

Working with Objects

Objectives

After completing this lesson, you should be able to:

- Implement Java objects that are stored in Coherence caches by using `Serializable`, `ExternalizableLite`, and Portable Object Format techniques
- Describe the purpose of deserialized caches
- List which cache types are not serialized

Java Objects and Coherence

A Coherence Java object:

- Is a Plain Old Java Object (POJO)
- Must support some form of serialization
- May support `Comparable`
- Should provide support for identity

```
public class AirPort implements Serializable, Comparable {  
    private int id;  
    private String code;  
    private String city;  
    private String name;  
    private String country;  
  
    public Airport() { }  
    . . .  
}
```

All must be serializable,
child classes must also
be serializable!

AirPort.java

4 - 3

Coherence objects, that is Java objects that can be inserted into a cache, have a single hard requirement: they must be serializable in some way. Because cached objects are sent across the network, they must be serializable in one form or another. The standard `java.io.Serializable` interface is technically all that is required. However, in reality several other requirements exist that make Coherence objects usable, such as:

- `java.lang.Comparable`: An interface that imposes a total order on the objects of the class it implements. In essence, `Comparable` allows you to sort objects against each other, and is a requirement for certain SQL-like queries, which include `ORDER BY` clauses.
- **Identity**: Identity is not a strict requirement; however, much like any database or persistent store, caches are storage. Storing an object that cannot be uniquely returned is of little value. Typically, objects can be identified by simple classes such as `String`, `Integer`, and so on, but may be identified by more complex objects.
- `toString`: A method that prints an object's content in a human-friendly manner. `toString()` is purely a convenience.

AirPort Object: Example

```
public class AirPort implements Serializable, Comparable {  
    private static final long serialVersionUID = . . . ;  
    private int id;  
    . . .  
    private String country;  
    public AirPort(int id, String code,  
        String city, String name, String country) {  
        this.id = id;  
        . . .  
    }  
    public AirPort() { }  
  
    public String getCode() { ... }  
    public void setCode(String code){ ... }  
    public String getCity() { ... }  
    public void setCity(String city){ ... }  
    public String getName() { ... }  
    public void setName(String name){ ... }  
    public String getCountry() { ... }  
    public void setCountry(String country) { ... }  
    . . .  
}
```

Must implement some form
of serialization

One or more constructors
Note no-arg constructor

"Standard" JavaBean
getter/setters

AirPort.java

4 - 4

The `AirPort` class shown here defines all the requirements of a baseline JavaBean that can be used with Coherence. It has the following characteristics:

1. **Serializable:** Though it is inefficient compared to other serialization mechanisms, the Java serializable interface meets the requirements for serialization.
2. **Constructors:** Has, as a minimum, a no-argument constructor. Other constructors are ignored. Note that it is a requirement for loading an instance of this class by Spring.
3. **Setters and Getters:** Has standard JavaBean getter/setter methods. These can be generated by both Eclipse and JDeveloper.

AirPort Object: Example

```
. . .  
  
public int compareTo(Object o) {  
    AirPort ap = (AirPort)o;  
    return name.compareTo(ap.getName());  
}  
  
public String toString() {  
    StrignBuilder sb = new StrignBuilder ();  
    sb.append("Airport: \n");  
    sb.append("\t").append("ID:").append(id).append("\n");  
    . . .  
    return sb.toString();  
}  
}
```

AirPort.java

4 - 5

The remainder of this example is optional, but recommended.

The class implements the `Comparable` interface, which requires a `compareTo` method and a `toString` method.

Objects and Identity

Objects become *entities* when they are assigned an *identity*.

Identity:

- Is used to uniquely identify one object from another
- Can be an attribute of an object, known as a *natural* identity
- Can be assigned by the system, known as a *surrogate* identity
- Must not change over the lifetime of the object

```
public class AirPort implements . . . {  
    private int id;  
    public AirPort(int id, ... ) { this.id = id; }  
    public int getId()      { ... }  
}
```

AirPort.java

4 - 6

Objects in Coherence are persistent entities. What this means is that there must be a way to identify an object so it can be returned when needed. Typically, an identity is some sort of generated value, which is uniquely assigned to each entity at creation time. In databases, you can typically `INSERT` a new row into a table, conceptually similar to adding an object to a Coherence cache, and after the insert, some sort of ID generator is used to assign the new row a unique ID. However, in Coherence the identity must be assigned up front, before the object is inserted into the cache. It is the responsibility of the developer rather than the database to assign a unique identity to each element.

