# Lab Notes Week 3
# Distributed Systems

External data representation and marshalling

# Marshalling and Unmarshalling

- The information stored in running programs is represented as data structures
  - for example, by sets of interconnected objects
  - whereas the information in message consists of sequences of bytes.

- Irrespective of the form of communication used, the data structures must be
  - Flattened, converted to a sequence of bytes before transmission
  - and rebuilt on arrival.

- The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures.

# Marshalling and Unmarshalling

- There are two variants for the ordering of integers: the so-called big-endian order, in which the most significant byte comes first; and little-endian order, in which it comes last.

- Another issue is the set of codes used to represent characters:
  - for example, the majority of applications on systems such as UNIX use ASCII character coding, taking *one* byte per character, whereas
  - the Unicode standard allows for the representation of texts in many different languages and takes *two* bytes per character.

# External data representation and marshalling

- One of the following methods can be used to enable any two computers to exchange binary data values:
  - The values are converted to an agreed external format before transmission and converted to the local form on receipt;
    - if the two computers are known to be the same type, the conversion to external format can be omitted.
  - The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

- An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

# Marshalling and Unmarshalling

- *Marshalling* is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
  - Marshalling consists of the translation of structured data items and primitive values into an external data representation.

- *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.
  - Unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

# External data representation and marshalling

Approaches:

- CORBA's common data representation
  - which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages

- Java's object serialization
  - which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

- XML (Extensible Markup Language)
  - which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

# Serializing data to XML

- Pros
  - XML is  human readable
  - There are binding libraries for lots of languages.
  - A good choice if you want to share data with other applications/projects.
- Cons
  - Space intensive
  - Encoding/decoding it can impose a huge performance addition on applications.
  - Navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class.

# External data representation: other techniques

- Two other techniques for external data representation:
  - Google uses an approach called *protocol buffers (aka* protobuf) to capture representations of both stored and transmitted data
  - JSON (JavaScript Object Notation) is an approach to external data representation [www.json.org].
- Protocol buffers and JSON
  - more lightweight approaches to data representation
    - when compared, for example, to XML).

# SERIALIZABLE OBJECTS IN JAVA

# Serializable Objects

- The ability to store and retrieve Java objects is essential to building all but the most transient applications.

- The key to storing and retrieving objects in a serialized form is representing the state of objects sufficient to reconstruct the object(s).

- To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object.
- A Java object is serializable if its class or any of its superclasses implements either the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`. Deserialization is the process of converting the serialized form of an object back into a copy of the object.

- For example:
  - `java.awt.Button` class implements the `Serializable` interface, so you can serialize a `java.awt.Button` object and store that serialized state in a file.
  - Later, you can read back the serialized state and deserialize into a `java.awt.Button` object.

# The Serializable Interface

- **Serialization**
  - Objects of any class that implements the *java.io.Serializable* interface may be transferred to and from disc files as whole objects, with no need for decomposition of those objects.

- The *Serializable* interface is a marker to tell Java that objects of this class may be transferred on an object stream to and from files.

- Implementation of the *Serializable* interface involves no implementation of methods. The programmer only has to ensure that the class to be used includes the declaration '*implements Serializable* ' in its header line.

# The Serializable Interface

- The Serializable interface is defined to identify classes which implement the serializable protocol

```
package java.io;

public interface Serializable {};
```

- A Serializable class must do the following:

- Implement the j`java.io.Serializable` interface
- Identify the fields that should be serializable

# Writing to an Object Stream

- Class *java.io.ObjectOutputStream* is used to save entire object directly to disc
- For output:
  - wrap an object of class ObjectOutputStream around an object of class java.io.FileOutputStream, which itself is wrapped around a *java.io.File* object or file name.

```
ObjectOutputStream ostream =
    new ObjectOutputStream (
          new FileOutputStream("personnel.dat"));
```

