Basics of programming 3

Java serialization

Serialization basics

- Problem
  - let's save our objects and later read them back
  - what should be stored?
    - objects' attributes
    - associations
    - static fields?
- Simple solution: *serialization*
  - built-in Java feature
  - converts objects' data and their associations to and from byte-streams

Serialization concepts

- Serialization
  - converting objects into binary form (byte stream)
    - also called marshalling, deflating
  - binary data can be stored, transmitted, etc.
- Deserialization
  - converting binary data (byte stream) into objects
    - also called unmarshalling, inflating
  - binary data can be read from a file, got from a network connection, etc.

Serialization example: write

```
import java.io.Serializable;
public class SerializableClass implements Serializable
{ ... }
```

```
//import java.io.*;
...
SerializableClass ser = ...;
try {
        FileOutputStream f =
                        new FileOutputStream(”filename”);
        ObjectOutputStream out =
                        new ObjectOutputStream(f);
        out.writeObject(ser);
        out.close();
}
catch(IOException ex) { ... }
```

Serialization example: read

```
//import java.io.*;
...
SerializableClass ser2;
try {
        FileInputStream f =
                        new FileInputStream("filename");
        ObjectInputStream in =
                        new ObjectInputStream(f);
        ser2 = (SerializableClass)in.readObject();

        in.close();

} catch(IOException ex) {
        ...
} catch(ClassNotFoundException ex) {
        ...
}
```

Rules of serialization

- Only classes implementing interface *Serializable* can be serialized
  - if superclass implements, it's OK
  - arrays, String, Integer, Double, etc. OK
- Interface *Serializable*
  - no methods
  - just formal notification for the compiler
- *Not serializable*
  - *Object, Socket, Input/OutputStream, System, etc.*

Rules of serialization

- What is serialized?
  - primitive attributes
  - serializable attributes
    - recursively
- What is not serialized?
  - static fields
  - *transient* fields

```
public class Serial implements Serializable {
        transient private String secret;
        private String other;
        ...
```

Serialization process

- Serialization is recursive
  - how to avoid cyclic graphs?
  - every objects is written only once totally
  - consecutively only reference is written

```
out.writeObject(a);
a.modify(...);
out.writeObject(a); // only reference!
```

- Serialization of inherited members
  - starts from topmost serializable superclass

Writing own serialization

- Use

```
private void writeObject(ObjectOutputStream out)
        throws IOException
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
```

- Super/subclass data handled automatically
- Default implementation
  - `out.defaultWriteObject(), in.defaultReadObject()`
- *out* and *in* have helper methods
  - reading/writing primitive and serializable types

