

Summary: Using Reflection To Break Encapsulation

Programs try to insulate the internal structure of their classes from their public interfaces by implementing the principle of encapsulation.

Using Reflection, we can not only find out about a class's internal structure; we can access and modify their private properties as well.

This presentation will give a cursory overview of the principle of encapsulation.

It will also show basic examples of how to access and modify private variables of another class using reflection, and how to invoke private functions of another class.

*source code: All the slides that use main(), are in the Main.java. There are commented out sections for each different main() that were used in the slides

Breaking Encapsulation By Using Reflection

by
Drew Goldberg

reference: http://en.wikipedia.org/wiki/File:Russian_Dolls.jpg

What Is Encapsulation?

"Encapsulation: Encapsulation refers to a set of language-level mechanisms or design techniques that hide implementation details of a class, module, or subsystem from other classes, modules, and subsystems."

reference: <http://www.cs.colorado.edu/~kena/classes/5448/f12/lectures/02-ooparadigm.pdf>

Why Is Encapsulation Important?

Encapsulation enables a class to modify its internal state without having to modify its public interface.

This means that a class's internal structure can be changed without breaking other classes using its public interface.

Encapsulation also protects the class from other classes modifying its internal state.

reference: <http://stackoverflow.com/questions/3982844/encapsulation-well-designed-class>

Simple Example Of Encapsulations

Some basic types of encapsulation are making variables private.

Allowing access to these variables via getter and setter functions.

What Isn't Encapsulation

Sometimes it is helpful to define something by what it isn't. For clarity, here is an example of what isn't encapsulation in regards to programming.



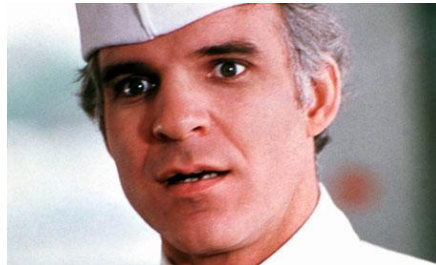
How Old Is Steve Martin?

Let's look at the class SteveMartinHumanAutomaton to see some basic examples of encapsulation

1977



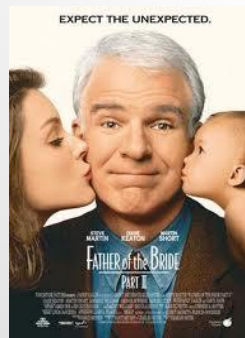
1979



1986



1995



2009



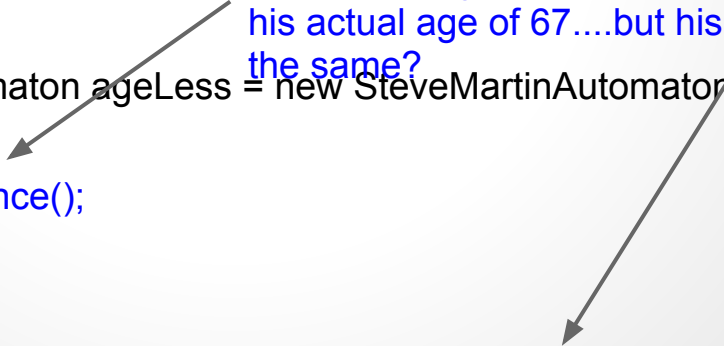
SteveMartinAutomaton

We have a simple program that instantiates a SteveMartinHumanAutomaton object.

The object ageLess displays the age/appearance of Steve Martin upon its construction.

```
public static void main(String args[]){  
    SteveMartinHumanAutomaton ageLess = new SteveMartinAutomaton();  
    ageLess.setAge(67);  
    ageLess.displayAppearance();  
}
```

Wait...I changed SteveMartin's age to his actual age of 67....but his age is still the same?



output:

Steve Martin has been 57 years old since 1977!

Steve Martin has been 57 years old since 1977!

