CHAPTER 12

Serialization

THIS chapter concerns *object serialization*, which is Java's framework for encoding objects as byte streams (*serializing*) and reconstructing objects from their encodings (*deserializing*). Once an object has been serialized, its encoding can be sent from one VM to another or stored on disk for later deserialization. This chapter focuses on the dangers of serialization and how to minimize them.

Item 85: Prefer alternatives to Java serialization

When serialization was added to Java in 1997, it was known to be somewhat risky. The approach had been tried in a research language (Modula-3) but never in a production language. While the promise of distributed objects with little effort on the part of the programmer was appealing, the price was invisible constructors and blurred lines between API and implementation, with the potential for problems with correctness, performance, security, and maintenance. Proponents believed the benefits outweighed the risks, but history has shown otherwise.

The security issues described in previous editions of this book turned out to be every bit as serious as some had feared. The vulnerabilities discussed in the early 2000s were transformed into serious exploits over the next decade, famously including a ransomware attack on the San Francisco Metropolitan Transit Agency Municipal Railway (SFMTA Muni) that shut down the entire fare collection system for two days in November 2016 [Gallagher16].

A fundamental problem with serialization is that its *attack surface* is too big to protect, and constantly growing: Object graphs are deserialized by invoking the `readObject` method on an `ObjectInputStream`. This method is essentially a magic constructor that can be made to instantiate objects of almost any type on the class path, so long as the type implements the `Serializable` interface. In the process of deserializing a byte stream, this method can execute code from any of these types, so the code for *all* of these types is part of the attack surface.

The attack surface includes classes in the Java platform libraries, in third-party libraries such as Apache Commons Collections, and in the application itself. Even if you adhere to all of the relevant best practices and succeed in writing serializable classes that are invulnerable to attack, your application may still be vulnerable. To quote Robert Seacord, technical manager of the CERT Coordination Center:

> Java deserialization is a clear and present danger as it is widely used both directly by applications and indirectly by Java subsystems such as RMI (Remote Method Invocation), JMX (Java Management Extension), and JMS (Java Messaging System). Deserialization of untrusted streams can result in remote code execution (RCE), denial-of-service (DoS), and a range of other exploits. Applications can be vulnerable to these attacks even if they did nothing wrong. [Seacord17]

Attackers and security researchers study the serializable types in the Java libraries and in commonly used third-party libraries, looking for methods invoked during deserialization that perform potentially dangerous activities. Such methods are known as *gadgets*. Multiple gadgets can be used in concert, to form a *gadget chain*. From time to time, a gadget chain is discovered that is sufficiently powerful to allow an attacker to execute arbitrary native code on the underlying hardware, given only the opportunity to submit a carefully crafted byte stream for deserialization. This is exactly what happened in the SFMTA Muni attack. This attack was not isolated. There have been others, and there will be more.

Without using any gadgets, you can easily mount a denial-of-service attack by causing the deserialization of a short stream that requires a long time to deserialize. Such streams are known as *deserialization bombs* [Svoboda16]. Here's an example by Wouter Coekaerts that uses only hash sets and a string [Coekaerts15]:

```
// Deserialization bomb - deserializing this stream takes forever
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // Make t1 unequal to t2
        s1.add(t1);  s1.add(t2);
        s2.add(t1);  s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // Method omitted for brevity
}
```

The object graph consists of 201 `HashSet` instances, each of which contains 3 or fewer object references. The entire stream is 5,744 bytes long, yet the sun would burn out long before you could deserialize it. The problem is that deserializing a `HashSet` instance requires computing the hash codes of its elements. The 2 elements of the root hash set are themselves hash sets containing 2 hash-set elements, each of which contains 2 hash-set elements, and so on, 100 levels deep. Therefore, deserializing the set causes the `hashCode` method to be invoked over 2^{100} times. Other than the fact that the deserialization is taking forever, the deserializer has no indication that anything is amiss. Few objects are produced, and the stack depth is bounded.

So what can you do defend against these problems? You open yourself up to attack whenever you deserialize a byte stream that you don't trust. **The best way to avoid serialization exploits is never to deserialize anything.** In the words of the computer named Joshua in the 1983 movie *WarGames*, "the only winning move is not to play." **There is no reason to use Java serialization in any new system you write.** There are other mechanisms for translating between objects and byte sequences that avoid many of the dangers of Java serialization, while offering numerous advantages, such as cross-platform support, high performance, a large ecosystem of tools, and a broad community of expertise. In this book, we refer to these mechanisms as *cross-platform structured-data representations.* While others sometimes refer to them as serialization systems, this book avoids that usage to prevent confusion with Java serialization.

What these representations have in common is that they're *far* simpler than Java serialization. They don't support automatic serialization and deserialization of arbitrary object graphs. Instead, they support simple, structured data-objects consisting of a collection of attribute-value pairs. Only a few primitive and array data types are supported. This simple abstraction turns out to be sufficient for building extremely powerful distributed systems and simple enough to avoid the serious problems that have plagued Java serialization since its inception.

The leading cross-platform structured data representations are JSON [JSON] and Protocol Buffers, also known as protobuf [Protobuf]. JSON was designed by Douglas Crockford for browser-server communication, and protocol buffers were designed by Google for storing and interchanging structured data among its servers. Even though these representations are sometimes called *language-neutral*, JSON was originally developed for JavaScript and protobuf for C++; both representations retain vestiges of their origins.

