Dynamic Proxies In Java

Dynamic Proxies

- Proxy objects are useful in many situations to act as an intermediary between a client object and a target object
- Usually, the proxy class is already available as Java bytecodes, having been compiled from the Java source file for the proxy class
- When needed, the bytecodes for the proxy class are loaded into the Java Virtual Machine and proxy objects can then be instantiated
- But, in some circumstances, it is useful to dynamically generate the bytecodes for the proxy class at runtime
- This module will look at the techniques for dynamically generating proxies in Java and the benefits of doing so

Vehicle Example With No Proxy

- First, let's show a client interacting with a target object directly
- Suppose we have an IVehicle interface as follows:

```
/**
 * Interface IVehicle.
 */
public interface IVehicle {
  public void start();
  public void stop();
  public void forward();
  public void reverse();
  public String getName();
}
```

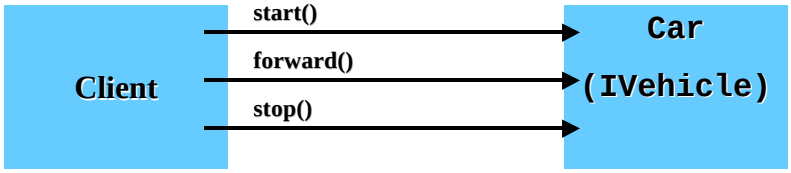

Vehicle Example With No Proxy (Continued)

- Here's a Car class that implements the IVehicle interface:

```
/**
 * Class Car
 */
public class Car implements IVehicle {
  private String name;

  public Car(String name) {this.name = name;}

  public void start() {
    System.out.println("Car " + name + " started");
  }

  // stop(), forward(), reverse() implemented similarly.
  // getName() not shown.
}
```

Vehicle Example With No Proxy (Continued)

```
/**
 * Class Client1.
 * Interacts with a Car Vehicle directly.
 */
public class Client1 {

  public static void main(String[] args) {

    IVehicle v = new Car("Botar");
    v.start();
    v.forward();
    v.stop();
  }
}
```



Vehicle Example With No Proxy (Continued)

- Output for the vehicle example with no proxy:

```
Car Botar started
Car Botar going forward
Car Botar stopped
```

Vehicle Example With Proxy

- Now let's have the client interact with the target object through a proxy
- Remember that the main intent of a proxy is to control access to the target object, rather than to enhance the functionality of the target object
- Ways that proxies can provide access control include:
  - ⇨ Synchronization
  - ⇨ Authentication
  - ⇨ Remote Access
  - ⇨ Lazy instantiation

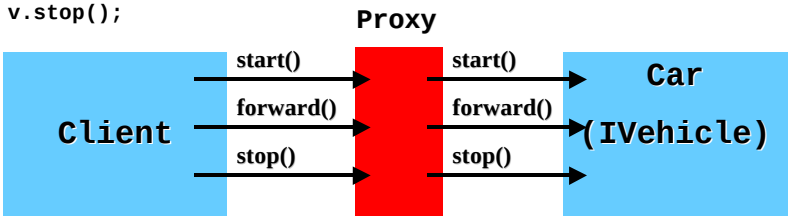
Vehicle Example With Proxy (Continued)

- Here's our VehicleProxy class:

```
/**
 * Class VehicleProxy.
 */
public class VehicleProxy implements IVehicle {
  private IVehicle v;

  public VehicleProxy(IVehicle v) {this.v = v;}

  public void start() {
    System.out.println("VehicleProxy: Begin of start()");
    v.start();
    System.out.println("VehicleProxy: End of start()");
  }
  // stop(), forward(), reverse() implemented similarly.
  // getName() not shown.
}
```

Vehicle Example With Proxy (Continued)

```
/**
 * Class Client2.
 * Interacts with a Car Vehicle through a VehicleProxy.
 */
public class Client2 {
  public static void main(String[] args) {
    IVehicle c = new Car("Botar");
    IVehicle v = new VehicleProxy(c);
    v.start();
    v.forward();
    v.stop();
  }
}
```



