquarium.Water`. Options B and C are correct because they import `Water` by class name. Since importing by class name takes precedence over wildcards, these compile. Option D is incorrect because Java doesn't know which of the two wildcard `Water` classes to use. Option E is incorrect because you cannot specify the same class name in two imports.

14. A, B, D, E. Line 3 does not compile because the `L` suffix makes the literal value a `long`, which cannot be stored inside a `short` directly, making option A correct. Line 4 does not compile because `int` is an integral type, but `2.0` is a `double` literal value, making option B correct. Line 5 compiles without issue. Lines 6 and 7 do not compile because `numPets` and `numGrains` are both primitives, and you can call methods only on reference types, not primitive values, making options D and E correct, respectively. Finally, line 8 compiles because there is a `length()` method defined on `String`.

15. C, E, F. In Java, there are no guarantees about when garbage collection will run. The JVM is free to ignore calls to `System.gc()`. For this reason, options A, B, and D are incorrect. Option C is correct as the purpose of garbage collection is to reclaim used memory. Option E is also correct that an object may never be garbage collected, such as if the program ends

before garbage collection runs. Option F is correct and is the primary means by which garbage collection algorithms determine whether an object is eligible for garbage collection. Finally, option G is incorrect as marking a variable `final` means it is constant within its own scope. For example, a local variable marked `final` will be eligible for garbage collection after the method ends, assuming there are no other references to the object that exist outside the method.

16. A, D. Option A is correct. There are two lines. One starts with `squirrel`, and the other starts with `pigeon`. Remember that a backslash means to skip the line break. Option D is also correct as `\s` means to keep whitespace. In a text block, incidental indentation is ignored, making option F incorrect.

17. D, F, G. The code compiles and runs without issue, so options A and B are incorrect. A `boolean` field initializes to `false`, making option D correct with `Empty = false` being printed. Object references initialize to `null`, not the empty `String`, so option F is correct with `Brand = null` being printed. Finally, the default value of floating-point numbers is `0.0`. Although `float` values can be declared with an `f` suffix, they are not printed with an `f` suffix. For these reasons, option G is correct and `Code = 0.0` is printed.

18. B, C, F. A `var` cannot be used for a constructor or method parameter or for an instance or class variable, making option A incorrect and option C correct. The type of a `var` is known at compile-time, and the type cannot be changed at runtime, although its value can change at runtime. For these reasons, options B and F are correct, and option E is incorrect. Option D is incorrect, as `var` is not permitted in multiple-variable declarations. Finally, option G is incorrect, as `var` is not a reserved word in Java.

19. A, D. The first two lines provide a way to convert a `String` into a number. The first is a `long` primitive and the second is a `Long` reference object, making option D one of the answers. The code is correct and the maximum is `100`, which is option A.

20. C. The key thing to notice is that line 4 does not define a constructor, but instead a method named `PoliceBox()`, since it has a return type of `void`. This method is never executed during the program run, and `color` and `age` are assigned the default values `null` and `0L`, respectively. Lines 11 and 12 change the values for an object associated with `p`, but then, on line 13, the `p` variable is changed to point to the object associated with `q`, which still has the default values. For this reason, the program prints `Q1=null`, `Q2=0`, `P1=null`, and `P2=0`, making option C the only correct answer.

21. D. We start with the `main()` method, which prints `7-` on line 10. Next, a new `Salmon` instance is created on line 11. This calls the two instance initializers on lines 3 and 4 to be executed in order. The default value of an instance variable of type `int` is `0`, so `0-` is printed next and `count` is assigned a value of `1`. Next, the constructor is called. This assigns a value of `4` to `count` and prints `2-`. Finally, line 12 prints `4-`, since that is the value of `count`. Putting it all together, we have `7-0-2-4-`, making option D the correct answer.

22. C, F, G. First, `0b` is the prefix for a binary value, and `0x` is the prefix for a hexadecimal value. These values can be assigned to many primitive types, including `int` and `double`, making options C and F correct. Option A is incorrect because naming the variable `Amount` will cause the `System.out.print(amount)` call on the next line to not compile. Option B is incorrect because `9L` is a `long` value. If the type was changed to `long amount = 9L`,

then it would compile. Option D is incorrect because `1_2.0` is a `double` value. If the type was changed to `double amount = 1_2.0`, then it would compile. Options E and H are incorrect because the underscore (`_`) appears next to the decimal point (`.`), which is not allowed. Finally, option G is correct, and the underscore and assignment usage are valid.

23. A, D. The first compiler error is on line 3. The variable `temp` is declared as a `float`, but the assigned value is `50.0`, which is a `double` without the `F/f` postfix. Since a `double` doesn't fit inside a `float`, line 3 does not compile. Next, `depth` is declared inside the `for` loop and only has scope inside this loop. Therefore, reading the value on line 10 triggers a compiler error. For these reasons, options A and D are the correct answers.

Chapter 2: Operators

1. A, D, G. Option A is the equality operator and can be used on primitives and object references. Options B and C are both arithmetic operators and cannot be applied to a `boolean` value. Option D is the logical complement operator and is used exclusively with `boolean` values. Option E is the modulus operator, which can be used only with numeric primitives. Option F is a bitwise complement operator and can only be applied to integer values. Finally, option G is correct, as you can cast a `boolean` variable since `boolean` is a type.

2. A, B, D. The expression `apples + oranges` is automatically promoted to `int`, so `int` and data types that can be promoted automatically from `int` will work. Options A, B, and D are such data types. Option C will not work because `boolean` is not a numeric data type. Options E and F will not work without an explicit cast to a smaller data type.

3. B, C, D, F. The code will not compile as is, so option A is not correct. The value `2 * ear` is automatically promoted to `long` and cannot be automatically stored in `hearing`, which is an `int` value. Options B, C, and D solve this problem by reducing the `long` value to `int`. Option E does not solve the problem and actually makes it worse by attempting to place the value in a smaller data type. Option F solves the problem by increasing the data type of the assignment so that `long` is allowed.

4. B. The code compiles and runs without issue, so option E is not correct. This example is tricky because of the second assignment operator embedded in line 5. The expression `(wolf=false)` assigns the value `false` to `wolf` and returns `false` for the entire expression. Since `teeth` does not equal `10`, the left side returns `true`; therefore, the exclusive or (`^`) of the entire expression assigned to `canine` is `true`. The output reflects these assignments, with no change to `teeth`, so option B is the only correct answer.

5. A, C. Options A and C show operators in increasing or the same order of precedence. Options B and E are in decreasing or the same order of precedence. Options D, F, and G are in neither increasing nor decreasing order of precedence. In option D, the assignment operator (`=`) is between two unary operators, with the multiplication operator (`*`) incorrectly being in place of highest precedence. In option F, the logical complement operator (`!`) has the highest order of precedence, so it should be last. In option G, the assignment operators have the lowest order of precedence, not the highest, so the last two operators should be first.

6. F. The code does not compile because line 3 contains a compilation error. The cast `(int)` is applied to `fruit`, not the expression `fruit+vegetables`. Since the cast operator has a higher operator precedence than the addition operator, it is applied to `fruit`, but the expression is promoted to a `float`, due to `vegetables` being `float`. The result cannot be returned as `long` in the `addCandy()` method without a cast. For this reason, option F is correct. If parentheses were added around `fruit+vegetables`, then the output would be `3, 5, 6`, and option B would be correct. Remember that casting floating-point numbers to integral values results in truncation, not rounding.

7. D. In the first `boolean` expression, `vis` is `2` and `ph` is `7`, so this expression evaluates to `true & (true || false)`, which reduces to `true`. The second `boolean` expression uses the conditional operator, and since `(vis > 2)` is `false`, the right side is not evaluated, leaving `ph` at `7`. In the last assignment, `ph` is `7`, and the pre-decrement operator is applied first, reducing the expression to `7 <= 6` and resulting in an assignment of `false`. For these reasons, option D is the correct answer.

8. A. The code compiles and runs without issue, so option E is incorrect. Line 7 does not produce a compilation error since the compound operator applies casting automatically. Line 5 increments `pig` by `1`, but it returns the original value of `4` since it is using the post-increment operator. The `pig` variable is then assigned this value, and the increment operation is discarded. Line 7 just reduces the value of `goat` by `1`, resulting in an output of `4 - 1` and making option A the correct answer.

9. A, D, E. The code compiles without issue, so option G is incorrect. In the first expression, `a > 2` is `false`, so `b` is incremented to `5`; but since the post-increment operator is used, `4` is printed, making option D correct. The `--c` was not applied, because only one of the right-hand expressions was evaluated. In the second expression, `a!=c` is `false` since `c` was never modified. Since `b` is `5` due to the previous line and the post-increment operator is used, `b++` returns `5`. The result is then assigned to `b` using the assignment operator, overriding the incremented value for `b` and printing `5`, making option E correct. In the last expression, parentheses are not required, but lack of parentheses can make ternary expressions difficult to read. From the previous lines, `a` is `2`, `b` is `5`, and `c` is `2`. We can rewrite this expression with parentheses as `(2 > 5 ? (5 < 2 ? 5 : 2) : 1)`. The second ternary expression is never evaluated since `2 > 5` is `false`, and the expression returns `1`, making option A correct.

10. G. The code does not compile due to an error on the second line. Even though both `height` and `weight` are cast to `byte`, the multiplication operator automatically promotes them to `int`, resulting in an attempt to store an `int` in a `short` variable. For this reason, the code does not compile, and option G is the only correct answer. This line contains the only compilation error.

11. D. First, `*` and `%` have the same operator precedence, so the expression is evaluated from left to right unless parentheses are present. The first expression evaluates to `8 % 3`, which leaves a remainder of `2`. The second expression is evaluated left to right since `*` and `%` have the same operator precedence, and it reduces to `6 % 3`, which is `0`. The last expression reduces to `5 * 1`, which is `5`. Therefore, the output on line 14 is `2, 0, 5`, making option D the correct answer.

12. D. The *pre-* prefix indicates the operation is applied first, and the new value is returned, while the *post-* prefix indicates the original value is returned prior to the operation. Next, increment increases the value, while decrement decreases the value. For these reasons, option D is the correct answer.

13. F. The first expression is evaluated from left to right since the operator precedence of & and `^` is the same, letting us reduce it to `false ^ sunday`, which is `true`, because `sunday` is `true`. In the second expression, we apply the negation operator (`!`) first, reducing the expression to `sunday && true`, which evaluates to `true`. In the last expression, both variables are `true`, so they reduce to `!(true && true)`, which further reduces to `!true`, aka `false`. For these reasons, option F is the correct answer.

14. B, E, G. The return value of an assignment operation in the expression is the same as the value of the newly assigned variable. For this reason, option A is incorrect, and option E is correct. Option B is correct, as the equality (`==`) and inequality (`!=`) operators can both be used with objects. Option C is incorrect, as `boolean` and numeric types are not comparable. For example, you can't say `true == 3` without a compilation error. Option D is incorrect, as logical operators evaluate both sides of the expression. The (`|`) operator will cause both sides to be evaluated. Option F is incorrect, as Java does not accept numbers for `boolean` values. Finally, option G is correct, as you need to use the negation operator (`-`) to flip or negate numeric values, not the logical complement operator (`!`).

15. D. The ternary operator is the only operator that takes three values, making option D the only correct choice. Options A, B, C, E, and G are all binary operators. While they can be strung together in longer expressions, each operation uses only two values at a time. Option F is a unary operator and takes only one value.

16. B. The first line contains a compilation error. The value `3` is cast to `long`. The `1 * 2` value is evaluated as `int` but promoted to `long` when added to the `3`. Trying to store a `long` value in an `int` variable triggers a compiler error. The other lines do not contain any compilation errors, as they store smaller values in larger or same-size data types, with lines 2 and 4 using casting to do so. Since only one line does not compile, option B is correct.

17. C, F. The starting values of `ticketsTaken` and `ticketsSold` are `1` and `3`, respectively. After the first compound assignment, `ticketsTaken` is incremented to `2`. The `ticketsSold` value is increased from `3` to `5`; since the post-increment operator was used, the value of `ticketsTaken++` returns `1`. On the next line, `ticketsTaken` is doubled to `4`. On the final line, `ticketsSold` is increased by `1` to `6`. The final values of the variables are `4` and `6`, for `ticketsTaken` and `ticketsSold`, respectively, making options C and F the correct answers. Note the last line does not trigger a compilation error as the compound operator automatically casts the right-hand operand.

18. C. Only parentheses, `( )`, can be used to change the order of operation in an expression, making option C correct. The other operators, such as `[ ]`, `< >`, and `{ }`, cannot be used as parentheses in Java.

19. B, F. The code compiles and runs successfully, so options G and H are incorrect. On line *5*, the pre-increment operator is executed first, so `start` is incremented to `8`, and the new value is returned as the right side of the expression. The value of `end` is computed by adding `8` to

the original value of 4, leaving a new value of 12 for end and making option F a correct answer. On line 6, we are incrementing one past the maximum byte value. Due to overflow, this 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.

20. A, D, E. Unary operators have the highest order of precedence, making option A correct. The negation operator (-) is used only for numeric values, while the logical complement operator (!) is used exclusively for boolean values. For these reasons, option B is incorrect, and option E is correct. Finally, the pre-increment/pre-decrement operators return the new value of the variable, while the post-increment/post-decrement operators return the original variable. For these reasons, option C is incorrect, and option D is correct.

21. E. The bitwise complement of 8 can be found by multiplying the number by negative one and subtracting one, making -9 the value of bird. By contrast, plane is -8 because it negates myFavoriteNumber. Since bird and plane are not the same, superman is assigned a value of 10. The pre-decrement operator takes superman, subtracts 1, and returns the new value, printing 9. For this reason, option E is correct.

Chapter 3: Making Decisions

1. A, B, C, E, F, G. A switch expression supports only the primitives int, byte, short, and char, along with their associated wrapper classes Integer, Byte, Short, and Character, respectively, making options B, C, and F correct and ruling out options D and H. It also supports enum and String, making options A and E correct. Finally, switch supports var if the type can be resolved to a supported switch data type, making option G correct.