The most significant differences between JSON and protobuf are that JSON is text-based and human-readable, whereas protobuf is binary and substantially more efficient; and that JSON is exclusively a data representation, whereas protobuf offers *schemas* (types) to document and enforce appropriate usage. Although protobuf is more efficient than JSON, JSON is extremely efficient for a text-based representation. And while protobuf is a binary representation, it does provide an alternative text representation for use where human-readability is desired (pbtxt).

If you can't avoid Java serialization entirely, perhaps because you're working in the context of a legacy system that requires it, your next best alternative is to **never deserialize untrusted data.** In particular, you should never accept RMI traffic from untrusted sources. The official secure coding guidelines for Java say "Deserialization of untrusted data is inherently dangerous and should be avoided." This sentence is set in large, bold, italic, red type, and it is the only text in the entire document that gets this treatment [Java-secure].

If you can't avoid serialization and you aren't absolutely certain of the safety of the data you're deserializing, use the object deserialization filtering added in Java 9 and backported to earlier releases (`java.io.ObjectInputFilter`). This facility lets you specify a filter that is applied to data streams before they're deserialized. It operates at the class granularity, letting you accept or reject certain classes. Accepting classes by default and rejecting a list of potentially dangerous ones is known as *blacklisting*; rejecting classes by default and accepting a list of those that are presumed safe is known as *whitelisting*. **Prefer whitelisting to blacklisting**, as blacklisting only protects you against known threats. A tool called Serial Whitelist Application Trainer (SWAT) can be used to automatically prepare a whitelist for your application [Schneider16]. The filtering facility will also protect you against excessive memory usage, and excessively deep object graphs, but it will not protect you against serialization bombs like the one shown above.

Unfortunately, serialization is still pervasive in the Java ecosystem. If you are maintaining a system that is based on Java serialization, seriously consider migrating to a cross-platform structured-data representation, even though this may be a time-consuming endeavor. Realistically, you may still find yourself having to write or maintain a serializable class. It requires great care to write a serializable class that is correct, safe, and efficient. The remainder of this chapter provides advice on when and how to do this.

In summary, serialization is dangerous and should be avoided. If you are designing a system from scratch, use a cross-platform structured-data representation such as JSON or protobuf instead. Do not deserialize untrusted data. If you must do so, use object deserialization filtering, but be aware that it is not guaranteed to thwart all attacks. Avoid writing serializable classes. If you must do so, exercise great caution.

Item 86: Implement `Serializable` with great caution

Allowing a class's instances to be serialized can be as simple as adding the words `implements Serializable` to its declaration. Because this is so easy to do, there was a common misconception that serialization requires little effort on the part of the programmer. The truth is far more complex. While the immediate cost to make a class serializable can be negligible, the long-term costs are often substantial.

A major cost of implementing `Serializable` is that it decreases the flexibility to change a class's implementation once it has been released. When a class implements `Serializable`, its byte-stream encoding (or *serialized form*) becomes part of its exported API. Once you distribute a class widely, you are generally required to support the serialized form forever, just as you are required to support all other parts of the exported API. If you do not make the effort to design a *custom serialized form* but merely accept the default, the serialized form will forever be tied to the class's original internal representation. In other words, if you accept the default serialized form, the class's private and package-private instance fields become part of its exported API, and the practice of minimizing access to fields (Item 15) loses its effectiveness as a tool for information hiding.

If you accept the default serialized form and later change a class's internal representation, an incompatible change in the serialized form will result. Clients attempting to serialize an instance using an old version of the class and deserialize it using the new one (or vice versa) will experience program failures. It is possible to change the internal representation while maintaining the original serialized form (using `ObjectOutputStream.putFields` and `ObjectInputStream.readFields`), but it can be difficult and leaves visible warts in the source code. If you opt to make a class serializable, you should carefully design a high-quality serialized form that you're willing to live with for the long haul (Items 87, 90). Doing so will add to the initial cost of development, but it's worth the effort. Even a well-designed serialized form places constraints on the evolution of a class; an ill-designed serialized form can be crippling.

A simple example of the constraints on evolution imposed by serializability concerns *stream unique identifiers*, more commonly known as *serial version UIDs*. Every serializable class has a unique identification number associated with it. If you do not specify this number by declaring a static final `long` field named `serialVersionUID`, the system automatically generates it at runtime by applying a cryptographic hash function (SHA-1) to the structure of the class. This value is affected by the names of the class, the interfaces it implements, and most of its members, including synthetic members generated by the compiler. If you change any of these things, for example, by adding a convenience method, the generated serial version UID changes. If you fail to declare a serial version UID, compatibility will be broken, resulting in an `InvalidClassException` at runtime.

A second cost of implementing `Serializable` is that it increases the likelihood of bugs and security holes (Item 85). Normally, objects are created with constructors; serialization is an *extralinguistic mechanism* for creating objects. Whether you accept the default behavior or override it, deserialization is a “hidden constructor” with all of the same issues as other constructors. Because there is no explicit constructor associated with deserialization, it is easy to forget that you must ensure that it guarantees all of the invariants established by the constructors and that it does not allow an attacker to gain access to the internals of the object under construction. Relying on the default deserialization mechanism can easily leave objects open to invariant corruption and illegal access (Item 88).