Vehicle Example With Proxy (Continued)

- Output for the vehicle example with a proxy:

```
VehicleProxy: Begin of start()
Car Botar started
VehicleProxy: End of start()
VehicleProxy: Begin of forward()
Car Botar going forward
VehicleProxy: End of forward()
VehicleProxy: Begin of stop()
Car Botar stopped
VehicleProxy: End of stop(
```

Dynamic Proxies In Java

- Java 1.3 supports the creation of dynamic proxy classes and instances
- A *dynamic proxy class* is a class that implements a list of interfaces specified at runtime when the class is created
- A *proxy interface* is 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
- 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

Dynamic Proxy Class

- Proxy classes are created using the new java.lang.reflect.Proxy class
- Proxy classes are public, final, non-abstract subclasses of java.lang.reflect.Proxy
- The unqualified name of a proxy class is unspecified. The space of class names that begin with the string "$Proxy" should be, however, reserved for proxy classes.
- A proxy class implements exactly the interfaces specified at its creation
- Since a proxy class implements all of the interfaces specified at its creation, invoking getInterfaces() on its Class object will return an array containing the same list of interfaces (in the order specified at its creation)

Dynamic Proxy Class

- Each proxy class has one public constructor that takes one argument, an implementation of the interface InvocationHandler, to set the invocation handler for a proxy instance
- Rather than having to use the reflection API to access the public constructor, a proxy instance can be also be created by calling the Proxy.newInstance() method, which combines the actions of calling Proxy.getProxyClass() with invoking the constructor with an invocation handler

The java.lang.reflect.Proxy Class

- public static Class getProxyClass(ClassLoader loader,
  Class[] interfaces)
  throws IllegalArgumentException
  - ⇨ Creates a proxy class defined in the specified class loader and which implements the specified interfaces. Returns the java.lang.Class object for the generated proxy class.
- protected Proxy(InvocationHandler ih)
  - ⇨ Constructs a new Proxy instance from a subclass (typically, a dynamic proxy class) with the specified value for its invocation handler
- public static boolean isProxyClass(Class c)
  - ⇨ Returns true if and only if the specified class was dynamically generated to be a proxy class using the getProxyClass() method or the newProxyInstance() method of the Proxy class

The java.lang.reflect.Proxy Class

- public static Object newProxyInstance(ClassLoader loader,
  Class[] interfaces,
  InvocationHandler ih)
  throws IllegalArgumentException
  - ⇨ Creates a proxy class defined in the specified class loader and which implements the specified interfaces. In addition, creates an instance of the proxy by invoking the one public proxy constructor which sets the associated invocation handler to the specified handler. Returns a reference to the proxy instance.
  - ⇨ `Proxy.newProxyInstance(cl, interfaces, ih);` is equivalent to

```
Proxy.getProxyClass(cl,
interfaces).getConstructor(new Class[] {
InvocationHandler.class }).newInstance(new
Object[] {ih});
```


The java.lang.reflect.Proxy Class

- public static InvocationHandler getInvocationHandler
  (Object proxy)
  throws IllegalArgumentException
  - ⇨ Returns the invocation handler for the specified proxy instance

The java.lang.reflect.InvocationHandler Interface

- Each proxy instance has an associated invocation handler. When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the invoke() method of its invocation handler
- public Object invoke(Object proxy, Method method,
  Object[] args) throws Throwable
  - ⇨ Processes a method invocation on a proxy instance and returns the result. The proxy parameter is the proxy instance that the method was invoked on. The method parameter is the Method instance corresponding to the interface method invoked on the proxy instance. The args parameter is an array of objects containing the values of the arguments passed in the method invocation on the proxy instance, or null if the interface method takes no arguments.

