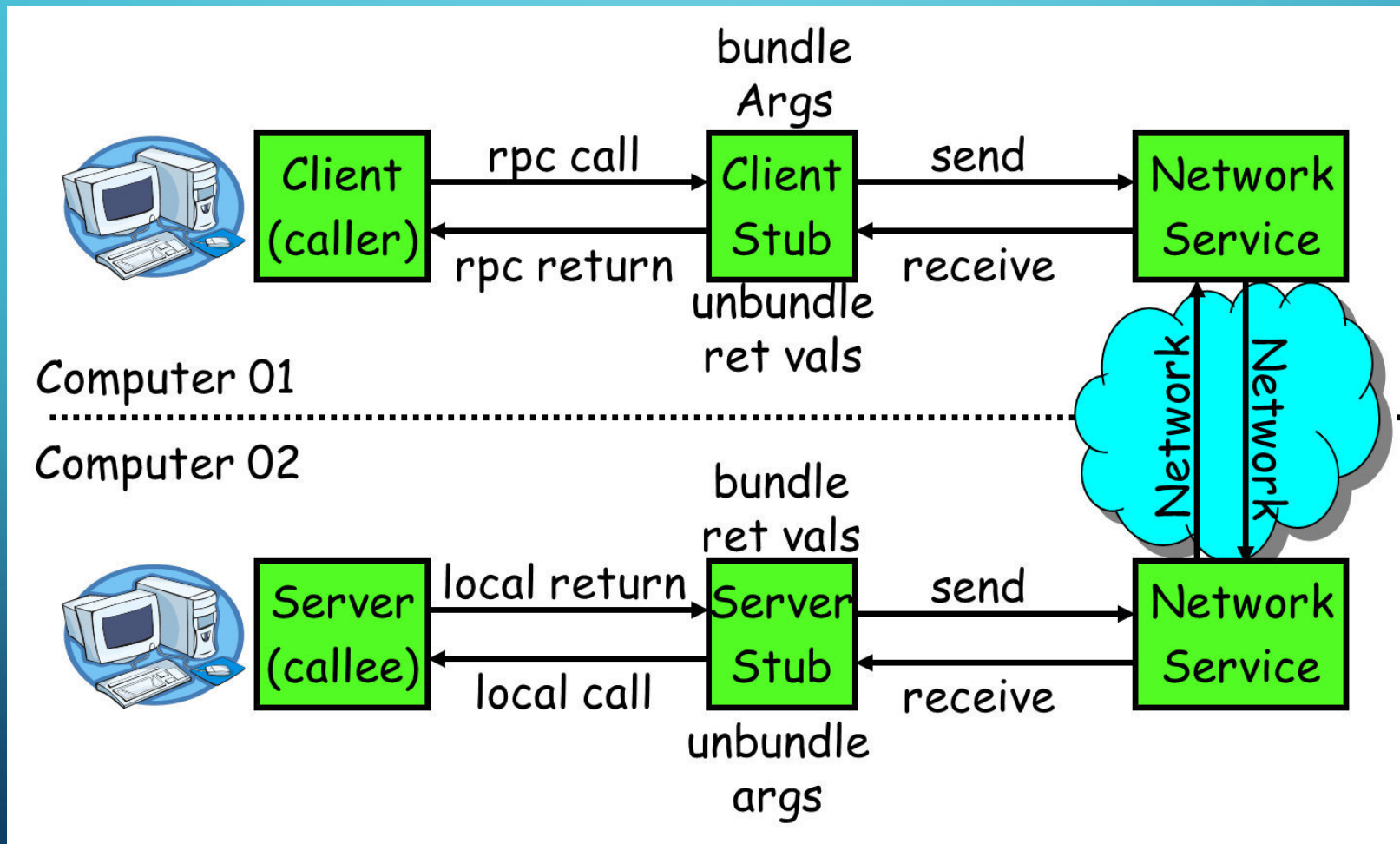


A decorative graphic on the left side of the slide, consisting of a network of white lines and small circles on a blue gradient background, resembling a circuit board or a neural network.

