CMSC 433 – Programming Language Technologies and Paradigms
Spring 2007

Prototype Pattern

Mar. 27, 2007

What is it?

- When creating an instance of a class is very time-consuming or complex in some way.
  - Rather than creating more instances, you make copies of the original instance, modifying them as appropriate.

An Example

- Consider a database of a large number of swimmers in a league or statewide organization.
  - Each swimmer swims several strokes and distances throughout a season.
- The "best times" for swimmers are tabulated by age group,
  - many swimmers will have birthdays and fall into new age groups within a single season. Thus the query to determine which swimmers did the best in their age group that season is dependent on the date of each meet and on each swimmer's birthday.
  - The computational cost of assembling this table of times is therefore fairly high.
- Once we have a class containing this table, sorted by sex, we could imagine wanting to examine this information sorted just by time, or by actual age rather than by age group.
  - It would not be sensible to recompute these data, and we don't want to destroy the original data order, so some sort of copy of the data object is desirable.

Cloning in Java

- You can make a copy of any Java object using the clone() method.
  - Jobj j1 = (Jobj) j0.clone();
- The clone method returns an object of type Object.
  - cast it to the actual type of the object you are cloning.
- Notes:
  - It is a protected method and can only be called from within the same class or the module that contains that class.
  - You can only clone objects which are declared to implement the Cloneable interface.
  - Objects that cannot be cloned throw the CloneNotSupported Exception.

A Typical Implementation

- package the actual clone method inside the class where it can access the real clone method:

```
public class SwimData implements Cloneable
{
  public Object clone()
  {
     try{
      return super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
}
```

Points to Note

- Advantage: encapsulates the try-catch block inside the public clone method.
- Note that if you declare this public method to have the same name "clone," it must be of type Object, since the internal protected method has that signature.

A Better Implementation

- You could, however, change the name and do the typecasting within the method instead of forcing it onto the user:

```
public SwimData cloneMe()
  {
     try{
      return (SwimData)super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
```

"make" vs. clone

- You can also make "special cloning procedures" that change the data or processing methods in the cloned class, based on arguments you pass to the clone method.
- In this case, method names such as "make" are probably more descriptive and suitable.

Implementing the Prototype

- Let's write a simple program that reads data from a database and then clones the resulting object.
- In our example program, SwimInfo, we just read these data from a file.
  - Use database for large amounts of data

Step 1

- create a class called Swimmer that holds one name, age, club name, sex and time

```
class Swimmer
{   String name;
    int age;
    String club;
    float time;
    boolean female;
```

Step 2

- a class called SwimData maintains a vector of the Swimmers we read in …

```
public class SwimData implements Cloneable
{
  Vector swimmers;
  public SwimData(String filename)
  {
     String s = "";
     swimmers = new Vector();
     //open data file
     InputFile f = new InputFile(filename);
     s= f.readLine();   //read in and parse each line
     while(s != null)
     {
        swimmers.addElement(new Swimmer(s));
        s= f.readLine();
     }
     f.close();
  }
```

Misc Steps…

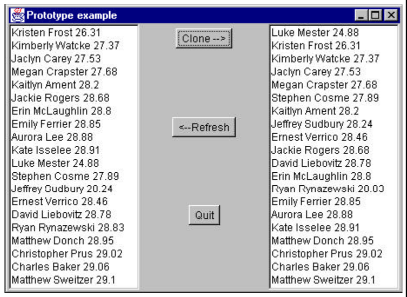
- also provide a getSwimmer method in SwimData and getName, getAge and getTime methods in the Swimmer class.
- Once we've read the data into SwimInfo, we can display it in a list box.

```
swList.removeAll(); //clear list
for (int i = 0; i < sdata.size(); i++)
{
    sw = sdata.getSwimmer(i);
    swList.addItem(sw.getName()+" "+sw.getTime());
}
```

The User Interface

- In the original class, the names are sorted by sex and then by time, while in the cloned class, they are sorted only by time.
- We see the simple user interface that allows us to display the original data on the left and the sorted data in the cloned class on the right

• The left-hand list box is loaded when the program starts.
• The right-hand list box is loaded from the clone when you click on the **Clone** button.
• The **Refresh** button reloads the left-hand list box from the original data object **sdata**.



```
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Kaitlyn Ament 28.2
Jackie Rogers 28.68
Erin McLaughlin 28.8
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Luke Mester 24.88
Stephen Cosme 27.89
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
David Liebovitz 28.78
Ryan Rynazewski 28.83
Matthew Donch 28.95
Christopher Prus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1
```

```
Luke Mester 24.88
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Stephen Cosme 27.89
Kaitlyn Ament 28.2
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
Jackie Rogers 28.68
David Liebovitz 28.78
Erin McLaughlin 28.8
Ryan Rynazewski 28.83
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Matthew Donch 28.95
Christopher Prus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1
```

The Cloning

- when the user clicks on the Clone button, we’ll clone **sdata** and sort the data differently in the new class (**sxdata**).