# Reading from an Object Stream

- Class *java.io.ObjectInputStream* is used to read them back from disc.
- For input:
  - wrap an object *ObjectInputStream* object around a *jav.io.FileInputStream* object, which in turn is wrapped around a *java.io.File* object or file name.

```
ObjectInputStrem istream =
   new ObjectInputStrem(
         new FileInputStrem(“personnel.dat”));
```

# Writing/reading

- Methods *writeObject* (of *ObjectOutputStream*) and *readObject* (of *ObjectInputStream*) are then used for the output and input respectively.
- These methods write/read objects of class Object, therefore any objects read back from file must be **typecast** into their original class before we try to use them.

```
Personnel person = (Personnel)istream.readObject();
```

# writeObject() method

- Serializes the specified object and traverses its references to other objects in the object graph recursively to create a complete serialized representation of the graph.

```
final void writeObject(Object obj)
                        throws IOException
```

- Write the specified object to the `ObjectOutputStream`
- The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written.

# readObject() method

- Deserializes the next object in the stream and traverses its references to other objects recursively to create the complete graph of objects serialized.

```
final Object readObject()
        throws IOException, ClassNotFoundException
```

- Reads an object from the ObjectInputStream.
- The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are read.

# Possible Exceptions

- All exceptions thrown by serialization classes are subclasses of `ObjectStreamException` which is a subclass of IOException

```
InvalidClassException
NotSerializableException
StreamCorruptedException
NotActiveException
InvalidObjectException
OptionalDataException
WriteAbortedException
```

# Exceptions

- *Exceptions that needs to be cared about:*
  - *java.io.IOException*
  - *java.io.ClassNotFoundException*
  - *java.io.EOFException*
- In order to detect end of file, the java relies on catching the *EOFException.* This is not ideal, as:
  - Exception handling is designed to cater for exceptional and erroneous situations that we do not expect to happen if all goes well and processing proceeds as planned.
  - However, we are going to be using exception handling to detect something that we do not only know will happen eventually, but also are dependant upon happening if processing is to reach a successful conclusion.

# EOF handling example

```
...
try{
   do{
   Personnel p=(Personnel)istream.readObject();//Typecast.
   …
   }while (true);
   }
catch (EOFException e){
   System.out.println("\n\n*** End of file ***\n");
   istream.close();
}
…
```

# Transient fields

- Not every piece of program state can, or should be, serialized. Some things, like *FileDescriptor* objects, are inherently platform-specific or virtual-machine-dependent. If a *FileDescriptor* were serialized, it would have no meaning when deserialized in a different virtual machine.

- Even when an object is serializable, it may not make sense for it to serialize all of its state. To tell the serialization mechanism that a field should not be saved, simply declare it `transient`.

```
Protected transient short last_x, last_y
//temporary field for mouse position
```

# Transient fields (I)

- A field may not be transient – but for some reason it cannot be successfully serialized.
- Example:
  - A custom AWTR component that computes its preferred size based on the size of text it displays. Because fonts have slight size variations from platform to platform

# ArrayLists

- An object of class *java.util.ArrayList* is like an array, but can dynamically increase or decrease its size according to an application's changing storage requirements.

- It can hold only references to objects, not values or primitive types

- An individual *ArrayList* can hold only references to instances of a single, specified class.

# ArrayLists Constructors

```
public ArrayList()
```

- Constructs an empty list with an initial capacity of ten

- The class of elements that may be held in an ArrayList structure is specified in angle brackets after the collection class name, both in the declaration of the ArrayList and in its creation.

```
ArrayList<String> stringArray = new ArrayList<String>();
```

Or shortened:

```
ArrayList<String> stringArray = new ArrayList<>();
```

# java.util.ArrayLists

- Objects are added to the end of an ArrayList via method `add` and then referenced/retrieved via method `get`, which takes a single argument that specifies the object's position within the ArrayList.

