Classes and Objects: Encapsulation

CMSC 202H
(Honors Section)

Encapsulation for Control

- We said we will use the term *encapsulation* in two different ways in this class (and in the text)
  - *Definition #1: "Inclusion" (“bundling"):*
    - *bundling of structure and function*
    - *Covered in lecture on “Object Design”*

  - **Definition #2: “Exclusion” (“access control”)**
    - **Strict, explicit control of how our objects can be used**
    - *This will be focus of this lecture*

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 be used and/or changed 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 instance variables and methods.
  - ***public*** visibility - accessible by everyone, in particular the client programmer
    - A class' interface is defined by its public methods.
  - ***private*** visibility - accessible only by the methods within the class
  - Two others—***protected*** and [package]—later

Date2 Class

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

```
public class Date2
{
   private String month;
   private int day;
   private int year;

   public String toString( )
   {
       return month + “ “ + day + “ “ + year;
   }
```

Any Date2 class method may use the class’ private instance variables.

```
   // setDate and monthString same as Date1 class
}
```

Access Control Example

Date1 class - **public** instance variables were used
Date2 class - **private** instance variables are now used

```
public class Date2Demo
{
  public static void main( String[ ] args )
  {
      Date2 myDate = new Date2( );

      myDate.month = “July”;  // compiler error
      myDate.day = 4;         // compiler error
      myDate.year = 1950;     // compiler error

      myDate.setDate( 7, 4, 1950 ); // OK - why?
      System.out.println( myDate.toString( ));
    }
}
```

Private Instance Variables

- Private instance variables are only directly accessible 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.
  - Instance variables primarily support class behavior.

Encapsulation Summary

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

"Keep it secret, keep it safe."

Accessors & Mutators

- Class *behavior* may allow access to, or modification of, individual private instance variables.

- Accessor method
  - retrieves the value of a private instance variable
  - conventional to start the method name with **`get`**
- Mutator method
  - 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

Question: Doesn’t the use of accessors and mutators defeat the purpose of making the instance variables `private`?

Answer: **No**

- The class implementer decides which instance variables will have accessors.
- Mutators can:
  - validate the new value of the instance variable, and
  - decide whether or not to actually make the requested change.

Date2 Accessor and Mutator

```
public class Date2
{
   private String month;
   private int day;     // 1 - 31
   private int year;    // 4-digit year

   // accessors return the value of private data
   public int getDay ( )
   { return day; }

   // mutators can validate the new value
   public boolean setYear( int newYear )
   {
        if ( 1000 <= newYear && newYear <= 9999 )
        {
            year = newYear;
            return true;
        }
        else // this is an invalid year
           return false;

   // rest of class definition follows
}
```

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.

Classes as Structures

- There are two possible exceptions to the “make everything private” rule:
  - When the class is actually just a simple data structure
    - No hard consistency rules
    - No behaviors
    - Local use
  - When performance is critical
    - However, this tradeoff is often not worthwhile

Private Methods

- Methods may 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 Date2
{
   private String month;
   private int day;          // 1 - 31
   private int year;         // 4-digit year

   // mutators should validate the new value
   public boolean setYear( int newYear )
   {
        if ( yearIsValid( newYear ) )
        {
            year = newYear;
            return true;
        }
        else     // year is invalid
           return false;

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