ENCAPSULATION SERVICES 2020

Ph.D. / Golden Gate Ave, San Francisco / Seoul National Univ / Carnegie Mellon / UC Berkeley / DevOps / Deep Learning / Visualization

- K Hong (http://bogotobogo.com/about_us.php)

Encapsulation Services

The idea of encapsulation is that an object's internal data should not be directly accessible from an object instance.

Rather, if the called wants to alter the state of an object, the user does so indirectly using accessor (getter) and mutator (setter) methods. In C#, encapsulation is enforced at the syntactic level using the **public**, **private**, **internal**, and **protected** keywords. To demonstrate the need for encapsulation services, assume we have created the following class definition:

```
class Book
{
        public int Pages();
}
```

The problem with public field data is that the item has no ability to intrinsically **understand** whether the current value to which they are assigned is valid with regard to the current rule of the system. As we know, the upper range of a C# **int** is quite large (2,147,483,647). Therefore, the compiler allows the following assignment:

C# 4.0

.NET Framework

```
static void Main(string[] args)
{
        Book pictureBook = new Book();
        pictureBook.Pages = 1500000000;
}
```


Even though we don't have overflow error, it should be clear that a **pictureBook** with a 1,500,000,000 pages is unreasonable. If our system has rule that limits the number of pages to 1000, we are at a loss to enforce this programmatically. Because of this, public field typically have no place in a production-level class definition.

Encapsulation provides a way to preserve the integrity of an object's state data. Rather than defining public fields, we should get in the habit of defining **private data**, which is indirectly manipulated using one of two main techniques:

1. Define a pair of accessor (getter) and mutator (setter) methods.
2. Define a type property.

In addition to those, C# provides the **readonly** keyword, which also delivers a level of data protection. Well-encapsulated class should hide the details of how it operates from the outside. This is often referred as **black box programming**. The beauty of this approach is that an object is free to change how a given method is implemented under the hood. It does this without breaking any existing code making use of it, provided that the signature of the method remains constant.

Accessors and Mutators

If we want the outside world to interact with our private string representing a work's name, we can use accessor and a mutator.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeApp
{
    class Employee
    {
        private string empName;
        private int empID;
        private float curPay;

        public Employee() { }
        public Employee(string name, int id, float pay)
        {
            empName = name;
            empID = id;
            curPay = pay;
        }

        public string GetName()
        {
            return empName;
        }

        public void SetName(string name)
        {
            empName = name;
        }

        public void GiveBonus(float amount)
        {
            curPay += amount;
        }

        public void DisplayStats()
        {
            Console.WriteLine("Name: {0}", empName);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("Pay: {0}", curPay);
        }
```

Interfaces (http://www.bogotobogo.com/CSharp/cs

Delegates (http://www.bogotobogo.com/CSharp/cs

System.Object (http://www.bogotobogo.com/CSharp/cs

Events (http://www.bogotobogo.com/CSharp/cs

Multi Threading I - Introduction and Simple Thread (http://www.bogotobogo.com/CSharp/cs

Multi Threading II - ThreadStart / Parameterized ThreadStart, Foreground/Background Threads (http://www.bogotobogo.com/CSharp/cs

Multi Threading III - Concurrency, Synchronization (http://www.bogotobogo.com/CSharp/cs

Networking I -

PORT, IPv4/IPv6, TCP/UDP, URI (http://www.bogotobogo.com/CSharp/cs

Networking II -

WebRequest / WebResponse, WebClient (http://www.bogotobogo.com/CSharp/cs

```
        static void Main(string[] args)
        {
            Employee e = new Employee("Ben", 987, 140000);
            e.GiveBonus(20000);
            e.DisplayStats();

            e.SetName("Ken");
            Console.WriteLine("Employee is named: {0}", e.GetName());
            Console.ReadLine();
        }
    }
}
```

Output we get from the run is:

```
Name: Ben
ID: 987
Pay: 160000
Employee is named: Ken
```

Type Properties

Though we can encapsulate field data using traditional get and set methods, .NET prefers to enforce data protection using **properties**. First of all, note that properties always map to accessor and mutator methods in terms of CIL (Common Intermediate Language) code. Thus, as a class designer, we're still able to perform any internal logic necessary before making the value assignment.

Here is the updated **Employee** class enforcing encapsulation of each field using property syntax rather than get and set methods.

WPF (Windows Presentation Foundation) and XAML - Part I (http://www.bogotobogo.com/CSharp/cs

GIT and GitHub on Windows (http://www.bogotobogo.com/cplusplus/

PowerShell 4 ... (http://bogotobogo.com/Powershell/Pow

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeApp
{
    class Employee
    {
        private string empName;
        private int empID;
        private float curPay;

        public string Name
        {
            get { return empName; }
            set { empName = value; }
        }

        public int ID
        {
            get { return empID; }
            set { empID = value; }
        }

        public float Pay
        {
            get { return curPay; }
            set { curPay = value; }
        }

        public Employee() { }
        public Employee(string name, int id, float pay)
        {
            empName = name;
            empID = id;
            curPay = pay;
        }

        public void GiveBonus(float amount)
        {
            curPay += amount;
        }
```

```
        public void DisplayStats()
        {
            Console.WriteLine("Name: {0}", empName);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("Pay: {0}", curPay);
        }

        static void Main(string[] args)
        {
            Employee e = new Employee("Ben", 987, 140000);
            e.GiveBonus(20000);
            e.DisplayStats();

            e.Name = "Ken";
            Console.WriteLine("Employee is named: {0}", e.Name);

            Console.ReadLine();
        }
    }
}
```

Output is:

```
Name: Ben
ID: 987
Pay: 160000
Employee is named: Ken
```

Properties also make our types easier to manipulate, in that properties are able to respond to the intrinsic operators of C#. To demonstrate, assume that the **Employee** class type has an internal private member variable representing the age of the employee.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeApp
{
    class Employee
    {
        private string empName;
        private int empID;
        private float curPay;
        private int empAge;

        public string Name
        {
            get { return empName; }
            set { empName = value; }
        }

        public int ID
        {
            get { return empID; }
            set { empID = value; }
        }

        public float Pay
        {
            get { return curPay; }
            set { curPay = value; }
        }

        public int Age
        {
            get { return empAge; }
            set { empAge = value; }
        }

        public Employee() { }
        public Employee(string name, int id, int age, float pay)
        {
            empName = name;
            empID = id;
```

```
            empAge = age;
            curPay = pay;
        }

        public void GiveBonus(float amount)
        {
            curPay += amount;
        }

        public void DisplayStats()
        {
            Console.WriteLine("Name: {0}", empName);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("Age: {0}", empAge);
            Console.WriteLine("Pay: {0}", curPay);
        }

        static void Main(string[] args)
        {
            Employee e = new Employee("Ben", 987, 30, 140000);
            e.GiveBonus(20000);
            e.DisplayStats();

            e.Name = "Ken";
            Console.WriteLine("Employee is named: {0}", e.Name);

            Console.ReadLine();
        }
    }
}
```

Output we get:

```
Name: Ben
ID: 987
Age: 30
Pay: 160000
Employee is named: Ken
```

Assume we have created an **Employee** object named **hermione**. On her birthday, we wish to increment the age by one. Using accessor and mutator methods, we would need to write code like this:

```
Employee hermione = new Employee();
hermione.SetAge(hermione.GetAge() + 1);
```

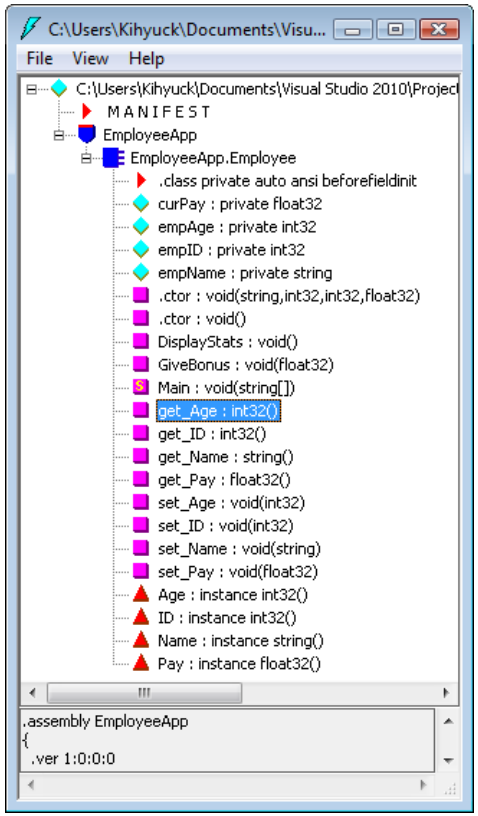
But if we encapsulate **empAge** using a property named **Age**, we are able to simply write:

```
Employee hermione = new Employee();
hermione.Age++;
```

Internal Representation of Properties

Quite a few programmers tend to name traditional accessor and mutator methods using **get_** and **set_** prefixes such as **get_Name()** and **set_Name()**. This naming convention is not problematic in itself. It is important to understand, however, that under the hood, a property is represented in CIL code using these same prefixes.

We can check it using **ildasm.exe**. Open up the **EmployeeApp.exe** assembly using **ildasm.exe**, we can see that each property is mapped to hidden **get_*** methods called internally by the CLR.



Assume the **Employee** type now has a private member variable **empSSN** to represent an individual's Social Security number, which is manipulated by a property named **SocialSecurityNumber**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeApp
{
    class Employee
    {
        private string empName;
        private int empID;
        private float curPay;
        private int empAge;
        private string empSSN;

        public string Name
        {
            get { return empName; }
            set { empName = value; }
        }

        public int ID
        {
            get { return empID; }
            set { empID = value; }
        }

        public float Pay
        {
            get { return curPay; }
            set { curPay = value; }
        }

        public int Age
        {
            get { return empAge; }
            set { empAge = value; }
        }

        public string SocialSecurityNumber
        {
            get { return empSSN; }
            set { empSSN = value; }
```

```
        }

        public Employee() { }
        public Employee(string name, int id, int age, float pay, string ssn)
        {
            empName = name;
            empID = id;
            empAge = age;
            curPay = pay;
            empSSN = ssn;
        }

        public void GiveBonus(float amount)
        {
            curPay += amount;
        }

        public void DisplayStats()
        {
            Console.WriteLine("Name: {0}", empName);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("Age: {0}", empAge);
            Console.WriteLine("SSN: {0}", empSSN);
            Console.WriteLine("Pay: {0}", curPay);
        }

        static void Main(string[] args)
        {
            Employee e = new Employee("Ben", 987, 30, 140000, "231988946");
            e.GiveBonus(20000);
            e.DisplayStats();

            e.Name = "Ken";
            Console.WriteLine("Employee is named: {0}", e.Name);

            Console.ReadLine();
        }
    }
}
```

Output is:

```
Name: Ben
ID: 987
Age: 30
SSN: 231988946
Pay: 160000
Employee is named: Ken
```

If we also define two methods **get_SocialSecurityNumber()** and **set_SocialSecurityNumber()** in the same class, we would get compile-time errors:

```
class Employee
{
        ...
        public string get_SocialSecurityNumber()
        {
                return empSSN;
        }

        public void set_SocialSecurityNumber(string ssn)
        {
                empSSN = ssn;
        }
}
```

Visibility Levels of Property Get/Set

Prior to .NET 2.0, the visibility of get and set was solely controlled by the access modifier of the property declaration:

```
public string SocialSecurityNumber
{
        get { return empSSN; }
        set { empSSN = value; }
}
```

It would be useful to specify unique accessibility levels for get an set simply by prefixing an accessibility keyword to the appropriate **get** or **set** keyword and the unqualified scope takes the visibility of the property's declaration:

```
public string SocialSecurityNumber
{
        get { return empSSN; }
        protected set { empSSN = value; }
}
```

In this case, the set logic of **SocialSecurityNumber** can only be called by the current class and derived classes and therefore cannot be called from an object instance.

Read/Write Only Property Get/Set

When encapsulating data, we may want to configure a **read-only property**. We can simply omit the **set** block. Similarly, if we want to have a **write-only property**, omit the **get** block.

