Appendix Answers to Review Questions

Chapter 1: Welcome to Java

1. B, E. C++ has operator overloading and pointers. Java made a point of not having either. Java does have references to objects, but these are pointing to an object that can move around in memory. Option B is correct because Java is platform independent. Option E is correct because Java is object-oriented. While it does support some parts of functional programming, these occur within a class.
2. C, D. Java puts source code in `.java` files and bytecode in `.class` files. It does not use a `.bytecode` file. When running a Java program, you pass just the name of the class without the `.class` extension.
3. C, D. This example is using the single-file source-code launcher. It compiles in memory rather than creating a `.class` file, making option A incorrect. To use this launcher, programs can only reference classes built into the JDK. Therefore, option B is incorrect, and options C and D are correct.
4. C, D. The `Tank` class is there to throw you off since it isn't used by `AquariumVisitor`. Option C is correct because it imports `Jelly` by class name. Option D is correct because it imports all the classes in the `jellies` package, which includes `Jelly`. Option A is incorrect because it only imports classes in the `aquarium` package— `Tank` in this case—and not those in lower-level packages. Option B is incorrect because you cannot use wildcards anywhere other than the end of an

`import` statement. Option E is incorrect because you cannot `import` parts of a class with a regular `import` statement. Option F is incorrect because options C and D do make the code compile.

5. A, C, D, E. Eclipse is an integrated development environment (IDE). It is not included in the Java Development Kit (JDK), making option B incorrect. The JDK comes with a number of command-line tools including a compiler, packager, and documentation, making options A, D, and E correct. The JDK also includes the Java Virtual Machine (JVM), making option C correct.
6. 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.
7. 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.
8. A, B. The wildcard is configured for files ending in `.java`, making options E and F incorrect. Additionally, wildcards aren't recursive, making options C and D incorrect. Therefore, options A and B are correct.
9. B. Option B is correct because arrays start counting from zero and strings with spaces must be in quotes. Option A is incorrect because it outputs `Blue`. C is incorrect because it outputs `Jay`. Option D is incorrect because it outputs `Sparrow`. Options E and F are incorrect because they output `java.lang.ClassNotFoundException: BirdDisplay.class`.
10. E. Option E is the canonical `main()` method signature. You need to memorize it. 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. Option D is incorrect because the `main()` method must be named `main`.

11. C, D. While we wish it were possible to guarantee bug-free code, this is not something a language can ensure, making option A incorrect. Deprecation is an indication that other code should be preferred. It doesn't preclude or require eventual removal, making option B incorrect. Option E is incorrect because backward compatibility is a design goal, not sideways compatibility. Options C and D are correct.
12. C, E. When compiling with `javac`, you can specify a classpath with `-cp` or a directory with `-d`, making options C and E correct. Since the options are case sensitive, option D is incorrect. The other options are not valid on the `javac` command.
13. C. When running a program using `java`, you specify the classpath with `-cp`, making option C correct. Options D and F are incorrect because `-d` and `-p` are used for modules. Options A and B are not valid options on the `java` command.
14. A, B, C, E. When creating a `jar` file, you use the options `-cf` or `-cvf`, making options A and E correct. The `jar` command allows the use of the classpath, making option C correct. It also allows the specification of a directory using `-C`, making option B correct. Options D and F are incorrect because `-d` and `-p` are used for modules.
15. E. The `main()` method isn't `static`. It is a method that happens to be named `main()`, but it's not an application entry point. When the program is run, it gives the error. If the method were `static`, the answer would be option D. Arrays are zero-based, so the loop ignores the first element and throws an exception when accessing the element after the last one.
16. D. The package name represents any folders underneath the current path, which is `named.A` in this case. Option C is incorrect because package names are case sensitive, just like variable names and other identifiers.
17. 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.

18. 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`.
19. B, C. Eclipse is an integrated development environment (IDE). It is available from the Eclipse Foundation, not from Oracle, making option C one of the answers. The other answer is option B because the Java Development Kit (JDK) is what you download to get started. The Java Runtime Environment (JRE) was an option for older versions of Java, but it's no longer a download option for Java 11.
20. A, B, E. Unfortunately, this is something you have to memorize. The code with the hyphenated word `class-path` uses two dashes in front, making option E correct and option D incorrect. The reverse is true for the unhyphenated `classpath`, making option B correct and option C incorrect. Finally, the short form is option A.

Chapter 2: Java Building Blocks

1. B, E, G. Option A is invalid because a single underscore is no longer allowed as an identifier as of Java 9. Option B is valid because you can use an underscore within identifiers, and a dollar sign (`$`) is also a valid character. Option C is not a valid identifier because `true` is a Java reserved word. Option D is not valid because a period (`.`) is not allowed in identifiers. Option E is valid because Java is case sensitive. Since `Public` is not a reserved word, it is allowed as an identifier, whereas `public` would not be allowed. Option F is not valid because the first character is not a letter, dollar sign (`$`), or underscore (`_`). Finally, option G is valid as identifiers can contain underscores (`_`) and numbers, provided the number does not start the identifier.
2. 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.

3. B, D, E, H. A `var` cannot be initialized with a `null` value without a type, but it can be assigned a `null` value 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 `Object`, 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.
4. A, B, D, E. Line 4 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 5 does not compile because `int` is an integral type, but `2.0` is a `double` literal value, making option B correct. Line 6 compiles without issue. Lines 7 and 8 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 9 compiles because there is a `length()` method defined on `String`.
5. A, D. The class does not compile, so options E, F, G, and H are incorrect. You might notice things like loops and increment/decrement operators in this problem, which we will cover in the next two chapters, but understanding them is not required to answer this question. 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. Note that the variable `Depth` on line 2 is never used. Java

is case sensitive, so `Depth` and `depth` are distinct variables. For these reasons, options A and D are the correct answers.

6. C, E. Option C is correct because `float` and `double` primitives default to `0.0`, which also makes option A incorrect. Option E is correct because all nonprimitive values default to `null`, which makes option F incorrect. Option D is incorrect because `int` primitives default to `0`. Option B is incorrect because `char` defaults to the NUL character, `'\u0000'`. You don't need to know this value for the exam, but you should know the default value is not `null` since it is a primitive.
7. G. Option G is correct because local variables do not get assigned default values. The code fails to compile if a local variable is used when not being explicitly initialized. If this question were about instance variables, options B, D, and E would be correct. A `boolean` primitive defaults to `false`, and a `float` primitive defaults to `0.0f`.
8. B, E. Option B is correct because `boolean` primitives default to `false`. Option E is correct because `long` values default to `0L`.
9. C, E, F. In Java, there are no guarantees 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.
10. C. The class does compiles without issue, so options E, F, and G are incorrect. The key thing to notice is 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` get 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.

11. A, D, E. From Chapter 1, a `main()` method must have a valid identifier of type `String...` or `String[]`. For this reason, option G can be eliminated immediately. Option A is correct because `var` is not a reserved word in Java and may be used as an identifier. Option B is incorrect as a period (`.`) may not be used in an identifier. Option C is also incorrect as an identifier may include digits but not start with one. Options D and E are correct as an underscore (`_`) and dollar sign (`$`) may appear anywhere in an identifier. Finally, option F is incorrect, as a `var` may not be used as a method argument.
12. A, E, F. An underscore (`_`) can be placed in any numeric literal, so long as it is not at the beginning, 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.
13. B, D, H. The `Rabbit` object from line 3 has two references to it: `one` and `three`. The references are set to `null` on lines 6 and 8, respectively. Option B is correct because this makes the object eligible for garbage collection after line 8. Line 7 sets the reference `four` to `null`, since that is the value of `one`, which means it has no effect on garbage collection. The `Rabbit` object from line 4 has only a single reference to it: `two`. Option D is correct because this single reference becomes `null` on line 9. The `Rabbit` object declared on line 10 becomes eligible for garbage collection at the end of the method on line 12, making option H correct. Calling `System.gc()` has no effect on eligibility for garbage collection.
14. 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 `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 in-

correct. 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.

15. C, F, G. First off, `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 is valid.
16. A, C, D. The code contains three compilation errors, so options E, F, G, and H are incorrect. Line 2 does not compile, as this is incorrect syntax for declaring multiple variables, making option A correct. The data type is declared only once and shared among all variables in a multiple variable declaration. 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, unlike some other programming languages, 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.
17. A, E, F, G. The question is primarily about variable scope. A variable defined in a statement such as a loop or initializer block is accessible only inside that statement. For this reason, options A and E are correct. Option B is incorrect because variables can be defined inside initializer blocks. Option C is incorrect, as a constructor argument is accessible only in the constructor itself, not for the life of the instance of the class. Constructors and instance methods can access any instance variable, even ones defined after their declaration, making option D incorrect and options F and G correct.

18. F, G. The code does not compile, so options A, B, C, and D are all incorrect. The first compilation error occurs on line 5. Since `char` is an unsigned data type, it cannot be assigned a negative value, making option F correct. The second compilation error is on line 9, since `mouse` is used without being initialized, making option G correct. You could fix this by initializing a value on line 4, but the compiler reports the error where the variable is used, not where it is declared.
19. F. To solve this problem, you need to trace the braces `{}` and see when variables go in and out of scope. You are not required to understand the various data structures in the question, as this will be covered in the next few chapters. We start with `hairs`, which goes in and out of scope on line 2, as it is declared in an instance initializer, so it is not in scope on line 14. The three variables— `water`, `air`, `twoHumps`, declared on lines 3 and 4—are instance variables, so all three are in scope in all instance methods of the class, including `spit()` and on line 14. The `distance` method parameter is in scope for the life of the `spit()` method, making it the fourth value in scope on line 14. The `path` variable is in scope on line 6 and stays in scope until the end of the method on line 16, making it the fifth variable in scope on line 14. The `teeth` variable is in scope on line 7 and immediately goes out of scope on line 7 since the statement ends. The two variables `age` and `i` defined on lines 9 and 10, respectively, both stay in scope until the end of the `while` loop on line 15, bringing the total variables in scope to seven on line 14. Finally, `Private` is in scope on 12 but out of scope after the `for` loop ends on line 13. Since the total in-scope variables is seven, option F is the correct answer.
20. D. The class compiles and runs without issue, so options F and G are incorrect. We start with the `main()` method, which prints `7-` on line 11. 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 altogether, we have `7-0-2-4-`, making option D the correct answer.

21. A, D, F. The class compiles and runs without issue, so option H is incorrect. The program creates two `Bear` objects, one on line 9 and one on line 10. The first `Bear` object is accessible until line 13 via the `brownBear` reference variable. The second `Bear` object is passed to the first object's `roar()` method on line 11, meaning it 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. In other words, both objects become eligible for garbage collection after line 13, making options A and D correct. Finally, garbage collection is never guaranteed to run or not run, since the JVM decides this for you. For this reason, option F is correct, and options E and G are incorrect. The class contains a `finalize()` method, although this does not contribute to the answer. For the exam, you may see `finalize()` in a question, but since it's deprecated as of Java 9, you will not be tested on it.
22. H. None of these declarations is a valid instance variable declaration, as `var` cannot be used with instance variables, only local variables. For this reason, option H is the only correct answer. If the question were changed to be about local variable declarations, though, then the correct answers would be options C, D, and E. An identifier must start with a letter, `$`, or `_`, so options F and G would be incorrect. As of Java 9, a single underscore is not allowed as an identifier, so option A would be incorrect. Options A and G would also be incorrect because their numeric expressions use underscores incorrectly. An underscore cannot appear at the end of literal value, nor next to a decimal point (`.`). Finally, `null` is a reserved word, but `var` is not, so option B would be incorrect, and option E would be correct.

Chapter 3: 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 relational operator that compares the values of two numbers. 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 in 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 or decreasing order of precedence. In option D, the assignment operator (`=`) is between two unary operators, with the multiplication operator (`*`) incorrectly having the highest order or 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 short-circuit 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 right side of the ternary expression 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 re-

sult 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. If the code were corrected to add parentheses around the entire expression and cast it to a `byte` or `short`, then the program would print `3`, `6`, and `2` in that order.
11. D. First off, the `*` 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 just 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 6 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, and the equality (`==`) and inequality (`!=`) operators can both be used with objects. Option C is incorrect, as `boolean` and numeric types are not comparable with each other. For example, you can't say `true == 3` without a compilation error. Option D is incorrect, as only the short-circuit operator (`&&`) may cause only the left side of the expression to be evaluated. 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 the third and fourth lines using casting to do so.
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 an-

swers. 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. 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.