There are two types of identity: natural and surrogate. A natural identity is one that is part of the object, a combination of one or more attributes that would be part of the object no matter how it was stored. A surrogate ID is one that exists for no other reason than to make the entity unique.

The final requirement of identity is that it must not change over time. Because Coherence caches use map semantics, the values must be 1:1 mapped between key and object. Any new object, inserted with the same key, replaces the old object.

Identity Types

Identity:

- May be based on a single primitive type, or primitive wrapper classes
- May be based on a class composed of the preceding items
- Must be serializable
- Must implement `equals()` and `hashCode()`

```
public class LoggingKey implements Serializable {  
    private Long severity;  
    private Timestamp time;  
    public LoggingKey (long severity) {  
        this.severity = severity;  
        this.time = new Timestamp(System.currentTimeMillis());  
    }  
    ... // hashCode and equal omitted  
}
```

LoggingKey.java

4 - 7

Most developers who are familiar with databases are familiar with the concept of a unique identifier that uses primitive types such as `integer` or `long`. However, Coherence works with composite keys built from some number of serializable data elements.

When implementing `equals()` and `hashCode()`, always ensure these methods use all fields for testing, not a subset. Keys are used extensively in Coherence for partitioning of data, and poorly coded `hashCode` and `equals` methods can cause unexpected behaviors.

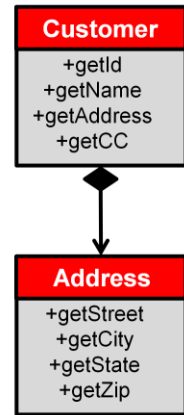
Aggregate Objects

Aggregate entities are:

- Composite entities
- Composed of a root entity and one or more dependent or weak entities
- Typically stored in the same cache

Dependent entities typically have no meaning without their owner. Examples include:

- Order lines, which have no meaning without their order
- Accounts, which have no meaning without their owner
- Phone call details, which have no meaning without their associated phone line



Address has no meaning without the associated customer!

4 - 8

Aggregate objects are those objects that contain one root or parent object and one or more associated or dependent objects. A common example is banking accounts, which have a set of associated transactions. The transactions have no meaning without their associated account record. Similarly, an order and its associated lines might be considered dependent objects.

In Coherence, the first reaction is to separate these two objects into separate caches, using the same logic used in databases where they would be in separate tables, with a foreign key association. However, such an approach may cause consistency concerns with updating both objects in the scope of a transaction. A second concern, and a reason to separate the two entities into different caches, might be sizing or differing caching retention policies. It may make sense to cache the last 90 days of banking transactions only, but not all bank accounts.

Properties of Relationships

Relationships include several core properties.

- **Cardinality:** What is the relationship between two objects? Typically expressed as 1:1, 1:Many, Many:1, Many:Many
- **Dependency:** Does one object depend on or own another? Typically thought of as:
 - *Association:* Entities in a relationship exist independently of the relationship.
 - *Composition:* Entities on one side of the relationship exist only if the relationship exists.
- **Direction:** Which elements reference which in a relationship? Typically thought of as:
 - Unidirectional or one way
 - Bidirectional or two way

4 - 9

In order to understand how two entities relate to each other, it is important to understand what the core properties of relationships are.

- First is cardinality. Cardinality defines the “how many” relationship between two entities. For example, a phone number applies to one phone, but a person may have many phones. So the relationship of phone to phone number is 1:1, but the relationship of person to phone is 1:M.
- Next is dependency. Can one object exist without the other? Certainly phones may exist without an owner, in which case the relationship is an association. However, in many cases, the relationship cannot exist without one or the other element. For example, you cannot have children without parents.
- Finally, there is direction, which defines how objects are referenced. A parent-child relationship is bidirectional. Parents have children; children have parents. Similarly, people have addresses, but addresses do not have people. Such a relationship is unidirectional or one way.

Modeling Relationships in Java

Relationships in RDBMS:

- Are implicitly bidirectional
- Can use keys to find object owners
- Are modeled using foreign keys

PERSONS
PID
ADDRID
...

ADDRESSES
ADDRID
INTEGER
...

