Cloning and Proxy

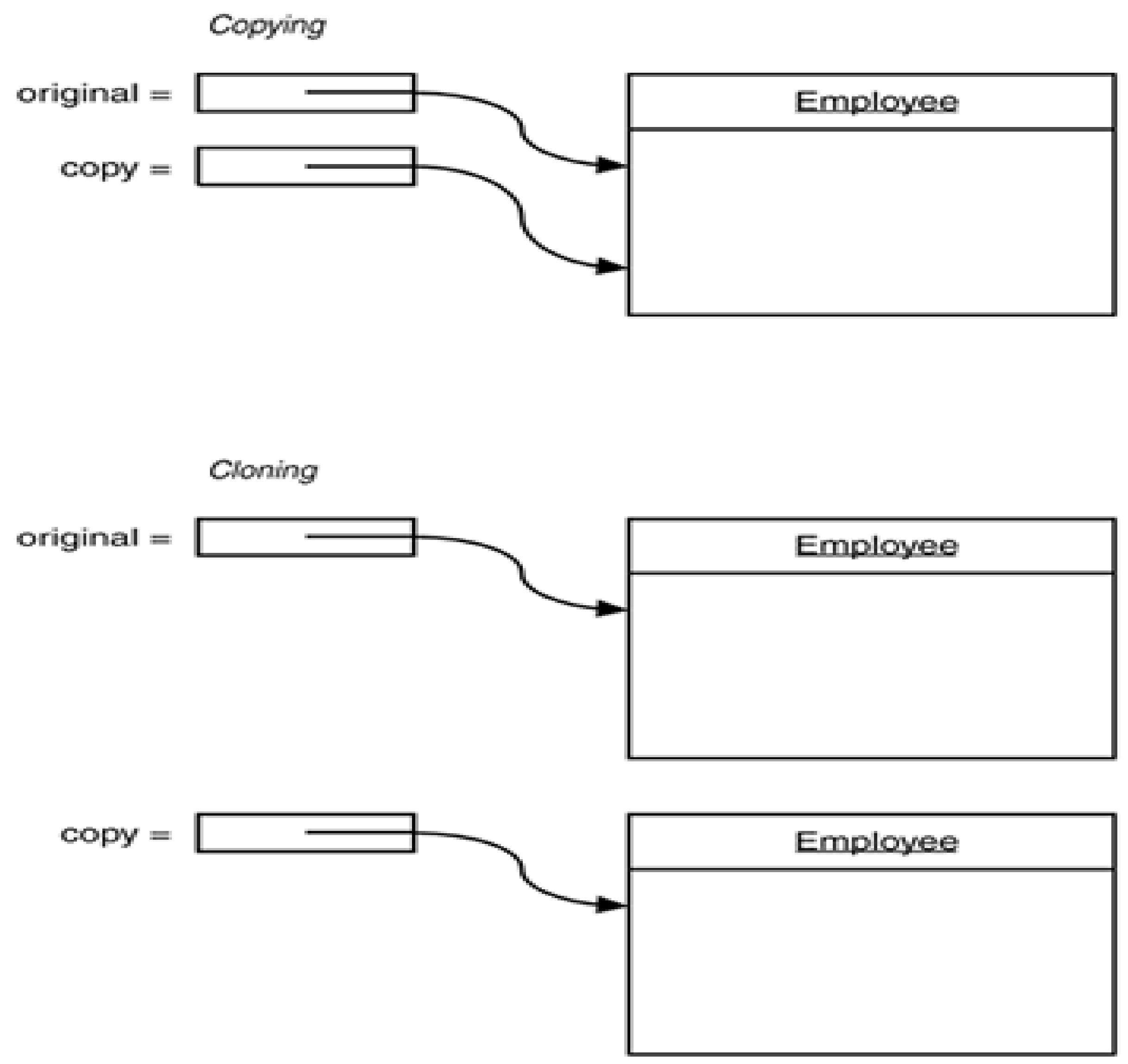
Object Cloning

- Making a copy of an object variable just makes a second reference to that object, it doesn't "copy" it.

```
Person somePerson=new Person("Student1");
Person otherPerson;
otherPerson=somePerson;  //both otherPerson and somePerson point to
                         // the same object.  A change in either will
                         //occur in both.
```

- This just creates two references to the same exact thing. This isn't very useful.