Encapsulation Part II

SteveMartinHumanAutomaton.java

```
public class SteveMartinHumanAutomaton {
    private static int ageFake = 57;
    private String secretFoodSource = "The Tears Of Children";
    private String hoursOfSleep = "Steve Martin Doesn't Sleep. He is an automaton";
    public String ageMessage = "Steve Martin has been " + ageFake + " years old since 1977!";
    public SteveMartinHumanAutomaton() {
        displayAppearance();
    }
    public void setAge(int age){
        if(age <= 55)
            this.ageFake = 55;
        else if(age >= 60)
            this.ageFake = 60;
    }
    public int getAgeFake(){
        return this.ageFake;
    }
    public void displayAppearance() {
        System.out.println(ageMessage);
    }
}
```



we have hidden some of our class's variables/fields as private.



Not only is the variable private, it has to be accessed via the class's public getters and setters methods. There is some simple error checking for setAge(). This basic validation ensures that Steve Martin's ageFake value is never greater than 60 nor smaller than 55.

Encapsulation Examples Recap

SteveMartinHumanAutomaton Class

In the previous example, another class can't access the variable `ageFake` directly because it has a private modifier.

To access the private variable, `ageFake`, you must go through `setAgeFake()`, which does error checking.

The `SteveMartinHumanAutomaton` class could change the data structure for `ageFake` from an `int` to some else, without breaking other classes' code that were already using its public method `getAgeFake()`.

In another words the internal structure of the `SteveMartinHumanAutomaton` class can change without other classes knowing it has been changed.

Changes to the internals of the `SteveMartinHumanAutomaton` can be made without breaking other classes's code.

Other class can't accidentally modify the `SteveMartinHumanAutomaton`'s internal structure.... or can they?

What's Reflection?

Be careful when using reflection, it can lead you down some strange paths.



REFLECTION DEFINITION

"Reflection is the ability of a **computer program** to examine (see **type introspection**) and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at **runtime**"

reference: [http://en.wikipedia.org/wiki/Reflection_\(computer_programming\)](http://en.wikipedia.org/wiki/Reflection_(computer_programming))



REFLECTION DEFINITION CONT'D

"In object oriented programming languages such as **Java**, reflection allows **inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods.**"

reference: [http://en.wikipedia.org/wiki/Reflection_\(computer_programming\)](http://en.wikipedia.org/wiki/Reflection_(computer_programming))

JAVA'S REFLECTION OVERVIEW

We will examine the reflection capabilities in the Java language.

Java's reflection's capabilities are stored in the `java.lang.reflect` package.

"Also the Java runtime system always maintains what is called runtime type identification on all objects.

This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute."

reference: Horstmann, Cay S., Gary Cornell, and Cay S. Horstmann. *Core Java*. Vol. 1. Upper Saddle River, NJ: Prentice Hall/Sun Microsystems, 2008. Print.

Java Reflection's Class Class

There is a Java Class called Class. It is typically used to get the name a of class via the getName()

The getName() gets the name of the more specific name of the object. In our example, it prints out that Rick Roll is a manager, which is a subclass of Employee.

```
public class Main {  
    public static void main(String args[]) {  
        Employee e[] = new Employee[] { new Employee("Billy Bob"), new Manager("Rick Roll")};  
        System.out.println("The name of the employee e is " + e[0].getName());  
        Class cl0 = e[0].getClass();  
        System.out.println("The name of the class of e[0] is " + cl0.getName());  
        System.out.println("The name of the employee e is " + e[1].getName());  
        Class cl1 = e[1].getClass();  
        System.out.println("The name of the class of e[1] is " + cl1.getName());  
    }  
}
```

output:

```
The name of the employee e is Billy Bob  
The name of the class of e[0] is  
Employee  
The name of the employee e is Rick Roll  
The name of the class of e[1] is Manager
```

Using Reflection To Analyze The Capabilities of Classes

We are going to look at 2 classes in the `java.lang.reflect` package: (`Field`, `Method`).

These classes allow for the discovery of another class's internal data structures, and methods.