Relationships in Java:

- Are implicitly unidirectional
- Can be made bidirectional using two unidirectional relationships

Objects do not know who owns them!

```
public class Address {  
    int addrId;  
    . . .  
}  
  
public class Person {  
    Address address;  
    . . .  
}
```

4 - 10

Bidirectional relationships are modeled in Java by two unidirectional relationships.

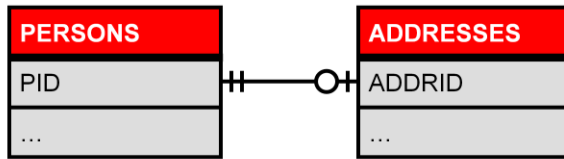
Two references in memory model the single value in the relational database.

- On read, JPA updates both values in memory based on one value in the database.
- On write, JPA uses the “owning side” of the relationship in memory to update the database.

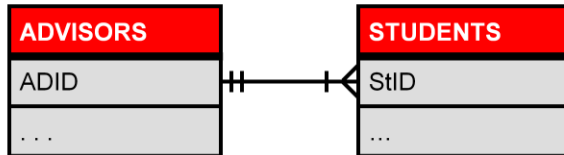
The application should manage the bidirectional relationship in Java.

The application’s burden to manage bidirectional relationships entails design choices. A bidirectional one-to-many relationship is typically modeled as a collection with a back reference. For example, the invoice has a collection of line items, and each line item has a reference to its invoice. It would be a dubious design choice to allow code outside the model to add directly to the collection of line items, because this code may not observe the constraint to update the line item’s back reference to the invoice. Failure to maintain the back reference means that the relationship is lost during persistence, because in the case of a bidirectional one-to-many relationship, the back reference is always the owning side of the relationship. Note, however, that Kodo does provide management of relationships as a vendor feature.

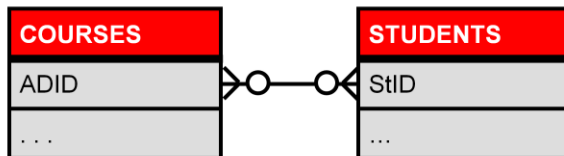
Cardinality: Examples



One-to-One
or 1:1



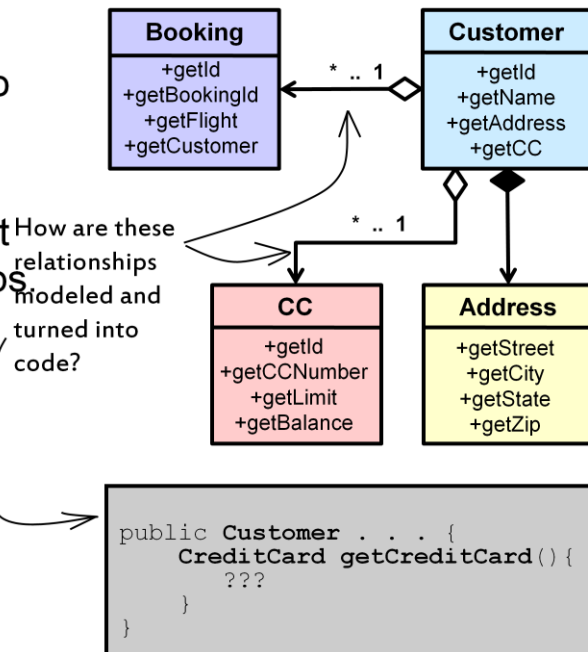
One-to-Many
or 1:M



Many-to-Many
or M:M

Modeling Relationships in Coherence

- Relationship modeling requires a mechanism to return related objects.
- Coherence does not implicitly provide support for modeling relationships



4 - 12

Turning a well-defined data model into a Coherence object model requires some thought. Coherence does not require nor enforce any specific mechanism. However, several patterns exist that allow for modeling such relationships.

Agenda



Developing Objects

Improving Performance by Using Serialization

- Serialization and Performance
- Serialization Options
- Deserialized Caches
- Serialization Behaviors

Serialization Concepts

Serialization:

- Is the process of transforming an object into and out of a format appropriate for storage or transfer across a network
- Provides:
 - A convenient method for writing objects to storage
 - A method for detecting changes in objects versus implementations  Can detect mismatches such as old data deserialized to a newer object definition
 - Support for overriding the process via implementation of the `Externalizable` interface
- Operations are referred to as `serialize` and `deserialize`
- Is supported in Java using `java.io.Serializable` interface  Marks the object "ok" to serialize nothing more

4 - 14

Serialization is the process of converting an object to a storable format. Many languages provide mechanism for marking an interface serializable. For example, .NET uses the `Serializable` attribute, whereas Java uses the `java.io.Serializable` marker interface. Marking a class serializable doesn't actually do anything, but rather announces to the application and language that the class can be written to disk. Such a marking means that only serializable fields are used in the class. Java supports the ability to mark fields transient; such fields are not serialized.