2. B. The code compiles and runs without issue, so options D, E, and F are incorrect. Even though two consecutive else statements on lines 7 and 8 look a little odd, they are associated with separate if statements on lines 5 and 6, respectively. The value of humidity on line 4 is equal to -4 + 12, which is 8. The first if statement evaluates to true on line 5, so line 6 is executed and evaluates to false. This causes the else statement on line 7 to run, printing Just Right and making option B the correct answer.

3. A, D, F, H. A for-each loop supports arrays, making options A and F correct. For Double[][], each element of the for-each loop would be a Double[]. A for-each loop also supports classes that implement java.lang.Iterable. Although this includes many of the Collections Framework classes, not all of them implement java.lang.Iterable. For this reason, option C is incorrect, and options D and H are correct. Options B, E, and G are incorrect, as they do not implement java.lang.Iterable. Although a String is a list of ordered characters, the class does not implement the required interface for a for-each loop.

4. F. The code does not compile because the `switch` expression requires all possible case values to be handled, making option F correct. If a valid `default` statement was added, then the code would compile and print `Turtle` at runtime. Unlike traditional `switch` statements, `switch` expressions execute exactly one branch and do not use `break` statements between `case` statements.

5. E. The second for-each loop contains a `continue` followed by a `print()` statement. Because the `continue` is not conditional and always included as part of the body of the for-each loop, the `print()` statement is not reachable. For this reason, the `print()` statement does not compile. As this is the only compilation error, option E is correct. The other lines of code compile without issue.

6. C, D, E. A for-each loop can be executed on any Collections object that implements `java.lang.Iterable`, such as `List` or `Set`, but not all Collections classes, such as `Map`, so option A is incorrect. The body of a `do/while` loop is executed one or more times, while the body of a `while` loop is executed zero or more times, making option E correct and option B incorrect. The conditional expression of `for` loops is evaluated at the start of the loop execution, meaning the `for` loop may execute zero or more times, making option C correct. A `switch` expression that takes a `String` requires a `default` branch if the result is assigned to a variable, making option D correct. Finally, each `if` statement has at most one matching `else` statement, making option F incorrect.

7. B, D. Option A is incorrect because on the first iteration, it attempts to access `weather[weather.length]` of the nonempty array, which causes an `ArrayIndexOutOfBoundsException` to be thrown. Option B is correct and will print the elements in order. Option C doesn't compile as `i` is undefined in `weather[i]`. For this to work, the body of the for-each loop would have to be updated as well. Option D is also correct and is a common way to print the elements of an array in reverse order. Option E does not compile and is therefore incorrect. You can declare multiple elements in a `for` loop, but the data type must be listed only once, such as in `for(int i=0, j=3; ...)`. Finally, option F is incorrect because the first element of the array is skipped. Since the conditional expression is checked before the loop is executed the first time, the first value of `i` used inside the body of the loop will be `1`.

8. G. The first two pattern matching statements compile without issue. The variable `bat` is allowed to be used again, provided it is no longer in scope. Line 36 does not compile, though. Due to flow scoping, if `s` is not a `Long`, then `bat` is not in scope in the expression `bat <= 20`. Line 38 also does not compile as `default` cannot be used as part of an `if/else` statement. For these two reasons, option G is correct.

9. B, C, E. The code contains a nested loop and a conditional expression that is executed if the sum of `col + row` is an even number; otherwise, `count` is incremented. Note that options E and F are equivalent to options B and D, respectively, since unlabeled statements apply to the most inner loop. Studying the loops, the first time the condition is true is in the second iteration of the inner loop, when `row` is `1` and `col` is `1`. Option A is incorrect because this causes the loop to exit immediately with `count` only being set to `1`. Options B, C, and E follow the same pathway. First, `count` is incremented to `1` on the first inner loop, and then

the inner loop is exited. On the next iteration of the outer loop, `row` is `2` and `col` is `0`, so execution exits the inner loop immediately. On the third iteration of the outer loop, `row` is `3` and `col` is `0`, so `count` is incremented to `2`. In the next iteration of the inner loop, the sum is even, so we exit, and our program is complete, making options B, C, and E each correct. Options D and F are both incorrect, as they cause the inner and outer loops to execute multiple times, with `count` having a value of `5` when done. You don't need to trace through all the iterations; just stop when the value of `count` exceeds `2`.

10. E. This code contains numerous compilation errors, making options A and H incorrect. Line 15 does not compile, as `continue` cannot be used inside a `switch` statement like this. Line 16 is not a compile-time constant since any `int` value can be passed as a parameter. Marking it `final` does not change this, so it doesn't compile. Line 18 does not compile because `Sunday` is not marked as `final`. Being effectively `final` is insufficient. Finally, line 19 does not compile because `DayOfWeek.MONDAY` is not an `int` value. While `switch` statements do support enum values, each `case` statement must have the same data type as the `switch` variable `otherDay`, which is `int`. The rest of the lines do compile. Since exactly four lines do not compile, option E is the correct answer.

11. A. The code compiles and runs without issue, printing `3` at runtime and making option A correct. The `default` statement on line 17 is optional since all the enum values are accounted for and can be removed without changing the output.

12. C. Prior to the first iteration, `sing = 8`, `squawk = 2`, and `notes = 0`. After the iteration of the first loop, `sing` is updated to `7`, `squawk` to `4`, and `notes` to the sum of the new values for `sing + squawk`, `7 + 4 = 11`. After the iteration of the second loop, `sing` is updated to `6`, `squawk` to `6`, and `notes` to the sum of itself plus the new values for `sing + squawk`, `11 + 6 + 6 = 23`. On the third iteration of the loop, `sing > squawk` evaluates to `false`, as `6 > 6` is `false`. The loop ends and the most recent value of `sing`, `23`, is output, so the correct answer is option C.

13. G. This example may look complicated, but the code does not compile. Line 8 is missing the required parentheses around the `boolean` conditional expression. Since the code does not compile and it is not because of line 6, option G is the correct answer. If line 8 was corrected with parentheses, then the loop would be executed twice, and the output would be `11`.

14. B, D, F. The code does compile, making option G incorrect. In the first for-each loop, the right side of the for-each loop has a type of `int[]`, so each element `penguin` has a type of `int`, making option B correct. In the second for-each loop, `ostrich` has a type of `Character[]`, so `emu` has a data type of `Character`, making option D correct. In the last for-each loop, `parrots` has a data type of `List<Integer>`. Since the generic type of `Integer` is used in the `List`, `macaw` will have a data type of `Integer`, making option F correct.

15. F. The code does not compile, although not for the reason specified in option E. The second `case` statement contains invalid syntax. Each `case` statement must have the keyword `case`—in other words, you cannot chain them with a colon (`:`). For this reason, option F is the correct answer. This line could have been fixed to say `case 'B', 'C'` or by adding the `case` keyword before `'C'`; then the rest of the code would have compiled and printed `great good` at runtime.

16. A, B, D. To print items in the `wolf` array in reverse order, the code needs to start with `wolf[wolf.length-1]` and end with `wolf[0]`. Option A accomplishes this and is the first correct answer. Option B is also correct and is one of the most common ways a reverse loop is written. The termination condition is often `m>=0` or `m>-1`, and both are correct. Options C and F each cause an `ArrayIndexOutOfBoundsException` at runtime since both read from `wolf[wolf.length]` first, with an index that is passed the length of the `0`-based array `wolf`. The form of option C would be successful if the value was changed to `wolf[wolf.length-z-1]`. Option D is also correct, as the `j` is extraneous and can be ignored in this example. Finally, option E is incorrect and produces an infinite loop, as `w` is repeatedly set to `r-1`, in this case `4`, on every loop iteration. Since the update statement has no effect after the first iteration, the condition is never met, and the loop never terminates.

17. B, E. The code compiles without issue and prints two distinct numbers at runtime, so options G and H are incorrect. The first loop executes a total of five times, with the loop ending when `participants` has a value of `10`. For this reason, option E is correct. In the second loop, `animals` starts out not less than or equal to `1`, but since it is a `do/while` loop, it executes at least once. In this manner, `animals` takes on a value of `3` and the loop terminates, making option B correct. Finally, the last loop executes a total of two times, with `performers` starting with `-1`, going to `1` at the end of the first loop, and then ending with a value of `3` after the second loop, which breaks the loop. This makes option B a correct answer twice over.

18. C, E. Pattern matching with an `if` statement is implemented using the `instanceof` operator, making option C correct and options A and B incorrect. Option D is incorrect as it is possible to access a pattern variable outside the `if` statement in which it is defined. Option E is a correct statement about flow scoping. Option F is incorrect. Pattern matching does not support declaring variables in `else` statements as `else` statements do not have a `boolean` expression.

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.

20. A, E. The most important thing to notice when reading this code is that the innermost loop is an infinite loop. Therefore, you are looking for solutions that skip the innermost loop entirely or that exit that loop. Option A is correct, as `break L2` on line 8 causes the second inner loop to exit every time it is entered, skipping the innermost loop entirely. For option B, the first `continue` on line 8 causes the execution to skip the innermost loop on the first iteration of the second loop but not the second iteration of the second loop. The innermost loop is executed, and with `continue` on line 12, it produces an infinite loop at runtime, making option B incorrect. Option C is incorrect because it contains a compiler error. The label `L3` is not visible outside its loop. Option D is incorrect, as it is equivalent to option B since the unlabeled `break` and `continue` apply to the nearest loop and therefore produce an infinite loop at runtime. Like option A, the `continue L2` on line 8 allows the innermost loop to be executed the second time the second loop is called. The `continue L2` on line 12 exits the infinite loop, though, causing control to return to the second loop. Since the first and second loops terminate, the code terminates, and option E is a correct answer.

21. E. Line 22 does not compile because `Long` is not a compatible type for a `switch` statement or expression. Line 23 does not compile because it is missing a semicolon after `"Jane"` and 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 four lines need to be corrected, option E is correct.

22. E. The code compiles without issue, making options F and G incorrect. Remember, `var` is supported in both `switch` and `while` loops, provided the compiler determines that the type is compatible with these statements. In addition, the variable `one` is allowed in a `case` statement because it is a `final` local variable, making it a compile-time constant. The value of `tailFeathers` is `3`, which matches the second `case` statement, making `5` the first output. The `while` loop is executed twice, with the pre-increment operator (`--`) modifying the value of `tailFeathers` from `3` to `2` and then to `1` on the second loop. For this reason, the final output is `5 2 1`, making option E the correct answer.

23. F. Line 19 starts with an `else` statement, but there is no preceding `if` statement that it matches. For this reason, line 19 does not compile, making option F the correct answer. If the `else` keyword was removed from line 19, then the code snippet would print `Success`.

24. G. The statement is not a valid for-each loop (or a traditional `for` loop) since it uses a nonexistent `in` keyword. For this reason, the code does not compile, and option G is correct. If the `in` was changed to a colon (`:`), then `Set`, `int[]`, and `Collection` would be correct.

25. D. The code compiles without issue, so option F is incorrect. The `viola` variable created on line 8 is never used and can be ignored. If it had been used as the `case` value on line 15, it would have caused a compilation error since it is not 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.

26. F. The code snippet does not contain any compilation errors, so options D and E are incorrect. There is a problem with this code snippet, though. While it may seem complicated, the key is to notice that the variable `r` is updated outside of the `do/while` loop. This is allowed from a compilation standpoint, since it is defined before the loop, but it means the innermost loop never breaks the termination condition `r <= 1`. At runtime, this will produce an infinite loop the first time the innermost loop is entered, making option F the correct answer.

27. F. Line 27 does not compile because the `case` block does not yield a value if `name` is not equal to `Frog`. For this reason, option F is correct. Every path within a `case` block must yield a value if the `switch` expression is expected to return a value.

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.

29. C. Since the pre-increment operator was used, the first value that will be displayed is `-1`, so options A and B are incorrect. On the second-to-last iteration of the loop, `y` will be incremented to `5`, and the loop will output `5`. The loop will continue since `5 <= 5` is `true`, and on the last iteration, `6` will be output. At the end of this last iteration, the `boolean` expression `6 <= 5` will evaluate to `false`, and the loop will terminate. Since `6` was the last value output by the loop, the answer is option C.

Chapter 4: Core APIs

1. F. Line *5* does not compile. This question is checking to see whether you are paying attention to the types. `numFish` is an `int`, and `1` is an `int`. Therefore, we use numeric addition and get *5*. The problem is that we can't store an `int` in a `String` variable. Suppose line *5* said `String anotherFish = numFish + 1 + "";`. In that case, the answers would be option A and option C. The variable defined on line *5* would be the string `"5"`, and both output statements would use concatenation.

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 a multidimensional 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. Option B throws an exception because there is no March 40. Option E also throws an exception because 2023 isn't a leap year and therefore has no February 29. Option F doesn't compile because the enum should be named `Month`, rather than `MonthEnum`. Option D is correct because it is just a regular date and has nothing to do with daylight saving time. Options A and C are correct because Java is smart enough to adjust for daylight saving time.

4. A, C, D. The code compiles fine. Line 3 points to the `String` in the string pool. Line 4 calls the `String` constructor explicitly and is therefore a different object than `s`. Line *5* checks for object equality, which is true, and so it prints `one`. Line 6 uses object reference equality, which is not true since we have different objects. Line 7 calls `intern()`, which returns the value from the string pool and is therefore the same reference as `s`. Line 8 also compares references but is true since both references point to the object from the string pool. Finally, line 9 is a trick. The string `Hello` is already in the string pool, so calling `intern()` does not change anything. The reference `t` is a different object, so the result is still `false`.

5. B. This example uses method chaining. After the call to `append()`, `sb` contains `"aaa"`. That result is passed to the first `insert()` call, which inserts at index 1. At this point, `sb` contains `abbaa`. That result is passed to the final `insert()`, which inserts at index 4, resulting in `abbaccca`.

6. C. Remember to watch return types on math operations. One of the tricks is 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. When dealing with time zones, it is best to convert to GMT first by subtracting the time zone. Remember that subtracting a negative is like adding. The first date/time is 9:00 GMT, and the second is 15:00 GMT. Therefore, the first one is earlier by six hours.

8. A, B, F. Remember that indexes are zero-based, which means index 4 corresponds to `5`, and option A is correct. For option B, the `replace()` method starts the replacement at index 2 and ends before index 4. This means two characters are replaced, and `charAt(3)` is called on the intermediate value of `1265`. The character at index 3 is `5`, making option B correct. Option C is similar, making the intermediate value 126 and returning `6`.

