

# Objects and Serialization

## Equals

### boolean equals(Object other)

vs ==

For objects, `a == b` tests if `a` and `b` are the same pointer  
"shallow" semantics

`boolean Object.equals(Object other)`

defined up in the `Object` class, does a `(this == other)` test -- still shallow semantics

Override

A class may override `equals()` to provide "deep" comparison semantics -- do the two  
objects represent the same state?

e.g. `String` overrides `equals()`

### Calling equals()

```
{  
    String a = "hello";  
    String b = "hello";  
  
    (a == b) ==> false  
    (a.equals(b)) ==> true  
}
```

### equals() strategy

`boolean equals(Object other) { ...`

Take `Object` argument, return `boolean` -- must have the exact same prototype as the  
version up in `Object` for overriding to work.

Return `true` on `(this == other)`

Use `(other instanceof Foo)` to test class of `other` -- `false` if not same class (`instanceof`  
returns `false` on null ptrs)

Otherwise do a field by field comparison of `this` and `other`

### Student equals()

// in `Student` class...

```
boolean equals(Object obj) {  
    if (obj == this) return(true);  
    if (!(obj instanceof Student)) return(false);  
    Student other = (Student)obj;  
    return(other.units == units)  
}
```

# Clone

Goal: create a "copy" of an object

Given "foo" obj of class Foo, copy = foo.clone().

Copy has same state as foo, but its own memory. Probably foo.equals(copy)

## Cloneable interface

Used as a marker that the class implements the clone() message

Not compiler enforced

Object.clone() is pre-built to (a) create a new instance of the right class, and (b) assign all the fields over with '=' semantics.

Object.clone() gives this default behavior if the class implements the Cloneable interface.

Otherwise it throws an exception.

## Implementing Clone

Implement the Cloneable interface

1. copy = (Class) super.clone() first (must use try/catch in case the clone() fails)
2. copy the fields where a simple '=' is not deep enough

## Alternatives

Copy Constructor -- Foo(Foo x) -- construct a new instance of Foo, based on the state of an existing foo.

"Factory" method -- static method that makes new instances. May use ctors internally...

static Foo Foo.newInstance(Foo x)

Advantage: simpler than clone(). Does not require extra concepts, since we already understand ctors.

Disadvantage: the client must know the class of the object. In contrast, a client can send x.clone() and get a copy without knowing the precise class of x.

## Eq Code Example

```
// Eq.java
/*
 * Demonstrates a simple class that defines equals and clone.
 */
public class Eq implements Cloneable {
    private int a;
    private int[] values;

    public Eq(int init) {
        a = init;
        values = new int[10];
    }

    /*
     * Does a "deep" compare of this vs. the other object.
     */
    public boolean equals(Object other) {
        if (other == this) return(true);
        if (!(other instanceof Eq)) return(false);

        Eq e = (Eq) other;
```

```

    // now test if this vs. e
    if (a != e.a) return(false);

    if (values.length != e.values.length) return(false);
    for (int i=0; i<values.length; i++) {
        if (values[i] != e.values[i]) return(false);
    }
    return(true);
}

/*
Returns a deep copy of the object.
*/
public Object clone() {
    try {
        // first, this creates the new memory and does '=' on all fields
        Eq copy = (Eq)super.clone();

        // copy the array over -- arrays respond to clone() themselves
        copy.values = (int[]) values.clone();
        return(copy);
    }
    catch (CloneNotSupportedException e) {
        return(null);
    }
}

public static void main(String[] args) {
    Eq x = new Eq(1);
    Eq y = new Eq(2);
    Eq z = (Eq) x.clone();

    System.out.println("x == z" + (x==z)); // false
    System.out.println("x.equals(z)" + (x.equals(z))); // true
}
}

```

## Serializing

Boring object <-> file problem

Serialization -- somewhat automatic

Serializable interface

To write out an object

```

ObjectOutputStream out;
out.writeObject(obj);

```

To read that object back in

```

ObjectInputStream in;
obj = in.readObject();

```

Same type

When reading, cast back to what it was when written

## Serialization / Archiving

State in memory -- objects

Write objects to streamed state

To a disk file, or across the network, or to the system clipboard  
 The notion of "address space" does not hold in the streamed form -- there are no pointers.