A third cost of implementing `Serializable` is that it increases the testing burden associated with releasing a new version of a class. When a serializable class is revised, it is important to check that it is possible to serialize an instance in the new release and deserialize it in old releases, and vice versa. The amount of testing required is thus proportional to the product of the number of serializable classes and the number of releases, which can be large. You must ensure both that the serialization-deserialization process succeeds and that it results in a faithful replica of the original object. The need for testing is reduced if a custom serialized form is carefully designed when the class is first written (Items 87, 90).

Implementing `Serializable` is not a decision to be undertaken lightly. It is essential if a class is to participate in a framework that relies on Java serialization for object transmission or persistence. Also, it greatly eases the use of a class as a component in another class that must implement `Serializable`. There are, however, many costs associated with implementing `Serializable`. Each time you design a class, weigh the costs against the benefits. Historically, value classes such as `BigInteger` and `Instant` implemented `Serializable`, and collection classes did too. Classes representing active entities, such as thread pools, should rarely implement `Serializable`.

Classes designed for inheritance (Item 19) should rarely implement `Serializable`, and interfaces should rarely extend it. Violating this rule places a substantial burden on anyone who extends the class or implements the interface. There are times when it is appropriate to violate the rule. For example, if a class or interface exists primarily to participate in a framework that requires all participants to implement `Serializable`, then it may make sense for the class or interface to implement or extend `Serializable`.

Classes designed for inheritance that do implement `Serializable` include `Throwable` and `Component`. `Throwable` implements `Serializable` so RMI can send exceptions from server to client. `Component` implements `Serializable` so GUIs can be sent, saved, and restored, but even in the heyday of Swing and AWT, this facility was little-used in practice.

If you implement a class with instance fields that is both serializable and extendable, there are several risks to be aware of. If there are any invariants on the instance field values, it is critical to prevent subclasses from overriding the `finalize` method, which the class can do by overriding `finalize` and declaring it final. Otherwise, the class will be susceptible to *finalizer attacks (*Item 8). Finally, if the class has invariants that would be violated if its instance fields were initialized to their default values (zero for integral types, `false` for `boolean`, and `null` for object reference types), you must add this `readObjectNoData` method:

```
// readObjectNoData for stateful extendable serializable classes
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

This method was added in Java 4 to cover a corner case involving the addition of a serializable superclass to an existing serializable class [Serialization, 3.5].

There is one caveat regarding the decision *not* to implement `Serializable`. If a class designed for inheritance is not serializable, it may require extra effort to write a serializable subclass. Normal deserialization of such a class requires the superclass to have an accessible parameterless constructor [Serialization, 1.10]. If you don't provide such a constructor, subclasses are forced to use the serialization proxy pattern (Item 90).

Inner classes (Item 24) should not implement `Serializable`. They use compiler-generated *synthetic fields* to store references to *enclosing instances* and to store values of local variables from enclosing scopes. How these fields correspond to the class definition is unspecified, as are the names of anonymous and local classes. Therefore, the default serialized form of an inner class is ill-defined. A *static member class* can, however, implement `Serializable`.

To summarize, the ease of implementing `Serializable` is specious. Unless a class is to be used only in a protected environment where versions will never have to interoperate and servers will never be exposed to untrusted data, implementing `Serializable` is a serious commitment that should be made with great care. Extra caution is warranted if a class permits inheritance.

Item 87: Consider using a custom serialized form

When you are writing a class under time pressure, it is generally appropriate to concentrate your efforts on designing the best API. Sometimes this means releasing a "throwaway" implementation that you know you'll replace in a future release. Normally this is not a problem, but if the class implements `Serializable` and uses the default serialized form, you'll never be able to escape completely from the throwaway implementation. It will dictate the serialized form forever. This is not just a theoretical problem. It happened to several classes in the Java libraries, including `BigInteger`.

Do not accept the default serialized form without first considering whether it is appropriate. Accepting the default serialized form should be a conscious decision that this encoding is reasonable from the standpoint of flexibility, performance, and correctness. Generally speaking, you should accept the default serialized form only if it is largely identical to the encoding that you would choose if you were designing a custom serialized form.

The default serialized form of an object is a reasonably efficient encoding of the *physical* representation of the object graph rooted at the object. In other words, it describes the data contained in the object and in every object that is reachable from this object. It also describes the topology by which all of these objects are interlinked. The ideal serialized form of an object contains only the *logical* data represented by the object. It is independent of the physical representation.

The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content. For example, the default serialized form would be reasonable for the following class, which simplistically represents a person's name:

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private final String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private final String firstName;
    /**
     * Middle name, or null if there is none.
     * @serial
     */
    private final String middleName;

    ... // Remainder omitted
}
```

Logically speaking, a name consists of three strings that represent a last name, a first name, and a middle name. The instance fields in `Name` precisely mirror this logical content.

Even if you decide that the default serialized form is appropriate, you often must provide a `readObject` method to ensure invariants and security. In the case of `Name`, the `readObject` method must ensure that the fields `lastName` and `firstName` are non-null. This issue is discussed at length in Items 88 and 90.

