Appendix D

Annotated Answers to Review Questions

1 Basics of Java Programming

1.1 *(c)*

A method is an operation defining a particular behavior of an abstraction. Java implements abstractions using classes that have properties and behaviors. Behaviors are defined by the operations of the abstraction.

1.2 *(b)*

An object is an instance of a class. Objects are created from classes that implement abstractions. The objects that are created are concrete realizations of those abstractions. An object is neither a reference nor a variable.

1.3 *(b)*

(2) is the first line of a constructor declaration. A constructor in Java is declared like a method that does not return a value. It has the same name as the class name, but it does not specify a return type and therefore does not return a value. (1) is the header of a class declaration, (3) is the first statement in the constructor body, and (4), (5), and (6) are instance method declarations.

1.4 *(b) and (f)*

Two objects are created and three references are declared by the code. Objects are normally created by using the `new` operator. The declaration of a reference creates a variable regardless of whether a reference value is assigned to it or not.

1.5 *(d)*

An instance member is a field or an instance method. These members belong to all instances of the class. Members that are not explicitly declared `static` in a class declaration are instance members.

[1.6](#) *(c)*

An object communicates with another object by calling an instance method of the other object, passing and receiving any information that might be necessary.

[1.7](#) *(d) and (f)*

Given the declaration `class B extends A {...}`, we can conclude that class `B` extends class `A`, class `A` is the superclass of class `B`, class `B` is a subclass of class `A`, and class `B` inherits from class `A`, which means that objects of class `B` inherit the field `value1` from class `A`.

[1.8](#) *(d)*

The compiler supplied with the JDK is named `javac`. The names of the source files to be compiled are listed on the command line after the command `javac`. (c) will compile and execute the program, but will not create a class file.

[1.9](#) *(a)*

Java programs are executed by the Java Virtual Machine (JVM). In the JDK, the command `java` is used to start the execution by the JVM. The `java` command requires the name of a class that has a valid `main()` method. The JVM starts the program execution by calling the `main()` method of the given class. The exact name of the class should be specified, and not the name of the class file—that is, the `.class` extension in the class file name should not be specified. Since it is specified that the source file is compiled creating a class file, (c) would not work.

[1.10](#) *(a) and (d)*

The file with a single-file source-code program can contain more than one class declaration and the first class declaration must provide a valid `main()` method. Such a program cannot access previously compiled user-defined classes, only those in the standard library. It cannot consist of multiple files obviously, but program arguments can be supplied on the command line.

[1.11](#) *(e)*

(a) is incorrect because the JVM must be compatible with the Java Platform on which the program was developed.

(b) is incorrect because the JIT feature of the JVM translates bytecode to machine code.

(c) is incorrect because other languages, like Scala, also compile to bytecode and can be executed by the JVM.

(d) is incorrect because a Java program can only create objects, but destroying objects is at the discretion of the automatic garbage collector.

2 Basic Elements, Primitive Data Types, and Operators

2.1 *(e)*

Everything from the start sequence (`/*`) of a multiple-line comment to the first occurrence of the end sequence (`*/`) of a multiple-line comment is ignored by the compiler. Everything from the start sequence (`//`) of a single-line comment to the end of the line is ignored by the compiler. In (e), the multiple-line comment ends with the first occurrence of the end sequence (`*/`), leaving the second occurrence of the end sequence (`*/`) unmatched.

2.2 *(d)*

An assignment statement is an expression statement. The value of the expression statement is the value of the expression on the right-hand side. Since the assignment operator is right associative, the statement `a = b = c = 20` is evaluated as follows: `(a = (b = (c = 20)))` . This results in the value `20` being assigned to `c` , then the same value being assigned to `b` and finally to `a` . The program will compile and print `20` at runtime.

2.3 *(c)*

In an assignment statement, the reference value of the source reference is assigned to the destination reference. Assignment does not create a copy of the object denoted by the source reference. After the assignment, both references denote the same object—that is, they are aliases.

The variables `a`, `b`, and `c` are references of type `String`. The reference value of the `"cat"` object is first assigned to `a`, then to `b`, and later to `c`. Just before the print statement, `a` denotes `"dog"`, whereas both `b` and `c` denote `"cat"`. The program prints the string denoted by `c`—that is, `"cat"`.

2.4 *(a), (d), and (e)*

A binary expression with any floating-point operand will be evaluated using floating-point arithmetic. Expressions such as `2/3`, where both operands are integers, will use integer arithmetic and evaluate to an integer value. In (e), the result of `(0x10 * 1L)` is promoted to a floating-point value.

2.5 *(b)*

The `/` operator has higher precedence than the + operator. This means that the expression is evaluated as `((1/2) + (3/2) + 0.1)`. The associativity of the binary operators is from left to right, giving `(((1/2) + (3/2)) + 0.1)`. Integer division results in `((0 + 1) + 0.1)`, which evaluates to 1.1.

2.6 *(b)*

The expression evaluates to `-6`. The whole expression is evaluated as `(((-(-1)) -((3 * 10) / 5)) - 1)` according to the precedence and associativity rules.

2.7 *(d)*

The expression `++k + k++ + + k` is evaluated as `((++k) + (k++)) + (+k)` → ((2) + (2) + (3)), resulting in the value `7`.

2.8 *(d)*

The types `char` and `int` are both integral. A `char` value can be assigned to an `int` variable since the `int` type is wider than the `char` type and an implicit widening conversion will be done. An `int` type cannot be assigned to a `char` variable because the `char` type is narrower than the `int` type. The compiler will report an error about a possible loss of precision at (4).

2.9 *(a)*

First, the expression `++i` is evaluated, resulting in the value `2`. Now the variable `i` also has the value `2`. The target of the assignment is now determined to be the element `array[2]`. Evaluation of the right-hand expression, `--i`, results in the value `1`.

The variable `i` now has the value `1`. The value of the right-hand expression `1` is then assigned to the array element `array[2]`, resulting in the array contents to become `{4, 8, 1}`. The program computes and prints the sum of these values—that is, `13`.

2.10 *(c) and (e)*

The remainder operator is not limited to integral values, but can also be applied to floating-point operands. Short-circuit evaluation occurs with the conditional operators (`&&`, `||`). The operators `*`, `/`, and `%` have the same level of precedence. The data type `short` is a 16-bit signed two's complement integer, thus the range of values is from `-32768` to `+32767`, inclusive. `(+15)` is a legal expression using the unary `+` operator.

2.11 *(a), (c), and (e)*

The `!=` and `^` operators, when used on boolean operands, will return `true` if and only if one operand is `true`, and `false` otherwise. This means that `d` and `e` in the program will always be assigned the same value, given any combination of truth values in `a` and `b`. The program will, therefore, print `true` four times.

2.12 *(b)*

The element referenced by `a[i]` is determined based on the current value of `i`, which is `0`—that is, the element `a[0]`. The expression `i = 9` will evaluate to the value `9`, which will be assigned to the variable `i`. The value `9` is also assigned to the array element `a[0]`. After execution of the statement, the variable `i` will contain the value `9`, and the array `a` will contain the values `9` and `6`. The program will print `9 9 6` when run.

2.13 *(c) and (d)*

Note that the logical and conditional operators have lower precedence than the relational operators. Unlike the `&` and `|` operators, the `&&` and `||` operators short-circuit the evaluation of their operands if the result of the operation can be determined from the value of the first operand. The second operand of the `||` operator in the program is never evaluated because the value of `t` remains `true`. All the operands of the other operators are evaluated. Variable `i` ends up with the value `3`, which is the first digit printed, and `j` ends up with the value `1`, which is the second digit printed.

2.14 *(b)*

Both `||` and `&&` are short-circuit conditional operators. In the conditional expression `(x < y || ++z > 4)` of the first `if` statement, since the first operand `x < y` evaluates to `true`, the second operand `++z > 4` is not evaluated, as the conditional operator is `||`. The `if` condition is `true` and the `if` block is executed, printing `a123`.

In the conditional expression `(x < y || ++z > 4)` of the second `if` statement, since the first operand `x < y` evaluates to `true`, the second operand `++z > 4` is evaluated, as the conditional operator is `&&.` The second operand is `false` (`4 > 4`); therefore, the `if` condition is `false` and the `if` block is not executed.

2.15 *(c), (e), and (f)*

In (a), the third operand has the type `double`, which is not assignment compatible with the type `int` of the variable `result1`. Blocks are not legal operands in the conditional operator, as in (b). In (c), the last two operands result in wrapper objects with type `Integer` and `Double`, respectively, which are assignment compatible with the type `Number` of the variable `number`. The evaluation of the conditional expression results in the reference value of an `Integer` object with value `20` being assigned to the `number` variable. All three operands of the operator are mandatory, which is not the case in (d). In (e), the last two operands are of type `int`, and the evaluation of the conditional expression results in an `int` value (`21`), whose text representation is printed. In (f), the value of the second operand is boxed into a `Boolean`. The evaluation of the conditional expression results in a string literal (`"i not equal to j"`), which is printed. The `println()` method creates and prints a text representation of any object whose reference value is passed as a parameter.

2.16 *(d)*

The condition in the outer conditional expression is `false`. The condition in the nested conditional expression is `true`, resulting in the value of `m1` (i.e., `20`) being printed.

3 Declarations

3.1 *(c)*

The local variable of type `float` will remain uninitialized. Fields and static variables are initialized with a default value. An instance variable of type `int[]` is a reference variable that will be initialized with the `null` value. Local variables remain uninitialized unless explicitly initialized.

3.2 *(e)*

The program will compile. The compiler can figure out that the local variable `price` will always be initialized, since the value of the condition in the `if` statement is `true`. The two instance variables and the two static variables are all initialized to the respective default value of their type.

3.3 *(a) and (e)*

The first and the third pairs of methods will compile. The second pair of methods will fail to compile, since their method signatures do not differ. The compiler has no way of differentiating between the two methods. Note that the return type and the names of the parameters are not a part of the method signature. Both methods in the first pair are named `fly` and have a different number of parameters, thus overloading this method name. The methods in the last pair do not overload the method name `glide`, since only one method has that name. The method named `Glide` is distinct from the method named `glide`, as identifiers are case sensitive in Java.

3.4 *(b) and (e)*

A constructor can be declared `private`, but this means that this constructor can only be used within the class. Constructors need not initialize all the fields when a class is instantiated. A field will be assigned a default value if not explicitly initialized. A constructor is non-static, and as such it can directly access both the static and non-static members of the class.

3.5 *(c)*

A compile-time error will occur at (3), since the class does not have a constructor accepting a single argument of type `int`. The declaration at (1) declares a method, not a constructor, since it is declared as `void`. The method happens to have the same name as the class, but that is irrelevant. The class has the default constructor, since the class contains no constructor declarations. This constructor will be invoked to create a `MyClass` object at (2).

3.6 *(b)*

The keyword `this` can only be used in non-`static` code, as in non-`static` methods, constructors, and instance initializer blocks. Only one occurrence of each `static` variable of a class is created, when the class is loaded by the JVM. This occurrence is shared among all the objects of the class (and for that matter, by other clients). Local variables are only accessible within the local scope, regardless of whether the local scope is defined within a static context.

3.7 *(e)*

The `[]` notation can be placed both after the type name and after the variable name in an array declaration. Multidimensional arrays are created by constructing arrays that can contain references to other arrays. The expression `new int[4][]` will create an array of length 4, which can contain references to arrays of `int` values. The expression `new int[4][4]` will also create a two-dimensional array, but will in addition create four more one-dimensional arrays, each of length 4 and of the type `int[]`. References to each of these arrays are stored in the two-dimensional array. The expression `int[][4]` will not work, because the arrays for the dimensions must be created from left to right.

3.8 *(a), (c), and (d)*

The size of the array cannot be specified, as in (b) and (e). The size of the array is given implicitly by the initialization code. The size of the array is never specified in the declaration of an array reference. The size of an array is always associated with the array instance (on the right-hand side), not the array reference (on the left-hand side).

3.9 *(e)*

The array declaration is valid, and will declare and initialize an array of length 20 containing `int` values. All the values of the array are initialized to their default value of `0`. The `for(;;)` loop will print all the values in the array—that is, it will print `0` twenty times.

3.10 *(d)*

The program will print `0 false 0 null` when run. All the instance variables, and the array element, will be initialized to their default values. When concatenated with a string, the values are converted to their text representation. Notice that the `null` literal is converted to the string "`null`", rather than throwing a `NullPointerException`.

3.11 *(b)*

Evaluation of the actual parameter `i++` yields `0`, and increments `i` to `1` in the process. The value `0` is copied into the formal parameter `i` of the method `addTwo()` during method invocation. However, the formal parameter is local to the method, and changing its value does not affect the value in the actual parameter. The value of the variable `i` in the `main()` method remains `1`.

3.12 *(d)*

The variables `a` and `b` are local variables of type `int`. When these variables are passed as arguments to another method, the method receives copies of the primitive values in the variables. The actual variables are unaffected by operations performed on the copies of the primitive values within the called method. The variable `bArr` contains a reference value that denotes an array object containing primitive values. When the variable is passed as a parameter to another method, the method receives a copy of the reference value. Using this reference value, the method can manipulate the object that the reference value denotes. This allows the elements in the array object referenced by `bArr` to be accessed and modified in the method `inc2()`.

3.13 *(c)*

In (a) and (b), the arguments are encapsulated as elements in the implicitly created array that is passed to the method. In (c), the `int` array object itself is encapsulated as an element in the implicitly created array that is passed to the method. (a), (b), and (c) are fixed arity calls. Note that `int[]` is not a subtype of

`Object[]`. In (d), (e), and (f), the argument is a subtype of `Object[]`, and the argument itself is passed without the need for an implicitly created array—that is, these are fixed arity method calls. However, in (d) and (e), the compiler issues a warning that both fixed arity and variable arity method calls are feasible, but chooses fixed arity method calls.

3.14 *(b)*

Local variable type inference with `var` is not allowed in a multiple-declaration statement, as at (2).

3.15 *(d), (e), and (f)*

The restricted keyword `var` cannot be used as a return type or as the type of a formal parameter, ruling out (a), (b), and (c).

The signature of the method call `divide(int, int)` is assignment compatible with the method signatures `divide(int, int)`, `divide(int, double)`, and `divide(double, int)` in (d), (e), and (f), respectively. The `double` value of the expression in the `return` statement in the `divide()` method is assignment compatible with the `return` type `double` of the method headers in (d), (e), and (f).

4 Control Flow

4.1 *(d)*

The program will display the letter `b` when run. The second `if` statement is evaluated since the boolean expression of the first `if` statement is `true`. The `else` clause belongs to the second `if` statement. Since the boolean expression of the second `if` statement is `false`, the `if` block is skipped and the `else` clause is executed.

4.2 *(c)*

The `case` label value `2 * iLoc` is a constant expression whose value is `6`, the same as the `switch` expression. Fall-through results in the program output shown in (c).

4.3 *(c)*

(a) contains a `switch` statement. Note that there is no `break` statement associated with the first `case` label, thus execution falls through to the second `case` label and assigns the string `"Composite"` to the reference `result`, which is then printed.

(b) uses a `switch` expression to yield a result. However, it does not provide an exhaustive set of `case` labels and will fail to compile without the `default` label.

(c) uses the identifier `yield` as both a variable name and a contextual keyword in the `yield` statement. There is no fall-through, and the `switch` expression yields the string `"Prime"` which is printed.

(d) is mixing two different types of notations for the `switch` constructs: the arrow notation and the colon notation, which is not permitted.

4.4 *(a)*

The value `1` of the `price` variable matches the `case` constant 1 in the first `case` label, and in this case the discount is calculated by subtracting 1 from the value of `price`, which results in the value of `0`. This code uses a `switch` expression with the arrow notation, so no fall-through to the next `case` label can occur. Case labels do not need to be listed in any particular order. The `switch` expression is exhaustive, because the `case` labels and the `default` label cover the range of `int` values. Code will compile and when executed will yield the value `0`.

4.5 *(e)*

The loop body is executed twice and the program will print `3`. The first time the loop is executed, the variable `i` changes value from `1` to `2` and the variable `b` changes value from `false` to `true`. Then the loop condition is evaluated. Since `b` is `true`, the loop body is executed again. This time the variable `i` changes value from `2` to `3` and the variable `b` changes value from `true` to `false`. The loop condition is now evaluated again. Since `b` is now `false`, the loop terminates and the current value of `i` is printed.

4.6 *(b) and (e)*

Both the first and the second numbers printed will be `10`. Both the loop body and the update expression will be executed exactly 10 times. Each execution of the loop body will be directly followed by an execution of the update expression. Afterwards, the condition `j < 10` is evaluated to see whether the loop body should be executed again.

4.7 *(f)*

The code will compile without error, but will never terminate when run. All the sections in the `for` header are optional and can be omitted (but not the semicolons). An omitted loop condition is interpreted as being `true`. Thus a `for(;;)` loop with an omitted loop condition will never terminate, unless an appropriate control transfer statement is encountered in the loop body. The program will enter an infinite loop at (4).

4.8 *(a) and (d)*

"`i=1, j=0`" and "`i=2, j=1`" are part of the output. The variable `i` iterates through the values `0`, `1`, and `2` in the outer loop, while `j` toggles between the values `0` and `1` in the inner loop. If the values of `i` and `j` are equal, the printing of the values is skipped and the execution continues with the next iteration of the outer loop. The following can be deduced when the program is run: variables `i` and `j` are both `0` and the execution continues with the update expression of the outer loop. "`i=1, j=0`" is printed and the next iteration of the inner loop starts. Variables `i` and `j` are both `1` and the execution continues with the update expression of the outer loop. "`i=2, j=0`" is printed and the next iteration of the inner loop starts. "`i=2, j=1`" is printed, `j` is incremented, `j < 2` is `false`, and the inner loop ends. Variable `i` is incremented, `i < 3` is `false`, and the outer loop ends.

4.9 *(c) and (d)*

The element type of the array `nums` must be assignment compatible with the type of the loop variable (i.e., `int`). Only the element type in (c), `Integer`, can be automatically unboxed to an `int`. The element type in (d) is `int`.

4.10 *(d) and (e)*

In the header of a `for(:)` loop, we can only declare one local variable. This rules out (a) and (b), as they specify two local variables. Also, the array expression in (a), (b), and (c) is not valid. Only (d) and (e) specify a legal `for(:)` header.

4.11 *(a), (b), and (c)*

Changing the value of the *`variable`* does not affect the data structure being iterated over. The `for(:)` loop cannot run backwards. We cannot iterate over several data structures simultaneously in a `for(:)` loop, as the syntax does not allow it.

5 Object-Oriented Programming

5.1 *(a) and (c)*

`Bar` is a subclass of `Foo` that overrides the method `g()`. The statement `a.j = 5` is not legal, since the member `j` in the class `Bar` cannot be accessed through a `Foo` reference. The statement `b.i = 3` is not legal either, since the `private` member `i` cannot be accessed from outside of the class `Foo`.

5.2 *(g)*

It is not possible to invoke the `doIt()` method in `A` from an instance method in class `C`. The method in `C` needs to call a method in a superclass two levels up in the inheritance hierarchy. The `super.super.doIt()` strategy will not work, since `super` is a keyword and cannot be used as an ordinary reference, nor accessed like a field. If the member to be accessed had been a field or a `static` method, the solution would be to cast the `this` reference to the class of the field and use the resulting reference to access the field, as illustrated in (f). Field access is determined by the declared type of the reference, whereas the instance method to execute is determined by the actual type of the object denoted by the reference at runtime.

5.3 *(e)*

The code will compile without errors. None of the calls to a `max()` method are ambiguous. When the program is run, the `main()` method will call the `max()` method on the `C` object referred to by the reference `b` with the parameters `13` and `29`. This method will call the `max()` method in `B` with the parameters `23` and `39`. The `max()` method in `B` will in turn call the `max()` method in `A` with the parameters `39` and `23`. The `max()` method in `A` will return `39` to the `max()` method in `B`. The `max()` method in `B` will return `29` to the `max()` method in `C`. The `max()` method in `C` will return `29` to the `main()` method.