The standard Java encoding method uses a simple translation of the fields into a byte stream. Primitives as well as non-transient, non-static referenced objects are encoded into the stream. Each object that is referenced by the serialized object and not marked as transient must also be serialized; and if any object in the complete graph of non-transient object references is not serializable, then serialization will fail. Developers can influence this behavior by marking objects as transient, or by redefining the serialization for an object via `Externalizable`.

The process of serialization is sometimes referred to as *marshalling* and *un-marshalling*.

Serialization and Performance

Serialization mechanisms impact:

Area	Impact
CPU	Some serialization forms use significantly more CPU than others to serialize the same object.
Network and memory	The more compact a serialized object is, the less network bandwidth is required to transmit it and the less memory required to store it, speeding garbage collection.
Cache capacity	The more compact the serialization format is, the more objects that can be managed by the grid for a given amount of memory.

4 - 15

Every object is transmitted and stored in memory in its serialized form. The overall performance of your cache is impacted by the way your objects are serialized. The following properties of the serialization mechanism that you choose affect your cache throughput and latency:

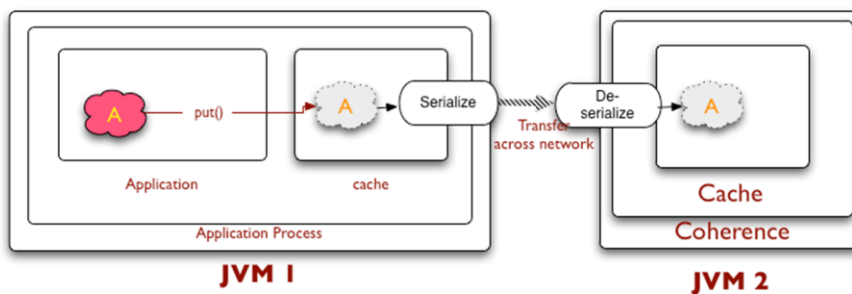
- **CPU utilization:** Some serialization forms take significantly more CPU than others to serialize the same object.
- **Network utilization:** The more compact a serialized object is, the less network bandwidth is required to transmit it.
- **Garbage collection:** A more memory-efficient serialization mechanism can allow for less garbage collection.
- **Cache capacity:** The more compact the serialization format is, the more objects can be managed by the grid for a given amount of memory.

Serialization Options with Coherence

Coherence supports a number serialization options, including:

- **Java serialization:** `java.io.Serializable`
- **ExternalizableLite:** A variation on Java Externalizable
- **Portable Object Format (POF):** Fast, compact, platform-independent serialization

Others as well, such as using XML to define the serialized format (not serialized into XML)!



4 - 16

Coherence provides the following options for serialization:

- **Standard Java Serialization:** The Java Serialization API provides a standard mechanism to handle object serialization. An object is marked serializable by implementing the `java.io.Serializable` interface. When you use the `Serializable` interface, your class is serialized automatically by default.
- **Externalizable interface:** Object serialization also uses the `Externalizable` interface. Java gives you control over the serialization process through the `java.io.Externalizable` interface.
- **ExternalizableLite:** Coherence provides `ExternalizableLite`, which is a highly optimized `Serializable` interface, along with `ExternalizableHelper` to serialize and deserialize attributes. `com.tangosol.io.ExternalizableLite` is very similar to `java.io.Externalizable`, but offers better performance and less memory usage by using a more efficient I/O stream implementation. The `com.tangosol.run.xml.XmlBean` class provides a default implementation of this interface.
- **Portable Object Format (POF):** POF is an extremely efficient binary format. It has many advantages over the standard Java serialization.

For backward-compatibility reasons, `ExternalizableLite` remains available, but where possible, POF should be used because it has many benefits over `ExternalizableLite`.

