Appendix F

Annotated Answers to Mock Exam I

This appendix provides annotated answers to the questions in the mock exam for the *Java SE 8 Programmer I* certification found in Appendix E.

Annotated Answers

Q1 *(a), (b), (c), and (e)*
Only local variables need to be explicitly initialized before use. Fields are assigned a default value if not explicitly initialized.

Q2 *(e)*
Giving parameters `(2, 3)` to the method `substring()` constructs a string consisting of the characters from index 2 (inclusive) to index 3 (exclusive) of the original string; that is, the string returned contains the character at index 2. The first character is at index 0 and the last character is at index 1 less than the number of characters in the string.

Q3 *(c)*
The loop prints out the values `12`, `9`, `6`, and `3` before terminating. The loop terminates when the value in the loop variable `i` becomes less than or equal to 0. This happens after the value `3` has been printed.

Q4 *(e)*
The fact that a field is static does not mean that it is not accessible from non-static methods and constructors. All fields are assigned a default value if no initializer is specified. Static fields are initialized when the class is loaded, and instance fields are initialized when the class is instantiated. Only local variables must be explicitly initialized before use.

Q5 *(e)*
An object of the class `Extension` is created. The first thing the constructor of `Extension` does is invoke the constructor of `Base`, using an implicit `super()` call. All calls to the

method `void add(int)` are dynamically bound to the `add()` method in the `Extension` class, since the actual object is of type `Extension`. Therefore, this method is called by the constructor of `Base`, the constructor of `Extension`, and the `bogo()` method with the parameters `1`, `2`, and `8`, respectively. The instance field `i` changes value accordingly: `2`, `6`, and `22`. The final value of `22` is printed.

Q6 *(d)*

At (1), a new `Widget` object is constructed with the message that is a concatenation of the message `"bye"` in the `Widget` object denoted by `b` and the string `"!"`. After line (2), `d` and `b` are aliases. After line (3), `b` and `a` are aliases, but `d` still denotes the `Widget` object with the message `"bye"` from line (0). After line (4), `d` and `a` are aliases. Reference `d` no longer denotes the `Widget` object created in line (0). This `Widget` object has no references that refer to it and, therefore, is a candidate for garbage collection.

Q7 *(d) and (e)*

`String` objects are immutable. None of the methods of the `String` class modifies a `String` object. Methods `toUpperCase()` and `replace()` in the `String` class will return a new `String` object that contains the modified string. However, `StringBuilder` objects are mutable. The `charAt()` method of the `StringBuilder` class is a get method, returning the character at a specific index, without modifying the contents of the `StringBuilder` object.

Q8 *(c)*

Statement (a) will execute without problem, but (b), (d), and (e) will cause compile-time errors. Statements (b) and (e) will cause compile-time errors because it is not possible to convert from the superclass `A` to the subclasses `C` and `B`, respectively. Statement (d) will cause compile-time errors because a cast from `B` to `C` is invalid. Being an instance of `B` excludes the possibility of being an instance of `C`. Statement (c) will compile, but will throw a runtime exception because the object that is cast to `B` is not an instance of `B`.

Q9 *(b), (c), (d), and (g)*

In (a), a `String` is neither an array nor an `Iterable`. The method `toCharArray()` of the `String` class returns an array of type `char`. A `char` value is assignable to the local variable of type `char` in (b), and after autoboxing, assignable to the local variable of type `Character` in (c) and (d). In (e), the local variable `str` is redeclared. In (f), the occurrence of the array operator `[]` is not permitted. In (g), the array `strs` is permissible in the `for(:)` loop. In (h), the `String` class does not have a method named `toArray`, but it has a method named `toCharArray`.

Q10 *(b) and (e)*

For the expression `matrix[3][2]` to access a valid element of a two-dimensional array, the array must have at least four rows and the fourth row must have at least three elements. (a) produces a 2×3 array. (c) tries to initialize a two-dimensional array as a one-dimensional array. (d) tries to specify array dimensions in the type of the array reference declaration.

