


Portable Objects: Best Practices for Seamless Java, .NET and C++ Interoperability

Aleksandar Seović
aleks@s4hc.com

What is POF?

- Platform independent serialization format
- Introduced in Coherence 3.2 to support access to Coherence from .NET clients
- Recommended serialization format for all applications from Coherence 3.4
- Some nice new features in Coherence 3.5...

Why use POF?

- POF is... well, portable
- POF serialization is fast
  - 5-10 times faster than Java or .NET binary serialization in most cases
- POF is extremely compact
  - Type IDs and property indexes instead of class and property names
  - Typically 3-5 times smaller serialized form

What POF isn't?

- Readable by humans
- Compatible with other serialization formats
- Completely self-describing
- Designed to support every possible user type implementation

In other words...

- *"POF is explicitly not intended to be able to answer all questions, nor to be all things to all people. If there is an 80/20 rule and a 90/10 rule, POF is designed for the equivalent of a 98/2 rule: it should suffice for all but the designs of an esoteric and/or convoluted mind."*

  -- Cameron Purdy

POF Basics

- Two ways to make your classes portable:
  - Implement PortableObject interface
  - Implement external PofSerializer
- User types must be registered within the PofContext on each platform
  - Type IDs have to match across platforms

Option 1: PortableObject

```
public class Person implements PortableObject {
    private long   id;
    private String name;
    private Date   dateOfBirth;

    public void readExternal(PofReader reader) throws IOException {
        id          = reader.readLong(0);
        name        = reader.readString(1);
        dateOfBirth = reader.readDate(2);
    }

    public void writeExternal(PofWriter writer) throws IOException {
        writer.writeLong(  0, id);
        writer.writeString(1, name);
        writer.writeDate(  2, dateOfBirth);
    }
}
```

Option 2: PofSerializer

```
public class PersonSerializer implements PofSerializer {

    public void serialize(PofWriter writer, Object o) throws IOException {
        Person p = (Person) o;

        writer.writeLong(  0, p.getId());
        writer.writeString(1, p.getName());
        writer.writeDate(  2, p.getDateOfBirth());

        writer.writeRemainder(null);
    }

    public Object deserialize(PofReader reader) throws IOException {
        long   id          = reader.readLong(0);
        String name        = reader.readString(1);
        Date   dateOfBirth = reader.readDate(2);

        reader.readRemainder();
        return new Person(id, name, dateOfBirth);
    }
}
```

Which approach to use?

- PortableObject is slightly simpler to implement and configure
  - But requires default constructor
- PofSerializer allows you to serialize existing classes
  - Can/should be simplified by providing abstract base class
  - Should be used when complex creational logic is required
- Consider object lifetime and evolution needs

Use PortableObject

- For short lived objects:
  - Entry processors, aggregators, etc.
- For non-evolvable objects, or evolvable objects in a flat class hierarchy
  - It doesn't handle deep hierarchies well

Use PofSerializer

- For long-lived objects
  - Your cached domain objects
- For evolvable objects
- For deep class hierarchies
  - Although you might be better off to avoid them altogether

SRP and PortableObject

- Single Responsibility Principle:
  - *"Class should have a single reason to change"*
- Addition or removal of an attribute **is** a single reason to change
  - It doesn't matter that the change needs to be reflected in multiple places
  - It is much easier to maintain serialization code when it's within or close to the class it serializes
    - Implement external PofSerializer as a static inner class


POF and Class Evolution

- You can make your classes forward and backward compatible by implementing Evolvable interface
  - Extending AbstractEvolvable or a similar class might be simpler
- ... but you might run into serialization issues

POF and Class Evolution

- POF requirement:
  - Attributes added to any class in a hierarchy must be written after all the attributes from the previous version of all classes in a hierarchy
- Makes addition of attributes to non-leaf classes problematic
  - PortableObject is not an option
  - You can use PofSerializer, but there are still issues

POF and Class Evolution

- Adding an attribute to a class effectively versions all of its subclasses as well
- Several ways to handle this:
  - Flatten your class hierarchy
  - Explicitly serialize attributes from all parent classes at each level (breaks DRY principle)
  - Implement multi-pass base serializer that invokes each serializer in the hierarchy for each version

POF Serialization Tips

- For most types, POF serialization is very straight forward
- But, there are a few tricky ones...
  - Enums
  - Collections/Maps/Arrays

Serializing Enums

- To ensure portability, serialize them as strings:
  - Java

```
writer.writeString(0, day.name());
day = (DayOfWeek) Enum.valueOf(DayOfWeek.class,
                               reader.readString(0));
```

  - .NET (C#)

```
writer.WriteString(0, day.ToString());
day = (DayOfWeek) Enum.Parse(typeof(DayOfWeek),
                             reader.ReadString(0));
```

- As an alternative, consider using external serializer...

Serializing Enums

```
public class EnumSerializer implements PofSerializer {

    public void serialize(PofWriter writer, Object o) throws IOException {
        writer.writeString(0, ((Enum) o).name());
        writer.writeRemainder(null);
    }

    public Object deserialize(PofReader reader) throws IOException {
        PofContext pofContext = reader.getPofContext();
        Class      enumType   = pofContext.getClass(reader.getUserTypeId());
        Enum       enumValue  = Enum.valueOf(enumType, reader.readString(0));
        reader.readRemainder();

        return enumValue;
    }
}
```

Serializing Enums

- Both approaches have pros and cons
  - Direct deserialization tends to be verbose
  - External serializer requires you to register each enum within POF context
- Ideally, we should probably add writeEnum and readEnum methods to PofWriter and PofReader interfaces respectively

Serializing Collections

- There are two possible collection representations in POF stream
  - Uniform
  - Non-uniform
- If the values in the collection are of the same type, use uniform collections
  - It will make serialized form more compact

Serializing Collections

- POF does not guarantee that deserialized collection, map or array will be of the same exact type as the originally serialized one
  - This is by design: different platforms have different collection types, so POF doesn't encode the type
  - You need to provide a template if you want a specific type to be returned:

```
reader.readCollection(0, new LinkedList());
reader.readMap(1, new TreeMap());
reader.readObjectArray(2, new Person[0]); // will be resized
```

POF

What's new in Coherence 3.5?

New in 3.5

- **PofExtractor**
  - Allows you to extract property without fully deserializing cached object
  - Can be used within filters, aggregators, indexes, etc.
  - Effectively allows .NET and C++ clients to use these features without matching Java classes in the cluster

New in 3.5

- **PofUpdater**
  - Allows you to modify property of a serialized object directly, completely avoiding “deserialize-update-serialize” cycle

New in 3.5

- **PofValue**
  - Lower-level feature that both PofExtractor and PofUpdater depend on
  - Allows you to access individual nodes within the POF hierarchy
    - User type properties
    - Individual array and collection elements
  - Allows you to add new properties to user types

New in 3.5

- **PofNavigator**
  - Enables POF hierarchy navigation
  - SimplePofPath
    - Static navigator that uses property indexes to navigate POF herarchy
- You can create your own navigators
  - Even dynamic ones

POF and Spring

Enabling Interoperability

Brief History of POF

- Idea: The Spring Experience, Dec 2005
- Cameron got involved: Dec 2005, Javapolis
- Cameron wrote the spec: early 2006
- Cam and Jason wrote Java implementation: June 2006
- Ported to .NET: August 2006
- Took another two years to port it to C++...


Closing the Loop

- Goal:
  - Seamless interoperability between Spring.NET and Spring
- Solution:
  - Client-side proxy factory
  - RemoteInvocationServer bean
  - POF as a wire format