Con’t


Copying
original =
copy =
Employee
Cloning
original =
Employee
copy =
Employee

Object Cloning

- Some Objects can be cloned – another Object of the same type is created, which starts in the same state, but is actually an entirely different Object.
- Some of the Java predefined types have this behavior built-in (the Date type, for instance)

An Example

- In this example, both Date references point to the same Object. A change in one will effect both

```
Date someDate=new Date(1234567);
Date newDate=someDate;
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime());   //outputs 1234568
System.out.println(newDate.getTime());    //outputs 1234568
```

- In this example, we create a full new object that starts in the same state. Now they are two unique objects

```
Date someDate=new Date(1234567);
Date newDate=(Date)someDate.clone();
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime());   //outputs 1234567
System.out.println(newDate.getTime());    //outputs 1234568
```

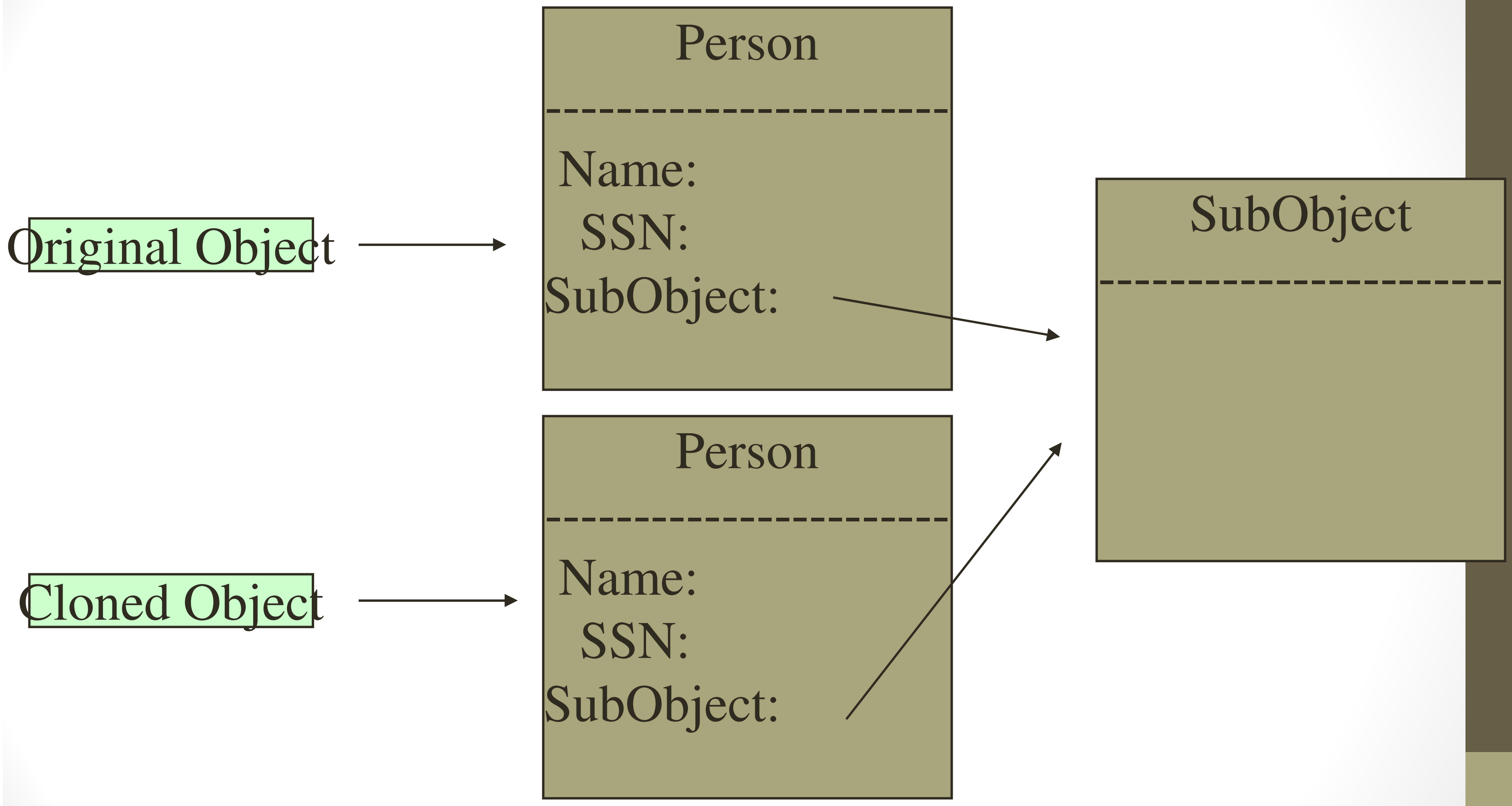
The Clone Method

- This works with Date objects because Date objects implement the Cloneable interface.
- The clone() method is actually inherited from the Object superclass, and is protected by default.
- Since clone() is already implemented in Object, it means that the Cloneable interface is a *tagged interface* – it actually contains no methods.
- Since clone() is inherited from Object, it doesn’t know anything about it’s subclasses (which is every class in Java).
- Is this a problem?

The Clone Method

- Since the clone() method belongs to the Object superclass, it doesn't know anything about the object it's copying.
- Because of this, it can only do a field-by-field copy of the Object.
- This is fine if all the fields are primitive types or immutable objects, but if the Object contained other mutable sub objects, only the reference will be copied, and the objects will share data.
- This is called shallow copying.

