Quiz yourself: Security threats and malicious code modifications

JAVA SE

Quiz yourself: Security threats and malicious code modifications

Here's what happens when good code meets bad people.

by Simon Roberts and Mikalai Zaikin

December 21, 2020

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.

The objective of this Java SE 11 quiz is to develop code that mitigates security threats, such as denial of service attacks or code injection.

Here's your secret mission: You are developing a highly loaded, multithreaded stock-quote analysis application. During a security audit, you were advised to make a defensive copy of stock-quote data as you receive it for analysis, isolating the calculations from any modifications made to the original input data.

Given this `StockQuote` class:

```
class StockQuote {
    public StockQuote(String n, LocalDateTime
        name = n;
        time = t;
        price = p;
    }

    private  String name;
    private  LocalDateTime time;
    private  Double price;
```

```
    // getters and setters hidden for brevity
}
```

and this method, which should protect itself by making a copy:

```
public void analyzeQuotes(ArrayList<StockQuot
    List<StockQuote> sqListProtected = … // m
}
```

Which code line (if any) can protect you from malicious modifications made to the stock-quote data during the processing? Choose one.

A. `List<StockQuote>)sqList.clone();`

The answer is A.

B. `new ArrayList<StockQuote>(sqList);`

The answer is B.

C.

```
(List)sqList.stream()
    .map(s -> s.clone())
    .collect(Collectors.toList());
```

The answer is C.

D.

```
sqList.stream()
    .map(s -> s.clone())
    .collect(Collectors.toCollection(ArrayLis
```

The answer is D.

E. None of the above

The answer is E.

Answer. In option A, the `clone()` method of `ArrayList` will perform "shallow" cloning. This means that the result will be a new list structure, but the values in that list will be references to the same objects as were referred to by the original list. Because of this, any changes made to the items in the original list will be visible through the new list. The cloning operation will protect only against additions, deletions, or changes of sequential order made to the original list.

This observation about shallow cloning applies not only to `ArrayList` but also to all objects that use the default implementation of `Object.clone()` and contain references to other mutable objects. If an object contains only primitives or references to immutable instances such as `String`, the cloned copy will be protected against changes made to the original list. From this you can determine that option A is incorrect.

If you were unsure whether the `clone()` method performs a shallow or deep copy, there's another hint that could help to determine that the code in option A doesn't work. Because the `StockQuote` class does not implement the `java.lang.Cloneable` marker interface, it cannot be cloned. Therefore, if `ArrayList` attempted a deep clone—including duplicating its member objects—the attempt would throw a `java.lang.CloneNotSupportedException`.

Option B is essentially equivalent to option A. Passing a collection to the `ArrayList` constructor will create a new list structure containing the same reference values, in the same order, as in the original collection. But the `StockQuote` objects are shared between both, and changes made to any of those objects via `sqList` will result in changes visible through the new list. Because of this, option B is also incorrect.

It's worth noting that the approach demonstrated in option B can be used to produce a different type of collection: perhaps changing between an unmodifiable collection (that is, one that disallows adding or removing elements or changing their order) and a fully modifiable collection.

There are three problems with option C:

- The `Object.clone()` method is declared as `protected`, so it can be called only from a subclass using the syntax `super.clone()` or `this.clone()` but not from some arbitrary object reference in the manner of `s.clone()` shown in the code.
- Even if the `s.clone()` code were valid, the `Object.clone` method throws a checked exception `java.lang.CloneNotSupportedException`. But the argument to the `map` method must be a `java.util.function.Function<A,B>` implementation, and the `apply()` method of that functional interface does not permit checked exceptions. Hence, the code would still not compile.
- Even if the first two problems didn't exist and the code could compile, it would still fail to execute. This is because, as was noted earlier, the `StockQuote` class does not permit cloning, so the `clone` method would definitely fail and throw a `CloneNotSupportedException`. This exception would crash the stream operation.

In view of all these problems, it's clear that option C is incorrect.

Option D has all the same problems as with option C, so option D is also incorrect.

In fact, option D has another difficulty, which is that the `Object.clone()` method returns `Object`. So even if all three problems from option C were somehow avoided, the `collect` operation would result in an `ArrayList<Object>`, and that's not assignment-compatible with the variable in the method, which is of type `List<StockQuote>`.

Since none of the options A–D provide a required solution, this makes option E correct.

Before leaving this quiz, let's consider how this code *could* be protected. The fields in the `StockQuote` class are of type `String`, `LocalDateTime`, and `Double`. Each of these are classes that create immutable instances, so it's safe for two `StockQuote` objects to share a reference to the same attributes. So, one approach is to create copies of the `StockQuote` objects and a new list to contain those copies.

You could pursue a solution that would work when used to build on the code of option C (note this is not the only possible solution). That code is built on the assumption that the `StockQuote` items can be cloned in the `map` method. So first you must ensure that `StockQuote` conforms to that expectation. This requires two steps:

1. The class implements the `Cloneable` interface (this grants permission to the JVM to perform cloning)
2. You provide a useful implementation of the `clone` method

Step 1 is simply achieved like this:

```
class StockQuote implements Cloneable {
```

At this point, it's possible to clone the `StockQuote` objects successfully. However `StockQuote` inherits the default implementation of `clone()` from `Object`, and this throws `CloneNotSupportedException`, which is a checked exception.

Checked exceptions are incompatible with Java's functional interfaces. In this code, if the exception arises, it is essentially unrecoverable, so create an overriding `clone()` implementation and wrap the exception in a `RuntimeException`.

There are three further steps for how you implement the `clone` method:

1. The default `clone` method is `protected`. However, you want to be able to call this from inside the `map` method. Consequently you must make this `public`.
2. The default `clone` method makes a shallow copy; that is, it copies all the member variables. In this situation, this is acceptable since all the fields are immutable objects (`String`, `LocalDateTime`, and `Double`). So, the method will delegate to `super.clone()` to do the bulk of its work.

3. The default `clone` method is declared to return `Object`, but Java permits an overriding method to provide a covariant return. That means that you can return a more specific type: something assignment-compatible with the overridden method's declared type. In your code, the sensible choice is to return the `StockOption` type. After all, that's what the object will actually be. Doing so avoids creating a raw `List`, and it allows you to remove the raw cast to `List` in option C. It also allows option D to work.

After all that, your implementation of `clone()` looks like this.

```
@Override
public StockQuote clone() {
    try {
        return (StockQuote)super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException(e);
    }
}
```

Unfortunately, even now, you're still not done! You still cannot expect objects that are presented to you as `StockQuote` instances to reliably clone as intended. Certainly, actual instances of the `StockQuote` class presented here will clone as you have defined; that part is correct.

However, you have not precluded an attacker from creating a subclass of `StockQuote`. Such a class could override the `clone()` method so that it simply returns itself (the `this` reference). In so doing, it would completely bypass all your good intentions. So, your last step should be to make the class itself, or at least the `clone()` method, `final`.

By the way, control of subclassing is a focus of the sealed classes feature that's previewed in Java 15. This allows the creator of a class to explicitly define the permitted subclasses, so an inheritance tree can exist, but it prevents arbitrary additions to that tree.

[You can read about sealed classes in "Inside the language: Sealed types," by Ben Evans. —*Ed*.]

In closing, we must acknowledge that the exact nature of the security concerns isn't clearly stated in the question, and other excellent approaches to this are possible. But this was meant to be a discussion, not a full analysis (which would require a book!). So we'll wrap this up here.

Conclusion: The correct answer is option E.

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.