Appendix F

Annotated Answers to Mock Exam

This appendix provides annotated answers to the questions in the mock exam for the *Java SE 17 Developer* exam found in **Appendix E**, **p. 1709**.

Annotated Answers

Q1 *(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.

Q2 *(e)*

Method `g()` modifies field `a`. Method `g()` modifies parameter `b`, not 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. A change to the first element is visible after return from the method `g()`.

Q3 *(b)*

Classes cannot extend interfaces; they can implement them. Interfaces can extend, but not implement other interfaces. A class must be declared as `abstract` if it does not provide an implementation for all abstract methods of the interfaces that it implements. Instance methods that have no implementation in an interface are implicitly `public` and `abstract`. Classes that implement these methods must explicitly declare these methods to be `public`.

Q4 *(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`.

Q5 *(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`.

Q6 *(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.

Q7 *(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 onward being printed. If the prefix operator is also used in the `println` statement, all even numbers from 2 onward would be printed.

Q8 *(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 (`i` in this case) is negative.

Q9 *(c)*

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. There is no fall-through in the `switch` statement, as the switch statement uses arrow notation in which the case labels are mutually exclusive.

Q10 *(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 the 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.

Q11 *(b)*

The thing to note is that the method `compareTo()` is overloaded in the subclass `Student`, and is not overridden. Thus objects of class `Student` have two methods with the same name: `compareTo`. 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 `compareTo()` in `Person` will always be executed. When the type of the reference is `Student` and the argument type is `Person`, the overridden method `compareTo()` in `Person` will always be executed. The overloaded method `compareTo()` defined in the subclass `Student` is executed by the last call `s1.compareTo(s2)` in the `main()` method, where the type of the reference is `Student` and the argument type is also `Student`.

Q12 *(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:

Click here to view code image

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

Q13 *(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.

Q14 *(d)*

In (a), the type of parameter `s` is inferred to be `String`. The new string returned by calling the `toLowerCase()` method is discarded. Only the string element `"Circle"` contains the substring `"c"` and is printed.

In (b), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The expression `s.contains("c")` in the `return` statement fails to compile since the method `contains()` is not defined in the class `Object`.

In (c), the type of parameter `s` is inferred to be `String`. Only the string element `"Circle"` contains the substring `"c"` and is printed.

In (d), the type of parameter `s` is inferred to be `Object`. The string elements are referenced by the parameter `s` of type `Object`. However, the chain of method calls is invoked on objects of type `String`, resulting in both stream elements being selected and printed.

In (e), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The statement fails to compile since the method `contains()` is not defined in the class `Object`.

In (f), the raw type `Predicate` is used. The type of parameter `s` is inferred to be `Object`. The class `Object` does not define the method `toLowerCase()` and the statement fails to compile.

Q15 *(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.

Q16 *(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: `2021-01-01 + 12 months (i.e., 1 year) ==> 2022-01-01`

Q17 *(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.

Q18 *(a)*

A `Thread` object executes the `run()` method of a `Runnable` object in a separate thread. A `Runnable` object can be provided when constructing a `Thread` object. If no `Runnable` object is supplied, the `run()` method of the `Thread` object (which implements the `Runnable` interface) is executed. A thread is initiated by calling the `start()` method of the `Thread` object.

Q19 *(b) and (d)*

In (a), the class need not be declared as a generic type if it defines any generic methods.

In (b), the method `choose(T, T)`, where `T` extends `Comparable<T>`, is not applicable to the arguments `(Integer, String)`. Note that `Object` is not `Comparable<Object>`.

In (c), the method `choose(T, T)`, where `T` extends `Comparable<T>`, is not applicable to the arguments `(Integer, Double)`. Note that `Number` is not `Comparable<Number>`.

In (d), the actual type parameter `Double` specified in the method call also requires that the `int` argument is cast to a `double` in order for the call to be valid. The method `choose(T, T)`, where `T` extends `Comparable<T>`, is then applicable to the argument list `(Double, Double)`.

(e) cannot convert the `Double` returned by the method to an `int` using a cast.