Chapter 4: Making Decisions

1. A, B, C, E, F, G. A `switch` statement supports 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. 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. Options D and H are incorrect as `long`, `float`, `double`, and their as-

sociated wrapped classes `Long`, `Float`, and `Double`, respectively, are not supported in `switch` statements.

2. B. The code compiles and runs without issue, so options D, E, and F are incorrect. Even though the 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 its associated `else` statement on line 8 is not. The `if` statement on line 6 evaluates to `false`, causing the `else` statement on line 7 to activate. The result is the code prints `Just Right`, making option B the correct answer.
3. 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. In particular, because the data type for the elements of `myFavoriteNumbers` is `Integer`, they can be easily unboxed to `int` or referenced as `Object`. For this reason, the lines containing the for-each expressions each compile.
4. C, 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. Option D is incorrect, as a `default` statement is not required in a `switch` statement. If no `case` statements match and there is no `default` statement, then the application will exit the `switch` statement without executing any branches. Finally, each `if` statement has at most one matching `else` statement, making option F incorrect. You can chain

multiple `if` and `else` statements together, but each `else` statement requires a new `if` statement.

5. 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. It is only a slight modification of a common `for` loop, with `i<weather.length` replaced with an equivalent `i<=weather.length-1`. Option C is incorrect because the snippet creates a compilation problem in the body of the `for` loop, 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. The loop update operation is optional, so that part compiles, but the increment is applied as part of the conditional check for the loop. 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`.
6. 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, else `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 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`.

7. E. This code contains numerous compilation errors, making options A and H incorrect. All of the compilation errors are contained within the `switch` statement. The `default` statement is fine and does not cause any issues. The first `case` statement does not compile, as `continue` cannot be used inside a `switch` statement. The second `case` statement also does not compile. While the `thursday` variable is marked `final`, it is not a compile-time constant required for a `switch` statement, as any `int` value can be passed in at runtime. The third `case` statement is valid and does compile, as `break` is compatible with `switch` statements. The fourth `case` statement does not compile. Even though `Sunday` is effectively `final`, it is not a compile-time constant. If it were explicitly marked `final`, then this `case` statement would compile. Finally, the last `case` statement 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`. Since exactly four lines do not compile, option E is the correct answer.
8. 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.
9. 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`.

10. 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`. Since no generic type is used, the default type is a `List` of `Object` values, and `macaw` will have a data type of `Object`, making option F correct.
11. 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 (`:`) as shown in `case 'B' : 'C' :`. For this reason, option F is the correct answer. If this line were fixed to add the keyword `case` before `'C'`, then the rest of the code would have compiled and printed `great good` at runtime.
12. 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, albeit not using any of `for` loop structures, and ends when the index is `0`. 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 at runtime, 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.
13. B, E. The code compiles without issue and prints three distinct numbers at runtime, so options G and H are incorrect. The first loop exe-

cutes 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` already 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.

14. 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 choice.
15. A, E. The most important thing to notice when reading this code is that the innermost loop is an infinite loop without a statement to branch out of it, since there is no loop termination condition. Therefore, you are looking for solutions that skip the innermost loop entirely or ones 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 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.

16. E. The code compiles without issue, making options F and G incorrect. Since Java 10, `var` is supported in both `switch` and `while` loops, provided the type can be determined by the compiler. 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.
17. 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`.
18. A, D, E. The right side of a for-each statement must be a primitive array or any class that implements `java.lang.Iterable`, which includes the `Collection` interface, although not all Collections Framework classes. For these reasons, options A, D, and E are correct. Option B is incorrect as `Map` does not implement `Collection` nor `Iterable`, since it is not a list of items, but a mapping of items to other items. Option C and F are incorrect as well. While you may consider them to be a list of characters, strictly speaking they are not considered `Iterable` in Java, since they do not implement `Iterable`. That said, you can iterate over them using a traditional `for` loop and member methods, such as `charAt()` and `length()`.
19. D. The code does compile 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.
20. 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.

Chapter 5: Core Java 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. Supposing line 5 said `String anotherFish = numFish + 1 + "";`. In that case, the answer 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. 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`. Lines 5 checks for object equality, which is true, and so 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`.
3. A, C, F. The code does compile, making option G incorrect. In the first for-each loop, `gorillas` has a type of `List<String>`, so each element `koko` has a type of `String`, making option A correct. In the second for-each loop, you might think that the diamond operator `<>` cannot be used with `var` without a compilation error, but it absolutely can. This result is `monkeys` having a type of `ArrayList<Object>` with `al-`

`bert` having a data type of `Object`, making option C correct. While `var` might indicate an ambiguous data type, there is no such thing as an undefined data type in Java, so option D is incorrect.

In the last for-each loop, `chimpanzee` has a data type of `List`. Since the left side does not define a generic type, the compiler will treat this as `List<Object>`, and `ham` will have a data type of `Object`, making option F correct. Even though the elements of `chimpanzees` might be `Integer` as defined, `ham` would require an explicit cast to call an `Integer` method, such as `ham.intValue()`.

4. 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`.
5. G. The question is trying to distract you into paying attention to logical equality versus object reference equality. The exam creators are hoping you will miss the fact that line 18 does not compile. Java does not allow you to compare `String` and `StringBuilder` using `==`.
6. 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.
7. A, B, F. Remember that indexes are zero-based, which means that 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.

8. A, D, E. `substring()` 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. The second call starts at index 7 and ends in the same place, resulting in an empty `String`. This prints out a blank line. The final call starts at index 7 and goes to the end of the `String`.
9. C, F. This question is tricky because it has two parts. The first is trying to see if you know that `String` objects are immutable. Line 17 returns `"PURR"`, but the result is ignored and not stored in `s1`. Line 18 returns `"purr"` since there is no whitespace present, but the result is again ignored. Line 19 returns `"ur"` because it starts with index 1 and ends before index 3 using zero-based indexes. The result is ignored again. 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.

   For the second part, `a += 2` expands to `a = a + 2`. A `String` concatenated with any other type gives a `String`. Lines 22, 23, and 24 all append to `a`, 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.
10. 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`.
11. A. First, we delete the characters at index 2 until the character one before index 8. At this point, `0189` is in `numbers`. The following line uses

method chaining. It appends a dash to the end of the characters sequence, resulting in `0189-`, and then inserts a plus sign at index 2, resulting in `01+89-`.

12. F. This is a trick question. The first line does not compile because you cannot assign a `String` to a `StringBuilder`. If that line were `StringBuilder b = new StringBuilder("rumble")`, the code would compile and print `rum4`. Watch out for this sort of trick on the exam. You could easily spend a minute working out the character positions for no reason at all.
13. 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 starting with 1.
14. 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 on the exam types you might not have learned. You aren't expected to know anything about them.
15. A, H. Arrays define a property called `length`. It is not a method, so parentheses are not allowed, making option A correct. The `ArrayList` class defines a method called `size()`, making option H the other correct answer.
16. A, F, G. An array is not able to change size, making option A correct and option B incorrect. Neither is immutable, making options C and D incorrect. The elements can change in value. An array does not override `equals()`, so it uses object equality, making option E incorrect. `ArrayList` does override `equals()` and defines it as the same elements in the same order, making option F correct.

The compiler does not know when an index is out of bounds and thus can't give you a compiler error, making option G correct. The code will throw an exception at runtime, though, making option H the final incorrect answer.

17. F. The code does not compile because `list` is instantiated using generics. Only `String` objects can be added to `list`, and 7 is an `int`.
18. C. The `put()` method is used on a `Map` rather than a `List` or `Set`, making options A and D incorrect. The `replace()` method does not exist on either of these interfaces. Finally, the `set` method is valid on a `List` rather than a `Set` because a `List` has an index. Therefore, option C is correct.
19. A, F. The code compiles and runs fine. However, an array must be sorted for `binarySearch()` to return a meaningful result. Option F is correct because line 14 prints a number, but the behavior is undefined. Line 8 creates a list backed by a fixed-size array of 4. Line 10 sorts it. Line 12 converts it back to an array. The brackets aren't in the traditional place, but they are still legal. Line 13 prints the first element, which is now –1, making option A the other correct answer.
20. B, C, E. Remember to watch return types on math operations. One of the tricks is option B on 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 option C on line 25. The `random()` method returns a `double`. Converting from an array to an `ArrayList` uses `Arrays.asList(names)`. There is no `asList()` method on an array instance, and option E is correct.
21. D. After sorting, `hex` contains `[30, 3A, 8, FF]`. Remember that numbers sort before letters, and strings sort alphabetically. This makes `30` come before `8`. A binary search correctly finds `8` at index 2 and `3A` at index 1. It cannot find `4F` but notices it should be at index 2. The rule when an item isn't found is to negate that index and subtract 1. Therefore, we get `-2-1`, which is `-3`.
22. A, B, D. Lines 5 and 7 use autoboxing to convert an `int` to an `Integer`. Line 6 does not because `valueOf()` returns an `Integer`. Line 8 does not because `null` is not an `int`. The code does compile.

However, when the `for` loop tries to unbox `null` into an `int`, it fails and throws a `NullPointerException`.

23. B. The first `if` statement is `false` because the variables do not point to the same object. The second `if` statement is `true` because `ArrayList` implements equality to mean the same elements in the same order.
24. D, E. The first line of code in the method creates a fixed size `List` backed by an array. This means option D is correct, making options B and F incorrect. The second line of code in the method creates an immutable list, which means no changes are allowed. Therefore, option E is correct, making options A and C incorrect.
25. A, B, D. The `compare()` method returns a positive integer when the arrays are different and `s1` 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 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 option B since the difference is at index 1. It is not the case for option D because the `s3` is shorter than the `s4` or option F because there is no difference.

Chapter 6: 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. Since there is only one parameter and it does not specify a type, the parentheses around the type 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 one is a `String`. 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, D, F. The `removeIf()` method expects a `Predicate`, which takes a parameter list of one parameter using the specified type. Options B and C are incorrect because they do not use the `return` keyword. This keyword is required to be inside the braces of a lambda body. Option E is incorrect because it is missing the parentheses around the parameter list. This is only optional for a single parameter with an inferred type.
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.
5. B, D. `Predicate<String>` takes a parameter list of one parameter using the specified type. Options A and F are incorrect because they specify the wrong number of parameters. Option C is incorrect because parentheses are required around the parameter list when the type is specified. Option E is incorrect because the name used in the parameter list does not match the name used in the body.
6. 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.
7. A, B, E, F. The `forEach()` method with one lambda parameter 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.
8. A, C, F. Option A is correct because a `Supplier` returns a value while a `Consumer` takes one and acts on it. Option C is correct because a `Comparator` returns a negative number, zero, or a positive number depending on the values passed. A `Predicate` always returns a

`boolean`. It does have a method named `test()`, making option F correct.

9. A, B, C. Since the scope of `start` and `c` is within the lambda, the variables can be declared after it without issue, making options A, B, and C correct. Option D is incorrect because setting `end` prevents it from being effectively final. Lambdas are only allowed to reference effectively final variables.
10. C. Since the `new ArrayList<>(set)` constructor makes a copy of `set`, there are two elements in each of `set` and `list`. The `forEach()` methods print each element on a separate line. Therefore, four lines are printed, and option C is the answer.
11. A. The code correctly sorts in descending order. Since uppercase normally sorts before lowercase, the order is reversed here, and option A is correct.
12. C, D, E. The first line takes no parameters, making it a `Supplier`. Option E is correct because Java can autobox from a primitive `double` to a `Double` object. Option F is incorrect because it is a `float` rather than a `double`.

