If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Imagine that your colleague is refactoring your project to simplify the application’s configuration management. He has introduced the following `Config` class, which extends `java.util.Properties`:

```java
public class Config extends Properties {
    static class ConfigException extends RuntimeException {
        ConfigException(String message) {
            super(message);
        }
    }

    public String getProperty(final String key) {
        String property = super.getProperty(key);
        if (property == null) {
            throw new ConfigException(String.format("Missing value for key ", key));
        }
        return property;
    }

    BiFunction<Boolean, String, Optional<String> flag, val) -> {
        return flag ? Optional.of(val) : Optional.empty();
    };
```
An outline of the expected usage in client code looks like the following:

```java
Config cfg = ... // load config from file
var path= cfg.getConfigString("path");
String pathStr = path.orElseGet(() -> getDefa
```

Unfortunately the `Config` class still throws exceptions sometimes when it calls `getConfigString()` with nonnull keys.

**Which approach will fix the problem?** Choose one.

A. The `Config` class is not thread-safe so you need to wrap it in `ThreadLocal`, as follows:

```java
ThreadLocal<Config> tl = ThreadLocal.withInit
```

The answer is A.

B. You need to override `put(Object, Object)` and `setProperty(String, String)` to avoid storing null values in the `Config` class.

```java
Supplier<String>
```

The answer is B.

C. The `Config` class may contain null values for some keys, so you need to use `Optional.ofNullable(...)` instead of `Optional.of(...)`.  

```java
Optional.ofNullable("
```

The answer is C.

D. You need to use a `Supplier<String>` as a second parameter for the `BiFunction`.

```java
null
```

The answer is D.

**Answer.** This quiz represents the style of some of the new questions that can be found in the 1Z0-819 Java SE 11 certification exam. Some of them are much more complex than was typical with previous exam versions, and you will need to analyze what’s going on very quickly since the exam requires you to answer 50 questions in 90 minutes. Takers of the 1Z0-819 exam often mention not having as much time as they would have liked.
Now, to the question itself. Since the description states that exceptions are still arising, the most direct route to solving this is to determine how that could happen. In this case, you’ll then see that the correct answer is readily identified. Note that the text of the question indicates that the problem to be resolved is not related to calling the `getConfigString` with a `null` argument, even though this would certainly throw an exception too.

It seems that this code was written to move the hypothetical project to a more functional programming style that avoids the exposure of null pointers and exceptions, and instead use the new monad-like `Optional` class. This class allows the representation of either a value or the absence of any value, while avoiding the use of a null pointer. (Tony Hoare, who is generally credited with inventing the null pointer concept for the ALGOL W language in 1965, apologized decades later for what he called his “billion-dollar mistake.”)

Sometimes the client will request a value for a key that does not exist. This, of course, indicates that there is no such value available. In this situation, the `java.util.Properties` class will return `null`. This code seems designed to return an empty `Optional` instead. In such a situation, client code (as shown in the example) can use one of the `Optional` class’s mechanisms to provide a default value as a fallback.

However, the `Config` class code contains a severe mistake. When the key exists, the code will work correctly, and an `Optional` will be returned containing the appropriate value. However, when the key does not exist the `Config` class code will throw a `Config$ConfigException` instead of returning `Optional.empty()`. The question is, why?

Look at the call to the `compose.apply` method, which is provided by the lambda expression that implements `BiFunction`. There are two arguments passed to this invocation. The first provides the boolean `flag` variable that’s used in the conditional expression in the body of the lambda to decide whether to return a populated `Optional` or an empty one. This is clearly intended to protect against the exception.

However, the second argument to the `compose.apply` method must be evaluated before the method can be invoked. Since that argument is evaluated by calling `this.getProperty(key)`, the exception will be thrown before the code ever enters the body of the lambda. Look at the following code, which is the content of the `getProperty` method that overrides the version in the base `Properties` class, and notice that it throws the `ConfigException` if the `super.getProperty(key)` call returns `null`.

```java
String property = super.getProperty(key);
if (property == null) {
    throw new ConfigException(String.
```
The cleanest solution to this problem would be to avoid evaluating `this.getProperty(key)` unless you know that the configuration contains a matching value. There are, of course, several ways to approach this, but only one of the options offered can work. That’s the only one that can be the correct answer to a quiz question in this form.

The workable option is to pass the second argument in an unevaluated form. A phrase used to describe this is “to evaluate that second argument lazily.” You can do this using the approach in option D, which suggests that instead of passing the result of calling `this.getProperty(key)` you pass a `Supplier` that embeds that behavior, so that if the conditional expression in the `compose.apply` lambda body decides to invoke the `Supplier` class’s `get` method then, but only then, will the `getProperty` method be called.

To build out this approach, change the argument to the lambda from `String` to `Supplier<String>` and change the implementation of the conditional expression, as follows:

```java
BiFunction<Boolean, Supplier<String>, Optional<String>> compose = (flag, val) -> {
    return flag ? Optional.of(val.get()) : Optional.empty();
};
```

Then change the invocation of `getConfigString` to pass a lambda instead of the result of the `getProperty` call. That change looks like the following:

```java
public Optional<String> getConfigString(String key) {
    return compose.apply(this.containsKey(key), () -> this.getProperty(key));
}
```

Then, if the key is missing, the conditional expression will not attempt to invoke `getProperty(key)` and the `ConfigException` will never be thrown. Based on the explanation above, you can conclude that option D is correct.

There are a couple of side notes that should be added here.

- What’s shown here is a pattern that’s sometimes called a *thunk*. This mechanism is provided in a neatly packaged syntax in some of the more functionally oriented programming languages, where it might be referred to as *pass-by-name* parameter passing.
- Under some circumstances, this lazy evaluation approach can save a very significant amount of unnecessary work (either computational or I/O work, or perhaps even other
Let's review the options that were rejected.

Option C suggests that some values returned for some keys might be null and, therefore, the Optional.of(val) part will fail. In fact, the java.util.Property class that provides the basic storage facility for the Config class code will reject an attempt to store null values in the first place, so this problem never arises for any key that is found. Further, the code in the following conditional expression offers protection in the case of any nonexistent key:

```
return flag ? Optional.of(val) : Optional.empty;
```

From this, it's clear that option C is incorrect.

Option B suggests protecting against writing null values into the store in the first place. In fact, the documentation (albeit somewhat indirectly) tells you that the Properties object will throw an exception if you attempt to store either a null key or a null value. So, if the storage of such an item is already impossible, it cannot be the cause of the problem that arises when reading the value. Note that the question specifically mentions that the problem arises when the code callsgetConfigString() with nonnull keys. From this you can see that option B is incorrect.

Option A is also incorrect, because the java.util.Properties class is documented as thread-safe. Since all the storage for the Config class code simply delegates to the Properties from which it extends, it's not reasonable to think that there is any kind of concurrency problem.

However, without recourse to the API documentation (and a close reading) this might be rather obscure knowledge. A more compelling reason to reject option A is that there is nothing in the question that suggests there is any threading in this codebase. Therefore, this option is simply not applicable. These two reasons should lead you to the conclusion that option A is incorrect.

One final note: The Properties class is a subclass of Hashtable, which is also documented as thread-safe but is not recommended for new projects. The API documentation recommends using either HashMap if thread safety is not
needed or `ConcurrentHashMap` if it is. Interestingly, the current implementation of `Properties` bypasses the parent `Hashtable` for storage, and it delegates to an instance of `ConcurrentHashMap` instead.

Conclusion. The correct answer is option D.

---

**Simon Roberts**

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**

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of 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.