In (f), the method returns a `Double` that is first converted to a `double`, which in turn is converted to an `int`.

Q20 *(a) and (d)*

Field `b` of the outer class is not shadowed by any local or inner class variables; therefore, (a) will work. Using `this.a` will access field `a` in the inner class. Using `this.b` will result in a compile-time error, since there is no field `b` in the inner class. Using `Outer.this.a` will successfully access the field of the outer class. The statement `c = c` will only reassign the current value of the local variable `c` to itself.

Q21 *(a)*

All enum types implement the `Comparable` interface. Comparison is based on natural order, which in this case is the order in which the constants are specified, with the first one being the smallest. The ordinal value of the first enum constant is 0, the next one has the ordinal value 1, and so on.

Q22 *(c) and (d)*

Executing synchronized code does not guard against executing non-synchronized code concurrently.

Q23 *(a), (c), and (e)*

Lists of type `Pet`, `Dog`, and `Cat` are subtypes of `List<? extends Pet>`, `List<? extends Wagger>` and `List<?>`.

`List<? super Pet>` is a supertype for a list of `Pet` itself or a supertype of `Pet`—for example, `Wagger`, but not `Dog` or `Cat`.

`List<? super Wagger>` is a supertype for a list of `Wagger` itself or a supertype of `Wagger`—for example, `Object`, but not `Pet`, `Dog`, or `Cat`.

Q24 *(c)*

We need to access the following:

- `Importing.JPEG` (to print `200`).
- `p1.Util.Format.JPEG` (to print `Jpeggy`). Since `p1.Util.Format` is statically imported by the second import statement, we need only specify `Format.JPEG`.
- `p1.Format.JPEG` (to print `JPEG`), which is explicitly specified to distinguish it from other `JPEG` declarations.

Q25 *(b)*

First, note that the `indexOf()` method returns the index of the *first* occurrence of its argument in the list. Although the value of variable `i` is successively changing during the execution of the loop, it is the *first* occurrence of this value that is replaced in each iteration:

Click here to view code image

```
                    0      1     2

                  [2019, 2020, 2021]
After iteration 1: [2020, 2020, 2021]
After iteration 2: [2021, 2020, 2021]
After iteration 3: [2022, 2020, 2021]
```

Note also that we are not removing or adding elements to the list, only changing the reference values stored in the elements of the list.

Q26 *(c)*

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 default constructors of the superclasses up the inheritance hierarchy of the `GraduateStudent` class are called, as none of the superclasses are `Serializable`.

Q27 *(c), (d), (g), and (h)*

The method header signature of the corresponding methods is the same after erasure—that is, `List fuddle()` and `List scuddle(Object)`. The return type of overriding methods can be a raw type or a parameterized type.

Q28 *(d)*

A `try` block must be followed by at least one `catch` or `finally` clause. No `catch` clause can follow a `finally` clause. Methods must declare any checked exceptions in a `throws` clause, if they do not catch the exceptions.

Q29 *(a), (b), and (c)*

First, note that nested packages or nested static members are not automatically imported.

In (d), `p2.DefenceInDepth` is not a static member and therefore cannot be imported statically.

With (e), `March.LEFT` becomes ambiguous because both the second and the third import statements statically import `March`. The enum constant `LEFT` cannot be resolved either, as its enum type `March` cannot be resolved.

With (f), the enum constant `LEFT` cannot be resolved, as none of the static import statements specify it.

The enum type `p2.March` is also not visible outside the package.

Q30 *(b)*

Statement (c) is `false`, since an object of `B` can be created using the implicit default constructor of the class. `B` has a default constructor since no constructor has been defined. Statement (d) is `false`, since the second constructor of `C` will call the first constructor of `C`.

Q31 *(c)*

The important thing to remember is that an instance of a class is also an instance of its superclasses in the inheritance hierarchy.

Q32 *(d)*

Enum constants can be used as `case` labels and are not qualified with the enum type name in the `case` label declaration. The `switch` selector expression is compatible with the `case` labels, as the reference `this` will refer to objects of the enum type `Scale5`, which is the type of the `case` labels. The call to the method `getGrade()` returns a `char` value, which in this case is `'C'`.

