SWEN221:

Software Development

14: Serialisation and Cloning

David J. Pearce & Nicholas Cameron & James Noble & Petra Malik
Engineering and Computer Science, Victoria University

Java.lang.Object.clone()

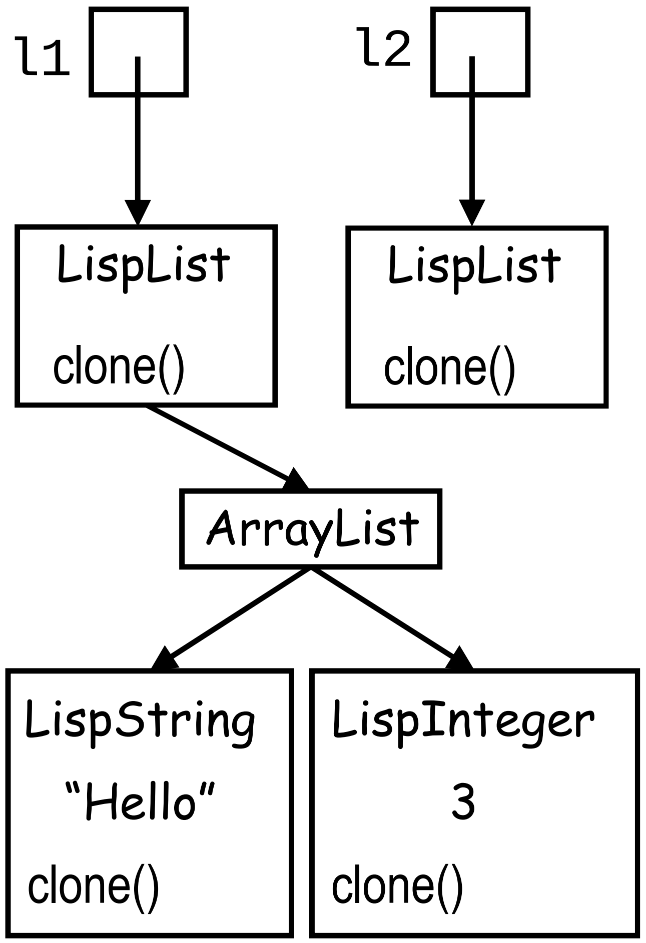
- Purpose is to create copy of object:

```
LispExpr e1 = new LispInteger(1);
LispExpr e2 = e1.clone();
// e1 != e2
// but, e1.equals(e2) must hold and
// e1.getClass() == e2.getClass() must hold
```

- Object.clone() provides default implementation
  - Is **`protected`** so must be explicitly overridden
  - Bitwise copy of all members, including those in subclass

Example clone() implementation

```
class LispList implements Cloneable {
 private List<LispExpr> elements =
              new ArrayList<LispExpr>();
 …
 public Object clone() {
  try { return super.clone(); }
  catch(CloneNotSupportedException e) {
    return null; // cannot get here
}}}
LispInteger i = new LispInteger(3);
LispString s = new LispString(“Hello”);
LispList l1 = new LispList();
l1.add(i);
l1.add(s);
LispList l2 = (LispList) l1.clone();
```



- What does this actually do?

