# Serialization

## Reflective Serialization

- A reflective serializer should serialize any type of object passed in as a parameter

**Basic design:**

- Give the object a unique identifier number
  - Could be done with `java.util.IdentityHashMap`
- Get a list of all the object's fields
  - Of all visibilities
    - Use `getDeclaredFields()` and traverse the inheritance hierarchy
  - Filter out static fields
- Uniquely identify each field with its:
  - Declaring class
  - Field name
- Get the value for each field
  - If a primitive, simply store it so it can be easily retrieved later
  - If a non-array object, recursively serialize the object
    - Use the new object's unique id number as a reference
    - Store the reference as the field value in the originating object
    - Don't serialize an object more than once
      - Occurs when you have several references to the same object
  - If an array object, serialize it
    - Then serialize each element of the array
      - Use recursion if the element is an object

## Dynamic Loading

- An ordinary class can be loaded at runtime using `public static Class forName(String className)`
- Throws `ClassNotFoundException` if the corresponding `.class` file is not found on the classpath

**Arrays**:

- Array classes do not have a `.class` file. No normal class name (are generated as needed by the JVM)
- Array classes are named using codes:

| Encoding | Element Type |
|---|---|
| B | byte |
| C | char |
| D | double |
| F | float |
| I | int |
| J | long |
| L `<type>` | reference type |
| S | short |
| Z | boolean |

- For each dimension of the array, use a `[`, then their element type code
  - 1D int array: `[I`
  - 2D float array: `[[F`
  - 1D array of objects: `[Ljava.lang.String`
- Array classes can be loaded using `forName()`
  - Eg. array of String objects:
    ```
    Class classObject = Class forName("[Ljava.lang.String");
    ```

## Reflective Deserialization

**Basic design:**

- Get a list of objects, stored in the XML document
  - Use `getRootElement()` from the `Document` class, and `getChildren()` from `Element` class
- For each object, create an uninitialized instance
  - Dynamically load its class using `forName()`
    - The class name is an attribute of the *object* element
- Create an instance of the class
  - If a non-array object get the declared no-arg constructor, then use `newInstance()`
    - May need to `setAccessible(true)`
  - If an array object, use `Array.newInstance(...)`
    - Use `getComponentType()` to find element type
    - The length is an attribute of the *object* element
- Associate the new instance with the object's unique identifier number using a table
  - `java.util.HashMap` is ideal
    - The id is the key
    - The object reference is the value
  - The id is an attribute of the *object* element
- Assign values to all instance variables in each non-array object:
  - Get a list of the child elements
    - Use `getChildren()` from `Element` class
    - Each child is a field of the object
  - Iterate through each field in the list
    - Find the name of its declaring class
      - Is an attribute of the *field* element
    - Load the class dynamically
    - Find the field name
      - Is an attribute of *field* element
    - Use `getDeclaredField()` to find `Field` metaobject
    - Initialize the value of the field using `set()`
      - If a primitive type, use the stored value
        - Use `getText()` and create appropriate wrapper object
      - If a reference, use the unique identifier to find the corresponding instance in the table
      - May need to `setAccessible(true)`
  - Array objects are treated specially:
    - Find the element type with `getComponentType()`
    - Iterate through each element of the array
      - Set the element's value using `Array.set()`
      - As above, treat primitives differently than references