Q11 *(a)*

The expression `arr.length` will evaluate to 4. The expression `arr[1]` will access the array `{ { "1", "2" }, { "1", null, "3" } }`, and `arr[1][2]` will try to access the third

element of this array. This results in an `ArrayIndexOutOfBoundsException`, since this array has only two elements.

Q12 *(c) and (d)*

`String` objects can have identical sequences of characters. The `==` operator, when used on `String` object references, will return `true` if and only if both references denote the same object (i.e., are aliases). The `equals()` method will return `true` whenever the contents of the `String` objects are identical. An array of `char` and a `String` are two totally different types, and when compared using the `equals()` method of the `String` class, the value returned will be `false`.

Q13 *(b) and (e)*

Unlike local variables, all fields are initialized with default initial values. All numeric fields are initialized to zero, `boolean` fields to `false`, `char` fields to `'\u0000'`, and *all* reference fields to `null`.

Q14 *(a)*

The `main()` method in (b) will always throw and catch an `ArrayIndexOutOfBoundsException`, since `args.length` is an illegal index in the `args` array. The `main()` method in (c) will always throw an `ArrayIndexOutOfBoundsException` since it also uses `args.length` as an index, but this exception is never caught. The `main()` method in (d) will fail to print the argument if only one program argument is supplied. The `main()` method in (e) will throw an uncaught `ArrayIndexOutOfBoundsException` if no program arguments are specified.

Q15 *(e)*

Method `g()` modifies the field `a`. Method `g()` modifies the parameter `b`, not the field `b`, since the parameter declaration shadows the field. Variables are passed by value, so the change of value in parameter `b` is confined to the method `g()`. Method `g()` modifies the array whose reference value is passed as a parameter. Change to the first element is visible after return from the method `g()`.

Q16 *(a)*

The program will fail to compile since the label `l2` cannot precede the declaration `int j = 0`. For a label to be associated with a loop, it must immediately precede the loop construct. If label `l2` preceded the `while` loop (instead of the declaration of `j`), the program would compile and print 9.

Q17 *(b)*

Classes cannot extend interfaces; they must implement them. Interfaces can extend other interfaces, but cannot implement them. A class must be declared as `abstract` if it does not provide an implementation for all abstract methods of the interfaces that it implements. Methods declared in interfaces are implicitly `public` and `abstract`. Classes that implement these methods must explicitly declare these methods to be `public`.

Q18 *(a)*

The code will fail to compile because the literal `4.3` has the type `double`. Assignment of a `double` value to a `float` variable without an explicit cast is not allowed. The code would compile and print `0` at runtime, if the literal `4.3` was replaced with `4.3F`.

Q19 *(b) and (e)*

The `&&` and `||` operators exhibit short-circuit behavior. The first operand of the ternary operator (`? :`) is always evaluated. Based on the result of this evaluation, either the second or third operand is evaluated.

Q20 *(e)*

No labels are mandatory (including the `default` label), and labels can be placed in any order within the `switch` body. The keyword `continue` may occur within the body of a `switch` statement as long as it pertains to a loop. An enum constant, a non-`long` integral constant expression, or a string constant expression can be used for `case` labels as long as the type is compatible with the expression in the `switch` expression.

Q21 *(a)*

Strings are immutable, so the `concat()` method has no effect on the original `String` object. The string on which the `trim()` method is called consists of eight characters, where the first and two last characters are spaces (`" hello "`). The `trim()` method returns a new `String` object in which the whitespace characters at each end have been removed. This leaves the five characters of the word `"hello"`.

Q22 *(a) and (e)*

Method overloading requires that the method signatures are different, but the method name is the same. The return type is irrelevant in this regard. The signature of the existing method is `sum(long, long)`.

The signature of the method in (a) is `sum(int, int)`. The signature of the method in (e) is `sum(long, int)`. Both signatures are different from the signature of the existing method.

Declarations (b) and (d) fail, since the method signature is identical to the existing method. Declaration (c) fails, since it declares an `abstract` method in a non-`abstract` class.

Q23 *(b)*

The method with the most specific signature is chosen. In this case the `int` argument `10` is boxed to an `Integer`, which is passed to the `Number` formal parameter, as type `Number` is more specific than `Object`.