Option D results in an exception since there is no character at index 5. Option E is incorrect. It does not compile because the parentheses for the `length()` method are missing. Finally, option F's `replace` results in the intermediate value `145`. The character at index 2 is `5`, so option F is correct.

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. An array does 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. All of these lines compile. The `min()` and `floor()` methods return the same type passed in: `int` and `double`, respectively. The `round()` method returns a `long` when called with a `double`. Option A is correct since the code compiles.

11. E. A `LocalDate` does not have a time element. Therefore, there is no method to add hours, making option E the answer.

12. A, D, E. First, notice that the `indent()` call adds a blank space to the beginning of `numbers`, and `stripLeading()` immediately removes it. Therefore, these methods cancel each other out and have no effect. 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 2 since it needs to stop before index 3. 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. B. A `String` is immutable. Calling `concat()` returns a new `String` but does not change the original. A `StringBuilder` is mutable. Calling `append()` adds characters to the existing character sequence along with returning a reference to the same object. Therefore, option B is correct.

14. A, F. Option A correctly creates the current instant. Option F is also a proper conversion. Option B is incorrect because `Instant`, like many other date/time classes, does not have a `public` constructor and is instantiated via methods. Options C, D, and E are incorrect because the source object does not represent a point in time. Without a time zone, Java doesn't know what moment in time to use for the `Instant`.

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, G. There are 11 characters in `base` because there are two escape characters. The `\n` counts as one character representing a new line, and the `\\` counts as one character representing a backslash. This makes option B one of the answers. The `indent()` method adds two characters to the beginning of each of the two lines of `base`. This gives us four additional characters. However, the method also normalizes by adding a new line to the end if it is missing. The extra character means we add five characters to the existing 11, which is option G. Finally, the `translateEscapes()` method turns any text escape characters into actual escape characters, making `\\t` into `\t`. This gets rid of one character, leaving us with 10 characters matching option A.

17. 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 as 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 `StringIndexOutOfBoundsException`. Finally, option H is incorrect because it returns an empty `String`.

18. C, F. This question is tricky because it has several parts. First, you have to know that the text block on lines 13 and 14 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 17–19 are ignored. Finally, on line 20, something happens. We concatenate three new characters to `s1` and now have a `String` of length 7, making option C correct.

Next, `s2 += 2` expands to `s2 = s2 + 2`. A `String` concatenated with any other type gives a `String`. Lines 22, 23, and 24 all append to `s2`, giving a result of `"2cfalse"`. The `if` statement on line 27 returns `true` because the values of the two `String` objects are the same using object equality. The `if` statement on line 26 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.

19. A, B, D. The `compare()` method returns a positive integer when the arrays are different and the first is larger. This is the case for option A since the element at index 1 comes first alphabetically. It is not the case for option C because the `s4` is longer or for option E because the arrays are the same.

The `mismatch()` method returns a positive integer when the arrays are different in a position index 1 or greater. This is the case for options B and D since the difference is at index 1. It is not the case for option F because there is no difference.

20. A, D. The `dateTime1` object has a time of 1:30 per initialization. The `dateTime2` object is an hour later. However, there is no 2:30 when springing ahead, setting the time to 3:30. Option A is correct because it is an hour later. Option D is also correct because the hour of the new time is `3`. Option E is not correct because we have changed the time zone offset due to daylight saving time.

21. A, C. The `reverse()` method is the easiest way of reversing the characters in a `StringBuilder`; therefore, option A is correct. In option B, `substring()` returns a `String`, which is not stored anywhere. Option C uses method chaining. First, it creates the value `"JavavaJ$"`. Then, it removes the first three characters, resulting in `"avaJ$"`. Finally, it removes the last character, resulting in `"avaJ"`. Option D throws an exception because you cannot delete the character after the last index. Remember that `deleteCharAt()` uses indexes that are zero-based, and `length()` counts the number of characters rather than the index.

22. A. The date starts out as April 30, 2022. Since dates are immutable and the plus methods' return values are ignored, the result is unchanged. Therefore, option A is correct.

Chapter 5: Methods

1. A, E. Instance and `static` variables can be marked `final`, making option A correct. Effectively final means a local variable is not marked `final` but whose value does not change after it is set, making option B incorrect. Option C is incorrect, as `final` refers only to the reference to an object, not its contents. Option D is incorrect, as `var` and `final` can be used together. Finally, option E is correct: once a primitive is marked `final`, it cannot be modified.

2. B, C. The keyword `void` is a return type. Only the access modifier or optional specifiers are allowed before the return type. Option C is correct, creating a method with `private` access. Option B is also correct, creating a method with package access and the optional specifier `final`. Since package access does not use a modifier, we get to jump right to `final`. Option A is incorrect because package access omits the access modifier rather than specifying `default`. Option D is incorrect because Java is case sensitive. It would have been correct if `public` were the choice. Option E is incorrect because the method already has a `void` return type. Option F is incorrect because labels are not allowed for methods.

3. A, D. Options A and D are correct because the optional specifiers are allowed in any order. Options B and C are incorrect because they each have two return types. Options E and F are incorrect because the return type is before the optional specifier and access modifier, respectively.

4. A, B, C, E. The value `6` can be implicitly promoted to any of the primitive types, making options A, C, and E correct. It can also be autoboxed to `Integer`, making option B correct. It cannot be both promoted and autoboxed, making options D and F incorrect.

5. A, C, D. Options A and C are correct because a `void` method is optionally allowed to have a `return` statement as long as it doesn't try to return a value. Option B does not compile because `null` requires a reference object as the return type. Since `int` is primitive, it is not a reference object. Option D is correct because it returns an `int` value. Option E does not compile because it tries to return a `double` when the return type is `int`. Since a `double` cannot be assigned to an `int`, it cannot be returned as one either. Option F does not compile because no value is actually returned.

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. D, F. 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. Option D is correct. A common practice is to set all fields to be `private` and all methods to be `public`. Option A is incorrect because `protected` access allows everything that package access allows and additionally allows subclasses access. Option B is incorrect because the class is `public`. This means that other classes can see the class. However, they cannot call any of the methods or read any of the fields. It is essentially a useless class. Option C is incorrect because package access applies to the whole package. Option E is incorrect because Java has no such wildcard access capability.

9. B, C, D, F. The two classes are in different packages, which means `private` access and package access will not compile. This causes compiler errors on lines 5, 6, and 7, making options B, C, and D correct answers. Additionally, `protected` access will not compile since `School` does not inherit from `Classroom`. This causes the compiler error on line 9, making option F a correct answer as well.

10. B. `Rope` runs line 3, setting `LENGTH` to 5, and then immediately after that runs the `static` initializer, which sets it to 10. Line 5 in the `Chimp` class calls the `static` method normally and prints `swing` and a space. Line 6 also calls the `static` method. Java allows calling a `static` method through an instance variable, although it is not recommended. Line 7 uses the `static` import on line 2 to reference `LENGTH`. For these reasons, option B is correct.

11. B, E. Line 10 does not compile because `static` methods are not allowed to call instance methods. Even though we are calling `play()` as if it were an instance method and an instance exists, Java knows `play()` is really a `static` method and treats it as such. Since this is the only line that does not compile, option B is correct. If line 10 is removed, the code prints `swing-swing`, making option E correct. It does not throw a `NullPointerException` on line 17 because `play()` is a `static` method. Java looks at the type of the reference for `rope2` and translates the call to `Rope.play()`.

12. B. The test for effectively final is if the `final` modifier can be added to the local variable and the code still compiles. The `monkey` variable declared on line 11 is not effectively final because it is modified on line 13. The `giraffe` and `name` variables declared on lines 13 and 14, respectively, are effectively final and not modified after they are set. The `name` variable declared on line 17 is not effectively final since it is modified on line 22. Finally, the `food` variable on line 18 is not effectively final since it is modified on line 20. Since there are two effectively final variables, option B is correct.

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 `8`, making option D correct.

14. E. If a variable is `static final`, it must be set exactly once, and it must be in the declaration line or in a `static` initialization block. Line 4 doesn't compile because `bench` is not set in either of these locations. Line 15 doesn't compile because `final` variables are not allowed to be set after that point. Line 11 doesn't compile because `name` is set twice: once in the declaration and again in the `static` block. Line 12 doesn't compile because `rightRope` is set twice as well. Both are in `static` initialization blocks. Since four lines do not compile, option E is correct.

15. B. The two valid ways to do this are `import static java.util.Collections.*;` and `import static java.util.Collections.sort;`. 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. The argument on line 17 is a `short`. It can be promoted to an `int`, so `print()` on line 5 is invoked. The argument on line 18 is a `boolean`. It can be autoboxed to a `Boolean`, so `print()` on line 11 is invoked. The argument on line 19 is a `double`. It can be autoboxed to a `Double`, so `print()` on line 11 is invoked. Therefore, the output is `int-Object-Object-`, and the correct answer is option E.

17. B. Since Java is pass-by-value and the variable on line 8 never gets reassigned, it stays as 9. In the method `square`, `x` starts as 9. The `y` value becomes 81, and then `x` gets set to –1. Line 9 does set `result` to 81. However, we are printing out `value`, and that is still 9, making option B correct.

18. B, D, E. Since Java is pass-by-value, assigning a new object to `a` does not change the caller. Calling `append()` does affect the caller because both the method parameter and the caller have a reference to the same object. Finally, returning a value does pass the reference to the caller for assignment to `s3`. For these reasons, options B, D, and E are correct.

19. B, C, E. The variable `value1` is a `final` instance variable. It can be set only once: in the variable declaration, an instance initializer, or a constructor. Option A does not compile because the `final` variable was already set in the declaration. The variable `value2` is a `static` variable. Both instance and `static` initializers are able to access `static` variables, making options B and E correct. The variable `value3` is an instance variable. Options D and F do not compile because a `static` initializer does not have access to instance variables.

20. A, E. The `100` parameter is an `int` and so calls the matching `int` method, making option A correct. When this method is removed, Java looks for the next most specific constructor. Java prefers autoboxing to varargs, so it chooses the `Integer` constructor. The `100L` parameter is a `long`. Since it can't be converted into a smaller type, it is autoboxed into a `Long`, and then the method for `Object` is called, making option E correct.

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 valid method overloads because the types of parameters in the list 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.

Chapter 6: Class Design

1. E. Options A and B will not compile because constructors cannot be called without `new`. Options C and D will compile but will create a new object rather than setting the fields in this one. The result is the program will print `0`, not `2`, at runtime. Calling an overloaded constructor, using `this()`, or a parent constructor, using `super()`, is only allowed on the first line of the constructor, making option E correct and option F incorrect. Finally, option G is incorrect because the program prints `0` without any changes, not `2`.

2. A, B, F. The `final` modifier can be used with `private` and `static`, making options A and F correct. Marking a `private` method `final` is redundant but allowed. A `private` method may also be marked `static`, making option B correct. Options C, D, and E are incorrect because methods marked `static`, `private`, or `final` cannot be overridden; therefore, they cannot be marked `abstract`.

3. B, C. Overloaded methods have the same method name but a different signature (the method parameters differ), making option A incorrect. Overridden instance methods and hidden `static` methods must have the same signature (the name and method parameters must match), making options B and C correct. Overloaded methods can have different return types, while overridden and hidden methods can have covariant return types. None of these methods are required to use the same return type, making options D, E, and F incorrect.

4. F. The code will not compile as is, because the parent class `Mammal` does not define a no-argument constructor. For this reason, the first line of a `Platypus` constructor should be an explicit call to `super(int)`, making option F the correct answer. Option E is incorrect, as line 7 compiles without issue. The `sneeze()` method in the `Mammal` class is marked `private`, meaning it is not inherited and therefore is not overridden in the `Platypus` class. For this reason, the `sneeze()` method in the `Platypus` class is free to define the same method with any return type.

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`. Options A, B, C, and D all print `0` at runtime. Option E is the only 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. The `Moose` class doesn't compile, as the `final` variable `antlers` is not initialized when it is declared, in an instance initializer, or in a constructor. `Caribou` and `Reindeer` are not immutable because they are not marked `final`, which means a subclass could extend them and add mutable fields. `Elk` and `Deer` are both immutable classes since they are marked `final` and only include `private final` members, making options D and E correct. As shown with `Elk`, a class doesn't need to declare any fields to be considered immutable.

7. A. The code compiles and runs without issue, so options E and F are incorrect. The `Arthropod` class defines two overloaded versions of the `printName()` method. The `printName()` method that takes an `int` value on line 5 is correctly overridden in the `Spider` class on line 9. Remember, an overridden method can have a broader access modifier, and `protected` access is broader than package access. Because of polymorphism, the overridden method replaces the method on all calls, even if an `Arthropod` reference variable is used, as is done in the `main()` method. For these reasons, the overridden method is called on lines 14 and 15, printing `Spider` twice. Note that the `short` value is automatically cast to the larger type of `int`, which then uses the overridden method. Line 16 calls the overloaded method in the `Arthropod` class, as the `long` value `5L` does not match the overridden method, resulting in `Arthropod` being printed. Therefore, option A is the correct answer.

8. D. The code compiles without issue. The question is making sure you know that superclass constructors are called in the same manner in abstract classes as they are in non-abstract classes. Line 9 calls the constructor on line 6. The compiler automatically inserts `super()` as the first line of the constructor defined on line 6. The program then calls the constructor on line 3 and prints `Wow-`. Control then returns to line 6, and `Oh-` is printed. Finally, the method call on line 10 uses the version of `fly()` in the `Pelican` class, since it is marked `private` and the reference type of `var` is resolved as `Pelican`. The final output is `Wow-Oh-Pelican`, making option D the correct answer. Remember that `private` methods cannot be overridden. If the reference type of `chirp` was `Bird`, then the code would not compile as it would not be accessible outside the class.

