The Science and Art of Backwards Compatibilty

Ian Robertson

Agenda

- Where does backwards compatibility matter, and why?
- Evolving Serializable types
- Ways a library can evolve
- Strategies for evolving a library in a backwards compatible fashion
- Best practices

About Me

Application Architect at Myriad Genetics

14 years experience of working on large-scale Java projects

Amateur carpenter

Blog: http://www.artima.com/weblogs/?blogger=ianr

Twitter: @nainostrebor

https://github.com/irobertson

MYRIAD®

Where Compatibility Matters: Serialization

- Distributed computing
  - Web Services, RMI, JMS, data storage, caching, etc.
  - Not all clients can upgrade at the same time
  - Old data can live for awhile (JMS, caches, files)
  - Backwards compatible serialization changes allows communications between different versions of writer and reader
  - Not just Java serialization – also XML, JSON, Avro, Protocol Buffers, etc.

Evolving Serialized Classes

- Step 0:

```
public class Foo implements Serializable {
  private static final long serialVersionUID = 1L;
  ...
}
```

  - If you forget this, you can use `serialver` later
- Change this only if you wish to intentionally break backwards compatibility

MYRIAD®

Ways a Serialization Form Can Evolve

- Adding a new field
- Removing an existing field
- Changing allowed values for a field
- Changing the type of a field
- Moving fields in inheritance hierarchy
- Change the values for an Enum

Adding a Field

- Reader upgrades first
  - Reader needs to accept absence of new field
- Writer upgrades first
  - Reader needs to already have been designed to ignore unknown fields
- Java Serialization handles both cases well, as do JaxB, Jackson, and many others
  - Unknown fields are ignored
  - Missing fields receive default values

Adding a Field

- Semantic Compatibility
  - Behave well when new field has default value
  - New field should not change meaning of old fields!

Version 1

```
public class Dimensions {
  // measurements in meters
  public int length;
  public int width;
  public int height;
}
```

Version 2

```
public class Dimensions {
  public boolean useMetric;
  public int length;
  public int width;
  public int height;
}
```

MYRIAD®

Removing a Field

- Similar to adding a field – switch the role of reader and writer
- Reader upgrades first – ignore the old field
- Writer upgrades first – reader gets default value
- Again, also pay attention to semantic compatibility!
- Renaming a field is an add and remove

Changing Allowed Values for a Field

- In general, this is a semantic issue
- If allowing more values, readers need to either upgrade first, or be able to handle previously unaccepted values
- If the set of allowed values is constrained, writers must upgrade first
- Default serialization bypasses all constructors!!!

Changing the Type of a Field

- Java Serialization:
  - Writer can send subtype of what reader expects (Liskov)
  - Reader cannot require a subtype of what writer sends
- More generally:
  - The reader needs to be able to process what the writer sends
- Alternative: add a new field with the new type
  - Have the old field “forward”

Forwarding Field

Version 1

```
public class Invoice implements Serializable {
  private float amount;
  public float getAmount() {
    return amount;
  }

  public void setAmount(float amount) {
    this.amount = amount;
  }
}
```

MYRIAD®

Forwarding Field

Version 2

```
public class Invoice implements Serializable {
  private float amount;
  private BigDecimal totalAmount;

  public void setAmount(float amount) {
    this.amount = amount;
    this.totalAmount = new BigDecimal(amount);
  }

  public BigDecimal getTotalAmount() {
    return totalAmount != null
      ? totalAmount : new Big Decimal(amount);
  }
}
```

MYRIAD®

Moving Fields in Class Hierarchy

- Fields in a serialzed form belong to a particular class

Version 1

```
public class Person
implements Serializable {
}

public class Employee
extends Person {
  String name;
}
```

Version 2

```
public class Person
implements Serializable {
  String name;
}

public class Employee
  extends Person {
}
```

Unrelated Fields!!!

MYRIAD®

Enums

Multiple options for encoding:

- Encode the name (Java Serialization does this)
  - Refactoring the name of a value will cause incompatiblity
- Encode the ordinal (Protocol Buffers does this)
  - Refactoring the order or names of enum values can cause a hard to diagnose incompatibility!
- Best practice: use independent names for en/decoding
  - Don’t use these for anything else!

Detecting Incompatible Changes

- For each new version, write a file consisting of serialized forms of various instances of your serialized classes
- Write unit tests that verify ability to read these classes
- Only delete tests for versions no longer supported for backwards compatibility
- Alternative – unit test loads older version of library to create serialized form on the fly

Custom Serialization and Deserialization

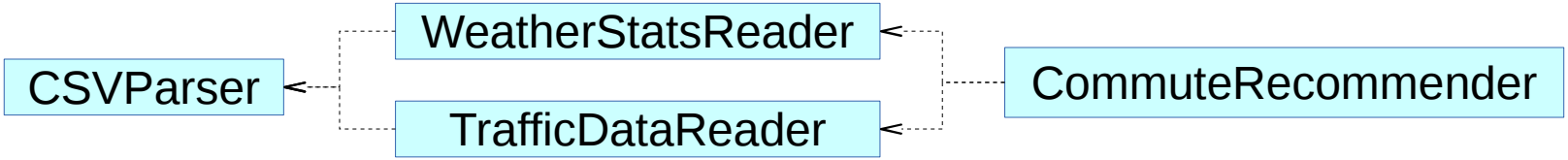
- Useful for otherwise backwards-incompatible changes

  private void readObject(ObjectInputStream ois)
- ObjectInputStream.readFields()
  - Easy to use
  - $O(n^2)$ in the number of fields!
- To allow older versions to read newer version, implement

  private void writeObject(ObjectOutputStream oos)
- Externalizable: Developer has near total control and responsibility

Where Compatibility Matters: APIs

- Library APIs
  - Why can't my clients just upgrade?



