CS525 Advanced Software Development

Lesson 11 – The Proxy Pattern

Design Patterns
Elements of Reusable Object-Oriented Software

Payman Salek, M.S.
March 2022

Setting the stage

Ever play good cop, bad cop?

You're the good cop and you provide all your services in a nice and friendly manner, but you don't want everyone asking you for services, so you have the bad cop control access to you.

Setting the stage (Remote Gumball)

Remote Gumball Machine

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

The location is passed into the constructor and stored in the instance variable.

Let's also add a getter method to grab the location when we need it.

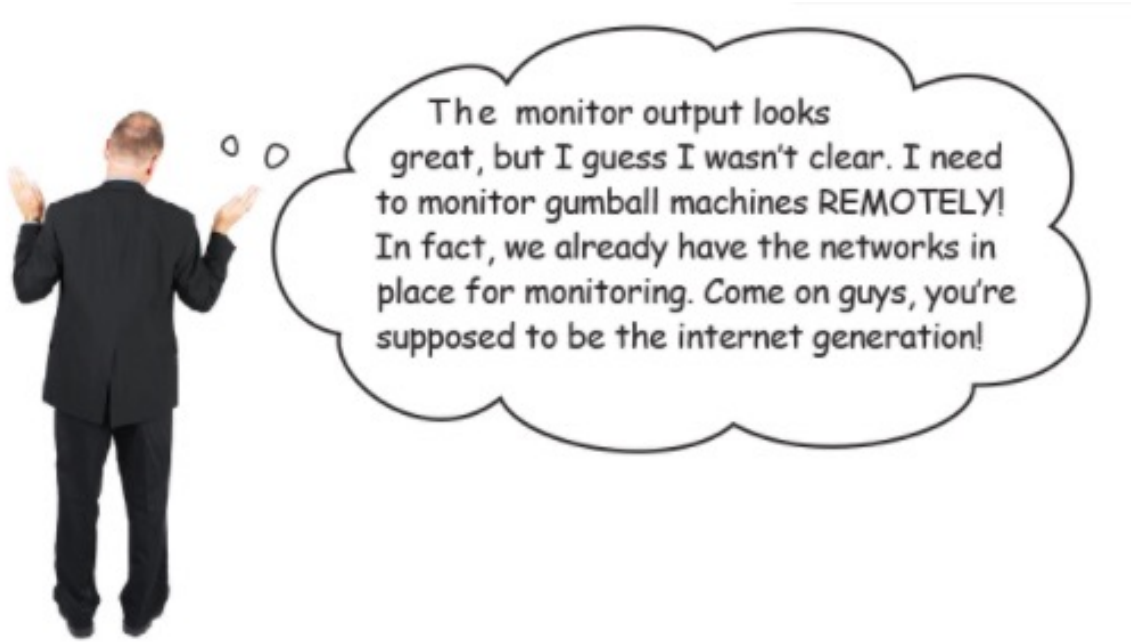
Gumball Monitor

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report() method just prints a report with location, inventory, and the machine's state.

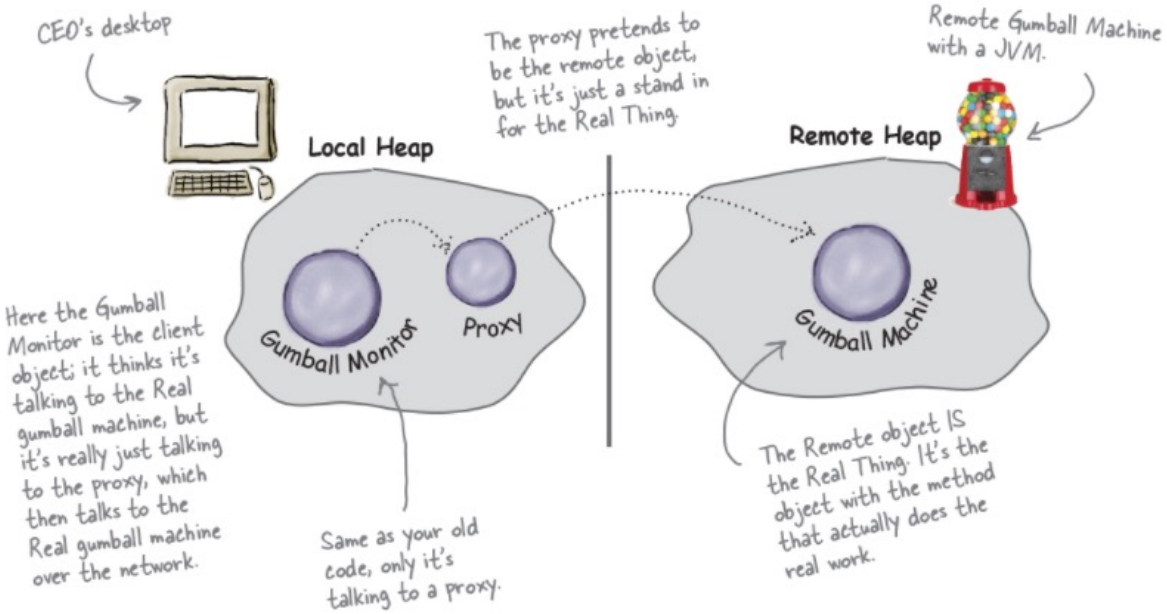
Not what the boss wanted!



The role of the “remote proxy”