Read

Read the streamed form, and re-create the object in memory

Synonyms

Flattening

Streaming

Dehydrate (Rehydrate = read)

Archiving

## 106a Memory<->Disk

Translate back and forth by hand

Typically use an ASCII text format

Custom arrangement between your data structures and some ASCII format for reading and writing.

## Java Automatic Serialization

**Serializable** interface

By implementing this interface, a class declares that it is willing to be read/written by the automatic serialization machinery.

Automatic Writing

Automatically writes out fields inside an object

The system knows how to recursively write out the state of an object

Recursively follows pointers and writes those objects out too

Built-Ins

Most built ins know how to serialize: int, array, Point, ...

"transient" fields -- do not serialize

Use to prevent the serialization from recurring down a branch you do not want written to disk. Comes back as null after reading.

Override: readObject(), writeObject() -- to put in more customized reading/writing

Versioning

serialization can detect version changes when reading and refuse to read if the code for a written out object has changed since it was written out. Programmer can control this.

## ObjectOutputStream out;

## out.writeObject(obj)

This one line calls the automatic serialization machinery to write out everything rooted at the given object.

Classes

Each written object will be identified by its class -- the reading code will need those same classes to read the stream.

Array

For a collection of things, it may be easier to cast the whole thing into a single array that can be written in one operation.

#### Transient

Fields should be declared transient if they should not be written. They will read back in as null.

## ObjectInputStream in;

### in.readObject()

#### CT type

Read back with the same CT type it was written (Object[] or DShapeModel[])

#### Class

If a class was written that is not present at read-time, there will be an error.

If the class has the same name but a changed implementation there will be an error.

It's safest to serialize classes that are stable everywhere such as Array and Point

Do not change the structure of DShapeModel, or you will not be able to open old files

## Circularity : Solved

The automatic machinery takes care of the case where the pointer graph of objects being written out has pointers in to itself. The read operation correctly re-creates the pointer graph in memory. (yay!)

## Dots example...

Our earlier Dots example had an ArrayList of Point objects.

Here's how it would write itself out...

```
/**
 * Given a file, write the data model to it with Java serialization.
 * Makes an Point[] array of points and writes it
 * which avoids the bother of iteration.
 * (We use an array instead of the ArrayList to avoid
 * requiring a 1.2 VM to read the file, although maybe
 * the ArrayList would have been fine.)
 */
public void saveSerial(File file) {
    try {
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream(file));

        // Use the standard collection -> array util
        // (the Point[0] tells it what type of array to return)
        Point[] points = (Point[]) dots.toArray(new Point[0]);

        out.writeObject(points);    // serialization!

        out.close();    // polite to close on the way out
        setDirty(false);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

```

    }
}

/**
 * Inverse of saveSerial.
 * Reads an Point[] array of Points, and adds
 * them to our data model.
 */
private void loadSerial(File file) {
    try {
        ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));

        // Read in the object -- the CT type should be exactly as it was written
        // -- Point[] in this case.
        // Transient fields would be null.
        Point[] points = (Point[])in.readObject();

        for (int i=0; i<points.length; i++) {
            dots.add(points[i]);
        }

        in.close(); // polite to close on the way out
        setDirty(false);
    }

    catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

## 193j Classes

For hw2, we have wrapper classes that shield you from the exceptions, but otherwise behave like `ObjectOutputStream` and `ObjectInputStream`

`SimpleObjectWriter w;`

```

SimpleObjectWriter w = SimpleObjectWriter.openFileForWriting(filename);
w.writeObject( <object>) -- write an array or object (Point[] in above example)
w.close()

```

`SimpleObjectReader r;`

```

SimpleObjectReader r = SimpleObjectReader.openFileForReading(filename);
obj = r.readObject() -- returns the object written -- cast to what it is (Point [] in above
example)
r.close()

```