Vehicle Example With Dynamic Proxy

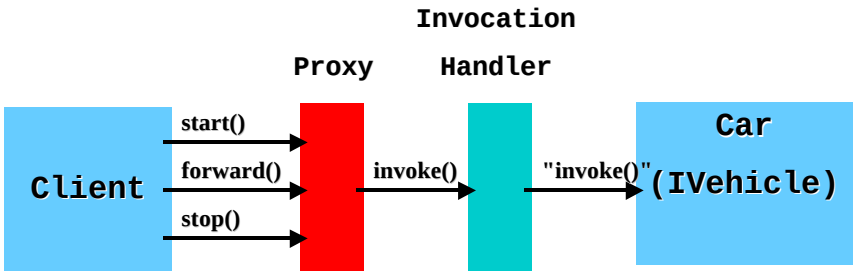
- To do our vehicle example with a dynamic proxy, we first need an invocation handler:

```
import java.lang.reflect.*;
/**
 * Class VehicleHandler.
 */
public class VehicleHandler implements InvocationHandler {
  private IVehicle v;

  public VehicleHandler(IVehicle v) {this.v = v;}

  public Object invoke(Object proxy, Method m, Object[] args)
         throws Throwable {
    System.out.println("Vehicle Handler: Invoking " +
                       m.getName());
    return m.invoke(v, args);
  }
}
```

Vehicle Example With Dynamic Proxy (Continued)

- Notice how we use the Reflection API to invoke the proper method on our target object:

```
m.invoke(v, args);
```



Vehicle Example With Dynamic Proxy (Continued)

```
import java.lang.reflect.*;
/**
 * Class Client3.
 * Interacts with a Car Vehicle through a dynamically
 * generated VehicleProxy.
 */
public class Client3 {
  public static void main(String[] args) {
    IVehicle c = new Car("Botar");
    ClassLoader cl = IVehicle.class.getClassLoader();
    IVehicle v = (IVehicle) Proxy.newProxyInstance(cl,
       new Class[] {IVehicle.class}, new VehicleHandler(c));
    v.start();
    v.forward();
    v.stop();
  }
}
```

Vehicle Example With Dynamic Proxy (Continued)

- Output for the vehicle example with a dynamic proxy:

```
Vehicle Handler: Invoking start
Car Botar started
Vehicle Handler: Invoking forward
Car Botar going forward
Vehicle Handler: Invoking stop
Car Botar stopped
```


Uses For Dynamic Proxies

- In the Vehicle example, there seems to be little benefit in dynamically generating the proxy:
  - ⇒ We still had to write the invocation handler class!
  - ⇒ There is now another object layer between the client and the target!
- So where would we use dynamic proxies??
  - ⇒ Generic Delegation
  - ⇒ Dynamic generation of proxies (stubs) for remote objects

Logged Vehicle Example

- To illustrate the idea of Generic Delegation, let's add a logging capability to our Vehicle Example
- Suppose that we want to log each action (start, stop, etc.) that we perform on a Car, but we do not want to modify the existing Car code
- Sounds like a job for the Decorator Pattern!
- We'll write a LoggedVehicle class that implements the IVehicle interface, logs each requested action and then delegates the actual action to a contained IVehicle object
- *The essence of the Decorator Pattern is delegation through composition!*

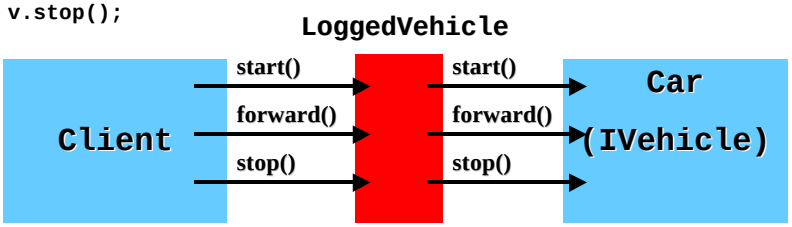
Logged Vehicle Example (Continued)

- Here's the LoggedVehicle class:

```
/**
 * Class LoggedVehicle.
 */
public class LoggedVehicle implements IVehicle {
  private IVehicle v;

  public LoggedVehicle(IVehicle v) {this.v = v;}

  public void start() {
    System.out.println("Log Entry: Vehicle " + v.getName() +
                       " started");
    v.start();
  }
  // stop(), forward(), reverse() implemented similarly.
  // getName() not shown.
}
```