5.4 *(g)*

In the class `Car`, the static method `getModelName()` hides the `static` method of the same name in the superclass `Vehicle`. In the class `Car`, the instance method `getRegNo()` overrides the instance method of the same name in the superclass `Vehicle`. The declared type of the reference determines the method to execute when a `static` method is called, but the actual type of the object at runtime determines the method to execute when an overridden method is called.

5.5 *(e)*

The class `MySuper` does not have a no-argument constructor. This means that constructors in subclasses must explicitly call the superclass constructor and provide the required parameters. The supplied constructor accomplishes this by calling `super(num)` in its first statement. Additional constructors can accomplish this either by calling the superclass constructor directly using the `super()` call, or by calling another constructor in the same class using the `this()` call which in turn calls the superclass constructor. (a) and (b) are not valid, since they do not call the superclass constructor explicitly. (d) fails, since the `super()` call must always be the first statement in the constructor body. (f) fails, since the `super()` and `this()` calls cannot be combined.

5.6 *(b)*

In a subclass without any declared constructors, the implicit default constructor will call `super()`. Use of the `super()` and `this()` statements is not mandatory as long as the superclass has a no-argument constructor. If neither `super()` nor `this()` is declared as the first statement in the body of a constructor, then the default `super()` will implicitly be the first statement. A constructor body cannot have both a `super()` and a `this()` statement. Calling `super()` will not always work, since a superclass might not have a no-argument constructor.

5.7 *(d)*

The program will print `12` followed by `Test`. When the `main()` method is executed, it will create a new instance of `B` by passing `"Test"` as an argument. This results in a call to the constructor of `B` that has one `String` parameter. The constructor does not explicitly call any superclass constructor nor any overloaded constructor in `B` using a `this()` call, but instead the no-argument constructor of the superclass `A` is called implicitly. The no-argument constructor of `A` calls the constructor in `A` that has two `String` parameters, passing it the argument list `("1"`, `"2")`. This constructor calls the constructor with one `String` parameter, passing the argument `"12"`. This constructor prints the argument, after implicitly invoking the no-argument constructor of the superclass `Object`. Now the execution of all the constructors in `A` is completed, and execution continues in the constructor of `B`. This constructor now prints the original argument `"Test"` and returns to the `main()` method.

5.8 *(c)*

Any non-`final` class can be declared `abstract`. A class cannot be instantiated if the class is declared `abstract`. The declaration of an `abstract` method cannot provide an implementation. The declaration of a non-`abstract` method must provide an implementation. If any method in a class is declared `abstract`, then the class must be declared `abstract`, so (a) is invalid. The declaration in (b) is not valid, since it omits the keyword `abstract` in the method declaration. The declaration in (d) is not valid, since it omits the keyword `class`. In (e), the return type of the method is missing.

5.9 *(b)*

Since the method is `abstract`, it cannot be inserted at (1) because class `Animal` is not `abstract`—thus ruling out (a) and (c). Class `Cat` is `abstract`, and the method can be inserted at (2)—thus ruling out (d).

5.10 *(d)*

We cannot create an object of an `abstract` class with the `new` operator.

5.11 *(d)*

An instance of `Bacteria` can be assigned to the `org` variable at (2), since a supertype reference can refer to a subtype object. There is no `@Overload` annotation.

5.12 *(a) and (b)*

The `extends` clause is used to specify that a class extends another class. A subclass can be declared `abstract` regardless of whether the superclass was declared `abstract`. Private, overridden, and hidden members from the superclass are not inherited by the subclass. A class cannot be declared both `abstract` and `final`, since an `abstract` class needs to be extended to be useful, and a `final` class cannot be extended. The accessibility of the class is not limited by the accessibility of its members. A class with all members declared `private` can still be declared `public`.

5.13 *(c)*

Only a `final` class cannot be extended, as in (c). (d) will fail to compile. A class cannot be declared both `final` and `abstract`, as in (d).

5.14 *(c)*

Line (3), `void k() { i++; }`, can be re-inserted without introducing errors. Reinserting line (1) will cause the compilation to fail, since `MyOtherClass` will try to override a `final` method. Re-inserting line (2) will fail, since `MyOtherClass` will no longer have the no-argument constructor. The `main()` method needs to call the no-argument constructor. Re-inserting line (3) will work without any problems, but reinserting line (4) will fail, since the method will try to access a `private` member of the superclass.

5.15 *(a) and (c)*

Abstract classes can declare both `final` methods and non-`abstract` methods. Non-`abstract` classes cannot, however, contain `abstract` methods. Nor can `abstract` classes be `final`. Only interfaces can declare `default` methods.

5.16 *(d)*

There is no problem compiling the code.

5.17 *(a)*

A `final` class cannot have `abstract` methods, as a `final` class is a concrete class, providing implementation for all methods in the class.

[5.18](#) *(b) and (g)*

The keywords `protected` and `final` cannot be applied to interface methods. The keyword `public` is implied, but can be specified for abstract and default interface methods. The keywords `private`, `default`, `abstract`, and `static` can be specified for `private`, `default`, `abstract`, and `static` methods, respectively. The keywords `private`, `default`, and `static` are required for `private`, `default`, and `static` methods, respectively, but the keyword `abstract` is optional as an `abstract` method is understood to be implicitly `abstract`.

[5.19](#) *(e)*

The `static` method `printSlogan()` is *not* inherited by the class `Firm`. It can only be invoked by using a static reference—that is, the name of the interface in which it is declared, regardless of whether the call is in a static or a non-static context.

[5.20](#) *(c)*

The instance method at (3) overrides the `default` method at (1). The `static` method at (2) is not inherited by the class `RaceA`. The instance method at (4) does not override the `static` method at (2).

The method to invoke by the call at (5) is determined at runtime by the object type of the reference, which in this case is `Athlete`, resulting in the method at (3) being invoked. Similarly, the call at (6) will invoke the instance method at (4).

[5.21](#) *(d)*

The code will compile without errors. The class `MyClass` declares that it implements the interfaces `Interface1` and `Interface2`. Since the class is declared `abstract`, it does not need to implement all `abstract` method declarations defined in these interfaces. Any non-`abstract` subclasses of `MyClass` must provide the missing method implementations. The two interfaces share a common `abstract` method declaration, `void g()`. `MyClass` provides an implementation for this `abstract` method declaration that satisfies both `Interface1` and `Interface2`. Both interfaces provide declarations of constants named `VAL_B`. This can lead to ambiguity when referring to `VAL_B` by its simple name from `MyClass`. The ambiguity can be resolved by using the qualified names `Interface1.VAL_B` and `Interface2.VAL_B`. However, there are no problems with the code as it stands.

5.22 *(b)*

The compiler will allow the statement, as the cast is from the supertype (`Super`) to the subtype (`Sub`). However, if at runtime the reference `x` does not denote an object of the type `Sub`, a `ClassCastException` will be thrown.

5.23 *(b)*

The expression `(o instanceof B)` will return `true` if the object referred to by `o` is of type `B` or a subtype of `B`. The expression `(!(o instanceof C))` will return `true` unless the object referred to by `o` is of type `C` or a subtype of `C`. Thus the expression `(o instanceof B) && (!(o instanceof C))` will only return `true` if the object is of type `B` or a subtype of `B` that is not `C` or a subtype of `C`. Given objects of the classes `A`, `B`, and `C`, this expression will only return `true` for objects of class `B`.

5.24 *(d)*

The program will print all the letters `I`, `J`, `C`, and `D` at runtime. The object referred to by the reference `x` is of class `D`. Class `D` extends class `C` and implements `J`, and class `C` implements interface `I`. This makes `I`, `J`, and `C` supertypes of class `D`. The reference value of an object of class `D` can be assigned to any reference of its supertypes and is, therefore, an `instanceof` these types.

5.25 *(c)*

The calls to the `compute()` method in the method declarations at (2) and at (3) are to the `compute()` method declaration at (1), as the argument is always an `int[]`.

The method call at (4) calls the method at (2). The signature of the call at (4) is

```
compute(int[], int[])
```

which matches the signature of the method at (2). No implicit array is created. The method call at (5) calls the method at (1). An implicit array of `int` is created to store the argument values.

The method calls at (6) and (7) call the method at (3). Note the type of the variable arity parameter at (3): an `int[][]`. The signature of the calls at (6) and (7) is

```
compute(int[], int[][])
```

which matches the signature of the method at (3). No implicit array is created.

5.26 *(f)*

The `instanceof` pattern match operator can introduce a pattern variable in certain boolean expressions. In the conditional of the `if` statement, both operands of the short-circuit `&&` operator must be `true` for the pattern variable `s` to be introduced in the `if` block—the scope of variable `s` is then the `if` block, and `s` is not accessible in the `else` block. The variable `s` is thus out of scope in the `else` block, and the code will not compile.

5.27 *(d)*

For the `instanceof` pattern match operator, the pattern type (i.e., the type specified for the right operand) must be a subtype of the expression type (i.e., the type of the left operand). This is not the case in (a), (b), or (e). In (a) and (b), both the pattern type and the expression type are `Integer`, and in (e), the pattern type `Number` is a supertype of the expression type `Integer`. Thus (a), (b), and (e) will result in a compile-time error.

In (c), the expression type `Integer` is incompatible with the pattern type `String` for comparing types, as one cannot be cast to the other, thus resulting in a compile-time error.

In (d), the pattern type `Integer` is a subtype of the expression type `Number` and will compile without any problem.

5.28 *(a) and (c)*

An `instanceof` pattern match operator returns `false` if the reference is `null`; therefore, it will not throw a `NullPointerException`. A pattern variable is only introduced when the `instanceof` pattern match operator returns `true`.

5.29 *(e)*

The program will print `2` when `System.out.println(ref2.f())` is executed. The object referenced by `ref2` is of class `C`, but the reference is of type `B`. Since `B` contains a method `f()`, the method call will be allowed at compile time. During execution, it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method in `C` to be executed.

5.30 *(c)*

The program will print `1` when run. The `f()` methods in `A` and `B` are `private`, and are not accessible by the subclasses. Because of this, the subclasses cannot overload or override these methods, but simply define new methods with the same signature. The object being called is of class `C`. The reference used to access the object is of type `B`. Since `B` contains a method `g()`, the method call will be allowed at compile time. During execution, it is determined that the object is of class `C`, and dynamic method lookup will cause the overriding method `g()` in `B` to be executed. This method calls a method named `f`. It can be determined during compilation that this can only refer to the `f()` method in `B`, since the method is `private` and cannot be overridden. This method returns the value `1`, which is printed.

5.31 *(b), (c), and (d)*

The code as it stands will compile. The use of inheritance in this code defines a `Planet` *is-a* `Star` relationship. The code will fail if the name of the field `starName` is changed in the `Star` class, since the subclass `Planet` tries to access it using the name `starName`. An instance of `Planet` is not an instance of `HeavenlyBody`. Neither `Planet` nor `Star` implements `HeavenlyBody`.

5.32 *(d)*

An enum type can be run as a standalone application, if it provides the appropriate `main()` method. The constants need not be qualified when referenced inside the enum type declaration. The constants *are* `static` members. The `toString()` method always returns the name of the constant, unless it is overridden.

5.33 *(b)*

(1), (2), and (3) define *constant-specific class bodies* that override the `toString()` method. For constants that do not override the `toString()` method, the name of the constant is returned.

5.34 *(c)*

An enum type cannot be instantiated to create more objects than those already created implicitly for its constants. All enum types override the `equals()` method from the `Object` class. The `equals()` method of an enum type compares its constants for equality according to reference equality (the same as with the `==` operator) based on their ordinal values. This `equals()` method is `final`.

5.35 *(d) and (e)*

In (a), the compiler recognizes a non-canonical constructor with no parameters in the record class definition. The first statement in such a non-canonical constructor must be an explicit invocation of the canonical constructor using the `this()` expression. For example, the following constructor declaration will compile, but it will not give the desired result.

```
public Product() {
  this(0, "No name", 0.00);
}
```

However, specifying the required parameters in the constructor header will result in the normal canonical constructor that will compile, and the code will print the right result:

Click here to view code image

```
public Product(int id, String name, double price) {
...
}
```

(b) does not compile because the parameter names in the normal canonical constructor do not match the ones defined in the header of the record class.

In (c), the compiler recognizes a record class that has no component fields. The constructor declared is a non-canonical constructor that must have an explicit invocation of the no-argument implicit canonical constructor using the `this()` expression. For example, the following constructor declaration will compile, but will not give the desired result.

Click here to view code image

```
public Product(int id, String name, double price) {
  this();
}
```

However, specifying the field components in the record class header will make the code compile and give the right result:

Click here to view code image

```
public record Product(int id, String name, double price) {
...
}
```

(d) correctly initializes a record using the compact constructor. The name will be stored in uppercase.

(e) correctly initializes a record using the implicit canonical constructor. The record class overrides the method `toString()` that returns the `name` field value represented as an uppercase `String`.

(f) correctly initializes a record using the implicit canonical record constructor, but its overridden `toString()` method accesses the fields directly, without converting the `name` field to uppercase. It does not invoke the `name()` method.

5.36 *(d)*

The compiler automatically generates an implementation of the `equals()` method for a record class, if one is not provided. The `equals()` method added by the compiler will compare all component fields of the record class. This means that the `equals()` method will return `true`, but the equality operator `==` will return `false`, as the two records that are created are distinct objects that have the same state.

5.37 *(c)*

All component fields defined by a record class are immutable. A record class can only declare static fields in addition to the component fields specified in its header. The compiler automatically generates the get methods for the component fields of the record class, but not the set methods, since such fields are immutable. Record classes implicitly extend the `java.lang.Record` class. Record classes cannot have an explicit `extends` clause.

5.38 *(c)*

Sealed classes can be `abstract`. In fact, this is often the case, as the `abstract sealed` class is intended to be extended by its permitted subclasses. A `non-sealed` class can also be `abstract` and can be freely extended. However, a `sealed` class can only be extended by its permitted subclasses. A class that extends a `sealed` class must be either `final`, `sealed`, or `non-sealed`.

5.39 *(c)*

In the code, subtypes `Y` and `Z` can be interfaces or classes that can either extend or implement the `sealed` interface `X`. A class or an interface that is marked `sealed` must be defined with the `permits` clause that specifies its permitted subtypes, unless the permitted subtypes are specified in the same compilation unit. Since the classes and interfaces are all `public`, each is defined in its own compilation unit.

In (a), interface `Z` is marked `sealed`, but does not provide the `permits` clause or its permitted subtypes in the same compilation unit.

A permitted subtype of a `sealed` supertype must be explicitly marked as either `final`, `non-sealed`, or `sealed`. In (b), interface `Z` is not marked with any of these markers, so it will not compile. (d) has the exact same problem with class `Y`.

In (c), interface `Z` is correctly marked as `sealed`, with the appropriate `permits` clause, and class `Y` correctly implements both its `sealed` superinterfaces

`X` and `Z`.

6 Access Control

6.1 *(a) and (c)*

Bytecode of all reference type declarations in the file is placed in the designated package, and all reference type declarations in the file can access the imported types.

6.2 *(e)*

Both classes are in the same package `app`, so the first `2` import statements are unnecessary. The package `java.lang` is always imported in all compilation units, so the next two import statements are unnecessary. The last static import statement is necessary to access the static variable `frame` in the `Window` class by its simple name.

6.3 *(b), (c), (d), and (e)*

In (a), the import statement imports types from the `mainpkg` package, but `Window` is not one of them.

In (b), the import statement imports types from the `mainpkg.subpkg1` package, and `Window` is one of them.

In (c), the import statement imports types from the `mainpkg.subpkg2` package, and `Window` is one of them.

In (d), the first import statement is a type-import-on-demand statement and the second import statement is a single-type-import statement. Both import the type `Window`. The second one overrides the first one.

In (e), the first import statement is a single-type-import statement and the second import statement is a type-import-on-demand statement. Both import the type `Window`. The first one overrides the second one.

In (f), both import statements import the type `Window`, making the import ambiguous.

In (g), both single-type-import statements import the type `Window`. The second import statement causes a conflict with the first one.

6.4 *(c) and (e)*

The name of the class must be fully qualified. A parameter list after the method name is not permitted. (c) illustrates single static import, and (e) illustrates static import on demand.

6.5 *(b) and (d)*

In (a) and (c), class `A` cannot be found. In (e) and (f), class `B` cannot be found—there is no package under the current directory `/top/wrk/pkg` to search for class `B`. Note that specifying `pkg` in the classpath in (d) is superfluous. The *parent* directory of the package must be specified—that is, the *location* of the package.

6.6 *(d) and (e)*

Static field `y` in class `a.b.X` is accessed in the method `xyz()` of class `a.b.c.Z`. Static import allows static members from reference type declarations in other packages to be accessed by their simple names.

This rules out (a) as it is a type-import-on-demand statement for all reference type declarations in package `a.b`, and also (b) as it is a type-import-on-demand statement from class `a.b.X`. (a) imports class `X`, but (b) does not import any type, as class `X` does not declare any non-static inner class members.

(d) is a static import-on-demand statement, meaning it imports all static members of the class `a.b.X`, including `y` which can be accessed by its simple name. (e) is a single-static-import statement, meaning only the designated static member `y` from class `a.b.X` is imported and can be accessed by its simple name.

6.7 *(a) and (d)*

The class `Farm` in package `habitat` accesses classes `Cat` and `Cow` by their simple names from package `life.animals`. (a) is a type-import-on-demand of all reference type declarations from package `life.animals`, including `Cat` and `Cow`. (b) and (c) are ruled out as these are static imports. (d) imports the classes `Cat` and `Cow` individually.

6.8 *(d)*

Packages are typically mapped to directories in a file system. A subpackage is an autonomous package that just happens to map to a subdirectory of a directory that represents some other package. There is no relationship between a package and its subpackages. Each package is treated independently, regardless of whether it appears to be implemented as a subdirectory, ruling out (a) and (b). (c) is incorrect because reference types and static members of types in other packages can be accessed by their fully qualified names, rather than using import statements.

Import statements are not present in the compiled code at all, as type names are always replaced with fully qualified names by the compiler.

6.9 *(b) and (e)*

If no access modifier (`public`, `protected`, or `private`) is given in the member declaration of a class, the member is only accessible by classes in the same package.

A subclass does not have access to members with package accessibility declared in a superclass, unless they are in the same package.

Local variables cannot be declared `static` or have an access modifier.

6.10 *(b)*

Outside the package, the member `j` is accessible to any class, whereas the member `k` is only accessible to subclasses of `MyClass`.

The field `i` has package access, and is only accessible by classes inside the package. The field `j` has public access, and is accessible from anywhere. The field `k` has protected access, and is accessible from any class inside the package and from subclasses anywhere. The field `l` has private access, and is only accessible within the class itself.

6.11 *(b)*

A private member is only accessible in the class in which it is declared. If no access modifier has been specified for a member, the member has package accessibility. The keyword `default` is not an access modifier. A member with package access is only accessible from classes in the same package. Subclasses in other packages cannot access a member with package accessibility.

6.12 *(d)*

A class that is declared as `final` cannot be extended. Making a class `final` is not enough to prevent its state from being modified. A static modifier can be applied to inner classes, but this is not relevant to the question of immutability. A field within an immutable object can refer to a mutable object, which means that members of an immutable object are not automatically immutable.

6.13 *(a)*

In (a), marking the field `name private` means it can only be accessible in the class. It can only be initialized once by the constructor when the object is created, and removing the `setName()` method means the value of `private` field `name` cannot be changed. The state of the object is thus immutable.

In (b), the assignment in the `setName()` method will not compile as it changes the value of the `final` field `name` which has already been initialized in the constructor. In (c), the assignment in the constructor will not compile as it changes the value of the `final` field `name` which has already been initialized in its declaration.