    The second line takes one parameter and returns a `boolean`, making it a `Predicate`. Since the lambda parameter is unused, any generic type is acceptable, and options C and D are both correct.
13. E. Lambdas are only allowed to reference effectively final variables. You can tell the variable `j` is effectively final because adding a `final` keyword before it wouldn't introduce a compile error. Each time the `else` statement is executed, the variable is redeclared and goes out of scope. Therefore, it is not re-assigned. Similarly, `length` is effectively final. There are no compile errors, and option E is correct.
14. 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 effectively final variables. Option C is tricky because it does compile but throws an exception at runtime. Since the question only asks about compilation, option C is correct.
15. C. `Set` is not an ordered `Collection`. Since it does not have a `sort()` method, the code does not compile, making option C correct.

16. A, D. Method parameters and local variables are effectively final if they aren't changed after initialization. Options A and D meet this criterion.
17. C. Line 8 uses braces around the body. This means the `return` keyword and semicolon are required.
18. D. Lambda parameters are not allowed to use the same name as another variable in the same scope. The variable names `s` and `x` are taken from the object declarations and therefore not available to be used inside the lambda.
19. A, C. This interface specifies two `String` parameters. We can provide the parameter list with or without parameter types. However, it needs to be consistent, making option B incorrect. Options D, E, and F are incorrect because they do not use the arrow operator.
20. A, C. `Predicate<String>` takes a parameter list of one parameter using the specified type. Options E and F are incorrect because it specifies the wrong type. Options B and D are incorrect because they use the wrong syntax for the arrow operator.

Chapter 7: Methods and Encapsulation

1. 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 default access and the optional specifier `final`. Since default access does not require a modifier, we get to jump right to `final`. Option A is incorrect because default 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.
2. 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 re-

turn type is before the optional specifier and access modifier, respectively.

3. 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.
4. 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.
5. 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.
6. D. Option D is correct. This is the common implementation for encapsulation by setting all fields to be private and all methods to be public. Option A is incorrect because protected access allows everything that package-private 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-private access applies to the whole package. Option E is incorrect because Java has no such wildcard access capability.

7. B, C, D, F. The two classes are in different packages, which means `private` access and default (package-private) access will not compile. This causes compile errors in 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.
8. 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.
9. B, D, F. Option A is incorrect because the methods differ only in return type. Option C is tricky. It is incorrect because `var` is not a valid return type. Remember that `var` can be used only for local variables. Option E is incorrect because the method signature is identical once the generic types are erased. Options B and D are correct because they represent interface and superclass relationships. Option F is correct because the arrays are of different types.
10. B. `Rope` runs line 3, setting `LENGTH` to 5, and then immediately after 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`.
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. If line 10 is removed, the code works. 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. 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 one object update this common `static` variable.

13. 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.
14. 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`.
15. 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.
16. 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.
17. 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`.
18. 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.

19. A, E. The `100` parameter is an `int` and so calls the matching `int` method. 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.
20. A, C, F. Option B is incorrect because `var` cannot be a method parameter. It must be a local variable or lambda parameter. Option D is incorrect because the method declarations are identical. Option E is tricky. The variable `long` is illegal because *long* is a reserved word. Options A, C, and F are correct because they represent different types.
21. 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 are all correct, although some are more useful than others. Since the class can be written to be encapsulated, options E and F are incorrect.

Chapter 8: 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. B, C. Overloaded methods have the 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.

3. 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.
4. 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.
5. 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-private 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 15 and 16, printing `Spider` twice. Note that the `short` value is automatically cast to the larger type of `int`, which then uses the overridden method. Line 17 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.

6. 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.
7. 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.

8. 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.
9. B, F. A valid override of a method with generic arguments must have a return type that is covariant, with matching generic type parameters. Option B is correct, as it is just restating the original return type. Option F is also correct, as `ArrayList` is a subtype of `List`. The rest of the method declarations do not compile. Options A and D are invalid because the access levels, package-private and `private`, are more restrictive than the inherited access modifier, `protected`. Option C is incorrect because while `CharSquence` is a subtype of `String`, the generic type parameters must match exactly. Finally, option E is incorrect as `Object` is a supertype of `List` and therefore not covariant.
10. D. The code doesn't compile, so option A is incorrect. The first compilation error is on line 2, as `var` cannot be used as a constructor argu-

ment type. The second 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 four compilation errors, although they are limited to three lines, making option D the correct answer.

11. B, C, E. An object may be cast to a supertype without an explicit cast but requires an explicit cast to be cast to a subtype, making option A incorrect. Option B is correct, as an interface method argument may take any reference type that implements the interface. Option C is also correct, as a method that accepts `java.lang.Object` can accept any variable since all objects inherit `java.lang.Object`. This also includes primitives, which can be autoboxed to their wrapper classes. Some cast exceptions can be detected as errors at compile-time, but others can only be detected at runtime, so option D is incorrect. Due to the nature of polymorphism, a `final` instance method cannot be overridden in a subclass, so calls in the parent class will not be replaced, making option E correct. Finally, polymorphism applies to classes and interfaces alike, making option F incorrect.
12. A, B, E, F. The code compiles if the correct type is inserted in the blank, so option G is incorrect. The `setSnake()` method requires an instance of `Snake` or any subtype of `Snake`. The `Cobra` class is a subclass of `Snake`, so it is a subtype. The `GardenSnake` class is a subclass of `Cobra`, which, in turn, is a subclass of `Snake`, also making `GardenSnake` a subtype of `Snake`. For these reasons, options A, B, and E are correct. Option C is incorrect because `Object` is a supertype of `Snake`, not a subtype, as all instances inherit `Object`. 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 correct.
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 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. 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.
16. B. A valid override of a method with generic arguments must have the same signature with the same generic types. For this reason, only option B is correct. Because of type erasure, the generic type parameter will be removed when the code is compiled. Therefore, the compiler requires that the types match. Options A and D do not compile for this reason. Options C, E, and F do compile, but since the generic class

changed, they are overloads, not overrides. Remember, covariant types only apply to return values of overridden methods, not method parameters.

17. F. Options A–E are incorrect statements about inheritance and variables, making option F the correct answer. Option A is incorrect because variables can only be hidden, not overridden via inheritance. This means that they are still accessible in the parent class and do not replace the variable everywhere, as overriding does. Options B, C, and E are also incorrect as they more closely match rules for overriding methods. Also, option E is invalid as variables do not throw exceptions. Finally, option D is incorrect as this is a rule for hiding `static` methods.
18. 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.
19. C, 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, making option C correct and option D 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 in-

valid override and does not compile, as `int` is not covariant with `void`, making options G and H both incorrect.

20. 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.
21. 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.
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 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. Since `Canine` does not extend any classes, the compiler will also insert a call to the no-argument `super()` constructor defined in `java.lang.Object`, although this has little impact on the output. 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. 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 incorrect. With hidden `static` methods, Java relies on the location and reference type to determine which method is called, making option B incorrect and F correct. Finally, making a method `final`, not `static`, prevents it from being overridden, making option D correct and option C incorrect.
25. 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.

26. 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. Note that if `Orangutan` was made a subclass of `Primate` and the `this()` references were changed to `this`, then the code would compile and print `3` at runtime.

Chapter 9: Advanced Class Design

1. B, E. A method that does not declare a body is by definition `abstract`, making option E correct. All abstract interface methods are assumed to be `public`, making option B correct. Interface methods cannot be marked `protected`, so option A is incorrect. Interface methods can be marked `static` or `default`, although if they are, they must provide a body, making options C and F incorrect. Finally, `void` is a return type, not a modifier, so option D is incorrect.
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. All classes inherit `Object`, so option E is also correct. Finally, `Long` is an

unrelated class that is not a superclass of `TurtleFrog` and is therefore incorrect.

3. 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; a superclass may have already implemented an inherited interface method. The concrete class only needs to implement the inherited abstract methods. Finally, a method in a concrete class that implements an inherited abstract method overrides the method. While the method signature must match, the method declaration does not need to match, such as using a covariant return type or changing the throws declaration. For these reasons, option E is incorrect.
4. 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. 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. Since the code fails to compile, options A and B are incorrect.
5. C, F. All interface variables are implicitly assumed to be `public`, `static`, and `final`, making options C and F correct. Option A and G, `private` and default (package-private), are incorrect since they conflict with the implicit `public` access modifier. Options B and D are incorrect, as `nonstatic` and `const` are not modifiers. Finally, option E is incorrect because a variable cannot be marked `abstract`.
6. D, E. Lines 1 and 2 are declared correctly, with the implicit modifier `abstract` being applied to the interface and the implicit modifiers `public`, `static`, and `final` being applied to the interface variable, making options B and C incorrect. Option D is correct, as an `abstract` method cannot include a body. Option E is also correct because the

wrong keyword is used. A class implements an interface; it does extend it. Option F is incorrect as the implementation of `eatGrass()` in `IsAPlant` does not have the same signature; therefore, it is an overload, not an override.

7. 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.
8. C. The code compiles without issue, so option A is incorrect. Option B is incorrect, as an abstract class could implement `HasVocalCords` without the need to override the `makeSound()` method. Option C is correct; a class that implements `CanBark` automatically inherits its abstract methods, in this case `makeSound()` and `bark()`. Option D is incorrect, as a concrete class that implements `Dog` may be optionally marked `final`. Finally, an interface can extend multiple interfaces, so option E is incorrect.
9. B, C, E, F. Member inner classes, including both classes and interfaces, can be marked with any of the four access modifiers: `public`, `protected`, default (package-private), or `private`. For this reason, options B, C, E, and F are correct. Options A and D are incorrect as `static` and `final` are not access modifiers.
10. C, G. The implicitly abstract interface method on line 6 does not compile because it is missing a semicolon (`;`), making option C correct. Line 7 compiles, as it provides an overloaded version of the `fly()` method. Lines 5, 9, and 10 do not contain any compilation errors, making options A, E, and F incorrect. Line 13 does not compile because the two inherited `fly()` methods, declared on line 6 and 10, conflict with each other. The compiler recognizes that it is impossible to create a class that overrides `fly()` to return both `String` and `int`, since they are not covariant return types, and therefore blocks the `Falcon` class from compiling. For this reason, option G is correct.
11. 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`.

12. A, E. Line 11 does not compile because a `Tangerine` and `Gala` are unrelated types, which the compiler can enforce for classes, making option A correct. Line 12 is valid since `Citrus` extends `Tangerine` and would print `true` at runtime if the rest of the class compiled. Likewise, `Gala` implements `Apple`, so line 13 would also print `true` at runtime if the rest of the code compiled.
    Line 14 does compile, even though `Apple` and `Tangerine` are unrelated types. While the compiler can enforce unrelated type rules for classes, it has limited ability to do so for interfaces, since there may be a subclass of `Tangerine` that implements the `Apple` interface. Therefore, this line would print `false` if the rest of the code compiled.
    Line 15 does not compile. Since `Citrus` is marked `final`, the compiler knows that there cannot be a subclass of `Citrus` that implements `Apple`, so it can enforce the unrelated type rule. For this reason, option E is correct.
13. G. The interface and 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. Since this is the only compilation error, option G is the correct answer. If the reference type of `whale` was changed to `Orca`, then the `main()` would compile and print `Orca diving deeper 3` at runtime, making option B the correct answer. Note that line 16 compiles because the interface variable `MAX` is inherited as part of the class structure.
14. A, C, E. A class may extend another class, and an interface may extend another interface, making option A correct. Option B is incorrect. An abstract class can contain concrete instance and `static` methods. Interfaces can also contain nonabstract methods, although knowing this is not required for the 1Z0-815 exam. Option C is correct, as both

can contain `static` constants. Option D is incorrect. The compiler only inserts implicit modifiers for interfaces. For abstract classes, the `abstract` keyword must be used on any method that does not define a body. An abstract class must be declared with the `abstract` keyword, while the `abstract` keyword is optional for interfaces. Since both can be declared with the `abstract` keyword, option E is correct. Finally, interfaces do not extend `java.lang.Object`. If they did, then Java would support true multiple inheritance, with multiple possible parent constructors being called as part of initialization. Therefore, option F is incorrect.