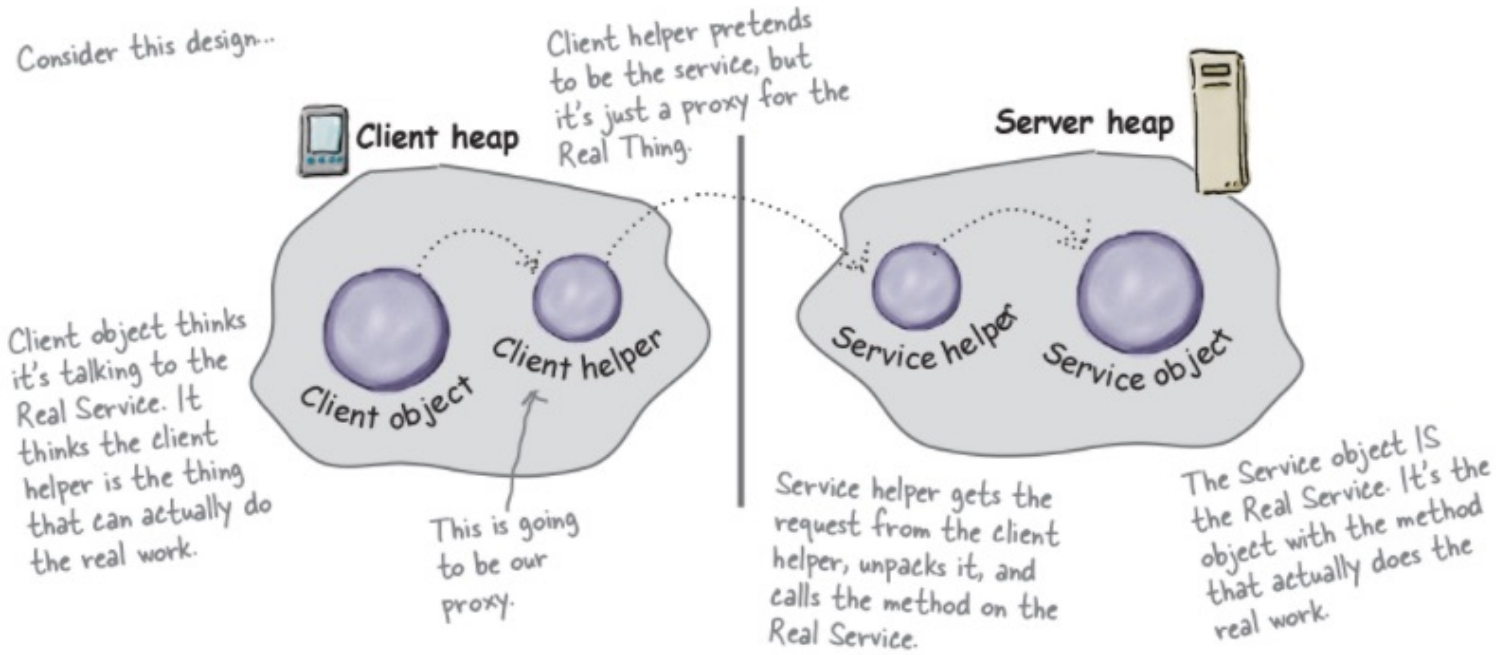
A remote proxy acts as a local representative to a remote object. What’s a “remote object”? It’s an object that lives in the heap of a different Java Virtual Machine (or more generally, a remote object that is running in a different address space). What’s a “local representative”? It’s an object that you can call local methods on and have them forwarded on to the remote object.

Remote Proxy

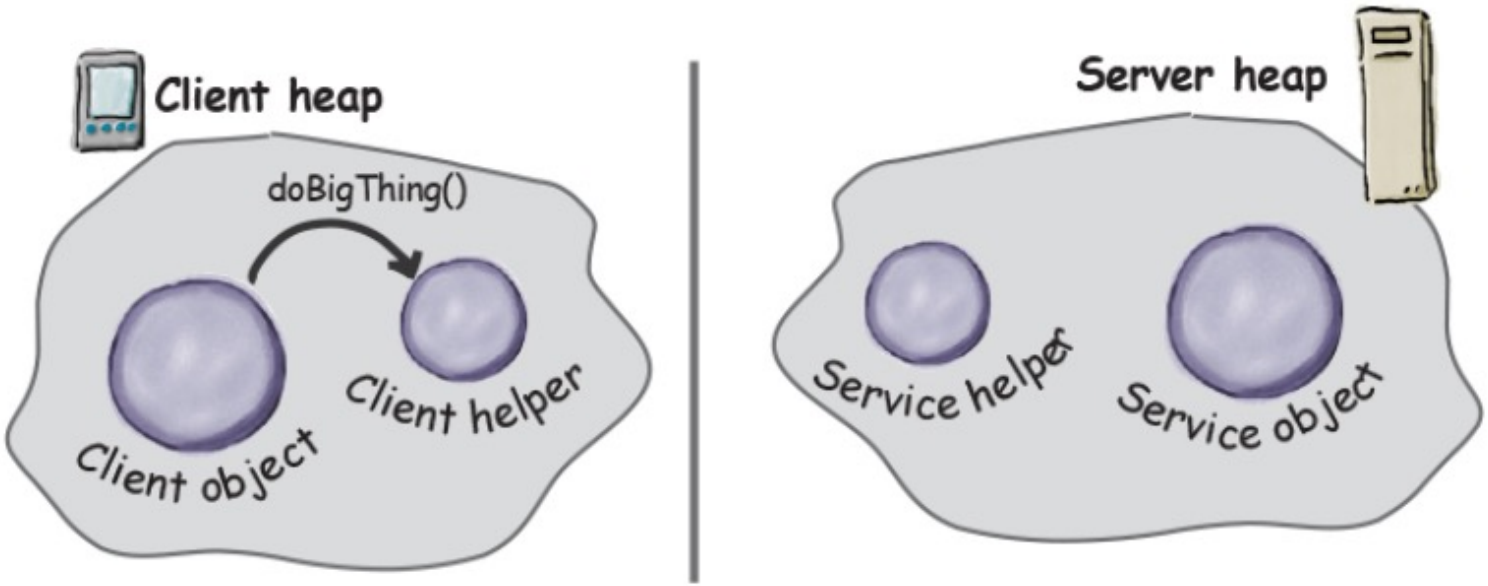


But how would you actually do it??