  - CSVParser releases new, backwards incompatible version
  - WeatherStatsReader adds new functionality, upgrades its dependence on CSVParser
  - TrafficDataReader has not updated to use new CSVParser
  - CommuteRecommender wants to use new version of WeatherStatsReader, but cannot!

API Backwards Compatibility

- Source compatible:
  - Existing source code compiles against new library
- Binary compatible:
  - Code compiled against the old version will successfully link against the new version
- Neither implies code will continue to successfully *run*, but it's a good start

MYRIAD®

Dangers of Source Compatibility

- Good enough for APIs only used by applications, but not for libraries used by other libraries
- Easy to make source compatible, binary incompatible changes and not notice

JVM Method Calls

- The JVM has two (traditional) ways of calling instance methods:
  - InvokeInterface – interface-defined methods
  - InvokeVirtual – class-defined methods
- Clients call a specific signature of a method, including parameter types and the return type
- No type conversions are performed in looking up methods

Example Byte Code

```
ArrayList<String> arrayList = new ArrayList<String>();
List<String> list = arrayList;
arrayList.add("x");
  // ldc #1 - String x
  // invokevirtual #2 - ArrayList.add:(LObject;)Z
list.size();
  // invokeinterface #3 - List.size()I
```

- Method signatures must match on parameter types, return type, and interface or class

MYRIAD®

Binary Compatibility: Adding Methods

- Adding a method to a class is fine
  - but consider whether subclasses may already have implemented the same method with different meaning
- Adding a method to an interface which clients do not implement is fine
- Adding a method to a client-implemented interface can lead to runtime NoSuchMethodError, but only if called!
  - Allows adding methods to the javax.sql interfaces

Default methods

- Allows adding new methods to client-implemented interfaces

```
public interface Person {
  double getHeightInFeet();

  /**
  * @since 1.1
  */
  default double getHeightInMeters() {
    return getHeightInFeet() * 0.3048;
  }
}
```

MYRIAD®

Binary Compatibility: Removing methods

- Removing a method which clients do not call is fine
- Best practice – deprecate a method long before removing it (and javadoc the preferred alternatives!)
- Note: changing a method signature has the same impact as removing it!

Binary Compatibility: Changing methods

- When parameter types change, method overloading allows you to keep old signatures around
- The Java language does not allow overloading on return type
- The Java Virtual Machine ***does*** allow overloading on return type
- In fact, it does this for you behind the scenes for subclasses

Specializing Return Types

```
public class SelfCaused extends Exception {
    @Override
    public SelfCaused getCause() { return this; }
}
```

```
public SelfCaused getCause();
   0:    aload_0
   1:    areturn
public Throwable getCause(); - marked “synthetic”
   0:    aload_0
   1:    invokevirtual #2 - getCause:()LSelfCaused;
   4:    areturn
```

Bridge Method Injector

- Written by Kohsuke Kawaguchi (creator of Jenkins)
- http://bridge-method-injector.infradna.com/
- Uses an annotation processor combined with a Maven plugin to add synthetic methods for overloading on return type

Bridge Method Injector Example

Version 1.0:

```
public Foo getFoo() { … }
public Bar getBar() { … }
```

Version 1.1:

```
@WithBridgeMethods(Foo.class)
public FooChild getFoo() { … }

@WithBridgeMethods(value = Bar.class,
                   checkCast = true)
public BarParent getBar() { … }
```

MYRIAD®

Binary Compatibility: Class Hierarchy

- Adding to the list of implemented interfaces is fine
- Changing the direct superclass can be fine:
  - Provided no client-referenced super class is removed from the hierarchy, and
  - Provided no superclass defining a non-overridden client-called method is removed
    - Adding method to the child class will help
- Similarly, removing interfaces is fine, subject to client references and calls

Binary Compatibility: Class vs Interface

- Recall that JVM has separate byte codes for invoking interface and class methods
- Just as the caller must have the right signature, it must also have the right invocation type
- Changing a class to an interface, or visa-versa, is a binary incompatible change

Detecting Incompatible Changes

- Writing compatibility tests
  - Exercise clients compiled against each older version
  - Also a useful strategy for web services, JMS, etc.
- Clirr: http://clirr.sourceforge.net/
  - Compares two versions of a jar, looking for incompatibilities
  - Plugins for ant, maven and gradle

If Compatibility Must be Broken

- Don’t!
- Really?
- Change your major version number
  - http://semver.org/
- Consider changing project and package names
  - Allows clients to use libraries depending on both versions
  - JarJarLinks *can* allow clients to do this, but it puts the burden on them

MYRIAD®

Questions?

MYRIAD®