SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

If you're a regular reader of this quiz, you know that these questions simulate the level of difficulty of two different certification tests. Those marked "intermediate" correspond to questions from the Oracle Certified Associate exam, which contains questions for a preliminary level of certification. Questions marked "advanced" come from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

These questions rely on Java 8. I'll begin covering Java 9 in future columns, of course, and I will make that transition quite clear when it occurs.

I'd also like to welcome Mikalai Zaikin to this column as a coauthor. He's been working on these questions with me for some time now, so you've already been seeing the benefit of his work.

Question 1 (advanced). Given this code:

```
public class OneValue {
    private final int x;
}
```

Consider these possible changes:

Change 1. Change the declaration of `x` as follows:

```
    private final int x = 99;
```

Change 2. Add to the class as follows:

```
    public OneValue() {
```

```
        x = 100;
    }
```

Change 3. Add to the class as follows:

```
    private void setX(int x) {
        this.x = x;
    }
    public OneValue() {
        setX(100);
    }
```

Which are true? Choose two.

A. The code compiles as it is.

B. The code compiles if change 1 is done.

C. The code compiles if change 2 is done.

D. The code compiles if change 3 is done.

E. The code compiles if change 1 and change 2 are both done.

Question 2 (advanced). Which of the following classes produce immutable objects? Choose two.

A.

```
public class Immut1 {
   final int[] data = { 1, 1, 2, 3, 5, 8, 13 };
   String name;
   public Immut1(String n) {  this.name = n; }
}
```

B.

```
public class Immut2 {
   final int[] data = { 1, 1, 2, 3, 5, 8, 13 };
   final String name;
```

```
   public Immut2(String n) { this.name = n; }
}
```

C.

```
public class Immut3 {
   private int x;
   public Immut3(int x) { this.x = x; }
}
```

D.

```
public class Immut4 {
   private List<String> ls;
   public Immut4() {
      ls = Arrays.asList("Fred", "Jim", "Sheila");
   }
   public String get(int idx) {
       return ls.get(idx);
   }
}
```

E.

```
public class Immut5 {
    private List<String> ls;
    public Immut5(String... strings) {
        ls = Collections.unmodifiableList(Arrays.asList(strings));
    }
    public String get(int idx) {
        return ls.get(idx);
    }
}
```

Question 3 (intermediate). Given this method:

```
public final void doStuff(int val) /* point A */ {
    if (val < 0) throw new NullPointerException();
    if (val < 1) throw new IOException();
    if (val < 2) throw new OutOfMemoryError();
}
```

Which of the following is best?

A. Insert the following at point A:
`throws Exception`

B. Insert the following at point A:
`throws NullPointerException, OutOfMemoryError`

C. Insert the following at point A:
`throws IOException, OutOfMemoryError, SQLException`

D. Insert the following at point A:
`throws IOException`

E. Insert the following at point A:
`throws NullPointerException`

Question 4 (intermediate). Given this:

```
String s = "Hello";
StringBuilder sb = new StringBuilder("Hello");
StringBuilder sb2 = new StringBuilder("Hello");
// line n1
```

Which is true?

A. Placed at line n1, the following fragment:
`System.out.println(sb + sb);`

prints this:
```
HelloHello
```

B. Placed at line n1, the following fragment:
```
System.out.println(sb.equals(sb2));
```

prints this:
```
true
```

C. Placed at line n1, the following fragment compiles successfully:
```
String val = sb.equals(s)?sb:"Differ";
System.out.println(val);
```

D. Placed at line n1, the following fragment:
```
CharSequence val = sb.equals(s)?sb:"Differ";
System.out.println(val);
```

prints this:
```
Hello
```

E. Placed at line n1, the following fragment:
```
System.out.println(sb.equals(s)?sb:"Differ");
```

prints this:
```
Differ
```

Answer 1. The correct answers are options B and C. *Java Language Specification* section 8.3.1.2 says this about final fields: "A blank final instance variable must be definitely assigned at the end of every constructor of the class in which it is declared, or a compile-time error occurs."

