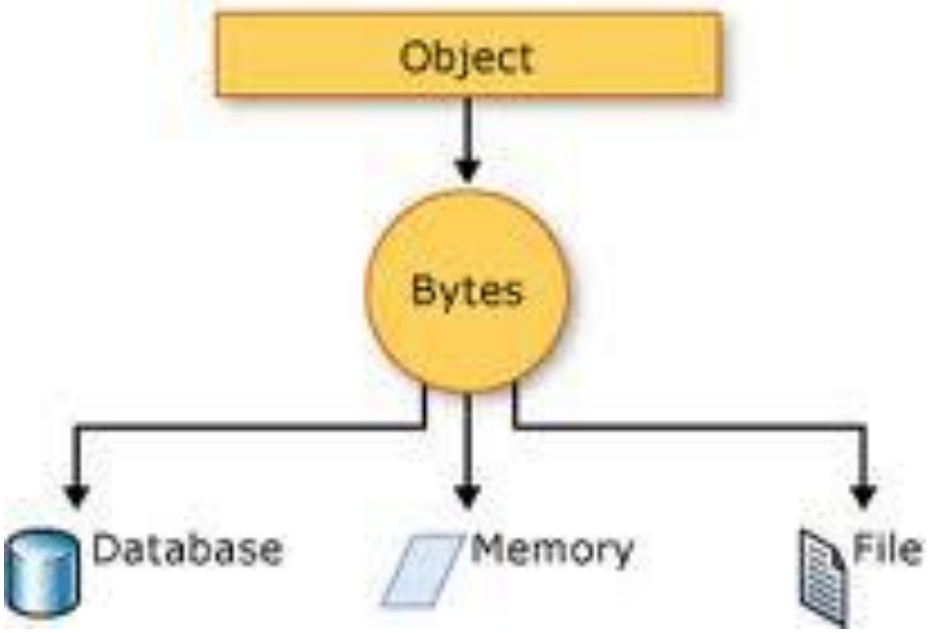
Serialization in Java



Object Serialization

We all know the Java platform allows us to create reusable objects in memory. However, all of those objects exist only as long as the Java virtual machine remains running. It would be nice if the objects we create could exist beyond the lifetime of the virtual machine, wouldn't it?

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time.

Serializable

Class is **serializable**

if

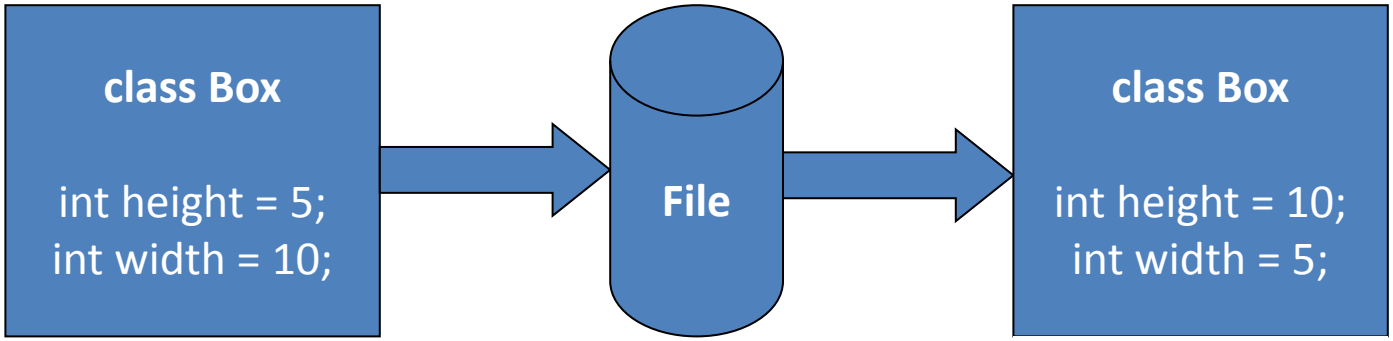
It can be transformed to array of bytes, and re-created from those bytes.

