CSC301

Serialization & Persistence

Individual Assignments

- A2 auto-marker results will be out today or tomorrow
  - A new branch, called `auto-marker`, in your original A2 repo (*not* your fork)
  - Please double check the results and let Adam know if there is a mistake
- A3 is out
  - Start early!
  - Finish the assignment (at least) 12 hours before the deadline.
    - There are no exceptions with auto-marked assignments, don't leave the submission to the last minute.
  - Do NOT change the interfaces or testing code!
    - These are the specifications.
  - A3 is not huge, but it's still heavier than the previous assignments
  - **Deadline for A3: February 16, 11:59pm**

Where does the data live?

- Most work is done in multiple sessions
  - You expect to be able to open files that you previously saved
  - You expect your desktop background to be the same, even after you restart the computer
  - You expect your repos to be available, the next time you log into GitHub
  - You expect an app to start where you left off

Where does the data live?

- When you restart a program, all of its in-memory objects are gone
  - Are these objects saved (aka *persisted*) anywhere?
  - If they are, how can we get (some of) them back?

Persistence

- Make data available beyond one invocation of an application
  - Save data to non-volatile storage
    - Storage that “survives a restart”
    - For example: Local hard-drive, flash drive, cloud, etc.
  - Load data back into memory when needed
    - E.g.: Open a local file, read from the network

Serialization

- *Serialize*
  - Convert in-memory objects into data so that it can be:
    - Written/saved somewhere
    - Streamed across a communication link
  - In this context, *data* (usually) means a string or an array of bytes
- *Deserialize*
  - Convert (serialized) data into in-memory objects

Terminology

- *Persist*
  - Save serialized data to non-volatile storage
- Serialization ≠ Persistence
  - We persist serialized objects
  - We deserialize data read from persistent storage
  - Distinction is important: If something goes wrong, you should know where to find the bug
- In this lecture, we will focus on serialization

Example, Java Built-in Serialization

```
LocalDateTime t = LocalDateTime.now();
System.out.println("Serializing " + t);

OutputStream out = new FileOutputStream(new File("tmp.data"));
ObjectOutputStream serializer = new ObjectOutputStream(out);
serializer.writeObject(t);
out.close();

InputStream in = new FileInputStream(new File("tmp.data"));
ObjectInputStream deserializer = new ObjectInputStream(in);
LocalDateTime t2 = (LocalDateTime) deserializer.readObject();
System.out.println("Deserialized " + t2);
in.close();
```

Built-in Serialization, Limitations

- Cannot serialize arbitrary objects
  - Objects must implement java.io.Serializable
  - The transitive closure (i.e., objects, their references, their references' references, and so on) must implement `Serializable` as well.
- Deserialization is messy
  - Requires casting
  - Requires us to know in advance what type of object we are going to read

Built-in Serialization, Limitations

- Not cross-language
  - Uses a Java-specific binary format
- Instances can be serialized, but not the classes themselves (e.g., methods are not serialized)
  - Therefore, cannot deserialize data without the compiled classes
- Not necessarily ideal
  - E.g: Images can be serialized using more suitable formats, such as png, jpg, svg, etc.

Serialize To Text

- How can we overcome the limitations of Java's built-in serialization?
- One option is to serialize objects to text
  - Flexibility - Easily define custom serialization
  - Convenient deserialization - Read text, then determine the type of the object
  - Cross-language - Every programming language is capable of text processing
  - Convenient debugging - Text is human-readable

XML

- EXtensible Markup Language
- Text format
- Tags provide structure to data
  - Tags are named
  - Tags can be nested
  - Tags can have attributes

XML Example

```xml
<?xml version="1.0"?>
<catalog>
    <product category="books">
        <title>Clean Code</title>
    </product>
    <product category="books">
        <title>GoF - Design Patterns</title>
    </product>
    <product category="music">
        <title>Abbey Road</title>
        <artist>The Beatles</artist>
        <year>1969</year>
    </product>
</catalog>
```

- Tag names are user defined
  (That's the eXtensible part of XML)
- White spaces are ignored
  Can be added for pretty-printing
- You've already seen an example of XML - The `pom.xml` of a Maven project

Java XML Serialization

- A few differents XML libraries
- Different libraries = Different trade-offs
  - Memory usage vs. Flexibility
    E.g.: If processing document as a stream of XML elements, we can reduce memory usage (only need to store the current element) at the cost of losing flexibility (when processing an element, we don't have the full document).
  - Performance vs. Convenience.
    E.g.: Transparently convert between XML documents and instances of user-defined classes
    (see code example on the next couple of slides).