Q33 *(b) and (g)*

The method in (a) and the method at (1) do not have the same or covariant return types required for overriding.

The method in (b) overrides the method at (2).

The instance method in (c) cannot override the static method at (4).

The static method in (d) and the static method at (4) do not have compatible return types for overriding.

The static method in (e) cannot hide the instance method at (3).

The instance method in (f) and the instance method at (5) do not have compatible return types for overriding.

The instance method in (g) overrides the instance method at (6), and they have covariant return types.

Q34 *(c)*

A primitive value cannot be widened and then boxed implicitly. The primitive value is boxed to its corresponding wrapper type, and an attempt is made to find a corresponding formal parameter with the most specific type to which it can be passed. The varargs value is passed in the method calls as follows:

Click here to view code image

```
printFirst(10);           // new Integer[] {Integer.valueOf(10)}
printFirst((byte)20);     // new Number[] {Byte.valueOf(20)}
printFirst('3', '0');     // new Object[] {Character.valueOf('3'),
                          //               Character.valueOf('0')}
printFirst("40");         // new Object[] {"40"}
printFirst((short)50, 55);// new Number[] {Short.valueOf(50),
                          //               Integer.valueOf(55)}
printFirst((Number[])new Integer[] {70, 75}); // Passed as array of Number
```

Q35 *(e)*

The program will compile and print `1`, `3`, and `2` at runtime. First, the static initializers are executed when the class is initialized, printing `1` and `3`. When the object is created and initialized, the instance initializer block is executed, printing `2`.

Q36 *(a)*

As there is no appropriate resource bundle file for the `no_NO` locale, the resource bundle for the default locale (US) is loaded. The value of the key `"greeting"` from this resource bundle is printed. Changing the default locale of the application does not change the locale associated with the resource bundle

`rbs`. The value of the key `"greeting"` from this resource bundle is printed one more time.

Q37 *(e)*

The `runner` thread can only proceed if `intArray[0]` is not `0`. If this element is not `0`, it has been initialized to `10` by the main thread. If this element is `0`, the `runner` thread is put into the wait set of the `intArray` object, and must wait for notification. The main thread only notifies after initializing both elements of the array to `10`. Calling the `notify()` method on an object with no threads in its wait set does not pose any problems. A thread can only call `notify()` on an object whose lock it holds. Therefore, the last `synchronized` statement in the `main()` method is necessary.

Q38 *(b)*

Notice that the array is declared as a raw type, yet the objects that are placed into this array are of a parameterized type. This is a legal assignment because an object of a parameterized type can be assigned to a raw type variable, in this case an element within the raw type array. However, in this case the `compareTo()` method accepts an argument of type parameter `T`, which allows invocation of any `Object` method using a reference of this type parameter. The code in the `compareTo()` method invokes the `toString()` method to convert each value into a `String` object and then compare these strings. The order of the comparison is therefore not numeric, but lexicographical. Note that the raw type `Thingy` is also used in the `for(:)` loop.

Q39 *(c)*

Each lambda expression implements a `Comparable` that returns an `int` value. These `int` values are accumulated and stored in the static field `result` of class `Widget`. The first lambda expression returns the length of the `"ACME"` string (i.e., 4), the second lambda expression returns the index of letter `"C"` within the string `“ACME"` (i.e., 1), and the third expression returns the day of the month in the `LocalDate` object (i.e., `20`). The sum in the `result` field is thus `25`, which is printed.

Q40 *(b)*

The `sort()` method of the `Arrays` class sorts the elements of the `Integer` array according to the total ordering defined by the `Comparator<Integer>` that is implemented by the lambda expression. The difference `x - y` between two values `x` and `y` determines whether `x` is less than, equal to, or greater than `y`, according to the contract of the `compare()` method of the `Comparator<E>` interface. The elements of the array are sorted in ascending order.