15. 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 nonabstract 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.
16. 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 default (package-private) modifier in the `ClownFish` class, line 6 does not compile, making option E the correct answer. If the method declaration was 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.
17. A, E. An inner class can be marked `abstract` or `final`, just like a regular class, making option A correct. A top-level type, such as a class, interface, or enum, can only be marked `public` or default (package-private), making option B incorrect. Option C is incorrect, as a member

inner class can be marked `public`, and this would not make it a top-level class. A `.java` file may contain multiple top-level classes, making option D incorrect. The precise rule is that there is at most one `public` top-level type, and that type is used in the filename. Finally, option E is correct. When a member inner class is marked `private`, it behaves like any other `private` members and can be referenced only in the class in which it is defined.

18. A, B, D. The `Run` interface correctly overrides the inherited method `move()` from the `Walk` interface using a covariant return type. Options A and B are both correct. Notice that the `Leopard` class does not implement or inherit either interface, so the return type of `move()` can be any valid reference type that is compatible with the body returning `null`. Because the `Panther` class inherits both interfaces, it must override a version of `move()` that is covariant with both interfaces. Option C is incorrect, as `List` is not a subtype of `ArrayList`, and using it here conflicts with the `Run` interface declaration. Option D is correct, as `ArrayList` is compatible with both `List` and `ArrayList` return types. Since the code is capable of compiling, options E and F are incorrect.
19. A, E, F. A class cannot extend any interface, as a class can only extend other classes and interfaces can only extend other interfaces, making option A correct. Java enables only limited multiple inheritance with interfaces, making option B incorrect. True multiple inheritance would be if a class could extend multiple classes directly. Option C is incorrect, as interfaces are implicitly marked `abstract`. Option D is also incorrect, as interfaces do not contain constructors and do not participate in object initialization. Option E is correct, an interface can extend multiple interfaces. Option F is also correct, as abstract types cannot be instantiated.
20. A, D. The implementation of `Elephant` and its member inner class `SleepsAlot` are valid, making options A and D correct. Option B is incorrect, as `Eagle` must be marked `abstract` to contain an abstract method. Option C is also incorrect. Since the `travel()` method does not declare a body, it must be marked `abstract` in an abstract class.

Finally, option E is incorrect, as interface methods are implicitly `public`. Marking them `protected` results in a compiler error.

Chapter 10: Exceptions

1. A, C, D, F. Runtime exceptions are unchecked, making option A correct and option B incorrect. Both runtime and checked exceptions can be declared, although only checked exceptions must be handled or declared, making options C and D correct. Legally, you can handle `java.lang.Error` subclasses, which are not subclasses of `Exception`, but it's not a good idea, so option E is incorrect. Finally, it is true that all exceptions are subclasses of `Throwable`, making option F correct.
2. B, D, E. In a method declaration, the keyword `throws` is used, making option B correct and option A incorrect. To actually throw an exception, the keyword `throw` is used. The `new` keyword must be used if the exception is being created. The `new` keyword is not used when throwing an existing exception. For these reasons, options D and E are correct, while options C and F are incorrect. Since the code compiles with options B, D, and E, option G is incorrect.
3. G. When using a multi-catch block, only one variable can be declared. For this reason, line 9 does not compile and option G correct.
4. B, D. A regular `try` statement is required to have a `catch` clause and/or `finally` clause. If a regular `try` statement does not have any `catch` clauses, then it must have a `finally` block, making option B correct and option A incorrect. Alternatively, a try-with-resources block is not required to have a `catch` or `finally` block, making option D correct and option E incorrect. Option C is incorrect, as there is no requirement a program must terminate. Option F is also incorrect. A try-with-resources statement automatically closes all declared resources. While additional resources can be created or declared in a try-with-resources statement, none are required to be closed by a `finally` block. Option G is also incorrect. The implicit or hidden `finally` block created by the JVM when a try-with-resources statement

is declared is executed first, followed by any programmer-defined `finally` block.

5. C. Line 5 tries to cast an `Integer` to a `String`. Since `String` does not extend `Integer`, this is not allowed, and a `ClassCastException` is thrown, making option C correct. If line 5 were removed, then the code would instead produce a `NullPointerException` on line 7. Since the program stops after line 5, though, line 7 is never reached.
6. E. The code does not compile, so options A, B, and F are incorrect. The first compiler error is on line 12. Each resource in a try-with-resources statement must have its own data type and be separated by a semicolon (`;`). The fact that one of the references is declared `null` does not prevent compilation. 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. Line 14 does compile; since it is an unchecked exception, it does not need to be caught, although in this case it is caught by the `catch` block on line 15.
7. C. The compiler tests the operation for a valid type but not a valid result, so the code will still compile and run. At runtime, evaluation of the parameter takes place before passing it to the `print()` method, so an `ArithmeticException` object is raised, and option C is correct.
8. 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.
9. 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.

10. B. The `main()` method invokes `start` on a new `Laptop` object. Line 4 prints `Starting up_` , and then line 5 throws an `Exception` . Line 6 catches the exception. Line 7 then prints `Problem_` , and line 8 calls `System.exit(0)` , which terminates the JVM. The `finally` block does not execute because the JVM is no longer running. For these reasons, option B is correct.
11. D. The `runAway()` method is invoked within `main()` on a new `Dog` object. Line 4 prints `1` . The `try` block executes, and `2` is printed. Line 7 throws a `NumberFormatException` , so line 8 doesn't execute. The exception is caught on line 9, and line 10 prints `4` . Because the exception is handled, execution resumes normally. The `runAway()` method runs to completion, and line 17 executes, printing `5` . That's the end of the program, so the output is `1245` , and option D is correct.
12. A. The `knockStuffOver()` method is invoked on a new `Cat` object. Line 4 prints `1` . The `try` block is entered, and line 6 prints `2` . Line 7 throws a `NumberFormatException` . It isn't caught, so `knockStuffOver()` ends. The `main()` method doesn't catch the exception either, so the program terminates, and the stack trace for the `NumberFormatException` is printed. For these reasons, option A is correct.
13. A, B, C, D, F. Any Java type, including `Exception` and `RuntimeException` , can be declared as the return type. However, this will simply return the object rather than throw an exception. For this reason, options A and B are correct. Classes listed in the `throws` part of a method declaration must extend `java.lang.Throwable` . This includes `Error` , `Exception` , and `RuntimeException` , making options C, D, and F correct. Arbitrary classes such as `String` can't be declared in a `throws` clause, making option E incorrect.
14. 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 ex-

ception type declared, so it's also correct. Option B is incorrect because a broader exception is not allowed.

15. G. The class does not compile because `String` does not implement `AutoCloseable`, making option G the only correct answer.
16. A, B, D, E, F. Any class that extends `RuntimeException` or `Error`, including the classes themselves, is an unchecked exception, making options D and F correct. The classes `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, and `NumberFormatException` all extend `RuntimeException`, making them unchecked exceptions and options A, B, and E correct. (Sorry, you have to memorize them.) Classes that extend `Exception` but not `RuntimeException` are checked exceptions, making options C and G incorrect.
17. B, F. The `try` block is not capable of throwing an `IOException`. For this reason, declaring it in the `catch` block is considered unreachable code, making 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, but not because of the data type. The variable `c` is declared already in the method declaration, so it cannot be used again as a local variable in the `catch` block. If the variable name was changed, option C would be correct. 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.
18. B. An `IllegalArgumentException` is used when an unexpected parameter is passed into a method. 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!

19. A, D, E, F. An overridden method is allowed to throw no exceptions at all, making option A correct. It is also allowed to throw new unchecked exceptions, making options E and F correct. Option D is also correct since it matches the signature in the interface. Option B is incorrect because it has the wrong return type for the method signature. Option C is incorrect because an overridden method cannot throw new or broader checked exceptions.
20. B, C, E. Checked exceptions are required to be handled or declared, making option B correct. Unchecked exceptions include both runtime exceptions and errors, both of which may be handled or declared but are not required to be making options C and E correct. Note that handling or declaring `Error` is a bad practice.
21. G. The code does not compile, regardless of what is inserted into the blanks. You cannot add a statement after a line that throws an exception. For this reason, line 8 is unreachable after the exception is thrown on line 7, making option G correct.
22. D, F. A `var` is not allowed in a `catch` block since it doesn't indicate the exception being caught, making option A incorrect. With multiple `catch` blocks, the exceptions must be ordered from more specific to broader, or be in an unrelated inheritance tree. For these reasons, options D and F are correct, respectively. Alternatively, if a broad exception is listed before a specific exception or the same exception is listed twice, it becomes unreachable. For these reasons, options B and E are incorrect, respectively. Finally, option C is incorrect because the method called inside the `try` block doesn't declare an `IOException` to be thrown. The compiler realizes that `IOException` would be an unreachable `catch` block.
23. A, E. The code begins normally and prints `a` on line 13, followed by `b` on line 15. On line 16, it throws an exception that's caught on line 17. Remember, only the most specific matching `catch` is run. Line 18 prints `c`, and then line 19 throws another exception. Regardless, the `finally` block runs, printing `e`. Since the `finally` block also throws an exception, that's the one printed.
24. C. The code compiles and runs without issue, so options E and F are incorrect. Since `Sidekick` correctly implements `AutoCloseable`, it can

be used in a try-with-resources statement. The first value printed is `O` on line 9. For this question, you need to remember that a try-with-resources statement executes the resource's `close()` method before a programmer-defined `finally` block. For this reason, `L` is printed on line 5. Next, the `finally` block is expected, and `K` is printed. The `requiresAssistance()` method ends, and the `main()` method prints `I` on line 16. The combined output is `OLKI`, making option C the correct answer.

25. D. The code compiles without issue since `ClassCastException` is a subclass of `RuntimeException` and it is properly listed first, so option E is incorrect. Line 14 executes dividing `1` by itself, resulting in a value of `1`. Since no exception is thrown, options B and C are incorrect. The value returned is on track to be `1`, but the `finally` block interrupts the flow, causing the method to return `30` instead and making option D correct. Remember, barring use of `System.exit()`, a `finally` block is always executed if the `try` statement is entered, even if no exception is thrown or a `return` statement is used.

Chapter 11: Modules

1. B. Option B is correct since modules allow you to specify which packages can be called by external code. Options C and E are incorrect because they are provided by Java without the module system. Option A is incorrect because there is not a central repository of modules. Option D is incorrect because Java defines types.
2. D. Modules are required to have a `module-info.java` file at the root directory of the module. Option D matches this requirement.
3. B. Options A, C, and E are incorrect because they refer to keywords that don't exist. The `exports` keyword 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 `requires` uses module names and not package names.
4. 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.

5. A, F, G. Options C and D are incorrect because there is no `use` keyword. Options A and F are correct because `opens` is for reflection and `uses` declares an API that consumes a service. Option G is also correct as the file can be completely empty. This is just something you have to memorize.
6. 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.
7. D, G. Options A, B, E, and F are incorrect because they refer to keywords that don't exist. The `requires transitive` keyword 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 to `require dog`, and it gets the transitive dependencies, making option D correct.
8. A, B, D. Options A and B are correct because the `-p` (`--module-path`) option can be passed when compiling or running a program. Option D is also correct because `jdeps` can use the `--module-path` option when listing dependency information.
9. A, B. The `-p` specifies the module path. This is just a directory, so all of the options have a legal module path. The `-m` specifies the module, which has two parts separated by a slash. Options E and F are incorrect since there is no slash. The first part is the module name. It is separated by periods (`.`) rather than dashes (`-`), making option C incorrect. The second part is the package and class name, again separated by periods. The package and class names must be legal Java identifiers. Dashes (`-`) are not allowed, ruling out option D. This leaves options A and B as the correct answers.