The Clone method (shallow copying)



The Cloning Decision

- For every class, you need to decide whether or not
  1) The default shallow copy clone is acceptable
  2) You need to deep-copy the class
  3) The object is not allowed to be cloned

- For either of the first two choices, you must
  - Implement the Cloneable interface
  - Redefine the clone() method with the public access modifier

A Cloning Example

- To implement the default clone, you must create a clone() method with a public access modifier and add it to the class.

```
public class cloneExample implements Cloneable {
    public Object clone() {
      try {
        return super.clone();
      }
      catch (CloneNotSupportedException exp)
       {  return null; }
       }
  }
```

Deep copying

- If you need to make a deep copy of a class, your clone method must copy any mutable objects within the class. This is more work, but necessary.
- Any objects within your class that implement Cloneable themselves should be cloned – if they do not, use the new operator to call the constructors, and use = from Strings and primitive types.

Deep copying

```
public class cloneExample implements Cloneable {
  public Object clone() {
    try {
        cloneExample cloned=(cloneExample)super.clone();
        cloned.someDate=(Date)someDate.clone();
      cloned.theSlider=new
 JSlider(theSlider.getMinimum(),theSlider.getMaximum());
        return cloned;
  }
  catch (CloneNotSuportedException e) { return null;}

  private Date someDate;    //cloneable
  private JSlider theSlider;   //not cloneable
}
```

Cloning via Serialization

- As you can see, cloning can be tedious.
- There is a way to automatically clone something using something called *Serialization.* (we'll cover Serialization later in the course)
- Java provides a way to save objects and their state to files. (for use in stuff like RMI, among others)
- We can use this as a way to make a Deep Copy without going through rewriting the Cloneable() method for every class
- To do this, our class must implement the Cloneable AND Serializable interfaces.

Serialized cloning

```
public class Serialtest implements Cloneable, Serializable {
  public Object clone() {
    try {
      //write out a copy of the object to a byte array
      ByteArrayOutputStream bout= new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(bout);
    out.writeObject(this);
    out.close();
    //read a clone of the object from the byte array
    ByteArrayInputStream bin = new
  ByteArrayInputStream(bout.toByteArray());
    ObjectInputStream in = new ObjectInputStream(bin);
    Object ret = in.readObject();
    in.close();
    return ret;
  }
  catch (Exception exp) { return null; }
   }
 }
```

Serialized cloning

- This produces a deep copy of object, without all the tedious work.
- So why not just do this every time? Why didn't the Java creators just implement this themselves?
- Mainly because this way will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the fields itself.

Proxy

- A dynamic proxy class is a class that implements a list of interfaces specified at runtime when the class is created.
- The proxy class has the following methods:
  - All methods required by the specified interfaces; and
  - All methods defined in the Object class (toString, equals, and so on).
- A proxy interface is such an interface that is implemented by a proxy class.
- A proxy instance is an instance of a proxy class. Each proxy instance has an associated invocation handler object, which implements the interface InvocationHandler.
- java.lang.reflect.InvocationHandler
  - Object invoke(Object proxy, Method method, Object[] args)
  - define this method to contain the action that you want carried out whenever a method was invoked on the proxy object.

Con’t

- A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler, passing the proxy instance, a java.lang.reflect.Method object identifying the method that was invoked, and an array of type Object containing the arguments.
- The invocation handler processes the encoded method invocation as appropriate and returns the result of the method invocation on the proxy instance.

Con’t

- To create a proxy object, you use the newProxyInstance method of the Proxy class. The method has three parameters:
  - A class loader
  - An array of Class objects, one for each interface to be implemented.
  - An invocation handler.
- Proxies can be used for many purposes, such as
  - Routing method calls to remote servers;
  - Associating user interface events with actions in a running program; and
  - Tracing method calls for debugging purposes.

Con’t

- java.lang.reflect.Proxy
- static Class getProxyClass(ClassLoader loader, Class[] interfaces)
  - returns the proxy class that implements the given interfaces.
- static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)
  - constructs a new instance of the proxy class that implements the given interfaces. All methods call the invoke method of the given handler object.
- static boolean isProxyClass(Class c)
  - returns true if c is a proxy class.

Properties of Proxy Classes

- Proxy classes are created on the fly in a running program. However, once they are created, they are regular classes.
- All proxy classes extend the class Proxy. A proxy class has only one instance variable—the invocation handler, which is defined in the Proxy superclass.
- All proxy classes override the toString, equals, and hashCode methods of the Object class.
- The names of proxy classes are not defined. The Proxy class in Sun's virtual machine generates class names that begin with the string $Proxy.
- There is only one proxy class for a particular class loader and ordered set of interfaces.
- A proxy class is always public and final. If all interfaces that the proxy class implements are public, then the proxy class does not belong to any particular package. Otherwise, all non-public interfaces and the proxy class also belongs to that package.

Properties of proxy instance

- A proxy instance has the following properties:
- Given a proxy instance proxy and one of the interfaces implemented by its proxy class Foo, the following expression will return true:

  1) proxy instanceof Foo and

  2) (Foo) proxy cast operation will succeed (rather than throwing a ClassCastException)
- Each proxy instance has an associated invocation handler, the one that was passed to its constructor.
- An interface method invocation on a proxy instance will be encoded and dispatched to the invocation handler's invoke method.