Example:

```
String name1 = "Jones";
nameList.add(Name1);
String name2 = nameList.get(0);
```

Example 2: Addinf to a specific position

```
nameList.add(2, "Smith");
```

//added to 3rd position

# ArrayLists and Serialisation

- More efficient to save a single ArrayList to disc than is to save a series of individual objects
- Placing a series of objects into a single `ArrayList` is a very neat way of packaging and transferring our objects.
- Having some form of **random** access, via the ArrayList class's *get* method. Without this, we have the considerable disadvantage of being restricted to serial access only.
- See Example `ArrayListSerialise.java`
- It creates three objects of class *Personnel* and then uses the add method of class ArrayList to place the objects into a ArrayList. It then employs a 'for-each' loop and 'get' method of class Personnel to retrieve the data members of the three objects.

# Personnel class

```
class Personnel implements Serializable {
        private long payrollNum;
        private String surname;
        private String firstNames;

        public Personnel(long payNum,String sName,

        String fNames){
                payrollNum = payNum;
                surname = sName;
                firstNames = fNames;
        }
        public long getPayNum(){
                return payrollNum;
        }

        public String getSurname(){
                return surname;}
        public String getFirstNames(){
                return firstNames;
        }
        public void setSurname(String sName){
                surname = sName;
        }
}
```

# ArrayListSerialise class

```
import java.io.*;
import java.util.*;

public class ArrayListSerialise {
public static void main(String[] args) throws IOException, ClassNotFoundException {
    ObjectOutputStream outStream = new ObjectOutputStream(new
FileOutputStream("personnelList.dat"));
            ArrayList<Personnel> staffListOut = new ArrayList<>();
                    ArrayList<Personnel> staffListIn =  new ArrayList<>();

            Personnel[] staff
                    {new Personnel(123456,"Smith", "John"),
                    new Personnel(234567,"Jones", "Sally Ann"),
                    new Personnel(999999,"Black", "James Paul")};

                    for (int i=0; i<staff.length; i++)
                            staffListOut.add(staff[i]);
                    outStream.writeObject(staffListOut);
                    outStream.close();
                    ObjectInputStream inStream = new ObjectInputStream(new
FileInputStream("personnelList.dat"));

                    int staffCount = 0;
```

# ArrayListSerialise class (Cont)

```
try{
        staffListIn =  (ArrayList<Personnel>)inStream.readObject();
        //The compiler will issue a warning for the
        //above line, but ignore this!

        for (Personnel person:staffListIn) {
                staffCount++;
                System.out.println( "\nStaff member " + staffCount);

                System.out.println("Payroll number: "+ person.getPayNum());
                System.out.println("Surname: "          + person.getSurname());
                System.out.println("First names: "      + person.getFirstNames());
                }
        System.out.println("\n");
}
catch (EOFException eofEx){
        System.out.println("\n\n*** End of file ***\n");
        inStream.close();
        }
}
}
```

# Vectors

- An object of class *java.util.Vector* is like an array, but can dynamically increase or decrease its size according to an application's changing storage requirements

- Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

# java.util.Vector (I)

- Constructor overloading allows us to specify the initial size and, if we wish, the amount by which a Vector's size will increase once it becomes full.

- Vector method *elementAt*:

  `public Object elementAt(int index)`
  Returns the component at the specified index.

- Vectors versus ArrayLists
  - The ArrayList is faster, but it is **not** trade safe
  - Vector **is** thread safe
  - If thread-safety is important to your program, you should use a Vector. If not, then you should use an ArrayList.

# Protecting Sensitive Information

- The easiest technique is to mark fields that contain sensitive data as *private transient.*
- Transient fields are not persistent and will not be saved by any persistence mechanism.
- 
- Marking the field will prevent the state from appearing in the stream and from being restored during deserialization. Since writing and reading (of private fields) cannot be superseded outside of the class, the transient fields of the class are safe.