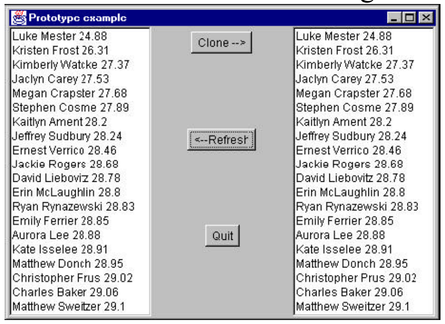
```
sxdata = (SwimData)sdata.clone();
sxdata.sortByTime();   //re-sort
cloneList.removeAll(); //clear list

//now display sorted values from clone
for(int i=0; i< sxdata.size(); i++)
  {
   sw = sxdata.getSwimmer(i);
   cloneList.addItem(sw.getName()+" "+sw.getTime());
 }
```

Are we there yet?

Are we done yet?

Wait a Minute?

- let’s click on the Refresh button to reload the left-hand list box from the original data.



```
Luke Mester 24.88
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Stephen Cosme 27.89
Kaitlyn Ament 28.2
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
Jackie Rogers 28.68
David Lieboviz 28.78
Erin McLaughlin 28.8
Ryan Rynazewski 28.83
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Matthew Donch 28.95
Christopher Frus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1
```

```
Luke Mester 24.88
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Stephen Cosme 27.89
Kaitlyn Ament 28.2
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
Jackie Rogers 28.68
David Liebovitz 28.78
Erin McLaughlin 28.8
Ryan Rynazewski 28.83
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Matthew Donch 28.95
Christopher Prus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1
```

The names in the left-hand list box also been re-sorted.

Shallow Copy

- This occurs in Java because the clone method is a “shallow copy” of the original class.
- In other words, the references to the data objects are copies, but they refer to the same underlying data.
- Thus, any operation we perform on the copied data will also occur on the original data in the Prototype class.
- Also, what about Mutable fields?

A “Deep Copying” Clone

```
public Object deepClone()
  {
    try{
     ByteArrayOutputStream b = new ByteArrayOutputStream();
     ObjectOutputStream out = new ObjectOutputStream(b);
     out.writeObject(this);
     ByteArrayInputStream bIn = new
               ByteArrayInputStream(b.toByteArray());
   ObjectInputStream oi = new ObjectInputStream(bIn);
   return (oi.readObject());
  }
  catch (Exception e)
  {  System.out.println("exception:"+e.getMessage());
    return null;
  }
}
```

Serializable

- This is possible because of the serializable interface.
  - A class is said to be serializable if you can write it out as a stream of bytes and read those bytes back in to reconstruct the class. This is how Java remote method invocation (RMI) is implemented.
- We can declare both the Swimmer and SwimData classes as Serializable,
  - public class SwimData implements Cloneable, Serializable
  - class Swimmer implements Serializable
- So we can write the bytes to an output stream and reread them to create a complete data copy of that instance of a class

More on Cloning

A Simple Example (pg. 1)

```java
import java.util.Date;

public abstract class Fruit implements Cloneable {

  public Fruit( String aColour, Date aBestBeforeDate ) {
    super();
    fColour = aColour;
    //defensive copy needed for this mutable object
    fBestBeforeDate = new Date( aBestBeforeDate.getTime() );
  }

  public abstract void ripen();

  public String getColour() {
    return fColour;
  }

  public Date getBestBeforeDate() {
    //return defensive copy of this mutable object
    return new Date ( fBestBeforeDate.getTime() );
  }
```

A Simple Example (pg. 2)