Each of these classes have a `getName()` that returns the name of each item they are investigating.

reference: Horstmann, Cay S., Gary Cornell, and Cay S. Horstmann. *Core Java*. Vol. 1. Upper Saddle River, NJ: Prentice Hall/Sun Microsystems, 2008. Print.

Java's Field Class Capabilities Formally Speaking

"A Field provides information about, and dynamic access to, a single field of a class or an interface. The reflected field may be a class (static) field or an instance field."

reference: <http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Field.html>

The Java Field Class Capabilities Informally Speaking

Encapsulation is about not exposing the internal structure of an object/class to the outside world.

With the `java.lang.reflect.Field` class, we can easily find out about all of a class's fields (variables) and each field type.

We will look at a simple class, `Employee`, and in a separate class discover all of its hidden treasure.

Or as a Flock Of Smeagols would Say, "The Preciousses"

FLOCK OF SMEAGOLS

Let's Use The Field Class To Find Out All Of Employee's Variables

```
import java.lang.reflect.Field;
public class Main {
    public static void main (String args[]) throws NoSuchFieldException, SecurityException{
        Field field[] = Employee.class.getDeclaredFields();
        int i = 0;
        while(i < field.length) {
            System.out.println("Employee contains the field " + field[i].getName() + " which is
a/an " + field[i].getType());
            i++;
        }
    }
}
```

 `getDeclaredFields()` will get all field types.

`getFields()` will only return public fields

Output:

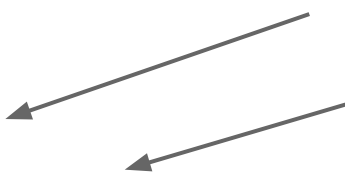
Employee contains the field num which is a/an int

Employee contains the field id which is a/an interface java.util.Map

Employee.java

```
import java.util.*;
public class Employee {
    private static int num = 1;
    private Map<String, Integer> id = new HashMap<String, Integer>();
    public void setId(String lastName){
        this.id.put(lastName, num);
        num++;
    }
    public int getId(String lastName){
        return this.id.get(lastName);
    }
}
```

Both field members we found out about, were also private!



Using Reflection To Find Field Parameter Types

We can even find out about the field's parameters for more complex data structures, such as Lists and Maps.

Let's use reflection to figure out Employee's id variable's parameters!

We already know that id is a type of Map, let's find out it's key/value pair type.

```
import java.lang.reflect.Field;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
public class Main {
    public static void main (String args[]) throws NoSuchFieldException, SecurityException{
        Field idField = Employee.class.getDeclaredField("id");
        ParameterizedType params = (ParameterizedType)idField.getGenericType();
        Type[] paramsType= params.getActualTypeArguments();
        int i = 0;
        while (i < paramsType.length){
            System.out.println("The type of the first parameter is " + paramsType[i].toString());
            i++;
        }
    }
}
```

Must use `getDeclaredField()` to retrieve a private variable.

output:

The type of the first parameter is class java.lang.String

The type of the first parameter is class java.lang.Integer

Using Reflection To Find Field Parameter Types Continued

Why could this be bad?

We can now make assumptions about the id's parameter types.

What if the id attribute data structure from a `Map<String, Integer>` to an `ArrayList<Integer>`?

Then any code based on the assumption of the structure of id, could easily break if the structure of the id variable is changed.

This is because we violated the principle of encapsulation.


Using Java's Field Class To Access And Modify Private Variables

What are other ways we could break the principle of encapsulation now that we know some of the class's internal structure?

How about accessing and modifying private variables directly!

THE CLASS MrCONSTANT

```
public class MrConstant {  
    private String name = "Mr.Constant";  
    public void printName() {  
        System.out.println("Hello my name is " + name);  
    }  
}
```



We are going to access and modify MrConstant private variable name.