Logged Vehicle Example (Continued)

```
/**
 * Class Client4.
 * Interacts with a Car Vehicle through a Logging Decorator.
 */
public class Client4 {
  public static void main(String[] args) {
    IVehicle c = new Car("Botar");
    IVehicle v = new LoggedVehicle(c);
    v.start();
    v.forward();
    v.stop();
  }
}
```



Logged Vehicle Example (Continued)

- Output for the vehicle example with a logging decorator:

```
Log Entry: Vehicle Botar started
Car Botar started
Log Entry: Vehicle Botar going forward
Car Botar going forward
Log Entry: Vehicle Botar stopped
Car Botar stopped
```

- Notice how similar this example is to the simple proxy example! The difference between the Proxy Pattern and the Decorator Pattern is one of intent: Proxy provides access control, while Decorator adds functionality, in this case a logging capability.

Logged Vehicle Example (Continued)

- While the LoggedVehicle decorator class provides a logging capability for any class that implements the IVehicle interface, there are two drawbacks to this approach:
  - ⇨ It was tedious to have to implement all of the methods of the IVehicle interface in the LoggedVehicle class
  - ⇨ Logging is a generic capability that we may want to add to other interfaces in which case we have to write another wrapper class
- Both of these drawbacks can be overcome by using dynamic proxies
- The dynamic proxy will automatically implement all of the methods of the interface we specify, relieving us of the tedium of doing this implementation ourselves
- And the reflective method invocation in our invocation handler supports the desired generic delegation!

Generic Delegation Example

- Here is a generic logger class:

```
import java.lang.reflect.*;
/**
 * Class GenericLogger.
 */
public class GenericLogger implements InvocationHandler {
  private Object target;

  public GenericLogger(Object target) {this.target = target;}

  public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable {

    System.out.println("Generic Logger Entry: Invoking " +
                       m.getName());
    return m.invoke(target, args);
  }
}
```

Generic Delegation Example (Continued)

```
import java.lang.reflect.*;
/**
 * Class Client5.
 * Interacts with a Car Vehicle through a dynamically
 * generated proxy and a Generic Logger.
 */
public class Client5 {
  public static void main(String[] args) {
    IVehicle c = new Car("Botar");
    ClassLoader cl = IVehicle.class.getClassLoader();
    IVehicle v = (IVehicle) Proxy.newProxyInstance(cl,
       new Class[] {IVehicle.class}, new GenericLogger(c));
    v.start();
    v.forward();
    v.stop();
  }
}
```


Generic Delegation Example (Continued)

- Output for the vehicle example with a generic logger:

```
Generic Logger Entry: Invoking start
Car Botar started
Generic Logger Entry: Invoking forward
Car Botar going forward
Generic Logger Entry: Invoking stop
Car Botar stopped
```

Generic Delegation Example (Continued)

- The great thing about this generic logger is that it can be used to add a logging capability to any interface!
- Consider an interface for shapes:

```
/**
 * Interface IShape.
 */
public interface IShape {
  public void draw();
  public void print();
  public void move();
  public void resize();
}
```


Generic Delegation Example (Continued)

```
import java.lang.reflect.*;
/**
 * Class Client6.
 * Interacts with a Rectangle Shape through a dynamically
 * generated proxy and a Generic Logger.
 */
public class Client6 {
  public static void main(String[] args) {
    IShape rect = new Rectangle();
    ClassLoader cl = IShape.class.getClassLoader();
    IShape s = (IShape) Proxy.newProxyInstance(cl,
      new Class[] {IShape.class}, new GenericLogger(rect));
    s.draw();
    s.move();
    s.resize();
  }
}
```

Generic Delegation Example (Continued)

- Output for the shape example with a generic logger:

```
Generic Logger Entry: Invoking draw
Rectangle drawn
Generic Logger Entry: Invoking move
Rectangle moved
Generic Logger Entry: Invoking resize
Rectangle resized
```