# CENG 443
# Introduction to Object-Oriented Programming Languages and Systems

# Serialization

# Serialization

- Ability to read or write an object to a stream
  - Process of "flattening" an object
- Used to save object to some permanent storage
  - Its state should be written in a serialized form to a file such that the object can be reconstructed at a later time from that file
- Used to pass on to another object via the *OutputStream* class
  - Can be sent over the network
- ObjectOutputStream
  - For serializing (flattening an object)
- ObjectInputStream
  - For deserializing (reconstructing an object)

# Requirement for Serialization

- To allow an object to be serializable:
  - Its class should implement the Serializable interface
    - Serializable interface is marker interface
  - Its class should also provide a default constructor
- Serializability is inherited
  - Don't have to implement Serializable on every class
  - Can just implement Serializable once along the class hierarchy

# Non-Serializable Objects

- Most Java classes are serializable
- Objects of some system-level classes are not serializable
  - Because the data they represent constantly changes
    - Reconstructed object will contain different value anyway
    - For example, thread running in my JVM would be using my system's memory. Persisting it and trying to run it in your JVM would make no sense at all.
- A NotSerializableException is thrown if you try to serialize non-serializable objects

# What can be serialized?

- Enough information that is needed to reconstruct the object instance at a later time
  - Only the object's data are preserved
  - Methods and constructors are not part of the serialized stream
  - Class information is included

# transient

- How do you serialize an object of a class that contains a non-serializable class as a field?
  - Like a Thread object
- What about a field that you don't want to to serialize?
  - Some fields that you want to recreate anyway
  - Performance reason
- You mark them as *transient*
  - The *transient* keyword prevents the data from being serialized
  - Serialization does not care about access modifiers such as *private* -- all nontransient fields are considered part of an object's persistent state and are eligible for persistence

# Example

```
1  class MyClass implements Serializable {
2
3      // Skip serialization of the transient field
4      transient Thread thread;
5      transient String fieldIdontwantSerialization;
6
7      // Serialize the rest of the fields
8      int data;
9      String x;
10
11     // More code
12 }
```

# Serialization: Writing an Object Stream

**public final void writeObject(Object obj) throws IOException**

where, *obj* is the object to be written to the stream

```
1  import java.io.*;
2  public class SerializeBoolean {
3     SerializeBoolean() {
4        Boolean booleanData = new Boolean("true");
5        try {
6           FileOutputStream fos = new
7                       FileOutputStream("boolean.ser");
8           ObjectOutputStream oos = new
9                       ObjectOutputStream(fos);
10          oos.writeObject(booleanData);
11          oos.close();

13       } catch (IOException ie) {
14          ie.printStackTrace();
15       }
16    }
17
18    public static void main(String args[]) {
19       SerializeBoolean sb = new SerializeBoolean();
20    }
21 }
```

# Deserialization: Reading an Object Stream

- **public final Object readObject() throws IOException, ClassNotFoundException**

  where, obj is the object to be read from the stream
- The Object type returned should be typecasted to the appropriate class name before methods on that class can be executed.

```
public static void main(String args[]) {
    UnserializeBoolean usb =
                        new UnserializeBoolean();
 }
```

```
1  import java.io.*;
2  public class UnserializeBoolean {
3      UnserializeBoolean() {
4        Boolean booleanData = null;
5        try {
6           FileInputStream fis = new
7                        FileInputStream("boolean.ser");
8           ObjectInputStream ois = new
9                        ObjectInputStream(fis);
10          booleanData = (Boolean) ois.readObject();
11          ois.close();
13       } catch (Exception e) {
14          e.printStackTrace();
15       }
16       System.out.println("Unserialized Boolean from "
17                           + "boolean.ser");
18       System.out.println("Boolean data: " +
19                           booleanData);
20       System.out.println("Compare data with true: " +
21             booleanData.equals(new Boolean("true")));
22    }
```

# Version Control: Problem Scenario

- Imagine you create a class, instantiate it, and write it out to an object stream
- That flattened object sits in the file system for some time
- Meanwhile, you update the class file, perhaps adding a new field
- What happens when you try to read in the flattened object?
- An exception will be thrown -- *java.io.InvalidClassException*
- *Why?*
  - Because all persistent-capable classes are automatically given a unique identifier
  - If the identifier of the class does not equal the identifier of the flattened object, the exception will be thrown

# Version Control: Problem Scenario

- Why should it be thrown just because I added a field? Couldn't the field just be set to its default value and then written out next time?
- Yes, but it takes a little code manipulation. The identifier that is part of all classes is maintained in a field called serialVersionUID.
- If you wish to control versioning, you simply have to provide the *serialVersionUID* field manually and ensure it is always the same, no matter what changes you make to the classfile.

# How Do I generate a Unique ID?

- *serialver* utility is used to generate a unique ID
- *Example*
- *serialver MyClass*
- *MyClass static final long serialVersionUID = 1027553947283749 5L;*

# own readObject() and writeObject() methods

- Used when the default behavior of *readObject*() and *writeObject*() are not sufficient
- You provide your own readObject() and writeObject() in order to add custom behavior
- Example

```
  // Provide your own readObject method
private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {

    // our "pseudo-constructor"
    in.defaultReadObject();
    // now we are a "live" object again, so let's run rebuild and start
    startAnimation();

}
```

# Externalizable Interface

- The default java serialization is not efficient.
- To solve this issue, you can write your own serialization logic by implementing Externalizable interface and overriding it's methods *writeExternal* and *readExternal*. By implementing these methods, you are telling the JVM how to encode/decode your object.
- The *writeExternal* and *readExternal* methods of the *Externalizable* interface can be implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes
- These methods must explicitly coordinate with the supertype to save its state
- These methods supersede customized implementations of writeObject and readObject methods

# Object Serialization with Externalizable

- Object Serialization uses the Serializable and Externalizable interfaces
- Each object to be stored is tested for the Externalizable interface
- If the object supports Externalizable, the writeExternal method is called
- If the object does not support Externalizable and does implement Serializable, the object is saved using ObjectOutputStream.