## Serializer Code

```
public class ObjectMap {
    private HashMap<Integer, Object> objects;
    public HashMap<Integer, Object> getObjects() { return objects; }
    public Object get(int i) { return objects.get(i); }
    public ObjectMap() {
        objects = new HashMap<Integer, Object>();
    }

    /* Recursively get all objects associated with obj */
    public void populate(Object obj) {
        objects.put(obj.hashCode(), obj);
        Field[] fields = FieldHelper.findFields(obj.getClass());
        for (Field f : fields) {
            Object value;
            value = f.get(obj); // NullPointerException, IllegalAccessException
            if (value == null) { continue; }
            if (f.getType().isArray()) {
                objects.put(value.hashCode(), value);
                int length = Array.getLength(value);
                for (int i = 0; i < length; i++) {
                    Object o = Array.get(value, i);
                    if (!isValidObject(o)) { continue; }
                    populate(o);
                }
            } else {
                // If it isn't a primitive object, populate it if it isn't already in the list
                if (!isValidObject(value)) { continue; }
                populate(value);
            }
        }
    }
    private boolean isValidObject(Object o) {
        if (o == null) { return false; }
        if (FieldHelper.isPrimitive(o)) { return false; }
        if (objects.containsKey(o.hashCode())) { return false; }
        return true;
    }
}
```

```
public class Serializer {
    ObjectMap objects;
    public Document serialize(Object obj) {
        Element root = new Element("serialized");
        Document doc = new Document(root);
        objects = new ObjectMap();
        objects.populate(obj);

        // Ensure that the given object is always the first one serialized
        Element e = serializeObject(obj);
        root.addContent(e);
        for (Integer i : objects.getObjects().keySet()) {
            if (i == obj.hashCode()) { continue; }
            e = serializeObject(objects.get(i));
            root.addContent(e);
        }
        return doc;
    }
    private Element serializeObject(Object obj) {
        Element e = new Element("object");
        e.setAttribute("class", obj.getClass().getName());
        e.setAttribute("id", Integer.toString(obj.hashCode()));
        if (obj.getClass().isArray()) {
            e.setAttribute("length", Integer.toString(Array.getLength(obj)));
            serializeArray(obj, e);
        } else {
            serializeNormalObject(obj, e);
        }
        return e;
    }
```

```
    private void serializeNormalObject(Object obj, Element element) {
        Field[] fields = FieldHelper.findFields(obj.getClass());
        if (fields.length == 0) { return; }
        for(Field f : fields) {
            if (Modifier.toString(f.getModifiers()).contains("static")) { continue; }
            Object value;
                        value = f.get(obj); // NullPointerException, IllegalAccessException
            Element e = serializeField(obj, f, value);
            element.addContent(e);
        }
    }
    private Element serializeField(Object obj, Field f, Object value) {
        Element e = new Element("field");
        e.setAttribute("name", f.getName());
        e.setAttribute("declaringclass", getDeclaringClass(obj.getClass(), f));
        Element v = serializeValue(value);
        e.addContent(v);
        return e;
    }
    private void serializeArray(Object obj, Element element) {
        int length = Array.getLength(obj);
        for (int i = 0; i < length; i++) {
            Object o = Array.get(obj, i);
            Element value = serializeValue(o);
            element.addContent(value);
        }
    }
    private Element serializeValue(Object o) {
        if (o == null) {
            Element f = new Element("value");
            f.addContent("null");
            return f;
        } else if (FieldHelper.isPrimitive(o)) {
            Element f = new Element("value");
            f.addContent(o.toString());
            return f;
        } else {
            Element r = new Element("reference");
            r.addContent(Integer.toString(o.hashCode()));
            return r;
        }
    }
    private String getDeclaringClass(Class c, Field f) {
        if (c == null) { return ""; }
        for (Field f2 : c.getDeclaredFields()) {
            if (f.equals(f2)) { return c.getName(); }
        }
        return getDeclaringClass(c.getSuperclass(), f);
    }
}
```