Q24 *(d), (f), and (g)*

The `main()` method must be declared as `public` and `static`, with return type `void`, and takes a single array of `String` objects as argument. The order of the `static` and `public` keywords is irrelevant. Also, declaring the method `final` is irrelevant in this respect.

Q25 *(d)*

The length of the array passed to the `main()` method is equal to the number of program arguments specified in the command line. Unlike some other programming languages, the element at index 0 does not contain the name of the program. The first program argument specified is retrieved using `args[0]`, and the last program argument specified is retrieved using `args[args.length-1]`, when

`args.length` is greater than `0`. A program argument is a string, and several arguments are separated by spaces on the command line. To pass several arguments as one argument, these must be enclosed in double quotes.

Q26 (e)

When the program is called with no program arguments, the `args` array will be of length 0. The program will in this case print `no arguments`. When the program is called with three arguments, the `args` array will have length 3. Using the index 3 in the `numbers` array will retrieve the string `"four"`, because the start index is 0.

Q27 (b) and (e)

Static import from a class does not automatically import static members of any nested types declared in that class. The order of the import statements is arbitrary as long as they are declared after any package statement and before any type declaration. Name conflicts must be disambiguated explicitly.

Q28 (e)

An object reference is needed to access non-static members. Static methods do not have the implicit object reference `this`, and must always supply an explicit object reference when referring to non-static members. The static method `main()` legally refers to the non-static method `func()`, using the reference variable `ref`. Static members are accessible from both static and non-static methods, using their simple names. No `NullPointerException` is thrown, as `ref` refers to an instance of `MyClass`.

Q29 (c)

Declaration (4) defines a static method that tries to access a variable named `a`, which is not locally declared. Since the method is `static`, this access will be valid only if variable `a` is declared as `static` within the class. Therefore, declarations (1) and (4) cannot occur in the same class declaration, while declarations (2) and (4) can.

Q30 (a), (f), and (h)

The type of the `switch` expression must be either an `enum` type or `String` type or one of the following: `byte`, `char`, `short`, `int`, or the corresponding wrapper type for these primitive types. This excludes (b) and (e). The type of the `case` labels must be assignable to the type of the `switch` expression. This excludes (c) and (d). The `case` label value must be a constant expression, which is not true in (g), where the `case` label value is of type `Byte`.

Q31 (a)

The value of the `case` label `iFour` is *not* a constant expression and, therefore, the code will not compile.

Q32 (d)

Implementation (4) will correctly return the largest value. The `if` statement does not return any value and, therefore, cannot be used as an expression statement in implementations (1) and (2). Implementation (3) is invalid since neither the `switch` expression nor the `case` label values can be of type `boolean`.

Q33 *(c)*

As it stands, the program will compile correctly and will print `3, 2` at runtime. If the `break` statement is replaced with a `continue` statement, the loop will perform all four iterations and will print `4, 3`. If the `break` statement is replaced with a `return` statement, the whole method will end when `i` equals 2, before anything is printed. If the `break` statement is simply removed, leaving the empty statement (`;`), the loop will complete all four iterations and will print `4, 4`.

Q34 *(a) and (c)*

The block statement `{}` is a compound statement. The compound statement can contain zero or more arbitrary statements. Thus, `{{}}` is a legal compound statement, containing one statement that is also a compound statement. This inner compound statement has no statements. The block `{ continue; }` by itself is not valid, since the `continue` statement cannot be used outside the context of a loop. (c) is a valid example of breaking out of a labeled block. (d) is not valid for the same reasons that (b) was not valid. The statement at (e) is not true, since the `break` statement can also be used to break out of labeled blocks, as illustrated by (c).

Q35 *(d)*

The type of `nums` is `int[][]`. The outer loop iterates over the rows, so the type of the loop variable in the outer loop must be `int[]`, and the loop expression is `nums`. The inner loop iterates over each row, `int[]`. The loop variable in the inner loop must be `int`, and the loop expression in the inner loop is a row given by the loop variable of the outer loop. Only in the loop headers in (d) are both element types compatible.