- By another class loader
- In another JVM
- On another computer

Usage

- RMI
- Enterprise Java Beans
- JMS
- Object Cache (disk storage)
- Application Server clustering
- …

Your own protocol



The Default Mechanism

To persist an object in Java, we must have a persistent object. An object is marked serializable by implementing the **java.io.Serializable** interface, which signifies to the underlying API that the object can be flattened into bytes and subsequently inflated in the future.

```
public interface Serializable {
}
```

Serializable is a *marker* interface; it has no methods to implement!

Working with ObjectOutputStream and ObjectInputStream

```
ObjectOutputStream.writeObject() // serialize and write
ObjectInputStream.readObject() // read and deserialize
```

Bare-bones example

```
import java.io.*;

class Cat implements Serializable { } // 1

public class SerializeCat {
  public static void main(String[] args) {
    Cat c = new Cat(); // 2
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(c); // 3
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      c = (Cat) ois.readObject(); // 4
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Object Graphs

What if the instance variables are themselves references to *objects*?

```
class Dog implements Serializable {
  private Collar theCollar;
  private int dogSize;
  public Dog(Collar collar, int size) {
    theCollar = collar;
    dogSize = size;
  }
  public Collar getCollar() { return theCollar; }
}
class Collar {
  private int collarSize;
  public Collar(int size) { collarSize = size; }
  public int getCollarSize() { return collarSize; }
}
```

What did we forget?

```
import java.io.*;
public class SerializeDog {
  public static void main(String[] args) {
    Collar c = new Collar(3);
    Dog d = new Dog(c, 8);
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

But when we run this code we get a runtime exception something like this:

```
java.io.NotSerializableException: Collar
```

The Collar class must ALSO be Serializable!

Good news!

Only objects marked **Serializable** can be persisted.

These classes are already serializable:
- Integer, Double, etc.
- String
- Date, Calendar
- ArrayList, LinkedList, HashSet, HashMap, etc.

Array of serializable objects is also serializable!

Transient

What if we didn't have access to the **Collar** class source code? In that case, can we ever persist objects of **Dog** type?

If you mark the Dog's Collar instance variable with **transient**, then serialization will simply skip the Collar during serialization:

```
class Dog implements Serializable {
  private transient Collar theCollar; // add transient
  // the rest of the class as before
}
```

Using writeObject and readObject

When the Dog is deserialized, it comes back with a **null** Collar.

```
class Dog implements Serializable {
  transient private Collar theCollar; // we can't serialize this
  private int dogSize;
  public Dog(Collar collar, int size) {
    theCollar = collar;
    dogSize = size;
  }
  public Collar getCollar() { return theCollar; }

  private void writeObject(ObjectOutputStream os) {
    try {
      os.defaultWriteObject(); // 1
      os.writeInt(theCollar.getCollarSize()); // 2
    } catch (Exception e) { e.printStackTrace(); }
  }
  private void readObject(ObjectInputStream is) {
    try {
      is.defaultReadObject(); // 3
      theCollar = new Collar(is.readInt()); // 4
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Serialization Is Not for Statics

You should think of static variables purely as CLASS variables.

Serialization applies only to OBJECTS.

Static variables are NEVER saved as part of the object's state…because they do not belong to the object!

Create Your Own Protocol: the Externalizable Interface

Instead of implementing the Serializable interface, you can implement **Externalizable**, which contains two methods:

```
public void writeExternal(ObjectOutput out) throws IOException;
public void readExternal(ObjectInput in) throws IOException,
                                              ClassNotFoundException;
```

Just override those methods to provide your own protocol. Although it's the more difficult scenario, it's also the most controllable. An example situation for that alternate type of serialization: read and write PDF files with a Java application.

Externalizable

```
package java.io;
public interface Externalizable extends java.io.Serializable
{
    void writeExternal(ObjectOutput out)
        throws IOException;

    void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException;
}
```

serialVersionID

```
[private] static final long serialVersionUID = 42L;
```

- Used when deserialization for deciding if the serialized and loaded classes are compatible
- If not present, JVM computes this value automatically

Links

- Java Object Serialization Specification
- Implementing Serializable: best practices