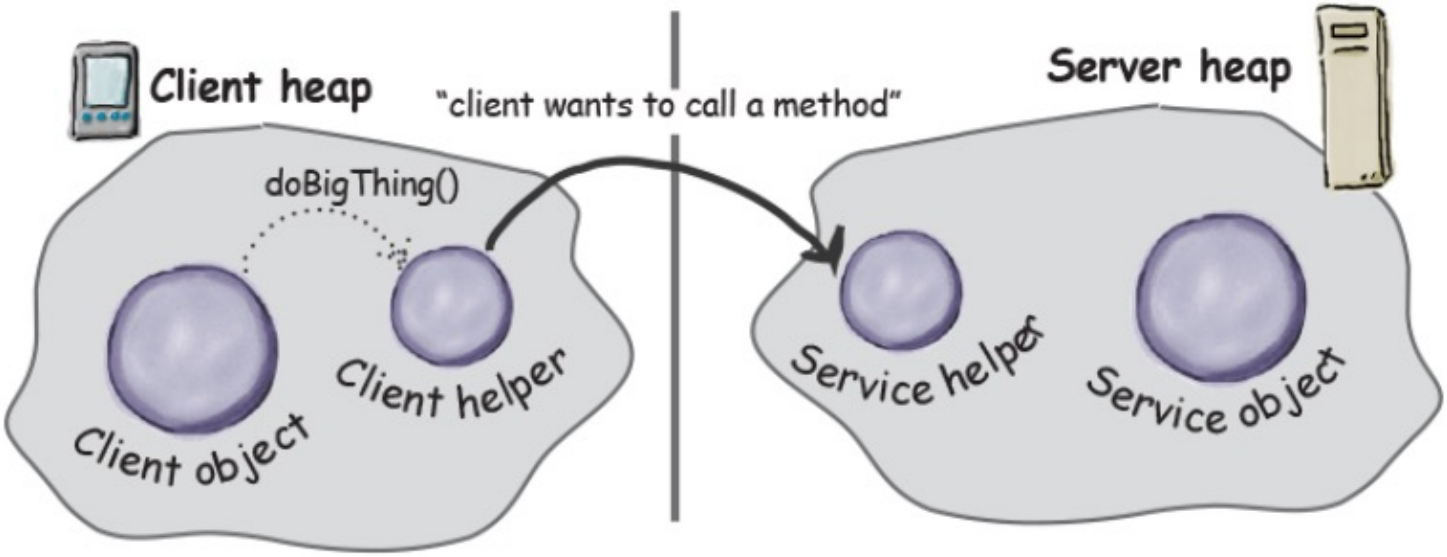
Remote Methods 101



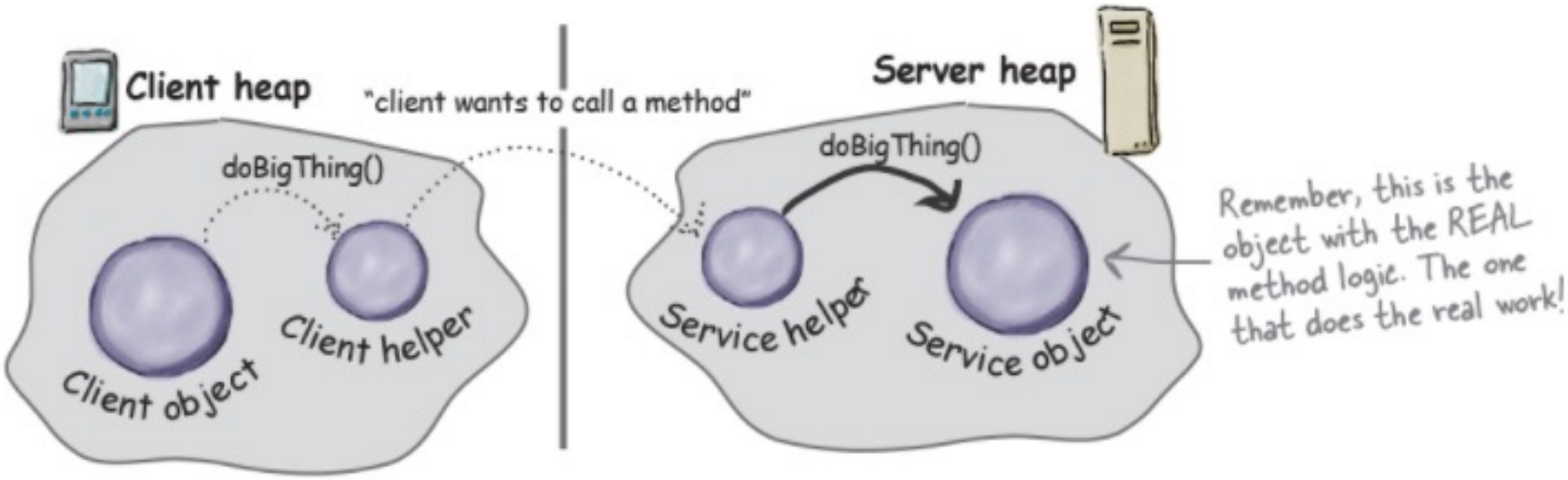
How the method calls happen (step 1)



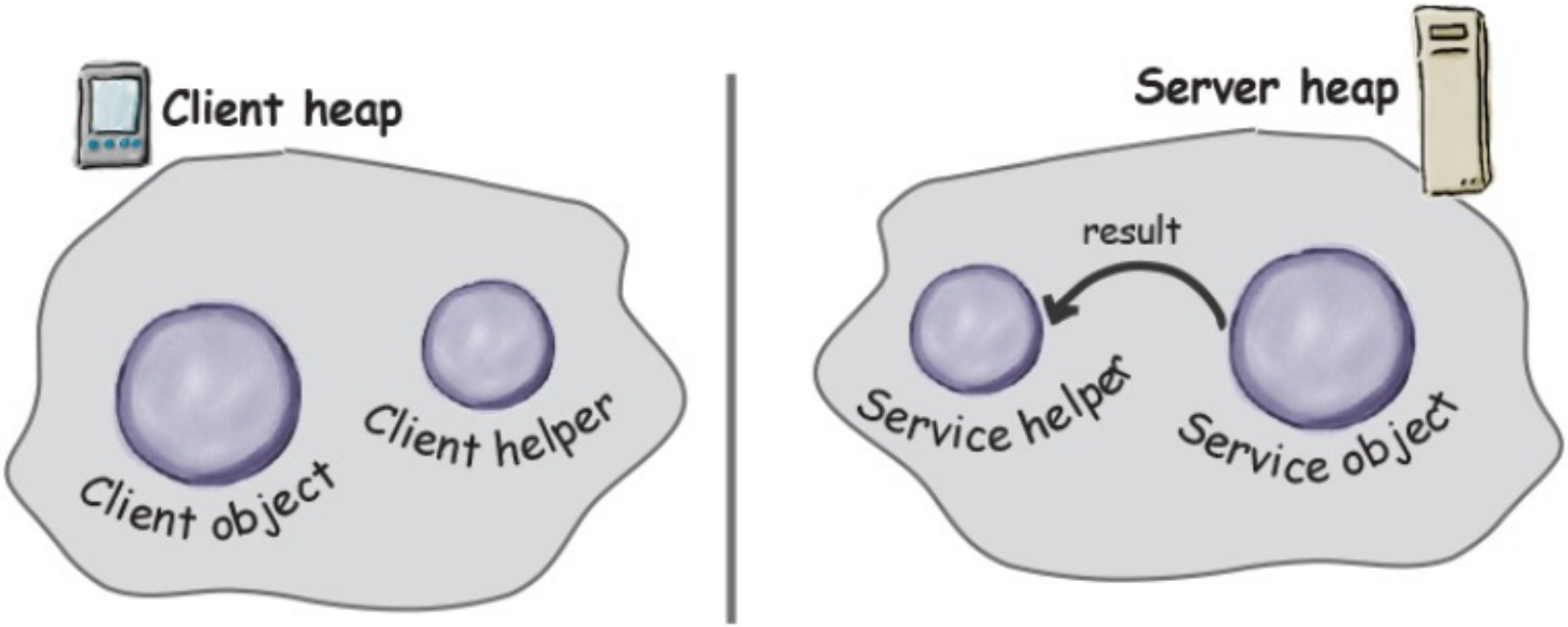
How the method calls happen (step 2)



