What We Hate About Serialization

And what we might do about it…

Brian Goetz (@briangoetz)
Java Language Architect, Oracle

Stuart Marks (@stuartmarks)
Java Libraries Team, Oracle

November 2019

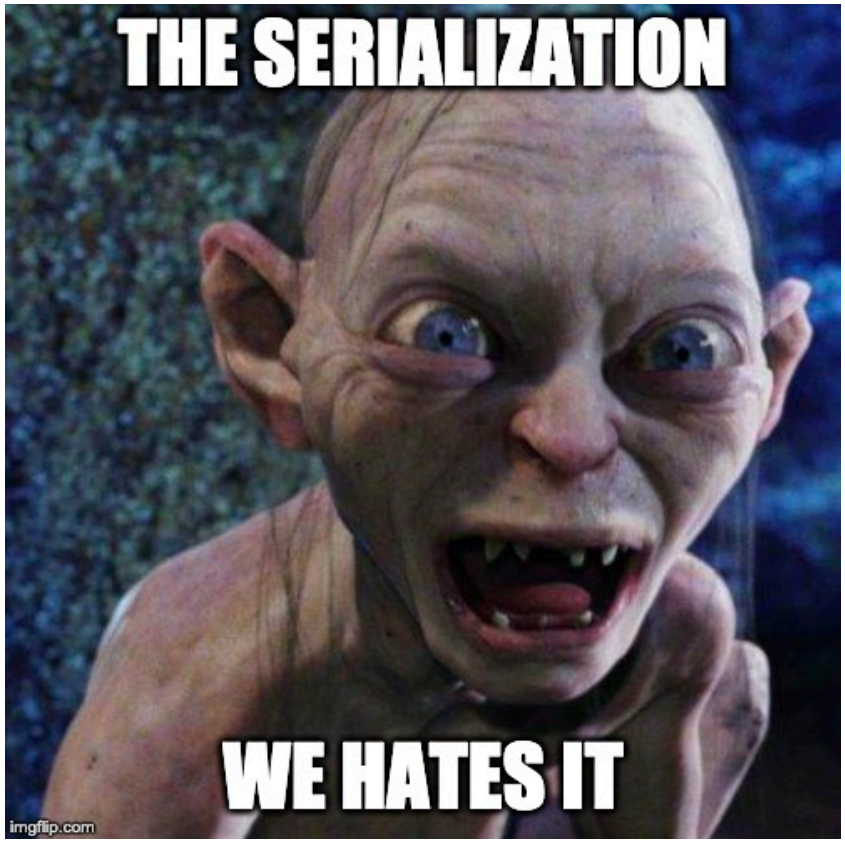
ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Everyone hates serialization

- Joshua Bloch
  - “Prefer alternatives to Java serialization” (*Effective Java*)
- Mark Reinhold
  - “Serialization was a horrible mistake”
- Brian Goetz
  - “Serialization is the gift that keeps on giving”
- Stuart Marks
  - “Serialization is a full-employment act for vulnerability engineers”

Everyone hates serialization



What's with all the hate?

- The *concept* of serialization makes perfect sense
  - Applications need to store documents on disk, in databases, and send them on the wire
  - Java might well not have been successful without serialization!
    - Essential to RMI and EJB remoting
  - *Today's problems come from yesterday's solutions*
- But, the *approach* taken to Java serialization in 1997 was a disaster
  - Undermines encapsulation in non-obvious ways
  - Hard to evolve serializable classes
  - Too hard to reason about security
  - Ongoing tax on evolving the language
  - Cannot verify correctness by reading the code (!)

The benefits...

Out-of-the-box (almost) persistence

- In the 80s and 90s, *persistence* (and later *remoting)* were hard but important problems
  - Every application had to write lots of fiddly, error-prone, ad-hoc code to write out and read in documents
  - In-memory representation was hard to map to stream representation
  - Main promise of “object databases” was eliminating the impedance mismatch between in-memory and offline representation
- “Orthogonal Persistence” was one of the big promises of OO
  - Rather than every application implementing its own ad-hoc, error-prone mechanism, OO systems with introspection could do it *once*
  - This was a *huge* benefit
  - But, at a cost…