Q41 *(a)*

The `LocalDate` class uses the ISO-8601 standard, where year 0 corresponds to 1 BC. The pattern `"dd MM yy G"` results in the number of the day in the month, the short name of the month, the two-digit value of the year, and BC/AD being used to format the local date.

Q42 *(c)*

Stream processing distinguishes `LocalDateTime` objects from all other objects. It extracts the number of seconds from a `LocalDateTime` object, substitutes 1 for all other elements in the stream, and calculates the sum for all the elements. There is only one `LocalDateTime` object in this stream, which yields the value of `2`; the other four objects are substituted with a value of `1` in the stream. Thus the result is `6`.

Q43 *(d)*

The text block has two incidental leading whitespace on each line that are removed. After the removal of the two incidental spaces from all lines, the line lengths in the text block are 1, 3, and 1. The resulting string literal is `"a\n b\nc\n"`.

Q44 *(c)*

The paths constructed by the code are as follows:

Click here to view code image

```
p1: /users/joe
p2: /users/bob
p1.relativize(p2):                  ../bob
```

```
p3 = p1.resolve(p1.relativize(p2)): /users/joe/../bob
p4 = p3.normalize():               /users/bob
```

Since the first component of a path has index 0, index 1 refers to `joe` in `p3` and `bob` in `p4`.

Q45 *(b)*

The resource bundles are loaded from the files `resources_en.properties` (which has an entry for key `f2`) and `resources.properties` (which has an entry for key `f1`). The `resources_en_GB` bundle is not loaded by the `getBundle()` method.

When the value for key `f1` is not present in the `resources_en` bundle, the parent bundle `resources` is searched for the value of key `f1`. The `en_GB` locale is applied to the formatter, which uses the value `"yy-MMM-dd"` of key `f1` to format the date.

Q46 *(b)*

The `action()` method uses short-circuit conditional operators `||` and `&&` to test whether the parameter `obj` is a `String` or an `Integer`. The `instanceof` pattern match operator only introduces a pattern variable if it evaluates to `true`. If it is a `String`, it checks whether this string contains the character `'1'`. Otherwise, it checks whether the parameter `obj` is an `Integer`, and if it is, it determines whether its value is less than 1. The object passed to the method is an `Integer` object with a value of 1, which is not less than 1, and thus the `boolean` expression returns `false`.

Q47 *(c)*

A non-canonical constructor is required, since the arguments passed in the constructor are of type `String` and `int`, and not `String` and `Duration`, as in the declaration. The first statement in a non-canonical constructor must be an explicit invocation of a constructor with the `this()` expression that leads to the canonical constructor being invoked so that the component fields `title` and `duration` are initialized.

(a) and (b) are incorrect because they do not provide a value for the `title` field. Moreover, in (a), the `this` reference cannot be used in a `this()`

expression.

(d) and (e) do not invoke the canonical record constructor with the `this()` expression.

(c) fulfills all the requirements.

Q48 *(b)*

The `walk()` method navigates the directory structure for the directory `./Sun`. The `map()` method extracts the name of the leaf element from each `Path` that is encountered in this depth-first walk in the directory tree. The names are sorted. Notice that although the names start with a character that represents a number, the natural ordering is that for strings.

The `limit()` method limits the length of the stream to just the first three elements. Given the natural ordering for strings, the first three elements in the list are `1_Io`, `1_Mercury`, and `1_Moon`. The `forEach()` operation prints a substring extracted from the name starting at index 2.

Q49 *(b)*

The method `lines()` creates a stream of text lines read from the text file. Lines are grouped on the substring extracted from each line after the `':'` character. The map created by grouping has these substrings as keys and the lines containing the key as values. All lines that end with 1.70 will be grouped together, all lines that end with 1.99 will be in another group, and so on.

Q50 *(c)*

Module `basic` is the service provider as it implements the service defined by the service interface `TransportType` in module `transport`. Module `basic` requires module `transport` and must declare that it provides an implementation (`basic.mode.Horse`) of the service interface `transport.mode.TransportType`.

The `requires` directive must specify modules, not packages, as in (b) and (d). (a) requires the wrong service module and implements the wrong service interface.