# DISTRIBUTED AND CLOUD COMPUTING

LAB4 JAVA DYNAMIC PROXY IN RMI

# WORKFLOW OF RPC CALLS



# GOOD DESIGN VS. BAD DESIGN

- A user-friendly RPC/RMI framework should be able to hide all low-level implementations to user.



```
public static void main(String[] args) {
```

```
    RpcThingInterface stub = RpcService.lookup("service_name");
```

*1. Get access to the remote object by name*

```
    float userParam = 123.45;
```

```
    int result = stub.userMethod(userParam);
```

*2. Invoke the remote method*

```
} // good design
```



```
public static void main(String[] args) {
```

```
    String remotep = RpcService.findIp("service_name");
```

*1. Get the IP address*

```
    int remotePort = RpcService.findPort("service_name");
```

*2. Get the port number*

```
    RpcClient client = new RpcClient(remotep, remotePort);
```

*3. Create a client*

```
    float userParam = 123.45;
```

```
    String result = client.sendInvokeParam("userMethod", new String[]{Float.toString(userParam)});
```

*4. Send the arguments via client and get the return as a string*

```
    int realResult = Integer.parseInt(result);
```

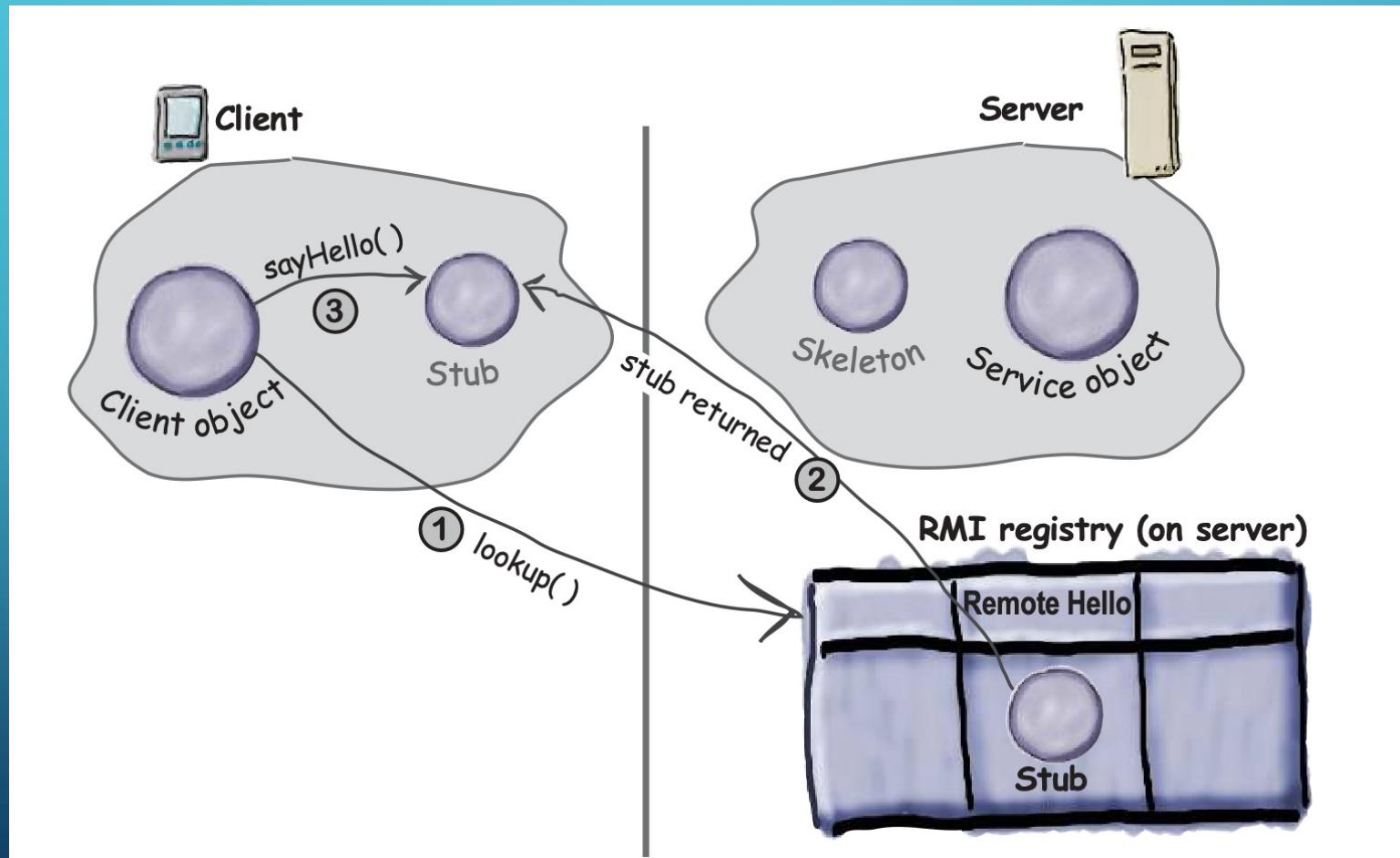
*5. Parse the returned string to get the result*

```
} // bad design
```