... and the costs

Too much to fit on a slide…

- Serialization is *extralinguistic*
  - State is extracted from object via magic field scraping
  - Objects reconstruction bypasses constructors, uses magic field rewriting
  - Bypasses language's accessibility model
- Serialization is *monolithic*
  - Complects graph walking, state extraction, wire encoding, etc – all or nothing
  - Tightly tied to specific (ugly) encoding
- Serialization is *hard for programmers to use*
  - Customization mechanisms are ad-hoc and confusing
    - readObject() is like a constructor, but nothing like a constructor
  - Too easy to create security risks

Serialization mechanics

- On serialization
  - For each object, reflectively scrape data from fields (including private ones)
  - Recursively serialize any referred-to objects
  - Write back-references to previously seen objects, to preserve cycles
- On deserialization
  - create an “empty” object, bypassing constructors
  - read data from stream, write fields reflectively (including private, final ones)
  - recursively reconstitute objects referred to from fields
  - if backref encountered, substitute reference to actual object

What could possibly go wrong?

Casualty: thread safety

```
public class Container implements Serializable {
    private final Map<Integer, Integer> map = new TreeMap<>();

    public synchronized void insert(Integer key, Integer value) {
        map.put(key, value);
    }

    public synchronized Integer remove(Integer key) {
        return map.remove(key);
    }
}
    private writeObject(ObjectOutputStream s) {
        s.writeObject(map);
    }
}
```

private final field, initialized at construction

access to field only via synchronized methods

NOT THREAD SAFE! New code effectively added by serialization. Unsynch'ed access leads to errors.

Should be thread-safe, right?

Casualty: invariants

```
class Range implements Serializable {

    private final int start;
    private final int end;

    public Range(int start, int end) {
        if (start > end)
            throw new IllegalArgumentException();
        this.start = start;
        this.end   = end;
    }
}
```

Preconditions checked in constructor. This is safe, right?

WRONG! Malicious stream could have start > end, but deserialization bypasses constructor, so no check is done.

Casualty: initialization mechanics

```
class DateHolder implements Serializable {

    private final Date date;

    public DateHolder(Date d) {
        date = new Date(d.getTime());
    }
}

class SonOfDate extends Date {
    public long getTime() {
        return random.nextLong();
    }
}
```

Defensive copy. We know 'date' is an instance of java.util.Date, right?

WRONG! Malicious stream could insert instance of Date subclass. Deserialization bypasses constructor, so no defensive copy is done.

Casualty: confinement

Imagine Date were final, averting subclass attack

```
class DateHolder implements Serializable {

    private final Date date;

    public DateHolder(Date d) {
        date = new Date(d.getTime());
        if (isInvalid(date))
            throw new IllegalArgumentException();
    }
}
```

Defensive copy. This class has exclusive access to this Date instance. Date is mutable, so this is safe, right?

WRONG! Malicious stream could include 'backrefs' in order to get another reference to the same object.

Casualty: confinement

```
class DateHolder implements Serializable {

    private ~~final~~ Date date;

    public DateHolder(Date d) {
        date = new Date(d.getTime());
        if (isInvalid(date))
            throw new IllegalArgumentException();
    }

    private void readObject(ObjectInputStream s) throws ... {
        s.defaultReadObject();
        date = new Date(date.getTime());
        if (isInvalid(date))
            throw new InvalidObjectException();
    }
}
```

field no longer final

Fix this by adding code to rewrite private field with defensive copy

Effective Java, Item 88

Write readObject() methods defensively

- Item 88 covers some of the preceding issues in more detail
- It makes several good recommendations:
  - Add a readObject() method
  - Make defensive copies in constructor *and* in readObject()
  - Perform invariant checks on copies in constructor *and* in readObject()
- Checks are easy to forget and are error-prone
- But after all that, ***they aren't good enough***

Special bonus attack: finalization

- If readObject() encounters an error, it throws an exception
  - Object not returned to the caller
  - Reference is thrown away, so object is unreachable, right?
- Finalization can “resurrect” an otherwise unreachable object!
  - Mechanics are complicated, but a malicious stream can include an object that has a finalizer, and resurrect it later with invalid data
  - Note, your object doesn’t have to use finalization; it is instead an innocent victim of finalization (in combination with serialization)

*Possible to defend against finalization attacks, at the cost of adding **even more** defensive code*

Serialization scorecard

- ✓ Break thread safety with hidden, unsynchronized access
- ✓ Bypass constructors, creating instances with bad data
- ✓ Use polymorphism to inject malicious subclass
- ✓ Defeat object containment using backrefs
- ✓ Interferes with sensible use of final fields
- ✓ Resurrect invalid, discarded object using finalization