Q36 *(b)*

The program will print `1` and `4`, in that order. An `InterruptedException` is handled in the first `catch` clause. Inside this clause, a new `RuntimeException` is thrown. This exception was not thrown inside the `try` block and will not be handled by the `catch` clauses, but will be sent to the caller of the `main()` method. Before this happens, the `finally` clause is executed. The code to print `5` is never reached, since the `RuntimeException` remains uncaught after the execution of the `finally` clause.

Q37 *(d)*

The method `nullify()` does not affect the array reference in the `main()` method. The array referenced by `args` is no longer reachable when control reaches (1). Only the array object and its four `Object` elements (i.e., five objects) are reachable when control reaches (1).

Q38 *(e)*

An object can be eligible for garbage collection even if there are references denoting the object, as long as the objects owning these references are also eligible for garbage collection. There is no guarantee that the garbage collector will destroy an eligible object before the program terminates. The order in which the objects are destroyed is not guaranteed. A thread cannot access an object once it becomes eligible for garbage collection.

Q39 (a), (b), (c), and (e)

The expressions `('c' + 'o' + 'o' + 'l')` and `('o' + 'l')` are of type `int` due to numeric promotion. Their evaluation would result in the values 429 and 219, respectively. Expression (d) is illegal, since the `String` class has no constructor taking a single `int` parameter. Expression (a) is legal, since string literals are references that denote `String` objects.

Q40 (d)

The expression `"abcdef".charAt(3)` evaluates to the character `'d'`. The `charAt()` method takes an `int` value as an argument and returns a `char` value. The expression `("abcdef").charAt(3)` is legal; it also evaluates to the character `'d'`. The index of the first character in a string is 0.

Q41 (e)

The expression `"Hello there".toLowerCase().equals("hello there")` will evaluate to `true`. The `equals()` method in the `String` class will return `true` only if the two strings have the same sequence of characters. The `compareTo()` method in the `String` class will return `0` only if the two strings have the same sequence of characters. The string comparison by these two methods is case sensitive, being based on the Unicode value of the characters in the strings.

Q42 (c)

The variable `middle` is assigned the value 6. The variable `nt` is assigned the string `"nt"`. The substring `"nt"` occurs three times in the string `"Contentment!"`, starting at indices 2, 5, and 9. The call `s.lastIndexOf(nt, middle)` returns the start index of the last occurrence of `"nt"`, searching backward from position 6.

Q43 (d)

The program will construct an immutable `String` object containing `"eeny"` and a mutable `StringBuilder` object containing `" miny"`. The `concat()` method returns a reference value to a new immutable `String` object containing `"eeny meeny"`, but the reference value is not stored; consequently, this `String` object cannot be referenced. The `append()` method appends the string `" mo"` to the string builder.

Q44 (c)

The code will fail to compile, since the package declaration cannot occur after an `import` statement. The `package` and `import` statements, if present, must always precede any type declarations. If a file contains both `import` statements and a `package` statement, the `package` statement must occur before the `import` statements.

Q45 (a) and (b)

Note that `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException` are subclasses of `IndexOutOfBoundsException`. The elements of the array are initialized as follows:

```
trio[0] = null;
trio[1][0] = null;
trio[2][0] = "Tom";
trio[3] = new String[0]; // {}, i.e., zero-length array
trio[4][0] = "Dick";
trio[4][1] = "Harry";
```

Element `trio[3][0]` does not exist because the array `trio[3]` is of zero length, resulting in an `ArrayIndexOutOfBoundsException` being thrown; this exception is also a subtype of `IndexOutOfBoundsException`. `IllegalIndexFoundException` is not defined.

Q46 (e)

The loop condition `++i == i` is always true, as we are comparing the value of `i` to itself, and the loop will execute indefinitely. The evaluation of the loop condition proceeds as follows: `((++i) == i)`, with the operands having the same value. For each iteration, the loop variable `i` is incremented twice: once in the loop condition and a second time in the parameter expression `i++`. However, the value of `i` is printed before it is incremented the second time, resulting in odd numbers from 1 and upward being printed. If the prefix operator is also used in the `println` statement, all even numbers from 2 and upward would be printed.

