Encapsulation

CMSC 202

Types of Programmers

- Class creators
  - Those developing new classes
  - Want to build classes that expose the minimum interface necessary for the ***client program*** and hide everything else
- Client programmers
  - Those who use the classes (a term coined by Scott Meyer)
  - Want to create applications by using a collection of interacting classes

OOP Techniques

- Class creators achieve their goal through ***encapsulation***.
- Encapsulation:
  - Combines data and operations into a single entity (a class)
  - Provides proper access control
  - Focuses on implementation
  - Achieved through ***information hiding*** (abstraction)

The Value of Encapsulation

- Client programmers do not need to know how the class is implemented, ***only how to use it***.
- The information the client programmer needs to use the class is ***kept to a minimum***.
- Class implementation may be changed with ***no impact*** on those who use the class.

Access Control

- Encapsulation is implemented using ***access control***.
  - Separates interface from implementation
  - Provides a boundary for the client programmer
- Visible parts of the class (the ***interface***)
  - Can only be used, not modified, by the client programmer
- Hidden parts of the class (the ***implementation***)
  - Can be changed by the class creator without impacting any of the client programmer's code
  - Can't be corrupted by the client programmer

Access Control in Java

- ***Visibility modifiers*** provide access control to ***data members*** and ***methods***.
  - ***public*** visibility – accessible by everyone, in particular the client programmer
    - A class's interface is defined by its public methods
  - ***private*** visibility – accessible only by the methods within the class
  - Others – later

Car Class

- In this new Car class, the instance variables have been labeled ***private***.

```
public class Car {
    private int horsepower;
    private int numDoors;
    private int year;

    private String vin;
    private String color;
    private String model;
    private String make;

    public String toString() {
        return year + " " + make + " " + model;
    }
    // ...
}
```

Any Car object may use its private members within a method.

Access Control Example

- Original Car class – no visibility modifiers were used
- New Car class – private members attempting to be used

```
public class CarDemo {
    public static void main(String[] args) {
        Car car = new Car();
        car.color = "black";        // compiler error
        car.make = "Mazda";         // compiler error
        car.model = "3";            // compiler error
        car.year = 2008;            // compiler error
        car.setColor("red");        // OK
        System.out.println(car);
    }
}
```

Private Instance Variables

- Private instance variables are ***only usable within*** the class.
- Private instance variables ***hide implementation*** details, promoting encapsulation.
- Private instance variables are ***not accessible*** by the client programmer (class user).
- Good programming practice:
  - Label ***all instance variables*** as ***private***.
  - The class has complete control over how/when/if the instance variables are changed.

Encapsulation Summary

- Combine methods and data in a single class.
- Use private instance variables for information hiding.
- Minimize the class's public interface.

Accessors & Mutators

- Class ***behavior may*** allow access to, or modification of, individual private instance variables.
- Accessor methods (getters)
  - Retrieves the value of a private instance variable
  - Java conventions have us start the method name with **`get`**.
  - Edge case for booleans — starts with **`is`**.
- Mutator methods (setters)
  - Changes the value of a private instance variable.
  - Conventional to start the name of the method with **`set`**.
- Gives the client program ***indirect access*** to the instance variables.

More Accessors and Mutators

- Doesn't the use of accessors and mutators defeat the purpose of making the instance variables private?
  - No, the class implementer decides which instance variables will have accessors.
- Mutators can:
  - Validate the new value of the instance variable
  - Decide whether or not to actually make the requested change

Example Car Accessor & Mutator

```
public class Car {

    private int year; // 4-digit year between 1000 and 9999
    private String vin;

    // accessor to return the vin member
    public String getVin() {
        return vin;
    }
    // mutator to change the year member
    public boolean setYear(int newYear) {
        if(1000 <= newYear && newYear <= 9999) {
            year = newYear;
            return true;
        }
        return false;
    }
    // ...
}
```