10. B. A module claims the packages underneath it. Therefore, options C and D are not good module names. Either would exclude the other package name. Options A and B both meet the criteria of being a higher-level package. However, option A would claim many other packages including `com.sybex`. This is not a good choice, making option B the correct answer.
11. B, D, E, F. This is another question you just have to memorize. The `jmod` command has five modes you need to be able to list: `create`, `extract`, `describe`, `list`, and `hash`. The `hash` operation is not an answer choice. The other four are making options B, D, E, and F correct.
12. B. The `java` command uses this option to print information when the program loads. You might think `jar` does the same thing since it runs a program too. Alas, this parameter does not exist on `jar`.
13. 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.
14. 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 that is a package that is imported by default in a class rather than a module.
15. B, D. The `java` command has an `--add-exports` option that allows exporting a package at runtime. However, it is not encouraged to use it, making options B and D the answer.
16. B, C. Option A will run, but it will print details rather than a summary. Options B and C are both valid options for the `jdeps` command. Remember that `-summary` uses a single dash (`-`).
17. 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.
18. A, C. Module names look a lot like package names. Each segment is separated by a period (`.`) and uses characters valid in Java identifiers.

Since identifiers are not allowed to begin with numbers, options E and F are incorrect. Dashes (`-`) are not allowed either, ruling out options B and D. That leaves options A and C as the correct answers.

19. B, C. Option A is incorrect because JAR files have always been available regardless of the JPMS. Option D is incorrect because bytecode runs on the JVM and is not operating system specific by definition. While it is possible to run the `tar` command, this has nothing to do with Java, making option E incorrect. Option B is one of the correct answers as the `jmod` command creates a JMOD file. Option C is the other correct answer because specifying dependencies is one of the benefits of the JPMS.
20. B, E. Option A is incorrect because `describe-module` has the `d` equivalent. Option C is incorrect because `module` has the `m` equivalent. Option D is incorrect because `module-path` has the `p` equivalent. Option F is incorrect because `summary` has the `s` equivalent. Options B and E are the correct answers because they do not have equivalents.
21. 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.

Chapter 12: Java Fundamentals

1. A, D. 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. The `final` modifier can be applied to classes, but not interfaces, making option C incorrect. Remember, interfaces are implicitly `abstract` , which is incompatible with `final` . Option D is correct, as the definition of a `final` class is that it cannot be extended. Option E is incorrect, as `final` refers only to the reference to an object, not its contents. Finally, option F is incorrect, as `var` and `final` can be used together.

2. 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, then it is required. Since the code is missing the semicolon, it does not compile, making option C the correct answer. There are no other compilation errors in this code. If the missing semicolon was added, then the program would print `0 1 2` at runtime.
3. C. `Popcorn` is an inner class. Inner classes are only allowed to contain `static` variables that are marked `final` . Since there are no other compilation errors, option C is the only correct answer. If the `final` modifier was added on line 6, then the code would print `10` at runtime. Note that `private` constructors can be used by any methods within the same class.
4. B, F. A `default` interface method is always `public` , whether you include the identifier or not, making option B correct and option A incorrect. Interfaces cannot contain `default static` methods, making option C incorrect. Option D is incorrect, as `private` interface methods are not inherited and cannot be marked `abstract` . Option E is incorrect, as a method can't be marked both `protected` and `private` . Finally, interfaces can include both `private` and `private static` methods, making option F correct.
5. 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.
   Option A is incorrect because it mixes `var` and non- `var` parameters. If one argument uses `var` , then they all must use `var` . 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. Finally, option G is incorrect. If the type is specified for one argument, then it must be specified for each and every argument.

6. C, E. You can reduce code duplication by moving shared code from `default` or `static` methods into a `private` or `private static` method. For this reason, option C is correct. Option E is also correct, as making interface methods `private` means users of the interface do not have access to them. The rest of the options are not related to `private` methods, although backward compatibility does apply to `default` methods.
7. F. When using an enum in a `switch` statement, 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. For example, `VANILLA` is acceptable but `Flavors.VANILLA` 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.
8. A, C. A functional interface can contain any number of nonabstract 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, and option E is incorrect. Finally, option F is incorrect. While a functional interface can be marked with the `@FunctionalInterface` annotation, it is not required.
9. G. Trick question—the code does not compile! The `Spirit` class is marked `final`, so it cannot be extended. The `main()` method uses an anonymous inner class that inherits from `Spirit`, which is not allowed. If `Spirit` was 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.
10. E. The code `OstrichWrangler` class is a `static` nested class; therefore, it cannot access the instance member `count`. For this reason, line 6 does not compile, and option E is correct. If the `static` modifier on line 4 was removed, then the class would compile and produce two files: `Ostrich.class` and `Ostrich$OstrichWrangler.class`. You

don't need to know that `$` is the syntax, but you do need to know the number of classes and that `OstrichWrangler` is not a top-level class.

11. D. In this example, `CanWalk` and `CanRun` both implement a `default walk()` method. The definition of `CanSprint` extends these two interfaces and therefore won't compile unless the interface overrides both inherited methods. The version of `walk()` on line 12 is an overload, not an override, since it takes an `int` value. Since the interface doesn't override the methods, the compiler can't decide which `default` method to use, leading to a compiler error and making option D the correct answer.
12. C. The functional interface takes two `int` parameters. The code on line `m1` attempts to use them as if one is an `Object`, resulting in a compiler error and making option C the correct answer. It also tries to return `String` even though the return data type for the functional interface method is `boolean`. It is tricky to use types in a lambda when they are implicitly specified. Remember to check the interface for the real type.
13. E, G. For this question, it helps to remember which implicit modifiers the compiler will insert and which it will not. Lines 2 and 3 compile with interface variables assumed to be `public`, `static`, and `final`. Line 4 also compiles, as `static` methods are assumed to be `public` if not otherwise marked. Line 5 does not compile. Non-`static` methods within an interface 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. Finally, line 8 compiles, with no modifiers being added by the compiler.
14. 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 `bc` at runtime.
15. 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. If the code were fixed to implement the `isHealthy()` method in each enum value, then the first three values printed would be `INSECTS`, `1`, and `true`, with the fourth being determined by the implementation of `COOKIES.isHealthy()`.

16. A, D, E. 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. 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.
    `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.
17. 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. Remember from your 1Z0-815 studies that `var` is a reserved type, not a reserved word, and 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.
18. C, D, G. Option C is the correct way to create an instance of an inner class `Cub` using an instance of the outer class `Lion`. The syntax looks weird, but it creates an object of the outer class and then an object of

the inner class from it. Options A, B, and E use incorrect syntax for creating an instance of the `Cub` class. Options D and G are the correct way to create an instance of the static nested `Den` class, as it does not require an instance of `Lion`, while option F uses invalid syntax. Finally, option H is incorrect since it lacks an instance of `Lion`. If `rest()` were an instance method instead of a `static` method, then option H would be correct.

19. D. First off, if a class or interface inherits two interfaces containing `default` methods with the same signature, then 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 the syntax is bizarre, but you need to learn it. Options A, B, and C are incorrect and result in compiler errors.
20. A, B, C, D. Effectively final refers to local variables whose value is not changed after it is set. For this reason, option A is correct, and options E and F are incorrect. Options B and C are correct, as lambda expressions can access `final` and effectively final variables. Option D is also correct and is a common test for effectively final variables.
21. B, E. Like classes, interfaces allow instance methods to access `static` members, but not vice versa. Non-`static` `private`, `abstract`, and `default` methods are considered instance methods in interfaces. Line 3 does not compile because the `static` method `hunt()` cannot access an `abstract` instance method `getName()`. Line 6 does not compile because the `private static` method `sneak()` cannot access the `private` instance method `roar()`. The rest of the lines compile without issue.
22. D, F. Java added `default` methods primarily for backward compatibility. Using a `default` method allows you to add a new method to an interface without having to recompile a class that used an older version of the interface. For this reason, option D is correct. Option F is also correct, as `default` methods in some APIs offer a number of convenient methods to classes that implement the interface. The rest of the options are not related to `default` methods.

23. 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.
24. 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` .
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.

Chapter 13: Annotations

1. E. In an annotation, an optional element is specified with the `default` modifier, followed by a constant value. Required elements are specified by not providing a `default` value. Therefore, the lack of the default term indicates the element is required. For this reason, option E is correct.
2. D, F. Line 5 does not compile because `=` is used to assign a default value, rather than the `default` modifier. Line 7 does not compile because annotation and interface constants are implicitly `public` and cannot be marked `private` . The rest of the lines do not contain any compilation errors.

3. B, D, E. The annotations `@Target` and `@Repeatable` are specifically designed to be applied to annotations, making options D and E correct. Option B is also correct, as `@Deprecated` can be applied to almost any declaration. Option A is incorrect because `@Override` can be applied only to methods. Options C and F are incorrect because they are not the names of built-in annotations.
4. D. Annotations should include metadata (data about data) that is relatively constant, as opposed to attribute data, which is part of the object and can change frequently. The price, sales, inventory, and people who purchased a vehicle could fluctuate often, so using an annotation would be a poor choice. On the other hand, the number of passengers a vehicle is rated for is extra information about the vehicle and unlikely to change once established. Therefore, it is appropriate metadata and best served using an annotation.
5. B, C. Line 4 does not compile because the default value of an element must be a non-`null` constant expression. Line 5 also does not compile because an element type must be one of the predefined immutable types: a primitive, `String`, `Class`, enum, another annotation, or an array of these types. The rest of the lines do not contain any compilation errors.
6. E, G. The annotation declaration includes one required element, making option A incorrect. Options B, C, and D are incorrect because the `Driver` declaration does not contain an element named `value()`. If `directions()` were renamed in `Driver` to `value()`, then options B and D would be correct. The correct answers are options E and G. Option E uses the shorthand form in which the array braces (`{}`) can be dropped if there is only one element. Options C and F are not valid annotation uses, regardless of the annotation declaration. In this question, the `@Documented` and `@Deprecated` annotations have no impact on the solution.
7. A, B, C, D, E, F. Annotations can be applied to all of the declarations listed. If there is a type name used, an annotation can be applied.
8. B, F. In this question, `Ferocious` is the repeatable annotation, while `FerociousPack` is the containing type annotation. The containing type annotation should contain a single `value()` element that is an array

of the repeatable annotation type. For this reason, option B is correct. Option A would allow `FerociousPack` to compile, but not `Ferocious`. Option C is an invalid annotation element type.

The repeatable annotation needs to specify the class name of its containing type annotation, making option F correct. While it is expected for repeatable annotations to contain elements to differentiate its usage, it is not required. For this reason, the usage of `@Ferocious` is a valid marker annotation on the `Lion` class, making option G incorrect.

9. D. To use an annotation with a value but not element name, the element must be declared with the name `value()`, not `values()`, making option A incorrect. The `value()` annotation may be required or optional, making option B incorrect. The annotation declaration may contain other elements, provided none is required, making option C incorrect. Option D is correct, as the annotation must not include any other values. Finally, option E is incorrect, as this is not a property of using a `value()` shorthand.
10. G. `Furry` is an annotation that can be applied only to types. In `Bunny`, it is applied to a method; therefore, it does not compile. If the `@Target` value was changed to `ElementType.METHOD` (or `@Target` removed entirely), then the rest of the code would compile without issue. The use of the shorthand notation for specifying a `value()` of an array is correct.
11. C, D, F. The `@SafeVarargs` annotation can be applied to a constructor or `private`, `static`, or `final` method that includes a varargs parameter. For these reasons, options C, D, and F are correct. Option A is incorrect, as the compiler cannot actually enforce that the operations are safe. It is up to the developer writing the method to verify that. Option B is incorrect as the annotation can be applied only to methods that cannot be overridden and `abstract` methods can always be overridden. Finally, option E is incorrect, as it is applied to the declaration, not the parameters.
12. B, C, D. Annotations cannot have constructors, so line 5 does not compile. Annotations can have variables, but they must be declared with a constant value. For this reason, line 6 does not compile. Line 7 does

not compile as the element `unit` is missing parentheses after the element name. Lines 8 compiles and shows how to use annotation type with a default value.