Q47 (d)

The expression `i % k` evaluates to the remainder value `3`. The expression `i % -k` also evaluates to the remainder value `3`. We ignore the sign of the operands, and negate the remainder only if the dividend (`j` in this case) is negative.

Q48 (e)

The class `Thingy` does not override the `equals()` method, so the `equals()` method from the `Object` class is executed each time. The method in the `Object` class compares the reference value for equality with the `==` operator. Having the same name in the second call to the `equals()` method does not make the `Thingy` objects equal. In the last call to the `equals()` method, the two references are aliases; that is, they have the same reference value.

Q49 (b)

Strings are immutable, so the method `concat()` does not change the state of the `s1` string. The `default` case is executed in the `switch` statement. Because of the fall-through in the `switch` statement, the last print statement is also executed.

Q50 (a), (d), and (e)

In (b), the right-hand side `int` value `255` requires a cast to convert to a `byte`. In (c), only integer literals can be specified in binary notation, not floating-point values. (e) will compile, but will throw a `NumberFormatException` at runtime. In (f), an underscore cannot occur at the start of an integer value. The compiler will interpret it as an identifier. In (g), an underscore cannot occur before or after any type designator (`L`).

Q51 (d), (e), and (i)

In (d) and (e), either the array length or the initializer block can be specified, as in (a), (c), and (f). In (i), the length of the leftmost dimension must be specified; the other dimensions are optional, as in (g) and (h).

Q52 (a) and (c)

The methods at (1) and (3) differ only in the return type, which is not sufficient for correct overloading. Method overloading requires that the method signatures are different, but the method name is the same. The return type is irrelevant in this regard. The code will not compile.

Q53 (d)

`StringBuilder` is mutable. The reference value in `giz.name` is copied to the formal parameter `sb` when the method is called. References `giz.name` and `sb` are aliases to the same string builder. Changes made to the string builder in the method are apparent when the call returns. In contrast, the `double` value in `giz.weight` is copied to the formal parameter `weight`, whose value is changed in the method, but this does not affect the value in the actual parameter, which remains unchanged.

Q54 (a), (b), (e), and (f)

`StringBuilder` is mutable. The methods `insert()` and `append()` of the `StringBuilder` return the reference value of the string builder, in addition to modifying it. The assignment in (b) and (f) is superfluous. In (c) and (d), only the local variable `meal` is assigned a reference value of a string builder, but it does it not change the string builder in the array `meals`.

Q55 (d)

After the first call to the overloaded `add()` method, the size of the array list is 1. Trying to insert an element at index 2 in the second call to the `add()` method results in a `java.lang.IndexOutOfBoundsException`, because index 2 is strictly greater than the current size of the array list.

Q56 (g)

The field `numOfGuests` is `static`, meaning the field belongs to the class `Room` and not to any object of the class. Such a field can be referenced by a reference whose type is the same as the class. The two references `r1` and `r2` refer to the same static field `numOfGuests`, which has the value 3. Because of string concatenation, the expression `"Number of guests: " + r1.numOfGuests + r2.numOfGuests` evaluates to `"Number of guests: 33"`.

Q57 (b), (c), and (d)

In (a), the theater cannot admit anyone unless there is a ticket-holder, so the test to see whether there is a ticket-holder comes first.

In (b), at least one guess has to be made, so the test can be done after making the guess.

In (c), some salt has to be added, as the food has too little salt initially. The test to see if the food tastes right can be done after some salt has been added.

In (d), at least one candle has to be added, so the test for the right number of candles can be done after adding a candle.

In (e), it a good idea to check first whether the cat is away before letting the mice play a little.

Q58 (b) and (f)

In both cases, the code in the `if` statement and the `while` loop is unreachable, so it can never be executed. In case of the `while` loop, the compiler flags an error. The `if` statement is treated as a special case by the compiler to simulate conditional compilation, allowing code that should not be executed.

Q59 (e)

The value `2014` is boxed into an `Integer`. The subclass `Integer` overrides the abstract method `intValue()` from the superclass `Number`, so that no cast or explicit parentheses are necessary. However, if this was not the case, only the syntax with the cast in (a) would be correct.