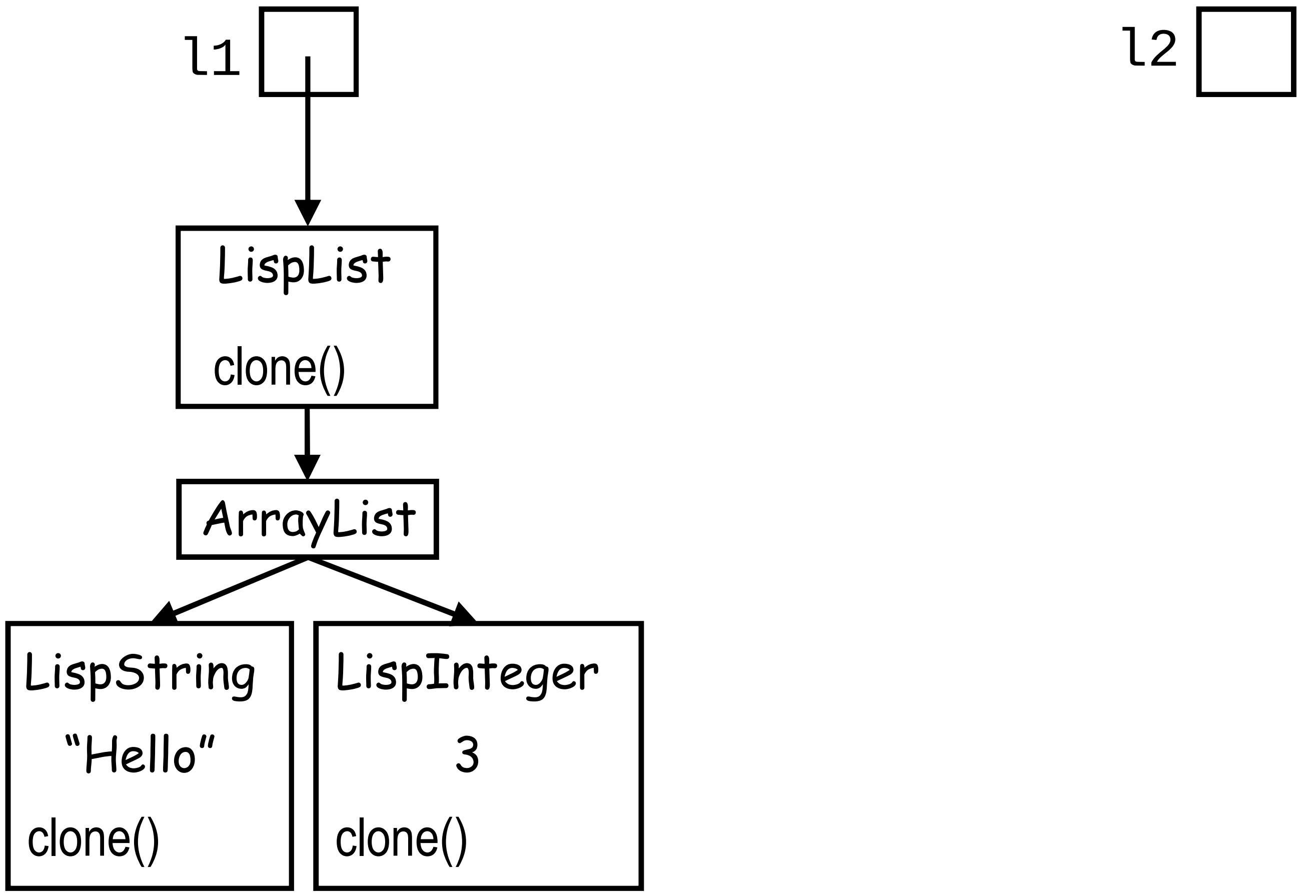
Deep Clone

- This version of clone gives a **deep copy***:*
  - (i.e. all children recursively cloned)

```
class LispList implements Cloneable {
 private List<LispExpr> elements =
              new ArrayList<ListExpr>();
 …
 public LispList clone() {
   LispList ne = new LispList();
   for(LispExpr e : elements) {
    ne.add(e.clone());
   }
   return ne;
}}
```