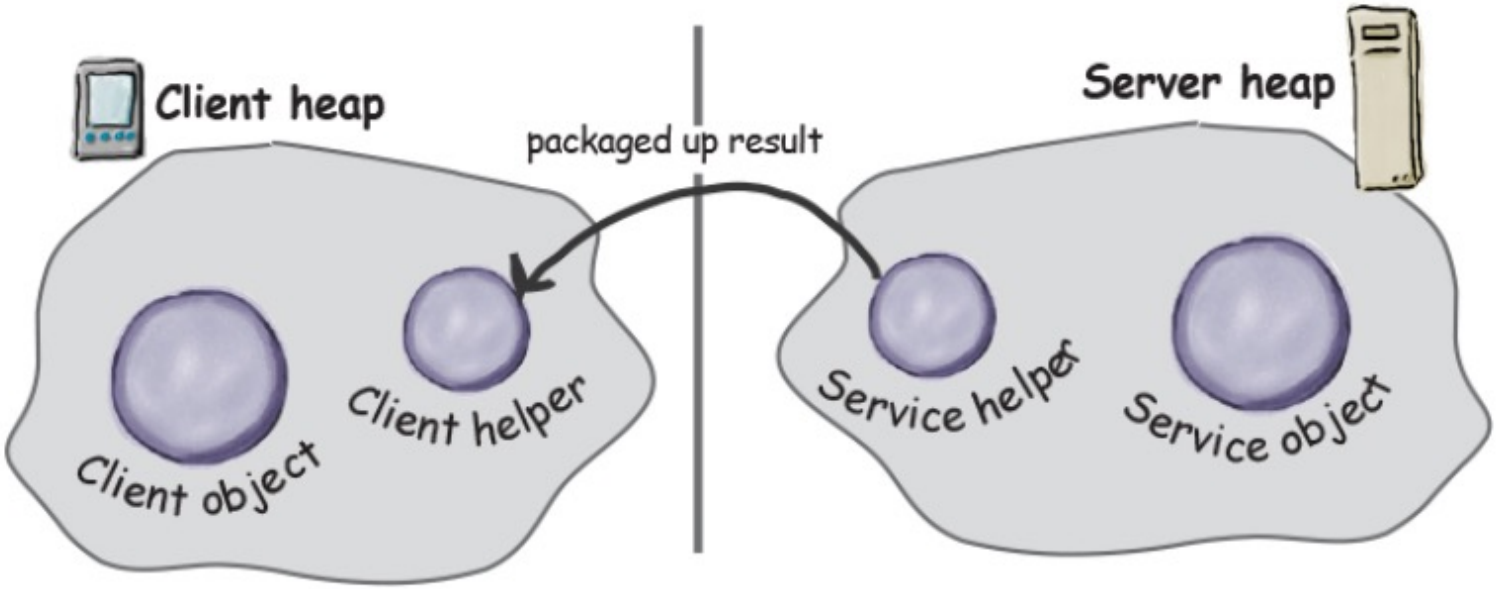
How the method calls happen (step 3)



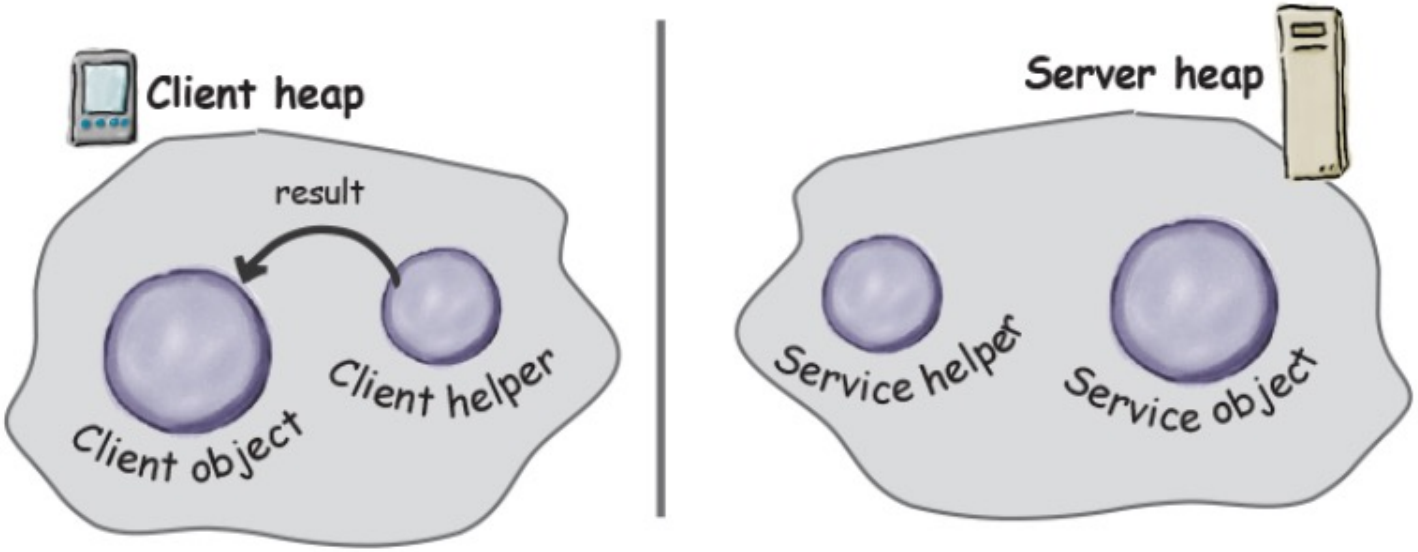
How the method calls happen (step 4)