Serialization Comparison

Type	Pros	Cons
Serializable	Built-in	Slow; large files; not cross platform
ExternalizableLite	Performs better than Java serialization, by a factor of 6:1, sizes are on the order of 3:1 smaller	Must code <code>readExternal()</code> and <code>writeExternal()</code> methods on entities; not cross platform
XmlBean	Similar in size to, but performs slightly better than <code>ExternalizableLite</code>	Requires minor configuration; classes and subclasses must implement <code>XmlBean</code> ; not cross platform
Portable Object Format (POF)	Cross platform; efficient and compact; highest performing of all serialization mechanisms, often 10:1 over Java serializable	Requires configuration and assignment of identifiers to each object, also supports runtime registration; must code <code>readExternal()</code> and <code>writeExternal()</code> methods on entities

Implementing Java Serialization

To implement Java Serialization:

- Member variables must be serializable
- Class members must be serializable
- `java.io.Serializable` must be implemented

```
import java.io.Serializable;

public class AirCraft implements . . . Serializable {
    . . .
    private Long id;
    private Long capacity;
    private String type;
    private String description;
    . . .
}
```

Annotations:

- Arrow from `Serializable` to `implements`: Add implements Serializable to class.
- Arrow from `private` keywords to member variables: Members must be serializable. Class members must implement serializable.
- Arrow from closing brace `}` to text: No specialized code required

File name: `AirCraft.java`

4 - 18

Classes that implement Java serialization are perhaps the simplest of all serializable classes. Classes that implement Java serialization must satisfy the following requirements:

- They must define that they are serializable with the `java.io.Serializable` marker interface.
- Member variables, including class variables, must be serializable.
- Variables that are not serializable, such as `HttpSession`, `Sockets`, and so on, must be marked `transient`.

Lastly, no special coding is required. Coherence can determine type information via reflection and write out the class contents appropriately.

Implementing ExternalizableLite

Classes implementing ExternalizableLite:

- Must provide a no-argument constructor and setters for all fields
- Must implement `com.tangosol.io.ExternalizableLite`
- Must implement `readExternal()` and `writeExternal()` methods
- Can serialize only primitive, primitive wrappers, and classes that themselves implement ExternalizableLite
- Can use ExternalizableHelper to assist with serialization of certain types, arrays, child classes, and others

Can expect a 6 to 8x improvement in performance and a 3:1 reduction in size over Java serialization

4 - 19

Classes that implement ExternalizableLite have several requirements beyond standard Java serialization and are responsible for defining how they serialize themselves.

Classes that implement Coherence ExternalizableLite:

- Must define that they are serializable with the `com.tangosol.io.ExternalizableLite` interface
- Must implement `readExternal()` and `writeExternal()` methods, which are called to externalize and re-create class instances
- May use the ExternalizableHelper class to assist in serializing content

ExternalizableLite: Example

```
import java.io.DataInput;
import java.io.DataOutput;

import com.tangosol.io.ExternalizableLite;
import com.tangosol.util.ExternalizableHelper;

public class Aircraft implements . . . ExternalizableLite {
    . . .

    public void readExternal(DataInput in) throws IOException {
        id = new Long(in.readLong());
        capacity = new Long(in.readLong());
        type = ExternalizableHelper.readSafeUTF(in);
        description = ExternalizableHelper.readSafeUTF(in);
    }

    public void writeExternal(DataOutput out) throws IOException {
        out.writeLong(id.longValue());
        out.writeLong(capacity.longValue());
        ExternalizableHelper.writeSafeUTF(out, type);
        ExternalizableHelper.writeSafeUTF(out, description);
    }
}

Data is read and written in the same order
```

Annotations in the original image:

- Required by read/writeExternal (points to DataInput and DataOutput imports)
- Implements... (points to ExternalizableLite interface)
- Repopulates object (points to readExternal method)
- Stores object (points to writeExternal method)

Aircraft.java

4 - 20

Classes that implement `ExternalizableLite` can expect serialization improvements along the lines of 6 or 8:1 in terms of speed and approximately a 3:1 reduction in size over standard Java serialization.

The two read and write routines required by the interface take `DataInput` and `DataOutput` objects for reading and writing content respectively. These are the same two classes used by the Java `Externalizable` object and behave, for the most part, the same way. However, Coherence provides a second helper class for handling certain object types. In the example given, strings are read and written by using the `readSafeUTF()` and `writeSafeUTF()` methods. Other methods also exist for reading and writing child objects, which themselves implement `ExternalizableLite`.

