🔐 💾 JAVA SERIALIZATION: PRACTICES THAT WON’T BITE YOU LATER

1/ Keep classes opt-in

2/ Shape the byte format yourself

3/ Lock down deserialization

4/ Avoid readResolve tricks when an enum will do

5/ Reach for the serialization-proxy pattern for robust, future-proof code

1/ Keep classes opt-in

- **Don’t make everything Serializable by default.** Only the few types that truly need Java’s native byte-stream format should implement it.
- **Avoid putting Serializable on base classes or widely reused abstractions.** If a superclass is serializable, *all* subclasses become serializable—even when that’s unsafe (e.g., they hold sockets, threads, caches, keys).
- **Create dedicated, stable “snapshot” types** (DTOs/value objects) for serialization instead of exposing rich domain objects.

Why this matters

- Reduces your attack surface during deserialization.
- Prevents accidental leakage of fields into the wire format.
- Keeps you free to refactor most classes without worrying about serialVersionUID and format compatibility.

A quick pattern

```
// Not serializable: rich domain object with invariants and live refs
final class OrderService {
  private final Executor executor;
  // ...
}
// Opt-in: minimal, immutable snapshot meant for the wire
final class OrderSnapshot implements java.io.Serializable {
  private static final long serialVersionUID = 1L;
  private final String id;
  private final int quantity;
  private void readObject(java.io.ObjectInputStream in) throws Exception {
    in.defaultReadObject();
    if (id == null || quantity < 0) throw new InvalidObjectException("Invalid snapshot");
  }
}
```

Rules of thumb

- Implement Serializable **sparingly** and only on types with a **stable external form**.
- Keep rich domain/services **non-serializable**; convert to a **snapshot/DTO** when you truly need serialization.
- If you inherit from a serializable parent but want to block it, you can add a readObject that throws InvalidObjectException.
- Prefer other formats (JSON/Proto) for system boundaries; use Java serialization only when you control both ends and need it.

That’s “opt-in”: serialization is an explicit decision per type, not the default.

2/ Shape the byte format yourself

don’t let default Java serialization decide what goes on the wire. Explicitly define **what** is serialized, **how**, and **how it’s read back** so you control compatibility, security, and invariants.

Here are the practical ways to do it:

Whitelist the fields (not all of them)

- Use serialPersistentFields to pick exactly which logical fields are serialized—ignoring caches, keys, or derived data.

```
private static final ObjectStreamField[] serialPersistentFields = {
  new ObjectStreamField("id", String.class),
  new ObjectStreamField("qty", int.class)
};
```

Write a custom external form

- Implement writeObject / readObject to serialize a **stable, minimal representation** (not your raw fields).
- Use PutField/GetField for versioning with safe defaults.

```
private void writeObject(ObjectOutputStream out) throws IOException {
  ObjectOutputStream.PutField fields = out.putFields();
  fields.put("id", this.id);
  fields.put("qty", this.quantity);
  // don’t serialize derived or sensitive fields
  out.writeFields();
}
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
  ObjectInputStream.GetField fields = in.readFields();
  this.id = (String) fields.get("id", null);
  this.quantity = fields.get("qty", 0);
  // Defensive checks (shape + validate)
  if (id == null || quantity < 0) throw new InvalidObjectException("Bad data");
}
```

Use writeReplace / readResolve when appropriate

- writeReplace lets you **convert your object to a simpler carrier** before it's serialized.
- readResolve lets you **replace** the just-deserialized object with a canonical instance (but prefer enums for true singletons).

```
private Object writeReplace() throws ObjectStreamException {
  return new Snapshot(this.id, this.quantity); // a tiny DTO
}
```

Prefer the Serialization Proxy pattern for invariants

- Serialize a small immutable proxy that knows how to rebuild the real object—this is the cleanest way to "shape" your format and keep invariants intact.

```
private Object writeReplace() { return new Proxy(this); }
private Object readObjectNoData() throws InvalidObjectException { throw new InvalidObjectException("no data"); }
private static class Proxy implements Serializable {
  private static final long serialVersionUID = 1L;
  final String id; final int qty;
  Proxy(MyType src) { this.id = src.id; this.qty = src.quantity; }
  private Object readResolve() throws ObjectStreamException {
    // reconstruct with validation
    return MyType.of(id, qty);
  }}
```

Version consciously

- Keep a serialVersionUID.
- When adding fields later, use GetField#get("newField", default) to remain backward compatible.
- Never depend on field order or nonessential internal details.

Why this matters

- Prevents **accidental field leakage** and sensitive data exposure.
- Gives you **forward/backward compatibility** knobs.
- Lets you **validate** during deserialization and **enforce invariants**.
- Minimizes attack surface vs. default, “whatever-is-there” serialization.