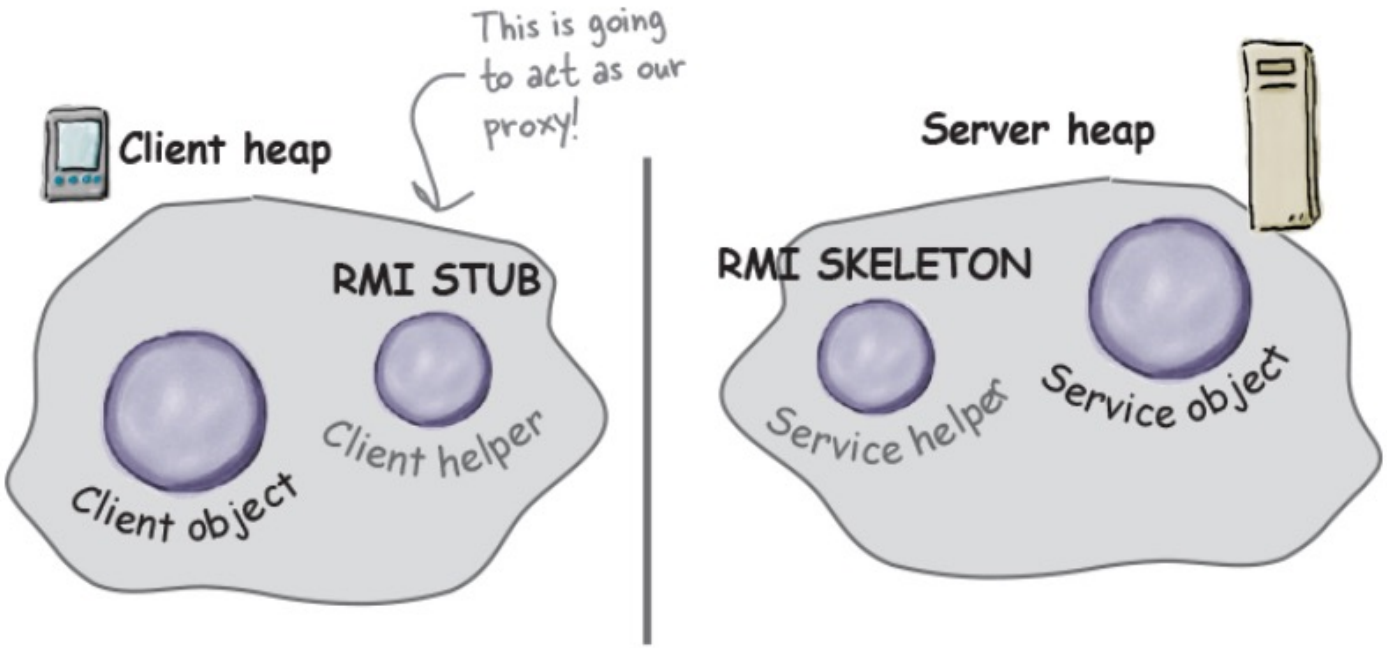
How the method calls happen (step 5)



How the method calls happen (step 6)



Java RMI, the Big Picture



Making the Remote Service

- Step 1 – Make a Remote Interface
- Step 2 – Make a Remote Implementation
- Step 3 – Start the RMI Registry
- Step 4 – Start the Remote Service
- Step 5 – Make a Remote Method Invocation!

Voila!

Step 1.1 – Remote Interface

Extend java.rmi.Remote

```
public interface MyRemote extends Remote {
```

This tells us that the interface is going to be used to support remote calls.

Step 1.2 – Remote Interface

Declare that all methods throw RemoteException

```
import java.rmi.*;

public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Remote interface is in java.rmi.

Every remote method call is considered "risky." Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

Step 1.3 – Remote Interface

Be sure arguments and return values are primitives or Serializable

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

Step 2.1 – Remote Implementation

Implement the Remote Interface

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

Step 2.2 – Remote Implementation

Extend UnicastRemoteObject

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;
```

UnicastRemoteObject implements Serializable, so we need the serialVersionUID field.

Step 2.3 – Remote Implementation

Write a no-arg constructor that declares RemoteException

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

Step 2.4 – Remote Implementation

Register the service with RMI registry

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

Step 3 – RMI Registry

Start RMI Registry

```
File  Edit  Window  Help  Huh?
%rmiregistry
```

Step 4 – Start the Service

Start the Service

File Edit Window Help Huh?

```
%java MyRemoteImpl
```

The Remote Service (interface)

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

RemoteException and the Remote interface are in the java.rmi package.

Your interface MUST extend java.rmi.Remote.

All of your remote methods must declare RemoteException.

The Remote Service (implementation)

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;

    public String sayHello() {
        return "Server says, 'Hey'";
    }


    public MyRemoteImpl() throws RemoteException { }


    public static void main (String[] args) {
        try {
            MyRemote service = new MyRemoteImpl();
            Naming.rebind("RemoteHello", service);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

UnicastRemoteObject is in the java.rmi.server package.

Extending UnicastRemoteObject is the easiest way to make a remote object.

You MUST implement your remote interface!!

You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

Your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor).

Make the remote object, then "bind" it to the rmiregistry using the static Naming.rebind(). The name you register it under is the name clients will use to look it up in the RMI registry.

The Client – Look up Service

The client always uses the remote interface as the type of the service. In fact, the client never needs to know the actual class name of your remote service.

lookup() is a static method of the Naming class.