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. Option A is correct, as `this(3)` calls the constructor declared on line 5, while `this("")` calls the constructor declared on line 10. Option B does not compile, as inserting `this()` at line 3 results in a compiler error, since there is no matching constructor. Option C is correct, as `short` can be implicitly cast to `int`, resulting in `this((short)1)` calling the constructor declared on line 5. In addition, `this(null)` calls the `String` constructor declared on line 10. Option D does not compile because inserting `super()` on line 14 results in an invalid constructor call. The `Howler` class does not contain a no-argument constructor. Option E is also incorrect. Inserting `this(2L)` at line 3 results in a recursive constructor definition. The compiler detects this and reports an error. Option F is incorrect, as using `super(null)` on line 14 does not match any parent constructors. If an explicit cast was used, such as `super((Integer)null)`, then the code would have compiled but would throw an exception at runtime during unboxing. Finally, option G is incorrect because the superclass `Howler` does not contain a no-argument constructor. Therefore, the constructor declared on line 13 will not compile without an explicit call to an overloaded or parent constructor.

11. C. The code compiles and runs without issue, making options F and G incorrect. Line 16 initializes a `PolarBear` instance and assigns it to the `bear` reference. The variable declaration and instance initializers are run first, setting `value` to `tac`. The constructor declared on line 5 is called, resulting in `value` being set to `tacb`. Remember, a `static main()` method can access `private` constructors declared in the same class. Line 17 creates another `PolarBear` instance, replacing the `bear` reference declared on line 16. First, `value` is initialized to `tac` as before. Line 17 calls the constructor declared on line 8, since `String` is the narrowest match of a `String` literal. This constructor then calls the overloaded constructor declared on line 5, resulting in `value` being updated to `tacb`. Control returns to the previous constructor, with line 10 updating `value` to `tacbf`, and making option C the correct answer. Note that if the constructor declared on line 8 did not exist, then the constructor on line 12 would match. Finally, the `bear` reference is properly cast to `PolarBear` on line 18, making the `value` parameter accessible.

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. The compiler will insert a default no-argument constructor if the class compiles and does not define any constructors. Options A and G fulfill this requirement, making them the correct answers. The `bird()` declaration in option G is a method declaration, not a constructor. Options B and C do not compile. Since the constructor name does not match the class name, the compiler treats these as methods with missing return types. Options D, E, and F all compile, but since they declare at least one constructor, the compiler does not supply one.

14. B, E, F. A class can only directly extend a single class, making option A incorrect. A class can implement any number of interfaces, though, making option B correct. Option C is incorrect because primitive variables types do not inherit `java.lang.Object`. If a class extends another class, then it is a subclass, not a superclass, making option D incorrect. A class that implements an interface is 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. The code does not compile because the `isBlind()` method in `Nocturnal` is not marked `abstract` and does not contain a method body. The rest of the lines compile without issue, making option C the only correct answer. If the `abstract` modifier was added to line 2, then the code would compile and print `false` at runtime, making option B the correct answer.

16. D. The code compiles, so option G is incorrect. Based on order of initialization, the `static` components are initialized first, starting with the `Arachnid` class, since it is the parent of the `Scorpion` class, which initializes the `StringBuilder` to `u`. The `static` initializer in `Scorpion` then updates `sb` to contain `uq`, which is printed twice by lines 13 and 14 along with spaces separating the values. Next, an instance of `Arachnid` is initialized on line 15. There are two instance initializers in `Arachnid`, and they run in order, appending `cr` to the `StringBuilder`, resulting in a value of `uqcr`. An instance of `Scorpion` is then initialized on line 16. The instance initializers in the superclass `Arachnid` run first, appending `cr` again and updating the value of `sb` to `uqcrcr`. Finally, the instance initializer in `Scorpion` runs and appends `m`. The program completes with the final value printed being `uq uq uqcrcrm`, making option D the correct answer.

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. The `eat()` method is `private` in the `Mammal` class. Since it is not inherited in the `Primate` class, it is neither overridden nor overloaded, making options A and B incorrect. The `drink()` method in `Mammal` is correctly hidden in the `Monkey` class, as the signature is the same and both are `static`, making option D correct and option C incorrect. The version in the `Monkey` class throws a new exception, but it is unchecked; therefore, it is allowed. The `dance()` method in `Mammal` is correctly overloaded in the `Monkey` class because the signatures are not the same, making option E incorrect and option F correct. For methods to be overridden, the signatures must match exactly. Finally, line 12 is an invalid override and does not compile, as `int` is not covariant with `void`, making options G and H both incorrect.

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. 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. 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. 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.

23. B. The program compiles, making option F incorrect. The constructors are called from the child class upward, but since each line of a constructor is a call to another constructor, via `this()` or `super()`, they are ultimately executed in a top-down manner. On line 29, the `main()` method calls the `Fennec()` constructor declared on line 19. Remember, integer literals in Java are considered `int` by default. This constructor calls the `Fox()` constructor defined on line 12, which in turn calls the overloaded `Fox()` constructor declared on line 11. Since the constructor on line 11 does not explicitly call a parent constructor, the compiler inserts a call to the no-argument `super()` constructor, which exists on line 3 of the `Canine` class. Line 3 is then executed, adding `q` to the output, and the compiler chain is unwound. Line 11 then executes, adding `p`, followed by line 14, adding `z`. Finally, line 21 is executed, and `j` is added, resulting in a final value for `logger` of `qpzj`, and making option B correct. For the exam, remember to follow constructors from the lowest level upward to determine the correct pathway, but then execute them from the top down using the established order.

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. Concrete classes are, by definition, not `abstract`, so option A is incorrect. A concrete class must implement all inherited abstract methods, so option B is correct. Concrete classes can be optionally marked `final`, so option C is correct. Option D is incorrect; concrete classes need not be immutable. A concrete subclass only needs to override the inherited abstract method, not match the declaration exactly. For example, a covariant return type can be used. For this reason, option E is incorrect.

26. D. The classes are structured correctly, but the body of the `main()` method contains a compiler error. The `Orca` object is implicitly cast to a `Whale` reference on line 7. This is permitted because `Orca` is a subclass of `Whale`. By performing the cast, the `whale` reference on line 8 does not have access to the `dive(int... depth)` method. For this reason, line 8 does not compile, making option D correct.

Chapter 7: Beyond Classes

1. B, D. `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. `Newt` does not compile because records are immutable, so any mutator methods that modify fields are not permitted. Overriding the `equals()` method is allowed, though.

2. A, B, D, E. The code compiles without issue, so option G is incorrect. The blank can be filled with any class or interface that is a supertype of `TurtleFrog`. Option A is the direct superclass of `TurtleFrog`, and option B is the same class, so both are correct. `BrazilianHornedFrog` is not a superclass of `TurtleFrog`, so option C is incorrect. `TurtleFrog` inherits the `CanHop` interface, so option D is correct. Option E is also correct, as `var` is permitted when the type is known. Finally, `Long` is an unrelated class that is not a superclass of `TurtleFrog` and is therefore incorrect.

3. C. When an enum contains only a list of values, the semicolon (`;`) after the list is optional. When an enum contains any other members, such as a constructor or variable, the semicolon is required. Since the enum list does not end with a semicolon, the code does not compile, making option C the correct answer. If the missing semicolon were added, the program would print `0 1 2` at runtime.

4. C. A class extending a sealed class must be marked `final`, `sealed`, or `non-sealed`. Since `Armadillo` is missing a modifier, the code does not compile, and option C is correct.

5. E. First, the declarations of `HasExoskeleton` and `Insect` are correct and do not contain any errors, making options C and D incorrect. The concrete class `Beetle` extends `Insect` and inherits two abstract methods, `getNumberOfSections()` and `getNumberOfLegs()`. The `Beetle` class includes an overloaded version of `getNumberOfSections()` that takes an `int` value. The method declaration is valid, making option F incorrect, although it does not satisfy the abstract method requirement inherited from `HasExoskeleton`. For this reason, only one of the two abstract methods is properly overridden. The `Beetle` class therefore does not compile, and option E is correct.

6. D, E. Line 4 does not compile, since an `abstract` method cannot include a body. Line 7 also does not compile because the wrong keyword is used. A class implements an interface; it does not extend it. For these reasons, options D and E are correct.

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.

8. A, B, C. Instance variables must include the `private` access modifier, making option D incorrect. While it is common for methods to be `public`, this is not required. Options A, B, and C fulfill this requirement.

9. A, E, F. The `setSnake()` method requires an instance of `Snake`. `Cobra` is a direct subclass, while `GardenSnake` is an indirect subclass. For these reasons, options A and E are correct. Option B is incorrect because `Snake` is `abstract` and requires a concrete subclass for instantiation. Option C is incorrect because `Object` is a supertype of `Snake`, not a subtype. Option D is incorrect as `String` is an unrelated class and does not inherit `Snake`. Finally, a `null` value can always be passed as an object value, regardless of type, so option F is also correct.

10. A, B, C, E. `Walk` declares a private method that is not inherited in any of its subtypes. For this reason, any valid class is supported on line `X`, making options A, B, and C correct. Line Z is more restrictive, with only `ArrayList` or subtypes of `ArrayList` supported, making option E correct.

11. B. Starting with Java 16, inner classes can contain `static` variables, so the code compiles. Remember that `private` constructors can be used by any methods within the outer class. The `butter` reference on line 8 refers to the inner class variable defined on line 6, with the output being `10` at runtime, and making option B correct.

12. A, B, E. Encapsulation allows using methods to get and set instance variables so other classes are not directly using them, making options A and B correct. Instance variables must be `private` for this to work, making option E correct and option D incorrect. While there are common naming conventions, they are not required, making option C incorrect.

13. F. When using an enum in a `switch` expression, the `case` statement must be made up of the enum values only. If the enum name is used in the `case` statement value, then the code does not compile. In this question, `SPRING` is acceptable, but `Seasons.SPRING` is not. For this reason, the three `case` statements do not compile, making option F the correct answer. If these three lines were corrected, then the code would compile and produce a `NullPointerException` at runtime.

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 `nonsealed`. Finally, option F is incorrect, as sealed classes can contain nested subclasses.

15. G. Trick question—the code does not compile! For this reason, option G 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 options C and F would be correct. Option A would print `Booo!!!`, while options B, D, and E would not compile for various reasons.

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. Lines 2 and 3 compile with interface variables implicitly `public`, `static`, and `final`. Line 4 also compiles, as `static` methods are implicitly `public`. Line 5 does not compile, making option E correct. Non-`static` interface methods with a body must be explicitly marked `private` or `default`. Line 6 compiles, with the `public` modifier being added by the compiler. Line 7 does not compile, as interfaces do not have `protected` members, making option G correct. Finally, line 8 compiles without issue.

18. E. `Diet` is an inner class, which requires an instance of `Deer` to instantiate. Since the `main()` method is `static`, there is no such instance. Therefore, the `main()` method does not compile, and option E is correct. If a reference to `Deer` were used, such as calling `new Deer().new Diet()`, then the code would compile and print `b` at runtime.

19. G. The `isHealthy()` method is marked `abstract` in the enum; therefore, it must be implemented in each enum value declaration. Since only `INSECTS` implements it, the code does not compile, making option G correct.

20. A, D, F. Polymorphism is the property of an object to take on many forms. Part of polymorphism is that methods are replaced through overriding wherever they are called, regardless of whether they're in a parent or child class. For this reason, option A is correct, and option E is incorrect. With hidden `static` methods, Java relies on the location and reference type to determine which method is called, making option B incorrect and option F correct. Finally, making a method `final`, not `static`, prevents it from being overridden, making option D correct and option C incorrect.

21. F. The record defines an overloaded constructor using parentheses, not a compact one. For this reason, the first line must be a call to another constructor, such as `this(500, "Acme", LocalDate.now())`. For this reason, the code does not compile and option F is correct. If the parentheses were removed from the constructor to declare a compact constructor, then options A, C, and E would be correct.

22. C, D, G. Option C correctly creates an instance of an inner class `Cub` using an instance of the outer class `Lion`. Options A, B, E, and H use incorrect syntax for creating an instance of the `Cub` class. Options D and G correctly create an instance of the `static` nested `Den` class, which does not require an instance of `Lion`, while option F uses invalid syntax.

23. D. First, if a class or interface inherits two interfaces containing `default` methods with the same signature, it must override the method with its own implementation. The `Penguin` class does this correctly, so option E is incorrect. The way to access an inherited `default` method is by using the syntax `Swim.super.perform()`, making option D correct. We agree that the syntax is bizarre, but you need to learn it. Options A, B, and C are incorrect and result in compiler errors.

24. B, E. Line 3 does not compile because the `static` method `hunt()` cannot access an `abstract` instance method `getName()`, making option B correct. Line 6 does not compile because the `private static` method `sneak()` cannot access the `private` instance method `roar()`, making option E correct. The rest of the lines compile without issue.

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, F. Enums are required to have a semicolon (`;`) after the list of values if there is anything else in the enum. Don't worry; you won't be expected to track down missing semicolons on the whole exam—only on enum questions. For this reason, line 5 should have a semicolon after it since it is the end of the list of enums, making option F correct. Enum constructors are implicitly `private`, making option C correct as well. The rest of the enum compiles without issue.

27. B, C, D, G. The compiler inserts an accessor for each field, a constructor containing all of the fields in the order they are declared, and useful implementations of `equals()`, `hashCode()`, and `toString()`, making options B, C, D, and G correct. Option A is incorrect, as the compiler would only insert a no-argument constructor if the record had no fields. Option E is incorrect, as records are immutable. Option F is also incorrect and not a property of records.

28. A, B, D. `Camel` does not compile because the `travel()` method does not declare a body, nor is it marked `abstract`, making option A correct. `EatsGrass` also does not compile because an interface method cannot be marked both `private` and `abstract`, making option B correct. Finally, `Eagle` does not compile because it declares an abstract method `soar()` in a concrete class, making option D correct. The other classes compile without issue.

29. F. The code does not compile, so options A through C are incorrect. Both lines 5 and 12 do not compile, as `this()` is used instead of `this`. Remember, `this()` refers to calling a constructor, whereas `this` is a reference to the current instance. Next, the compiler does not allow casting to an unrelated class type. Since `Orangutan` is not a subclass of `Primate`, the cast on line 15 is invalid, and the code does not compile. Due to these three lines containing compilation errors, option F is the correct answer.

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.

Chapter 8: Lambdas and Functional Interfaces

1. A. This code is correct. Line 8 creates a lambda expression that checks whether the age is less than 5, making option A correct. Since there is only one parameter and it does not specify a type, the parentheses around the parameter are optional. Lines 11 and 13 use the `Predicate` interface, which declares a `test()` method.