Portable Object Concepts

Portable Object Format (POF):

- Is a language-independent binary format created for Coherence
- Has APIs for Java, .NET, and C++
- Serializes data faster and produces smaller results than other methods
- Supports data versioning for forward and backward compatibility
- Requires external serialization

4 - 21

The Portable Object Format (POF) is a language-independent binary format with accompanying APIs. It supports Java, C++, and .NET. POF has many advantages over the standard Java serialization.

POF is far more efficient than the standard Java serialization. On an average, POF is 10 to 12 times faster than the standard serialization and produces a much more compact result. A standard single Java Integer requires 76 bytes in serialized form, whereas with POF it takes about 4 bytes. Note that the number of bytes required varies depending on the type of object that is serialized.

POF data types can be versioned and are backward compatible. If you add a data element to an object, you can increment the version number of `EvolvablePortableObject` and it still works with the older versions. By using POF, the serialization logic can be externalized, giving you the added flexibility of not having to directly modify classes that support POF. The POF binary format is self-descriptive and indexed, which enables the extraction of values without having to fully deserialize the object. Though this functionality is not available in the product currently, POF is the foundation for future improvements. Similar to `Externalizable`, POF also requires that you implement the serialization methods manually.

PortableObject Requirements

Classes implementing `PortableObject`:

- **Must Implement**
`com.tangosol.io.pof.PortableObject`
- **Must Implement** `readExternal()` and `writeExternal()`
- **Can serialize** primitive, primitive wrappers, arrays, and child objects
- **Must be registered** declaratively or programmatically

Can expect a 10 to 12 improvement in performance and a 6:1 reduction in size over Java serialization

4 - 22

Classes that implement `PortableObject` have several requirements beyond standard Java serialization and are responsible for defining how they serialize themselves.

Classes which implement Coherence `PortableObject`:

- **Must define** that they are serializable with the `com.tangosol.io.ExternalizableLite` interface.
- **Must implement** `readExternal` and `writeExternal` methods, which are called to externalize and re-create class instances.
- **May use** the `ExternalizableHelper` class to assist in serializing content

PortableObject: Example

```
import com.tangosol.io.pof.PofReader;
import com.tangosol.io.pof.PofWriter;
import com.tangosol.io.pof.PortableObject;

public class AirCraft implements . . . PortableObject {
    . . .

    public void readExternal(PofReader in) throws IOException {
        id = in.readLong(0);
        capacity = in.readLong(1);
        type = in.readString(2);
        description = in.readString(3);
    }

    public void writeExternal(PofWriter out) throws IOException {
        out.writeLong(0, id);
        out.writeLong(1, capacity);
        out.writeString(2, type);
        out.writeString(3, description);
    }
}
```

Required imports

Implements...

Repopulates object

Stores object

Each data element is assigned a numeric identifier, and read and written by its identifier.

AirCraft.java

4 - 23

With Portable Object Format, developers are required to develop serialization code themselves. Each class must implement the portable object interface and its two required methods `readExternal` and `writeExternal`. These methods repopulate and externalize any object, and when coupled with registration, allow Coherence to efficiently read and write. Portable Object provides two classes, `PofReader`, and `PofWriter` to support reading and writing object contents. These two classes provide a set of support methods for reading and writing primitive types, arrays of types, collections of types, and so on.

POF Indices: Requirements

POF attribute indices should be assigned using the following guidelines:

- Order reads and writes from lowest index to highest.
- Read and write in the same order.
- Indices can be noncontiguous, but must be written in order.
- Indices cannot change over the life of the object class.
- Indices between a class and its encapsulated classes are independent and may overlap. For example, a `Customer` class may use values 1 to 4, and an included class such as `Address` may use the same values.

4 - 24

In addition to the points listed in the slide, inheritance should be avoided in POF-based classes because it requires the derived objects to be aware of the indices used by the parent objects, breaking the basic inheritance model.

Registering Portable Objects

Classes implementing `PortableObject` must be registered.

Registration can be:

- Programatic, using `PofContext()` instance
- Declarative, using `pof-config.xml`

```
SimplePofContext ctx = new SimplePofContext();
ctx.registerUserType(1000, AirCraft.class,
                    new PortableObjectSerializer(1000));
```

app.java

```
<pof-config>
  <user-type-list>
    <include>includes</include>
    ...
    <user-type>
      <type-id>1000</type-id>
      <class-name>com.oracle...AirCraft</class-name>
    </user-type>
    ...
</pof-config>
```

pof-config.xml

4 - 25

Objects that implement `PortableObject` must be registered in order for Coherence to take advantage of the custom serialization code. Registration can happen in one of two ways: programmatically or declaratively.

