Lecture 21 — Encapsulation & Constructors

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Part I

Encapsulation

Encapsulation

Encapsulation is another principle of OOP: making the fields (variables) of a class inaccessible from outside the class.

Then access to the data is granted through the use of externally accessible methods.

Encapsulation can also mean bundling the data with methods that operate on that data.

To provide access to just the data, we use Properties.
These are a category of methods in C#.

Properties

Properties provide a public mechanism for accessing and manipulating the private data members of a class.

Properties have an identifier that normally matches the name of the private data member to be accessed / manipulated.

Properties are a convenient mechanism for implementing object-oriented methods for accessors and mutators.

Accessors (getters) read the value of a private data member.
Mutators (setters) write the value of a private data member.

Property Example in C#

A private data member `x` might have a property `X` defined as:

```
public float X
{
    get { return x; }
    set { x = value; }
}
```

Property `X` allows the private value of `x` to be set or read:

```
c2.X = 6.5f;
Console.WriteLine( c2.X );
```

The keyword `value` holds the value being assigned in the mutator.

The `set` part is optional; a property that does not contain the `set` part is read-only.

Get-and-Set Methods

Properties are not the only way to implement accessors and mutators.

Another common approach is to write get- and set- methods.

```
private double price;

public double GetPrice()
{
    return price;
}

public void SetPrice( double newPrice )
{
    price = newPrice;
}
```

Car Class Revisited

Here's an improved version of the `Car` class:

```
public class Car
{
    private string make;
    private string model;
    private int year;

    public string Make
    {
        get { return make; }
        set { make = value; }
    }
    public string Model
    {
        get { return model; }
        set { model = value; }
    }
    public int Year
    {
        get { return year; }
        set { year = value; }
    }
    // Other methods not shown for space reasons
}
```

Encapsulation: `public` and `private`

In the `struct` as well as the `class` until now, you've had to prefix your methods with `public` without being told why.

Then, in the last example, a new keyword appeared: `private`.

These are two access modifiers or visibility modifiers.

Access modifiers are a technique to restrict where a variable may be read/written from, or where a method can be called from.

Encapsulation: `public` and `private`

The two access modifiers we've seen so far are `public` and `private`.

Marking a method or variable `public` makes it accessible from everywhere in the program.

Any function may call a `public` function, for example.

Denoting a method or variable as `private` makes it accessible only from within that object type.

It's a compile time error to try to read a `private` variable from outside its declaring class.

As a rule of thumb, methods of a class tend to be `public` and the variables tend to be `private`.

Motivation for Encapsulation

It may seem silly to have private methods in the system - why bother?
Especially because we can easily change private into public.

The real purpose of the public/private distinction is to encourage good programming practice and maintainability.

In a large, complex system, any method or variable that is public will be relied upon by other classes.

In the future, it will be expensive or impossible to change.

Motivation for Encapsulation

Consider Microsoft's problem: maintaining program compatibility.

Any time they want to change something in Windows, they have to consider very carefully what the impact is on all the programs.

Microsoft can change private things, but not those that are public.

They know that people won't buy the new version of Windows if their old programs no longer function.

For more on this, see `http://blogs.msdn.com/b/oldnewthing/archive/2003/12/24/45779.aspx`

Accessors and Mutators for All!

A very common implementation choice is to make all member variables private and create accessor and mutator methods for each.

This provides some minor benefits over just making all variables public, but not all that much.

Accessor and mutator methods should exist only where there is a reason to create them.

Advantages of Encapsulation

Proper use of encapsulation results in better-designed software that:

- Maximizes cohesion – related elements being kept together; and
- Minimizes coupling – dependency of one part on another.

Much like re-using the `Date struct` we saw earlier, objects can and should be re-used to build larger objects.

If data is properly encapsulated, the implementation may be changed without having an impact on other parts of the program.

Change is inevitable in software, so we should plan for it.