Note that there are documentation comments on the `lastName`, `firstName`, and `middleName` fields, even though they are private. That is because these private fields define a public API, which is the serialized form of the class, and this public API must be documented. The presence of the `@serial` tag tells Javadoc to place this documentation on a special page that documents serialized forms.

Near the opposite end of the spectrum from `Name`, consider the following class, which represents a list of strings (ignoring for the moment that you would probably be better off using one of the standard `List` implementations):

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry  next;
        Entry  previous;
    }

    ... // Remainder omitted
}
```

Logically speaking, this class represents a sequence of strings. Physically, it represents the sequence as a doubly linked list. If you accept the default serialized form, the serialized form will painstakingly mirror every entry in the linked list and all the links between the entries, in both directions.

Using the default serialized form when an object's physical representation differs substantially from its logical data content has four disadvantages:

- **It permanently ties the exported API to the current internal representation.** In the above example, the private `StringList.Entry` class becomes part of the public API. If the representation is changed in a future release, the `StringList` class will still need to accept the linked list representation on input and generate it on output. The class will never be rid of all the code dealing with linked list entries, even if it doesn't use them anymore.
- **It can consume excessive space.** In the above example, the serialized form unnecessarily represents each entry in the linked list and all the links. These entries and links are mere implementation details, not worthy of inclusion in the serialized form. Because the serialized form is excessively large, writing it to disk or sending it across the network will be excessively slow.
- **It can consume excessive time.** The serialization logic has no knowledge of the topology of the object graph, so it must go through an expensive graph traversal. In the example above, it would be sufficient simply to follow the `next` references.
- **It can cause stack overflows.** The default serialization procedure performs a recursive traversal of the object graph, which can cause stack overflows even for moderately sized object graphs. Serializing a `StringList` instance with 1,000–1,800 elements generates a `StackOverflowError` on my machine. Surprisingly, the minimum list size for which serialization causes a stack overflow varies from run to run (on my machine). The minimum list size that exhibits this problem may depend on the platform implementation and command-line flags; some implementations may not have this problem at all.

A reasonable serialized form for `StringList` is simply the number of strings in the list, followed by the strings themselves. This constitutes the logical data represented by a `StringList`, stripped of the details of its physical representation. Here is a revised version of `StringList` with `writeObject` and `readObject` methods that implement this serialized form. As a reminder, the `transient` modifier indicates that an instance field is to be omitted from a class's default serialized form:

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size   = 0;
    private transient Entry head = null;

    // No longer Serializable!
    private static class Entry {
        String data;
        Entry  next;
        Entry  previous;
    }

    // Appends the specified string to the list
    public final void add(String s) { ... }

    /**
     * Serialize this {@code StringList} instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted ({@code int}), followed by all of
     * its elements (each a {@code String}), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
            throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        int numElements = s.readInt();

        // Read in all elements and insert them in list
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }

    ... // Remainder omitted
}
```

The first thing `writeObject` does is to invoke `defaultWriteObject`, and the first thing `readObject` does is to invoke `defaultReadObject`, even though all of `StringList`'s fields are transient. You may hear it said that if all of a class's instance fields are transient, you can dispense with invoking `defaultWriteObject` and `defaultReadObject`, but the serialization specification requires you to invoke them regardless. The presence of these calls makes it possible to add nontransient instance fields in a later release while preserving backward and forward compatibility. If an instance is serialized in a later version and deserialized in an earlier version, the added fields will be ignored. Had the earlier version's `readObject` method failed to invoke `defaultReadObject`, the deserialization would fail with a `StreamCorruptedException`.

Note that there is a documentation comment on the `writeObject` method, even though it is private. This is analogous to the documentation comment on the private fields in the `Name` class. This private method defines a public API, which is the serialized form, and that public API should be documented. Like the `@serial` tag for fields, the `@serialData` tag for methods tells the Javadoc utility to place this documentation on the serialized forms page.

To lend some sense of scale to the earlier performance discussion, if the average string length is ten characters, the serialized form of the revised version of `StringList` occupies about half as much space as the serialized form of the original. On my machine, serializing the revised version of `StringList` is over twice as fast as serializing the original version, with a list length of ten. Finally, there is no stack overflow problem in the revised form and hence no practical upper limit to the size of `StringList` that can be serialized.

While the default serialized form would be bad for `StringList`, there are classes for which it would be far worse. For `StringList`, the default serialized form is inflexible and performs badly, but it is *correct* in the sense that serializing and deserializing a `StringList` instance yields a faithful copy of the original object with all of its invariants intact. This is not the case for any object whose invariants are tied to implementation-specific details.

For example, consider the case of a hash table. The physical representation is a sequence of hash buckets containing key-value entries. The bucket that an entry resides in is a function of the hash code of its key, which is not, in general, guaranteed to be the same from implementation to implementation. In fact, it isn't even guaranteed to be the same from run to run. Therefore, accepting the default serialized form for a hash table would constitute a serious bug. Serializing and deserializing the hash table could yield an object whose invariants were seriously corrupt.