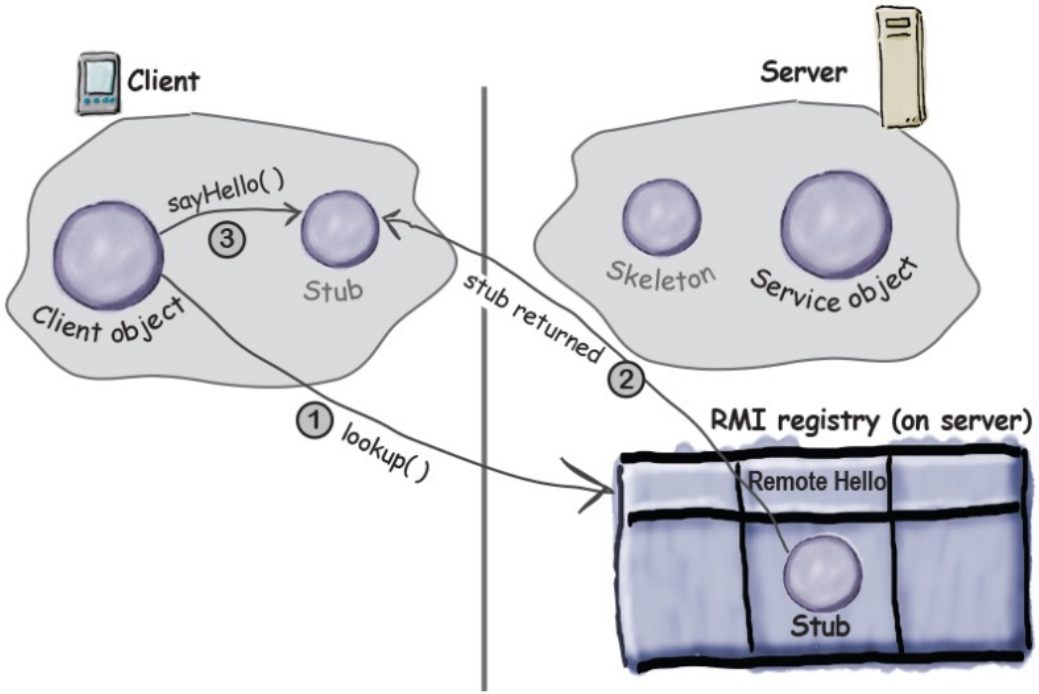
This must be the name that the service was registered under.

```
MyRemote service =
    (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

You have to cast it to the interface, since the lookup method returns type Object.

The host name or IP address where the service is running. (127.0.0.1 is localhost.)

The Client – Look up Service



The Complete Client

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

   public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

The Naming class (for doing the rmiregistry lookup) is in the java.rmi package.

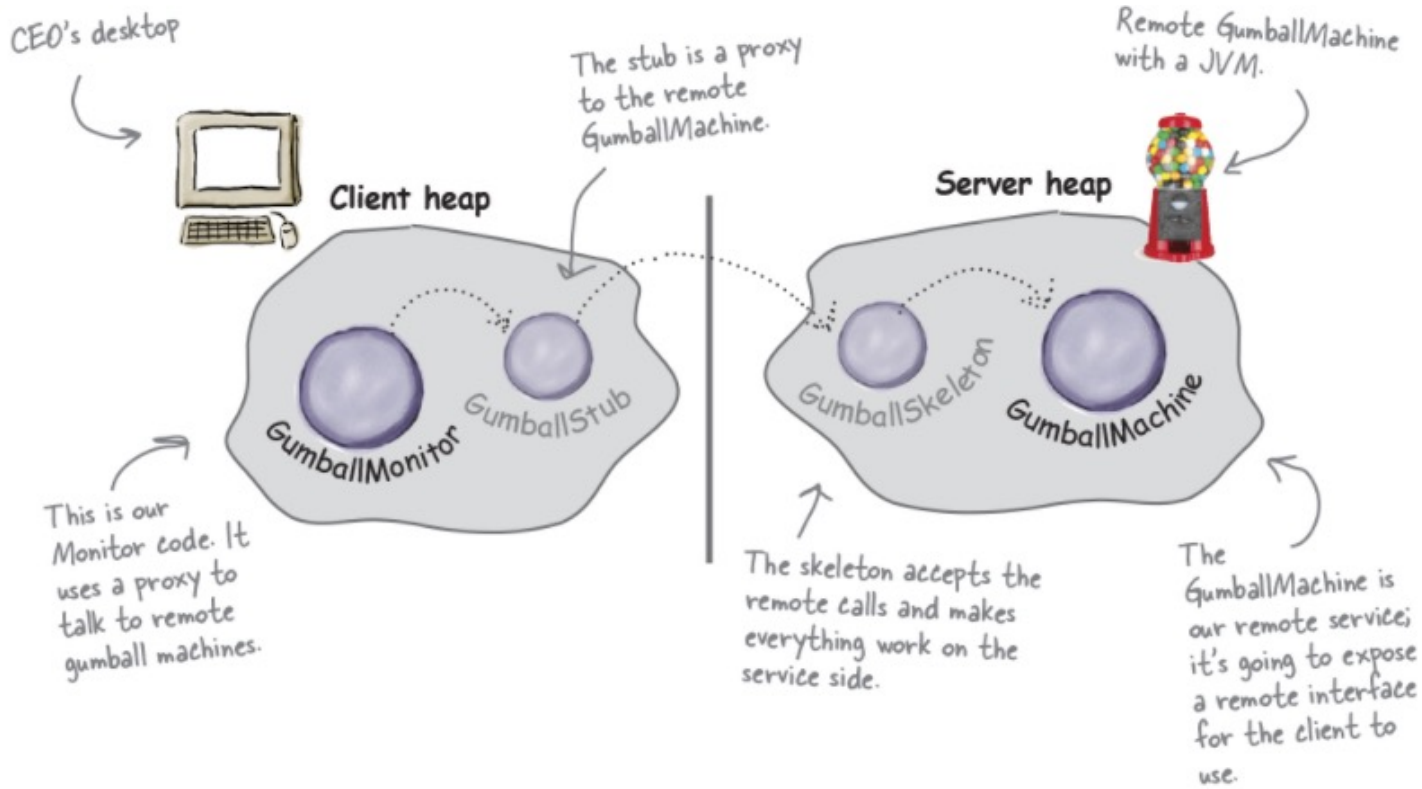
It comes out of the registry as type Object, so don't forget the cast.

You need the IP address or hostname...

...and the name used to bind/rebind the service.

It looks just like a regular old method call! (Except it must acknowledge the RemoteException.)

Back to the Gumball Machine



Remote Interface

Don't forget to import java.rmi.*

This is the remote interface.

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

All return types need to be primitive or Serializable...

Here are the methods we're going to support. Each one throws RemoteException.

Serializable Return Type

```
import java.io.*;
```

Serializable is in the java.io package.