To register a class programmatically:

1. Create a new instance of `SimplePofContext()`;
2. Call the `registerUserType()` method with an identifier, the class the implements `Portable Object`, and a serializer.

To register a class declaratively:

Use user-type entries in the `pof-config.xml` configuration file.

In both cases, the serializer must be assigned an identifier. Identifiers are any integer 1000 or more. In the example provided, the same identifier is used in both cases: 1000.

Choice of identifier is somewhat arbitrary, but need only be 1000 or more.

Selected PofWriter Methods

```
package com.tangosol.io.pof;
```

```
public interface PofWriter {
```

```
    public PofContext getPofContext();
```

The `PofContext` interface represents a set of user types that can be serialized to and deserialized from a POF stream.

```
    public void writeBoolean(int iProp, boolean value);
```

Writes a boolean from the input stream

```
    public void writeChar(int iProp, char value);
```

Writes a character from the input stream

```
    public void writeShort(int iProp, short value);
```

Writes a short integer from the input stream

```
    public void writeLong(int iProp, long value);
```

Writes a integer from the input stream

```
    . . .
```

All throw `IOException` on error.

```
}
```

*An informative subset of
PofWriter methods.
See docs for complete list!*

4 - 26

The methods of the `PofWriter` interface provide the basis for reading data from an input stream to repopulate an objects. In addition to a variety of `public void write<Type>(int iProperty, <Type> value)` methods, the interface also provides `getPofContext()`, which provides information about the underlying class being processed.

Selected PofReader Methods

```
package com.tangosol.io.pof;
```

```
public interface PofReader {
```

```
    public PofContext getPofContext();
```

The `PofContext` interface represents a set of user types that can be serialized to and deserialized from a POF stream.

```
    public boolean readBoolean(int iProp);
```

Reads a boolean from the input stream

```
    public char readChar(int iProp);
```

Reads a character from the input stream

```
    public short readShort(int iProp);
```

Reads a short integer from the input stream

```
    public long readLong(int iProp);
```

Reads a integer from the input stream

```
    . . .
```

```
}
```

An informative subset of `PofReader` methods.
See docs for complete list!

4 - 27

`PofReader` methods are similar to `PofWriter`, but perform the inverse operation, reading data *into* variables rather than storing data *from* variables.

External Serialization

Serialization code can be externalized via `PofSerializer`.

Serialization code might be externalized because:

- The same code is repeated over and over
- Serialization code is complex and obscures entity functionally
- Other classes need to be serialized, which are outside of the control of the current development
- Other reasons

```
public class AirPortSerializer implements PofSerializer {  
    public Object deserialize(PofReader reader) throws IOException { . . . }  
    public void serialize(PofWriter writer, Object o) throws IOException{ . . . }  
}
```

4 - 28

Coherence supports a variety of mechanisms for serialization, including the ability to separate, or externalize, the serialization of a class or set of classes.

There are several reasons why a developer might consider externalizing, such as the following:

- Serialization code is complex or repeating and cannot be developed in a common abstract or parent class.
- Serialization code must be applied to classes outside the control of the developer.
- Serialization code obscures the underlying functionality of a class to such a degree that the class becomes overly complex.

In such circumstances, serialization can be externalized via a class that implements `PofSerializer`.

PofSerializer Basics

```
import java.io.IOException;
import com.tangosol.io.pof.PofContext;
import com.tangosol.io.pof.PofReader;
import com.tangosol.io.pof.PofSerializer;
import com.tangosol.io.pof.PofWriter;

public class CustomSerializer implements PofSerializer {

    public Object deserialize(PofReader reader)
        throws IOException {
        . . .
    }

    public void serialize(PofWriter writer, Object o)
        throws IOException {
        . . .
    }
}
```

Required imports

Implements...

Creates object from stream

Stores object to stream

Typically checks input type to insure it's correct via instanceof

CustomSerializer.java

4 - 29

Implementing the PofSerializer Interface

PofSerializer provides a way to externalize the serialization logic of a class. This is useful for classes that you do not have access to change.

To implement PofSerializer, you must provide the `deserialize()` and `serialize()` methods in the class definition.