In short: by shaping the byte format, you’re declaring a deliberate, stable contract—rather than letting the JVM serialize your object’s guts by default.

3/ Lock down deserialization

treat deserialization as **hostile input** and actively constrain what the JVM is allowed to create and how your objects are rebuilt.

Here’s how to do it, practically:

PRINCIPLES

- **Never deserialize untrusted data** if you can avoid it; prefer JSON/Proto/etc.
- **Assume attackers control the byte stream.** Validate everything and whitelist what’s allowed.

HARDEN THE PIPE (WHAT MAY BE CREATED)

▪ **Use JEP-290 Object Input Filtering** (Java 9+): allowlist classes, cap sizes/depth, block risky packages.

```
ObjectInputStream ois = new ObjectInputStream(in);

ois.setObjectInputFilter(ObjectInputFilter.Config.createFilter(

  "com.acme.model.*;java.base/*;maxdepth=20;maxrefs=10_000;maxbytes=5M;reject:*"

));
```

Or system-wide:

```
-Djdk.serialFilter=com.acme.model.*;java.base/*;maxdepth=20;reject:*
```

▪ **Custom ObjectInputStream** (older JDKs / extra control):

```
class WhitelistingOIS extends ObjectInputStream {
  WhitelistingOIS(InputStream in) throws IOException { super(in); }
  @Override protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {
    String name = desc.getName();
    if (!name.startsWith("com.acme.model.") && !name.startsWith("java.")) {
      throw new InvalidClassException("Rejected: " + name);
    }
    return super.resolveClass(desc);
  }
}
```

HARDEN THE OBJECT (HOW IT’S REBUILT)

▪ **Defensive readObject**: validate invariants, ranges, nullability; rebuild **defensive copies** of mutable inputs.

```
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
  ObjectInputStream.GetField f = in.readFields();
  this.id = (String) f.get("id", null);
  this.items = List.copyOf((List<?>) f.get("items", List.of()));
  if (id == null || id.isBlank() || this.items.size() > 1000) {
    throw new InvalidObjectException("Invalid state");
  }
}
```

▪ **Mark sensitive/derived fields transient** (keys, caches, sockets, threads). Recompute them post-deserialization.

▪ **Prefer the Serialization Proxy pattern** to preserve invariants and shrink the attack surface:

- writeReplace() → return a small immutable proxy
- Proxy’s readResolve() → reconstruct via validated factory

▪ **Forbid deserialization entirely** for types that must never be created from a stream:

```
private void readObject(ObjectInputStream in) throws InvalidObjectException {
  throw new InvalidObjectException("Deserialization not allowed");
}
private void readObjectNoData() throws InvalidObjectException {
  throw new InvalidObjectException("No-data deserialization not allowed");
}
```

INSTANCE CONTROL & IMMUTABILITY

- Use **enum** for singletons/fixed sets (built-in safe deserialization).
- Make fields **final** where possible and reconstruct through validated constructors/factories.

LIMIT BLAST RADIUS

- Cap **size/depth/refs/bytes** via filters to prevent DoS payloads.
- Keep the serializable surface **small and explicit** (serialPersistentFields, custom writeObject).
- Avoid mixing rich domain objects with serialization; use **DTO/snapshots**.

CHECKLIST

- Is Java serialization really necessary here?
- Filter configured (per-stream or global)?
- readObject validates & copies defensively?
- Sensitive fields transient?
- Proxy pattern used for complex invariants?
- Enums instead of readResolve for singletons?

That's "locking it down": tightly control what can be instantiated and how state is restored, with validation at every step.

4/ Avoid readResolve tricks when an enum will do

Avoid readResolve "singletons" when an enum gives you the same instance-control with fewer foot-guns.

Why?

- **readResolve is manual and fragile.** You must write it exactly right, forever, across versions. Miss it or change signatures and serialized data can create *new* instances, breaking singletons or invariants.
- **Edge cases.** readResolve doesn't protect against all creation paths (e.g., careless cloning, custom deserialization code, mistakes in subclasses). You're relying on convention.

- **enum makes it bulletproof.** Java's spec guarantees one instance per enum constant per classloader. Serialization is handled by name; the JVM ensures you get the *same* instance back—no custom code required. Reflection can't instantiate extra enum constants.

Compare

Singleton with readResolve (easy to get wrong):

```
public final class Config implements java.io.Serializable {
  private static final long serialVersionUID = 1L;
  public static final Config INSTANCE = new Config();
  private Config() {}
  private Object readResolve() {
    // Must be present and correct or deserialization breaks singleton
    return INSTANCE;
  }
}
```

Pitfalls: must keep serialVersionUID, ensure constructor stays private, block cloning, keep readResolve correct; any slip can yield multiple instances.