```
public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

Then we just extend the Serializable interface (which has no methods in it). And now State in all the subclasses can be transferred over the network.

Serialization Hint

```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;
    // all other methods here
}
```

In each implementation of State, we add the serialVersionUID and the transient keyword to the GumballMachine instance variable. The transient keyword tells the JVM not to serialize this field. Note that this can be slightly dangerous if you try to access this field once the object's been serialized and transferred.

Completed Gumball Service

First, we need to import the RMI packages.

GumballMachine is going to subclass the UnicastRemoteObject; this gives it the ability to act as a remote service.

GumballMachine also needs to implement the remote interface...

```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    private static final long serialVersionUID = 2L;
    // other instance variables here

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // code here
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }
    // other methods here
}
```

...and the constructor needs to throw a remote exception, because the superclass does.

That's it! Nothing changes here at all!

Registering Gumball Service

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine = new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

First we need to add a try/catch block around the gumball instantiation because our constructor can now throw exceptions.

We also add the call to Naming.rebind, which publishes the GumballMachine stub under the name gumballmachine.

The Gumball Monitor

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

We need to import the RMI package because we are using the RemoteException class below...

Now we're going to rely on the remote interface rather than the concrete GumballMachine class.

We also need to catch any remote exceptions that might happen as we try to invoke methods that are ultimately happening over the network.

Test Drive

Here's the monitor test drive. The CEO is going to run this!

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://austin.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0; i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for (int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

Here's all the locations we're going to monitor.

We create an array of locations, one for each machine.

We also create an array of monitors.

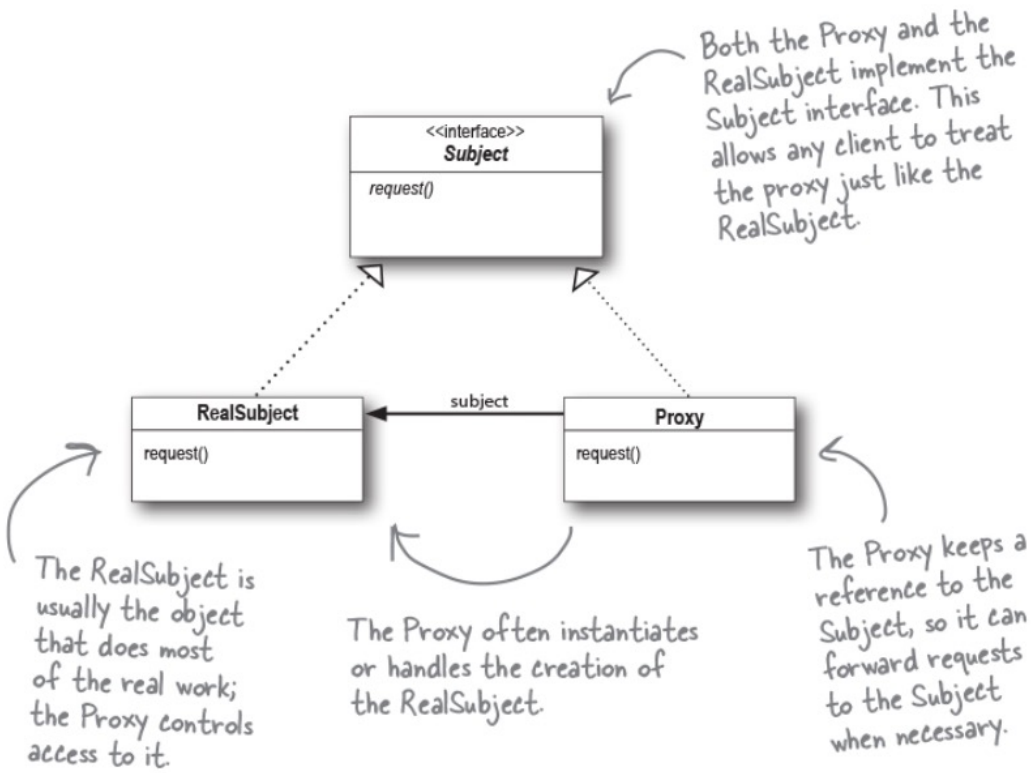
Now we need to get a proxy to each remote machine.

Then we iterate through each machine and print out its report.

The Proxy Pattern Defined

Provides a surrogate or placeholder for another object to control access to it.

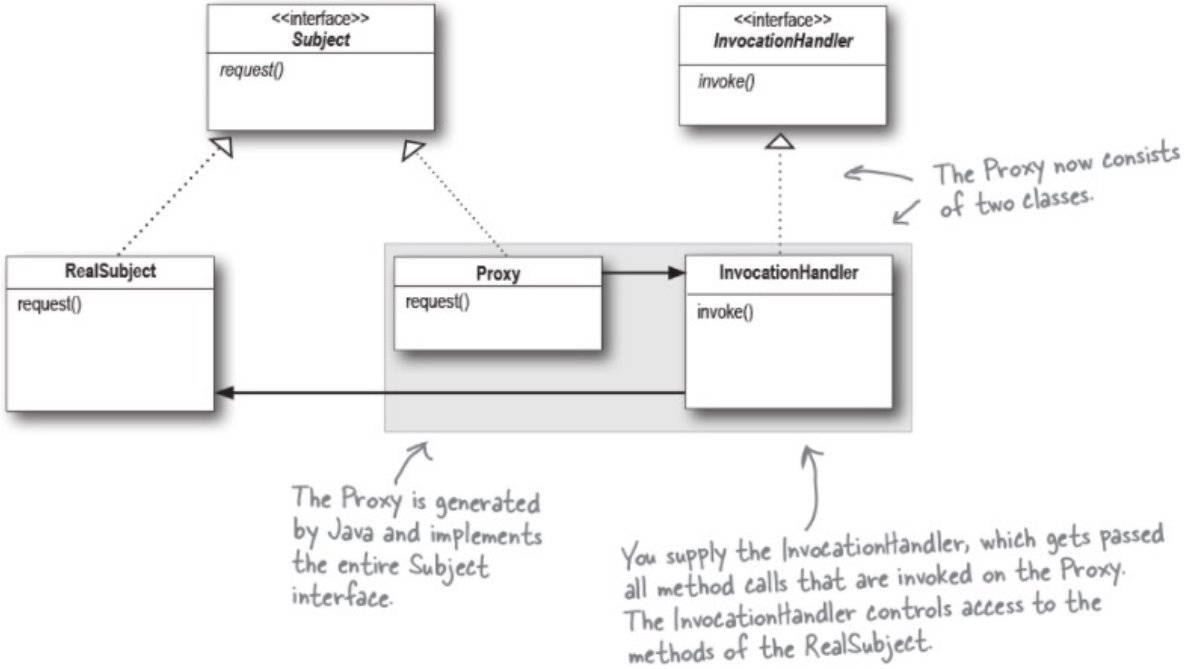
The Proxy Pattern UML



Different Types of Proxy

- Type 1 – Remote Proxy: Controls access to remote object
- Type 2 – Virtual Proxy: Control access to expensive to create object
- Type 3 – Protection Proxy: Control access, based on access rights