This means that the final field x must receive exactly one explicit assignment, which must happen before the constructor is complete. This tells you immediately that option A must be incorrect, because in the original code presented in the question there is no assignment to the field. Note that the field as declared is termed a "blank final" (the terminology used in the Java specification paragraph above) and as such, the default assignment to zero that is implicit for all object members does not satisfy the requirement.

Change 1 assigns a value to x as part of its declaration and, therefore, x is definitely assigned even before any constructor runs. Therefore, option B is correct.

Change 2 adds a simple constructor that initializes the value of x. This change, made in isolation, would result in exactly one constructor and causes that constructor to unconditionally assign a value to x. Because the blank final is definitely assigned, exactly once, before the end of the only constructor, this change works, and option C is correct.

Change 3 suggests adding a constructor that might seem functionally equivalent to the one proposed in change 2. However, in this case, the change fails. The private method that attempts to assign the value to x will not compile, because it's possible for it to be invoked after the object has been initialized. Because this fails to compile, option D must be incorrect.

Performing both change 1 and change 2 also fails, because this would result in an attempt to perform two assignments to the variable x, and the Java specification demands exactly one assignment. Therefore, option E is also incorrect.

Answer 2. The correct answers are options C and D. An object is immutable if no syntactically permissible interaction with it by external code can change its state after construction, and no code within the class ever makes any such change either. Literally, once it is created, the value remains the same.

Now, before analyzing this question, be aware that if you decide to create classes that yield immutable objects (which is a design style that can reap significant rewards in terms of correctness, particularly in concurrent systems), you should do a better job than the examples shown here. In particular, although you'll see that the `final` keyword is *not* sufficient to render everything it touches unalterable, it should almost certainly be used anyway. In particular, it has some value in concurrency that is not part of this discussion. Also keep in mind that it's possible to break many forms of immutability through reflection, which might have unexpected consequences.

This question investigates the rules and purpose of Java's exception mechanism, and it also dares to stray into that troublesome territory of asking what's "best" rather than merely what's "correct."

In option A, the fields have default accessibility, rather than being private, so it's a simple matter for any other class that has access (that is, any other class in the same package) to mutate the value of the `String name` to point to a different string. Therefore, option A is incorrect.

In option B, the `String name` field has been marked final, so even though it's accessible to other members of the package, it cannot be mutated; it must refer to the string object that's passed into the constructor. Strings themselves are immutable, so that field's value can never be changed. However, the contents of `final int [] data` can, in fact, be changed (and actually, it could be changed in option A, too, although you already know option A is incorrect based on the string field). This is because the `final` keyword prevents the value of `data`—which is a pointer—from being modified. So, `data` can never refer to any array other than the one with which it's initialized. Of course, the length of arrays can never change after they are created, but their contents *can* be changed. Therefore, the values in the `data` array are actually mutable by any code in other classes in the same package. Hence, option B is incorrect.

In option C, there is a single field: `int x`. The field is private but not final. The value of the field is initialized with a *copy* of the value passed to the constructor. (All arguments in non-remote Java method invocations are passed by value, and with primitive types, the "value" really is the value being represented, not the "value of the reference.") Because of this, changes to the original variable that was passed as an argument to the constructor do not affect the value of `x`. Also, no code in the class ever changes the value of `x` after the object is constructed. So, even though the field is not marked final, instances of this class are immutable, and option C is correct.

In option D, you again see a private, nonfinal field. This time, it's `List<String> ls`. Because it's private, and nothing outside the class ever has a copy of the reference value in `ls`, nothing will ever change the contents of the list or point the variable at a different list. Therefore, option D is correct.

It's prohibited for an overriding method to declare checked exceptions that were not already permitted in the context of the overridden method.

Option E is a little more subtle. You have a variable, `ls`, which is identical to the one described in option D. Therefore, you know that nothing changes the value of `ls` to make it refer to a different list object. If you can be sure that the list that `ls` refers to cannot be altered in any way, you would know the object is immutable.