7 Exception Handling

7.1 *(d)*

The program will only print `1`, `4`, and `5`, in that order. The expression `5/k` will throw an `ArithmeticException`, since `k` equals 0. Control is transferred to the first `catch` clause, since it is the first `catch` clause that can handle the arithmetic exceptions. This exception handler simply prints `1`. The exception has now been caught and normal execution can resume. Before leaving the `try` statement, the `finally` clause is executed. This `finally` clause prints `4`. The last statement of the `main()` method prints `5`.

7.2 *(b) and (e)*

If run with no program arguments, the program will print `The end`. If run with one program argument, the program will print the specified argument followed by `The end`. The `finally` clause will always be executed, no matter how control leaves the `try` block.

7.3 *(c) and (d)*

Normal execution will only resume if the exception is caught by the method. The uncaught exception will propagate up the JVM stack until some method handles it. An overriding method need only declare that it can throw a subset of the checked exceptions the overridden method can throw. The `main()` method can declare that it throws checked exceptions just like any other method. The `finally` clause will always be executed, no matter how control leaves the `try` block.

7.4 *(b)*

The only thing that is wrong with the code is the ordering of the `catch` and `finally` clauses. If present, the `finally` clause must always appear last in a `try`-`catch-finally` construct. Note that since `B` is a subclass of `A`, catching `A` is sufficient to catch exceptions of type `B`.

7.5 *(b)*

An invocation of the `average()` method throws an `ArithmeticException`, which is then caught in the `main()` method. The catch block prints `"error"`. This means that the execution of the `average()` method is stopped, and the method does not return any value, leaving the local variable `value` still initialized to `1`, which is printed.

7.6 *(e)*

A `null` value is passed as an argument to the `reaction()` method, resulting in a `PlayerException` being thrown, containing the `"Invalid action"` message. This exception is then caught in the `main()` method, where its error message is assigned to the local variable `message` in the `catch` block. As this exception was successfully handled, normal execution resumes. The print statement prints the error message `"Invalid action"`.

7.7 *(c)*

As a `null` value is passed to the `readFile()` method, it throws a `FileNotFound-Exception`, which is a subclass of `IOException`. This exception is caught by the corresponding `catch` block in the `main()` method, printing `"IO error: invalid file name"`. Upon resumption of normal execution, the `finally` block prints `" finally"`, followed by the last print statement printing `" the end"`.

7.8 *(g)*

The `readFile()` method executes normally, which means that no `catch` block is executed in the `main()` method. The `finally` block prints `"finally"` and the last print statement prints `" the end"`.

7.9 *(d)*

A `null` value is passed to the `readFile()` method which then throws an unchecked `NullPointerException`, which is a subclass of `RuntimeException`. It is not required to explicitly specify unchecked exceptions in the `throws` clause or to handle them. The `NullPointerException` is propagated to the invoking method `main()`, where it is caught by the `catch` block that catches an `Exception`, since `RuntimeException` is a subclass of `Exception`. The `catch` block prints `"Other error: invalid file name"`. Although this `catch` block contains a `return` statement, the `finally` block is executed first, printing `" finally"`, before returning from the `main()` method. Thus the last print statement in the `main()` method is not executed.

7.10 *(b), (c), and (e)*

(b), (c), and (e) correspond to (2), (3), and (5). `FileNotFoundException` is thrown by the constructor call `FileReader(filename)`. The `close()` method of the `Buffered-Reader` throws an `IOException`. Either the `try`-with-resources statement must catch it or the exception must be specified in the `throws` clause of the method—the catch-or-declare rule. (1), (4), and (6) do not fulfill this criteria. Also, the resource variables are `final` and cannot be assigned to in the body of the `try`-with-resources statement, ruling out (7). At (5), the resource declaration statements are valid.

7.11 *(h)*

The top-level `try` block in the method `justDoIt()` throws an `IOException`. The nested `try` block in the `finally` clause throws an `EOFException` that is caught and associated as a suppressed exception with the `IOException`. It is the `IOException` that is propagated. The `IOException` is caught in the `catch` clause in the `main()` method and its information is printed, including its suppressed exception `EOFException`. The supertype exception references are used polymorphically to handle objects of subtype exceptions.

7.12 *(f)*

In (a), the program does not compile because the checked `Exception` thrown in the `close()` method does not comply with the catch-or-declare rule.

In (b), although the `close()` method will abide by the catch-or-declare rule, the `main()` method does not.

In (c), adding `throws Exception` clause only to the `main()` method does not change the fact that the `close()` method does not abide by the catch-or-declare rule.

In (d), both methods abide by the catch-or-declare rule. When run, the program will throw an `Exception` that is not caught.

In (e), adding `catch (Exception e) {}` clause to the `try` statement in the `main()` method does not change the fact that the `close()` method does not abide by the catch-or-declare rule.

In (f), the `close()` method will abide by the catch-or-declare rule, and the `main()` method will catch and handle the exception thrown at runtime.

7.13 *(a), (b), and (c)*

In (a), the exception parameter `e` is implicitly `final` and cannot be reassigned in the multi-`catch` clause.

In (b), in the two assignments to the exception parameter `e`, objects of the superclass `IOException` cannot be assigned to references of subtypes `EOFException` and `FileNotFoundException`.

In (c), in the assignment to the exception parameter `e` of type `Exception`, an object of the subtype `IOException` is assigned to `e`, but an exception of type `Exception` is thrown in the `catch` clause. This exception is not covered by the subtype `IOException` specified in the `throws` clause. In other words, `Exception` thrown in the `catch` clause is not handled.

In (d), the compiler can infer that only `FileNotFoundException` can be thrown in the `try` statement. Such an exception can only be thrown in the `catch` clause, as the parameter `e` of type `Exception` can be inferred to be effectively `final`, and can thus only refer to a `FileNotFoundException`. This exception is covered by the `throws` clause.

In (e), the compiler can infer that only `FileNotFoundException` can be thrown in the `try` statement. This exception is caught by parameter `e` of the superclass `IOException`. `IOException` is covered by the `throws` clause that specifies its supertype `Exception`.

7.14 *(a)*

In this code example, the `Resource` object is used in a `try`-with-resources statement. Its `action()` method will print `"action "` and it will be closed by the implicit `finally` block by invoking the `close()` method that prints `"closure "`. There are no exceptions thrown. The last print statement prints `" the end"`.

7.15 *(b)*

The `Resource` object is used in the `try`-with-resources statement, which means it will be closed by the implicit `finally` block invoking the `close()` method after the execution of the `try` block.

There are two exceptions thrown in the code: The first is an `IOException` that is thrown by the `action()` method, and the second is thrown by the `close()` method of the `Resource` class. The `IOException` is then caught in the `main()` method. However, notice that the `IOException` handler does not attempt to retrieve and print information about suppressed exceptions thrown by the implicit `finally` block of the `try`-with-resources statement. The `catch` block prints `"IO action error "`. Once the exception is handled, execution of the rest of the method `main()` resumes. The last print statement prints `" the end"`.

7.16 *(b)*

There is no reason why explicit and implicit `finally` blocks cannot coexist. If an explicit `finally` block is added after the `try`-with-resources statement, its code is executed after the implicit `finally` block.

8 Selected API Classes

8.1 *(e)*

Neither the `hashCode()` method nor the `equals()` method is declared `final` in the `Object` class, and it cannot be guaranteed that implementations of these methods will differentiate between *all* objects. All arrays are genuine objects and inherit from the `Object` class, including the `clone()` method.

8.2 *(b)*

Values in the range –128 to +127, inclusive, are boxed in `Integer` objects and cached by the method `Integer.valueOf()`.

8.3 *(c)*

There is a minor performance penalty associated with the conversion of a primitive value to a wrapper object and vice versa. Wrapper references can be assigned the `null` value, but they cannot be assigned to a variable of a primitive type. An attempt to convert an uninitialized wrapper reference to a primitive value will result in a `NullPointerException`. However, if the reference is a local variable then the code will not compile.

8.4 *(b)*

`Integer` objects with a value between –128 and +127 are interned. Therefore, two references that reference the same interned `Integer` object will return `true` when compared with the `==` operator—that is, they are aliases. The reference `i1` is assigned the reference value of a new `Integer` object with value `10`. This `Integer` object is interned. The reference `i2` is assigned the reference value of this interned `Integer` object, instead of creating a new `Integer` object. The expression `i1 == i2` is thus `true`, resulting in `A` being printed. The expression `i1 == i3` is also `true`, since the `Integer` object referenced by `i1` is unboxed to the `int` value `10` which is also the value in `i3`, resulting in `B` being printed.

However, values boxed by the references `x1` and `x2` are greater than 127, and therefore these references refer to two different `Integer` objects which are not interned. The expression `x1 == x2` returns the value `false`. The expression `x1 == x3` returns `true`, since the `Integer` object referenced by `x1` is unboxed to the `int` value `1000` which is also the value in `x3`, resulting in `D` being printed.

8.5 *(d)*

The expression `str.substring(2,5)` will extract the substring `"kap"`. The method extracts the characters from index 2 to index 4, inclusive.

8.6 *(d)*

The program will print `str3str1` when run. The `concat()` method will create and return a new `String` object, which is the concatenation of the current `String` object and the `String` object passed as an argument. The expression statement `str1.concat(str2)` creates a new `String` object, but its reference value is not stored after the expression is evaluated. Therefore, this `String` object gets discarded.

8.7 *(d)*

The constant expressions `"ab" + "12"` and `"ab" + 12` will, at compile time, be evaluated to the string-valued constant `"ab12"`. Both variables `s` and `t` are assigned a reference to the same interned `String` object containing `"ab12"`. The variable `u` is assigned a new `String` object, created by using the `new` operator.

8.8 *(b)*

The reference value in the reference `str1` never changes and it refers to the string literal `"lower"` all the time. The calls to `toUpperCase()` and `replace()` return a new `String` object whose reference value is ignored.

[8.9](#) *(d)*

The call to the `putO()` method does not change the `String` object referred to by the `s1` reference in the `main()` method. The reference value returned by the call to the `concat()` method is ignored.

[8.10](#) *(b)*

The reference value in the reference `str1` never changes and it refers to the string literal `"lower"` all the time. The calls to `toUpperCase()` and `replace()` return a new `String` object whose reference value is ignored.

[8.11](#) *(b)*

The `substring()` method returns the characters from the start index inclusive to the end index exclusive. The start index is returned by the `indexOf(' ')` method call, which is the first occurrence of a space character `' '` within the string, namely index 4. The expression `s.indexOf(' ', s.indexOf(' ') + 1)` finds the next occurrence of the space character `' '`, where the search starts after the first occurrence of the space character (`' '`), returning the index 7. As 1 is added to this index, the end index passed to the `substring()` method is 8. The resulting substring is from index 4 inclusive to index 8 exclusive—that is, `" is "`. The `strip()` method removes both leading and trailing whitespace from this string, resulting in the string `"is"`. To this string, the character `'-'` is concatenated at either end.

[8.12](#) *(a)*

This text block does not have any incidental whitespace because the last line has no leading whitespace before the closing delimiter of the text block. The `while` loop splits the text block into individual lines, extracting a substring from the start to the line terminator (`\n`) of each line. The length of each line does not include the line terminator. The lengths are 3, 5, and 3, as no incidental whitespace is removed. The length of each line is then printed.

[8.13](#) *(d)*

In (a) and (b), the content of the text block does not start after the line terminator of the opening delimiter (`"""`).

In (c), the text block does not end with the closing delimiter (`"""`), but with four double quotes. Note that there is no requirement that double quotes should be balanced in a text block, and can be specified with or without escaping.

In (d), the text block ends correctly, as it uses the `\"` escape character for the double quote that should be part of the text block, allowing it to be distinguished from the closing delimiter. However, the last line of the block will not end with a line terminator. The resulting string literal is `"\"a\"\"b\""`. When printed, the output will be a single line containing the characters `"a""b"`.

(e) is syntactically correct because the text block is correctly terminated. However, in this case the closing delimiter is on a line on its own, resulting in the last line of the text block content to end with a line terminator. The resulting string literal is `"\"a\"\"b\"\n"`. When printed, the output will be a line containing the characters `"a""b"` followed by a newline. (f) is incorrect because the last `\"` escape character results in the subsequent two double quotes also to be escaped, resulting in no closing delimiter being found— that is, `\"""` results in `\"\"\"`.

8.14 *(a) and (e)*

The content of a text block starts on a new line of text immediately after the line that contains the opening delimiter, and ends just before the closing delimiter. This makes (a) correct, but not (b).

A text block is not a subtype of the `String` class, as the `String` class is final, and the type of a text block is `String`.

Although trailing whitespace is removed from the end of each line in the text block, only incidental whitespace is removed from the start of each line in the text block.

8.15 *(a)*

The code will fail to compile, since the expression `(s == sb)` is illegal. It compares references of two classes that are not related. Also, the `StringBuffer` class does not override the `equals()` method from the `Object` class, but inherits it.

8.16 *(e)*

The program will compile without errors and will print `Have a` when run. The contents of the string buffer are truncated to six characters by the method call `sb.setLength(6)`.

8.17 *(c)*

The `trimtoSize()` only changes the capacity to match the length of the string builder. It does not the change the length of the string builder. The methods `append()`, `reverse()`, and `setLength()` change the string builder successively by appending `"!"` (`" 1234 !"`), reversing the string builder (`"! 4321 "`), and setting the length to 5 (`"! 43"`). The print statement prints `|! 43|`.

8.18 *(b)*

The references `sb1` and `sb2` are not aliases. The `StringBuilder` class does not override the `equals()` method so the result will be the same as with the `==` operator. The correct answer is (b).

8.19 *(a)*

The `StringBuilder` class does not override the `hashCode()` method, but the `String` class does. The references `s1` and `s2` refer to a `String` object and a `StringBuilder` object, respectively. The hash values of these objects are computed by the `hashCode()` method in the `String` and the `Object` class, respectively—giving different results. The references `s1` and `s3` refer to two different `String` objects that are equal, hence they have the same hash value.

8.20 *(b)*

String builders are mutable. When created, the string builder `s1` has the sequence `"W"`. The call to the `append()` method in the `putO()` method appends `"O"`, resulting in `"WO"`. On return from the `put0()` method, the call to the `append()` method in the `main()` method appends `"W!"` to the string builder. The string builder `s1` now contains the sequence `"WOW!"` which is printed.

8.21 *(i)*

A `StringBuilder` is manipulated by different methods. First, the string `"12"` is appended, then the string `"34"` is inserted at index 1, resulting in the string `"1342"` in the `StringBuilder` object. Next, the `delete()` method does not modify the contents because the start and the end indexes are the same. Finally, the `replace()` method replaces the characters between the start indices 0 inclusive and the end index 1 exclusive with an empty string—that is, effectively removing the character `'1'` from index 0. The resulting string is `"342"`.

8.22 *(b)*

Remember that the default capacity of the empty `StringBuilder` is 16 characters, which can change as its contents are modified. The string `"42"` is appended first, then the second character is deleted from this string, resulting in the `StringBuilder` object containing the string `"4"`. The print statement concatenates the string `"4"` in the `StringBuilder` with the sum of its capacity (which still has the default value 16) and its length (which is 1)—in other words, the string `"4"` is concatenated with `17`. The resulting string `"417"` is printed.

8.23 *(b) and (d)*

The method call `Math.ceil(v)` returns the `double` value `11.0`, which is printed as `11.0` at (1), but as `11` at (4) after conversion to an `int`.

The method call `Math.round(v)` returns the `long` value `11`, which is printed as `11` at (2).

The method call `Math.floor(v)` returns the `double` value `10.0`, which is printed as `10.0` at (3), but as `10` at (5) after conversion to an `int`. (b) and (d), corresponding to (2) and (4), will print `11`.

8.24 *(b)*

The value `-0.5` is rounded up to `0` and the value `0.5` is rounded up to `1`.

8.25 *(b), (c), and (d)*

The expression will evaluate to one of the numbers 0, 1, 2, or 3. Each number has an equal probability of being returned by the expression.

9 Nested Type Declarations

9.1 *(e)*

The code will compile and print `123` at runtime. An instance of the `Outer` class will be created and the field `secret` will be initialized to `123`. A call to the `createInner()` method will return the reference value of the newly created `Inner` instance. This object is an instance of a non-static member class and is associated with the outer instance. This means that an object of a non-static member class has access to the members within the outer instance. Since the `Inner` class is nested in the class containing the field `secret`, this field is accessible to the `Inner` instance, even though the field `secret` is declared `private`.

9.2 *(b) and (e)*

A static member class is in many respects like a top-level class, and can contain non-`static` fields. Instances of non-static member classes are created in the context of an outer instance. The inner instance is associated with the outer instance. Several non-static member class instances can be created and associated with the same outer instance. Static member classes do not have any implicit outer instance. A static member interface, just like top-level interfaces, cannot contain non-`static` fields. Nested interfaces are always `static`.

9.3 *(d)*

The program will compile without error, and will print `1`, `3`, `4`, in that order, at runtime. The expression `B.this.val` will access the value `1` stored in the field `val` of the (outer) `B` instance associated with the (inner) `C` object referenced by the reference `obj`. The expression `C.this.val` will access the value `3` stored in the field `val` of the `C` object referenced by the reference `obj`. The expression `super.val` will access the field `val` from `A`, the superclass of `C`.

9.4 *(c) and (d)*

The class `Inner` is a non-static member class of the `Outer` class, and its qualified name is `Outer.Inner`. The `Inner` class does not inherit from the `Outer` class. The method named `doIt` is, therefore, neither overridden nor overloaded. Within the scope of the `Inner` class, the `doIt()` method of the `Outer` class is hidden by the `doIt()` method of the `Inner` class.

[9.5](#) *(e)*

Non-static member classes, unlike top-level classes, can have any access modifier. Static member classes can be declared in a top-level class and any nested class. Methods in all nested classes can be declared `static`. Only static member classes can be declared `static`. Declaring a class `static` only means that instances of the class are created without having an outer instance. This has no bearing on whether the members of the class can be `static` or not.

[9.6](#) *(c), (d), and (e)*

The method at (1) will not compile, since the parameter `i` is neither `final` nor effectively `final`, and therefore not accessible from within the inner class. The syntax of the anonymous class in the method at (2) is not correct, as the empty argument list is missing. The parameter `i` at (3) is effectively `final`, and at (4) it is `final`. The method at (5) is legally declared.

[9.7](#) *(d)*

Other static members, not only `static final` fields declared as constant variables, can be declared within a non-static member class. Members in outer instances are directly accessible using simple names (provided they are not hidden). Fields in nested static member classes need not be `final`. Anonymous classes cannot have constructors, since they have no names. Nested classes define types that are distinct from the enclosing class, and the `instanceof` type comparison operator does not take the type of the outer instance into consideration.

[9.8](#) *(d)*

Note that the nested classes are locally declared in a static context. (a) and (b) refer to the field `str1` in `Inner`. (c) refers to the field `str1` in `Access`. (e) requires the `Helper` class to be in the `Inner` class in order to compile, but this will not print the right answer. (f), (g), and (h) will not compile, as the `Helper` local class cannot be accessed using the enclosing class name.

[9.9](#) *(c)*

The field `t` denotes an instance of the anonymous inner class that extends the `Test` class. The `toString()` method is implicitly called on `t` in the print statement. The anonymous inner class overrides the `toString()` method, which is invoked. It returns the result of the following `return` statement:

Click here to view code image

```
return this.x + super.toString() + x;
```

Here, both `this.x` and `x` refer to the field `x` declared in the anonymous class, which has the character value `'>'`. This field shadows the local variable `x` in the `main()` method, which in turn shadows the field `x` in the `Test` class.

The call `super.toString()` results in the `toString()` method in the superclass `Test` to be invoked. It returns the result of the following statement:

```
return x + "42";
```

Here, the `x` refers to the field `x` in the `Test` class, which has the character value `'='`. The statement returns the string `"=42”`.