Whether or not you accept the default serialized form, every instance field that isn't labeled `transient` will be serialized when the `defaultWriteObject` method is invoked. Therefore, every instance field that can be declared transient should be. This includes derived fields, whose values can be computed from primary data fields, such as a cached hash value. It also includes fields whose values are tied to one particular run of the JVM, such as a `long` field representing a pointer to a native data structure. **Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object.** If you use a custom serialized form, most or all of the instance fields should be labeled `transient`, as in the `StringList` example above.

If you are using the default serialized form and you have labeled one or more fields `transient`, remember that these fields will be initialized to their *default values* when an instance is deserialized: `null` for object reference fields, zero for numeric primitive fields, and `false` for `boolean` fields [JLS, 4.12.5]. If these values are unacceptable for any transient fields, you must provide a `readObject` method that invokes the `defaultReadObject` method and then restores transient fields to acceptable values (Item 88). Alternatively, these fields can be lazily initialized the first time they are used (Item 83).

Whether or not you use the default serialized form, **you must impose any synchronization on object serialization that you would impose on any other method that reads the entire state of the object.** So, for example, if you have a thread-safe object (Item 82) that achieves its thread safety by synchronizing every method and you elect to use the default serialized form, use the following `write-Object` method:

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();
}
```

If you put synchronization in the `writeObject` method, you must ensure that it adheres to the same lock-ordering constraints as other activities, or you risk a resource-ordering deadlock [Goetz06, 10.1.5].

Regardless of what serialized form you choose, declare an explicit serial version UID in every serializable class you write. This eliminates the serial version UID as a potential source of incompatibility (Item 86). There is also a small performance benefit. If no serial version UID is provided, an expensive computation is performed to generate one at runtime.

Declaring a serial version UID is simple. Just add this line to your class:

```
private static final long serialVersionUID = randomLongValue;
```

If you write a new class, it doesn't matter what value you choose for *randomLongValue*. You can generate the value by running the `serialver` utility on the class, but it's also fine to pick a number out of thin air. It is *not* required that serial version UIDs be unique. If you modify an existing class that lacks a serial version UID, and you want the new version to accept existing serialized instances, you must use the value that was automatically generated for the old version. You can get this number by running the `serialver` utility on the old version of the class—the one for which serialized instances exist.

If you ever want to make a new version of a class that is *incompatible* with existing versions, merely change the value in the serial version UID declaration. This will cause attempts to deserialize serialized instances of previous versions to throw an `InvalidClassException`. **Do not change the serial version UID unless you want to break compatibility with all existing serialized instances of a class.**

To summarize, if you have decided that a class should be serializable (Item 86), think hard about what the serialized form should be. Use the default serialized form *only* if it is a reasonable description of the logical state of the object; otherwise design a custom serialized form that aptly describes the object. You should allocate as much time to designing the serialized form of a class as you allocate to designing an exported method (Item 51). Just as you can't eliminate exported methods from future versions, you can't eliminate fields from the serialized form; they must be preserved forever to ensure serialization compatibility. Choosing the wrong serialized form can have a permanent, negative impact on the complexity and performance of a class.

Item 88: Write `readObject` methods defensively

Item 50 contains an immutable date-range class with mutable private `Date` fields. The class goes to great lengths to preserve its invariants and immutability by defensively copying `Date` objects in its constructor and accessors. Here is the class:

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                              start + " after " + end);
    }

    public Date start () { return new Date(start.getTime()); }

    public Date end () { return new Date(end.getTime()); }

    public String toString() { return start + " - " + end; }

    ... // Remainder omitted
}
```

Suppose you decide that you want this class to be serializable. Because the physical representation of a `Period` object exactly mirrors its logical data content, it is not unreasonable to use the default serialized form (Item 87). Therefore, it might seem that all you have to do to make the class serializable is to add the words `implements Serializable` to the class declaration. If you did so, however, the class would no longer guarantee its critical invariants.

The problem is that the `readObject` method is effectively another public constructor, and it demands all of the same care as any other constructor. Just as a constructor must check its arguments for validity (Item 49) and make defensive copies of parameters where appropriate (Item 50), so must a `readObject` method. If a `readObject` method fails to do either of these things, it is a relatively simple matter for an attacker to violate the class's invariants.

Loosely speaking, readObject is a constructor that takes a byte stream as its sole parameter. In normal use, the byte stream is generated by serializing a normally constructed instance. The problem arises when readObject is presented with a byte stream that is artificially constructed to generate an object that violates the invariants of its class. Such a byte stream can be used to create an *impossible object*, which could not have been created using a normal constructor.

Assume that we simply added implements Serializable to the class declaration for Period. This ugly program would then generate a Period instance whose end precedes its start. The casts on byte values whose high-order bit is set is a consequence of Java's lack of byte literals combined with the unfortunate decision to make the byte type signed:

```
public class BogusPeriod {
  // Byte stream couldn't have come from a real Period instance!
  private static final byte[] serializedForm = {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
    0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
    0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
    0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
    0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
    0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
    (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
    0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
    0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
    0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
    0x00, 0x78
  };

  public static void main(String[] args) {
    Period p = (Period) deserialize(serializedForm);
    System.out.println(p);
  }

  // Returns the object with the specified serialized form
  static Object deserialize(byte[] sf) {
    try {
      return new ObjectInputStream(
          new ByteArrayInputStream(sf)).readObject();
    } catch (IOException | ClassNotFoundException e) {
      throw new IllegalArgumentException(e);
    }
  }
}
```

The byte array literal used to initialize serializedForm was generated by serializing a normal Period instance and hand-editing the resulting byte stream. The details of the stream are unimportant to the example, but if you're curious, the serialization byte-stream format is described in the *Java Object Serialization Specification* [Serialization, 6]. If you run this program, it prints Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984. Simply declaring Period serializable enabled us to create an object that violates its class invariants.