The `deserialize()` method takes a `PofReader` as a parameter. It reads the data from the underlying POF input stream to load an object's state.

The `serialize()` method takes a `PofWriter` stream and an object as parameters. It saves the content of the object by storing its state into the underlying POF stream through `PofWriter`.

Note that the object elements are indexed in POF, and it is important that each element is read from the same index that it was written to.

Registering POF Serializers

Class implementing `PofSerializer` must be registered.

Registration can be:

- Programatic, using `PofContext()` instance
- Declarative, using `pof-config.xml`

```
SimplePofContext ctx = new SimplePofContext();
ctx.registerUserType(1000, Aircraft.class , new AircraftSerializer(1000));
```

```
<pof-config>
...
<user-type>
  <type-id>1000</type-id>
  <class-name>com.oracle...Aircraft</class-name>
  <serializer>
    <class-name>com.oracle...AircraftSerializer</class-name>
  </serializer>
</user-type>
...
</pof-config>
```

`pof-config.xml`

4 - 30

POF serializers are registered in exactly the same way as `PortableObject` serializers are, either programmatically via a `PofContext` or declaratively via `pof-config.xml`.

The primary differences are:

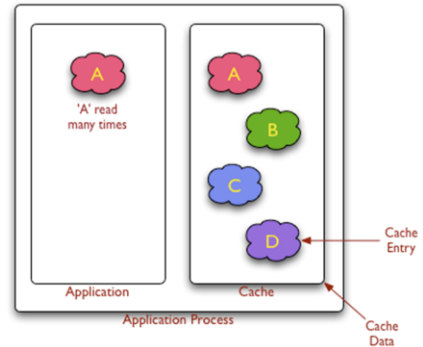
- **Programmatically:** This is the registration of the class that implements the `PofSerializer` interface.
- **Declaratively:** A `serializer` element is added to the `user-type` element, which specifies the name of the associated serializer class.

Serialized Versus Unserialized Caches

Not all caches serialize data.

Caches that perform no serialization include:

- **Near cache:** A form of cache that attempts to provide the best of replicated cache performance and partitioned cache scalability
- **Local cache:** A cache that is completely contained within, or local to, a particular cluster node



Serialization Testing Support

Serialization can be exercised using:

- `ExternalizableHelper`:
Can be used with or without a `SimplePofContext` to convert to and from `Binary` representations of data
- `Binary`:
A serialized representation of an object, providing convenience routines such as `length`, `toString()`, and others
- `Base`:
A base class containing a variety of support functions, including `toHex()` returning a string representation of an object

All are in the `com.tangosol.util` package.

Serialization Testing: Example

Serialization can be tested using `ExternalizableHelper`.

```
import com.tangosol.util.ExternalizableHelper; binary
import com.tangosol.util.Binary;
import com.tangosol.util.Base;
...
AirCraft ac = new AirCraft (. . . );
Binary binary = ExternalizableHelper.toBinary(ac);

int length = binary.length();
Byte[] bytes = binary.getByteArray();

String hex = Base.toHexString(bytes,16);

AirCraft copy (AirCraft) =
    ExternalizableHelper.fromBinary(binary);
```

Can use helper to convert to/from

Binary has convenience routines for examining data.

Base has convenience routines for conversion, etc.

4 - 33

Serialization can be tested by using both caching and retrieving objects. However, `ExternalizableHelper` can be used to simplify the process by using its `toBinary()` and `fromBinary()` methods. In the example shown, an `AirCraft` object is constructed and then the method is used to force serialization, and return a binary version of the object. Likewise, `fromBinary` can re-create an instance from its binary representation.

The `Binary` and `Base` classes provide several additional convenience methods for converting, examining, pretty-printing, and otherwise examining the serialized data.

Note that a second version of `toBinary` exists, which accepts a context to use with serialization, and can be used as follows:

```
SimplePofContext ctx = new SimplePofContext();
ctx.registerUserType(1000, AirPort.class, new AirPortSerializer());
AirPort original = new AirPort(. . .);
Binary binary = ExternalizableHelper.toBinary(original, ctx);
```

Summary

In this lesson, you should have learned how to:

- Implement Java objects that are stored in Coherence caches by using `Serializable`, `ExternalizableLite`, and Portable Object Format techniques
- Describe the purpose of deserialized caches
- List which cache types are not serialized