## Deserializer Code

```
public HashMap<Integer, Object> objects;

public Object deserialize(Document document) {
        Object o = null;
        objects = new HashMap<Integer, Object>();
        mapElements(document);

        for (Element e : document.getRootElement().getChildren()) {
                if (e.getAttribute("length") != null) {
                        deserializeArray(e);
                } else {
                        int id;
                        id = e.getAttribute("id").getIntValue(); // DataConversionException
                        o = objects.get(id);
                        deserializeNormalObject(e, o);
                }
        }

        // Return the first object that was serialized
        int id = document.getRootElement()
                        .getChildren().get(0)             // IndexOutOfBoundsException
                        .getAttribute("id").getIntValue();// DataConversionException
        o = objects.get(id);
        return o;
}
```

```
private void mapElements(Document document) {
        for (Element e : document.getRootElement().getChildren()) {
                String classString = e.getAttribute("class").getValue();
                try {
                        Class c = Class.forName(classString);
                        int id = e.getAttribute("id").getIntValue();
                        if (c.isArray()) {
                                // Add an array of correct length full of default values to objects
                                int length = e.getAttribute("length").getIntValue();
                                Class type = c.getComponentType();
                                Object o = Array.newInstance(type, length);
                                objects.put(id, o);
                        } else {
                                // Add the object by default constructor to objects
                                Constructor con = c.getDeclaredConstructor(new Class[0]);
                                con.setAccessible(true);
                                Object o = con.newInstance();
                                objects.put(id, o);
                        }
                } catch(Exception err) {
                        System.out.println(err);
                        return;
                }
        }
}
```

```
private void deserializeNormalObject(Element e, Object o) {
        for (Element field : e.getChildren("field")) {
                // Read name and class from attributes
                String fieldName = field.getAttributeValue("name");
                String declaringClass = field.getAttributeValue("declaringclass");

                // Get a copy of the Class and Field
                Class c = null;
                c = Class.forName(declaringClass); // ClassNotFoundException
                Field f = null;
                f = c.getDeclaredField(fieldName); // NoSuchFieldException
                f.setAccessible(true);

                // Set primitive value, or object reference ID
                if (field.getChildren("value").size() > 0) {
                        String text = field.getChildren("value").get(0).getText();
                        Object value = wrapObject(text, f.getType());  //<Type>.valueOf(text)
                        f.set(o, value); // IllegalAccessException
                } else if (field.getChildren("reference").size() > 0) {
                        String value = field.getChildren("reference").get(0).getTextTrim();
                        int objId = Integer.valueOf(value);  // NumberFormatException
                        f.set(o, objects.get(objId));        // IllegalAccessException
                }
        }
}
```

```
private void deserializeArray(Element e) {
        int id;
        int length;
        id = e.getAttribute("id").getIntValue();        // DataConversionException
        length = e.getAttribute("length").getIntValue();// ^

        if (e.getChildren("value").size() > 0) {
                List<Element> children = e.getChildren("value");
                for (int i = 0; i < length; i++) {
                        String text = children.get(i).getText();
                        Object value = wrapObject(text, objects.get(id).getClass().getComponentType());
                        Array.set(objects.get(id), i, value); // IllegalArgumentException
                }
        } else if (e.getChildren("reference").size() > 0) {
                List<Element> children = e.getChildren("reference");
                for (int i = 0; i < length; i++) {
                        String text = children.get(i).getTextTrim();
                        int objId = Integer.valueOf(text); // NumberFormatException
                        Array.set(objects.get(id), i, objects.get(objId)); // IllegalArgumentException
                }
        }
}
```

# Java Sockets

```
public class Sender {
        private static final String host = "localhost";
    private static final int port = 6666;

        public void sendDocument(Document doc) {
            Socket socket = null;
                socket = new Socket(host, port); // IOException

                BufferedOutputStream bufferedStream = null;
                bufferedStream = new BufferedOutputStream(socket.getOutputStream()); // IOException

                XMLOutputter out = new XMLOutputter(Format.getPrettyFormat());
        ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();

                // IOException
                out.output(doc, byteOutputStream);
                byte[] byteList = byteOutputStream.toByteArray();
                bufferedStream.write(byteList);
                bufferedStream.flush();

                // IOException
                bufferedStream.close();
                byteOutputStream.close();
                socket.close();
        }
}
```

```
public class Receiver {
    private static final int port = 6666;
    private ServerSocket server;
    private Document document;

        public void start() {
                // Open the socket connection
                server = new ServerSocket(port); // IOException
                Socket sender;
                sender = server.accept(); // IOException

                // Receive the serialized document
        BufferedReader in;
        in = new BufferedReader(new InputStreamReader(sender.getInputStream())); // IOException
                SAXBuilder saxBuilder = new SAXBuilder();
                document = saxBuilder.build(in); // JDOMException

                // Close sockets
                server.close();
                sender.close();
                in.close();
        }
}
```