# Example

```
class UserSettings implements Externalizable {
    //This is required
    public UserSettings(){ }

    private String doNotStoreMe;

    private Integer fieldOne;
    private String fieldTwo;
    private boolean fieldThree;

    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        fieldOne = in.readInt();
        fieldTwo = in.readUTF();
        fieldThree = in.readBoolean();
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        //We are not storing the field 'doNotStoreMe'
        out.writeInt(fieldOne);
        out.writeUTF(fieldTwo);
        out.writeBoolean(fieldThree);
    }
}
```

```java
private static void storeUserSettings(UserSettings settings)
{
    try {
        FileOutputStream fos = new FileOutputStream("object.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(settings);
        oos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```java
private static UserSettings loadSettings() {
    try {
        FileInputStream fis = new FileInputStream("object.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        UserSettings settings =  (UserSettings) ois.readObject();
        ois.close();
        return settings;
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
    return null;
}
```

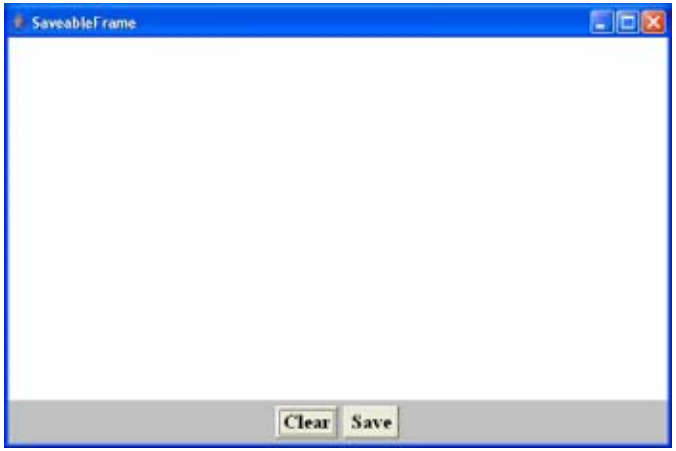
# Example

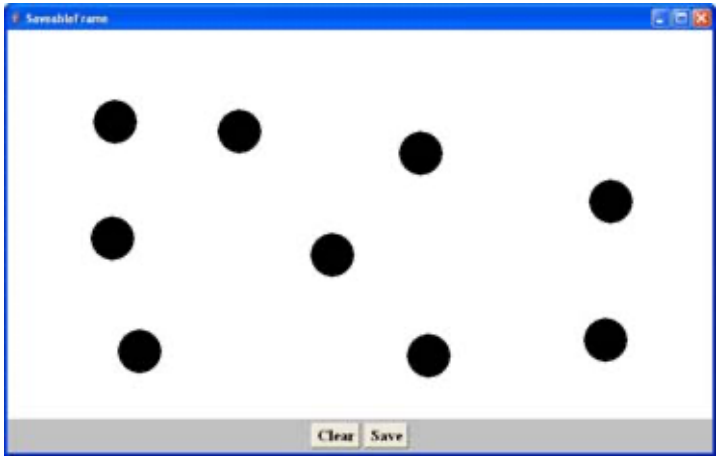
## Saving to File

- Open frame (600x400, no circles, top left corner)
- Move window around
- Resize it
- Click to add circles
- Press “Save”

## Next time program runs

- Frame pops up at previous location, with previous size, including previous circles



# Use Cases

- **Stashing**
  - Rather holding a large object in memory, it's better to cache it to a local file via serialization.
- **Data transmission**
  - Java permits to serialize an object over a network using RMI (Remote Method Invocation), a distributed technology of Java. RMI enables a Java client object communicates with the instance of a Java server hosted on a remote system. For example, an ATM center of your locality can interact with a bank server located in a different country.
- **Persistence**
  - If you want to preserve the state of a particular operation to a database, just serialize it to a byte array, and save to the database for later use.
- **Deep cloning**
  - In Java, it is also known as the deep copy. It causes an object to copy along with the objects to which it refers. You need to write a customized clone class to achieve this. Java serialization can save you the trouble of adding a clone class. Serializing the object into a byte array and then deserializing it to another object will fulfill the purpose.
- **Cross JVM communication.**
  - Serialization works the same across different JVMs irrespective of the architectures they are running on.

# Effective Java Items

- Item 85: Prefer alternatives to Java serialization
  - Without using any gadgets, you can easily mount a denial-of-service attack by causing the deserialization of a short stream that requires a long time to deserialize. Such streams are known as *deserialization bombs*
- Item86: Implement Serializable with great caution
  - A major cost of implementing Serializable is that it decreases the flexibility to change a class's implementation once it has been released.

# Example

```
// Deserialization bomb - deserializing this stream takes forever
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // Make t1 unequal to t2
        s1.add(t1);  s1.add(t2);
        s2.add(t1);  s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // Method omitted for brevity
}
```

The object graph consists of 201 HashSet instances, each of which contains 3 or fewer object references. The entire stream is 5,744 bytes long

The problem is that deserializing a HashSet instance requires computing the hash codes of its elements. The 2 elements of the root hash set are themselves hash sets containing 2 hash-set elements, each of which contains 2 hash-set elements, and so on, 100 levels deep. Therefore, deserializing the set causes the hashCode method to be invoked over 2^100 times.