Lessons

- Serialization was *intended* to be simple
  - It provides the *appearance* of simplicity, but lots of hidden complexity
  - The code that isn't there ***can*** hurt you
  - Surprising and unexpected interactions with existing mechanisms
- You can add code to defend against these problems
  - Easy to forget, it's error-prone, and you have to be a serialization expert
  - It wasn't supposed to be this difficult!
- Well-intentioned effort to implement orthogonal persistence
  - Error was use of an *extralinguistic* mechanism to accomplish its goals
  - ***Undermines integrity of Java's object model***

Why not “just” use JSON?

- JSON is merely a wire encoding of the serialized information
  - The same information could be represented in JSON, XML, YAML...
  - None of the problems described earlier are related to the wire encoding!
- *In theory*, serializing to JSON could be safer
  - If all you do is unmarshal stuff by hand out of a DOM tree
  - ... but nobody actually does this
- Core problem: framework’s relationship to the object model
  - Particularly, how deserialization creates and populates objects
  - JSON frameworks all provide ways to reconstruct objects
    - Because it’s so darn convenient!
  - And now we're back to the same exploits we've been talking about

Why not “just” use JSON?

> “As we will conclude, the format used for the serialization is not relevant. It can be binary data, text such as XML, JSON or even custom binary formats. [...] Attackers may be able to gain code execution opportunities regardless of the format.”

Muñoz and Mirosh, *Friday the 13th JSON Attacks*, Black Hat USA 2017

https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf

Why not “just” use JSON?

- There might be good reasons to switch the wire encoding
- Switching wire encodings doesn’t address the core problem
- The framework’s relationship to objects is the core problem

Deserialization of JSON (or any other data format) to Java objects can give rise to all the same issues as occur in Java serialization.

A language designer looks at serialization

- ***DISCLAIMER: What follows is merely an exploration, not a plan!***
- The problems of serialization have a common root – it undermines the language and object model
  - External libraries that use JSON or XML make the same mistakes
- If we want serialization to be safe and reliable, *it has to be brought into the object model*
  - The class author has to be in control of serialization
  - In a way that is natural, and readers can reason about
- Secondarily, Serialization 1.0 probably aimed too high
  - Should have focused on serializing *data*, rather than *objects*
  - Many of the sins of serialization were committed due to this overreach

The root problem(s)

Magic is dangerous

- The big problem: everything about serialization is *magic*
  - Magic object deconstruction (field scraping)
  - Magic object reconstruction (bypasses constructors)
  - Magic accessibility back door
- Humans can't easily reason about extralinguistic behavior we can't see
  - So let's make these things explicit

Banishing the magic

- Where a class meets serialization should be part of the design process
- For any class, the author should be able to control
  - The serialized form (which need not be the same as the in-memory representation)
    - A data-modeling consideration, not a wire-encoding one
  - How the in-memory representation maps to the serialized form
  - How to validate the serialized form and create the instance
- This should be done through *ordinary Java code*
  - Constructors and factories already know how to validate their inputs, let's just lean on those
  - We may even be able to reuse an existing public member

Deserialization is construction

- We all know how to write defensive constructors
  - Argument validation, defensive copies, encapsulation
  - These are needed during deserialiation too
- By default, deserialization is entirely undefended
  - And, writing a readObject() method is too hard
  - Because it's not a constructor
- Rather than invent an alternate, harder way of doing the same thing
  - Let's use what we already know – constructors
- ***Ensure deserialization proceeds through standard constructors***
  - Sometimes, we can even re-use an existing constructor
  - Sometimes, we might have to write a constructor just for deserialization

Digression: pattern matching

- *Pattern matching* (destructuring) is coming to Java

```
if (x instanceof Range(int low, int high)) { … }
```

- Patterns combines an *applicability test* along with *conditional extraction*
  - Above pattern invokes a *deconstruction pattern* declared in Range
  - Deconstruction patterns are like a constructor in reverse
    - Constructor aggregates state components into an object
    - Deconstruction pattern decomposes an an object into state components
- We can use a constructor for reconstructing objects from serial form
  - And deconstruction patterns for extracting the serial form