The variable `ls` is initialized to refer to a list created by the `Arrays.asList` method, which is a utility that describes itself as creating a "structurally immutable list"—which sounds promising; the list will not allow the addition or removal of elements. However, the list created by `Arrays.asList` actually *honors* attempts to reassign any given element of the list. But to counter that, this list is wrapped in `Collections.unmodifiableList`, which puts a proxy wrapper around the object, so that any attempt to modify the list will throw an exception. Surely this must be immutable, right? Well, it turns out that the list that's created uses the *provided* array as its backing storage. Therefore, if the caller of the constructor chooses to provide an explicit `String []` as its argument, any changes made to that array will be reflected in the list. Because of this, the objects are not reliably immutable and option E is incorrect.

If you want to examine this effect, try running this code:

```
String [] names = {"Tony", "Jane"};
Immut5 i5 = new Immut5(names);
System.out.println("i5.get(0) " + i5.get(0));
names[0] = "Anthony";
System.out.println("i5.get(0) " + i5.get(0));
```

Answer 3. The correct answer is option D. This question investigates the rules and purpose of Java's exception mechanism, and it also dares to stray into that troublesome territory of asking what's "best" rather than merely what's "correct." However, we hope to make a good case for that value judgment, and while we are happy to include this question because it creates a useful discussion—both about Java's exception mechanism and about how to evaluate a judgment like this—we doubt that this question would survive unchanged in the real exam.

The first point is that Java distinguishes checked exceptions from unchecked exceptions and errors. In particular, a method that might throw a checked exception must announce this in that method's signature. In this question, the appropriate point for the syntax that declares such information is marked `/* point A */`. Therefore, the question is really asking what exception declaration would best suit this method. It's pretty clear that any situation that doesn't even compile cannot be considered satisfactory, so as long as some of the options would compile they must be "better than" any that do not. Consider the issue of compilation first.

If the method might throw any checked exceptions, it must carry a declaration that announces that. In this case, the only checked exception that is potentially thrown is the `IOException`; so at a minimum, the method must declare something that encompasses that exception. Options B and E fail on that point, because they declare unchecked exceptions. (Note that `OutOfMemoryError` falls into the category of "unchecked," although it's a subclass of `Throwable`, not of `Exception`.) For convenience, we'll simply use the term *unchecked exceptions* to include errors that are not parents of `IOException`. Therefore, options B and E are incorrect.

However, the remaining options all declare either `IOException` or a parent class of `IOException`. It's important to note that a `throws` clause that mentions a parent exception class is sufficient to encompass any child classes. Because of this, options A, C, and D all allow the code to compile correctly, so how can you choose the "best" option among these?

When you declare an exception in a `throws` clause, you impose an obligation on the caller of the method; the caller must do something about the exception. Also, the `throws` clause is a form of description of the type of problems that can arise when calling the method. These points both suggest that a `throws` clause should be as specific as possible. To be more general or to mention irrelevant exceptions might place an additional burden on the caller by creating a perceived requirement to handle situations that don't in fact arise. Further, additional generality will likely have the effect of hiding the real problem that might arise, making it harder for the caller to know how to respond if an exception is reported. On this basis, it's clear that option C, which reports one unchecked exception (`OutOfMemoryError`) with a checked exception that *cannot* arise, is unlikely to qualify as "best." Therefore, option C is incorrect.

Testing equality between different types almost always returns false regardless of the contents.

By the same arguments, you can also see that option A, which simply (and vaguely) reports that an `Exception` might arise, is also less helpful than option D, which gets directly to the point of reporting the single checked exception that could arise from the method. As a result, you can conclude that option A is incorrect, and option D is the correct answer.

There's another small point to consider as part of this discussion. The question mentions that the method is final. Why would that make any difference? It's certainly a tenuous point in this case, but it helps justify the "best" value judgment. Often, an abstract method in an interface declares a fairly general exception (consider the `close()` method in the `AutoCloseable` interface, which throws `Exception`). Given that such a method cannot possibly throw any exceptions because it doesn't have any implementation, why would this be? The answer is that it's prohib-

ited for an overriding method (which, of course, includes the methods that implement interface abstract methods) to declare checked exceptions that were not already permitted in the context of the overridden method. Without this restriction, you could have a reference of a parent type, and the compiler would let you call a method on it without handling a particular checked exception, but if the reference turns out to refer—at runtime—to an implementation that *does throw* that checked exception, you would have effectively cheated the checked exception mechanism. Generally, allowing an overriding method to do something not permitted for the overridden method would break the Liskov Substitution Principle, and in this specific case, it would break the protections provided by checked exceptions.

