



## External Data Representation

How middleware encodes *data* for transmission

# + Course update

- Characterization of DS (Ch. 1)
- Distributed System Models (Ch. 2)
- Interprocess Communication (Ch. 4)
  - Sockets
  - Marshalling
- Remote Invocation (Ch. 5)
  - Case study: Java RMI





## + External Data Representation and Marshalling

Information in programs consists of data structures

Information in messages consists of sequences of bytes

Data structures must be **FLATTENED** before transmission

# + Interprocess communication

## Java object serialization

- In Java RMI, objects and primitive values may be passed as arguments and return values from remote method invocations
  - One object's class passed as an argument to a remote method must implement **java.io.Serializable**
  - This interface has no methods (It's a "marker interface")
  - Allows instances of a class to be serialized
  - ObjectOutputStream

# + Interprocess communication

## Java object serialization



**Serialization:** Flattening an object into a serial form suitable for transmission or disk storage

**Deserialization:** Reinstating an object from its serialized byte stream

- In Java, the deserializing process has NO previous knowledge of the types of the objects included in the serialized form
- The serialized form itself contains information about the serialized objects

### defaultWriteObject

```
public void defaultWriteObject()  
    throws IOException
```

Write the non-static and non-transient fields of the current class to this stream. This may only be called from the writeObject method of the class being serialized. It will throw the NotActiveException if it is called otherwise.

**Throws:**

`IOException` - if I/O errors occur while writing to the underlying `OutputStream`

### annotateProxyClass

```
protected void annotateProxyClass(Class<?> c1)  
    throws IOException
```

Subclasses may implement this method to store custom data in the stream along with descriptors for dynamic proxy classes.

### writeInt

```
public void writeInt(int val)  
    throws IOException
```

Writes a 32 bit int.



# Interprocess communication

Java objects contain references to other objects



A Java object contains primitive values AND object references

- When the object is serialized all **values** are serialized
- When the object is serialized all its **object references** are serialized
  - **References are serialized as Handles**
    - Handles are references to other objects within the serialized format
- Serialization is a **recursive** procedure
- Each class is assigned a **Handle** and is **written only once** to the stream

# + Interprocess communication

## Java objects primitive-type instance values

- int, char, boolean, etc
- These are written to the output stream in a portable data format by using methods of ObjectOutputStream:
  - writeInt(), writeChar(), writeBoolean
  - UTF-8 (UNICODE Transformation Format)
    - Unicode 1-byte representation for ASCII
- **Serialization** is normally performed by **middleware**
- Occasionally the app programmer may have to write serialization
  - Consult the Java Tutorial (Serialization) for further details



# Figure 4.9

## Indication of Java serialized form

```
• Person p = new Person("Smith", "London", 1984);
```

<i>Serialized values</i>				<i>Explanation</i>
Person	8-byte version number		h0	<i>class name, version number</i>
3	int year	java.lang.String name:	java.lang.String place:	<i>number, type and name of instance variables</i>
1984	5 Smith	6 London	h1	<i>values of instance variables</i>

The true serialized form contains additional type markers; h0 and h1 are handles (references to other objects)



# + Interprocess communication

## Java Reflection in Serialization



- Reflection: Ability of a class of reporting its properties
  - Which methods it has
  - Which fields it has
  - Which constructors
- Reflection allows us to carry out **Serialization in a completely generic manner**
  - With no previous knowledge about the properties of any object
  - No need to have its source code
- **Serialization** uses Java Reflection to find out the name of an object's class to be serialized, its types and its values

# + Interprocess communication

## Java Reflection in DeSerialization

1. The class name in the serialized form is used to CREATE a NEW CLASS
2. Create a new constructor with arguments from serialized form
3. The **constructor** is executed to create the new object and its instance variables from the serialized form

# + Interprocess communication

## Extensible Markup Language (XML)



- Defined by W3C
- **A markup language**
  - Textual encoding for text itself and its structure
  - Structured WEB documents
- **HTML** -> Appearance of web pages
  - XHTML is html compatible with XML
- **XML** -> Structured documents for the web
- Data in XML
  - Markup strings, tags
  - Define the logical structure of a document

# + Interprocess communication

## XML instance file example

```
<person id="12345678">  
  <name> Pedro </name>  
  <familyname> Pérez </familyname>  
  <year> 1984</year>  
  <!-- a comment -->  
</person>
```

# + Interprocess communication

## Uses of XML

- Clients consume Web Services by exchanging XML data with the WS point of access
  - **Marshalling** with XML
- Also, Web Services interfaces are specified in XML
- Other uses include data archiving and retrieval

# + Interprocess communication

## Extensible



- HTML tags are fixed
- Users can make their own tags in XML
  - Tags need be published so interacting programs can communicate
- **CORBA** CDR is not self-describing
  - Because both interacting **entities must have prior knowledge** of the information being exchanged
- To resolve conflicts with naming of tags and provide meaning:
  - Namespaces

# + Interprocess communication

## XML elements and attributes

- Element
- Attribute

```
<person id="12345678">
```

```
  <name> Pedro </name>
```

```
  <familyname> Pérez </familyname>
```

```
  <year>1984</year>
```

```
  <!-- a comment -->
```

```
</person>
```

# + Interprocess communication

## Well formed documents and parsing

- Rules about its **structure**.  
*A document is well formed if:*
  - Every start tag has its closing tag
  - Tags are correctly nested
- An XML document must contain a **Single root** element
- XML prolog
  - `<?XML version="1.0" encoding="UTF-8" standalone="yes"?>`
- XML namespaces
  - Scoping of names: set of names for a collection of element types and attributes
  - `xmlns:pers = http://www.cdk5.net/person`



# + Interprocess communication

## XML Schema and DTDs

- Defines the **elements** and **attributes legal** in a document
- Its **nesting**
- Its **order**
- The **number of elements**, whether empties, text are accepted
- The type of each element and its default
- The syntax for the **XML Schema** spec of an XML file is specified in XML!
- **DTD**: Document Type Definition, a more basic syntax than Schema and not completely XML

# + Interprocess communication

## XML and Java marshalling

- Java objects can be marshalled/unmarshalled in XML
- This is the technology used in **Web Services**

### Marshalling

Marshalling is the opposite of unmarshalling, where a bean is marshalled into some binary or textual format for transmission over some transport via a Camel **Component**. Marshalling is used in the same way as unmarshalling above; in the **DSL** you can use a **DataFormat** instance, you can configure the **DataFormat** dynamically using the **DSL** or you can refer to a named instance of the format in the **Registry**.

The following example unmarshals via serialization then marshals using a named JAXB data format to perform a kind of **Message Translator**

```
from("file://foo/bar").
  unmarshal().serialization().
  marshal("jaxb").
  to("activemq:Some.Queue");
```

### Using Spring XML

This example shows how to configure the data type just once and reuse it on multiple routes

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <jaxb id="myJaxb" prettyPrint="true" contextPath="org.apache.camel.example"/>
  </dataFormats>
```

I used the following references in the composition of the present work (In order of importance):

1. Dollimore, Kindberg, Blair, Coulouris  
Distributed Systems 5th ed, Ch. 14  
Prentice Hall 2012