Example: XML Serialization

```java
@XmlRootElement
public class Person {

	private String name;
	private int age;

	// Constructor is required for serialization
	public Person() {
	}

	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}

	@XmlElement
	public String getName() { return name; }

	@XmlAttribute
	public int getAge() { return age; }
}
```

Annotate user-defined class with JAXB (Java Architecture for XML Binding)

Example: XML Serialization

Serialize instances of our annotated class

```
public static void main(String[] args) throws Exception{
          Marshaller serializer = JAXBContext.newInstance(Person.class).createMarshaller();

          StringWriter sw = new StringWriter();
          serializer.marshal(new Person("Alice", 27), sw);
          System.out.println(sw.toString());
}
```

Marshalling ~= Serializing

And print the resulting XML document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person age="27"><name>Alice</name></person>
```

XML Advantages

- All the advantages of a text-based format
  - Cross-language, flexible, human readable.
- Schema validation using DTD
- Namespaces
  - Avoid conflicts between documents
- Transformation using XSLT
- Many other tools and libraries
  - XML was the standard in the Java world for many years

XML Disadvantages

- Performance overhead!
  - Sometimes there’s more markup than actual data
    Significant when data is transferred over a network
  - Parser is complicated and very slow
- Not so human-readable
- Features turned out to be of low value to developers, who wanted a simpler format...

JSON

- JSON (JavaScript Object Notation)
- Light, cross-language text format
  - Strings, numbers, booleans and null
  - Arrays
  - Objects (aka maps, dictionaries, hashes, etc.)
    - One limitation: Keys must be strings

JSON, Example

```
{
    "name" : "Alice",
    "age"    : 37,
    "married" : true,
    "education" : [
              {"where" : "UofT", "degree" : "Hon. B.Sc in Comp Sci"},
              {"where" : "OCAD", "degree" : "M.A Digital Arts"}
     ]
}
```

- Easily nest objects
- White spaces are ignored, and can be added for pretty-printing

Example, JSON Serialization

Serialize/deserialize using Jackson-databind

```
public static void main(String[] args) throws Exception{

      ObjectMapper mapper = new ObjectMapper();
      System.out.println(mapper.writeValueAsString(new Person("Alice", 27)));

      String s = "{\"name\":\"Alice\",\"age\":27}";
      Person p = mapper.readValue(s, Person.class);
      System.out.println(p.getName() + ", " + p.getAge());
}
```

The main method above will result in

```
{"name":"Alice","age":27}
Alice, 27
```

JSON advantages

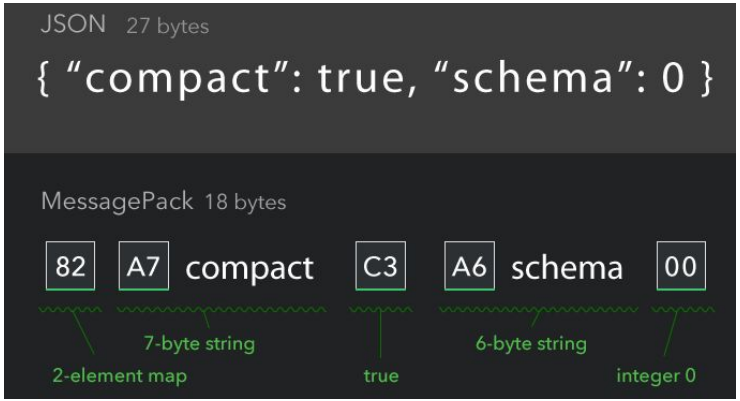
- Many tools & libraries in many languages
  - JSON has been the standard format in the last ~8 years
  - Jackson JSON Processor is the most popular Java library
    - Different modes allow different trade-offs between convenience and performance.
- Much lighter than XML
  - Less markup
  - Allows for simpler (and faster) parsing

JSON Disadvantages

- Lack of types.
  - Date/Time
  - Objects with non-string keys
  - Sets
  - etc.
- Performance might be better than with XML, but newer binary formats can result in smaller data and faster parsing

Cross-Language Binary Serialization

- Newest trend in serialization frameworks
  - Serialize to a binary format
  - Provide libraries for multiple languages
  - Result in smaller serialized data & better performance than text formats (e.g., XML and JSON).
  - E.g.: Protocol buffers, MessagePack, Avro, Thrift



An example from `MessagePack`’s home page

Cross-Language Binary Serialization

- Better performance than text formats, like JSON or XML
- The trade-off: Not as easy to use
  - Frameworks and libraries are more complex
  - Hard to debug when data is not human readable

Serialization, Recap

- Built-in Java serialization
  - Good: Built-in, simple binary format
  - Bad: Java-only, limited features
- XML and JSON
  - Good: Flexible, human-readable text formats
  - Bad: Data size and parsing time overhead
- Newer Serialization Framework
  - Good: Compact binary format + fast parsing
  - Bad: Less convenient than text formats

Before we wrap up the discussion on serialization, let’s take a detailed look at a couple of related challenges…

Serialization, Object References

Consider a hypothetical `IUser` interface