This means that if a method is expected to be overridden, it's not unreasonable to declare it as throwing some checked exceptions that simply don't arise in its current form. Had the method not been final, it would have been much harder to make a convincing case that option C was not the "best" choice (because it allows additional flexibility). But as it is, declaring throws `SQLException` is just a source of confusion, because the method does not throw `SQLException` nor is it possible that any overriding method might do so.

Answer 4. The correct answer is option E. This question investigates several aspects of `StringBuilder` and its relationship with `String`.

In option A, the code uses the `+` operator with two operands that are `StringBuilder` objects. One of Java's "special case" rules is that the only allowed operator overloading is the language-defined ability to concatenate `String` types using the `+` operator. Another fundamental rule is that when a `+` operator has a `String` type as an operand, if the other operand is not a `String`, it will be converted into one—and that, of course, brings up a third fundamental rule: in Java *any* data type can be converted into a `String`. (Admittedly, the conversion isn't always very helpful, but it's definitely legal.) However, in this case, although both operands represent "text" in the general sense—indeed, they're both instances of the interface `CharSequence`—neither operand is a `String`. Therefore, the code fails to compile, because it attempts to use the `+` operator with illegal arguments. Because of this, option A is incorrect.

Option B considers equality testing. This turns out to be a pretty simple rule, too. Testing equality between different types almost always returns false regardless of the contents (with some exceptions—take a look at the API-documented requirements for the `equals` methods in the `List` and `Set` implementations). However, in this question, you have two `StringBuilder`s that contain the same text. In this situation, it's easy to assume that the two objects will test as equal. However, that's not at all the case; indeed, relatively few of the core Java API classes implement a useful `equals` method, and `StringBuilder` is *not* one of them. The way you can determine this is by looking at the documentation of the class. Look at `String`'s `equals` method, and you'll see the API docs define how it tests for identical character sequences. But, look at `StringBuilder`; the only mention of the `equals` method is that it's "inherited from `java.lang.Object`." Of course, the default `equals` method defined by `Object` tests to see if two references refer to the same object in memory. As a result, the fragment in option B actually prints `false`, and option B is incorrect.

The final three options all hinge on related points. `String` and `StringBuilder` are different types on independent branches of the inheritance tree. As such, they are not assignment-compatible with one another. However, they also have elements of a shared type hierarchy; they're both subclasses of `Object`, and they both implement the `CharSequence` interface.

In these options, a ternary expression has `String` and `StringBuilder` as the two option values. The type of such an expression cannot be `String`; it must be some common parent of both arguments. Therefore, the attempt to assign the result of the ternary expression to the `String` variable `val` in option C will cause a compilation failure. Therefore, option C is incorrect.

In option D, the type of the variable `val` has been changed to `CharSequence`. This now forms a legal, compilable fragment. However, the test in the ternary expression `sb.equals(s)` will evaluate to false, because the arguments are of differing types, and `StringBuilder` does not handle that. Given that the test evaluates to false, the ternary expression as a whole evaluates to the third operand, and the fragment prints `Differ`. Because of this, option D is incorrect.

In option E, the intermediate variable `val` has been removed and the ternary expression is the argument to the `println` call. In this case, it's up to the compiler to find a suitable type

for the expression, and it doesn't really matter if it chooses `Object` or `CharSequence`. Either is a legitimate argument to the `println` method and, consequently, the code compiles successfully. Of course, the expression `sb.equals(s)` still evaluates to false and the output that is printed is `Differ`—as it was in option D. Therefore, option E is correct.

As a side note, the `CharSequence` interface isn't explicitly mentioned in the exam objectives. However, both `String` and `StringBuilder` are, and this interface is an aspect of both. We doubt you'll come across it in the real exam, but our excuse is that by using it here, we were able to make the example a little more interesting and, perhaps, teach something useful. We hope you'll forgive the indulgence! `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of the Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.