The print statement concatenates the following expression to print `">=42>"`—that is, (c):

```
'>' + "=42" + '>'
```

9.10 *(d)*

The `String` class is `final`, and therefore, cannot be extended. An anonymous inner class tries to extend the `String` class, but it will be flagged as an error by the compiler.

10 Object Lifetime

10.1 *(d)*

An object is only eligible for garbage collection if all remaining references to the object are from other objects that are also eligible for garbage collection. Therefore, if object `obj2` is eligible for garbage collection and object `obj1` contains a reference to it, then object `obj1` must also be eligible for garbage collection. Java does not have a keyword `delete`. An object will not necessarily be garbage collected immediately after it becomes unreachable. However, the object will be eligible for garbage collection. Circular references do not prevent objects from being garbage collected, only reachable references do. An object is not eligible for garbage collection as long as the object can be accessed by any live thread.

10.2 *(b)*

Before (1), the `String` object initially referenced by `arg1` is denoted by both `msg` and `arg1`. After (1), the `String` object is only denoted by `msg`. At (2), the reference `msg` is assigned a new reference value. This reference value denotes a new `String` object created by concatenating the contents of several other `String` objects. After (2), there are no references to the `String` object initially referenced by `arg1`. The `String` object is now eligible for garbage collection.

10.3 *(a)*

The only object created is the array, and it is reachable when control reaches (1).

10.4 *(a)*

All the objects created in the loop are reachable via `p`, when control reaches (1).

10.5 *(a)*

It may seem that since the method `removeAll()` sets the `songs` array reference to `null`, there would be three objects (i.e., the array itself and its two `Song` objects) eligible for garbage collection when control reaches (1). However, prior to this method invocation, this array reference is also assigned to a local array variable `songs` declared in the `main()` method. As a result, even though the `songs` array field in the `Album` object no longer references the `Song` array, the local array variable `songs` still references this array object, which is thus reachable.

10.6 *(c), (e), and (f)*

The `static` initializer blocks (a) and (b) are not legal, since the fields `alive` and `STEP` are non-`static` and `final`, respectively. (d) is not a syntactically legal `static` initializer block. The `static` block in (e) will have no effect, as its body is an empty block. The `static` block in (f) will change the value of the `static` field `count` from 5 to 1.

10.7 *(c)*

The program will compile and print `50, 70, 0, 20, 0` at runtime. All fields are given default values unless they are explicitly initialized. Field `i` is assigned the value `50` in the `static` initializer block that is executed when the class is initialized. This assignment will override the explicit initialization of field `i` in its declaration statement. When the `main()` method is executed, the `static` field `i` is `50` and the `static` field `n` is `0`. When an instance of the class is created using the `new` operator, the value of the `static` field `n` (i.e., `0`) is passed to the constructor. Before the body of the constructor is executed, the instance initializer block is executed, which assigns the values `70` and `20` to the fields `j` and `n`, respectively. When the body of the constructor is executed, the fields `i`, `j`, `k`, and `n`, and the parameter `m`, have the values `50`, `70`, `0`, `20`, and `0`, respectively.

10.8 *(f)*

This class has a blank `final boolean` instance variable `active`. This variable must be initialized when an instance is constructed, or else the code will not compile. This also applies to blank `final static` variables. The keyword `static` is used to signify that a block is a `static` initializer block. No keyword is used to signify that a block is an instance initializer block. (a) and (b) are not instance initializer blocks, and (c), (d), and (e) fail to initialize the blank `final` variable `active`.

10.9 *(c)*

The program will compile and print `2`, `3`, and `1` at runtime. When the object is created and initialized, the instance initializer block is executed first as it is declared first, printing `2`. Then the instance initializer expression is executed, printing `3`. Finally, the constructor body is executed, printing `1`. The forward reference in the instance initializer block is legal, as the use of the field `m` is on the left-hand side of the assignment.

10.10 *(c)*

This question tests understanding of execution order of initializers and constructors when an object is created. First the static initializers are executed, when classes `Music` and `Song` are loaded into memory. Therefore, the string `"-C--F-"` is printed first. The static initializers are invoked only once, so neither `"-C-"` nor `"-F-"` is printed again. This excludes (a) and (b).

When the first new `Song()` object is created, it first triggers initialization starting from its superclass instance initializer and constructor, which prints `"-D--E-"`, after which the instance initializer and constructor in the `Song` class are executed, printing `"-G--A-"`. This process is repeated for the second new song, resulting in `"-D--E--G--A-"` being printed. The final printout is `"-C--F--D--E--G--A--D--E--G--A-"`.

10.11 *(c) and (e)*

Line (1) will cause illegal redefinition of the field `width`. Line (2) uses an illegal forward reference to the fields `width` and `height`. The assignment in line (3) is legal. Line (4) is an assignment statement, and therefore illegal in this context. Line (5) declares a local variable inside an initializer block with the same name as the instance variable `width`, which is allowed. The simple name in this block will refer to the local variable. To access the instance variable `width`, the `this` reference must be used in this block.

11 Generics

11.1 *(b)*

The type of `intList` is `List` of `Integer` and the type of `numList` is `List` of `Number`. The compiler issues an error because `List<Integer>` is *not* a subtype of `List<Number>`.

11.2 *(c)*

With a reference of type `List<? super Integer>`, a set/put/write/add operation can only add an `Integer` or a subtype of `Integer` to the list. Calls to the `add()` method in the code are not a problem, as an `Integer` is added to the list.

With a reference of type `List<? super Integer>`, a get/read operation can only get an `Object` from the list. This object is not assignable to a reference of type `Number`. (3) will not compile.

11.3 *(b)*

The compiler issues an unchecked conversion warning at (1), as we are assigning a raw list to a generic list.

11.4 *(b), (f), and (g)*

We cannot create an array of a type parameter, as at (2). We cannot refer to the type parameters of a generic class in a static context—for example, in static initializer blocks, static field declarations, and as types of local variables in static methods, as at (6) and (7).

11.5 *(b), (c), (e), and (f)*

In (b), (c), (e), and (f), the parameterized type in the object creation expression is a subtype of the type of the reference. This is not the case in (a): Just because `HashMap<Integer, String>` is a subtype of `Map<Integer, String>`, it does not follow that `HashMap<Integer, HashMap<Integer, String>>` is a subtype of `Map<Integer, Map<Integer, String>>`—there is no subtype covariance relationship between concrete parameterized types. In (d) and (g), wild cards cannot be used to instantiate the class.

11.6 *(b)*

`ArrayList<Fruit>` is not a subtype of `List<? extends Apple>` at (1), and `ArrayList<Apple>` is not a subtype of `List<? super Fruit>` at (4). Any generic list can be assigned to a raw list reference. A raw list and an unbounded wildcard list are assignment compatible.

11.7 *(d)*

The compiler issues unchecked warnings for calls to the `add()` method. The `TreeSet` class orders elements according to their natural ordering. A `ClassCastException` is thrown at runtime when the statement `set.add(2)` is executed, as an `Integer` is not comparable to a `String`.

[11.8](#) *(a) and (b)*

The type of reference `g` is of raw type `Garage`. We can put any object in such a `Garage`, but only get `Object`s out. The type of value returned by the `get()` method at (6) through (8) is `Object`, and therefore, is not assignment compatible with `Vehicle`, `Car`, or `Sedan`.

[11.9](#) *(d), (e), and (f)*

In (a), the arguments in the call are `(List<Number>, List<Integer>)`. No type inferred from the arguments satisfies the formal parameters `(List<? extends T>, List<? super T>)`.

In (b), the arguments in the call are `(List<Number>, List<Integer>)`. The actual type parameter is `Number`. The arguments do not satisfy the formal parameters `(List<? extends Number>, List<? super Number>)`. `List<Number>` is a subtype of `List<? extends Number>`, but `List<Integer>` is not a subtype of `List<? super Number>`.

In (c), the arguments in the call are `(List<Number>, List<Integer>)`. The actual type parameter is `Integer`. The arguments do not satisfy the formal parameters `(List<? extends Integer>, List<? super Integer>)`. `List<Number>` is not a subtype of `List<? extends Integer>`, although `List<Integer>` is a subtype of `List<? super Integer>`.

In (d), the arguments in the call are `(List<Integer>, List<Number>)`. The inferred type is `Integer`. The arguments satisfy the formal parameters `(List<? extends Integer>, List<? super Integer>)`.

In (e), the arguments in the call are `(List<Integer>, List<Number>)`. The actual type parameter is `Number`. The arguments satisfy the formal parameters `(List<? extends Number>, List<? super Number>)`.

In (f), the arguments in the call are `(List<Integer>, List<Number>)`. The actual type parameter is `Integer`. The arguments satisfy the formal parameters `(List<? extends Integer>, List<? super Integer>)`.

[11.10](#) *(f)*

(a) invokes the zero-argument constructor at (1). (b) invokes the constructor at (2) with `T` as `String` and `V` as `String`.

(c) invokes the constructor at (2) with `T` as `String` and `V` as `Integer`.

(d) invokes the constructor at (3) with `T` as `Integer` and `V` as `String`.

(e) invokes the constructor at (3) with `T` as `String` and `V` as `Integer`.

(f) cannot infer type arguments for `Box<>`. From the constructor call signature `(String, Integer)` one would assume that `T` was `String` and `V` was `Integer`. The parameterized type `Box<Integer>` of the reference on the left-hand side implies `T` is `Integer`, which contradicts that `T` is `String` on the right-hand side.

11.11 *(b)*

It is the fully qualified name of the class after erasure that is printed at runtime. Note that it is the type of the object, not the reference, that is printed. The erasure of all the lists in the program is `ArrayList`.

11.12 *(e)*

(a) contains incompatible types for assignment in the `main()` method. The method will return a `Collection` whose element type is some unknown subtype of `CharSequence` (`Collection<? extends CharSequence>`). As it is not known which subtype, assignment to `Collection<String>` cannot be allowed `.`

(b) contains an incompatible return value in the `delete4LetterWords()` method. The declared return type is `List<E>` but the `return` statement returns a `Collection<E>`. It cannot convert from `Collection<E>` to `List<E>`.

In (c), the reference `words` denotes a `Collection` whose element type is some unknown subtype of `CharSequence` (`Collection<? extends CharSequence>).` In the `for(:)` loop, the loop variable `word` is of type `E`. The unknown element type of `words` cannot be converted to `E`.

(d) contains an incompatible return value in the `delete4LetterWords()` method: It cannot convert from `Collection<E>` to `List<E>`, as explained in (b). In the `for(:)` loop, the unknown element type of `words` cannot be converted to an element of type `E`, as explained in (c). (e) is OK.

In (f), the keyword `super` cannot be used in a constraint. It can only be used with a wildcard (`?`).

11.13 *(b) and (f)*

After erasure, the method at (1) has the signature `overloadMe(List, List)`. Since all methods are declared `void`, they must differ in their parameter list after erasure in order to be overloaded with the method at (1). All methods have different parameter lists from that of the method at (1), except for the declarations (b) and (f). In other words, all methods have signatures that are not override equivalent to the signature of the method at (1), except for (b) and (f).

11.14 *(b)*

Passing or assigning a raw list to either a list of `Integer`s or to a list of type parameter `T` is not type-safe. Passing or assigning a raw `List` to a `List<?>` is always permissible.

11.15 *(c), (f), (i), and (k)*

The type parameter `N` in `SubC1` does *not* parameterize the supertype `SupC`. The erasure of the signature at (3) is the same as the erasure of the signature at (1) (i.e., it is a name clash). Therefore, of the three alternatives (a), (b), and (c), only (c) is correct. The type parameter `N` in `SubC1` cannot be guaranteed to be a subtype of the type parameter `T` in `SupC`—that is, incompatible return types for the `get()` methods at (4) and (2), which are not overridden. Also, methods cannot be overloaded if only return types are different. Therefore, of the three alternatives (d), (e), and (f), only (f) is correct.

The type parameter `N` in `SubC2` is a subtype of the type parameter `M`, which parameterizes the supertype `SupC`. The erasure of the signature at (5) is still the same as the erasure of the signature at (1) (i.e., it is a name clash). Therefore, of the three alternatives (g), (h), and (i), only (i) is correct.

The type parameter `N` in `SubC1` is a subtype of the type parameter `T` (through `M`) in `SupC`—that is, covariant return types for the `get()` methods at (6) and

(2), which are overridden. Therefore, of the three alternatives (j), (k), and (l), only (k) is correct.

11.16 *(a), (c), and (e)*

In (a), because of the way an enum type `E` is implemented as a subtype of the `java.lang.Enum<E>` class in Java, we cannot define a generic enum type.

In (c), generic exceptions or error types are not allowed because the exception handling mechanism is a runtime mechanism and the JVM is oblivious to generics.

In (e), anonymous classes do not have a name, but a class name is needed for declaring a generic class and specifying its formal type parameters. A *parameterized* anonymous class can always to declared.

11.17 *(d)*

Casts are permitted, as at (2) through (6), but can result in an unchecked warning. The *assignment* at (5) is from a raw type (`List`) to a parameterized type (`List<Integer>`), resulting in an unchecked assignment conversion warning. Note that at (5) the cast does not pose any problem. It is the assignment from generic code to legacy code that can be a potential problem, and flagged as an unchecked warning.

At (6), the cast is against the erasure of `List<Integer>`—that is, `List`. The compiler cannot guarantee that `obj` is a `List<Integer>` at runtime, and therefore flags the cast with an unchecked warning.

Only reifiable types in casts do not result in an unchecked cast warning.

11.18 *(e)*

Instance tests in the `scuddle()` method use the reified type `List<?>`. All assignments in the `main()` method are type-safe.

11.19 *(c)*

The erasure of `E[]` in the method `copy()` is `Object[]`. The array type `Object[]` is actually cast to `Object[]` at runtime—that is, an identity cast. The method `copy()` returns an array of `Object`. In the `main()` method, the assignment of this array to an array of `String`s results in a `ClassCastException`.

11.20 *(e)*

The method header at (1) is valid. The type of the variable arity parameter can be generic. The type of the formal parameter `aols` is an array of `List`s of `T`. However, the compiler issues a potential heap pollution warning because of variable arity parameter `aols`.

The `main()` method at (2) can be declared as `String...`, as it is equivalent to `String[]`, but no potential heap pollution warning is issued, as it is a reifiable type. The statement at (3) creates an array of `List`s of `String`s. However, the compiler issues an unchecked conversion warning, since a raw type (`List[]`) is being assigned to a parameterized type (`List<String>[]`).

The formal type parameter `T` is inferred to be `String` in the method call at (4). The method `doIt()` prints each list in its variable arity parameter `aols`.

12 Collections, Part I: ArrayList<E>

12.1 *(e)*

The `for(;;)` loop correctly increments the loop variable so that all the elements in the list are traversed. Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

12.2 *(b) and (c)*

In the method `doIt1()`, one of the common elements (`"Ada"`) between the two lists is reversed. The value `null` is added to one of the lists but not the other.

In the method `doIt2()`, the two lists have common elements. Swapping the elements in one does not change their position in the other.

12.3 *(c)*

The element at index 2 has the value `null`. Calling the `equals()` method on this element throws a `NullPointerException`.

12.4 *(f)*

Deleting elements when iterating over a list requires care, as the size changes and any elements to the right of the deleted element are shifted left. Incrementing the loop variable after deleting an element will miss the next element (i.e., the last occurrence of `"Bob"`). Removing elements using the `for(;;)` loop does not throw a `ConcurrentModificationException` at runtime.

12.5 *(f)*

The `while` loop will execute as long as the `remove()` method returns `true`—that is, as long as there is an element with the value `"Bob"` in the list. The `while` loop body is the empty statement. The `remove()` method does not throw an exception if an element value is `null`, or if it is passed a `null` value.

12.6 *(b)*

An `ArrayList` object is populated with the content from the `String` array. Just like with an array, an array list has a 0-based index. The item at index 1 in this array list is replaced with the string `"X"`, making this array list content `[A,X,B,A]`. Then a new item is added at the same index position, causing all other items in the list to be shifted by one position, making this array list content `[A,X,X,B,A]`. Lastly, an item at index 2 is removed, giving the result `[A,X,B,A]`.

12.7 *(a)*

The method `Arrays.asList()` creates a fixed-size list, which does not allow items to be added or removed, but its content can be changed, which is what the `set()` operations do, replacing items at index 1 and 2 with `"X"`.

12.8 *(c)*

The two arrays and the list in the `main()` method contain references to the same `Song` objects. These are not independent copies, so modifications on a shared `Song` object will be visible no matter how this object is accessed.

12.9 *(b)*

A list that is created using the `List.of()` method shares the elements with the original array. However, changes applied to the original array are not reflected in the list.

12.10 *(a)*

The method `toArray()` returns an array with all the elements in the list. The type of the array is given by the array passed as a parameter. If the length of the argument array is equal to the size of the list, the argument array is used. The argument array is also used if its length is greater than the size of the list, but after copying the elements to the array, the remaining elements in the array are filled with `null` values. Otherwise, a new array of appropriate size is created. In the sample code, the length of the array is equal to the size of the list. Therefore, the argument array is used. Afterwards, the lowercase version of the element at index 0 in the original list is assigned to the element at index 1 in the array.

12.11 *(b)*

An empty `ArrayList` object is created to store `Character` objects, using a constructor with a capacity of 3. Five `char` values from `'a'` to `'e'` are boxed as `Character` objects and added to this list. Remember that a list auto-expands its capacity as required.

13 Functional-Style Programming

13.1 *(e)*

A functional interface can be implemented by lambda expressions and classes. A functional interface declaration can only have one abstract method declaration. In the body of a lambda expression, all members in the enclosing class can be accessed. In the body of a lambda expression, only final or effectively final local variables in the enclosing scope can be accessed.

13.2 *(e), (f), (g), and (i)*

The assignments at (5), (6), (7), and (9) will not compile. We must check whether the function type of the target type and the type of the lambda expression are compatible. The function type of the target type `p1` in the assignment statements from (1) to (5) is `String -> void` (i.e., a `void` return). The function type of the target type `p2` in the assignment statements from (6) to (10) is `String -> String` (i.e., a non-`void` return). Below, the functional type of the target type is shown in a comment with the prefix `LHS` (left-hand side), and the type of the lambda expression for each assignment from (1) to (10) is shown in a comment with the prefix `RHS` (right-hand side).

Click here to view code image

```
    Funky1 p1;                                      //      LHS: String -> void
    p1 = s -> System.out.println(s);                // (1) RHS: String -> void
    p1 = s -> s.length();                           // (2) RHS: String -> int
    p1 = s -> s.toUpperCase();                      // (3) RHS: String -> String
    p1 = s -> { s.toUpperCase(); };                 // (4) RHS: String -> void
//  p1 = s -> { return s.toUpperCase(); };          // (5) RHS: String -> String

    Funky2 p2;                                      //      LHS: String -> String
//  p2 = s -> System.out.println(s);                // (6) RHS: String -> void
//  p2 = s -> s.length();                           // (7) RHS: String -> int
    p2 = s -> s.toUpperCase();                      // (8) RHS: String -> String
//  p2 = s -> { s.toUpperCase(); };                 // (9) RHS: String -> void
    p2 = s -> { return s.toUpperCase(); };          // (10)RHS: String -> String
```

Remember that the non-`void` return of a lambda expression with an *expression statement* as the body can be interpreted as a `void` return, if the function type of the target type returns `void`. This is the case at (2) and (3). The return value is ignored. The type `String -> String` of the lambda expression at (5) is not compatible with the function type `String -> void` of the target type `p1`.

The type of the lambda expression at (6), (7), and (9) is not compatible with the function type `String -> String` of the target type `p2`.

13.3 *(d)*

The three interfaces are functional interfaces. `AgreementB` explicitly provides an abstract method declaration of the `public` method `equals()` from the `Object` class, but such declarations are excluded from the definition of a functional interface. Thus `AgreementB` effectively has only one abstract method. A functional interface can be implemented by a concrete class, such as `Beta`. The function type of the target type in the assignments (1) to (3) is `void -> void`.