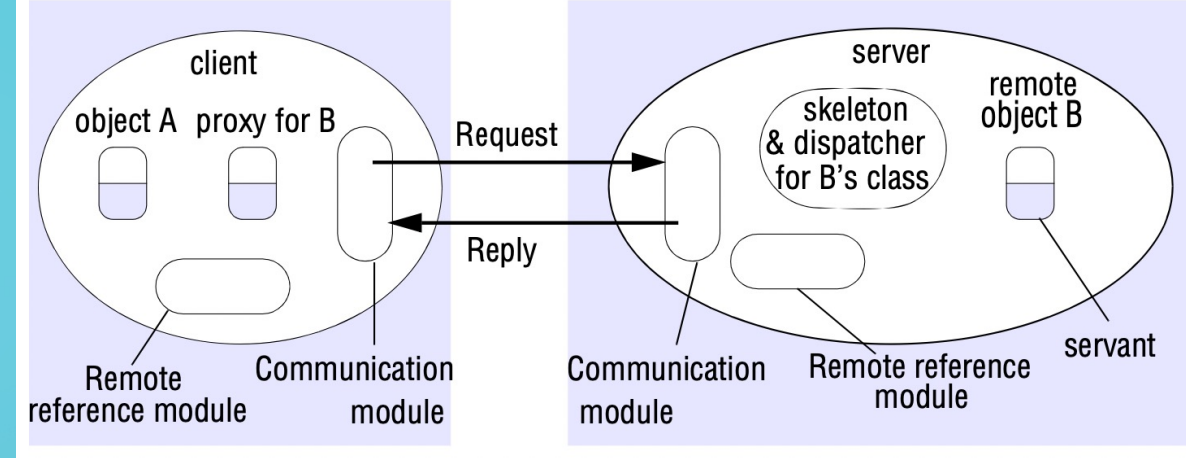


# WORKFLOW OF RMI

From the perspective  
of the client



# PROXY

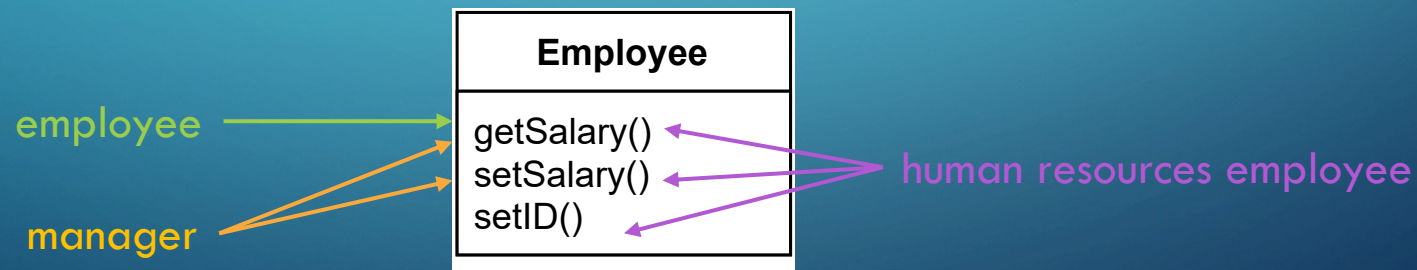


- **The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.
  - In simple words, a proxy is a class functioning as an interface to something else.
  - A proxy is a **wrapper or agent object** that is being called by the client to access the real serving object behind the scenes.
- For the client, **usage of a proxy object is like using the real object**, because they both implement the same interface.
- Extra functionality can be provided by the proxy:
  - Caching when operations on the real object are resource intensive
  - Checking preconditions before operations on the real object are invoked