Singleton with enum (the safe default):

```
public enum Config {
  INSTANCE;
  // fields, methods, whatever
}
```

Benefits: true one-instance semantics, safe serialization by default, simple and self-documenting.

Beyond singletons

- If you need a *fixed set* of canonical instances (e.g., LOW, MEDIUM, HIGH), enums are ideal: they **are** the instance-control mechanism.
- If you need *unbounded* instance control (e.g., value objects with many possible states), enums won't fit; prefer the **Serialization Proxy** pattern to preserve invariants during deserialization instead of relying on readResolve.

Rule of thumb

- **Singleton or small, closed set of instances? → use enum.**

- **Complex or unbounded instances with invariants? → use a Serialization Proxy, not readResolve.**

That's the idea: enums give you built-in, correct-by-construction instance control, while readResolve is a delicate workaround you can avoid.

5/ Reach for the serialization-proxy pattern for robust, future-proof code

What it is

The **serialization-proxy pattern** means you **don't serialize the real object at all**. Instead, you serialize a **small, immutable proxy** that captures just the logical, validated state. On deserialization, the proxy **rebuilds** the real object via a safe constructor/factory.

Why it's better

- **Stronger invariants:** Only a minimal, validated state crosses the wire; no half-built objects.
- **Defense-in-depth:** Fewer attack surfaces than readObject on the main class.
- **Versioning friendly:** You control the external form; adding fields becomes manageable.
- **Final fields stay final:** The real object is constructed normally, not mutated in-place by deserialization.

Minimal template

```
public final class Money implements java.io.Serializable {
 private static final long serialVersionUID = 1L;

 private final String currency; // ISO code
 private final long minorUnits; // cents

 private Money(String currency, long minorUnits) {
  if (currency == null || currency.length() != 3) throw new IllegalArgumentException("bad currency");
  if (minorUnits < 0) throw new IllegalArgumentException("neg amount");
  this.currency = currency.toUpperCase();
  this.minorUnits = minorUnits;
 }

 public static Money of(String currency, long minorUnits) {
  return new Money(currency, minorUnits);
 }

 // --- Serialization proxy hook: never serialize the real object ---
 private Object writeReplace() { return new Proxy(this); }

 // If someone tries to deserialize directly, forbid it.
 private void readObject(java.io.ObjectInputStream in) throws java.io.InvalidObjectException {
  throw new java.io.InvalidObjectException("Use proxy");
 }
 private void readObjectNoData() throws java.io.InvalidObjectException {
  throw new java.io.InvalidObjectException("No data");
 }

 // --- The proxy that actually gets serialized ---
 private static final class Proxy implements java.io.Serializable {
  private static final long serialVersionUID = 1L;
  private final String currency;
  private final long minorUnits;

  Proxy(Money m) {
    this.currency = m.currency;
    this.minorUnits = m.minorUnits;
  }
  // Reconstitute the real object with validation
  private Object readResolve() throws java.io.ObjectStreamException {
   return Money.of(currency, minorUnits);
  }
 }
}
```

How it works (flow)

1. writeReplace() returns new Proxy(real) → only proxy goes on the stream.
2. Proxy's fields are **tiny, immutable, validated**.
3. On input, proxy's readResolve() calls your **factory/constructor**, rebuilding a correct Money.

When to use it

- Classes with **nontrivial invariants** (e.g., ranges, cross-field rules).
- Immutable value objects that must remain **final** and **consistent**.
- Types likely to **evolve** (add/remove fields) but need backward compatibility.
- Any time readObject would be long/fragile—prefer a proxy.

When not to

- Trivial DTOs with flat, stable state (custom writeObject/readObject may suffice).
- Performance-critical hot paths where the extra proxy hop is proven costly (rare).

Versioning tips

- Keep the proxy as your **stable external form**.
- Add new proxy fields with safe defaults (GetField#get("field", default)) if you later drop custom code in; with the pure proxy pattern, you usually just add new final fields and handle them in the factory.
- Maintain serialVersionUID on the proxy class.

Quick checklist

- Does the main class implement Serializable? **Yes**, but only to expose writeReplace.

- Is direct deserialization blocked? **Yes** (readObject, readObjectNoData throw).
- Is the proxy **static**, **private**, **immutable**, and **minimal**?
- Does readResolve() call a **validating** factory/constructor?
- Are invariants enforced only in one place (the factory)? ✔

That’s the serialization-proxy pattern: smaller surface, safer invariants, cleaner evolution.

TAKEAWAYS

- Treat deserialization as input validation, not a free constructor.
- Shape and version your byte format deliberately.
- Enums > readResolve for true singletons.
- Serialization Proxy = safer, cleaner, future-ready design.