Changing Mr.Constant's Name Variable

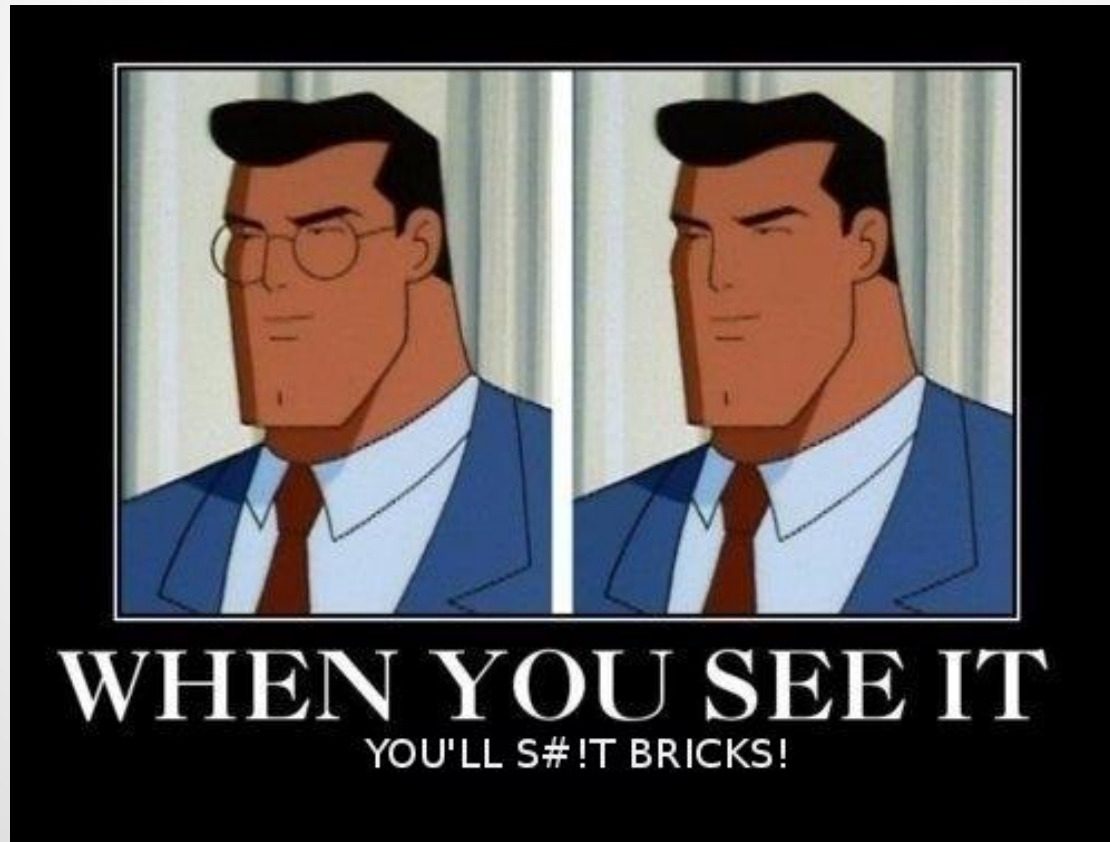
```
import java.lang.reflect.Field;
public class Main {
    public static void main (String args[]) throws NoSuchFieldException,
SecurityException, IllegalArgumentException, IllegalAccessException{
        MrConstant mrConstant = new MrConstant();
        mrConstant.printName();
        Field field = mrConstant.getClass().getDeclaredField("name");
        field.setAccessible(true);
        field.set(mrConstant, "NotSoMuch");
        mrConstant.printName();
    }
}
```

output:

Hello my name is Mr.Constant
Hello my name is NotSoMuch

Using Java Method Class To Break Encapsulation

We can also investigate methods inside a class. These include private methods!



For the method section on reflection, we are going to try enter the Mines of Moria two different ways.