2. C. The interface takes two `int` parameters. The code on line 7 attempts to use them as if `h` is a `String` making option C correct. It is tricky to use types in a lambda when they are implicitly specified. Remember to check the interface for the real type.

3. A, C. A functional interface can contain any number of non-abstract methods, including `default`, `private`, `static`, and `private static`. For this reason, option A is correct, and option D is incorrect. Option B is incorrect, as classes are never considered functional interfaces. A functional interface contains exactly one abstract method, although methods that have matching signatures as `public` methods in `java.lang.Object` do not count toward the single method test. For these reasons, option C is correct. Finally, option E is incorrect. While a functional interface can be marked with the `@FunctionalInterface` annotation, it is not required.

4. A, F. Option B is incorrect because it does not use the `return` keyword. Options C, D, and E are incorrect because the variable `e` is already in use from the lambda and cannot be redefined. Additionally, option C is missing the `return` keyword, and option E is missing the semicolon. Therefore, options A and F are correct.

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. `Predicate<String>` takes a parameter list of one parameter using the specified type. Options E and F are incorrect because they specify the wrong type. Options B and D are incorrect because they use the wrong syntax for the arrow operator. This leaves us with options A and C as the answers.

7. E. While there appears to have been a variable name shortage when this code was written, it does compile. Lambda variables and method names are allowed to be the same. The `x` lambda parameter is scoped to within each lambda, so it is allowed to be reused. The type is inferred by the method it calls. The first lambda maps `x` to a `String` and the second to a `Boolean`. Therefore, option E is correct.

8. E. The question starts with a `UnaryOperator<Integer>`, which takes one parameter and returns a value of the same type. Therefore, option E is correct, as `UnaryOperator` extends `Function`. Notice that other options don't even compile because they have the wrong number of generic types for the functional interface provided. You should know that a `BiFunction<T,U,R>` takes three generic arguments, a `BinaryOperator<T>` takes one generic argument, and a `Function<T,R>` takes two generic arguments.

9. A, F. Option A is correct and option B is incorrect because a `Supplier` returns a value while a `Consumer` takes one and acts on it. Option C is tricky. `IntSupplier` does return an `int`. However, the option asks about `IntegerSupplier`, which doesn't exist. Option D is incorrect because a `Predicate` returns a `boolean`. It does have a method named `test()`, making option F correct. Finally, option E is incorrect because `Function` has an `apply()` method.

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.

11. D. The code does not compile because the lambdas are assigned to `var`. The compiler does not have enough information to determine they are of type `Predicate<String>`. Therefore, option D is correct.

12. A. The `a.compose(b)` method calls the `Function` parameter `b` before the reference `Function` variable `a`. In this case, that means that we multiply by `3` before adding `4`. This gives a result of `7`, making option A correct.

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. 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. 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. Lambdas are not allowed to redeclare local variables, making options A and B incorrect. Option D is incorrect because setting `end` prevents it from being effectively final. Lambdas are only allowed to reference `final` or effectively final variables. Option C compiles since `chars` is not used.

17. C. Line 8 uses braces around the body. This means the `return` keyword and semicolon are required. Since the code doesn't compile, option C is the answer.

18. B, F, G. We can eliminate four choices right away. Options A and C are there to mislead you; these interfaces don't exist. Option D is incorrect because a `BiFunction<T,U,R>` takes three generic arguments, not two. Option E is incorrect because none of the examples returns a `boolean`.

The declaration on line 6 doesn't take any parameters, and it returns a `String`, so a `Supplier<String>` can fill in the blank, making option F correct. The declaration on line 7 requires you to recognize that `Consumer` and `Function`, along with their binary equivalents, have an `andThen()` method. This makes option B correct. Finally, line 8 takes a single parameter, and it returns the same type, which is a `UnaryOperator`. Since the types are the same, only one generic parameter is needed, making option G correct.

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 a 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. 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`.

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 to test whether an interface is a functional interface is to apply the `@FunctionalInterface` annotation and check if the code still compiles.

Chapter 9: Collections and Generics

1. A, E. For the first scenario, the answer needs to implement `List` because the scenario allows duplicates, narrowing it down to options A and D. Option A is a better answer than option D because `LinkedList` is both a `List` and a `Queue`, and you just need a regular `List`.

For the second scenario, the answer needs to implement `Map` because you are dealing with key/value pairs per the unique `id` field. This narrows it down to options B and E. Since the question talks about ordering, you need the `TreeMap`. Therefore, the answer is option E.

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. 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. 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. B. The record compiles and runs without issue. Line 8 gives a compiler warning for not using generics but not a compiler error. Line 7 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. 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 ascending order. Finally, it reverses the result. This is just what we want, so option F is correct.

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. A. The array is sorted using `MyComparator`, which sorts the elements in reverse alphabetical order in a case-insensitive fashion. Normally, numbers sort before letters. This code reverses that by calling the `compareTo()` method on `b` instead of `a`. Therefore, option A is correct.

9. 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. A, B, E, F. The `forEach()` method works with a `List` or a `Set`. Therefore, options A and B are correct. Additionally, options E and F return a `Set` 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.

11. 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`.

12. 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.

13. A. When using `binarySearch()`, the `List` must be sorted in the same order that the `Comparator` uses. Since the `binarySearch()` method does not specify a `Comparator` explicitly, the default sort order is used. Only `c2` uses that sort order and correctly identifies that the value `2` is at index `0`. Therefore, option A is correct. The other two comparators sort in descending order. Therefore, the precondition for `binarySearch()` is not met, and the result is undefined for those two. The two calls to `reverse()` are just there to distract you; they cancel each other out.

14. A, B. `Y` is both a class and a type parameter. This means that within the class `Z`, 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.

15. 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]`.

16. 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`.

17. 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.

18. B. When using generic types in a method, the generic specification goes before the return type and option B is correct.

19. F. The first call to `merge()` calls the mapping function and adds the numbers to get `13`. It then updates the map. The second call to `merge()` sees that the map currently has a `null` value for that key. It does not call the mapping function but instead replaces it with the new value of `3`. Therefore, option F is correct.

20. B, D, F. The `java.lang.Comparable` interface is implemented on the object to compare. It specifies the `compareTo()` method, which takes one parameter. The `java.util.Comparator` interface specifies the `compare()` method, which takes two parameters. This gives us options B, D, and F as the answers.

Chapter 10: Streams

1. D. No terminal operation is called, so the stream never executes. The first line creates an infinite stream reference. If the stream were executed on the second line, it would get the first two elements from that infinite stream, `""` and `"1"`, and add an extra character, resulting in `"2"` and `"12"`, respectively. Since the stream is not executed, the reference is printed instead, giving us option D.

2. F. Both streams created in this code snippet are infinite streams. The variable `b1` is set to `true` since `anyMatch()` terminates. Even though the stream is infinite, Java finds a match on the first element and stops looking. However, when `allMatch()` runs, it needs to keep going until the end of the stream since it keeps finding matches. Since all elements continue to match, the program hangs, making option F the answer.

3. E. An infinite stream is generated where each element is twice as long as the previous one. While this code uses the three-parameter `iterate()` method, the condition is never `false`. The variable `b1` is set to `false` because Java finds an element that matches when it gets to the element of length `4`. However, the next line tries to operate on the same stream. Since streams can be used only once, this throws an exception that the "stream has already been operated upon or closed" and making option E the answer. If two different streams were used, the result would be option B.

4. A, B. Terminal operations are the final step in a stream pipeline. Exactly one is required, because it triggers the execution of the entire stream pipeline. Therefore, options A and B are correct. Option C is true of intermediate operations rather than terminal operations. Option D is incorrect because `peek()` is an intermediate operation. Finally, option E is incorrect because once a stream pipeline is run, the `Stream` is marked invalid.

5. C, F. Yes, we know this question is a lot of reading. Remember to look for the differences between options rather than studying each line. These options all have much in common. All of them start out with a `LongStream` and attempt to convert it to an `IntStream`. However, options B and E are incorrect because they do not cast the `long` to an `int`, resulting in a compiler error on the `mapToInt()` calls.

Next, we hit the second difference. Options A and D are incorrect because they are missing `boxed()` before the `collect()` call. Since `groupingBy()` is creating a `Collection`, we need a nonprimitive `Stream`. The final difference is that option F specifies the type of `Collection`. This is allowed, though, meaning both options C and F are correct.

6. A. Options C and D do not compile because these methods do not take a `Predicate` parameter and do not return a `boolean`. When working with streams, it is important to remember the behavior of the underlying functional interfaces. Options B and E are incorrect. While the code compiles, it runs infinitely. The stream has no way to know that a match won't show up later. Option A is correct because it is safe to return `false` as soon as one element passes through the stream that doesn't match.

7. F. There is no `Stream<T>` method called `compare()` or `compareTo()`, so options A through D can be eliminated. The `sorted()` method is correct to use in a stream pipeline to return a sorted `Stream`. The `collect()` method can be used to turn the stream into a `List`. The `collect()` method requires a collector be selected, making option E incorrect and option F correct.

8. D, E. The `average()` method returns an `OptionalDouble` since averages of any type can result in a fraction. Therefore, options A and B are both incorrect. The `findAny()` method returns an `OptionalInt` because there might not be any elements to find. Therefore, option D is correct. The `sum()` method returns an `int` rather than an `OptionalInt` because the sum of an empty list is zero. Therefore, option E is correct.

9. B, D. Lines 4–6 compile and run without issue, making option F incorrect. Line 4 creates a stream of elements `[1, 2, 3]`. Line 5 maps the stream to a new stream with values `[10, 20, 30]`. Line 6 filters out all items not less than 5, which in this case results in an empty stream. For this reason, `findFirst()` returns an empty `Optional`.

Option A does not compile. It would work for a `Stream<T>` object, but we have a `LongStream` and therefore need to call `getAsLong()`. Option C also does not compile, as it is missing the `::` that would make it a method reference. Options B and D both compile and run without error, although neither produces any output at runtime since the stream is empty.

10. F. Only one of the method calls, `forEach()`, is a terminal operation, so any answer in which M is not the last line will not execute the pipeline. This eliminates all but options C, E, and F. Option C is incorrect because `filter()` is called before `limit()`. Since none of the elements of the stream meets the requirement for the `Predicate<String>`, the `filter()` operation will run infinitely, never passing any elements to `limit()`. Option E is incorrect because there is no `limit()` operation, which means that the code would run infinitely. Only option F is correct. It first limits the infinite stream to a finite stream of ten elements and then prints the result.

11. B, C, E. As written, the code doesn't compile because the `Collectors.joining()` expects to get a `Stream<String>`. Option B fixes this, at which point nothing is output because the collector creates a `String` without outputting the result. Option E fixes this and causes the output to be `11111`. Since the post-increment operator is used, the stream contains an infinite number of the character `1`. Option C fixes this and causes the stream to contain increasing numbers.

12. F. The code does not compile because `Stream.concat()` takes two parameters, not the three provided. This makes the answer option F.

13. F. If the `map()` and `flatMap()` calls were reversed, option B would be correct. In this case, the `Stream` created from the source is of type `Stream<List>`. Trying to use the addition operator (`+`) on a `List` is not supported in Java. Therefore, the code does not compile, and option F is correct.

14. B, D. Line 4 creates a `Stream` and uses autoboxing to put the `Integer` wrapper of `1` inside. Line 5 does not compile because `boxed()` is available only on primitive streams like `IntStream`, not `Stream<Integer>`. This makes option B one answer. Line 6 converts to a `double` primitive, which works since `Integer` can be unboxed to a value that can be implicitly cast to a `double`. Line 7 does not compile for two reasons making option D the second answer. First, converting from a `double` to an `int` would require an explicit cast. Also, `mapToInt()` returns an `IntStream`, so the data type of `s2` is incorrect. The rest of the lines compile without issue.

15. B, D. Options A and C do not compile because they are invalid generic declarations. Primitives are not allowed as generics, and `Map` must have two generic type parameters. Option E is incorrect because partitioning only gives a `Boolean` key. Options B and D are correct because they return a `Map` with a `Boolean` key and a value type that can be customized to any `Collection`.

16. B, C. First, this mess of code does compile. While it starts with an infinite stream on line 23, it becomes finite on line 24 thanks to `limit()`, making option F incorrect. The pipeline preserves only nonempty elements on line 25. Since there aren't any of those, the pipeline is empty. Line 26 converts this to an empty map.

Lines 27 and 28 create a `Set` with no elements and then another empty stream. Lines 29 and 30 convert the generic type of the `Stream` to `List<String>` and then `String`. Finally, line 31 gives us another `Map<Boolean, List<String>>`.

The `partitioningBy()` operation always returns a map with two `Boolean` keys, even if there are no corresponding values. Therefore, option B is correct if the code is kept as is. By contrast, `groupingBy()` returns only keys that are actually needed, making option C correct if the code is modified on line 31.

17. D. The terminal operation is `count()`. Since there is a terminal operation, the intermediate operations run. The `peek()` operation comes before the `filter()`, so both numbers are printed, making option D the answer. After the `filter()`, the `count()` happens to be `1` since one of the numbers is filtered out. However, the result of the stream pipeline isn't stored in a variable or printed, and it is ignored.

18. D. This compiles, ruling out options E, F, and G. Since line 29 filters by names starting with `E`, that rules out options A and B. Finally, line 31 counts the entire list, which is of size 2, giving us option D as the answer.

19. B. Both lists and streams have `forEach()` methods. There is no reason to collect into a list just to loop through it. Option A is incorrect because it does not contain a terminal operation or print anything. Options B and C both work. However, the question asks about the simplest way, which is option B.

20. C, E, F. Options A and B compile and return an empty string without throwing an exception, using a `String` and `Supplier` parameter, respectively. Option G does not compile as the `get()` method does not take a parameter. Options C and F throw a `NoSuchElementException`. Option E throws a `RuntimeException`. Option D looks correct but will compile only if the `throw` is removed. Remember, the `orElseThrow()` should get a lambda expression or method reference that returns an exception, not one that throws an exception.