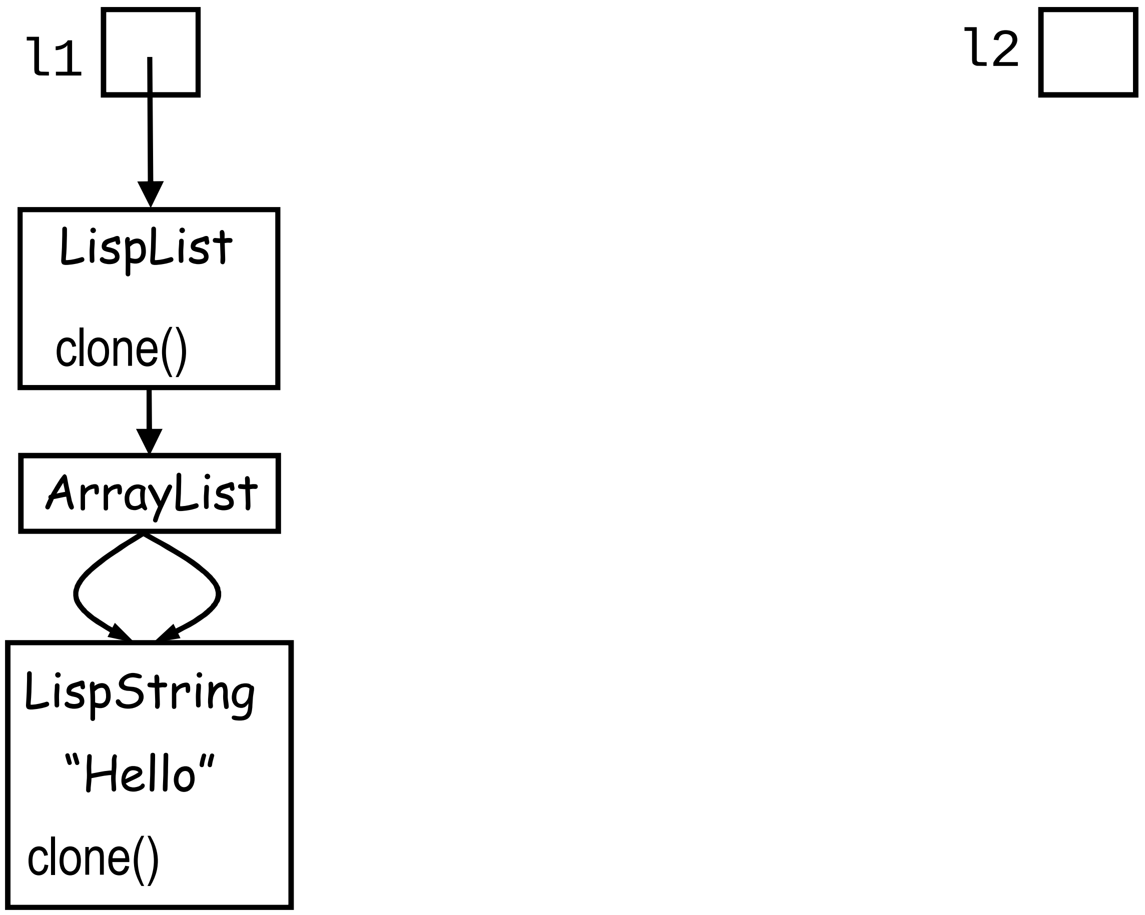
Deep Clone

```
l2 = l1.clone();
```



Deep Clone --- What Happens?