THE LEGITIMATE WAY

THE SNEAKY REFLECTION WAY

Legitimately Get into Moria by Guessing the Password

```
public static void main(String args[]) {  
    HiddenMethods hm = new HiddenMethods();  
    while (!hm.getAnswerFlag()) {  
        hm.askPassword();  
        try {  
            BufferedReader bufferRead = new BufferedReader(  
                new InputStreamReader(System.in));  
            String guess = bufferRead.readLine();  
            hm.validatePassword(guess);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

ouput:

The doors of Durin, Lord of Moria, speak friend and enter.

blah

Ask Frodo for some help?

The doors of Durin, Lord of Moria, speak friend and enter.

mellon

The doors Of Durin Opens. Fly you fools!

Getting Into Moria Continued

In our sample program, it looks like we are going to be stuck for a while, unless we know the password, mellon.

Or...

We could investigate the class, `HiddenMethods`, via reflection to figure out the password.

Let's Get All Of The Names Of All Of The Methods In The HiddenMethods class.

```
public static void main(String args[]){
    //Get the Class object associated with this class.
    HiddenMethods hm = new HiddenMethods();
    Class objClass = hm.getClass();

    //Get all of the public methods associated with this class.
    Method[] publicMethods = objClass.getMethods();
    for(Method publicMethod: publicMethods)
        System.out.println("Public method found: " + publicMethod.toString());

    //Get the all methods associated with this class.
    Method[] allMethods = objClass.getDeclaredMethods();
    for(Method method: allMethods)
        System.out.println("Public method found: " + method.toString());
}
```

reference: <http://stackoverflow.com/questions/8524011/java-reflection-how-can-i-get-the-all-getter-methods-of-a-java-class-and-invoke>

Output From Using Reflection

output:

Public method found: public java.lang.Boolean HiddenMethods.

getAnswerFlag()

Public method found: public void HiddenMethods.askPassword()

Public method found: public void HiddenMethods.validatePassword(java.lang.String)

Public method found: public void HiddenMethods.response(java.lang.Boolean)

.....

Method found: private void HiddenMethods.setPassword(java.lang.String)

Method found: private java.lang.String HiddenMethods.getPassword()

Method found: private void HiddenMethods.setAnswerFlag(java.lang.Boolean)

Method found: public java.lang.Boolean HiddenMethods.getAnswerFlag()

Scrolling through the printout, there appears to be two private functions that we could manipulate in order to bypass the application's validation process.

We can infer that their method's signature looks like this.

private void setPassword(String)

private void setAnswerFlag(Boolean)

Let's Change The Password!

Let's use reflection to access
private void setPassword(String)
and change the value of the password
to something we want.

Changing The Password

```
public static void main(String args[]) throws NoSuchMethodException, SecurityException, IllegalAccessException,
IllegalArgumentException, InvocationTargetException{
```

```
    HiddenMethods hm = new HiddenMethods();
```

```
    Class<? extends HiddenMethods> objClass = hm.getClass();
```

Get the Class object associated with this class.

Get the setPassword(String password), you need to provide the class type for each parameter.

In the setPassword(String password), it has a string parameter named guess.

```
    Method setPasswordMethod = objClass.getDeclaredMethod("setPassword", String.class);
```

Next set the method to true, so it is no longer private and therefore we can access it.

```
    setPasswordMethod.setAccessible(true);
```

```
    setPasswordMethod.invoke(hm, "blah");
```

```
    hm.validatePassword("blah");
```

Next invoke the setPassword() using invoke()

The first parameter is an instance of the object that contains the function, hm

The next is the parameter that we are passing to the setPassword() function

reference: <http://stackoverflow.com/questions/6704190/java-invoke-its-own-private-method-with-fix-parameter>

reference: <http://www.coderanch.com/t/532965/java/java/Accessing-methods-primitive-argument-via>

Protect Your Classes' Encapsulation From Reflection

So maybe you wish your program's encapsulation could stay intact by preventing the use of reflection!

Minizing Reflection To Help Ensure Security And Encapsulation

You can do this by changing java's security architecture.

Unfortunately, that can be very difficult because java's security architecture is quite complicated.

Check out the reference below for more specific instructions on how to do that.

reference: <http://stackoverflow.com/questions/770635/disable-java-reflection-for-the-current-thread>

Try To Use Reflection Judiciously

In general violating the principles of Encapsulation is very bad.

One way of doing this is by using reflection.

Keep in mind, reflection is a very useful tool.

It shouldn't be avoided, but rather used appropriately.