21. B. We start with an infinite stream where each element is `x`. The `spliterator()` method is a terminal operation since it returns a `Spliterator` rather than a `Stream`. The `tryAdvance()` method gets the first element and prints a single `x`. The `trySplit()` method takes a large number of elements from the stream. Since this is an infinite stream, it doesn't attempt to take half. Then `tryAdvance()` is called on the new `split` variable, and another `x` is printed. Since there are two values printed, option B is correct.

Chapter 11: Exceptions and Localization

1. A, C, D, E. A method that declares an exception isn't required to throw one, making option A correct. Unchecked exceptions can be thrown in any method, making options C and E correct. Option D matches the exception type declared, so it's also correct. Option B is incorrect because a broader exception is not allowed.

2. F. The code does not compile because the `throw` and `throws` keywords are incorrectly used on lines 6, 7, and 9. If the keywords were fixed, the rest of the code would compile and print a stack trace with `YesProblem` at runtime. For this reason, option F is correct.

3. A, D, E. Localization refers to user-facing elements. Dates, currency, and numbers are commonly used in different formats for different countries, making options A, D, and E correct. Class and variable names, along with lambda expressions, are internal to the application, so there is no need to translate them for users.

4. E. The order of `catch` blocks is important because they're checked in the order they appear after the `try` block. Because `ArithmeticException` is a child class of `RuntimeException`, the `catch` block on line 7 is unreachable (if an `ArithmeticException` is thrown in the `try` block, it will be caught on line 5). Line 7 generates a compiler error because it is unreachable code, making option E correct.

5. C, F. The code compiles and runs without issue. When a `CompactNumberFormat` instance is requested without a style, it uses the `SHORT` style by default. This results in both of the first two statements printing `100K`, making option C correct. If the `LONG` style were used, then `100 thousand` would be printed. Option F is also correct, as the full value is printed with a currency formatter.

6. E. A `LocalDate` does not have a time element. Therefore, a date/time formatter is not appropriate. The code compiles but throws an exception at runtime, making option E correct. If `ISO_LOCAL_DATE` were used, the code would print `2022 APRIL 30`.

7. E. The first compiler error is on line 12 because each resource in a try-with-resources statement must have its own data type and be separated by a semicolon (`;`). Line 15 does not compile because the variable `s` is already declared in the method. Line 17 also does not compile. The `FileNotFoundException`, which inherits from `IOException` and `Exception`, is a checked exception, so it must be handled in a `try/catch` block or declared by the method. Because these three lines of code do not compile, option E is the correct answer.

8. C. Java will first look for the most specific matches it can find, starting with `Dolphins_en_US.properties`. Since that is not an answer choice, it drops the country and looks for `Dolphins_en.properties`, making option C correct. Option B is incorrect because a country without a language is not a valid locale.

9. D. When working with a custom number formatter, the `0` symbol displays the digit as `0`, even if it's not present, while the `#` symbol omits the digit from the start or end of the `String` if it is not present. Based on the requested output, a `String` that displays at least three digits before the decimal (including a comma) and at least one after the decimal is required. It should display a second digit after the decimal if one is available. For this reason, option D is the correct answer.

10. B. An `IllegalArgumentException` is used when an unexpected parameter is passed into a method, making option B correct. Option A is incorrect because returning `null` or `-1` is a common return value for searching for data. Option D is incorrect because a `for` loop is typically used for this scenario. Option E is incorrect because you should find out how to code the method and not leave it for the unsuspecting programmer who calls your method. Option C is incorrect because you should run!

11. B, E, F. An exception that must be handled or declared is a checked exception. A checked exception inherits `Exception` but not `RuntimeException`. The entire hierarchy counts, so options B and E are both correct. Option F is also correct, as a class that inherits `Throwable` but not `RuntimeException` or `Error` is also checked.

12. B, C. The code does not compile as is because the exception declared by the `close()` method must be handled or declared. Option A is incorrect because removing the exception from the declaration causes a compilation error on line 4, as `FileNotFoundException` is a checked exception that must be handled or declared. Option B is correct because the unhandled exception within the `main()` method becomes declared. Option C is also correct because the exception becomes handled. Option D is incorrect because the exception remains unhandled.

13. A, B. A try-with-resources statement does not require a `catch` or `finally` block. A traditional `try` statement requires at least one of the two. Neither statement can be written without a body encased in braces, `{}`. For these reasons, options A and B are correct.

14. C. Starting with Java 15, `NullPointerException` stack traces include the name of the variable that is `null` by default, making option A incorrect. The first `NullPointerException` encountered at runtime is when `dewey.intValue()` is called, making option C correct. Options E and F are incorrect as only one `NullPointerException` exception can be thrown at a time.

15. C, D. The code compiles with the appropriate input, so option G is incorrect. A locale consists of a required lowercase language code and optional uppercase country code. In the `Locale()` constructor, the language code is provided first. For these reasons, options C and D are correct. Option E is incorrect because a `Locale` is created using a constructor or `Locale.Builder` class. Option F is really close but is missing `build()` at the end. Without that, option F does not compile.

16. F. The code compiles, but the first line produces a runtime exception regardless of what is inserted into the blank, making option F correct. When creating a custom formatter, any nonsymbol code must be properly escaped using pairs of single quotes (`'`). In this case, it fails because `o` is not a symbol. Even if you didn't know `o` wasn't a symbol, the code contains an unmatched single quote. If the properly escaped value of `"hh' o''clock'"` were used, then the correct answers would be `ZonedDateTime`, `LocalDateTime`, and `LocalTime`. Option B would not be correct because `LocalDate` values do not have an hour part.

17. D, F. Option A is incorrect because Java will look at parent bundles if a key is not found in a specified resource bundle. Option B is incorrect because resource bundles are loaded from `static` factory methods. Option C is incorrect, as resource bundle values are read from the `ResourceBundle` object directly. Option D is correct because the locale is changed only in memory. Option E is incorrect, as the resource bundle for the default locale may be used if there is no resource bundle for the specified locale (or its locale without a country code). Finally, option F is correct. The JVM will set a default locale automatically.

18. C. After both resources are declared and created in the try-with-resources statement, `T` is printed as part of the body. Then the try-with-resources completes and closes the resources in the reverse of the order in which they were declared. After `W` is printed, an exception is thrown. However, the remaining resource still needs to be closed, so `D` is printed. Once all the resources are closed, the exception is thrown and swallowed in the `catch` block, causing `E` to be printed. Last, the `finally` block is run, printing `F`. Therefore, the answer is `TWDEF` and option C is correct.

19. D. Java will use `Dolphins_fr.properties` as the matching resource bundle on line 7 because it is an exact match on the language of the requested locale. Line 8 finds a matching key in this file. Line 9 does not find a match in that file; therefore, it has to look higher up in the hierarchy. Once a bundle is chosen, only resources in that hierarchy are allowed. It cannot use the default locale anymore, but it can use the default resource bundle specified by `Dolphins.properties`. For these reasons, option D is correct.

20. G. The `main()` method invokes `go()`, and `A` is printed on line 3. The `stop()` method is invoked, and `E` is printed on line 14. Line 16 throws a `NullPointerException`, so `stop()` immediately ends, and line 17 doesn't execute. The exception isn't caught in `go()`,

so the `go()` method ends as well, but not before its `finally` block executes and `C` is printed on line 9. Because `main()` doesn't catch the exception, the stack trace displays, and no further output occurs. For these reasons, `AEC` is printed followed by a stack trace for a `NullPointerException`, making option G correct.

21. C. The code does not compile because the multi-catch block on line 7 cannot catch both a superclass and a related subclass. Options A and B do not address this problem, so they are incorrect. Since the `try` body throws `SneezeException`, it can be caught in a `catch` block, making option C correct. Option D allows the `catch` block to compile but causes a compiler error on line 6. Both of the custom exceptions are checked and must be handled or declared in the `main()` method. A `SneezeException` is not a `SniffleException`, so the exception is not handled. Likewise, option E leads to an unhandled exception compiler error on line 6.

22. B. For this question, the date used is April 5, 2022 at 12:30:20pm. The code compiles, and either form of the formatter is correct: `dateTime.format(formatter)` or `formatter.format(dateTime)`. The custom format `m` returns the minute, so `30` is output first. The next line throws an exception as `z` relates to time zone, and date/time does not have a zone component. This exception is then swallowed by the `try/catch` block. Since this is the only value printed, option B is correct. If the code had not thrown an exception, the last line would have printed `2022`.

23. A, E. Resources must inherit `AutoCloseable` to be used in a try-with-resources block. Since `Closeable`, which is used for I/O classes, extends `AutoCloseable`, both may be used, making options A and E correct.

24. G. The code does not compile because the resource `walk1` is not `final` or effectively final and cannot be used in the declaration of a try-with-resources statement. For this reason, option G is correct. If the line that set `walk1` to `null` were removed, then the code would compile and print `blizzard 2` at runtime, with the exception inside the `try` block being the primary exception since it is thrown first. Then two suppressed exceptions would be added to it when trying to close the `AutoCloseable` resources.

25. A. The code compiles and prints the value for Germany, `2,40 €`, making option A the correct answer. Note that the default locale category is ignored since an explicit currency locale is selected.

26. B, F. The `try` block is not capable of throwing an `IOException`, making the `catch` block unreachable code and option A incorrect. Options B and F are correct, as both are unchecked exceptions that do not extend or inherit from `IllegalArgumentException`. Remember, it is not a good idea to catch `Error` in practice, although because it is possible, it may come up on the exam. Option C is incorrect because the variable `c` is declared already in the method declaration. Option D is incorrect because the `IllegalArgumentException` inherits from `RuntimeException`, making the first declaration unnecessary. Similarly, option E is incorrect because `NumberFormatException` inherits from `IllegalArgumentException`, making the second declaration unnecessary. Since options B and F are correct, option G is incorrect.

Chapter 12: Modules

1. E. Modules are required to have a `module-info.java` file at the root directory of the module. Option E matches this requirement.

2. B. Options A, C, and E are incorrect because they refer to directives that don't exist. The `exports` directive is used when allowing a package to be called by code outside of the module, making option B the correct answer. Notice that options D and F are incorrect because of `requires`.

3. G. The `-m` or `--module` option is used to specify the module and class name. The `-p` or `--module-path` option is used to specify the location of the modules. Option D would be correct if the rest of the command were correct. However, running a program requires specifying the package name with periods (`.`) instead of slashes. Since the command is incorrect, option G is correct.

4. D. A service consists of the service provider interface and logic to look up implementations using a service locator. This makes option D correct. Make sure you know that the service provider itself is the implementation, which is not considered part of the service.

5. E, F. Automatic modules are on the module path but do not have a `module-info.java` file. Named modules are on the module path and do have a `module-info`. Unnamed modules are on the classpath. Therefore, options E and F are correct.

6. A, F. Options C and D are incorrect because there is no `use` directive. Options A and F are correct because `opens` is for reflection and `uses` declares that an API consumes a service.

7. A, B, E. Any version information at the end of the JAR filename is removed, making options A and B correct. Underscores (`_`) are turned into dots (`.`), making options C and D incorrect. Other special characters like a dollar sign (`$`) are also turned into dots. However, adjacent dots are merged, and leading/trailing dots are removed. Therefore, option E is correct.

8. A, D. A cyclic dependency is when a module graph forms a circle. Option A is correct because the Java Platform Module System does not allow cyclic dependencies between modules. No such restriction exists for packages, making option B incorrect. A cyclic dependency can involve two or more modules that require each other, making option D correct, while option C is incorrect. Finally, option E is incorrect because unnamed modules cannot be referenced from an automatic module.

9. F. The `provides` directive takes the interface name first and the implementing class name second and also uses `with`. Only option F meets these two criteria, making it the correct answer.

10. B, C. Packages inside a module are not exported by default, making option B correct and option A incorrect. Exporting is necessary for other code to use the packages; it is not necessary to call the `main()` method at the command line, making option C correct and option D incorrect. The `module-info.java` file has the correct name and compiles, making options E and F incorrect.

11. D, G, H. Options A, B, E, and F are incorrect because they refer to directives that don't exist. The `requires transitive` directive is used when specifying a module to be used by the requesting module and any other modules that use the requesting module. Therefore, `dog` needs to specify the transitive relationship, and option G is correct. The module `puppy` just needs `requires dog`, and it gets the transitive dependencies, making option D correct. However, `requires transitive` does everything `requires` does and more, which makes option H the final answer.

12. A, B, C, F. Option D is incorrect because it is a package name rather than a module name. Option E is incorrect because `java.base` is the module name, not `jdk.base`. Option G is wrong because we made it up. Options A, B, C, and F are correct.

13. D. There is no `getStream()` method on a `ServiceLoader`, making options A and C incorrect. Option B does not compile because the `stream()` method returns a list of `Provider` interfaces and needs to be converted to the `Unicorn` interface we are interested in. Therefore, option D is correct.

14. C. The `-p` option is a shorter form of `--module-path`. Since the same option cannot be specified twice, options B, D, and F are incorrect. The `--module-path` option is an alternate form of `-p`. The module name and class name are separated with a slash, making option C the answer. Note that `x-x` is legal because the module path is a folder name, so dashes are allowed.

15. B. A top-down migration strategy first places all JARs on the module path. Then it migrates the top-level module to be a named module, leaving the other modules as automatic modules. Option B is correct as it matches both of those characteristics.

16. A. Since this is a new module, you need to compile it. However, none of the existing modules needs to be recompiled, making option A correct. The service locator will see the new service provider simply by having that new service provider on the module path.

17. E. Trick question! An unnamed module doesn't use a `module-info.java` file. Therefore, option E is correct. An unnamed module can access an automatic module. The unnamed module would simply treat the automatic module as a regular JAR without involving the `module.info` file.

18. D. The `jlink` command creates a directory with a smaller Java runtime containing just what is needed. The `JMOD` format is for native code. Therefore, option D is correct.

19. E. There is a trick here. A module definition uses the keyword `module` rather than `class`. Since the code does not compile, option E is correct. If the code did compile, options A and D would be correct.

20. A. When running `java` with the `-d` option, all the required modules are listed. Additionally, the `java.base` module is listed since it is included automatically. The line ends with `mandated`, making option A correct. The `java.lang` is a trick since it is a package that is imported by default in a class rather than a module.