Digression: pattern matching

A class designed for deconstruction and reconstruction

```
class Range {
    private final int low, high;

    public Range(int low, int high) {
        if (low >= high)
            throw new IllegalArgumentException(...);
        this.low = low;
        this.high = high;
    }

    public pattern Range(int low, int high) {
        low = this.low;
        high = this.high;
    }
}
```

Serialization is deconstruction

- From this simple example, we can already see shape of the solution
  - Use patterns to extract object state into a serialized form
  - Use matching constructors or factories to reconstruct from serialized form
    - State validated in the usual way!
  - Replace magic field scraping and object reconstitution with ordinary code
    - Some of which might already have been written and tested
- Still need to somehow indicate which patterns and constructors should be used for serialization and deserialization
  - Communicating design intent to frameworks is what annotations are for!

Choosing a serial form

- Serial form need not be the same as in-memory representation
  - For example, the serial form of a LinkedList is just its elements

```
class LinkedList {

    @Deserializer
    public static LinkedList deserialize(Object[] elements) {
        LinkedList list = new LinkedList();
        for (Object o : elements)
            list.add(o);
        return list;
    }

    @Serializer
    public pattern LinkedList(Object[] elements) {
        elements = this.toArray();
    }

}
```

Versioning

- As classes evolve, the serialized form may evolve with them
  - But serialized instances from past versions may still be floating around
- Programming model should capture past serial forms, so readers can reason about *all* the ways an instance can come into existence
  - Should be able to explicitly support (or not) specific past serial forms
  - Easy – have multiple deserializers, one for each supported version
- Explicit serializers and deserializers can have version metadata
  - So that it is obvious which old versions, and how, we support

Access control

- There's still one bit of magic left to banish – access control
  - In default serialization, a private class in a non-exported package is still effectively public, because we can construct it through serialization
- If we want serialization to be an ordinary, nonprivileged library, we need to either:
  - Make these public (yuck)
  - Provide an explicit way to express "this member is for serialization only", which reflection will make limited exceptions for
- Similar to what we do for "open" modules, just at a finer granularity
  - Dynamically accessible, but not statically accessible

Towards better serialization

- With the following:
  - Patterns for state extraction
  - Constructors or factories for reconstruction
  - Annotations to capture serialization intent and versioning
  - Explicit mechanism to capture dynamic accessibility
- Split the problem in two
  - *Author* is in charge of serialized form, state extraction, object reconstruction
    - Which can be defended against bad data
    - Can often reuse existing constructors and deconstruction patterns
  - *Serialization framework* is in charge of finding serialization members, and wire encoding and decoding – and that's it
- Serialization frameworks no longer need privileged magic
  - Not for extraction, not for reconstruction, not for accessibility

The bad news

- Writing explicit serializers is more work than just saying “implements Serializable”
  - “Don’t take away my magic”
  - But, can make this less painful with language improvements
- Cyclic object graphs – the “always deserialize through a constructor” makes it much harder to serialize cyclic object graphs
  - If serialization is about *data*, rather than *programs*, this is unlikely to be a serious limitation
- Every class must bring its own serialization – cannot inherit serialization from supertypes
  - Tricky for lambdas and anonymous classes

The long road ahead

- This approach doesn't address all the problems of serialization
  - But it puts it on a solid footing, where it can be reasoned about
  - Makes it possible for ordinary developers to write *correct* serializable classes
  - Provides a migration path away from the more dangerous mechanism
- More work needed to defend against presence of (often unknown) insecure serializable classes in existing libraries
  - But, there can be no security without correctness

The long road ahead

Updating the JDK

- Teach core serialization about explicit serialization members
  - Use this in preference to existing serialization approaches
- Migrate serializable JDK classes to use new approach
  - Three releases (18 months) to maintain +1/-1 compatibility
  - But not all classes have to migrate at once
- If other serialization frameworks support this approach too
  - Then a class authored for safe deconstruction and reconstruction can be used with these frameworks too
  - Frameworks free to compete on the basis of encoding and performance

Summary

- If we knew then what we know now …
  - And had the language then that we have now …
- We're at the beginning of a very long migration
- Goals include
  - Reduce security serialization attack surface
  - Make it easier to reason about correctness
  - Level the playing field for serialization frameworks by eliminating the need for privilege
- For more information
  - *Towards Better Serialization*
  - http://cr.openjdk.java.net/~briangoetz/amber/serialization.html

Q & A