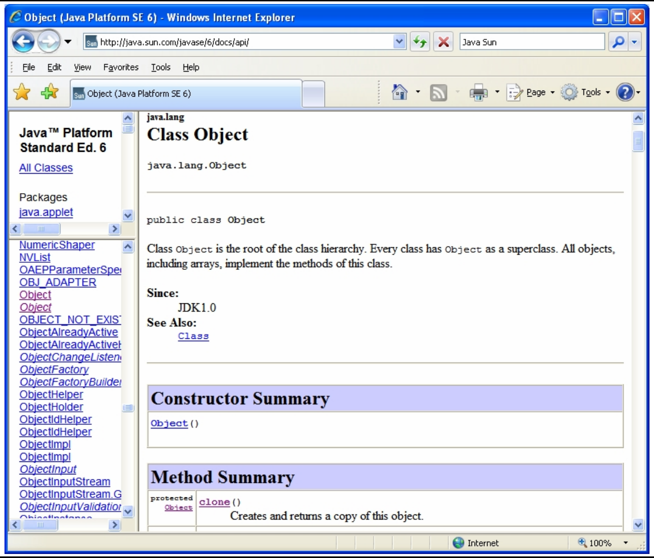
```java
/**
* Implement clone as follows
* <ul>
* <li>the class declaration "implements Cloneable" (not needed if already
* declared in superclass)
* <li>declare clone method as public
* <li>if the class is final, clone does not need to throw CloneNotSupportedException
* <li>call super.clone and cast to this class
* <li>as in defensive copying, ensure each mutable field has an independent copy
* constructed, to avoid sharing internal state between objects
* </ul>
*/
public Object clone() throws CloneNotSupportedException {
  //get initial bit-by-bit copy, which handles all immutable fields
  Fruit result = (Fruit)super.clone();

  //mutable fields need to be made independent of this object, for reasons
  //similar to those for defensive copies - to prevent unwanted access to
  //this object's internal state
  result.fBestBeforeDate = new Date( this.fBestBeforeDate.getTime() );

  return result;
}
```

A Simple Example (pg. 3)

```java
  /// PRIVATE ////

  /**
  * Strings are always immutable.
  */
  private String fColour;

  /**
  * Date is a mutable object. In this class, this object field is to be treated
  * as belonging entirely to this class, and no user of this class is
  * to be able to directly access and change this field's state.
  */
  private Date fBestBeforeDate;
}
```

A Simple Example (pg. 4)

```java
import java.util.Date;

public final class Apple extends Fruit {

  public Apple( String aColour, Date aBestBeforeDate ) {
    super( aColour, aBestBeforeDate );
  }

  public void ripen() {
    //empty implementation of abstract method
  }

  /**
  * The Apple subclass does not support clone.
  */
  public final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```


Object (Java Platform SE 6) - Windows Internet Explorer
http://java.sun.com/javase/6/docs/api/
Java Sun
File Edit View Favorites Tools Help
Object (Java Platform SE 6)
Page
Tools
Internet
100%

Java™ Platform
Standard Ed. 6

All Classes

Packages
java.applet

NumericShaper
NVList
OAEPParameterSpe
OBJ_ADAPTER
Object
Object
OBJECT_NOT_EXIS
ObjectAlreadyActive
ObjectAlreadyActive
ObjectChangeListen
ObjectFactory
ObjectFactoryBuilde
ObjectHelper
ObjectHolder
ObjectIdHelper
ObjectIdHelper
ObjectImpl
ObjectImpl
ObjectInput
ObjectInputStream
ObjectInputStream.G
ObjectInputValidatio

java.lang

Class Object

```
java.lang.Object
```

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:
JDK1.0
See Also:
`Class`

Constructor Summary

`Object()`

Method Summary

`protected Object`	`clone()` Creates and returns a copy of this object.

public interface Cloneable

- A class implements the *Cloneable* interface to indicate to the **Object.clone()** method that it is legal for that method to make a field-for-field copy of instances of that class.
  - Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
  - By convention, classes that implement this interface should override Object.clone (which is protected) with a public method. Note that this interface does not contain the clone method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

What is cloning?

protected Object clone()
throws CloneNotSupportedException

- Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:
  - x.clone() != x will be true,
- and that the expression:
  - x.clone().getClass() == x.getClass() will be true,
  - but these are not absolute requirements.
- While it is typically the case that:
  - x.clone().equals(x) will be true, this is not an absolute requirement.
- By convention, the returned object should be obtained by calling super.clone. If a class and all of its superclasses (except Object) obey this convention, it will be the case that
  - x.clone().getClass() == x.getClass().

Some Rules of Cloning

- The method clone for class Object performs a specific cloning operation.
  - If the class of this object does not implement the interface Cloneable, then a CloneNotSupportedException is thrown.
    - Note that all arrays are considered to implement the interface Cloneable.
  - Otherwise, this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment;
  - the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.
- The class Object does not itself implement the interface Cloneable, so calling the clone method on an object whose class is Object will result in throwing an exception at run time.