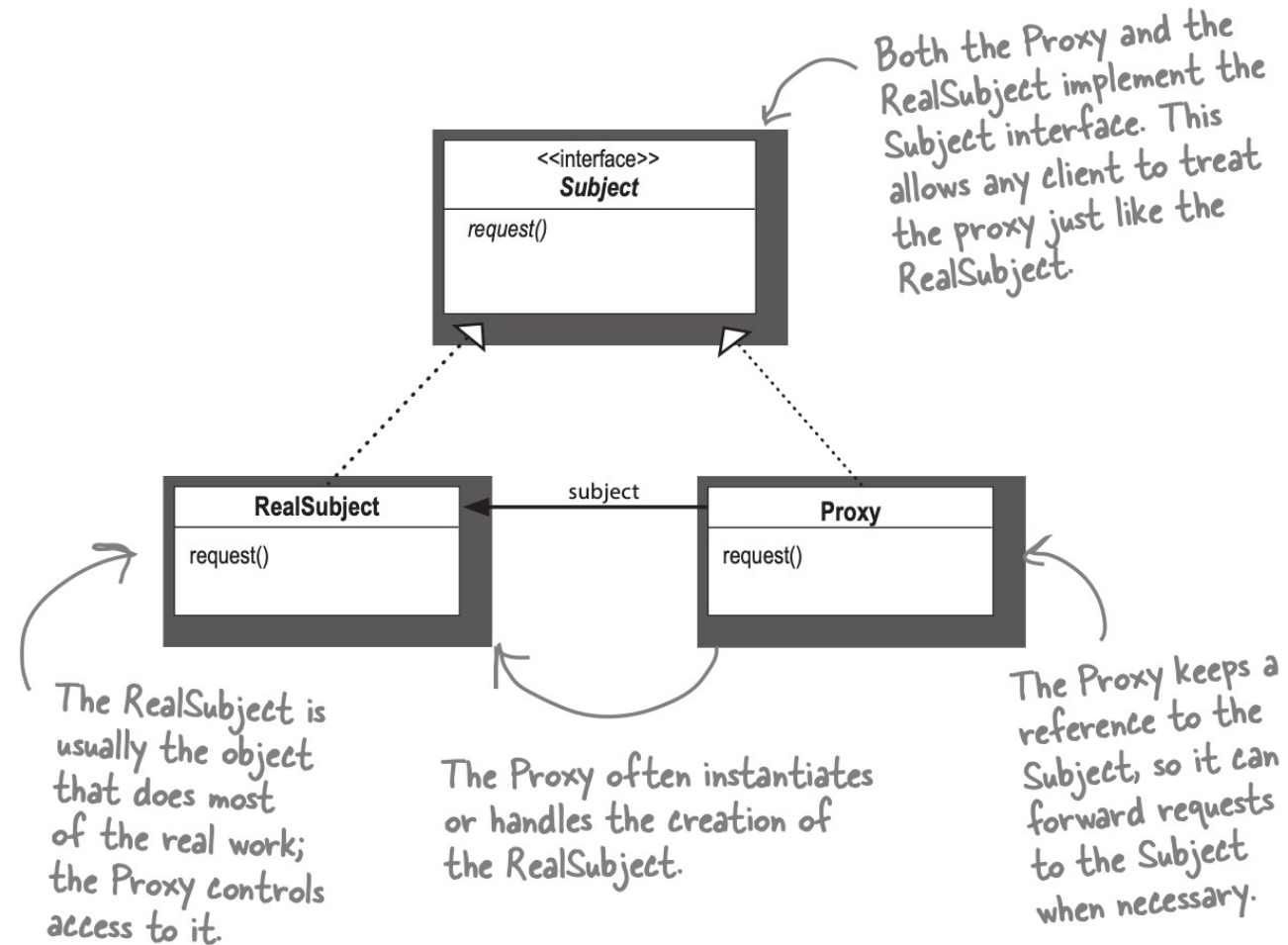


# PROXY

- **Protection Proxy:** controlling access to an object based on access rights
  - For instance, if we had an employee object, a Protection Proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like `setSalary()`), and a human resources employee to call any method on the object.