21. H. This question is tricky. The service locator must have a `uses` directive, but that is on the service provider interface. No modules need to specify `requires` on the service provider since that is the implementation. Since none are correct, option H is the answer.

22. A, F. An automatic module exports all packages, making option A correct. An unnamed module is not available to any modules on the module path. Therefore, it doesn't export any packages, and option F is correct.

23. E. The module name is valid, as are the `exports` statements. Lines 4 and 5 are tricky because each is valid independently. However, the same module name is not allowed to be used in two `requires` statements. The second one fails to compile on line 5, making option E the answer.

24. A. Since the JAR is on the classpath, it is treated as a regular unnamed module even though it has a `module-info.java` file inside. Remember from learning about top-down migration that modules on the module path are not allowed to refer to the classpath, making options B and D incorrect. The classpath does not have a facility to restrict packages, making option A correct and options C and E incorrect.

25. A, C, D. Options A and C are correct because both the consumer and the service locator depend on the service provider interface. Additionally, option D is correct because the service locator must specify that it `uses` the service provider interface to look it up.

Chapter 13: Concurrency

1. D, F. There is no such class within the Java API called `ParallelStream`, so options A and E are incorrect. The method defined in the `Stream` class to create a parallel stream from an existing stream is `parallel()`; therefore, option F is correct, and option C is incorrect. The method defined in the `Collection` class to create a parallel stream from a collection is `parallelStream()`; therefore, option D is correct, and option B is incorrect.

2. A, D. The `tryLock()` method returns immediately with a value of `false` if the lock cannot be acquired. Unlike `lock()`, it does not wait for a lock to become available. This code fails to check the return value on line 8, resulting in the protected code being entered regardless of whether the lock is obtained. In some executions (when `tryLock()` returns `true` on every call), the code will complete successfully and print `45` at runtime, making option A correct. On other executions (when `tryLock()` returns `false` at least once), the `unlock()` method on line 10 will throw an `IllegalMonitorStateException` at runtime, making option D correct. Option B would be possible if line 10 did not throw an exception.

3. B, C, F. `Runnable` returns `void` and `Callable` returns a generic type, making options A and D incorrect and option F correct. All methods are capable of throwing unchecked exceptions, so option B is correct. Only `Callable` is capable of throwing checked exceptions, so option E is incorrect. Both `Runnable` and `Callable` are functional interfaces that can be implemented with a lambda expression, so option C is also correct.

4. B, C. The code does not compile, so options A and F are incorrect. The first problem is that although a `ScheduledExecutorService` is created, it is assigned to an `ExecutorService`. The type of the variable on line `w1` would have to be updated to `ScheduledExecutorService` for the code to compile, making option B correct. The second problem is that `scheduleWithFixedDelay()` supports only `Runnable`, not `Callable`, and any attempt to return a value is invalid in a `Runnable` lambda expression; therefore, line `w2` will also not compile, and option C is correct. The rest of the lines compile without issue, so options D and E are incorrect.

5. C. The code compiles and runs without throwing an exception or entering an infinite loop, so options D, E, and F are incorrect. The key here is that the increment operator `++` is not atomic. While the first part of the output will always be `100`, the second part is nondeterministic. It may output any value from `1` to `100`, because the threads can overwrite each other's work. Therefore, option C is the correct answer, and options A and B are incorrect.

6. C, E. The code compiles, so option G is incorrect. The `peek()` method on a parallel stream will process the elements concurrently, so the order cannot be determined ahead of time, and option C is correct. The `forEachOrdered()` method will process the elements in the order in which they are stored in the stream, making option E correct. None of the methods sort the elements, so options A and D are incorrect.

7. D. Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. A race condition is an undesirable result that occurs when two tasks that should have been completed sequentially are completed at the same time. For these reasons, option D is correct.

8. B. Be wary of `run()` vs. `start()` on the exam! The method looks like it executes a task concurrently, but it runs synchronously. In each iteration of the `forEach()` loop, the process waits for the `run()` method to complete before moving on. For this reason, the code is thread-safe. Since the program consistently prints `500` at runtime, option B is correct. Note that if `start()` had been used instead of `run()` (or the stream was parallel), then the output would be indeterminate, and option C would have been correct.

9. C. If a task is submitted to a thread executor, and the thread executor does not have any available threads, the call to the task will return immediately with the task being queued internally by the thread executor. For this reason, option C is the correct answer.

10. A. The code compiles without issue, so option D is incorrect. The `CopyOnWriteArrrayList` class is designed to preserve the original list on iteration, so the first loop will be executed exactly three times and, in the process, will increase the size of `tigers` to six elements. The `ConcurrentSkipListSet` class allows modifications, and since it enforces the uniqueness of its elements, the value `5` is added only once, leading to a total of four elements in `bears`. Finally, despite using the elements of `lions` to populate the collections, `tigers` and `bears` are not backed by the original list, so the size of `lions` is `3` throughout this program. For these reasons, the program prints `3 6 4`, and option A is correct.

11. F. The code compiles and runs without issue, so options C, D, E, and G are incorrect. There are two important things to notice. First, synchronizing on the first variable doesn't impact the results of the code. Second, sorting on a parallel stream does not mean that `findAny()` will return the first record. The `findAny()` method will return the value from the first thread that retrieves a record. Therefore, the output is not guaranteed, and option F is correct. Option A looks correct, but even on serial streams, `findAny()` is free to select any element.

12. B. The code snippet submits three tasks to an `ExecutorService`, shuts it down, and then waits for the results. The `awaitTermination()` method waits a specified amount of time for all tasks to complete and the service to finish shutting down. Since each five-second task is still executing, the `awaitTermination()` method will return with a value of `false` after two seconds but not throw an exception. For these reasons, option B is correct.

13. C. The code does not compile, so options A and E are incorrect. The problem here is that `c1` is an `Integer` and `c2` is a `String`, so the code fails to combine on line `q2`, since calling `length()` on an `Integer` is not allowed, and option C is correct. The rest of the lines compile without issue. Note that calling `parallel()` on an already parallel stream is allowed, and it may return the same object.

14. C, E. The code compiles without issue, so option D is incorrect. Since both tasks are submitted to the same thread executor pool, the order cannot be determined, so options A and B are incorrect, and option C is correct. The key here is that the order in which the resources `o1` and `o2` are synchronized could result in a deadlock. For example, if the first thread gets a lock on `o1` and the second thread gets a lock on `o2` before either thread can get their second lock, the code will hang at runtime, making option E correct. The code cannot produce a livelock, since both threads are waiting, so option F is incorrect. Finally, if a deadlock does occur, an exception will not be thrown, so option G is incorrect.

15. A. The code compiles and runs without issue, so options C, D, E, and F are incorrect. The `collect()` operation groups the animals into those that do and do not start with the letter `p`. Note that there are four animals that do not start with the letter `p` and three animals that do. The logical complement operator `(!)` before the `startsWith()` method means that results are reversed, so the output is `3 4`, and option A is correct, making option B incorrect.

16. A, B. The code compiles just fine. If the calls to `fuel++` are ordered sequentially, then the program will print `100` at runtime, making option B correct. On the other hand, the calls may overwrite each other. The `volatile` attribute only guarantees memory consistency, not thread-safety, making option A correct and option C incorrect. Option E is also incorrect, as no `InterruptedException` is thrown by this code. Remember, `interrupt()` only impacts a thread that is in a `WAITING` or `TIMED_WAITING` state. Calling `interrupt()` on a thread in a `NEW` or `RUNNABLE` state has no impact unless the code is running and explicitly checking the `isInterrupted()` method.

17. F. The `lock()` method will wait indefinitely for a lock, so option A is incorrect. Options B and C are also incorrect, as the correct method name to attempt to acquire a lock is `tryLock()`. Option D is incorrect, as fairness is set to `false` by default and must be enabled by using an overloaded constructor. Finally, option E is incorrect because a thread that holds the lock may have called `lock()` or `tryLock()` multiple times. A thread needs to call `unlock()` once for each call to `lock()` and successful `tryLock()`. Option F is the correct answer since none of the other options are valid statements.

18. C, E, G. A `Callable` lambda expression takes no values and returns a generic type; therefore, options C, E, and G are correct. Options A and F are incorrect because they both take an input parameter. Option B is incorrect because it does not return a value. Option D is not a valid lambda expression, because it is missing a semicolon at the end of the `return` statement, which is required when inside braces `{}`.

19. E, G. The application compiles and does not throw an exception. Even though the stream is processed in sequential order, the tasks are submitted to a thread executor, which may complete the tasks in any order. Therefore, the output cannot be determined ahead of time, and option E is correct. Finally, the thread executor is never shut down; therefore, the code will run but never terminate, making option G also correct.

20. F. The key to solving this question is to remember that the `execute()` method returns `void`, not a `Future` object. Therefore, line `n1` does not compile, and option F is the correct answer. If the `submit()` method had been used instead of `execute()`, option C would have been the correct answer, as the output of the `submit(Runnable)` task is a `Future<?>` object that can only return `null` on its `get()` method.

21. A, D. The `findFirst()` method guarantees the first element in the stream will be returned, whether it is serial or parallel, making options A and D correct. While option B may consistently print `1` at runtime, the behavior of `findAny()` on a serial stream is not guaranteed, so option B is incorrect. Option C is likewise incorrect, with the output being random at runtime.

22. B. The code compiles and runs without issue. The key aspect to notice in the code is that a single-thread executor is used, meaning that no task will be executed concurrently. Therefore, the results are valid and predictable, with `100 100` being the output, and option B is the correct answer. If a thread executor with more threads was used, then the `s2++` operations could overwrite each other, making the second value indeterminate at the end of the program. In this case, option C would be the correct answer.

23. F. The code compiles without issue, so options B, C, and D are incorrect. The limit on the cyclic barrier is 10, but the stream can generate only up to 9 threads that reach the barrier; therefore, the limit can never be reached, and option F is the correct answer, making options A and E incorrect. Even if the `limit(9)` statement was changed to `limit(10)`, the program could still hang since the JVM might not allocate 10 threads to the parallel stream.

24. A, F. The class compiles without issue, so option A is correct. Since `getInstance()` is a `static` method and `sellTickets()` is an instance method, lines `k1` and `k4` synchronize on different objects, making option D incorrect. The class is not thread-safe because the `addTickets()` method is not synchronized, and option E is incorrect. One thread could call `sellTickets()` while another thread calls `addTickets()`, possibly resulting in bad data. Finally, option F is correct because the `getInstance()` method is `synchronized`. Since the constructor is `private`, this method is the only way to create an instance of `TicketManager` outside the class. The first thread to enter the method will set the `instance` variable, and all other threads will use the existing value. This is a singleton pattern.

25. C, D. The code compiles and runs without issue, so options F and G are incorrect. The return type of `performCount()` is `void`, so `submit()` is interpreted as being applied to a `Runnable` expression. While `submit(Runnable)` does return a `Future<?>`, calling `get()` on it always returns `null`. For this reason, options A and B are incorrect, and option C is correct. The `performCount()` method can also throw a runtime exception, which will then be thrown by the `get()` call as an `ExecutionException`; therefore, option D is also a correct answer. Finally, it is also possible for our `performCount()` to hang indefinitely, such as with a deadlock or infinite loop. Luckily, the call to `get()` includes a timeout value. While each call to `Future.get()` can wait up to a day for a result, it will eventually finish, so option E is incorrect.

Chapter 14: I/O

1. C. Since the question asks about putting data into a structured object, the best class would be one that deserializes the data. Therefore, `ObjectInputStream` is the best choice, which is option C. `ObjectWriter`, `BufferedStream`, and `ObjectReader` are not I/O stream classes. `ObjectOutputStream` is an I/O class but is used to serialize data, not deserialize it. `FileReader` can be used to read text file data and construct an object, but the question asks what would be the best class to use for binary data.

2. A, F. Paths that begin with the root directory are absolute paths, so option A is correct, and option C is incorrect. Option B is incorrect because the path could be a file or directory within the file system. There is no rule that files have to end with a file extension. Option D is incorrect, as it is possible to create a `File` reference to files and directories that do not exist. Option E is also incorrect. The `delete()` method returns `false` if the file or directory cannot be deleted. Character stream classes often include built-in convenience methods for working with `String` data, so option F is correct. There is no such optimization for multi-threading, making option G incorrect.

3. B, D. If the console is unavailable, `System.console()` will return `null`, making option D correct and options E and F incorrect. The `writer` methods throw a checked `IOException`, making option C incorrect. The code works correctly, prompting for input and printing it. Therefore, option A is incorrect and option B is correct.

4. F. The code does not compile, as `Files.deleteIfExists()` declares the checked `IOException` that must be handled or declared. Remember, most `Files` methods declare `IOException`, especially the ones that modify a file or directory. For this reason, option F is correct. If the method were corrected to declare the appropriate exceptions, option C would be correct. Option B would also be correct if the method were provided a symbolic link that pointed to an empty directory. Options A and E would not print anything, as `Files.isDirectory()` returns `false` for both. Finally, option D would throw a `DirectoryNotEmptyException` at runtime.

5. C. The `filter()` operation applied to a `Stream<Path>` takes only one parameter, not two, so the code does not compile, and option C is correct. If the code were rewritten to use the `Files.find()` method with the `BiPredicate` as input (along with a `maxDepth` value), the output would be option B, `Has Sub`, since the directory is given to be empty. For fun, we reversed the expected output of the ternary operation.

6. C. The code compiles and runs without issue, so options F and G are incorrect. The key here is that while `Eagle` is serializable, its parent class, `Bird`, is not. Therefore, none of the members of `Bird` will be serialized. Even if you didn't know that, you should know what happens on deserialization. During deserialization, Java calls the constructor of the first non-serializable parent. In this case, the `Bird` constructor is called, with `name` being set to `Matt`, making option C correct. Note that none of the constructors or instance initializers in `Eagle` are executed as part of deserialization.

7. B, C. The code snippet will attempt to create a directory if the target of the symbolic link exists and is a directory. If the directory already exists, though, it will throw an exception. For this reason, option A is incorrect, and option B is correct. It will be created in `/mammal/kangaroo/joey` and also reachable at `/kang/joey` because of the symbolic link, making option C correct.

8. B. The `readAllLines()` method returns a `List`, not a `Stream`. Therefore, the call to `flatMap()` is invalid, and option B is correct. If the `Files.lines()` method were used instead, it would print the contents of the file one capitalized word at a time with the commas removed.