Accessor/Mutator Caution

- In general you should ***NOT*** provide accessors and mutators for all private instance variables.
  - Recall that the principle of encapsulation is best served with a ***limited class interface***.
- Too many accessors and mutators lead to writing procedural code rather than OOP code.
  - More on this later

Private Methods

- Methods may also be private.
  - Cannot be invoked by a client program
  - Can only be called by other methods within the same class definition
  - Most commonly used as “helper” methods to support top-down implementation of a public method

Private Method Example

```
public class Car {

    private int year; // 4-digit year between 1000 and 9999

    // helper method - internal use only
    private boolean isValidYear(int year) {
        return 1000 <= year && year <= 9999;
    }

    // mutator to change the year member
    public boolean setYear(int newYear) {
        if(isValidYear(newYear)) {
            year = newYear;
            return true;
        }
        return false;
    }
    // ...
}
```

More About Methods

- Different classes can define a method with the same name.
- Java can determine which method to call based on the type of the calling object.
- Example:

```
Cat fluffy = new Cat();
Dog fido = new Dog();
System.out.println(fluffy);
System.out.println(fido);
```

- `println(fluffy)` will call the `toString()` method defined in the Cat class because `fluffy`'s type is Cat.
- `println(fido)` will call the `toString()` method defined in the Dog class because `fido`'s type is Dog.

Method Overloading

- Two or more methods ***in the same class*** may also have the same name.
- This technique is known as ***method overloading***.

Overloaded setStyle Method

- The Car class setStyle method:

```
public boolean setStyle(String make, String model)
```

- Suppose we wanted to change only the model?
  - Define another method named setSytle():

```
public boolean setStyle(String make)
```

- After all, ***setStyle*** is a good descriptive name for what both methods do.

Car Class — Overloaded setStyle

```
public class Car {
    // ...
    private String model;
    private String make;
    public boolean setStyle(String make, String model) {
        if(isValidMake(make) && isValidModel(model)) {
            this.make = make;
            this.model = model;
            return true;
        }
        return false;
    }
    public boolean setStyle(String make) {
        if(isValidMake(make)) {
            this.make = make;
            return true;
        }
        return false;
    }
    private boolean isValidMake(String make) {
        return make != null && !make.equals("");
    }
    private boolean isValidModel(String model) {
        return model != null && !model.equals("");
    }
}
```

CarDemo Class

```
public class CarDemo {

    public static void main(String[] args) {
        Car car = new Car();

        car.setStyle("Mazda");
        System.out.println(car);

        car.setStyle("Audi", "A8");
        System.out.println(car);
    }

}
```

- How does Java know which setStyle method to call?

Method Signature

- A method is uniquely identified by
  - Its name and
  - Its parameter list (types and order)
- This is known as its ***signature***.
- Examples …

```
public boolean setStyle(String make, String model)
public boolean setStyle(int year, String make, String model)
public boolean setStyle(String make, String model, String color)
public boolean setStyle(int year, String color)
public boolean setStyle(String make)
```

Return Type is Not Enough

- Suppose we attempt to overload Car's setStyle() method by using different return types.

```
public void setSytle(String make) { /* code here */ }
public boolean setStyle(String model) { /* code here */ }
```

- This is ***NOT*** valid method overloading because the code that calls `setStyle()` can ignore the return value.

```
car.setStyle("Mazda");
```

- The compiler can't tell which `setStyle()` method to call.
- Just because a method returns a value doesn't mean the caller has to use it.

Too Much of a Good Thing

- Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler — for example …

```
//version 1
public double calculateAverage(int a, double b) { /* code */ }

//version 2
public double calculateAverage(double a, int b) { /* code */ }
```

- And then consider this:

```
MathUtils math = new MathUtils();
math.calculateAverage(5, 7);
```

- The Java compiler can't decide whether to
  - promote 7 to 7.0 and call the first version of calculateAverage, or
  - promote 5 to 5.0 and call the second.
- Solution
  - Cast the arguments so that they match one of the signatures.