13. A, D. An optional annotation element is one that is declared with a default value that may be optionally replaced when used in an annotation. For these reasons, options A and D are correct.
14. D. The `@Retention` annotation determines whether annotations are discarded when the code is compiled, at runtime, or not at all. The presence, or absence, of the `@Documented` annotation determines whether annotations are discarded within generated Javadoc. For these reasons, option D is correct.
15. B. A marker annotation is an annotation with no elements. It may or may not have constant variables, making option B correct. Option E is incorrect as no annotation can be extended.
16. F. The `@SafeVarargs` annotation does not take a value and can be applied only to methods that cannot be overridden (marked `private`, `static`, or `final`). For these reasons, options A and B produce compilation errors. Option C also does not compile, as this annotation can be applied only to other annotations. Even if you didn't remember that, it's clear it has nothing to do with hiding a compiler warning. Option D does not compile as `@SuppressWarnings` requires a value. Both options E and F allow the code to compile without error, although only option F will cause a compile without warnings. The `unchecked` value is required when performing unchecked generic operations.
17. B, E. The `@FunctionalInterface` marker annotation is used to document that an interface is a valid functional interface that contains exactly one abstract method, making option B correct. It is also useful in determining whether an interface is a valid functional interface, as the compiler will report an error if used incorrectly, making option E correct. The compiler can detect whether an interface is a functional interface even without the annotation, making options A and C incorrect.
18. C, D, E, F. Line 5 and 6 do not compile because `Boolean` and `void` are not supported annotation element types. It must be a primitive,

`String`, `Class`, enum, another annotation, or an array of these types. Line 7 does not compile because annotation elements are implicitly `public`. Finally, line 8 does not compile because the `Strong` annotation does not contain a `value()` element, so the shorthand notation cannot be used. If line 2 were changed from `force()` to `value()`, then line 8 would compile. Without the change, though, the compiler error is on line 8. The rest of the lines do not contain any compilation errors, making options C, D, E, and F correct.

19. A, F. The `@Override` annotation can be applied to a method but will trigger a compiler error if the method signature does not match an inherited method, making option A correct. The annotation `@Deprecated` can be applied to a method but will not trigger any compiler errors based on the method signature. The annotations `@FunctionalInterface`, `@Repeatable`, and `@Retention` cannot be applied to methods, making these options incorrect. Finally, `@SafeVarargs` can be applied to a method but will trigger a compiler error if the method does not contain a varargs parameter or is able to be overridden (not marked `private`, `static`, or `final`).
20. D, F. Line 6 contains a compiler error since the element name `buoyancy` is required in the annotation. If the element were renamed to `value()` in the annotation declaration, then the element name would be optional. Line 8 also contains a compiler error. While an annotation can be used in a cast operation, it requires a type. If the cast expression was changed to `(@Floats boolean)`, then it would compile. The rest of the code compiles without issue.
21. G. The `@Inherited` annotation determines whether or not annotations defined in a super type are automatically inherited in a child type. The `@Target` annotation determines the location or locations an annotation can be applied to. Since this was not an answer choice, option G is correct. Note that `ElementType` is an enum used by `@Target`, but it is not an annotation.
22. F. If `@SuppressWarnings("deprecation")` is applied to a method that is using a deprecated API, then warnings related to the usage will not be shown at compile time, making option F correct. Note that there are no built-in annotations called `@Ignore` or `@IgnoreDeprecated`.

23. A. This question, like some questions on the exam, includes extraneous information that you do not need to know to solve it. Therefore, you can assume the reflection code is valid. That said, this code is not without problems. The default retention policy for all annotations is `RetentionPolicy.CLASS` if not explicitly stated otherwise. This means the annotation information is discarded at compile time and not available at runtime. For this reason, none of the members will print anything, making option A correct.
    If `@Retention(RetentionPolicy.RUNTIME)` were added to the declaration of `Plumber`, then the `worker` member would cause the default annotation `value()`, `Mario`, to be printed at runtime, and option B would be the correct answer. Note that `foreman` would not cause Mario to be printed even with the corrected retention annotation. Setting the value of the annotation is not the same as setting the value of the variable `foreman`.
24. A, E. The annotation includes only one required element, and it is named `value()`, so it can be used without an element name provided it is the only value in the annotation. For this reason, option A is correct, and options B and D are incorrect. Since the type of the `value()` is an array, option B would work if it was changed to `@Dance({33, 10})`. Option C is incorrect because it attempts to assign a value to `fast`, which is a constant variable not an element. Option E is correct and is an example of an annotation replacing all of the optional values. Option F is incorrect, as `value()` is a required element.
25. C. The Javadoc `@deprecated` annotation should be used, which provides a reason for the deprecation and suggests an alternative. All of the other answers are incorrect, with options A and B having the wrong case too. Those annotations should be written `@Repeatable` and `@Retention` since they are Java annotations.

Chapter 14: Generics and Collections

1. B. The answer needs to implement `List` because the scenario allows duplicates. Since you need a `List`, you can eliminate options C and D

immediately because `HashMap` is a `Map` and `HashSet` is a `Set`. Option A, `Arrays`, is trying to distract you. It is a utility class rather than a `Collection`. An array is not a collection. This leaves you with options B and E. Option B is a better answer than option E because `LinkedList` is both a `List` and a `Queue`, and you just need a regular `List`.

2. D. The answer needs to implement `Map` because you are dealing with key/value pairs per the unique `id` field. You can eliminate options A, C, and E immediately since they are not a `Map`. `ArrayList` is a `List`. `HashSet` and `TreeSet` are `Set`s. Now it is between `HashMap` and `TreeMap`. Since the question talks about ordering, you need the `TreeMap`. Therefore, the answer is option D.
3. 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.
4. D. This is a FIFO (first-in, first-out) queue. On line 7, we remove the first element added, which is `"hello"`. On line 8, we look at the new first element (`"hi"`) but don't remove it. On lines 9 and 10, we remove each element in turn until no elements are left, printing `hi` and `ola` together. Note that we don't use an `Iterator` to loop through the `LinkedList` to avoid concurrent modification issues. The order in which the elements are stored internally is not part of the API contract.
5. 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 be-

cause 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 `=`.

6. B. The class compiles and runs without issue. Line 10 gives a compiler warning for not using generics but not a compiler error. Line 4 compiles fine because `toString()` is defined on the `Object` class and is therefore always available to call.
   Line 9 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 5. Therefore, the `String` argument on line 9 applies only to the class. The method can actually take any object as a parameter including autoboxed primitives. Line 10 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.
7. A, D. The code compiles fine. It allows any implementation of `Number` to be added. Lines 5 and 8 succesfully autobox the primitives into an `Integer` and `Long`, respectively. `HashSet` does not guarantee any iteration order, making options A and D correct.
8. 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 the `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.

9. E. Trick question! The `Map` interface uses `put()` rather than `add()` to add elements to the map. If these examples used `put()`, the answer would be options A and C. Option B is no good because a `long` cannot be placed inside a `Double` without an explicit cast. Option D is no good because a `char` is not the same thing as a `String`.
10. 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`.
11. A. Line 3 uses local variable type inference to create the map. Lines 5 and 7 use autoboxing to convert between the `int` primitive and the `Integer` wrapper class. The keys map to their squared value. `1` maps to `1`, `2` maps to `4`, `3` maps to `9`, `4` maps to `16`, and so on.
12. A, B, D. The generic type must be `Exception` or a subclass of `Exception` since this is an upper bound. Options C and E are wrong because `Throwable` is a superclass of `Exception`. Option D uses an odd syntax by explicitly listing the type, but you should be able to recognize it as acceptable.
13. 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`.
14. 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 did specify a `Comparator` when calling the constructor, so it uses the `compare()` method, which sorts by the `int`.

15. 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.
16. 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.
17. B, D. Line 1 is a generic class that requires specifying a name for the type. Options A and C are incorrect because no type is specified. While you can use the diamond operator `<>` and the wildcard `?` on variables and parameters, you cannot use them in a class declaration. This means option B is the only correct answer for line 1. Knowing this allows you to fill in line 3. Option E is incorrect because `T` is not a class and certainly not one compatible with `String`. Option F is incorrect because a wildcard cannot be specified on the right side when instantiating an object. We're left with the diamond operator, making option D correct.
18. 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`.
19. A, D. 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. `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`.
20. 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. If `containsKey()` was called, the answer would be `false` because `123` is an `Integer` key in the `Map`, rather than a `String`.

21. 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 compile error on line 56 since we can't multiply an `Entry` object by two.
    Lines 51 through 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.
22. B. When using generic types in a method, the generic specification goes before the return type.
23. B, E. Both `Comparator` and `Comparable` are functional interfaces. However, `Comparable` is intended to be used on the object being compared, making option B correct. The `removeIf()` method allows specifying the lambda to check when removing elements, making option E correct. Option C is incorrect because the `remove()` method takes an instance of an object to look for in the `Collection` to remove. Option D is incorrect because `removeAll()` takes a `Collection` of objects to look for in the `Collection` to remove.
24. F. The first two lines correctly create a `Set` and make a copy of it. Option A is incorrect because `forEach` takes a `Consumer` parameter, which 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.
25. F. The first call to `merge()` calls the mapping function and adds the two 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.

Chapter 15: Functional Programming

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.
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.
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." 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 10 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. B, F, G. We can eliminate four choices right away. Options A and C are there to mislead you; these interfaces don't actually 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`.

    Moving on to the remaining choices, 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. Another clue is that it uses a constructor reference, which should scream `Supplier`! This makes 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.

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>`. 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. First, converting from a `double` to an `int` would require an explicit cast. Also, `mapToInt()` returns an `IntStream` so the data type of `s3` 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 this code starts out with an infinite stream on line 23, it does become 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. 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` actually 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.
18. 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. 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.
19. 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.
20. 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. There is one exception to this rule. While there is no `BooleanStream` class, 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.
21. 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 simpliest way, which is option B.
22. 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.

Chapter 16: Exceptions, Assertions, and Localization

1. C. `Exception` and `RuntimeException`, along with many other exceptions in the Java API, define a no-argument constructor, a constructor that takes a `String`, and a constructor that takes a `Throwable`. For this reason, `Danger` compiles without issue. `Catastrophe` also compiles without issue. Just creating a new checked exception, without throwing it, does not require it to be handled or declared. Finally, `Emergency` does not compile. The no-argument constructor in `Emergency` must explicitly call a parent constructor, since `Danger` does not define a no-argument constructor.
2. A, D, E. Localization refers to user-facing elements. Dates, currency, and numbers are commonly used in different formats for different countries. Class and variable names, along with lambda expressions, are internal to the application, so there is no need to translate them for users.
3. G. A try-with-resources statement uses parentheses, `()`, rather than braces, `{}`, for the `try` section. This is likely subtler than a question that you'll get on the exam, but it is still important to be on alert for details. If parentheses were used instead of braces, then the code would compile and print `TWDF` at runtime.
4. 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, then the rest of the code would compile and print a stack track with `YesProblem` at runtime.
5. E. A `LocalDate` does not have a time element. Therefore, a Date/Time formatter is not appropriate. The code compiles but throws an excep-

tion at runtime. If `ISO_LOCAL_DATE` was used, then the code would compile and option B would be the correct answer.

6. 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.
7. 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. In case you are curious, option A displays at most only one value to the right of the decimal, printing `<5.2> <8.5> <1234>`. Option B is close to the correct answer but always displays four digits to the left of the decimal, printing `<0,005.21> <0,008.49> <1,234.0>`. Finally, option C is missing the zeros padded to the left of the decimal and optional two values to the right of the decimal, printing `<5.2> <8.5> <1,234.0>`.
8. A, D. An assertion consists of a `boolean` expression followed by an optional colon (`:`) and message. The `boolean` expression is allowed to be in parentheses, but this is not required. Therefore, options A and D are correct.
9. B, E. 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.
10. B, C. The code does not compile, so option E is incorrect. 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.

Finally, option F is incorrect because the changes for option B or C will allow the code to compile.