To fix this problem, provide a readObject method for Period that calls defaultReadObject and then checks the validity of the deserialized object. If the validity check fails, the readObject method throws InvalidObjectException, preventing the deserialization from completing:

```
// readObject method with validity checking - insufficient!
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

While this prevents an attacker from creating an invalid Period instance, there is a more subtle problem still lurking. It is possible to create a mutable Period instance by fabricating a byte stream that begins with a valid Period instance and then appends extra references to the private Date fields internal to the Period instance. The attacker reads the Period instance from the ObjectInputStream and then reads the "rogue object references" that were appended to the stream. These references give the attacker access to the objects referenced by the private Date fields within the Period object. By mutating these Date instances, the attacker can mutate the Period instance. The following class demonstrates this attack:

```
public class MutablePeriod {
    // A period instance
    public final Period period;

    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Append rogue "previous object refs" for internal
             * Date fields in Period. For details, see "Java
             * Object Serialization Specification," Section 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4;     // Ref # 4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
                new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start  = (Date)   in.readObject();
            end    = (Date)   in.readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new AssertionError(e);
        }
    }
}
```

To see the attack in action, run the following program:

```
public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;

    // Let's turn back the clock
    pEnd.setYear(78);
    System.out.println(p);

    // Bring back the 60s!
    pEnd.setYear(69);
    System.out.println(p);
}
```

In my locale, running this program produces the following output:

```
Wed Nov 22 00:21:29 PST 2017 - Wed Nov 22 00:21:29 PST 1978
Wed Nov 22 00:21:29 PST 2017 - Sat Nov 22 00:21:29 PST 1969
```

While the `Period` instance is created with its invariants intact, it is possible to modify its internal components at will. Once in possession of a mutable `Period` instance, an attacker might cause great harm by passing the instance to a class that depends on `Period`'s immutability for its security. This is not so far-fetched: there are classes that depend on `String`'s immutability for their security.

The source of the problem is that `Period`'s `readObject` method is not doing enough defensive copying. **When an object is deserialized, it is critical to defensively copy any field containing an object reference that a client must not possess.** Therefore, every serializable immutable class containing private mutable components must defensively copy these components in its `readObject` method. The following `readObject` method suffices to ensure `Period`'s invariants and to maintain its immutability:

```
// readObject method with defensive copying and validity checking
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

Note that the defensive copy is performed prior to the validity check and that we did not use `Date`'s `clone` method to perform the defensive copy. Both of these details are required to protect `Period` against attack (Item 50). Note also that defensive copying is not possible for final fields. To use the `readObject` method, we must make the `start` and `end` fields nonfinal. This is unfortunate, but it is the lesser of two evils. With the new `readObject` method in place and the `final` modifier removed from the `start` and `end` fields, the `MutablePeriod` class is rendered ineffective. The above attack program now generates this output:

```
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
```

Here is a simple litmus test for deciding whether the default `readObject` method is acceptable for a class: would you feel comfortable adding a public constructor that took as parameters the values for each nontransient field in the object and stored the values in the fields with no validation whatsoever? If not, you must provide a `readObject` method, and it must perform all the validity checking and defensive copying that would be required of a constructor. Alternatively, you can use the *serialization proxy pattern* (Item 90). This pattern is highly recommended because it takes much of the effort out of safe deserialization.

There is one other similarity between `readObject` methods and constructors that applies to nonfinal serializable classes. Like a constructor, a `readObject` method must not invoke an overridable method, either directly or indirectly (Item 19). If this rule is violated and the method in question is overridden, the overriding method will run before the subclass's state has been deserialized. A program failure is likely to result [Bloch05, Puzzle 91].

To summarize, anytime you write a `readObject` method, adopt the mind-set that you are writing a public constructor that must produce a valid instance regardless of what byte stream it is given. Do not assume that the byte stream represents an actual serialized instance. While the examples in this item concern a class that uses the default serialized form, all of the issues that were raised apply equally to classes with custom serialized forms. Here, in summary form, are the guidelines for writing a `readObject` method:

- For classes with object reference fields that must remain private, defensively copy each object in such a field. Mutable components of immutable classes fall into this category.
- Check any invariants and throw an `InvalidObjectException` if a check fails. The checks should follow any defensive copying.
- If an entire object graph must be validated after it is deserialized, use the `ObjectInputValidation` interface (not discussed in this book).
- Do not invoke any overridable methods in the class, directly or indirectly.