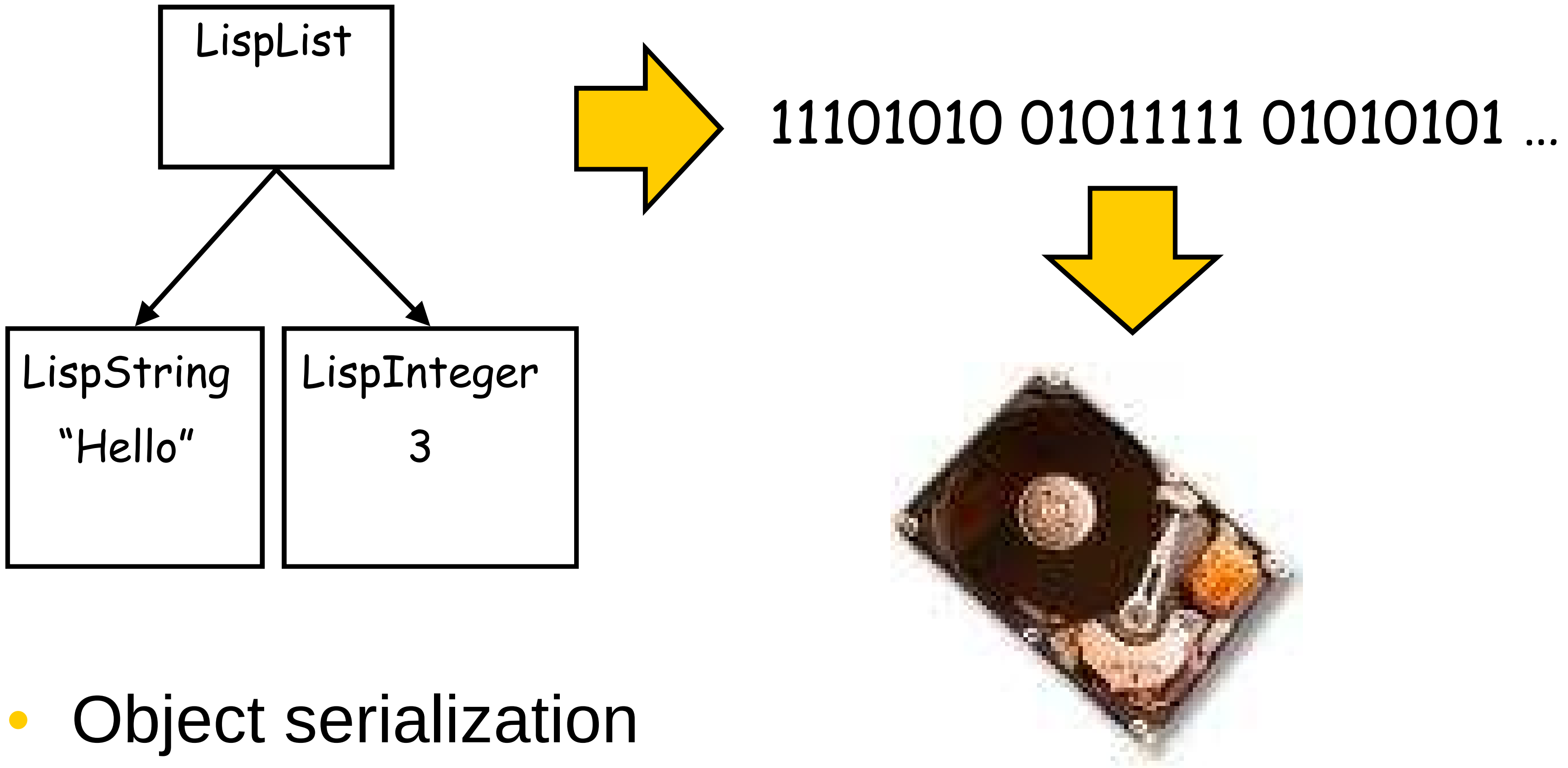
```
l2 = l1.clone();
```



Few last points on Cloning

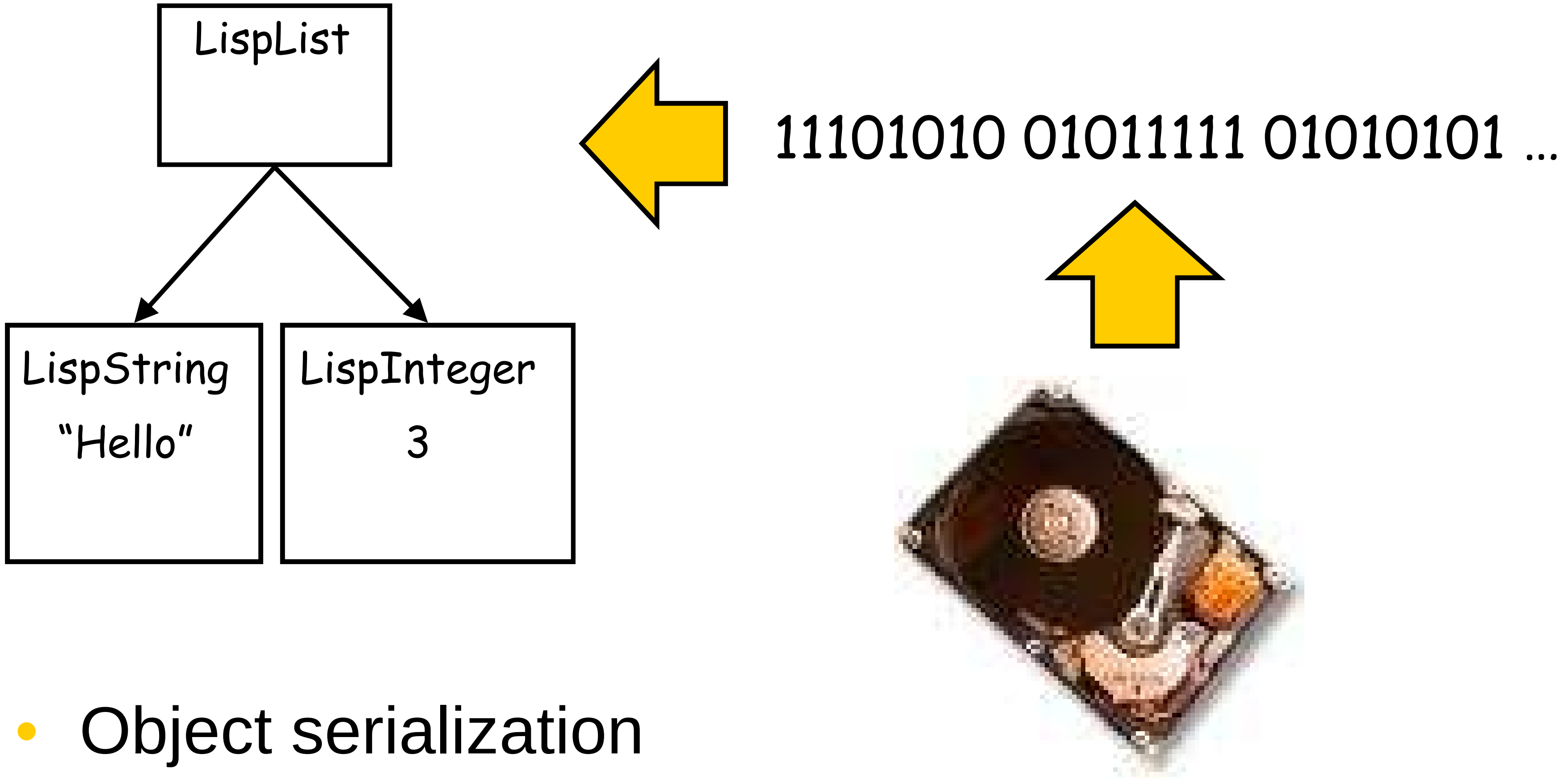
- Don’t need to clone immutable types!
  - E.g. Integer, String etc.
  - Why?
- Arrays & Collections
  - clone() is *shallow* – beware!!
- Use super.clone()
- Which to use: deep or shallow copy?
  - Depends upon the situation
  - Always at least clone hidden state
  - One solution is to do both!
    - E.g. by adding a deepClone() method

Serialization



- Object serialization
  - Process of converting object into byte sequence
  - Can write sequence to file and…

