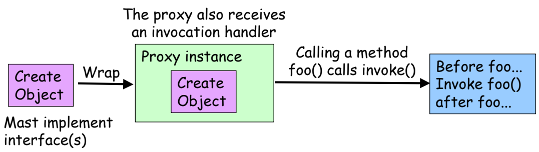
Dynamic Proxies

Amit Shabtay

Dynamic Proxies

- Main idea:
  - The proxy wraps objects and adds functionality.
  - All proxy methods invocations are dispatched to the invoke(...) method of the instance invocation handler.
  - These methods will be handled by the proxy instance.

Dynamic Proxies Scheme



The proxy also receives an invocation handler

Create Object

Wrap

Proxy instance

Create Object

Calling a method foo() calls invoke()

Before foo...
Invoke foo()
after foo...

Mast implement interface(s)

The heart is the invocation handler

Dynamic Proxies

- Support for creating classes at runtime
  - Each such class implements interface(s)
  - Every method call to the class will be delegated to a handler, using reflection
  - The created class is a proxy for its handler
- Applications
  - Aspect-Oriented Programming: standard error handling, log & debug for all objects
  - Creating dynamic event handlers

Invocation Handlers

- Start by defining the handler:
  - `interface java.lang.reflect.InvocationHandler`
  - With a single method:

```
Object invoke(     // return value of call
  Object proxy,    // call's target
  Method method,   // the method called
  Object[] args)   // method's arguments
```

- The "real" call made: proxy.method(args)
  - Simplest invoke(): method.invoke(proxy,args)

Creating a Proxy Class

- Define the proxy interface:

```
interface Foo { Object bar(Object obj); }
```

- Use `java.lang.reflect.Proxy` static methods to create the proxy class:

```
Class proxyClass = Proxy.getProxyClass(
     Foo.class.getClassLoader(), new
  Class[] { Foo.class });
```

- First argument – the new class's class loader
- 2nd argument – list of implemented **interfaces**
- The expression *C.class* for a class C is the static version of *C_obj.getClass()*

Creating a Proxy Instance

- A proxy class has one constructor which takes one argument – the invocation handler
- Given a proxy class, find and invoke this constructor:

```
Foo foo = (Foo)proxyClass.
  getConstructor(new Class[] { InvocationHandler.class }).
  newInstance(new Object[] { new MyInvocationHandler() });
```

- Class Proxy provides a shortcut:

```
Foo f = (Foo) Proxy.newProxyInstance(
                Foo.class.getClassLoader(),
                new Class[] { Foo.class },
                new MyInvocationHandler());
```

- Note that all members of Class[] should be interfaces only.

A Few More Details I

- We ignored a bunch of exceptions
  - *IllegalArgumentException* if proxy class can't exist
  - *UndeclaredThrowableException* if the handler throws an exception the interface didn't declare
  - *ClassCastException* if return value type is wrong
  - *InvocationTargetException* wraps checked exceptions
- A proxy class's name is undefined
  - But begins with Proxy$
- The syntax is very unreadable!
  - Right, but it can be encapsulated inside the handler

A Few More Details II

- Primitive types are wrapped by *Integer*, *Boolean*, and so on for argument
- This is also true for the return values.
- If null is returned for a primitive type, then a NullPointerException will be thrown by the method invocation.

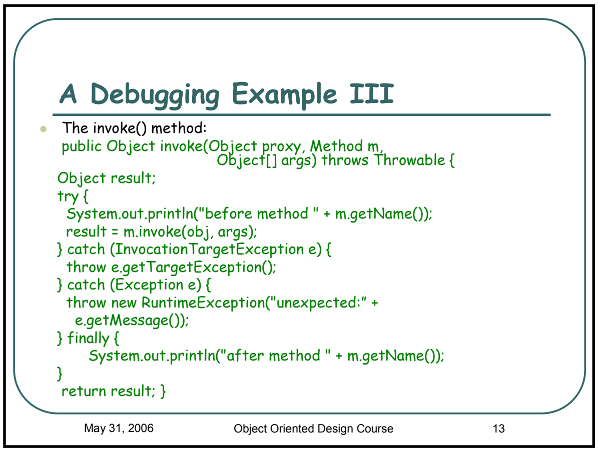
A Debugging Example

- We'll write an extremely generic class, that can wrap any object and print a debug message before and after every method call to it
- Instead of a public constructor, it will have a static factory method to encapsulate the proxy instance creation
- It will use InvocationTargetException to be exception neutral to the debugged object

A Debugging Example II

- The class's definition and construction:

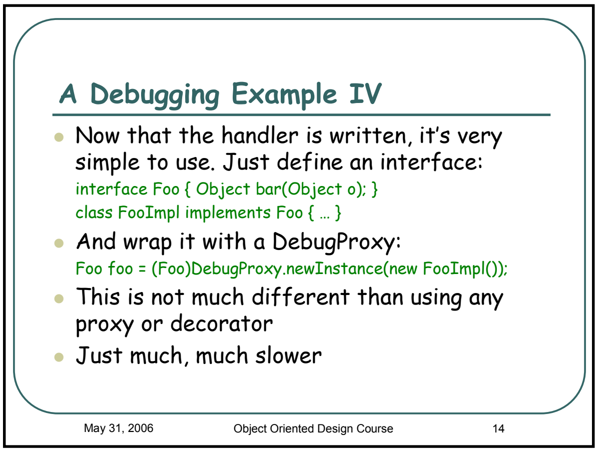
```
public class DebugProxy
implements java.lang.reflect.InvocationHandler {
  private Object obj;
  public static Object newInstance(Object obj) {
    return java.lang.reflect.Proxy.newProxyInstance(
          obj.getClass().getClassLoader(),
          obj.getClass().getInterfaces(),
          new DebugProxy(obj)); }
  private DebugProxy(Object obj) {
          this.obj = obj; }
```



A Debugging Example III

- The invoke() method:

```
 public Object invoke(Object proxy, Method m,
                               Object[] args) throws Throwable {
Object result;
try {
  System.out.println("before method " + m.getName());
  result = m.invoke(obj, args);
} catch (InvocationTargetException e) {
  throw e.getTargetException();
} catch (Exception e) {
  throw new RuntimeException("unexpected:” +
    e.getMessage());
} finally {
       System.out.println("after method " + m.getName());
}
 return result; }
```



A Debugging Example IV

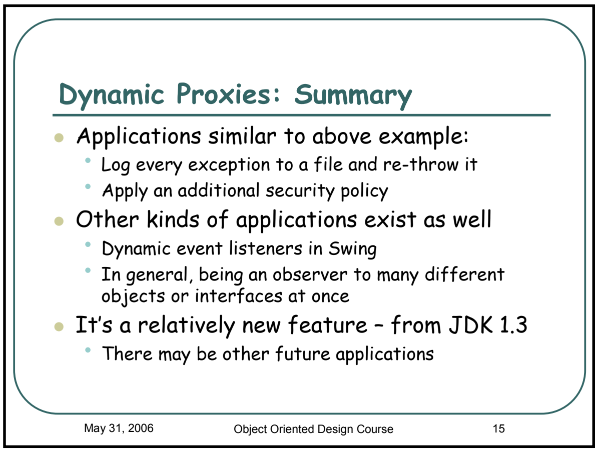
- Now that the handler is written, it’s very simple to use. Just define an interface:

```
interface Foo { Object bar(Object o); }
class FooImpl implements Foo { … }
```

- And wrap it with a DebugProxy:

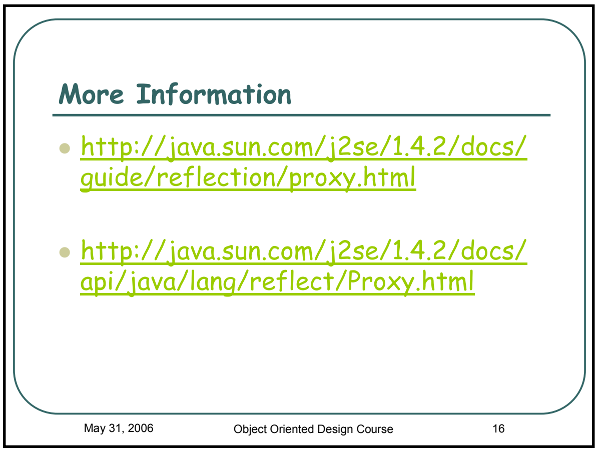
```
Foo foo = (Foo)DebugProxy.newInstance(new FooImpl());
```

- This is not much different than using any proxy or decorator
- Just much, much slower



Dynamic Proxies: Summary

- Applications similar to above example:
  - Log every exception to a file and re-throw it
  - Apply an additional security policy
- Other kinds of applications exist as well
  - Dynamic event listeners in Swing
  - In general, being an observer to many different objects or interfaces at once
- It’s a relatively new feature – from JDK 1.3
  - There may be other future applications



More Information

- http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html

- http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/Proxy.html