# CSC301

Serialization & Persistence

# 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 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”
    - Ex: Local hard-drive, cloud
  - Load data back into memory when needed
    - Ex: Open a local file, read from the network

# Serialization

- *Serialize*
  - Convert in-memory objects into data (that can be written/saved somewhere)
- *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

# 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 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
- Classes themselves are not serialized
  - Cannot deserialize data without the compiled classes
- Not necessarily ideal
  - Ex: 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
      Ex: Process document as a stream of XML elements
    - Performance vs. Convenience.
      Ex: Transparently convert between XML documents and instances of user-defined classes.

# 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

# 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 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.
- Binary formats can result in smaller data and faster parsing
  - Ex: 123456789 vs. “123456789”

# 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).
  - Ex: Protocol buffers, MessagePack, Avro, Thrift

# 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

# Your Next Individual Assignment

- Serialization will be the main topics of your next individual assignment.
- We are mentioning it now, because we do not have a lecture next week (Thanksgiving Monday)
- Two other topics that might be useful for the next assignment are:
  - Lambda expressions
  - Iterators (which you should have seen in CSC207)