```
public interface IUser{
     String getName();
     Iterator<IPost> getPosts();
     // ...
}
```

Now, say we serialize an `IUser` instance ...

Serialization, Object References

- When deserializing an `IUser`
  - We expect `getPosts` to return an iterator that visits all of the user's posts.
  - Therefore, when serializing an `IUser`, we need to serialize all of its `IPosts`.
- Therefore, we need to serialize/deserialize a network of objects.

Serialization, Object References

- Garbage Collection makes things challenging
  - Java object references ≠ C pointers
  - Compaction results in objects being moved around in memory.

    Asynchronously to your program!
  - Cannot identify an object based on “its address”
  - True for every memory-managed language
- Idea: Replace object references with some identification during serialization/deserialization

Serialization, Object References

- Possible solution: Serialize objects and relations separately
  - E.g., serialize users and posts separately from the *is-author* relation between users and their posts
  - In some domains, we might have to follow relations/links recursively
- If objects have unique IDs, we can use them.
  Otherwise, make up IDs during serialization/deserialization.
- Need to handle cycles!
- Might need to limit how much of the object graph we actually serialize
  - Can use a lazy-loading for deserialization (i.e., some parts of the graph will be loaded to memory, only if we actually try to access them)

From Serialization To Persistence

- Once we decide on a serialization method, how do we persist serialized data?
- Simple approach - Write to a file
  - Serialize all of our program's objects
  - Write them, one by one, to a flat file

Flat File Persistence

- Good for simple cases
  - Very small files that change infrequently
  - E.g.: Settings or configuration files
- But limited
  - Impractical for a large set of objects

    E.g.: Facebook's data cannot fit in a single file
  - Search/update require going through the whole file

Additional Limitations

- Concurrent writes may corrupt the data
- Potential data loss
  - Need a backup
  - Hard to guarantee no data loss, even with backup

Persistent Data Store

- Persistence is challenging when the data
  - is large
  - is accessed simultaneously by many clients
  - changes frequently
  - needs to be queried efficiently
  - needs be highly-available (i.e., no down time, ever)
- There are many persistent data store solutions, of many shapes and flavours.

The *Data Access Object* Design Pattern

- *Data Access Object*
  - Define data-access methods in an interface
  - Implement the DAO interface for different data stores
  - Code is written against the interface, without knowing/caring about the underlying data store.
- DAO is a design pattern for abstracting the details of an underlying data store
  - Allows us to write database-agnostic applications

Data Access Object

Allows your application code to look like

```
public static void example(ITweetEngine dao){
     Iterator<ITweet> itr = dao.getTrendingTweets();
     while(itr.hasNext()){
          System.out.println(itr.next());
     }
}
```

- Application code is not responsible for persisting data
- Application code does not depend on a specific data store
- DAO methods use domain language (e.g., trending tweets)

Data Access Object

- Can be extremely useful when starting to work on a project:
  - Many applications (especially on the web) rely on a database / data store.
  - When you start working on a project, you might want to avoid
    - Choosing (i.e., committing to) data store(s)
    - Integrating the data store into your application
  - A simple solution is to define the DAO, and create a simple in-memory implementation
    - The in-memory implementation is not persistent and does not scale, but it allows us to get started quickly.
    - Once the basic structure and design of your application is ready, you can replace your in-memory implementation with a more realistic one (e.g., backed by a relational database)
  - You can leverage this for testing as well
    - DAO returns you test data instead of retrieving from the database to reduce test run time

Individual Assignments

- A3 is about serialization
  - First, implement the `IGrid<T>` data type
    - Maps keys (of type `GridCell`) to values (either objects of type `T` or `null`)
    - We use an `IGrid<Rack>` to represent the floor plan of a warehouse.
  - Then create two grid implementations
    - One supports only rectangular shaped grids
    - The other supports any shape
  - Then, implement two serializer/deserializers
    - Each implementation uses a different text format
    - Important: Both implementations serialize/deserialize `IGrid<T>` instances (i.e., they are NOT limited to a specific implementation)
  - With this design we can convert between files in different formats
    - Deserialize `.rect.txt` file to `IGrid<Rack>` instance, then serialize to `.flex.txt` file
    - Similar to the way one might convert between Word documents and PDF files.