Java Dynamic Proxy



The Person Interface

This is the interface; we'll get to the implementation in just a sec...

Here we can get information about the person's name, gender, interests, and Geek rating (1–10).

```
public interface Person {

    String getName();
    String getGender();
    String getInterests();
    int getGeekRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setGeekRating(int rating);

}
```

We can also set the same information through the respective method calls.

setGeekRating() takes an integer and adds it to the running average for this person.

The Person Implementation

```
public class PersonImpl implements Person {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getGeekRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }


    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setGeekRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

The instance variables.

All the getter methods; they each return the appropriate instance variable...

...except for getGeekRating(), which computes the average of the ratings by dividing the ratings by the ratingCount.

And here's all the setter methods, which set the corresponding instance variable.

Finally, the setGeekRating() method increments the total ratingCount and adds the rating to the running total.

Invocation Handlers

1 Let's say the setGeekRating() method is called on the proxy.

```
proxy.setGeekRating(9);
```

2 The proxy then turns around and calls invoke() on the InvocationHandler.

```
invoke(Object proxy, Method method, Object[] args)
```

The Method class, part of the reflection API, tells us what method was called on the proxy via its getName() method.

Here's how we invoke the method on the RealSubject.

3 The handler decides what it should do with the request and possibly forwards it on to the RealSubject. How does the handler decide? We'll find out next.

```
return method.invoke(person, args);
```

Here we invoke the original method that was called on the proxy. This object was passed to us in the invoke call.

Only now we invoke it on the RealSubject...

...with the original arguments.

Owner Invocation Handler

```
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    Person person;

    public OwnerInvocationHandler(Person person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setGeekRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

InvocationHandler is part of the java.lang.reflect package, so we need to import it.

All invocation handlers implement the InvocationHandler interface.

We're passed the RealSubject in the constructor and we keep a reference to it.

Here's the invoke() method that gets called every time a method is invoked on the proxy.

If the method is a getter, we go ahead and invoke it on the real subject.

Otherwise, if it is the setGeekRating() method we disallow it by throwing IllegalAccessException.

This will happen if the real subject throws an exception.

Because we are the owner, any other set method is fine and we go ahead and invoke it on the real subject.

If any other method is called, we're just going to return null rather than take a chance.

Calling the dynamic proxy…

This method takes a Person object (the real subject) and returns a proxy for it. Because the proxy has the same interface as the subject, we return a Person.

This code creates the proxy. Now this is some mighty ugly code, so let's step through it carefully.

To create a proxy we use the static newProxyInstance() method on the Proxy class.

```
Person getOwnerProxy(Person person) {

    return (Person) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person));
}
```

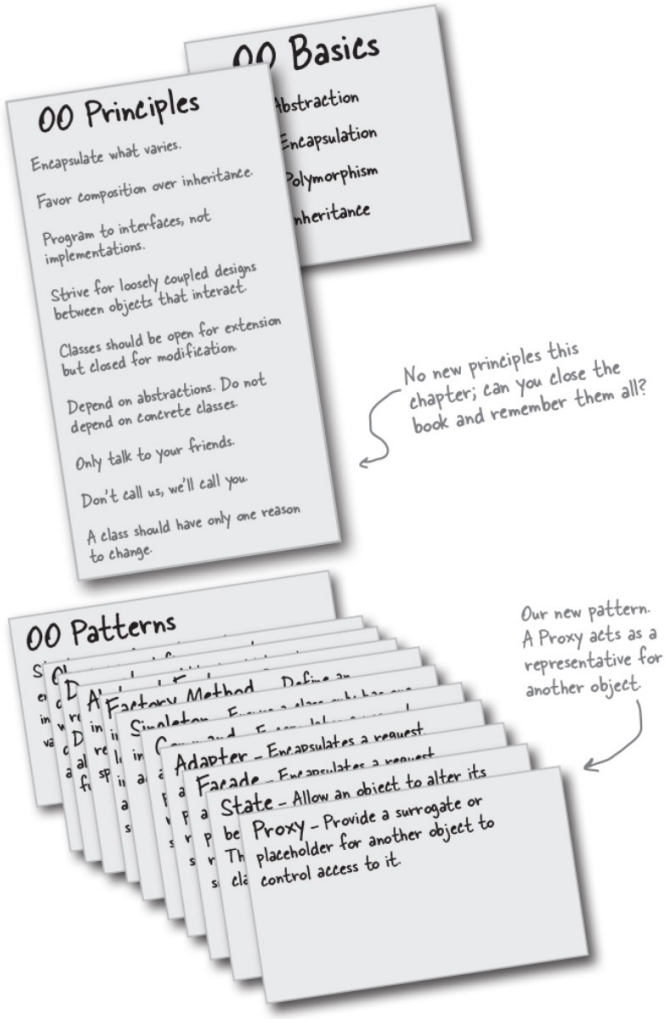
We pass it the class loader for our subject…

…and the set of interfaces the proxy needs to implement…

We pass the real subject into the constructor of the invocation handler. If you look back two pages, you'll see this is how the handler gets access to the real subject.

…and an invocation handler, in this case our OwnerInvocationHandler.

Summary


OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
No new principles this chapter; can you close the book and remember them all?
OO Patterns
Factory Method
Adapter – Encapsulates a request
Facade – Encapsulates a request
State – Allow an object to alter its
Proxy – Provide a surrogate or placeholder for another object to control access to it.
Our new pattern. A Proxy acts as a representative for another object.