Item 89: For instance control, prefer enum types to `readResolve`

Item 3 describes the *Singleton* pattern and gives the following example of a singleton class. This class restricts access to its constructor to ensure that only a single instance is ever created:

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

    public void leaveTheBuilding() { ... }
}
```

As noted in Item 3, this class would no longer be a singleton if the words `implements Serializable` were added to its declaration. It doesn't matter whether the class uses the default serialized form or a custom serialized form (Item 87), nor does it matter whether the class provides an explicit `readObject` method (Item 88). Any `readObject` method, whether explicit or default, returns a newly created instance, which will not be the same instance that was created at class initialization time.

The `readResolve` feature allows you to substitute another instance for the one created by `readObject` [Serialization, 3.7]. If the class of an object being deserialized defines a `readResolve` method with the proper declaration, this method is invoked on the newly created object after it is deserialized. The object reference returned by this method is then returned in place of the newly created object. In most uses of this feature, no reference to the newly created object is retained, so it immediately becomes eligible for garbage collection.

If the `Elvis` class is made to implement `Serializable`, the following `readResolve` method suffices to guarantee the singleton property:

```
// readResolve for instance control - you can do better!
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

This method ignores the deserialized object, returning the distinguished `Elvis` instance that was created when the class was initialized. Therefore, the serialized form of an `Elvis` instance need not contain any real data; all instance fields should be declared transient. In fact, **if you depend on `readResolve` for instance**

control, all instance fields with object reference types *must* be declared `transient`. Otherwise, it is possible for a determined attacker to secure a reference to the deserialized object before its `readResolve` method is run, using a technique that is somewhat similar to the `MutablePeriod` attack in Item 88.

The attack is a bit complicated, but the underlying idea is simple. If a singleton contains a nontransient object reference field, the contents of this field will be deserialized before the singleton's `readResolve` method is run. This allows a carefully crafted stream to "steal" a reference to the originally deserialized singleton at the time the contents of the object reference field are deserialized.

Here's how it works in more detail. First, write a "stealer" class that has both a `readResolve` method and an instance field that refers to the serialized singleton in which the stealer "hides." In the serialization stream, replace the singleton's nontransient field with an instance of the stealer. You now have a circularity: the singleton contains the stealer, and the stealer refers to the singleton.

Because the singleton contains the stealer, the stealer's `readResolve` method runs first when the singleton is deserialized. As a result, when the stealer's `readResolve` method runs, its instance field still refers to the partially deserialized (and as yet unresolved) singleton.

The stealer's `readResolve` method copies the reference from its instance field into a static field so that the reference can be accessed after the `readResolve` method runs. The method then returns a value of the correct type for the field in which it's hiding. If it didn't do this, the VM would throw a `ClassCastException` when the serialization system tried to store the stealer reference into this field.

To make this concrete, consider the following broken singleton:

```
// Broken singleton - has nontransient object reference field!
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }

    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

Here is a “stealer” class, constructed as per the description above:

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private Object readResolve() {
        // Save a reference to the "unresolved" Elvis instance
        impersonator = payload;

        // Return object of correct type for favoriteSongs field
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
```

Finally, here is an ugly program that deserializes a handcrafted stream to produce two distinct instances of the flawed singleton. The deserialize method is omitted from this program because it’s identical to the one on page 354:

```
public class ElvisImpersonator {
  // Byte stream couldn't have come from a real Elvis instance!
  private static final byte[] serializedForm = {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
    0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
    (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
    0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
    0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
    0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
    0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
    0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
    0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
    0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
    0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
    0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
  };

  public static void main(String[] args) {
    // Initializes ElvisStealer.impersonator and returns
    // the real Elvis (which is Elvis.INSTANCE)
    Elvis elvis = (Elvis) deserialize(serializedForm);
    Elvis impersonator = ElvisStealer.impersonator;

    elvis.printFavorites();
    impersonator.printFavorites();
  }
}
```

Running this program produces the following output, conclusively proving that it's possible to create two distinct `Elvis` instances (with different tastes in music):

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

You could fix the problem by declaring the `favoriteSongs` field `transient`, but you're better off fixing it by making `Elvis` a single-element enum type (Item 3). As demonstrated by the `ElvisStealer` attack, using a `readResolve` method to prevent a "temporary" deserialized instance from being accessed by an attacker is fragile and demands great care.

If you write your serializable instance-controlled class as an enum, Java guarantees you that there can be no instances besides the declared constants, unless an attacker abuses a privileged method such as `AccessibleObject.setAccessible`. Any attacker who can do that already has sufficient privileges to execute arbitrary native code, and all bets are off. Here's how our `Elvis` example looks as an enum:

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

The use of `readResolve` for instance control is not obsolete. If you have to write a serializable instance-controlled class whose instances are not known at compile time, you will not be able to represent the class as an enum type.

The accessibility of `readResolve` is significant. If you place a `readResolve` method on a final class, it should be private. If you place a `readResolve` method on a nonfinal class, you must carefully consider its accessibility. If it is private, it will not apply to any subclasses. If it is package-private, it will apply only to subclasses in the same package. If it is protected or public, it will apply to all subclasses that do not override it. If a `readResolve` method is protected or public and a subclass does not override it, deserializing a subclass instance will produce a superclass instance, which is likely to cause a `ClassCastException`.

To summarize, use enum types to enforce instance control invariants wherever possible. If this is not possible and you need a class to be both serializable and instance-controlled, you must provide a `readResolve` method and ensure that all of the class's instance fields are either primitive or transient.