# Protecting Sensitive Information

- Particularly sensitive classes should not be serialized at all. To accomplish this, the object should not implement either the `Serializable` or the `Externalizable` interface.
- Some classes may find it beneficial to allow writing and reading but specifically handle and revalidate the state as it is deserialized.
- The class should implement writeObject and readObject methods to save and restore only the appropriate state.
- If access should be denied, throwing a `NotSerializableException` will prevent further access.

# Serialisation Today

- Many Java projects serialize Java objects using different mechanisms than the Java serialization mechanism.
- E.g. Java objects are serialized into JSON or other more optimized binary formats (BSON - (Binary JSON).
- This has the advantage of the objects also being readable by non-Java applications.
  - For example, JavaScript running in a web browser can natively serialize and deserialize objects to and from JSON.

# Java Serialisation – Future

According to InfoWorld (May 24, 2018)

- As of May 2018 – ORACLE intends to:
- Replace the current serialization technology

- A small serialization framework would be placed in the platform once *records*, the Java version of data classes, are supported.

- The framework could support a graph of records, and developers could plug in a serialization engine of their choice, supporting formats such as JSON or XML, enabling serialization of records in a safe way.

- Not confirmed in which release of Java

# EXTERNAL DATA REPRESENTATION: OTHER TECHNIQUES

# JSON (JavaScript Object Notation)

- JSON is a lightweight data-interchange format based on a subset of the JavaScript Programming Language Standard ECMA-262
- Derived from JavaScript that is used in web services and other connected applications.
- Browsers can parse JSON into JavaScript objects natively.
- On the server, JSON needs to be parsed and generated using JSON APIs.

# JSON

- Built on two structures:
  - A collection of name/value pairs
    - E.g. object, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values
    - E.g. array, vector, list, or sequence.

- These are universal data structures. Virtually all modern programming languages support them in one form or another.
  - It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

# JSON Syntax

- JSON defines seven value types:
  - `string, number, object, array, true, false, and null`
- An object is an unordered collection of zero or more name/value pairs.
- An array is an ordered sequence of zero or more values.

# JSON Syntax

- Objects are enclosed in braces ({}),
  - their name-value pairs are separated by a comma (,),
  - and the name and value in a pair are separated by a colon (:).
  - Names in an object are strings, whereas values may be of any of the seven value types, including another object or an array.
- Arrays are enclosed in brackets ([]),
  - their values are separated by a comma (,).
  - Each value in an array may be of a different type, including another array or an object.
- When objects and arrays contain other objects or arrays, the data has a tree-like structure.

# JSON Example

JSON representation of an object that describes a person. The object has string values for first name and last name, a number value for age, an object value representing the person's address, and an array value of phone number objects.

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

# Uses of JSON

- Ajax applications
- Configurations
- Databases
- RESTful web services:
  - All popular websites offer JSON as the data exchange format with their RESTful web services.
  - RESTful web services are web services which are REST based.
    - Representational State Transfer (REST) is an approach in which clients use URLs and the HTTP operations GET, PUT, DELETE and POST to manipulate resources that are represented in XML.
    - The emphasis is on the manipulation of data resources rather than on interfaces.

# JSON - Java

- Java API for JSON Processing (JSON-P)

- JSON-B (Java API for JSON Binding)
  - built on top of JSON-P
  - Included in  Java EE 8

# Java API for JSON Processing (JSON-P)

- API used for parsing, generating, querying, and transforming JSON documents

- It produces and consumes JSON text in a streaming fashion and allows to build a Java object model for JSON text using API classes

# Java API for JSON Binding (JSON-B)

- JSON-B is a standard binding layer for converting Java objects to/from JSON data structures (messages).
- Built on top of JSON-P
- It defines a default mapping algorithm for converting existing Java classes to JSON
- Enables developers to customize the mapping process through the use of Java annotations.