The type of the lambda expression at (1) to (3) is also `void -> void`. The assignments (1) to (3) are legal.

The assignment at (4) is legal. Subtype references are assigned to supertype references. References `o`, `a`, and `c` refer to the lambda expression at (3).

The assignment at (5) is legal. The reference `b` has the type `AgreementB` and class `Beta` implements this interface.

The code at (6), (7), and (8) invokes the method `doIt()`. The code at (6) evaluates the lambda expression at (3), printing `Jingle|`. The code at (7) invokes the `doIt()` method on an object of class `Beta`, printing `Jazz|`. The code at (8) also evaluates the lambda expression at (3), printing `Jingle|`.

At (9), the reference `o` is cast down to `AgreementA`. The reference `o` actually refers to the lambda expression at (3), which has target type `AgreementC`. This interface is a subtype of `AgreementA`. A subtype is cast to a supertype, which is allowed, so no `ClassCastException` is thrown at runtime. Invoking the `doIt()` method again results in evaluation of the lambda expression at (3), printing `Jingle|`.

Apart from the declarations of the lambda expressions, the rest of the code is plain-vanilla Java. Note also that the following assignment that defines a lambda expression would not be valid, since the `Object` class is not a functional interface and therefore cannot provide a target type for the lambda expression:

Click here to view code image

```
Object obj = () -> System.out.println("Jingle"); // Compile-time error!
```

13.4 *(c)*

The method `removeIf()` accepts as an argument a lambda expression that implements a `Predicate<E>` interface. This method removes all strings of length 3 from the list. The `for (:)` loop calculates the sum of the lengths of the remaining strings in the list, producing a result of 9.

13.5 *(c)*

The method `removeIf()` accepts a lambda expression that first converts a string to lowercase and then tests whether the resulting string starts with the character `'a'`. Note that the predicate only performs the test, and it does not actually modify the strings in the list. Only the strings `"ANNA"` and `"ALICE"` pass the test and are removed.

13.6 *(i)*

The lambda expression uses identifier `s` as a parameter name, which is illegal because a variable called `s` is already defined in the enclosing context of the lambda expression.

13.7 *(c)*

There are two predicates defined in this code. The first predicate determines whether a string contains the letter `O`, and the second one determines whether a string ends with the letter `P`. The composed predicate `filter1.and(filter2).negate()` determines whether a string does *not* contain an `O` *or* it does *not* end with a `P`. Only the strings `"PLOT"` and `"LEAP"` pass this test and are removed from the list by the `removeIf()` method, leaving only the strings `"FLOP"` and `"LOOP"` in the list.

13.8 *(d)*

The `compose()` method is inherited by the `UnaryOperator<T>` from its super-interface `Function<T, T>`. This method returns a `Function<T, T>`. As an instance of a super-type (`Function<T, T>`) cannot be assigned to a subtype (`UnaryOperator<T>`), the assignment to `f3` results in a compile-time error.

13.9 *(b)*

All `String` values in the list are replaced with their lowercase equivalents using the `replaceAll()` method which accepts a lambda expression that implements a `Unary-Operator<String>`. The two consumers are applied to the values in this `List`.

Consumer `c1` is changing the first letter of every string in the list to uppercase, but it does not replace actual `String` objects stored within this list. Next, consumer `c2` prints the content of this list, which has been produced by the `replaceAll()` method.

13.10 *(b)*

Regarding method references, the method `isEven()` is static and therefore should be referred to using the class name `Test`, while the method `print-Value()` is an instance method, and therefore should be referred to using a reference of the class `Test`.

13.11 *(a)*

The target reference for the bounded instance method reference is set explicitly. The unbounded instance method reference interprets the first argument as the target reference.

13.12 *(d)*

Notice that the `BiFunction` in this example is using raw type. Therefore, the `x` and `y` parameters are of the `Object` type. This means that a division operator cannot be applied in this case.

13.13 *(a)*

Functions `f1` and `f2` are combined to concatenate the prefix and the postfix around the value supplied to the `apply()` method argument. Notice that conversion to `String` works for any object in Java. Therefore, `Function` objects can use raw types.

14 Object Comparison

14.1 *(b) and (d)*

It is recommended that (a) is fulfilled, but it is not a requirement. (c) is also not required, but such objects will lead to collisions in the hash table, as they will map to the same bucket.

14.2 *(a), (b), (d), and (h)* (c) is eliminated, since the `hashCode()` method cannot claim inequality if the `equals()` method claims equality. (e) and (f) are eliminated, since the `equals()` method must be reflexive, and (g) is eliminated, since the `hashCode()` method must consistently return the same hash code during execution.

14.3 *(b), (d), and (e)* (a) and (c) fail to satisfy the properties of an equivalence relation. (a) is not transitive, and (c) is not symmetric.

14.4 *(a) and (e)* (b) is not correct, since it will throw an `ArithmeticException` when called on a newly created `Measurement` object. (c) and (d) are not correct, since they may return unequal hash codes for two objects that are equal according to the `equals()` method.

14.5 *(c)*

The generic static method `cmp()` returns a comparator (implemented as a lambda expression) that reverses the natural ordering of a `Comparable` type. The natural ordering of the class `Person` is ordering by `name` first and then by `age`, using the reverse comparators `strCmp` and `intCmp`. `p1` is *less* than `p2` because of `name`, and `p1` is *greater* than `p3`, because of `age`, as their names are equal.

14.6 *(d)*

All methods implement reverse natural ordering, except the method at (4). The method reference `Comparable::compareTo` is equivalent to the lambda expression `(e1, e2) -> e1.compareTo(e2)`—that is, natural ordering.

14.7 *(b)*

A lambda expression that implements the `Comparator<String, String>` is used to sort the array in ascending order based on the text represe`ntation` of `Integer` objects. Basically, each array element is converted to a `String` before it is compared. The ordering is that of `String` objects, where `"-23"` is less than `"-41"` lexicographically.

14.8 *(a)*

The lambda expression that implements the `Comparator<Album>` interface defines a total ordering of `Album`s based on the difference between the lengths of the album titles, resulting in the list being sorted in ascending order by `title` length. The resulting list is then printed using the lambda expression that implements the `Consumer<Album>` interface.

14.9 *(b)*

The `equals()` method of the `Album1` class checks whether the object is not `null` and of the same type as the current object before comparing album titles. This is a strict check that verifies whether the object with which the current object is compared is of exactly the same type, using the following condition: `(getClass() != obj.getClass())`. Alternatively, a less strict check that allows type substitution is also possible: `(obj instanceof Album1)`. The difference between these two approaches is that the `instanceof` operator can return `true` when comparing this object to another object that is an instance of the subtype. Of course, this would not be the case if specific class types are compared.

Note that the logic in the `main()` method compares an `Album1` to an `LP`, which is actually a subclass of `Album1`. This means that even though both of these objects have the same title, they would not be considered equal because the logic of the `equals()` method implements a strict type comparison.

14.10 *(b) and (d)*

The `Comparator<A>` interface defines the `compare()` method that is designed to compare two argument objects of class `A` to establish their ordering. Each `Comparator<A>` implementation can define a different total ordering for the objects.

15 Collections: Part II

15.1 *(a)*

The expression in the `for(:)` loop header (in this case the call to the `makeCollection()` method) is only evaluated once.

15.2 *(c) and (d)*

The `for(:)` loop does not allow the list to be modified structurally. In (a) and (b), the code will throw a `java.util.ConcurrentModificationException`. Note that the iterator in (d) is less restrictive than the `for(:)` loop, allowing elements to be removed in a controlled way.

15.3 *(d)*

The iterator implemented will iterate over the elements of the list in the reverse order, and so will the `for(:)` loop. The `Iterable<E>` and the `Iterator<E>` interfaces are implemented correctly. Note that the anonymous class that implements the iterator is parameterized by the formal type parameter `T` of the generic class `AnotherListIterator<T>`.

15.4 *(b) and (d)*

Some operations on a collection may throw an `UnsupportedOperationException`. This exception type is unchecked, and the user code is not required to explicitly handle unchecked exceptions. A `List<E>` allows duplicate elements. An `Array-List<E>` is implemented using a resizable array. The capacity of the array will be expanded automatically when needed. The `Set<E>` allows at most one `null` element.

15.5 *(d)*

The program will compile without error, and will print all primes below 25 at runtime. All collection implementations used in the program implement the `Collection<E>` interface. The implementation instances are interchangeable when denoted by `Collection` references. None of the operations performed on the implementations will throw an `UnsupportedOperationException`. The program finds the primes below 25 by removing all values divisible by 2, 3, and 5 from the set of values from 2 through 25.

15.6 *(b)*

The `remove()` method removes the last element returned by either the `next()` or `previous()` method. The four `next()` calls return `A`, `B`, `C`, and `D`. `D` is subsequently removed. The two `previous()` calls return `C` and `B`. `B` is subsequently removed.

15.7 *(c), (d), (e), and (f)*

Sets cannot have duplicates. `HashSet<E>` does not guarantee the order of the elements in (a) and (b), so there is no guarantee that the program will print `[1, 9]`. Because `LinkedHashSet<E>` maintains elements in insertion order in (c) and (d), the program is guaranteed to print `[1, 9]`. Because `TreeSet<E>` maintains elements sorted according to the natural ordering in (e) and (f), the program is guaranteed to print [`1, 9]`.

15.8 *(c) and (d)*

The output from each statement is shown below. (a) `[sea, shell, soap]`

(b) `[sea, shell]`

(c) `[soap, swan]`

(d) `[swan]`

(e) `[shell, soap]`

(f) `[sea, shell]`

15.9 *(b) and (d)*

Although all *keys* in a map must be unique, duplicate *values* can occur. Since values are not unique, the `values()` method returns a `Collection<V>` and not a `Set<V>`. The collections returned by the `keySet()`, `entrySet()`, and `values()` methods are backed by the underlying map. This means that changes made in one are reflected in the other. Although implementations of the `SortedMap<K, V>` interface maintain the entries sorted according to key-sort order, this is not a requirement for classes that implement the `Map<K, V>` interface. For instance, the entries in a `HashMap<K, V>` are not sorted.

15.10 *(a), (c), and (d)*

The key of a `Map.Entry<K, V>` cannot be changed, since the key is used for locating the entry within the map. There is no `set()` method. The `setValue()` method is optional.

15.11 *(b)*

A set is a collection of unique elements, so an attempt to insert the same element twice is ignored, with no exception raised. The ordering of elements in the set is determined by the `Comparator<E>` passed to the `TreeSet` constructor. The comparator passed compares the element strings in the *reverse natural ordering*.

15.12 *(b)*

The `set1` object sorts elements according to the *reverse natural ordering*. The `set2` object retains that ordering. In the statement

Click here to view code image

```
NavigableSet<String> set2 = new TreeSet<>(set1);
```

the signature of the constructor called is the following:

Click here to view code image

```
TreeSet<String>(SortedSet<String> set)
```

resulting in the same ordering for the elements in `set2` as in `set1` (i.e., reverse natural ordering). Note that class `NavigableSet<E>` is a subclass of class `SortedSet<E>` class.

15.13 *(a)*

The `set1` object sorts the elements according to reverse natural ordering. In the statement

Click here to view code image

```
NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
```

the signature of the constructor called is

Click here to view code image

```
TreeSet<String>(Collection<? extends String> collection)
```

resulting in the elements in `set2` being sorted according to *natural ordering*, and not according to the reverse natural ordering of `set1`.

15.14 *(a)*

The `set1` object sorts its elements in reverse natural ordering. It is polled from the tail, so its elements are fetched according to natural ordering. On the other hand, the elements in `set2` are sorted according to natural ordering. `set2` is polled from the head, so its elements are fetched according to natural ordering.

15.15 *(b)*

A map view method creates half-open intervals (i.e., the upper bound is not included), unless the inclusion of the bounds is explicitly specified. Clearing a map view clears the affected entries from the underlying map. The argument to the `sumValues()` method can be any subtype of `Map<K, V>`, where the type of the value is `Integer`.

15.16 *(b), (e), and (f)* (a) throws a `ConcurrentModificationException`. We cannot remove an entry in a `for(:)` loop. (c) throws a `ConcurrentModificationException` as well, even though we use an iterator. The `remove()` method is called on the map, not on the iterator. The argument to the `remove()` method of the map must implement `Comparable`. `Map.Entry<K, V>` does not, resulting in a `ClassCastException` in (d).

We can remove an entry from the underlying map when iterating over the key set using an iterator, as in (b). (e) creates a map view of one entry and clears it, thereby clearing it also from the underlying map. (f) removes the entry for `"Shampoo"` from the map, since the lambda expression returns the value `null`.

15.17 *(e)*

The variable `sumVal` is not effectively `final` when referenced in the lambda expression body, as it is incremented for each entry in the map.

15.18 *(c)*

The `computeIfAbsent()` method returns an empty `TreeSet` if the key is not found in the map. If the key is found, it returns the associated `TreeSet`. The `add()` method is invoked on the `TreeSet` that is returned by the `computeIfAbsent()` method. The `add()` method adds its argument to this `TreeSet`. The resulting map is a *multimap*— that is, a key can be associated with a collection of values.

15.19 *(b)*

The `BiFunction<Integer, String, String>` implemented by the lambda expression computes a new value for the key of an entry in the map. The `switch` statement determines the new value based on the key. The lambda expression returns the value `"FIRST"` for key 1, the value `"SECOND"` for key 2, and so on. The `replaceAll()` method replaces the value of each entry in the map with the new value computed for the key by the `BiFunction<T, U, R>`.

15.20 *(a)*

The class `StringBuilder` implements the `Comparable<E>` interface. The `sort()` method sorts the elements in reverse natural ordering: `[C, B, A]`. The method `sublist()` returns the elements in the open interval `[1, 2)`—that is, the element at index `1`, which is `"B"`.

15.21 *(b), (c), (f), and (g)*

The `Collections.addAll()` method adds the elements to an existing list when it is called. All three elements are in `list1` when (1) is executed. The `Arrays.asList()` method returns a new list every time it is called. Only the string `"Howdy"` is in `list2` when (2) is executed. The `Collection.addAll()` method adds the elements of its argument collection to the collection on which it is called. In this case, `list3` has the same elements as `list1`. Calling the constructor with a collection as an argument initializes the new list with the elements of the specified collection. In this case, `list4` has the same elements as `list2`.

Creating a new list by calling the constructor with a collection as an argument returns an `ArrayList` initialized with the elements of the collection.

15.22 *(a) and (f)*

The largest value a match can return is the largest index—that is, *array.length – 1* (`==3`). The key must be equal to the largest element in the array. If no match is found, a negative value is returned, which is computed as follows: *– (insertion point + 1)*. The smallest value is returned for a key that is greater than the largest element in the array. This key must obviously be placed at the index *array.length (==4)*, after the largest element—that is, the insertion point is 4. The value of the expression *– (insertion point + 1)* is `-5`, which is the smallest value printed by the method.

15.23 *(c)*

The operation `pollFirst()` does not throw an exception, but rather returns `null` when the `Deque` object is empty. The operations `peekFirst()` and `peekLast()` return the first and last elements from the `Deque` object, respectively, but do not remove elements from the `Deque`. The operations `pollFirst()` and `pollLast()` return the first and last elements from the `Deque` object, respectively, and remove them from the `Deque`. The operation `offerFirst()` inserts elements at the head of the `Deque`. The operation `offerLast()` inserts elements at the tail of the `Deque`.

15.24 *(b)*

A set cannot have duplicates. This means that object `x` was only added once to the set. The `add()` method does not throw an exception, but rather returns `false` when an element cannot be added to the set.

15.25 *(d)*

The first two add operations result in the list `[1, 2]`. Next, a `null` value is inserted at index 2, and the value `3` is inserted at index 3, which results in the list `[1, 2, null, 3]`. Next, the value `4` is inserted at index 2, shifting elements towards the end of the list, resulting in the list `[1, 2, 4, null, 3]`. The element at index 2 is replaced with the value `3`, giving the list `[1, 2, 3, null, 3]`, and then the element at index 2 is removed, giving the list `[1, 2, null, 3]`. Finally, the value `2` is inserted at index 2, which results in the list `[1, 2, 2, null, 3]`.

16 Streams

16.1 *(b)*

The `mapToInt()` operation converts a `Stream<String>` to an `IntStream` whose elements are the length of the strings in its input stream. The `int` stream will contain the values `1`, `3`, `2`, and `4`, corresponding to the length of the strings. The `filter()` operation discards strings of length `4`. Its output stream will only contain the values `1`, `3`, and `2`. The `reduce()` method performs a functional reduction, starting with the initial value of `1`. Its accumulator multiplies the cumulative result with the current value in the `int` stream, with the computation proceeding as follows:

```
(x, y) -> x * y
(1, 1) -> 1 * 1 => 1
(1, 3) -> 1 * 3 => 3
(3, 2) -> 3 * 2 => 6
```

16.2 *(d)*

The `filter()` intermediate operation is designed to return a stream whose elements match the given `Predicate`. The `findFirst()` terminal operation does not necessarily return the first element from the stream when this stream is processed in parallel mode. The `reduce()` terminal operation performs a functional reduction on the elements of the stream, and it uses an accumulator and not a `Predicate`. The `sorted()` intermediate operation sorts the elements according to their natural order, or according to the total order specified by a `Comparator`.

16.3 *(d) and (e)*

(a) performs a functional reduction starting with the initial value `0` and adding all values in the stream to compute the sum of the values.

(b) performs the same functional reduction as in (a) but in parallel mode.

(c) performs the same functional reduction as in (a), but does not use the initial value of `0`. It uses the value of the first element in the stream, if there is one. Since the stream can be empty, it returns an `OptionalInt` object. The `orElse()` operation on this `OptionalInt` object retrieves an `int` value if it has one; otherwise, the operation returns the value `0`.

(d) uses `0` as the initial value, which means that this value will be returned if the steam is empty. Therefore, the operation is guaranteed to return an `int` value, and not an `OptionalInt`. The `orElse()` operation cannot be invoked on an `int` value, so this code will not compile.

(e) refers to the variable `sum` within the lambda expression. As it has not been initialized, the code will fail to compile. Note that only final or effectively final variables can be referenced within a lambda expression.

(f) computes the sum of all values in the stream.

16.4 *(b) and (d)*

The stream will contain the following values: `0`, `1`, `2`, `3`, and `4`. Note that `x` designates the cumulative value and `y` designates the current element. (a) performs functional reduction using the identity value `0` as the initial value and the accumulator adds 1 to the cumulative result for each element. The reduction proceeds as follows:

```
(x, y) -> x + 1
(0, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(b) performs a similar functional reduction as in (a), but uses the value of the first element (`0`) as the initial value. So it would result in one addition less than (a). The reduction proceeds as follows:

```
(x, y) -> x + 1
(0, 1) -> 0 + 1 => 1
(1, 2) -> 1 + 1 => 2
(2, 3) -> 2 + 1 => 3
(3, 4) -> 3 + 1 => 4
```

(c) performs a functional reduction similar to (a), but now the accumulator increases the value of the stream element `y` by 1. The reduction proceeds as follows:

```
(x, y) -> y + 1
(0, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(d) performs a functional reduction using the initial value `0` and where the accumulator returns the value of the stream element `y`. The reduction proceeds as follows:

```
(x, y) -> y
(0, 0) -> 0 => 0
(0, 1) -> 1 => 1
(1, 2) -> 2 => 2
(2, 3) -> 3 => 3
(3, 4) -> 4 => 4
```

(e) performs a function reduction which is similar to (c), except that it uses the identity value of `1` as the initial value. The reduction proceeds as follows:

```
(x, y) -> y + 1
(1, 0) -> 0 + 1 => 1
(1, 1) -> 1 + 1 => 2
(2, 2) -> 2 + 1 => 3
(3, 3) -> 3 + 1 => 4
(4, 4) -> 4 + 1 => 5
```

(f) performs a functional reduction of the stream elements using the `count()` operation, which in this case results in the value `5`.