Item 90: Consider serialization proxies instead of serialized instances

As mentioned in Items 85 and 86 and discussed throughout this chapter, the decision to implement `Serializable` increases the likelihood of bugs and security problems as it allows instances to be created using an extralinguistic mechanism in place of ordinary constructors. There is, however, a technique that greatly reduces these risks. This technique is known as the *serialization proxy pattern*.

The serialization proxy pattern is reasonably straightforward. First, design a private static nested class that concisely represents the logical state of an instance of the enclosing class. This nested class is known as the *serialization proxy* of the enclosing class. It should have a single constructor, whose parameter type is the enclosing class. This constructor merely copies the data from its argument: it need not do any consistency checking or defensive copying. By design, the default serialized form of the serialization proxy is the perfect serialized form of the enclosing class. Both the enclosing class and its serialization proxy must be declared to implement `Serializable`.

For example, consider the immutable `Period` class written in Item 50 and made serializable in Item 88. Here is a serialization proxy for this class. `Period` is so simple that its serialization proxy has exactly the same fields as the class:

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;

    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }

    private static final long serialVersionUID =
        234098243823485285L; // Any number will do (Item 87)
}
```

Next, add the following `writeReplace` method to the enclosing class. This method can be copied verbatim into any class with a serialization proxy:

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

The presence of this method on the enclosing class causes the serialization system to emit a `SerializationProxy` instance instead of an instance of the enclosing class. In other words, the `writeReplace` method translates an instance of the enclosing class to its serialization proxy prior to serialization.

With this `writeReplace` method in place, the serialization system will never generate a serialized instance of the enclosing class, but an attacker might fabricate one in an attempt to violate the class's invariants. To guarantee that such an attack would fail, merely add this `readObject` method to the enclosing class:

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream)
        throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

Finally, provide a `readResolve` method on the `SerializationProxy` class that returns a logically equivalent instance of the enclosing class. The presence of this method causes the serialization system to translate the serialization proxy back into an instance of the enclosing class upon deserialization.

This `readResolve` method creates an instance of the enclosing class using only its public API and therein lies the beauty of the pattern. It largely eliminates the extralinguistic character of serialization, because the deserialized instance is created using the same constructors, static factories, and methods as any other instance. This frees you from having to separately ensure that deserialized instances obey the class's invariants. If the class's static factories or constructors establish these invariants and its instance methods maintain them, you've ensured that the invariants will be maintained by serialization as well.

Here is the `readResolve` method for `Period.SerializationProxy` above:

```
// readResolve method for Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end);  // Uses public constructor
}
```

Like the defensive copying approach (page 357), the serialization proxy approach stops the bogus byte-stream attack (page 354) and the internal field theft attack (page 356) dead in their tracks. Unlike the two previous approaches, this one allows the fields of `Period` to be final, which is required in order for the `Period` class to be truly immutable (Item 17). And unlike the two previous approaches, this one doesn't involve a great deal of thought. You don't have to figure out which fields might be compromised by devious serialization attacks, nor do you have to explicitly perform validity checking as part of deserialization.

There is another way in which the serialization proxy pattern is more powerful than defensive copying in `readObject`. The serialization proxy pattern allows the deserialized instance to have a different class from the originally serialized instance. You might not think that this would be useful in practice, but it is.

Consider the case of `EnumSet` (Item 36). This class has no public constructors, only static factories. From the client's perspective, they return `EnumSet` instances, but in the current OpenJDK implementation, they return one of two subclasses, depending on the size of the underlying enum type. If the underlying enum type has sixty-four or fewer elements, the static factories return a `RegularEnumSet`; otherwise, they return a `JumboEnumSet`.

Now consider what happens if you serialize an enum set whose enum type has sixty elements, then add five more elements to the enum type, and then deserialize the enum set. It was a `RegularEnumSet` instance when it was serialized, but it had better be a `JumboEnumSet` instance once it is deserialized. In fact that's exactly what happens, because `EnumSet` uses the serialization proxy pattern. In case you're curious, here is `EnumSet`'s serialization proxy. It really is this simple:

```
// EnumSet's serialization proxy
private static class SerializationProxy <E extends Enum<E>>
        implements Serializable {
    // The element type of this enum set.
    private final Class<E> elementType;

    // The elements contained in this enum set.
    private final Enum<?>[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(new Enum<?>[0]);
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum<?> e : elements)
            result.add((E)e);
        return result;
    }

    private static final long serialVersionUID =
        362491234563181265L;
}
```

The serialization proxy pattern has two limitations. It is not compatible with classes that are extendable by their users (Item 19). Also, it is not compatible with some classes whose object graphs contain circularities: if you attempt to invoke a method on such an object from within its serialization proxy's `readResolve` method, you'll get a `ClassCastException` because you don't have the object yet, only its serialization proxy.

Finally, the added power and safety of the serialization proxy pattern are not free. On my machine, it is 14 percent more expensive to serialize and deserialize `Period` instances with serialization proxies than it is with defensive copying.

In summary, consider the serialization proxy pattern whenever you find yourself having to write a `readObject` or `writeObject` method on a class that is not extendable by its clients. This pattern is perhaps the easiest way to robustly serialize objects with nontrivial invariants.