# PROXY PATTERN - EXAMPLE



# PROXY PATTERN - EXAMPLE

```
public interface Subject {  
    public void request();  
}
```

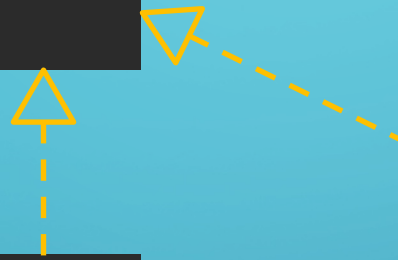
Subject interface

```
public class RealSubject implements Subject{  
    @Override  
    public void request() {  
        System.out.println("requested");  
    }  
}
```

Implementation (proxy target)

```
public class Proxy implements Subject{  
    private Subject target;  
    public Proxy(Subject target) {  
        this.target = target;  
    }  
  
    @Override  
    public void request() {  
        // extensions to save()  
        System.out.println("start requesting");  
        target.request();  
        System.out.println("request complete");  
    }  
}
```

Proxy



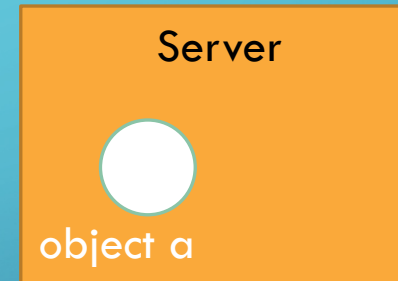
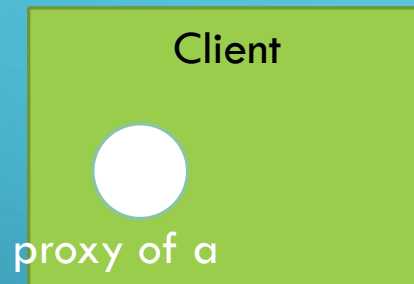


# PROXY PATTERN - EXAMPLE

- What if objects with new interfaces are implemented in the server?

```
class ProxyApple  
implements IApple  
{  
  ...  
}
```

Before

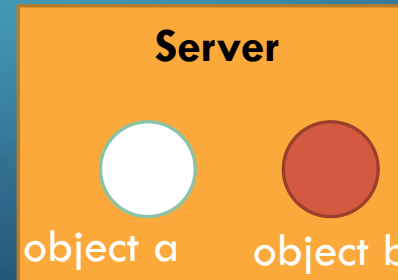
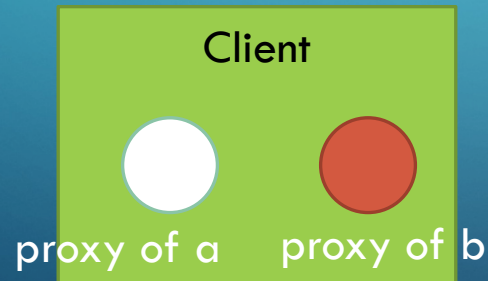


`IApple a = new Apple();`

```
class ProxyApple  
implements IApple
```

```
class ProxyBanana  
implements IBanana  
{  
  ...  
}
```

After



New interface *IBanana* and class *Banana* are added.  
`IBanana b = new Banana();`