16.5 *(d)*

(a) produces three groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream. The `Predicate` expression discards any value containing the string `"C"`.

```
1 []
2 [AA, DD]
3 [BBB, EEE]
```

(b) produces three groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream. The `filter()` operation discards all values except those that contain the string `"C"`.

```
1 [C]
2 []
3 []
```

(c) produces two groups based on the `Integer` values corresponding to the lengths of `String` objects in the stream. However, the `filter()` operation discards any values containing the string `"C"`, before the groups are created.

```
2 [AA, DD]
3 [BBB, EEE]
```

(d) results in a single group based on the `Integer` values corresponding to the length of `String` objects in the stream. The `filter()` operation discards any values except those that contain the string `"C"`, before any groups are created.

```
1 [C]
```

16.6 *(d)*

An infinite stream of string `"A"` is generated. The first `peek()` operation prints the string `"B"`. The `Predicate` of the `takeWhile()` operation returns `false` immediately on encountering the first element in the stream which is `"A"`. The `takeWhile()` operation only takes an element if it is not `"A"`. It short-circuits the stream processing, resulting in an output stream that is empty. The `Consumer` of the second `peek()` operation does not execute, as the stream is empty. The `anyMatch()` terminal operation returns `false` on encountering an empty stream.

16.7 *(a)*

A stream of `int` values that correspond to character codes for letters `'a'`, `'b'`, `'c'`, and `'d'` is generated. These values are mapped to single-letter strings that are converted to uppercase. The `filter()` operation discards a letter if it does not match a vowel, which results in an output stream with only the element `"A"`, which is printed.

16.8 *(b) and (c)*

A stream of `int` values 0, 1, 2, 3, and 4 is generated. The `filter()` operation discards all even numbers from this stream, retaining only the odd numbers 1 and 3, which are then printed.

(a) generates a stream of `int` values 0, 1, 2, 3, 4, and 5 which is one value more than in the program. Even numbers are discarded, retaining only the odd numbers 1, 3, and 5 which are then printed.

(b) generates a stream of `int` values between 0 and 10. The `takeWhile()` operation only takes values less than 5. It truncates the stream when the element is greater than or equal to 5. The `filter()` operation discards all even numbers from the truncated stream, retaining only the odd numbers 1 and 3, which are then printed.

(c) generates a stream of `int` values between 0 and 10, which is then truncated to the first five values. The `filter()` operation discards all even numbers from this truncated stream, retaining only the odd numbers 1 and 3, which are then printed.

(d) generates an infinite stream of 0s. The expression `x++` will always evaluate to 0, when `x` is initialized to 0. The `takeWhile()` operation will continue to take elements from the stream, as its `Predicate` will always return `true`. The `filter()` operation will continue to discard each element, as its value will always be `0`. The terminal operation will never get to process an element. This state of affairs will continue indefinitely, with nothing being printed.

(e) does not compile because the variable `x` referred to in the lambda expression is not final. The expression `x++` will change the value in `x`, which is not permitted.

16.9 *(d)*

Two streams of `String` objects containing the values `"A"`, `"B"`, `"C"` and the values `"X"`, `"Y"`, `"Z"` are concatenated into a single stream. The resulting stream has the values `"X"`, `"Y"`, `"Z"`, `"A"`, `"B"`, `"C"`.

The functional reduction concatenates the elements from this new stream into a single string. This operation returns an `Optional<String>`, as the reduction uses the first element as the initial value. The result string in the `Optional<String>` is returned by the `get()` method of the `Optional` class.

Note that `a` denotes the cumulative result and `b` denotes the current element in the stream. The reduction operation is performed as follows:

Click here to view code image

```
(a,          b)  ->  b  +  a
("X",       "Y") -> "Y" + "X" => "YX"
("YX",      "Z") -> "Z" + "YX" => "ZYX"
("ZYX",     "A") -> "A" + "ZYX" => "AZYX"
("AZYX",    "B") -> "B" + "AZYX" => "BAZYX"
("BAZYX",   "C") -> "C" + "BAZYX" => "CBAZYX"
```

16.10 *(a)*

All process a stream of `String` objects that are one-letter strings from `"A"` to `"E"`. Grouping is done based on a classifier which is a `Function`, whereas partitioning is done based on a `Predicate`. Identical lambda expressions implement the classifier and the predicate in all options. The lambda expression returns `true` if the single-letter string is a vowel. The map created by both operations will have the type `Map<Boolean, List<String>>`, where the keys are `Boolean` and the value associated with a key is a `List<String>`. The list is created implicitly, as in (a) and (b), or explicitly in a downstream collector, as in (c) and (d).

The filtering is done by the same predicate in all options, discarding any one-letter string that is greater than the string `"A"`. Effectively, the only element processed by the stream is the string `"A"`.

The `partitioningBy()` operation always creates entries for the `Boolean.TRUE` and `Boolean.FALSE` keys in the result map, even if no values can be computed for these keys from the stream elements. On the other hand, the `groupingBy()` operation creates entries for keys computed by its classifier—that is, an entry is created for the key `Boolean.TRUE` or `Boolean.FALSE` depending on the elements in the stream. However, when the `filtering()` operation is used as a downstream collector in the grouping operation, entries for both the `Boolean.TRUE` and `Boolean.FALSE` keys are created, regardless of whether any value is associated with these keys.

In (a), grouping creates only one entry for the `Boolean.TRUE` key in the result map based on its `Predicate` being `true`, since the only element `"A"` in the stream is a vowel. The output is the following:

```
true [A]
```

In (b), partitioning creates two entries in the result map: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants). The only element `"A"` is associated with the `true` key as it a vowel. The output is the following:

```
false [] true [A]
```

In (c), grouping creates two entries: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants), as its downstream collector is a `filtering()` operation. Since the string `"A"` is a vowel, it is accumulated in the list associated with the `Boolean.TRUE` key. The list associated with the `Boolean.FALSE` key remains empty. The output is the following:

```
false []
true [A]
```

In (d), partitioning creates two entries: one for the `Boolean.TRUE` key (vowels) and one for the `Boolean.FALSE` key (consonants), regardless of its downstream collector. Since the string `"A"` is a vowel, it is accumulated in the list associated with the `Boolean.TRUE` key. The list associated with the `Boolean.FALSE` key remains empty. The output is the following:

```
false []
true [A]
```

This means that (a) resulted in only one entry in the map, while the other resulted in two identical entries.

16.11 *(d)*

It is important to note that the stream of strings is not processed separately from the stream of chars, but rather they are fused into a single stream pipeline. This is because only one terminal operation exists in the stream pipeline.

This means that the parallel processing applies to the entire pipeline. The `sort()` operation sorts the characters in the flattened stream, but the `forEach()` operation cannot be relied upon to respect the order, especially in a parallel stream. The `forEachOrdered()` operation will give a deterministic result regardless of the execution mode of a stream. The result from the program is therefore unpredictable.

16.12 *(c) and (d)*

The `filter()` method accepts a `Predicate`. The methods `peek()` and `forEach()` accept a `Consumer`. `map()` accepts a `Function`, `max()` accepts a `Comparator`, and `findAny()` does not accept any parameters.

16.13 *(b) and (f)*

The methods `peek()`, `map()`, `filter()`, and `sorted()` are all intermediate operations. The methods `forEach()` and `min()` are terminal operations.

16.14 *(b) and (d)*

Short-circuit methods may produce finite results for potentially an infinite stream. For example, the operations `limit()` and `anyMatch()` are short-circuit operations. A short-circuit operation terminates the stream pipeline, whether or not all elements in the stream have been processed.

16.15 *(f)*

These statements all perform an equivalent functional reduction of counting the number of elements in the stream. Empty string or `null` elements are still counted as elements. Thus all of these operations return the value `6`. Counting the number of elements in the stream can be achieved using the `count()` method of the `Stream` interface, or the `counting()` method provided by the `Collectors` class. Another solution is to map all stream elements to the value `1`, and then summing up the `1`s will give the same result.

16.16 *(b) and (d)*

In (a), a set of `String` objects is constructed that contains the values `"XX"`, `"XXXX"`, `""`, and `"X"` because the `filter()` method removes all `null` elements from the stream. Notice the absence of the duplicate values due to the fact that a `Set` does not allow duplicates. All strings of this set are then processed in another stream that maps the strings to `int` values according to the length of each string. This results in the output `0124`, because sorting of values is done before printing.

In (b), another set of `Integer` objects is constructed based on the same values. However, in this case all `null` elements and empty strings are converted to the `int` value of `0`, and then removed from the stream by the `filter()` method. All values in the result set are then processed in another stream that sorts the elements and prints the output `124`.

(c) applies similar logic to that in (b), except that it uses a collector that assembles the values in a `List` rather than a `Set`. Duplicate elements are allowed in lists, resulting in the output `1224`.

(d) is similar to (c), but it applies the `distinct()` operation to the stream elements, removing any duplicates, and resulting in the output `124`.

17 Date and Time

17.1 *(b)*

The first `LocalDate` object represents the date January 31, 2021, thus representing 31 days since the start of the year. The second `LocalDate` object is the result of adding exactly one month to the first `LocalDate` object. Since 2021 is not a leap year, it represents the date February 28, 2021, which is 59 days from the start of the year. The third `LocalDate` object is the result of subtracting one month from the second `Local-Date` object, resulting in the date January 28, 2021.

17.2 *(a)*

A `LocalDate` object is initially constructed to represent the date January 1, 2021. A new `LocalDate` object is then constructed based on this date, by first changing the day of the month to be 31, which is the last day of this month. Next, the month in this date is changed to February. It is important to remember that February in 2021 has only 28 days, so the resulting `LocalDate` object would have to represent the last day of February.

17.3 *(d)*

The `LocalDateTime` denoted by `d1` represents `2021-04-01T00:00`.

The method `toInstant()` converts `d1` by applying zone offset +18:00—that is, 18 hours ahead of the time at UTC.

To convert `d1` to an `Instant` denoted by `i1` at zero UTC offset, we must subtract 18 hours, resulting in the instant `2021-03-31T06:00:00Z`.

The `ofInstant()` method converts `i1` to `LocalDate`, but no offset adjustment is necessary to the date represented in `i1` as it represents a point in time on the UTC timeline.

17.4 *(c)*

Two `ZonedDateTime` objects are constructed exactly one hour apart. However, two new zoned date-time objects are create from these two, and the duration between them is calculated. These new zoned data-time objects are an extra hour apart because one subtracts and the other one adds 30 minutes, thus increasing the time difference to two hours between 23:30CET and 00:30GMT. Another way to view this is to convert the time in one time zone (23:30CET) to the other time zone (22:30GMT). The difference between 22:30GMT and 00:30GMT is two hours.

17.5 *(d)*

Both `Instant` and `LocalTime` can express values with nanosecond precision. The `between()` method of the `Duration` can be used to calculate the time difference between two temporal objects—that is, between objects of the classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `Instant`. The `between()` method of the `Period` can be used to calculate the date-based amount of time between two `Local-Date` objects.

17.6 *(c)*

A `LocalDateTime` object is created that represents `2021-04-01T08:15`. Thirty minutes are subtracted from this date-time object, returning a new date-time object. The day of the month is set to 12 for the resulting date-time object. However, the reference value of the final object is not assigned to any reference. `LocalDateTime` objects are immutable, thus the date-time object denoted by `dt` is never modified.

17.7 *(e)*

Unlike `Period`, when `Duration` is applied to a `ZonedDateTime` object, it disregards daylight savings.

A `Period` is a date-based amount of time (in terms of years, months, and days), and therefore cannot express an amount of time smaller than one day. A `Period` of one hour cannot be created.

Unlike `ZonedDateTime` objects, `LocalTime` and `LocalDateTime` objects have no time zone, so they do not take into consideration daylight savings.

A `Period` of one day may or may not be treated the same as a `Duration` of 24 hours, when using these objects with `ZonedDateTime` because of the differences in the handling of daylight savings.

Finally, both `Period` and `Duration` can express positive and negative amounts of time.

17.8 *(e)*

The `plus()` method of the `LocalDate` class returns a `LocalDate`. The `plus()` method of the `LocalDateTime` class returns a `LocalDateTime`.

17.9 *(d)*

The `plus()` method of the `LocalDate` class can accept a `Duration` object, but in this scenario it will throw an exception at runtime. The reason for this is that `Duration` is expressed in seconds and nanoseconds, which cannot be applied to a `LocalDate` object.

17.10 *(a) and (e)*

In order to compute the desired result, a time of 30 minutes and two days should be added to the given `LocalDate` object.

(a) combines a `LocalTime` object with a value of 30 minutes to the `LocalDate` object using the `atTime()` method, returning a `LocalDateTime` object. Then a duration of 48 hours is added to this `LocalDateTime` object.

(b) attempts to add a `Duration` of 48 hours to the `LocalDate` object, which will result in an `UnsupportedTemporalTypeException`. The reason for this is that `Duration` is expressed in seconds and nanoseconds, which cannot be applied to a `LocalDate` object.

(c) attempts to create a `LocalTime` object, with an amount of time greater than 23 hours, which is invalid for a `LocalTime` object, and will result in a `DateTimeException`.

(d) attempts to create a `LocalTime` object with a negative number of hours, which is also invalid, and would result in a `DateTimeException`.

(e) combines a value of 30 minutes to the `LocalDate` object using the `atTime()` method and returning a `LocalDateTime` object. Then a duration of 48 hours is added to this `LocalDateTime` object. The `atTime()` method is an overloaded method.

17.11 *(c)*

Five `LocalDate` objects are created in this example. The `atTime()` method creates a `LocalDateTime` object. The other five methods create a new `LocalDate` object.

17.12 *(d)*

The method `between()` calculates the amount of time that has elapsed between a `LocalTime` object and a `LocalDateTime` object. A `LocalTime` object is derived from the second argument, which is a `LocalDateTime` object. The result would have been a runtime exception if the two arguments had been interchanged: We cannot derive a `LocalDateTime` object from a `LocalTime` object.

The value of the first argument of the `between()` method represents a time that is after the time represented by the derived `LocalTime` object. Therefore, the resulting `Duration` object will have negative values. The time difference between 17:30 and 15:15 is two hours and 15 minutes.

[17.13](#) *(d)*

The required `LocalDateTime` object is exactly 25 hours (one day and one hour) ahead of the initial `LocalDateTime` object.

(a) adds one hour to the initial `LocalDateTime` object and changes the date to the next day.

(b) adds one day to the initial `LocalDateTime` object and changes the time by one hour.

(c) adds two days to the initial `LocalDateTime` object and subtracts 23 hours, which results in the required 25 hours being added to the initial `LocalDateTime` object.

(d) adds two days to the initial `LocalDateTime` object and subtracts 16 hours and 15 minutes, which results in the `LocalDateTime` object having the value `2021-04-03T23:15`.

(e) and (f) both adds a `Duration` of 25 hours (60 * 25 minutes) to the initial `Local-DateTime` object.

18 Localization

[18.1](#) *(a)*

For the French locale, the resource bundle for the default locale (US) is loaded, as there is no resource bundle file named `MyResources_fr_FR.properties`. The values of the keys in this resource bundle are printed. The key `"farewell"` has duplicates. The last value (`"Bye!"`) specified for this key is returned.

[18.2](#) *(b)*

The resource bundles loaded for `Locale.ENGLISH` are:

```
MyResources_en.properties
MyResources.properties
```

The method `getString()` returns the value `"Have a good one!"` for the key `"farewell"` in the resource bundle file `MyResources_en.properties`.

18.3 *(b)*

The code prints all available key–value pairs in the resource bundle for the English locale. The resource bundles loaded for the English locale are:

```
resources_en.properties
resources.properties (parent)
```

The key set contains the keys from both bundles: `k1` and `k2`. The key `k1` is found in the `resources_en.properties` bundle with the value `c`, and the key `k2` is found in the `resources.properties` bundle with the value `b`. Only if a key is not found in the current resource bundle, will it be looked up in its parent bundle, and so on. The `resource_en_GB.properties` bundle is not loaded by this code.

18.4 *(b)*

The code sets the default locale to the Russian locale, but then prints all available key–value pairs in the resource bundle for the English locale. The resource bundles loaded for the English locale are:

```
resources_en.properties
resources.properties (parent)
```

As appropriate bundles were found for the English locale, the default locale Russian bundle is not loaded.

The key set contains the keys from both bundles: `k1` and `k2`. The key `k1` is found in the `resources_en.properties` bundle with the value `c`, and the key `k2` is found in the `resources.properties` bundle with the value `b`. Only if a key is not found in the current resource bundle, will it be looked up in its parent bundle, and so on.

18.5 *(a), (b), (c), (d), and (e)*

The patterns produce the following output: Pattern Output

Click here to view code image

```
       Pattern      Output

  (a)  .00      |.46|
  (b)  .##      |.46|
  (c)  .0#      |.46|
  (d) #.00      |.46|
  (e) #.0#      |.46|
  (f) #.##      |0.46|
  (g)  .#0      Throws java.lang.IllegalArgumentException.
```

18.6 *(c)*

Notice that the code sets the number of decimal digits to two. First, the rounding mode is set to `HALF_UP`, which would round a double value of 9876.54321 to 9876.54. Then the same number is formatted again, but this time with the rounding mode set to `HALF_DOWN`, which would actually produce the same result. Because the third digit after the decimal point is 3 and not 5, it has no effect on the rounding by the `HALP_UP` or `HALF_DOWN` rounding mode.

18.7 *(d)*

Notice that the code sets the number of decimal digits to two. The value is a `Big-Decimal` and therefore is represented with a higher precision than it would be in a `double` value. The third digit after the decimal point is 5; therefore, the discarded fraction is 0.5. In this case, `HALF_DOWN` mode rounds up if the discarded fraction is > 0.5, which it is not, resulting in the formatted value `$9,876.54`. `HALF_UP` mode rounds up if the discarded fraction is >= 0.5, resulting in the formatted value `$9,876.55`.

18.8 *(d)*

The default short date format for the British locale is `dd/MM/YYYY`. However, this is not relevant in this case, because a `DateTimeFormatter` is used to format a `LocalDate` object, which has no time part, and thus results in an `UnsupportedTemporalType-Exception`.

18.9 *(h)*

A `DateTimeFormatter` object is configured to use a pattern, where `d` stand for the day of the month, `a` for an am/pm marker and `y` for a year. It is configured to use the UK locale. Notice that the time value in the `LocalDateTime` object is 14:30, which makes it pm.

18.10 *(e)*

In the code, the default locale is initially set to UK (British). A `DateTimeFormatter` is created to format a date according to the `MEDIUM` style format (`MMM d, yyyy`). This formatter formats the date to the string `"Apr 1, 2021"`. The reference `s1` denotes this string.

Next, the default locale is set to France. However, the `DateTimeFormatter` is immutable, and the date is formatted to the string `"Apr 1, 2021"`. The reference `s2` denotes this string.

Lastly, the `localizedBy()` method returns a new `DateTimeFormatter` object with the US locale, but it is not assigned to any reference. The date is again formatted by the previous `DateTimeFormatter` to the string `"Apr 1, 2021"`. The reference `s3` denotes this string. This means that only the condition in the first `if` statement is true.

18.11 *(d)*

`NumberFormat` interprets a floating-point number as a percentage, considering 1.0 to be equal to 100%. The default percentage format object rounds the value to two decimal digits, so it would round the double value `0.987654321` to `0.99`. The value `0.99` represents 99%.

18.12 *(a)*

The code creates a `LocalDateTime` object (`date1`) first and adds a London time zone to it to create a `ZonedDateTime` object (`date2`).

Next, two `DateTimeFormatter` objects, `df1` and `df2`, are created based on the same pattern `"hz"`, and the time zones for London (GMT) and Paris (CET) are associated with them, respectively. The `"hz"` pattern stands for hour and time zone. Both `date1` and `date2` are formatted by each `DateTimeFormatter`.