Part II

Constructors

Constructors

A constructor is a special kind of method that is used to instantiate and initialize objects.

Every class has a default constructor that initializes each data member of a class to a default value. It has no parameters.

The default constructor for a class is accessed using the class name followed by round brackets.

For example, if you define a class named ABC, the default constructor for ABC is `ABC( )`.

Constructors

If you create a constructor, regardless of the number of parameters, for a class, the default constructor no longer exists.

Thus, if you want to have a no-parameters constructor in addition to any number of others, you must explicitly define it.

A constructor can have as many parameters as required.

Multiple constructors for a class may exist.
    This is an example of function overloading.

Coordinates Constructor Example

```
class Coordinates
{
    float x;
    float y;
    float z;

    public Coordinates( )              // Constructor 1
    {
        x = 0.0f;
        y = 0.0f;
        z = 0.0f;
    }

    public Coordinates( float a, float b, float c )    // Constructor 2
    {
        x = a;
        y = b;
        z = c;
    }
}
```

Coordinates Constructor Comments

In this example, an instance of a Coordinates object can be created using two different constructors:

```
public Coordinates( )
public Coordinates( float a, float b, float c )
```

A constructor does not have an explicitly-specified return type and always has the same name as its class.

The first constructor is used to create an instance of a Coordinates object that is initialized to (0, 0, 0).

The second constructor is used to create an instance of a Coordinates object initialized to (a, b, c).

Using Constructors

To utilize the class declaration on the previous slide, you need to create an object using the constructor.

To declare a variable, `c1`, using the first constructor:

```
Coordinates c1 = new Coordinates( );
```

This creates a reference type variable named `c1`.

This variable refers to a new object of type Coordinates that has been initialized using the first constructor.

Note the use of the `new` keyword to create a new object (like an array).

Using Constructors

To declare a variable, c2, initialized to (10.0, -5.5, 7.5):

```
Coordinates c2 = new Coordinates( 10.0f, -5.5f, 7.5f );
```

This creates a reference named `c2` that refers to a new object of type Coordinates that has been initialized using the second constructor.

When the object is created, the coordinates x, y, and z are set to values of 10.0, -5.5, and 7.5 respectively.

Copy Coordinates

Suppose we want to make a copy of the `Coordinates c2` object.

Recall that objects are reference types.

`Coordinates c3 = c2;` is not correct;

That statement will create another reference to the same instance.

Correct solution:

```
Coordinates c3 = new Coordinates( c2.X, c2.Y, c2.Z );
```

Or by manually assigning each variable:

```
Coordinates c3 = new Coordinates( );
c3.X = c2.X;
c3.Y = c2.Y;
c3.Z = c2.Z;
```

Copy Constructors

We can make creation of a copy easier by adding a copy constructor.

```
public Coordinates( Coordinates toCopy )
{
    x = toCopy.x;
    y = toCopy.y;
    z = toCopy.z;
}
```

This makes the syntax of creating a copy much simpler:

```
Coordinates c3 = new Coordinates( c2 );
```

Use of a copy constructor also allows better encapsulation; no need to know the internals of the class to make a correct copy.

Copy Constructor Alternative

Another strategy for making a copy of an object is a static method.

```
public static Coordinates MakeCopy( Coordinates toCopy )
{
    Coordinates result = new Coordinates( );
    result.X = toCopy.X;
    result.Y = toCopy.Y;
    result.Z = toCopy.Z;

    return result;
}
```

Then, to make a copy of `Coordinates c1`, call:

```
Coordinates c2 = Coordinates.MakeCopy( c1 );
```

About Destructors

Destructors are unnecessary in C#; C# uses garbage collection.
We will learn about this subject soon.

In a system with garbage collection, old objects that are no longer needed are automatically cleaned up.

In other languages, a destructor does the opposite of the constructor: it frees up resources that the object has accumulated over its lifetime.

This is needed in languages with manual memory management (such as C++) and the definition is included here only for completeness.