Q60 (b)

The thing to note is that the method `compare()` is overloaded in the subclass `Student`, and not overridden. Thus objects of the class `Student` have two methods with the same name `compare`. For overloaded methods, the method to be executed is determined at compile time, depending on the type of the reference used to invoke the method, and the type of the actual parameters. When the type of the reference is `Person` (as is the case for `p1` and `p2`), the method `compare()` in `Person` will always be executed. The method defined in the subclass `Student` is executed only by the last call `s1.compare(s2)` in the `main()` method.

Q61 (b) and (e)

The `add(element)` method adds an element at the end of the list. The `add(index, element)` method adds the element at the specified index in the list, shifting elements to the right from the specified index. The index satisfies `(index >= 0 && index <= size())`. The `set(index, element)` method replaces the element at the specified index in the list with the specified element. The index satisfies `(index >= 0 && index < size())`. The `for(;;)` loop adds the elements currently in the list at the end of the list. The list changes as follows:

```
[Ada]
[Ada, Alyla]
[Ada, Otto]
[Ada, Anna, Otto]
[Ada, Anna, Otto, Otto, Anna, Ada]
```

Q62 (e)

Elements with the `null` value count toward the size of the list. The lists have different sizes. The lists are two distinct lists, having unique reference values. The `equals()` test fails because the lists have different sizes.

Q63 (a), (c), (e), (f), (g), and (k)

The program prints the following, where the list contents are shown before and after each print statement. Note the return value from the `ArrayList` methods.

```
[3, 4, 1, null, 0]
(1) prints null
[3, 4, 1, null, 0]
(2) prints 4
[3, 3, 1, null, 0]
(3) prints 1
[3, 3, 1, null, 0]
(4) prints true
[3, 3, 1, null, 0]
(5) prints null
[3, 3, 1, 0]
(6) prints 0
[3, 3, 1, 0]
(7) prints false
[3, 3, 1, 0]
```

Q64 *(b), (f), and (g)*

(1): The initial capacity can be 0. The capacity can change as the list changes structurally.

(2): `List<String>` is not a subtype of `List<Object>`. Assignment is not allowed.

(3), (4), (5): Although the `Number` class is abstract, we can create an `ArrayList` of an abstract class. However, only reference values of objects of its concrete subtypes can be stored in such a list.

(6): The diamond operator can be used only with the `new` operator.

(7): `ArrayList<Integer>` is not a subtype of `ArrayList<ArrayList<Integer>>`. Assignment is not allowed. The `ArrayList` creation expression must declare the full element type or use the diamond operator.

(8): The declaration statement compiles, but an unchecked conversion warning is issued by the compiler. All bets are off regarding the type-safety of the `ArrayList`.

Q65 *(d) and (e)*

Textual representation of the current contents of the list is added as a string on each iteration of the loop. The loop body is executed 3 times. The default textual representation of a list is enclosed in brackets (`[]`), where textual representation of each element is separated by a comma (`,`).

Q66 *(d)*

The `add(index, element)` method accepts an index that satisfies the condition `(index >= 0 && index <= size())`. The `for(;;)` loop swaps elements to reverse the elements in the list.

Q67 *(c) and (d)*

(a): The `return` statement is mandatory in a lambda expression only if the lambda body is a statement block that has a non-`void` return.

(b): A `return` statement is illegal in a lambda expression if the lambda body is a single expression.

(e): A local variable declaration in the block scope of a lambda expression cannot shadow or redeclare a local variable with the same name in the enclosing method.

Q68 (a), (c), (g), (h), and (i)

The declarations at (1), (3), (7), (8), and (9) will not compile. In (1), the declared-type parameter must be in parentheses. In (3), the `final` modifier can be applied only to declared-type parameters. In (7), the `return` keyword cannot be used when the lambda body is a single expression. In (8) and (9), the `return` keyword is required for a non-`void` return from a lambda expression with a statement block. In (8), the statement terminator (`;`) is also missing.