11. C. The code compiles fine, so option E is incorrect. The command line has only two arguments, so `args.length` is `2` and the `if` statement is `true`. However, because assertions are not enabled, it does not throw an `AssertionError`, so option B is incorrect. The `println()` method attempts to print `args[2]`, which generates an `ArrayIndexOutOfBoundsException`, so the answer is option C.
12. 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, `{}`.
13. 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. Options E and F are incorrect because a `Locale` is created using a constructor or `Locale.Builder` class.
14. D. You can create custom checked, unchecked exceptions, and even errors. The default constructor is used if one is not supplied. There is no requirement to implement any specific methods.
15. F. The code compiles, but the first line produces a runtime exception regardless of what is inserted into the blank. 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'"` was 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.
16. B, C. The code compiles, so option E is incorrect. While it is a poor practice to modify variables in an assertion statement, it is allowed. To enable assertions, use the flag `-ea` or `-enableassertions`. To disable assertions, use the flag `-da` or `-disableassertions`. The colon indicates a specific class. Option A is incorrect, as assertions are already

disabled by default. Option B is correct because it turns on assertions for all classes (except system classes). Option C is correct because it disables assertions for all classes but then turns them back on for this class. Finally, option D is incorrect as it enables assertions everywhere except the `On` class.

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. In fact, `ResourceBundle` is an `abstract` class, so calling that constructor is not even possible. 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, making it possible to use a resource bundle for a locale, even if a locale was not explicitly set.
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 reverse order from 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`.
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`.
20. B. The `MessageFormat` class supports parametrized `String` values that take input values, while the `Properties` class supports providing

a default value if the property is not set. For this reason, option B is correct.

21. C. The code does not compile because the multi-`catch` 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. E. Even though `ldt` has both a date and time, the formatter outputs only time.
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.
24. E. The `Properties` class defines a `get()` method that does not allow for a default value. It also has a `getProperty()` method, which returns the default value if the key is not provided.
25. G. The code does compile because the resource `walk1` is not `final` or effectively final and cannot be used in the declaration of a try-with-resources statement. If the line that set `walk1` to `null` was 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.
26. A, E. Line 5 does not compile because `assert` is a keyword, making option A correct. Options B and C are both incorrect because the parentheses and message are both optional. Option D is incorrect because assertions should never alter outcomes, as they may be disabled at runtime. Option E is correct because checking an argument passed from elsewhere in the program is an appropriate use of an assertion.
27. E. The `Locale.Builder` class requires that the `build()` method be called to actually create the `Locale` object. For this reason, the two

`Locale.setDefault()` statements do not compile because the input is not a `Locale`, making option E the correct answer. If the proper `build()` calls were added, then the code would compile and print the value for Germany, `2,40 €`. As in the exam, though, you did not have to know the format of currency values in a particular locale to answer the question. Note that the default locale category is ignored since an explicit currency locale is selected.

Chapter 17: Modular Applications

1. 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.
2. E, F. Automatic modules are on the module path but do not have a `module-info` 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.
3. 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.
4. A, E. 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 E correct, while options C and D are incorrect. Finally. Option F is incorrect because unnamed modules cannot be referenced from an automatic module.
5. B. Option B is correct because `java.base` is provided by default. It contains the `java.lang` package among others.

6. F. The `provides` directive takes the interface name first and the implementing class name second. The `with` keyword is used. Only option F meets these two criteria, making it the correct answer.
7. 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.
8. 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.
9. C. The `jdeps` command has an option `--internal-jdk` that lists any code using unsupported/internal APIs and prints a table with suggested alternatives. This makes option C correct. Option D is incorrect because it does not print out the table with a suggested alternative. Options A, B, E, F, and G are incorrect because those options do not exist.
10. 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.
11. A. Since this is a new module, you need to compile the new module. 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 the new service provider on the module path.
12. B. The most commonly used packages are in the `java.base` module, making option B correct.
13. A, E, F. Option A is correct because the service provider interface must specify `exports` for any other modules to reference it. Option F is correct because the service provider needs access to the service provider interface. Option E is also correct because the service provider needs to declare that it provides the service.

14. B, E, F. Since the new project extracts the common code, it must have an `exports` directive for that code, making option B correct. The other two modules do not have to expose anything. They must have a `requires` directive to be able to use the exported code, making options E and F correct.
15. H. This question is tricky. The service provider 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.
16. A. Since the JAR is on the classpath, it is treated as a regular unnamed module even though it has a `module-info` 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.
17. 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.
18. A, C, D. Option 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.
19. C, E. The `jdeps` command provides information about the class or package level depending on the options passed, making option C correct. It is frequently used to determine what dependencies you will need when converting to modules. This makes it useful to run against a regular JAR, making option E correct.
20. E. Trick question! An unnamed module doesn't use a `module-info` 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.

Chapter 18: 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, 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 will throw an `IllegalMonitorStateException` at runtime, making option D correct. Option B would be possible if there was no lock at all, although in this case, failure to acquire a lock results in an exception at runtime.
3. A, C, D, F. All methods are capable of throwing unchecked exceptions, so option A is correct. `Runnable` and `Callable` statements both do not take any arguments, so option B is incorrect. Only `Callable` is capable of throwing checked exceptions, so option C is also correct. Both `Runnable` and `Callable` are functional interfaces that can be implemented with a lambda expression, so option D is also correct. Finally, `Runnable` returns `void` and `Callable` returns a generic type, making option F correct and making options E and G incorrect.
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 could 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 they are stored in the stream, making option E correct. It does not sort the elements, so option D is 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 are completed at the same time, which should have been completed sequentially.
8. A. The method looks like it executes a task concurrently, but it actually 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 actually thread-safe. It executes a total of 499 times, since the second value of `range()` excludes the 500. Since the program consistently prints `499` at runtime, option A 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 im-

mediately 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 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 actually 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 `int` and `c2` is a `String`, so the code fails to combine on line `q2`, since calling `length()` on an `int` 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 in fact 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, then 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. 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 `tryLock()`.
17. D. The methods on line 5, 6, 7, and 8 each throw `InterruptedException`, which is a checked exception; therefore, the method does not compile, and option D is the only correct answer. If `InterruptedException` was declared in the method signature on line 3, then the answer would be option F, because adding items to the `queue` may be blocked at runtime. In this case, the queue is passed into the method, so there could be other threads operating on it. Finally, if the operations were not blocked and there were no other op-

erations on the `queue`, then the output would be `103 20`, and the answer would be option B.

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. F, H. The application compiles and does not throw an exception, so options B, C, D, E, and G are incorrect. 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 F is correct. Finally, the thread executor is never shut down; therefore, the code will run but it will never terminate, making option H 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()`, then 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. Option E is incorrect because any of the previous options will allow the code to compile.
22. B. The code compiles and runs without issue, so options D, E, F, and G are incorrect. 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 pooled thread executor was used with at least two threads, then the `sheepCount2++`

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, and options B and C are incorrect. 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. For example, one thread could call `sellTickets()` while another thread calls `addTickets()`. These methods are not synchronized with each other and could cause an invalid number of tickets due to a race condition.
    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 actually a singleton pattern.
25. A, D. By itself, concurrency does not guarantee which task will be completed first, so option A is correct. Furthermore, applications with numerous resource requests will often be stuck waiting for a resource, which allows other tasks to run. Therefore, they tend to benefit more from concurrency than CPU-intensive tasks, so option D is also correct. Option B is incorrect because concurrency may in fact make an application slower if it is truly single-threaded in nature. Keep in mind that there is a cost associated with allocating additional memory and CPU time to manage the concurrent process. Option C is incorrect because single-processor CPUs have been benefiting from concurrency for

decades. Finally, option E is incorrect; there are numerous examples in this chapter of concurrent tasks sharing memory.

26. 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 19: I/O

1. F. 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. `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. C, E, G. The command to move a file or directory using a `File` is `renameTo()`, not `mv()` or `move()`, making options A and D incorrect, and option E correct. The commands to create a directory using a `File` are `mkdir()` and `mkdirs()`, not `createDirectory()`, making option B incorrect, and options C and G correct. The `mkdirs()` differs from `mkdir()` by creating any missing directories along the path. Finally, option F is incorrect as there is no command to copy a file in the `File` class. You would need to use an I/O stream to copy the file along with its contents.

3. B. 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 nonserializable parent. In this case, the `Bird` constructor is called, with `name` being set to `Matt`, making option B correct. Note that none of the constructors or instance initializers in `Eagle` is executed as part of deserialization.
4. A, D. 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 B is incorrect, as `FileInputStream` operates on a file, not another stream. Finally, option D is correct—an `ObjectInputStream` is a high-level stream that operates on other streams.
5. B, E. The JVM creates one instance of the `Console` object as a singleton, making option C incorrect. If the console is unavailable, `System.console()` will return `null`, making option B correct. The method cannot throw an `IOException` because it is not declared as a checked exception. Therefore, option A is incorrect. Option D is incorrect, as a `Console` can be used for both reading and writing data. The `Console` class includes a `format()` method to write data to the output stream, making option E correct. Since there is no `println()` method, as `writer()` must be called first, option F is incorrect.
6. C, D, E. All I/O streams should be closed after use or a resource leak might ensue, making option C correct. While a try-with-resources

statement is the preferred way to close an I/O stream, it can be closed with a traditional `try` statement that uses a `finally` block. For this reason, both options D and E are correct.

7. G. Not all I/O streams support the `mark()` operation; therefore, without calling `markSupported()` on the stream, the result is unknown until runtime. If the stream does support the `mark()` operation, then the result would be `XYZY`, and option D would be correct. The `reset()` operation puts the stream back in the position before the `mark()` was called, and `skip(1)` will skip `X`. If the stream does not support the `mark()` operation, a runtime exception would likely be thrown, and option F would be correct. Since you don't know if the input stream supports the `mark()` operation, option G is the only correct choice.
8. A, F. In Java, serialization is the process of turning an object to a stream, while deserialization is the process of turning that stream back into an object. For this reason, option A is correct, and option B is incorrect. Option C is incorrect, because many nonthread classes are not marked `Serializable` for various reasons. The `Serializable` interface is a marker interface that does not contain any abstract methods, making options D and E incorrect. Finally, option F is correct, because `readObject()` declares the `ClassNotFoundException` even if the class is not cast to a specific type.
9. A. 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.
10. E, F. 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 E and F are correct. Marking a class `final` does not impact its ability to be serialized, so option A is incorrect. Option B is incorrect, as `Serializable` is an interface, not a class. Option C is incorrect. While it is a good practice for a serializable class to include a `static serialVersionUID` variable, it

is not required. Finally, option D is incorrect as `static` members of the class are ignored on serialization already.

11. C. The code compiles, so options D and E are incorrect. The method looks like it will delete a directory tree but contains a bug. It never deletes any directories, only files. The result of executing this program is that it will delete all files within a directory tree, but none of the directories. For this reason, option C is correct.
12. E. The code does not compile, as the `Writer` methods `append()` and `flush()` both throw an `IOException` that must be handled or declared. Even without those lines of code, the try-with-resources statement itself must be handled or declared, since the `close()` method throws a checked `IOException` exception. For this reason, option E is correct. If the `main()` method was corrected to declare `IOException`, then the code would compile. If the `Console` was not available, it would throw a `NullPointerException` on the call to `c.writer()`; otherwise, it would print whatever the user typed in. For these reasons, options B and D would be correct.
13. B, E. 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.
14. A, C, E. The `System` class has three streams: `in` is for input, `err` is for error, and `out` is for output. Therefore, options A, C, and E are correct. The others do not exist.
15. E. `PrintStream` and `PrintWriter` are the only I/O classes that you need to know that don't have a complementary `InputStream` or `Reader` class, so option E is correct.
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 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. C. The code compiles without issue. Since we're told the `Reader` supports `mark()`, the code also runs without throwing an exception. `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, 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.
18. B, C, D. Since you need to write primitives and `String` values, the `OutputStream` classes are appropriate. Therefore, you can eliminate options A and F since they use `Writer` classes. Next, `DirectoryOutputStream` is not a `java.io` class, so option E is incorrect. The data should be written to the file directly using the `FileOutputStream` class, buffered with the `BufferedOutputStream` class, and automatically serialized with the `ObjectOutputStream` class, so options B, C, and D are correct. `PrintStream` is an `OutputStream`, so it could be used to format the data. Unfortunately, since everything is converted to a `String`, the underlying data type information would be lost. For this reason, option G is incorrect.
19. 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 was called after every write. Finally, option G is correct as the `reader` stream is never closed.