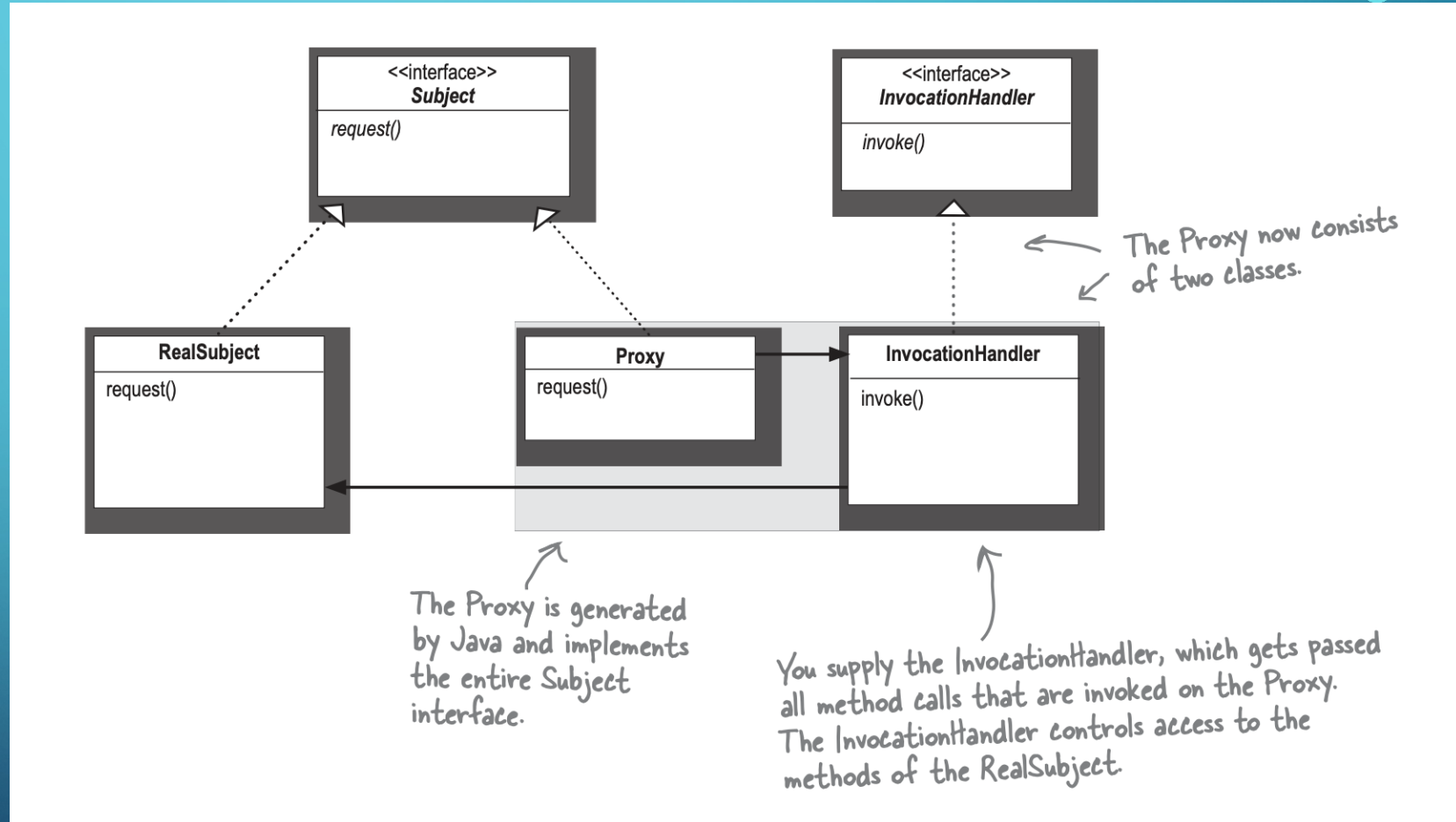
- We need to write a new proxy class (e.g. called *BananaProxy*) that implements *IBanana*.

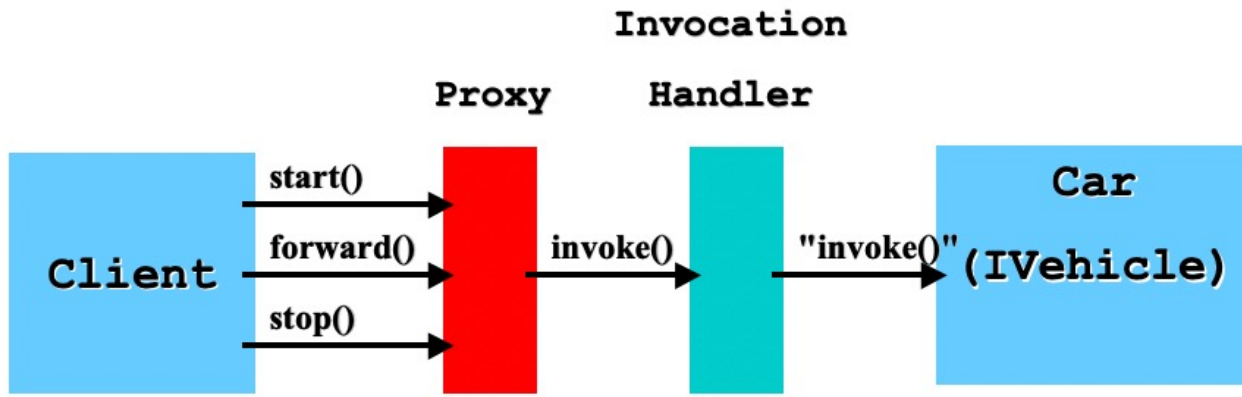
# DYNAMIC PROXY

Java's got its own proxy support right in the `java.lang.reflect` package. With this package, Java lets you create a proxy class *on the fly* that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this Java technology as a *dynamic proxy*.