9. C, E, G. First, the method does compile, so options A and B are incorrect. Methods to read/write `byte[]` values exist in the abstract parent of all I/O stream classes. This implementation is not correct, though, as the return value of `read(buffer)` is not used properly. It will only correctly copy files whose character count is a multiple of `10`, making option C correct and option D incorrect. Option E is also correct as the data may not have made it to disk yet. Option F would be correct if the `flush()` method were called after every write. Finally, option G is correct as the `reader` stream is never closed.

10. B, D, G. Options A and E are incorrect because `Path` and `FileSystem`, respectively, are abstract types that should be instantiated using a factory method. Option C is incorrect because the `static` method in the `Path` interface is `of()`, not `get()`. Option F is incorrect because the `static` method in the `Paths` class is `get()`, not `getPath()`. Options B and D are correct ways to obtain a `Path` instance. Option G is also correct, as there is an overloaded `static` method in `Path` that takes a `URI` instead of a `String`.

11. A, E. The code will compile if the correct classes are used, so option G is incorrect. Remember, a try-with-resources statement can use resources declared before the start of the statement. The reference type of `wrapper` is `InputStream`, so we need a class that inherits `InputStream`. We can eliminate `BufferedWriter`, `ObjectOutputStream`, and `BufferedReader` since their names do not end in `InputStream`. Next, we see the class must take another stream as input, so we need to choose the remaining streams that are high-level streams. `BufferedInputStream` is a high-level stream, so option A is correct. Even though the instance is already a `BufferedInputStream`, there's no rule that it can't be

wrapped multiple times by a high-level stream. Option D is incorrect, as `FileInputStream` operates on a file, not another stream. Finally, option E is correct—an `ObjectInputStream` is a high-level stream that operates on other streams.

12. C, E. The method to create a directory in the `Files` class is `createDirectory()`, not `mkdir()`. For this reason, line 6 does not compile, and option C is correct. In addition, the `setTimes()` method is available only on `BasicFileAttributeView`, not the read-only `BasicFileAttributes`, so line 8 will also not compile, making option E correct.

13. A, G. For a class to be serialized, it must implement the `Serializable` interface and contain instance members that are serializable or marked `transient`. For these reasons, options A and G are correct and option F is incorrect. Option B is incorrect because even records are required to implement `Serializable` to be serialized. Option C is incorrect because it describes deserialization. The `Serializable` interface is a marker interface that does not contain any abstract methods, making option D incorrect. While it is a good practice for a serializable class to include a `static serialVersionUID` variable, it is not required. Therefore, option E is incorrect as well.

14. B, D, E. `Path` is immutable, so line 23 is ignored. If it were assigned to `p1`, option A would be correct. Since it is not assigned, the original value is still present, which is option B. Moving on to the second section, the `subpath()` method on line 27 is applied to the absolute path, which returns the relative path `animals/bear`. Next, the `getName()` method is applied to the relative path, and since this is indexed from `0`, it returns the relative path `bear`. Therefore, option D is correct. Finally, remember calling `resolve()` with an absolute path as a parameter returns the absolute path, so option E is correct.

15. B, E, F. Option A does not compile, as there is no `File` constructor that takes three parameters. Option B is correct and is the proper way to create a `File` instance with a single `String` parameter. Option C is incorrect, as there is no constructor that takes a `String` followed by a `File`. There is a constructor that takes a `File` followed by a `String`, making option E correct. Option D is incorrect because the first parameter is missing a slash (`/`) to indicate it is an absolute path. Since it's a relative path, it is correct only when the user's current directory is the root directory. Finally, option F is correct as it creates a `File` from a `Path`.

16. A, D. The method compiles, so option E is incorrect. The method creates a `new-zoo.txt` file and copies the first line from `zoo-data.txt` into it, making option A correct. The try-with-resources statement closes all of the declared resources, including the `FileWriter o`. For this reason, the `Writer` is closed when the last `o.write()` is called, resulting in an `IOException` at runtime and making option D correct. Option F is incorrect because this implementation uses the character stream classes, which inherit from `Reader` or `Writer`.

17. B, C, E. Options B and C are properties of NIO.2 and are good reasons to use it over the `java.io.File` class. Option A is incorrect as both APIs can delete only empty directories, not a directory tree. Using a view to read multiple attributes leads to fewer round trips between the process and the file system and better performance, making option E correct. Views can be used to access file system–specific attributes that are not available in `Files` methods; therefore, option D is correct. `Files` is part of `NIO.2`, whereas `File` is part of `java.io`, which means option F is incorrect.

18. C. Since a `Reader` may or may not support `mark()`, we can rule out options E, F, G, and H. Assuming `mark()` is supported, `P` is added to the `StringBuilder` first. Next, the position in the stream is marked before `E`. The `E` is added to the `StringBuilder`, with `AC` being skipped, and then the `O` is added to the `StringBuilder`, with `CK` being skipped. The stream is then `reset()` to the position before the `E`. The call to `skip(0)` doesn't do anything since there are no characters to skip, so `E` is added onto the `StringBuilder` in the next `read()` call. The value `PEOE` is printed, and option C is correct.

19. C. The code compiles and runs without issue, so option G is incorrect. If you simplify the redundant path symbols, `p1` and `p2` represent the same path, `/lizard/walking.txt`. Therefore, `isSameFile()` returns `true`. The second output is `false`, because `equals()` checks only if the path values are the same, without reducing the path symbols. Finally, `mismatch()` sees that the contents are the same and returns `-1`. For these reasons, option C is correct.

20. D. The target path of the file after the `move()` operation is `/animals`, not `/animals/monkey.txt`, so options A and B are both incorrect. Both will throw an exception at runtime since `/animals` already exists and is a directory. Next, the `NOFOLLOW_LINKS` option means that if the source is a symbolic link, the link itself and not the target will be copied at runtime, so option C is also incorrect. The option `ATOMIC_MOVE` means that any process monitoring the file system will not see an incomplete file during the move, so option D is correct.

21. C. The code compiles and runs without issue, so options D, E, and F are incorrect. The most important thing to notice is that the depth parameter specified as the second argument to `find()` is `0`, meaning the only record that will be searched is the top-level directory. Since we know that the top directory is a directory and not a symbolic link, no other paths will be visited, and nothing will be printed. For these reasons, option C is the correct answer.

22. G. The code compiles, so option F is incorrect. To be serializable, a class must implement the `Serializable` interface, which `Zebra` does. It must also contain instance members that either are marked `transient` or are serializable. The instance member `stripes` is of type `Object`, which is not serializable. If `Object` implemented `Serializable`, all objects would be serializable by default, defeating the purpose of having the `Serializable` interface. Therefore, the `Zebra` class is not serializable, with the program throwing an exception at runtime if serialized and making option G correct. If `stripes` were removed from the class, options A and D would be the correct answers, as `name` and `age` are both marked `transient`.

23. A, D. The code compiles without issue, so options E and F are incorrect. The `toRealPath()` method will simplify the path to `/animals` and throw an exception if it does not exist, making option D correct. If the path does exist, calling `getParent()` on it returns the root directory. Walking the root directory with the filter expression will print all `.java` files in the root directory (along with all `.java` files in the directory tree), making option A correct. Option B is incorrect because it will skip files and directories that do not end in the `.java` extension. Option C is also incorrect as `Files.walk()` does not follow symbolic links by default. Only if the `FOLLOW_LINKS` option is provided and a cycle is encountered will the exception be thrown.

24. B. The method compiles without issue, so option E is incorrect. Option F is also incorrect. Even though `/flip` exists, `createDirectories()` does not throw an exception if the path already exists. If `createDirectory()` were used instead, option F would be correct. Next, the `copy()` command takes a target that is the path to the new file location, not the directory to be copied into. Therefore, the target path should be `/flip/sounds.txt`, not `/flip`. For this reason, options A and C are incorrect. Since the question says the file already exists, the `REPLACE_EXISTING` option must be specified or an exception will be thrown at runtime, making option B the correct answer.

25. B, D. Since you need to read characters, the `Reader` classes are appropriate. Therefore, you can eliminate options A, C, and F. Additionally, options E and G are incorrect, as they reference classes that do not exist. Options B and D are correct since they read from a file and buffer for performance.

Chapter 15: JDBC

1. B, F. The `Driver` and `PreparedStatement` interfaces are part of the JDK, making options A and E incorrect. Option C is incorrect because we made it up. The concrete `DriverManager` class is also part of the JDK, making option D incorrect. Options B and F are correct since the implementation of these interfaces is part of the database-specific driver JAR file.

2. A. A JDBC URL has three main parts separated by single colons, making options B, C, E, and F incorrect. The first part is always `jdbc`, making option D incorrect. Therefore, the correct answer is option A. Notice that you can get this right even if you've never heard of the Sybase database before.

3. B, D. When setting parameters on a `PreparedStatement`, there are only options that take an index, making options C and F incorrect. The indexing starts with `1`, making option A incorrect. This query has only one parameter, so option E is also incorrect. Option B is correct because it simply sets the parameter. Option D is also correct because it sets the parameter and then immediately overwrites it with the same value.

4. C. A `Connection` is created using a `static` method on `DriverManager`. It does not use a constructor. Therefore, option C is correct. If the `Connection` was created properly, the answer would be option B.

5. B. The first line has a return type of `boolean`, making it an `execute()` call. The second line returns the number of modified rows, making it an `executeUpdate()` call. The third line returns the results of a query, making it an `executeQuery()` call. Therefore, option B is the answer.

6. B. The first line enables autocommit mode. This is the default and means to commit immediately after each update. When the `rollback()` runs, there are no uncommitted statements, so there is nothing to roll back. This gives us the initial two rows in addition to the inserted one making option B correct. If `setAutoCommit(false)` were called, option A would be the answer. The `ResultSet` types are just there to mislead you. Any types are valid for `executeUpdate()` since no `ResultSet` is involved.

7. C. This code works as expected. It updates each of the five rows in the table and returns the number of rows updated. Therefore, option C is correct.

8. A, B. Option A is one of the answers because you are supposed to use braces (`{}`) for all SQL in a `CallableStatement`. Option B is the other answer because each parameter should be passed with a question mark (`?`). The rest of the code is correct. Note that your database might not behave the way that's described here, but you still need to know this syntax for the exam.

9. E. This code declares a bind variable with `?` but never assigns a value to it. The compiler does not enforce bind variables have values, so the code compiles, but produces a `SQLException` at runtime, making option E correct.

10. D. JDBC code throws a `SQLException`, which is a checked exception. The code does not handle or declare this exception, and therefore it doesn't compile. Since the code doesn't compile, option D is correct. If the exception were handled or declared, the answer would be option C.

11. D. JDBC resources should be closed in the reverse order from that in which they were opened. The order for opening is `Connection`, `CallableStatement`, and `ResultSet`. The order for closing is `ResultSet`, `CallableStatement`, and `Connection`, which is option D.

12. C. This code calls the `PreparedStatement` twice. The first time, it gets the numbers greater than 3. Since there are two such numbers, it prints two lines. The second time, it gets the numbers greater than 100. There are no such numbers, so the `ResultSet` is empty. Two lines are printed in total, making option C correct. The `ResultSet` options are just there to trick you since only the default settings are used by the rest of the code.

13. B, F. In a `ResultSet`, columns are indexed starting with 1, not 0. Therefore, options A, C, and E are incorrect. There are methods to get the column as a `String` or `Object`. However, option D is incorrect because an `Object` cannot be assigned to a `String` without a cast.

14. C. Since an `OUT` parameter is used, the code should call `registerOutParameter()`. Since this is missing, option C is correct.

15. C, D. Rolling back to a point invalidates any savepoints created after it. Options A and E are incorrect because they roll back to lines 19 and 17, respectively. Option B is incorrect because you cannot roll back to the same savepoint twice. Options C and D are the answers because those savepoints were created after `curly`.

16. E. First, notice that this code uses a `PreparedStatement`. Options A, B, and C are incorrect because they are for a `CallableStatement`. Next, remember that the number of parameters must be an exact match, making option E correct. Remember that you will not be tested on SQL syntax. When you see a question that appears to be about SQL, think about what it might be trying to test you on.

17. D. This code calls the `PreparedStatement` twice. The first time, it gets the numbers greater than 3. Since there are two such numbers, it prints two lines. Since the parameter is not set between the first and second calls, the second attempt also prints two rows. Four lines are printed in total, making option D correct.

18. D. Before accessing data from a `ResultSet`, the cursor needs to be positioned. The call to `rs.next()` is missing from this code causing a `SQLException` and option D to be correct.

19. E. This code should call `prepareStatement()` instead of `prepareCall()` since it is not executing a stored procedure. Since we are using `var`, it does compile. Java will happily create a `CallableStatement` for you. Since this compile safety is lost, the code will not cause issues until runtime. At that point, Java will complain that you are trying to execute SQL as if it were a stored procedure, making option E correct.

20. B. The `prepareStatement()` method requires SQL to be passed in. Since this parameter is omitted, line 27 does not compile, and option B is correct.

21. B, D. The code starts with autocommit off. As written, we turn autocommit mode back on and immediately commit the transaction. This is option B. When line W is commented out, the update gets lost, making option D the other answer.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Online Test Bank

To help you study for your OCP Java SE 17 Developer certification exam, register to gain one year of FREE access after activation to the online interactive test bank—included with your purchase of this book! All of the chapter review and practice questions in this book are included in the online test bank so you can study in a timed and graded setting.

Register and Access the Online Test Bank

To register your book and get access to the online test bank, follow these steps:

1. Go to `www.wiley.com/go/sybextestprep`.
2. Select your book from the list.
3. Complete the required registration information, including answering the security verification to prove book ownership. You will be emailed a pin code.
4. Follow the directions in the email or go to `www.wiley.com/go/sybextestprep`.
5. Find your book on that page and click the "Register or Login" link with it. Then enter the pin code you received and click the "Activate PIN" button.
6. On the Create an Account or Login page, enter your username and password, and click Login or, if you don't have an account already, create a new account.
7. At this point, you should be in the test bank site with your new test bank listed at the top of the page. If you do not see it there, please refresh the page or log out and log back in.

SYBEX®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.