The `DateTimeFormatter df1` will format the hour in `date1`, but will supply the time zone, as this date object has no time zone, creating the string `"1GMT"`. It will format `date2` as `"1GMT"`, as the `ZonedDateTime date2` and the `DateTimeFormatter df1` have the London time zone.

The `DateTimeFormatter df2` will format the hour in `date1`, but will supply the time zone, as a date object has no time zone, creating the string `"1CET"`. It will format `date2` as `"2CET"`, as the `ZonedDateTime date2` and the `DateTimeFormatter df2` have different time zones. Therefore, the London time (`1h`) in `date2` is interpreted as Paris time (`2h`) by the `DateTimeFormatter`.

18.13 *(d)*

Values in single quotes are treated as verbatim text, rather than format elements. Only two values are actually supplied for the message pattern, so the format element with index 3 receives no value and will thus be formatted as text.

18.14 *(b)*

During the format operation, elements of the `values` array are interpreted by the `MessageFormat` and `ChoiceFormat` objects to determine the limit value that affects the selection of the appropriate choice format and the value that should be substituted into a message pattern. In this code, the value `4` in `values[0]` is the limit value, and the value `5` in `values[1]` is the value to format, resulting in the choice pattern `"{1}th"` at `formats[4]` to be chosen to format the value `5`. The formatted result is `"5th"`.

18.15 *(b)*

The limit values are given by the `limits` array. Note that the limit values are not ordered.

Click here to view code image

```
limits[0]   limits[1]   limits[2]
    0          -1           1
```

The corresponding choice formats are given by the `formats` array:

Click here to view code image

```
formats[0]  formats[1]  formats[2]
  "zero"     "negative"  "positive"
```

The supplied value `0.9` is greater than 0, and of course also greater than –1, but less than 1. It satisfies the following relation, determining index 1 in the `limits` array to choose the choice format.

Click here to view code image

```
limits[1] <= 0.9 < limits[2]
  -1      <= 0.9 <    1
```

Index 1 in the `formats` array determines the choice format to be the string `"negative"`.

The value `0.9` results in the choice format at index 1 in the `formats` array to be chosen.

18.16 *(a)*

The default locale is set to the US locale and a `DateTimeFormatter` object is created to format dates in the `MEDIUM` style format (`MMM d, yyyy`).

At (1), a string is parsed using the `dtf` formatter, according to the `MEDIUM` date format style and the default locale (in this case, US). The string is specified in the `MEDIUM` format style for the US locale.

At (2), a string is parsed to a date. As no formatter is specified, the string is expected to be in the ISO format, which it is.

At (3), the `LocalDate d` is formatted using the `dtf` formatter that uses the `MEDIUM` format style and the default locale (in this case, US).

18.17 *(c)*

The default locale is set to the US locale and a `DateTimeFormatter` object is created to format dates in the `SHORT` style format (`M-d-yy`).

At (1), a string is parsed to a date. As no formatter is used, the string is expected to be in the ISO format, which it is.

At (2), a string is parsed using the `dtf` formatter, according to the `SHORT` date format style and the default locale (in this case, US). The string is specified in the `SHORT` format style for the US locale.

Finally, this date is printed using the default ISO format.

18.18 *(d)*

The default locale can be defined explicitly; otherwise, it is the platform locale that is supplied by the runtime environment. The default locale is not necessarily the US locale. The default format for `LocalDate` objects is `ISO_DATE`.

19 Java Module System

19.1 *(b)*

Code in the module `ui` can access public types defined in the packages `store.fron-tend` and `store.backend` via direct dependency, but also public types defined in the package `product.data` via transitive dependency. However, module `ui` does not require module `customer`, thus it cannot access public types from the `customer.data` package. This means that (a) is false, but (b) is true.

Code in the module `customer` cannot access public types from the `product.data` package, despite the presence of transitive dependency via the modules `ui` and `store`. This is because module `customer` does not have a direct dependency on module `store`, which has a transitive dependency on module `product`. This means that (c) and (d) are both false.

Code in the module `product` cannot access public types from the `customer.data` package because of the absence of a dependency between these modules. This means (e) is false.

19.2 *(e)*

Only public types in the exported packages of a module are accessible to code in modules that require this module.

19.3 *(e)*

The `java.se` module is at the root of the module graph, as it depends on the highest number of modules in the graph. The `java.base` module is at the bottom of the module graph, and does not depend on any module. The `java.logging` module depends on the `java.base` module.

19.4 *(c) and (f)*

Only public types in the exported package `animals.primates` are accessible to code in module `zoo`.

19.5 *(a) and (d)*

Module `music` should declare a `requires` directive that should specify module `production`. Also, module `production` should export package `production.company`.

19.6 *(c) and (d)*

An automatic module is a plain JAR that is loaded from the module path. Plain JARs loaded from the class path are included in the unnamed module.

19.7 *(a)*

In (a), code in module `store` can access types defined in package `product.data` because this package is exported by module `product`, which module `store` actually requires. Code in module `store` can access types defined in package `market-ing.offers` by reflection as this package is opened by module `marketing`.

(b) is incorrect because code in module `marketing` cannot access types defined in packages `product.data` and `product.pricing` because it does not require module `product`.

(c) is incorrect because code in module `marketing` cannot access types defined in package `product.data` as it does not require module `product`. However, code in module `marketing` can access types defined in package `store.frontend`, as module `marketing` requires module `store`.

(d) is incorrect because code in module `store` can access types defined in package `product.data`, but not in package `product.pricing`, as this package is only exported to module `marketing`.

(e) is incorrect because code in module `product` cannot access types defined in package `store.frontend` as module `product` does not have a dependency on module `store`. However, code in module `product` can access types in the open package `marketing.offers`.

19.8 *(b)*

The `requires` directive specifies module names, not package names, which disqualifies (a) and (d). In (c), module `music` does not depend on module `artist`, and therefore cannot access types in package `artist.recoding`. This leaves (b), which works because module `music` has a direct dependency on module `production`, and also has a dependency on module `artist` via the `requires-transitive` directive in module `production`.

19.9 *(b)*

Service consumer module `player` does not need to declare any dependency on service provider module `brass`. Service consumer module `player` needs to declare a dependency on service module `music`, and specify which abstract type (in this case `music.sound.Instrument`) defines the service.

19.10 *(b) and (e)*

The `jlink` tool creates platform-specific runtime images that can be deployed. Runtime images contain application code, as well as the necessary JDK modules and tools, among other artifacts. However, no installation of a separate JVM is required to run the application.

19.11 *(a) and (b)*

Module `music` requires module `instrument`, which in turn requires module `music`. This results in a cyclic dependency, and thus is illegal and would cause these module declarations not to compile. Both modules `music` and `artist` export the same package `preferences.style`, which implies that this package is a split package, which is illegal and would cause this module declaration not to compile. It is allowed to declare the `opens` directive in a module declaration. It is also allowed to declare a qualified exports directive even if the specified module does not require this module.

19.12 *(e)*

Both modular and plain JARs can be used in the context of the class path as well as the module path. A listing of modules using the `--list-modules` of the `java` tool includes all observable modules, but not the unnamed module, as it has no name. The name of an automatic module is derived from the JAR file name, unless it is specified in the `MANIFEST.MF` file. There is only one unnamed module, and it obviously does not have a name, which makes the JAR file name irrelevant in this case.

19.13 *(c) and (d)*

An automatic module implicitly requires all other modules. An explicit module cannot access code in the unnamed module using the `requires` directive, since the unnamed module has no name. If an explicit module needs to access code in an automatic module, it must declare a `requires` directive to specify this automatic module.

20 Java I/O: Part I

20.1 *(d)*

The `read()` method will return `-1` when the end of the stream has been reached. Normally an unsigned 8-bit `int` value is returned (range from `0` to `255`). I/O errors result in an `IOException` being thrown.

20.2 *(d)*

The `print()` methods of the `PrintWriter` do not throw an `IOException` when the end of the file is reached, but instead sets an error status that can be checked.

20.3 *(d)*

The `read()` method of an `InputStreamReader` returns `-1` when the end of the stream is reached.

20.4 *(b)*

The `readLine()` method of a `BufferedReader` returns `null` when the end of the file is reached.

20.5 *(c)*

An `ObjectOutputStream` can write both objects and Java primitive types, as it implements the `ObjectInput` and the `DataInput` interfaces. The serialization mechanism will follow references in objects and write the complete object graph.

20.6 *(a), (b), (d), and (i)*

Static fields and `transient` instance fields are not serialized—they are treated the same way when it comes to serialization. The accessibility modifier `private` does not determine whether an instance field should be serialized or not. `Serializable` is a marker interface. Subclass objects are serializable if the superclass is serializable. The modifier `final` of a class does not determine whether the class is serializable or not.

20.7 *(b)*

(a) is incorrect because there is no requirement that all versions of a serializable class must provide a declaration of a `serialVersionUID`.

In (b), there is no guarantee that a streamed object based on one version can be deserialized based on the other, even if the two versions of the class have the same `serialVersionUID`. It depends on whether the changes in the class versions are compatible with deserialization. For example, changing the type of a field is an incompatible change.

(c) is incorrect because `serialVersionUID` in two unrelated serializable classes is also unrelated and need not be unique.

(d) is incorrect because there is no requirement that the `serialVersionUID` of a serializable class must be incremented every time a new version of the class is created.

(e) is incorrect because any class can declare a `static final` field of type `long` having the name `serialVersionUID`, but it has meaning for serialization only in a serializable class.

20.8 *(e)*

During deserialization, the zero-argument constructor of the superclass `Person` is called because this superclass is not `Serializable`.

20.9 *(c)*

If the superclass is `Serializable`, then the subclass is also `Serializable`—resulting in the printout in (c).

20.10 *(e)*

Note that only `GraduateStudent` is `Serializable`. The field `name` in the `Person` class is `transient`. During serialization of a `GraduateStudent` object, the fields `year` and `studNum` are included as part of the serialization process, but not the field `name`. During deserialization, the private method `readObject()` in the `GraduateStudent` class is called. This method first deserializes the `GraduateStudent` object calling the no-argument constructor in the superclasses, but then initializes the fields with new values. Without the private `readObject()` method, the output would be as in (d).

20.11 *(a)*

Constructors for `Product` and `Food` are triggered when a new `Food` object is created. These constructors print `"product food "`. After that, the `product` object is serialized. `Product` is a superclass of `Food` and is marked as `Serializable`, which implies that all of its subclasses, such as `Food`, are also serializable. Values of `Product` name and `Food` calories are included in the serialization of the `product` object.

No constructors are triggered during deserialization, thus `Product` and `Food` constructors do not print any values. No errors are triggered during deserialization, and values of `Product` name and `Food` calories are restored and printed.

20.12 *(c)*

A buffer (a `char` array of length 4) is used to read and write characters to files. In the `while` loop, this buffer is filled with characters from the `test1.txt` file and written to the `text2.txt` file. On the first iteration, the characters `a`, `b`, `c`, and `d` are read from the `text1.txt file`, filling the buffer to full capacity, and then written to the t`est2.txt` file. On the second iteration, the remaining characters `e`, `f`, and `g` are read from the `test1.txt` file into the buffer. These characters are read into the first three elements of the buffer. The fourth element in the buffer still contains the character `d` from the previous read operation. The buffer contains the characters `e`, `f`, `g`, and `d`, which are written to the `text2.txt` file. The next read operation returns `-1` since the end of the file has been reached in the `text1.txt` file, thereby terminating the loop.

20.13 *(c)*

Both fields `numberOfTracks` and `currentTrack` are not included when an `Album` object is serialized, as the field `numberOfTracks` is `static` and the instance field `current-Track` is `transient`. Only the instance field `title` (having the value `"Songs"`) is included in the serialization of the `Album` object.

The `readObject()` method of the `Album` class is not `private`, but `public`, and is never called during deserialization to change the state of the `Album` object created at deserialization.

The `Album` object created at deserialization is initialized with the instance field `title` having the value `"Songs"` and the `transient` field `currentTrack` initialized to the default `int` value `0`.

Deserialization requires definition of the class, thus an `Album` object created at deserialization can access the `static` field `numberOfTracks` in its class that has the value `5`.

21 Java I/O: Part II

21.1 *(b) and (c)*

Compiling and running the program results in the following output:

Click here to view code image

```
/wrk/./document/../book/../chapter1
/wrk/chapter1
chapter1
./document/../book/../chapter1
./document/../book/..
```

Note that only the `Path.toRealPath()` method requires that the file exists; otherwise, it throws a `java.io.IOException`.

21.2 *(c)*

Compiling and running the program results in the following output:

```
./wrk/src

./wrk
./wrk/src
./wrk/src/readme.txt

./wrk
./wrk/src
./wrk/src/readme.txt
```

The `Files.list()` method creates a stream based on the *immediate* entries of the directory path passed as a parameter. The `Files.walk()` method traverses depth-first every entry in the hierarchy of the directory passed as a parameter. The `Files.find()` method will find every entry in the hierarchy of the directory passed as a parameter, since the matcher argument is always `true` and will traverse to depth 2 (i.e., equal to the actual depth of the directory).

21.3 *(a)*

There are three absolute `Path` objects (starting at the root (/) of the file system) constructed in this example: `Path earth` is defined as `"/planets/earth"`.

`Path moonOrbit` is defined as `"/planets/earth/moon/orbit.param"`—that is, as a child of `Path earth`.

`Path mars` is defined as `"/planets/mars"`—that is, as a sibling of `Path earth`.

There is one relative path:

`Path fromMarsToMoon` is defined as `"../earth/moon/orbit.param "`—that is, as a relative path between `Path mars` and `Path moonOrbit`.

These `Path` objects do not have to actually exist in the file system, so long as a program makes no attempt to validate or access these paths. Thus no runtime exception will be thrown.

21.4 *(b)*

First, a `Path` object is created with the relative path `"./mars/../earth"`, that has four name elements. Next, this `Path` is normalized, resulting in a `Path` object with the path string `"earth"` that has one name element. Then it is converted to the absolute path `"/planets/earth"` which has two name elements. If this path had not have been normalized, then the absolute path would be `"/planets/./mars/../earth"`, which has five name elements. Whether the paths exists in the file system is irrelevant, since this program makes no attempts to actually validate or access any of these paths.

21.5 *(d)*

The `list()` method of the `Files` class creates a stream of `Path` objects denoting the immediate entries in the given directory. Unlike the method `walk()`, the `list()` method does not traverse the contents of the subdirectories. The stream created by the `list()` method will include the `Path` objects that denote the following paths:

```
/test/a.txt
/test/c
/test/e.txt
/test/f.txt
```

A filter is applied to this stream of `Path` objects, which uses the method `getFileName()` of the `Path` interface to return the last name element of the `Path`. The filter will discard any entry whose file name does not end with `"txt"`. This leaves only the following paths in the stream:

```
/test/a.txt
/test/e.txt
/test/f.txt
```

These paths are then printed to the console.

21.6 *(g)*

The `walk()` method of the `Files` class creates a stream of `Path` objects that denote all entries in the given directory, including its subdirectories, by traversing the directory hierarchy depth-first. The stream will include `Path` objects that denote the following paths:

```
/test/a.txt
/test/a.txt/b.txt
/test/c
/test/c/d.txt
/test/e.txt
/test/f.txt
```

A map operation is applied to this stream of `Path` objects, converting it to a stream of `String` objects, where each `String` represents the last name element of the path— that is, the file name:

```
a.txt
b.txt
c
d.txt
e.txt
f.txt
```

A filter is applied to this stream of `String` objects which excludes strings that do not end in the file extension `"txt"`. This leaves only the following paths in the stream (notice that these strings are sorted):

```
a.txt
b.txt
d.txt
```

```
e.txt
f.txt
```

These `String` objects are then printed to the console.

21.7 *(b)*

The method `Files.createDirectories()` does not throw an exception when trying to create a directory that already exists. However, the method `Files.createDirectory()` does.

Method `Files.delete()` does throw an exception when trying to delete a nonexistent directory.

Method `Files.move()` does throw an exception when moving a non-empty directory, but only if it actually needs to move all files within this directory to another file system. Moving a directory within the same file system does not actually perform any move operations for the directory. It only changes the path of the directory. Method `Files.exists()` returns a `boolean` value to indicate the existence or nonexistence of a path.

21.8 *(c)*

Two `Path` objects are initialized. However, both of these `Path` objects reference the same file. This is because `p1.getName(1)` returns a relative path to directory `joe`, which is the second component in this path, considering that the first component is directory `users` with index 0. This path is then used to resolve another relative path `test/a.jpg`, which results in the path `joe/test/a.jpg`. And finally, the path `/users` is used as a root directory to resolve this path, resulting in the path `/users/joe/test/a.jpg`, which is identical to the path referenced by `p1`.

Next, an attempt is made to move the entry at path `p1` to path `p2`, where both denote the same directory entry. In this scenario, `Files.move()` method performs no action because it can detect that both the source and the destination path are the same.

21.9 *(c)*

In this example, the `p1` reference represents an absolute path, and the `p2` reference represents a relative path. Keep in mind that an absolute path starts from the root of a file system, in this case designated by the slash character (`/`).

The purpose of the method `resolve()` is to construct a path where a relative path is appended to another relative path, or to an absolute path. `p1.resolve(p2)` results in the relative path `store` being appended to the absolute path `/test`. As it is not possible to append an absolute path to another path, `p2.resolve(p1)` returns the absolute path `/test` denoted by the reference `p2`.

