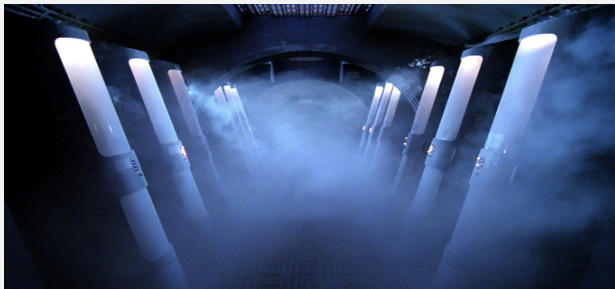


# Object Serialization



# Object Serialization: *Time to freeze-dry?*

- ▶ **Serialization** is an (automatic) way to save and load the state of an object from a stream. The serialized object could then be stored in a file, database or sent over the network.
- ▶ To be able to serialize a class, two conditions must be met:
  - ▶ The class must implement the `java.io.Serializable` interface. The interface is empty and serves simply as permission to serialize!
  - ▶ All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.
- ▶ `ObjectInputStream`, `ObjectOutputStream` are special stream subclasses used to serialize objects. Subclasses of `Serializable` classes are also serializable.
- ▶ When an object is serialized, any object references it contains are also serialized. Serialization can capture entire “graphs” of interconnected objects and put them back together on the receiving end.

[Examples are in the repository folder `examples/serialization`.]

# Basic Examples

- ▶ Example 1: `SaveTable.java`, `LoadTable.java`.
  - ▶ Shows how to (de)serialize a standard Java data structure.
- ▶ Example 2: `MioAlma.java`, `Cryogenics.java`
  - ▶ Shows how to (de)serialize our own class. Also shows the affect of the `transient` keyword.

# Serial Version UID

- ▶ By default, serialization uses a calculated hash based on a given source file — method names, field names, field types, access modifiers, etc., and compares that hash value against the hash value in the serialized stream.
- ▶ To convince the compiler that two versions of a class are equivalent, each class that is serializable can declare a version number, called a `serialVersionUID`
- ▶ The `serialVersionUID` is declared via a field named "serialVersionUID" that must be static, final, and of type long:  
`<access-modifier> static long serialVersionUID = <UID>L;`
- ▶ Use the program `serialver` (bundled with Java) to generate the `serialVersionUID` for a given class. Or use your IDE to generate it for you!

# Serialization versus Refactoring

- ▶ Serialization permits some variation in classes such that serialized objects can be revived by various versions of a class.
- ▶ Using `serialVersionUID`, Java serialization can automatically manage the following refactorings:
  - ▶ Adding new fields to a class
  - ▶ Changing the fields from static to nonstatic
  - ▶ Changing the fields from transient to nontransient
- ▶ Going the other way requires additional processing depending upon the amount of backwards compatibility desired.

# Serialization Experiments

Examples: `MioAlma.java`, `Cryogenics.java`, `MioAlmaDos.java`.

1. Run and freeze a `MioAlma` object (version 1) using the `Cryogenics` program.
2. Refactor the `MioAlma` class to add an email field. Or simply copy and paste the code from `MioAlmaDos.java` over the code in `MioAlma.java` to create version 2 of the class.
  - 2.1 Using this new class, attempt to revive an `MioAlma` version 1 serialized object.
  - 2.2 Note that this will fail as the class has changed and we didn't use a serial version UID.
3. Revert `MioAlma` to version 1. Add the serial version UID to the `Mio` class (version 1)
  - 3.1 Refactor `MioAlma` into version 2 (as before).
  - 3.2 Using this new class, attempt to revive an `MioAlma` version 1 serialized object.
  - 3.3 This time it will work. Note that the email field will be set to null.
4. Run and freeze a `MioAlma` object (version 2)
  - 4.1 Revert `MioAlma` back to version 1.
  - 4.2 Revive a `MioAlma` version 1 object from a freeze-dried `MioAlma` version 2 object!

- ▶ It is strongly recommended that all serializable classes explicitly declare `serialVersionUID` values.
  - ▶ *Reason:* The default `serialVersionUID` computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected `InvalidClassException` exceptions.

# Custom Deserialization

- ▶ Simple deserialization may not be enough to reconstruct the full state of an object. Objects can do their own setup after deserialization by implementing the `readObject()` (as well as `writeObject` method).
- ▶ The methods are defined in the interfaces `ObjectInput` and `ObjectOutput` in `java.io` package.

```
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream stream)
    throws IOException;
```

- ▶ Example:

```
private void readObject (ObjectInputStream s)
{
    s.defaultReadObject(); //standard deserialization
    initialize(); //our custom initialization
    //call optional method after customization
    if (isRunning)
        start();
}
```



# Overriding Deserialization

- ▶ Use the `java.io.Externalizable` interface to override the serialization process.

<code>void readExternal(ObjectInput in)</code>
<code>void writeExternal(ObjectOutput out)</code>

- ▶ `readExternal`: Implement the method to restore its contents by calling the methods of `DataInput` for primitive types and `readObject` for objects, strings and arrays.
- ▶ `writeExternal`: Implement the method to save its contents by calling the methods of `DataOutput` for its primitive values or calling the `writeObject` method of `ObjectOutput` for objects, strings, and arrays.

# Serialization and Security

- ▶ Serialized objects contain all the data in a easily readable format so it isn't secure. The security can be dealt with in multiple ways.
  - ▶ Override the `readObject` and `writeObject` and implement our own algorithms for obscuring the data.
  - ▶ Encrypt and sign the entire object using `javax.crypto.SealedObject` and/or `java.security.SignedObject` wrapper. However, this requires managing symmetric keys.
  - ▶ For secure transport over the network, use SSL (Secure Sockets Layer) layer to encrypt the data. This requires minimal change in our code and is a widely used technique.

# Exercises

- ▶ Experiment with a class to see what changes do or don't modify the generated serial version UID?
- ▶ Write a loop that creates a large number of simple objects (a million or more) serializes them one by one. Note the time taken (use `java.lang.System.currentTimeMillis()`). Then write another loop that creates the same objects but now place them into a `Collection` like `ArrayList`, `List` or `HashMap`. Now serialize the collection instead of serializing the objects one by one. Note the time again. Is there a significant difference? Is so, why?
- ▶ Research how to use the `Kryo` serialization framework for Java. Convert the program from the previous exercise to use *Kryo* and then time the program again.
- ▶ Research how to use the `Protocol Buffers` serialization framework for Java. Convert the program from the previous exercise to use *Protocol Buffers* and then time the program again.

# References

- ▶ Other object serialization frameworks: Kryo, JSON, Google Protocol Buffers, Apache Avro, Facebook Thrift etc.
- ▶ [Serialization \(Wikipedia\)](#)
- ▶ [Performance comparison of various serialization frameworks](#)
- ▶ [More on warnings when reading back a serialized object](#)