20. C. `Console` includes a `format()` method that takes a `String` along with a list of arguments and writes it directly to the output stream, making option C correct. Options A and B are incorrect, as `reader()` returns a `Reader`, which does not define any print methods. Options D and E would be correct if the line was just a `String`. Since neither of those methods take additional arguments, they are incorrect.
21. A, C. Character stream classes often include built-in convenience methods for working with `String` data, so option A is correct. They also handle character encoding automatically, so option C is also correct. The rest of the statements are irrelevant or incorrect and are not properties of all character streams.
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`, then 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, then options A and C would be the correct answers, as `name` and `age` are both marked `transient`.

Chapter 20: NIO.2

1. E. The `relativize()` method takes a `Path` value, not a `String`. For this reason, line 5 does not compile, and option E is correct. If line 5 was corrected to use a `Path` value, then the code would compile, but it would print the value of the `Path` created on line 4. Since `Path` is immutable, the operations on line 5 are not saved anywhere. For this reason, option D would be correct. Finally, if the value on line 5 was assigned to `path` and printed on line 6, then option A would be correct.
2. 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 was corrected to declare the appropriate exceptions, then 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.

3. 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.
4. C. First, the code compiles and runs without issue, so options F and G are incorrect. Let's take this one step at a time. First, the `subpath()` method 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 zero, it returns the relative path `bear`. Finally, the `toAbsolutePath()` method is applied to the relative path `bear`, resulting in the current directory `/user/home` being incorporated into the path. The final output is the absolute path `/user/home/bear`, making option C correct.
5. 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.
6. 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 was rewritten to use the `Files.find()` method with the `BiPredicate` as input (along with a `maxDepth` value), then 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.

7. F. The code compiles without issue, so options D and E are incorrect. The method `Files.isSameFile()` first checks to see whether the `Path` values are the same in terms of `equals()`. Since the first path is relative and the second path is absolute, this comparison will return `false`, forcing `isSameFile()` to check for the existence of both paths in the file system. Since we know `/zoo/turkey` does not exist, a `NoSuchFileException` is thrown, and option F is the correct answer. Options A, B, and C are incorrect since an exception is thrown at runtime.
8. 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`.
9. C. The code compiles and runs without issue, so option E is incorrect. For this question, you have to remember two things. First, the `resolve()` method does not normalize any path symbols, so options A and B are not correct. Second, calling `resolve()` with an absolute path as a parameter returns the absolute path, so option C is correct, and option D is incorrect.
10. B, C. The methods are not the same, because `Files.lines()` returns a `Stream<String>` and `Files.readAllLines()` returns a `List<String>`, so option F is incorrect. Option A is incorrect, because performance is not often the reason to prefer one to the other. `Files.lines()` processes each line via lazy evaluation, while `Files.readAllLines()` reads the entire file into memory all at once. For this reason, `Files.lines()` works better on large files with limited memory available, and option B is correct. Although a `List` can be converted to a stream, this requires an extra step; therefore, option C is correct since the resulting object can be chained directly to a

stream. Finally, options D and E are incorrect because they are true for both methods.

11. 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. Option B will actually 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. Option E is incorrect, since there are circumstances in which the operation would be allowed. In particular, if `/animals` did not exist then the operation would complete successfully.
12. A, C, E. Options A, C, and E are all properties of NIO.2 and are good reasons to use it over the `java.io.File` class. Option B is incorrect, as both `java.io.File` and NIO.2 include a method to list the contents of a directory. Option D is also incorrect as both APIs can delete only empty directories, not a directory tree. Finally, option F is incorrect, as sending email messages is not a feature of either API.
13. A. The code compiles and runs without issue, so options C, D, and E are incorrect. Even though the file is copied with attributes preserved, the file is considered a separate file, so the output is `false`, making option A correct and option B incorrect. Remember, `isSameFile()` returns `true` only if the files pointed to in the file system are the same, without regard to the file contents.
14. 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.
15. E. The `java.io.File` method `listFiles()` retrieves the members of the current directory without traversing any subdirectories. Option E is correct, as `Files.list()` returns a `Stream<Path>` of a single direc-

tory. `Files.walk()` is close, but it iterates over the entire directory tree, not just a single directory. The rest of the methods do not exist.

16. D, E, F. Whether a path is a symbolic link, file, or directory is not relevant, so options A and C are incorrect. Using a view to read multiple attributes leads to fewer round-trips between the process and the file system and better performance, so options D and F are correct. For reading single attributes, there is little or no expected gain, so option B is incorrect. Finally, views can be used to access file system–specific attributes that are not available in `Files` methods; therefore, option E is correct.
17. 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 instead used, it would print the contents of the file one capitalized word at a time, with commas removed.
18. 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.
19. D. The code compiles and runs without issue, so option F is incorrect. If you simplify the redundant path symbols, then `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, the normalized paths are the same, since all extra symbols have been removed, so the last line outputs `true`. For these reasons, option D is correct.
20. B. The `normalize()` method does not convert a relative path into an absolute path; therefore, the path value after the first line is just the

current directory symbol. The `for()` loop iterates the name values, but since there is only one entry, the loop terminates after a single iteration. Therefore, option B is correct.

21. 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, then 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.
22. B, E. Option F is incorrect, as the code does compile. The method copies the contents of a file, but it removes all the line breaks. The `while()` loop would need to include a call to `w.newLine()` to correctly copy the file. For this reason, option A is incorrect. Option B is correct, and options C and D are incorrect. The `APPEND` option creates the file if it does not exist; otherwise, it starts writing from the end of the file. Option E is correct because the resources created in the method are not closed or declared inside a try-with-resources statement.

Chapter 21: JDBC

1. B, F. The `Driver` and `PreparedStatement` interfaces are part of the JDK, making options A and E incorrect. The concrete `DriverManager` class is also part of the JDK, making options C and D incorrect. Options B and F are correct since the implementation of these interfaces is part of the database-specific driver JAR file.
2. C, F. A JDBC URL has three parts. The first part is the string `jdbc`, making option C correct. The second part is the subprotocol. This is the vendor/product name, which isn't an answer choice. The subname is vendor-specific, making option F correct as well.

3. 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.
4. 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.
5. 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.
6. 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.
7. A, B, E. CRUD stands for Create, Read, Update, Delete, making options A, B, and E correct.
8. 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.
9. 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.
10. E. The code compiles because `PreparedStatement` extends `Statement` and `Statement` allows passing a `String` in the `executeQuery()` call. While `PreparedStatement` can have bind variables, `Statement` cannot. Since this code uses `executeQuery(sql)` in

`Statement`, it fails at runtime. A `SQLException` is thrown, making option E correct.

11. 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.
12. 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`.
13. 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. A total of two lines is printed, making option C correct.
14. 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.
15. C. Since an `OUT` parameter is used, the code should call `registerOutParameter()`. Since this is missing, option C is correct.
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. A total of four lines are printed, 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.

19. E. This code should call `prepareStatement()` instead of `prepareCall()` since it 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. E. Since the code calls `registerOutParameter()`, we know the stored procedure cannot use an `IN` parameter. Further, there is no `setInt()`, so it cannot be an `INOUT` parameter either. Therefore, the stored procedure must use an `OUT` parameter, making option E the answer.
21. B. The `prepareStatement()` method requires SQL be passed in. Since this parameter is omitted, line 27 does not compile, and option B is correct. Line 30 also does not compile as the method should be `getInt()`. However, the question asked about the first compiler error.

Chapter 22: Security

1. D. A distributed denial of service (DDoS) attack requires multiple requests by definition. Even a regular denial of service attack often requires multiple requests. For example, if you forget to close resources, it will take a number of tries for your application to run out resources. Therefore, option D is correct.
2. A, C, D. Since the class is `final`, it restricts extensibility, making option D correct. The `private` variable limits accessibility, making option C correct. Finally, option A is correct. This is an immutable class since it's not possible to change the state after construction. This class does not do any validation, making option B incorrect.
3. A. The `PutField` class is used with the `writeObject()` method, making option A correct. There is also a `GetField` class used with the `readObject()` method.
4. B, D. Option A is incorrect because it does not make a copy. Options E and F are incorrect because `ArrayList` does not have a `copy()`

method. Option C is incorrect because the `clone()` method returns an `Object` and needs to be cast, so that option does not compile. Options B and D are correct because they copy the `ArrayList` using the copy constructor and `clone()` method, respectively.

5. D. When deserializing an object, Java does not call the constructor. Therefore, option D is correct.
6. A, F. The `clone()` method is declared on the `Object` class. Option A is correct because it will always compile. However, the call will throw an exception if the class that is being cloned does not implement `Cloneable`. Assuming this interface is implemented, the default implementation creates a shallow copy, making option F correct. If the class wants to implement a deep copy, it must override the `clone()` method with a custom implementation.
7. F. This is a trick question—there is no attack. Option E is incorrect because SQL leak is not the name of an attack. Option C is incorrect because the `PreparedStatement` and `ResultSet` are closed in a try-with-resources block. While we do not see the `Connection` closed, we also don't see it opened. The exam allows us to assume code that we can't see is correct. Option D is an incorrect answer because bind variables are being used properly with a `PreparedStatement`. Options A and B are incorrect because they are not related to the example. Since none of these attacks applies here, option F is correct.
8. E. The policy compiles and uses correct syntax. However, it gives permissions that are too broad. The user needs to be able to read a book, so `write` permissions should not be granted.
9. A, C. Many programs use confidential information securely, making option A correct. After all, you wouldn't be able to bank online if programs couldn't work with confidential information. It is also OK to put it in certain data structures. A built-in Java API puts a password in a `char[]`, making option C correct. Exposing the information unintentionally is not OK, making option B incorrect. Sharing confidential information with others is definitely not OK, making option D incorrect.
10. C, D. Any resource accessing things outside your program should be closed. Options C and D are correct because I/O and JDBC meet this criteria.

11. A, F. An inclusion attack needs to include something. Options A and F are correct because they are used with XML and ZIP file respectively. Options B and D are incorrect because injection is not an inclusion attack. Options C and E are not inclusion attacks either. In fact, you might not have heard of them. Both are attacks used against web applications. Don't worry if you see something on the exam that you haven't heard of; it isn't a correct answer.
12. D. The validation code checks that each character is between 0 and 9. Since it is comparing to allowed values, this is an example of a whitelist, and option D is correct. If it were the opposite, it would be a blacklist. There is no such thing as a gray or orange list.
13. E, F. Option A is incorrect because good encapsulation requires `private` state rather than declaring the class `final`. Option B is incorrect because the well-encapsulated `Camel` class can have a getter that exposes the `species` variable to be modified. Options C and D are incorrect because a class can be `final` while having `public` variables and be mutable. Option E is correct because methods that expose `species` could change it, which would prevent immutability. Option F is correct because you cannot enforce immutability in a subclass.
14. B, C, D. Any information the user can see requires care. Options B, C, and D are correct for this reason. Comments and variable names are part of the program, not the data it handles, making options A and E incorrect.
15. A, B, F. The `serialPersistentFields` field is used to specify which fields should be used in serialization. It must be declared `private static final`, or it will be ignored. Therefore, options A, B, and F are correct.
16. C, D. The application should log a message or throw an exception, making options C and D correct. It should not immediately terminate the program with `System.exit()` as that does not execute gracefully, making option A incorrect. It also should not ignore the issue, making option B incorrect.
17. A, B, E. Options A and E are correct because they prevent subclasses from being created outside the class definition. Option B is also correct because it prevents overriding the method. Options C and D are incor-

rect because `transient` is a modifier for variables, not classes or methods.

18. A. Reading an extremely large file is a form of a denial of service attack, making option A correct.
19. C. Options B and F are incorrect because these method names are not used by any serialization or deserialization process. Options A and D are incorrect because the return type for these methods is `void`, not `Object`. Option E is almost correct, as that is a valid method signature, but our question asks for the method used in deserialization, not serialization. Option C is the correct answer.
20. C. A shallow copy does not create copies of the nested objects, making option C correct. Options B and D are incorrect because narrow and wide copies are not terms. Option A is incorrect because a deep copy does copy the nested objects.