21.10 *(d)*

This question assumes the existence of the destination file, yet this code example does not specify the replace-existing file copy option:

Click here to view code image

```
Files.copy(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

Therefore, the code will throw a `java.nio.file.FileAlreadyExistsException`.

21.11 *(d)*

The code reads lines of text from a file as a stream of strings. A filter operation discards lines that do not start with `<`. Then each line in the stream is mapped by the map operation to a `>`. The reduction operation is applied to concatenate each `>`. The resulting string `">>>>>"` is printed as there are five lines that are mapped to `>`.

21.12 *(d)*

Only permissions explicitly added to the set are applied, all other permissions are removed. In order to be able to access files inside a directory, the directory needs to have execute permission. However, the permissions for the `/test/data` directory are changed to read-only in the code. Therefore, an attempt to access the `info.txt` file in this directory by the `walk()` method will throw an `AccessDeniedException`, that will terminate the stream processing.

21.13 *(d)*

The thing to note is that a `PosixFileAttributeView` can be used to set permissions for the file that is associated with it. However, the `PosixFileAttributes` object obtained from the `PosixFileAttributeView` will only reflect the file attribute values at the time it was obtained from the view. It is not updated automatically when the file attribute values change in the file. A new `PosixFileAttributes` object must be obtained from the view to reflect any changes in the file attribute values.

The program first removes all permissions from the file. A `PosixFileAttributeView` is created on the file, and the `PosixFileAttributes` object associated with the view is obtained. This `PosixFileAttributes` object is used in the rest of the program, and it will always reflect that the file has no permissions, regardless of any permissions set in the file through the view.

Note also that removing an element (`OWNER_WRITE` permission) from an empty set does not throw an exception. The permissions of the file are changed as follows, but the changes are not reflected by the `PosixFileAttributes` object:

```
---------
r---w-r--
-w-------
```

21.14 *(a)*

First, this program converts a `URI` object to a `Path` object. Path `p1` references the same file as path `p2` . No action is taken by the `copy()` method when the source file and the destination file are the same. Lastly, a `Path` object is converted to a legacy `File` object. This program runs successfully and produces no output.

22 Concurrency: Part I

22.1 *(e)*

The program will compile without errors, and will simply terminate without any output when run. Two thread objects will be created, but they will never be started. The `start()` method must be called on the thread objects to make the threads execute the `run()` method asynchronously.

22.2 *(d)*

Note that calling the `run()` method on a `Thread` object does not start a thread. In the statement:

Click here to view code image

```
new Thread(new R1(),"|R1a|").run();
```

the `run()` method of the `Thread` class will invoke the `run()` method of the `Runnable` object (`R1`) that is passed as an argument in the constructor call. In other words, the `run()` method of the `R1` class is executed in the `R2` thread—that is, the thread that called the `run()` method of the `Thread` class and whose name will be printed. However, the statement:

Click here to view code image

```
new Thread(new R1(),"|R1b|").start();
```

starts the `|R1b|` thread, and the `run()` method of the `Thread` class will invoke the `run()` method of the `Runnable` object (`R1`) that is passed as an argument in the constructor call, but it is executed by the `|R1b|` thread whose name will be printed. The last statement in the `run()` method of the `R2` class is executed by the `|R2|` thread whose name will be printed.

22.3 *(c)*

Note that the complete signature of the `run()` method does not specify a `throws` clause, meaning it does not throw any *checked* exceptions. However, a method can always be implemented with a `throws` clause that specifies *unchecked* exceptions, as in the case of the `run()` method.

22.4 *(a) and (e)*

Because the exact behavior of the thread scheduler is undefined, the text `A`, `B`, and `End` can be printed in any order. The thread printing `B` is a daemon thread, which means that the program may terminate before the thread manages to print the letter.

22.5 *(a) and (e)*

The lock is also released when an uncaught exception occurs in the statement.

22.6 *(c) and (d)*

First note that a call to `sleep()` does not release the lock on the `Smiley.class` object once a thread has acquired this lock. Even if a thread sleeps, it does not release any locks it might possess.

(a) does not work, as `run()` is not called directly by the client code.

(b) does not work, as the infinite `while` loop becomes the critical region and the lock will never be released. Once a thread has the lock, other threads cannot participate in printing smileys.

(c) works, as the lock will be released between each iteration, giving other threads the chance to acquire the lock and print smileys.

(d) works for the same reason as (c), since the three print statements will be executed as one atomic operation.

(e) may not work, as the three print statements may not be executed as one atomic operation, since the lock will be released after each print statement. Synchronizing on `this` does not help, as the printout from each of the three print statements executed by each thread can be interspersed.

22.7 *(d)*

A thread terminates when the execution of the `run()` method ends. The call to the `start()` method is asynchronous—that is, it returns immediately, and it moves the thread to the *READY* substate. Calling the `sleep()` or `wait()` method will block the thread.

22.8 *(b) and (d)*

The nested `createThread()` call is evaluated first, and will print `23` as the first number. The last number the main thread prints is `14`. After the main thread ends, the thread created by the nested `createThread()` completes its `join()` call and prints `22`. After this thread ends, the thread created by the outer `createThread()` call completes its `join()` call and prints the number `12` before the program terminates.

22.9 *(e)*

The exact behavior of the scheduler is not defined. There is no guarantee that a call to the `yield()` method will grant other threads use of the CPU.

22.10 *(b)*

The `final` method `notify()` is defined in the `Object` class.

22.11 *(c)*

An `IllegalMonitorStateException` will be thrown if the `wait()` method is called and the current thread does not hold the lock of the object.

22.12 *(d)*

Since the two methods `emptying()` and `filling()` are `synchronized`, only one operation at a time can take place on the tank that is a shared resource between the two threads.

The method `emptying()` waits to empty the tank if it is already empty (i.e., `isEmpty` is `true`). When the tank becomes full (i.e., `isEmpty` becomes `false`), it empties the tank and sets the condition that the tank is empty (i.e., `isEmpty` is `true`).

The method `filling()` waits to fill the tank if it is already full (i.e., `isEmpty` is `false`). When the tank becomes empty (i.e., `isEmpty` becomes `true`), it fills the tank and sets the condition that the tank is full (i.e., `isEmpty` is `false`).

Since the tank is empty to start with (i.e., `isEmpty` is `true`), it will be filled first. Once started, the program will continue to print the string `"filling"` followed by the string `"emptying"`.

Note that the `while` loop in the `pause()` method must always check against the field `isEmpty`.

23 Concurrency: Part II

23.1 *(c)*

A single thread executor service does not allow scheduling of tasks, but a scheduled executor service allows a task to be scheduled with a specified delay, and also allows a task to be executed periodically. The work stealing mechanism is specific to the work stealing thread pool, and the work stealing thread pool is designed to maintain enough threads to support a given level of parallelism.

23.2 *(b)*

The `shutdown()` method of the executor service initiates the shutdown of the executor service, allowing currently running tasks to continue, but preventing new tasks from being submitted. It does not wait for the termination of currently running tasks. The `awaitTermination()` method can throw an `InterruptedException`. The `shutdownNow()` method also initiates the shutdown of the executor service, but it cancels all running tasks and returns.

23.3 *(b)*

The read lock does not prevent other operations from reading data. Methods that acquire read and write locks can throw an `InterruptedException`. The write lock is designed to allow only a single exclusive write operation on the data, preventing other read and write operations to be performed, thus preserving memory consistency and preventing data corruption.

23.4 *(b)*

The contents of `list1` are created by spawning a number of threads, concurrently writing data to the copy-on-write list. There is no guarantee that these threads would actually complete copying all elements from the initial list by the time `list1` is printed.

The contents of `list2` are created by processing elements from the initial list using a parallel stream with a list collector reduction operation combining processed data into a single list. The `collect()` terminal operation ensures that the stream is exhausted and the collector ensures that all data is assembled into the list. When printed, `list2` has the same contents as the initial list.

The contents of `list3` are created by processing elements from the initial list using a parallel stream, but manually adding elements into `list3`. This does not guarantee the consistency of `list3` because of potential contention between threads trying to access `list3`, and is likely to corrupt data.

23.5 *(d)*

Synchronized collections provide blocking (synchronized) methods that modify collection content. However, synchronized collections do not provide a synchronized iterator. It is the programmer's responsibility to implement synchronized iteration behavior for such collections.

Copy-on-write collections achieve concurrency by creating a copy of a collection for each thread that tries to modify the collection, and then automatically merging these copies without any need to implement thread synchronization. Immutable collections are read only, and thus are automatically considered to be memory safe, without any need for synchronization.

23.6 *(d)*

Atomic variables are designed to be thread-safe without the use of synchronization and intrinsic locking. Methods provided for atomic variables do not throw an `InterruptedException`.

23.7 *(a) and (b)*

The Atomic API provides operations that guarantee object consistency. However, no specific order is enforced when a number of atomic operations are performed concurrently. In this example, a number of `incrementAndGet()` calls are executed on an `AtomicLong` object, resulting in the consistent increment of its value. This means that the last value in this example would always be `3`. The order of the `Future` objects is the same as that of the invoked tasks, but the stream iterating through the list of these `Future` objects can print the numbers 1, 2, and 3 in any order, as the order of the concurrent increment operations is unpredictable.

23.8 *(a)*

In this example, an attempt is made to upgrade a read lock to a write lock, which is not possible. It should be noted that a write lock can be downgraded to a read lock. Consider the following scenarios: Attempting to obtain the read lock using the `lock()` method after the write lock has been acquired is allowed:

```
writeLock.lock();
readLock.lock();
```

Attempting to obtain a write lock using the `lock()` method after the read lock has been acquired will not succeed:

```
readLock.lock();
writeLock.lock();
```

Attempting to obtain a write lock using the `tryLock()` method after the read lock has been acquired will return `false`—that is, the write lock is not acquired:

```
readLock.lock();
writeLock.tryLock();
```

In the `finally` block, the `isWriteLocked()` method checks whether the write lock has been acquired, but we have already established that this would not be the case. So only the `"Read lock acquired"` and `"The end"` messages will be printed in this scenario.

23.9 *(d)*

In this example, the variable `counter` is declared as `volatile`. A `volatile` variable guarantees visibility of write operations. It does not guarantee memory consistency when several concurrent threads attempt to modify this volatile variable with a non-atomic operation (`--`). There is a danger of interleaving of read and write operations on the variable by different threads; thus the results are unpredictable.

23.10 *(e)*

The submit() method does not throw an exception. It is possible to submit both a `Callable` (`() -> "acme"`) that returns a value and a `Runnable` (`() -> {}`) that does not. The `shutdown()` method does not throw an exception. An invocation of the `shutdown()` method initiates the shutdown of the executor service, but the two already submitted tasks are allowed to complete. However, the `shutdown()` method does not wait for the tasks to complete execution.

Although the `get()` method can throw checked exceptions, no exceptions are thrown in this case. Invocation of the `get()` method on a `Future` blocks until the task represented by the `Future` completes execution. The first `get()` method call returns the result of executing the `Callable`, which in this case returns the string `"acme"`. Since a `Runnable` does not return a value, the second `get()` method call returns the `null` value to indicate normal completion of the task represented by the `Runnable`. The print statement does not throw an exception. It prints `"acme null"`, which is the concatenation of the results `"acme"` and the `null` value returned by the `get()` methods, respectively.

23.11 *(f) and (h)*

In the `for(;;)` loop, there is no guarantee that a task will actually be cancelled before it completes. Cancelled or not, each task is added to the `results` list. The `shutdown()` method initiates the shutdown of the executor service, allowing those tasks that are already running to complete execution.

The method `isDone()` only returns `true` if the task completed due to normal termination, an exception, or cancellation. If any task was still running, the `allMatch(r>r.isDone())` expression will return `false`, causing the letter `Z` to be printed.

If all tasks completed, then there could be some among them that were cancelled. An attempt to get the value of a `Future` whose task has been cancelled will result in an exception. In order to concatenate the values returned by the tasks in the `Future` objects, all cancelled tasks are filtered from the stream by calling the `isCancelled()` method. Since it is unpredictable which tasks were cancelled and which terminated normally, the output from the program may contain any of the letters `A`, `B`, `C`, `D`, or `E`.

24 Database Connectivity

24.1 *(c)*

When no rows are returned by the query, invoking the `next()` method simply returns `false` to indicate the absence of the next row in the result set. This is considered normal behavior and does not cause an exception, ruling out (a) and (b). The first row in the result set has index 1.

24.2 *(c)*

Programmers should ensure that result sets are closed first, then statements, and only then the connection. The closure order is important because unclosed result sets and statements can cause a memory leak on the database side. This has nothing to do with the transactional behavior of the program, and thus is not related to auto-commit mode.

24.3 *(b)*

In this code example, the marker parameter in the select statement is set to match rows that start with the string `"Where"`. The table contains exactly two rows that match the where clause, so these rows will be retrieved and printed by the program. When retrieving a column value, the column might not have a value—that is, it might be `null`. Testing a reference for `null` before using the reference avoids the `NullPointerException` at (3).

24.4 *(b)*

Notice that in this code example, the value `103` that is set for the `id` column in the query does not match any id in the rows in the table. Therefore, when executed, this query will not return any rows. This is not an error, so it would not cause any exceptions. Instead, the method `next()` will return `false`, as the result set is empty. As a result, the body of the `if` statement will not be executed and nothing will be printed.

24.5 *(b)*

First note that the auto-commit mode has been disabled for the connection. Next, this program sets the marker parameters and executes an update statement. However, when it sets the marker parameter for the select statement, it uses index 2, but there is only one marker parameter in this select statement, so the line of code `ps1.setInt(2, id)` will throw an exception. This will interrupt normal program execution and control will be transferred directly to the `catch` block. This means that the explicit commit statement will not be executed.

However, the `catch` block does not attempt to roll back this transaction, so once the exception is handled by the `catch` block, the program will resume its normal execution, which will correctly close the statements and the connection in the implicit `finally` block of the outer `try`-with-resources statement. Therefore, the database will not have any indication that it is supposed to perform a rollback, as no rollback is executed in the `catch` block. Its reaction to a normal connection closure is to commit any outstanding changes.

24.6 *(b)*

The resources will be closed in the following order: result set, statement, and connection.

24.7 *(b)*

Marker parameters in a prepared statement are set with the `setXXX()` methods, not with the `executeQuery()` method, which rules out (a). Each prepared statement represents a single SQL statement, which can be parameterized and executed multiple times, contradicts (c) and (d).

24.8 *(a) and (b)*

The SQL query in this example selects rows from the `questions` table, using a `where` clause that selects rows that do not have any value (`null`) for the `answer` column. Then the code iterates through the result set containing these rows and updates the answer column value to be `"no answer"`.

(c) is incorrect because the `SELECT` statement will return some, but not necessarily all, rows from the `questions` table. (d) is incorrect because potential exceptions will be caught by the `catch` block, which does not invoke the `rollback()` method. Therefore, once an exception is caught, the program will resume normal execution and will correctly close the connection in the implicitly `finally` block of the outer `try-` with-resources statement. Databases assume that if a program has correctly closed its JDBC connection, then there is no reason not to commit any outstanding changes made within the context of this connection.

24.9 *(c)*

There are two problems with the code.

First, only one of the marker parameters is actually set before the statement is executed.

And second, an update SQL operation cannot be executed using the `executeQuery()` method.

Either one of these issues would cause the `executeQuery()` method to throw an exception.

24.10 *(a)*

Marker parameters in a prepared statement need not be set in any specific order, as long as they are all set before the statement is executed. In the given code, all marker parameters are set before the statement is executed, so the code executes normally.

24.11 *(b), (c), and (d)*

A prepared statement can be executed multiple times, making (b) a correct option. If a statement is a `SELECT` statement, then it can be executed using either the `execute()` or `executeQuery()` methods; otherwise, it can be executed using the `execute()` and `executeUpdate()` methods, making (c) and (d) correct options. Statements executed by the `executeQuery()` method and the `executeUpdate()` method are mutually exclusive—the former executes `SELECT` statements and the latter non-`SELECT` statements like `INSERT`, `UPDATE`, and `DELETE`. The `execute()` method can execute all statements.

24.12 *(a)*

The default navigation direction in a result set is forward only, meaning starting with the first row and successively proceeding to the last row each time the `next()` method is called. Forward-only navigation is supported by all databases. Other result set options, such as reflection of changes, scroll sensitivity, and cursor closure on commit, may or may not be supported by different databases.

24.13 *(a)*

The `relative(int rows)` method moves the cursor a specified number of rows in relation to the current row in the result set. The parameter value can be a positive or a negative `int` value.

Calling the method `relative(1)` moves the cursor forward by one row, which is the same as calling the method `next()`.

Calling the method `relative(0)` does not move the cursor from its current position. Calling the method `relative(-1)` moves the cursor backward by one row, which is the same as calling the method `previous()`.

Calling the method `absolute(1)` moves the cursor to the first row in the result set, which is the same as calling the method `first()`.

Calling the method `absolute(0)` moves the cursor to before the first row in the result set.

Calling the method `absolute(-1)` moves the cursor to the last row in the result set, which is the same as calling the method `last()`.

25 Annotations

25.1 *(a) and (b)*

Annotations are compiled into classes just like any other classes or interfaces. The purpose of an annotation is to provide metadata for program elements in the code (like Java classes, interfaces, methods, and variables). Annotations can be applied to other annotations, and can also be used as a type of an annotation element.

25.2 *(c)*

Annotations having the target `ElementType.TYPE` can be applied to classes, interfaces, enums, and other annotations. Annotations having the target `Element-Type.TYPE_PARAMETER` can be applied to type parameters in generic code. Annotations having the target `ElementType.FIELD` can be applied to constants, as constants are static fields. Annotations having the target `ElementType.METHOD` can be applied to only methods, but annotations having the target `ElementType.CONSTRUCTOR` can be applied to constructors.

25.3 *(a) and (c)*

The annotation in question is applied to the class, and requires no parameters to be supplied. This means that its target must be either default, or explicitly declared to be applicable to `ElementType.TYPE`. This would exclude (b) and (d).

25.4 *(b) and (g)*

The annotation type declaration defines a single element type of `int` array called `value`, and provides a default value for this element. This question asks to identify incorrect ways of applying this annotation. (a) and (f) supply a single value for this annotation element, in which case no `{}` are required to enclose the value, and it is not required to specify the element name `value` when only one value is specified. (c) is legal because the element name `value` does not have to be specified when it is the only element specified in the annotation type declaration. (d) and (e) are legal because the `value` element does not have to be set, since it has a default value, and parentheses `()` are optional when no value is specified. (b) and (g) are illegal because they are missing the block notation `{}` to enclose the list of values specified.

25.5 *(b)*

The annotation type `Test5Annotation` should be defined to be applicable to at least the targets of `TYPE`, `TYPE_PARAMETER`, and `FIELD`. It should also define an element named `value` (and not `values`), whose type should be a `String` array and have a default value.

25.6 *(d)*

Annotations are not reflected in the documentation of the class in which they are applied, unless the `@Documented` meta-annotation is applied to its annotation type.

25.7 *(b) and (e)*

Default values are not mandatory for annotation type elements, and a default value cannot be `null`. An annotation element of an array type can be assigned a single default value, in which case it does not need to be enclosed in block notation `{}`. Annotation element names must be specified as method names with no parameters, but mandatory parentheses `()`.

25.8 *(a), (b), and (c)*

The annotation type can be applied to Java types—that is, classes, enums, and interfaces. It can also be applied to fields and constructors.

(a) applies annotation to an interface, which is valid because it is defined as applicable to a type. It provides a string value for the `value()` element, and it does not set any other elements, relying instead on their default values. This means it does not have to explicitly qualify the element `value()` name.

(b) applies annotation to enum fields, which is allowed by this annotation definition. It provides both `value()` and `details()` element values for the first three fields, and relies on the default value of the `details()` element for the fourth field.

(c) applies the annotation to a field, explicitly qualifying the element `value` and relying on the default value of the `details()` element.

(d) applies annotation to a class, which is allowed by this annotation type. However, it does not qualify the name of the `value()` element, which must be qualified when it is not the only element specified.

(e) applies annotation to a method, which is not allowed by this annotation type.

(f) applies annotation to an expression (i.e., in a *type context*), which is not allowed by this annotation type.

25.9 *(b)*

The `Containee` annotation type is defined as a repeatable annotation type. Thus it can be used as an array type of the mandatory `value()` element declared in the `Container` annotation type. However, in order for this construct to work, any other elements specified in the `Container` annotation type must be declared with default values.

25.10 *(c)*

The way in which the annotation `@Folder` is applied to the `Storage` class requires the- `Folder` annotation type to be repeatable—that is, its type declaration must be marked with the `@Repeatable` meta-annotation having the argument `Folders.class`. Because of the way the `@Folder` annotation is applied to the `Storage` class, its type declaration must define two elements: a `value()` element and a `temp()` element. The `temp()` element of the `Folder` annotation type must be defined with a default value. The container annotation type `Folders` must define a `value()` element of type `Folder[]`.

26 Secure Coding

26.1 *(c)*

Data obfuscation is considered to be an important measure for securing sensitive information. Sanitizing input values is a countermeasure that helps preventing code injections. Encapsulation helps to prevent code corruption. Terminating recursive data references helps to prevent a type of denial-of-service attack that attempts to cause a program to start an infinite data processing loop.

26.2 *(d)*

Encapsulation is a software design strategy that only allows access to an object’s state through specifically defined operations. Mutable objects are those whose state can be modified. Code corruption is a category of threats that can be used to corrupt application logic. A code injection allows executable code to be passed as an input parameter.

26.3 *(d)*

Normally an addition of 1 to a maximum value of a primitive type would result in the value wrapping around to the minimum value. The maximum value of the `int` type is `2147483647` and the minimum value is `-2147483648`. However, in this case, the addition is performed by the `addExact()` method, which actually checks value boundaries and will throw an `ArithmeticException` to prevent the value wrapping around.

26.4 *(c)*

The Secure Hash Algorithm (SHA) is designed to produce a fixed-length result, known as a message digest, from a variable-length input. The number 256 is the length of the digest—that is, the result produced by the hash algorithm.

26.5 *(a)*

Java security policies are specified as a grant, defining restrictions and permissions on code execution and on access to resources.

26.6 *(c)*

Any checked exceptions thrown within the `PrivilegedAction.run()` method must be handled within this method. Alternatively, if checked exceptions need to be propagated outside the `run()` method, then the `PrivilegedExceptionAction` interface should be implemented instead.