# Protocol Buffers (*protobuf*)

- A mechanism for serializing structured data.
- Similar to XML
  - smaller, faster, and simpler
- Uses binary format
  - rather than text format of XML and JSON
- Is in fact an IDL (Interface Definition Language)

# Protocol Buffers

- Properties:
  - Efficient, binary serialization
  - Support protocol evolution
    - Can add new parameters
    - Order in which parameters are specified is not important
    - Skip non-essential parameters
  - Supports types, which give you compile-time errors
  - Supports quite complex structures

- Usage:
  - It is a binary encoding format that allows you to specify a *schema* for your data
  - Protocol buffers are used for other things, e.g., serializing data to non-relational databases – their backward-compatible feature make them suitable for long-term storage formats

- Google uses them

# Google Protocol Buffers

- Supported languages
  - Java, C++, Python
- As a developer you just need to define data structure (message) formats in a *.proto* definition file
- Compile it with `protoc` compiler
  - that Google provides. The protocol buffer compiler generates source code from .proto file in Java programming language and an API to read and write messages on *protobuf* object.

# Google Protocol Buffers: Developing

- Write a `.proto` description of the data structure you wish to store.

- The protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format.
- The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit.

- The protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.
- Details as how to use protocol buffers:

`http://developers.google.com/protocol-buffers/docs/javatutorial`

# `.proto` file example

- You add a *message* for each data structure you want to serialize, then specify a name and a type for each field in the message.

- the `.proto` file that defines your messages, `addressbook.proto`

```
syntax = "proto2";
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phones = 4;
}
message AddressBook {
  repeated Person people = 1;
}
```

# JSON-P AND JSON-B EXAMPLES

# Example - JSON-P: The JSON representation file

```
{
    "name": "Falco",
    "age": 3,
    "bitable": false
}
```

# Example - JSON-P

JSON-P Object Model calls:

```
public static void main(String[] args) {
    // Create Json and serialize
    JsonObject json = Json.createObjectBuilder()
      .add("name", "Falco")
      .add("age", BigDecimal.valueOf(3))
      .add("biteable", Boolean.FALSE).build();
    String result = json.toString();

    System.out.println(result);
  }
```

# Example - JSON-P Streaming API calls

```
public static void main(String[] args) {
    // Parse back
    final String result =
"{\"name\":\"Falco\",\"age\":3,\"bitable\":false}";
    final JsonParser parser = Json.createParser(new
StringReader(result));
    String key = null;
    String value = null;
    while (parser.hasNext()) {
        final Event event = parser.next();
        switch (event) {
        case KEY_NAME:
            key = parser.getString();
            System.out.println(key);
            break;
        case VALUE_STRING:
            value = parser.getString();
            System.out.println(value);
            break;
        }
    }
    parser.close();
}
```

# Example - JSON-B

Java Class:

```
public class Dog {
	public String name;
	public int age;
	public boolean bitable;
}
```

The JSON representation:

```
{
    "name": "Falco",
    "age": 4,
    "bitable": false
}
```

# Example – JSON-B: To serialize/deserialize a Java Object to/from JSON

```
public static void main(String[] args) {
// Create a dog instance
Dog dog = new Main.Dog();
dog.name = "Falco";
dog.age = 4;
dog.bitable = false;

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dog);

System.out.println(result);

// Deserialize back
dog =
jsonb.fromJson("{\"name\":\"Falco\",\"age\":4,\"bitable\":false}",
Dog.class);
}
```

# References


- Chapter 4, *Introduction to Network Programming in Java* by Jan Graba
- Chapter 4: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design
- http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
- https://www.infoworld.com/article/3275924/java/oracle-plans-to-dump-risky-java-serialization.html
- https://developers.google.com/protocol-buffers/docs/javatutorial
- https://www.json.org/
- https://www.oracle.com/technetwork/articles/java/json-1973242.html

- https://www.javaworld.com/article/3353559/java-xml-and-json-document-processing-for-java-se-part-2-json-b.html
- https://javaee.github.io/jsonp/