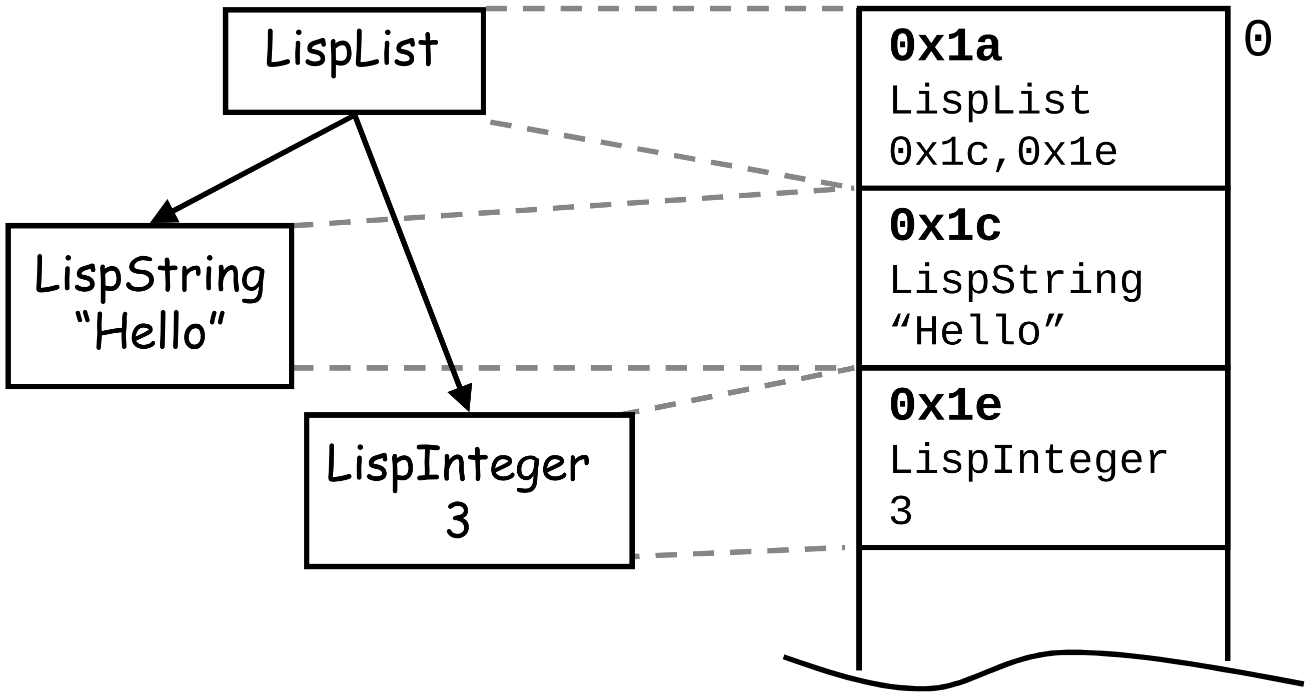
Serialization



- Object serialization
  - Process of converting object into byte sequence
  - Can write sequence to file and reload it later!

How does seralization work?

- Consider the following object graph:
  - References are turned into *handles*
  - Primitives (e.g. int) stored in **platform-neutral format**
    - So, can be loaded into machine with different architecture



Using Serialization

- To serialize an object:
  - Class must implement **Serializable**
    - Like Cloneable, is a marker interface
    - Does not require any methods be implemented
    - If interface not implemented => NotSerializableException
  - Serialization mechanism uses **deep copy**
    - Otherwise, what to do with references?
    - Cannot use memory address as want platform neutrality
  - Fields marked **transient** are not serialized
    - Useful for classes which can’t be serialized (e.g. Thread)

Example Code

```
interface LispExpr extends Serializable { … }
class LispInteger implements LispExpr { … }
class LispString implements LispExpr { … }
class LispList implements LispExpr { … }

LispList l1 = new LispList();
l1.add(new LispString(“Hello”));
l1.add(new LispInteger(3));

// write objects to file “expr.dat”
FileOutputStream fout = new FileOutputStream(“expr.dat”);
ObjectOutputStream out = new ObjectOutputStream(fout);
out.writeObject(l1);
out.close();

// now, read objects back
FileInputStream fin = new FileInputStream(“expr.dat”);
ObjectInputStream in = new ObjectInputStream(fin);

LispList l2 = (LispList) in.readObject(); // deep-copy of l1
```

Serialization Pitfalls – Versioning

- Scenario:
  1. Object X instance of class Y
  2. Write X to file “X.dat”
  3. Change class Y (e.g. add field)
  4. Read “X.dat” back into program

- Will raise InvalidCastException!
  - Class given unique ID based on implementation
  - Modified class has different ID

- Versioning
  - Define value for `serialVersionUID`
  - If modification **compatible** leave `serialVersionUID` as is
  - If change **incompatible** increment `serialVersionUID`

Serialization Pitfalls – Caching

- Problem:

```
Customer o = new Customer("Dave");
o.setAddress("3 Kelburn Parade");
out.writeObject(o);
o.setAddress("122 Upland Road");
out.writeObject(o);
out.close();
```

- Only one copy of "o" written to stream
  - OutputObjectStream caches objects
    - So subsequent writeObject() calls share handles
  - Can use `ObjectOutputStream.reset()`
    - Causes it to flush object cache
    - Might cause object to be written more than once!?