You don't write the Proxy class. The Proxy class is dynamically created on demand from the set of interfaces you pass it.

InvocationHandlers implement the behaviour of the proxy.





## Interface

```

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

```

## Interface implementation

```

/**
 * 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.
}

```

## 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);
    }
}

```

*The invoke() method does the invocation on the real object*

## Client

```

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();
    }
}

```

*Dynamic creation of an IVehicle proxy at runtime*



# DYNAMIC PROXY

- A dynamic proxy class is a class that implements a list of interfaces specified at **runtime** such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to **another object through a uniform interface**. Thus, a dynamic proxy class can be used to create a **type-safe proxy object for a list of interfaces without requiring pre-generation of the proxy class**, such as with compile-time tools. Method invocations on an instance of a dynamic proxy class are dispatched to a single method in the instance's **invocation handler**, and they are encoded with a `java.lang.reflect.Method` object identifying the method that was invoked and an array of type `Object` containing the arguments.

<https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html>

# INVOCATION HANDLER

- InvocationHandler is the interface implemented by the invocation handler of a proxy instance.
- 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**.
- Object invoke(Object proxy, Method method, Object[] args) throws Throwable
  - proxy - the **proxy instance** that the method was invoked on
  - method - the **Method instance corresponding to the interface method** invoked on the proxy instance. The declaring class of the Method object will be the interface that the method was declared in, which may be a superinterface of the proxy interface that the proxy class inherits the method through.
  - args - **an array of objects containing the values of the arguments** passed in the method invocation on the proxy instance, or null if interface method takes no arguments. Arguments of primitive types are wrapped in instances of the appropriate primitive wrapper class, such as java.lang.Integer or java.lang.Boolean.

<https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html>



# PROXY INSTANCE

- `public static Object newInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)` throws `IllegalArgumentException`
  - loader - the **class loader** to define the proxy class
  - interfaces - the list of **interfaces** for the proxy class to implement
  - h - the **invocation handler** to dispatch method invocations to
- Returns: **a proxy instance** with the specified invocation handler of a proxy class that is defined by the specified class loader and that implements the specified interfaces
- You don't write the Proxy class. The Proxy class is dynamically created on demand from the set of interfaces you pass it.



# Another Example

Implementation of the invoke() method in a customised invocation handler

```
public class DebugProxy implements java.lang.reflect.InvocationHandler {  
    private Object obj;  
  
    .....  
  
    private DebugProxy(Object obj) {  
        this.obj = obj;  
    }  
  
    @Override  
    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 invocation exception: " + e.getMessage());  
        } finally {  
            System.out.println("after method " + m.getName());  
        }  
        return result;  
    }  
}
```

Equals to obj.m(args)

```
public interface Foo {  
    Object bar(Object obj) throws BazException;  
}
```

```
public class FooImpl implements Foo {  
    Object bar(Object obj) throws BazException {  
        // ...  
    }  
}
```

```
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));  
    }
```

*Dynamically create a proxy of obj*

*We plug in our customised invocation handler*

```
    private DebugProxy(Object obj) {  
        this.obj = obj;  
    }
```

```
    @Override
```

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

```
    public static void main(String[] args) {  
        Foo foo = (Foo) DebugProxy.newInstance(new FooImpl());  
        foo.bar(null);  
    }
```

# PRACTICE

- Try to create a proxy instance of an **interface**.

```
public interface IHelloWorld {  
    public void hello();  
}
```

- What to expect: Calling method `hello()` will print a line: Hello world!
- Note: You shall not create any implementation class for the interface

Further readings (available on *Blackboard*):

- Chapter 11 the Proxy Pattern, Head First Design Patterns, 2020.