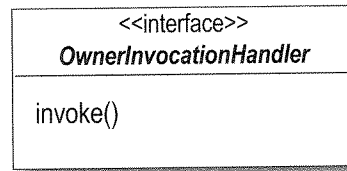
Step one: creating Invocation Handlers

We know we need to write two invocation handlers, one for the owner and one for the non-owner. But what are invocation handlers? Here's the way to think about them: when a method call is made on the proxy, the proxy forwards that call to your invocation handler, but *not* by calling the invocation handler's corresponding method. So, what does it call? Have a look at the InvocationHandler interface:



There's only one method, invoke(), and no matter what methods get called on the proxy, the invoke() method is what gets called on the handler. Let's see how this works:

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

```
proxy.setHotOrNotRating(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.

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.

Here's how we invoke the method on the Real Subject.

```
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.

Creating Invocation Handlers continued...

When invoke() is called by the proxy, how do you know what to do with the call? Typically, you'll examine the method that was called on the proxy and make decisions based on the method's name and possibly its arguments. Let's implement the OwnerInvocationHandler to see how this works:

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

All invocation handlers implement the InvocationHandler interface.

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

public class OwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public OwnerInvocationHandler(PersonBean 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("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

We're passed the Real Subject 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 setHotOrNotRating() method we disallow it by throwing a IllegalAccessException.

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

This will happen if the real subject throws an exception.

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