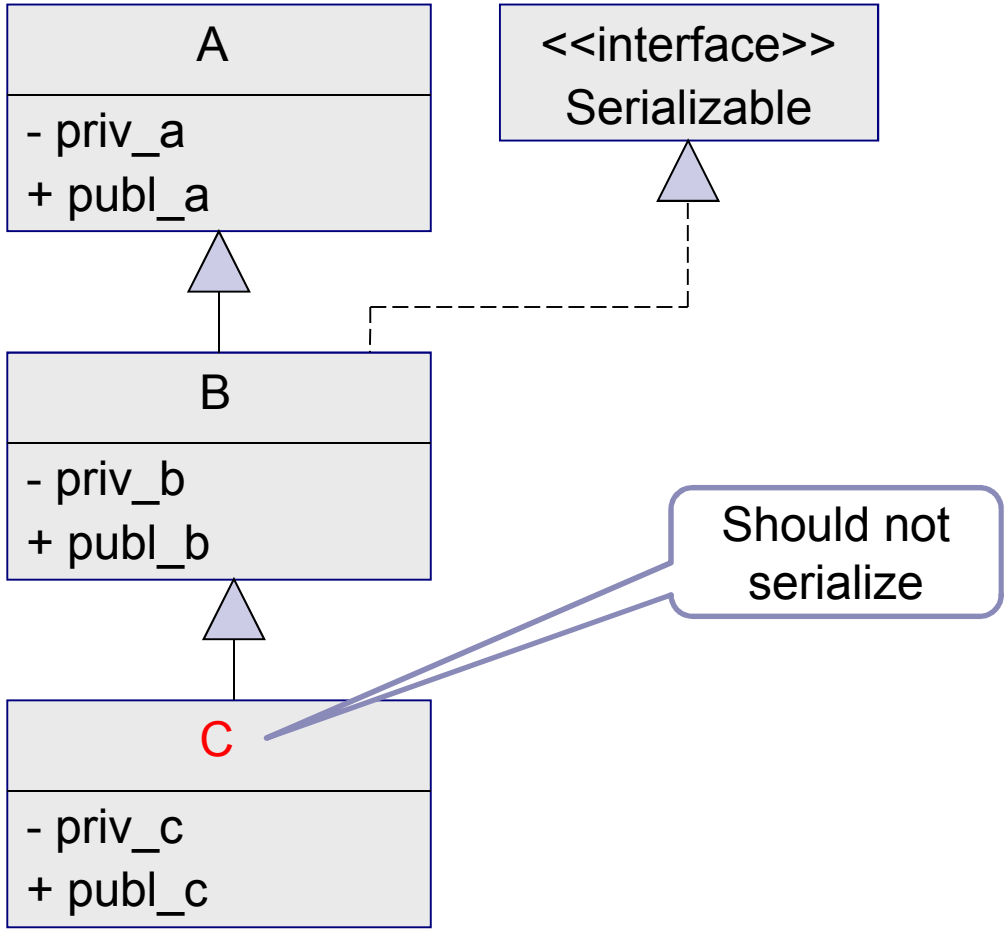
Writing own serialization 2

- For total control implement

```
interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
    throws IOException;
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
}
```

- Handle super/subclass data explicitly

Stopping serialization



Stopping serialization

```
private void writeObject(ObjectOutputStream out)
throws IOException {

      throw new NotSerializableException("C");
}


private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException {

      throw new NotSerializableException("C");
}
```

Versioning

- Each class has a unique version ID
  - showing it: `serialver ClassName`
- Ensuring compatibility
  - use same version ID

```
static final long serialVersionUID
 = 1027553947283749 5L;
```

- Compatible changes
  - method/field add/remove or access modification
  - static/transient → persistent
  - class hierarchy change

Java IO Specialties

java.io.File

- Meta-information about files and directories
  - trivial constructors
    - from String, other File objects
    - with path and filename
  - OS dependent info
    - path separator, directory separator
  - file operations
    - delete, create tmp file, access right modification
  - information
    - exists, name, parent, access rights, type, length, content

File example

- List recursively current directory

```
void lsdir(File f, String tab) {

   File[] list = f.listFiles();

   for (int i = 0; i < list.length; i++) {
      System.out.println(tab+list[i].getName());

      if (list[i].isDirectory()) {
         lsdir(list[i], tab+" ");
      }

   }
}
```

Piped streams

- Filtering is not always convenient
  - e.g. implement unix grep command
    - read lines, print those matching an expression
    - String.matches helps
- PipedReader, PipedWriter

  PipedInputStream, PipedOutputStream
  - two objects are connected
  - what's written into the writer, can be read from the reader

Grep implementation

```
public class Grep {
	Reader in;
	Writer out;
	String pattern;

	public Grep(Reader in, Writer out, String pat) {
		this.in = in;
		this.out = out;
		this.pattern = pat;
	}
...
```

Grep implementation

```
...
        public void run() {
                BufferedReader br = new BufferedReader(in);
                PrintWriter pw = new PrintWriter(out);
                while (true) {
                        String line = br.readLine();
                        if (line != null) {
                                if (line.matches(pattern)) {
                                        pw.println(line);
                                }
                        } else break;
                }
                pw.close();
        }
...
```

Grep implementation

```
...
        static public Reader
        grep(Reader r, String pattern) {
                PipedWriter pw = new PipedWriter();
                PipedReader pr = new PipedReader(pw);
                Grep grep = new Grep(r, pw, “hello”);
                // starting grep in the background
                // must actively handle both endpoints
                grep.run();
                return pr;
        }
}
```