Q69 (a), (d), and (e)

The declarations at (1), (4), and (5) will compile. In (1), the parameter `lock` is local in the block scope of the lambda expression. In (2), the lambda expression must explicitly return a value, regardless of whether the `if` statement is executed. In (3), the local `final` variable `lock3` in the enclosing scope cannot be redeclared. In (4) and (5), local variables `lock3` and `lock4` in the enclosing scope can be accessed in the lambda expression, as they are both `final` and effectively final, respectively. In (6), the local variable `lock4` in the enclosing scope cannot be redeclared as parameter.

Q70 (a), (c), (g), and (h)

In (b), the `return` keyword cannot be used in a lambda expression with a single expression body. In (d), the `return` keyword must be used in a lambda expression with a non-`void` statement block body. In (e) and (f), the class `String` does not have a constructor that takes an integer value.

Q71 (b)

A functional interface is an interface that has only one abstract method, aside from the abstract method declarations of `public` methods from the `Object` class. This single abstract method declaration can be the result of inheriting multiple declarations of the abstract method from superinterfaces.

All except `IA` are functional interfaces. `IA` does not define an abstract method, as it provides only an abstract method declaration of the concrete `public` method `equals()` from the `Object` class. `IB` defines a single abstract method, `doIt()`. `IC` overrides the abstract method from `IB`, so effectively it has only one abstract method. `IC` inherits the abstract method `doIt()` from `IB` and overrides the `equals()` method from `IA`, so effectively it also has only one abstract method.

Q72 (e)

(a): The `isBefore()` and `isAfter()` methods are strict in their comparison.

(b): The `withYear()` and `withMonth()` methods will adjust the day. The `withDayOfMonth()` and `withDayOfYear()` methods will throw a `DateTimeException` if the argument value will result in an invalid date.

(c): The `Period` class does not have the method `toTotalDays()`, but it does have the method `toTotalMonths()` that considers only the years and the months. The `Period` class has no notion of time of day or date in the year.

(d): The `Period` class does not implement the `Comparable` interface.

Q73 (d)

The calculation of `time.withHour(10).plusMinutes(120)` proceeds as follows:

```
12:00 with 10 hour ==> 10:00 + 120 min (i.e., 2 hours) ==> 12:00
```

Q74 (f)

The date value `2015-01-01` in the `date` reference never changes. The `withYear()` method returns a new `LocalDate` object (with the date value `0005-01-01`) that is ignored. The `plusMonths()` method also returns a new `LocalDate` object whose value is printed. The calculation of `date.plusMonths(12)` proceeds as follows:

```
2015-01-01 + 12 months (i.e., 1 year) ==> 2016-01-01
```

Q75 (c)

The static method call `Period.ofYears(10)` returns a `Period` with the value `P10Y`. This `Period` object is used to invoke the static method `ofMonths()` with the argument value of 16 months, resulting in a new `Period` object with the period value `P16M`. Its reference value is assigned to the `period` reference. The `of()` methods do not normalize the date-based values of a `Period`.

Q76 (d)

(a): The formatter will format a `LocalTime` object and the time part of a `LocalDateTime` object, but not a `LocalDate` object, as it knows nothing about formatting the date part. It will use the ISO standard.

(b): The formatter will format a `LocalDate` object and the date part of a `LocalDateTime` object, but not a `LocalTime` object, as it knows nothing about formatting the time part. It will use the ISO standard.

(c): The formatter will format a `LocalDateTime` object, but neither a `LocalDate` object nor a `LocalTime` object, as it requires both the date and the time parts. It will use the ISO standard.

Q77 (a), (d), and (f)

The input string matches the pattern. The input string specifies the time-based values that can be used to construct a `LocalTime` object in (a) by a formatter, based on the time-related pattern letters in the pattern. No date-based values can be interpreted from the input string, as this pattern has only time-related pattern letters. (b) and (c), which require a date part, will throw a `DateTimeParseException`.

To use the pattern for formatting, the temporal object must provide values for the parts corresponding to the pattern letters in the pattern. The `LocalTime` object in (d)

has the time part required by the pattern. The `LocalDate` object in (e) does not have the time part required by the pattern, so an `UnsupportedTemporalTypeException` will be thrown. The `LocalDateTime` object in (f) has the time part required by the pattern. In (f), only the time part of the `LocalDateTime` object is formatted.