For read-only:

```
public string SocialSecurityNumber
{
        get { return empSSN; }
}
```

Given this adjustment, the only manner in which an employee's SSN can be set is through a constructor argument. Thus, it would now be a compiler error to attempt to set an employee's SSN value:

```
static void Main(string[] args)
{
        Employee e = new Employee("Ben", 987, 30, 140000, "231988946");
        e.GiveBonus(20000);
        e.DisplayStats();

        // Error because SSN is read only!
        e.SocialSecurityNumber = "231988946";

        Console.ReadLine();
}
```

Static Properties

C# also supports static properties. Static members are accessed at the class level, not from an instance of that class. Here we added a static point of data to represent the name of the company employing these workers.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeApp
{
    class Employee
    {
        private string empName;
        private int empID;
        private float curPay;
        private int empAge;
        private string empSSN;
        private static string companyName;

        public string Name
        {
            get { return empName; }
            set { empName = value; }
        }

        public int ID
        {
            get { return empID; }
            set { empID = value; }
        }

        public float Pay
        {
            get { return curPay; }
            set { curPay = value; }
        }

        public int Age
        {
            get { return empAge; }
            set { empAge = value; }
        }

        public string SocialSecurityNumber
        {
            get { return empSSN; }
```

```
        set { empSSN = value; }
    }

    public static string Company
    {
        get { return companyName; }
        set { companyName = value; }
    }

    public Employee() { }
    public Employee(string name, int id, int age, float pay, string ssn)
    {
        empName = name;
        empID = id;
        empAge = age;
        curPay = pay;
        empSSN = ssn;
    }

    public void GiveBonus(float amount)
    {
        curPay += amount;
    }

    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);
        Console.WriteLine("ID: {0}", empID);
        Console.WriteLine("Age: {0}", empAge);
        Console.WriteLine("SSN: {0}", empSSN);
        Console.WriteLine("Pay: {0}", curPay);
    }

    static void Main(string[] args)
    {
        Employee.Company = "Bogotobogo Inc.";
        Console.WriteLine("These folks work at {0} ", Employee.Company);

        Employee e = new Employee("Ben", 987, 30, 140000, "231988946");
        e.GiveBonus(20000);
        e.DisplayStats();

        e.Name = "Ken";
        Console.WriteLine("Employee is named: {0}", e.Name);
```

```
            Console.ReadLine();
        }
    }
}
```

With output:

```
These folks work at Bogotobogo Inc.
Name: Ben
ID: 987
Age: 30
SSN: 231988946
Pay: 160000
Employee is named: Ken
```

Partial Types

Classes can be defined with a type modifier **partial** that allows us to define a type across multiple ***cs** files. Traditionally all code for a given type required to be defined with a single ***.cs** file. Given the fact that a production-level C# class could end up being thousands of lines of code, this could be problematic.

In that case, it may be beneficial to partition a type's implementation across several ***.cs** files in order to separate code that is in some way more important from other aspects of the type definition. For instance, using the **partial** class modifier, we could place all of the **Employee** constructors and properties into a new file named **Employee.Internals.cs**:

```
partial class Employee
{
        // Constructors
        ...
        // Properties
        ...
}
```

So, here are the two files: **Employee.cs** and **Employee.Internals.cs**:

```
// Employee.Internals.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Employee
{
    partial class Employee
    {
        public string Name
        {
            get { return empName; }
            set { empName = value; }
        }

        public int ID
        {
            get { return empID; }
            set { empID = value; }
        }

        public float Pay
        {
            get { return curPay; }
            set { curPay = value; }
        }

        public int Age
        {
            get { return empAge; }
            set { empAge = value; }
        }

        public string SocialSecurityNumber
        {
            get { return empSSN; }
            set { empSSN = value; }
        }

        public static string Company
        {
```

```
            get { return companyName; }
            set { companyName = value; }
        }

        public Employee() { }
        public Employee(string name, int id, int age, float pay, string ssn)
        {
            empName = name;
            empID = id;
            empAge = age;
            curPay = pay;
            empSSN = ssn;
        }
    }
}
```

The private field data and type methods are defined within the initial **Employee.cs**.

```
// Employee.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Employee
{
    partial class Employee
    {
        private string empName;
        private int empID;
        private float curPay;
        private int empAge;
        private string empSSN;
        private static string companyName;

        public void GiveBonus(float amount)
        {
            curPay += amount;
        }

        public void DisplayStats()
        {
            Console.WriteLine("Name: {0}", empName);
            Console.WriteLine("ID: {0}", empID);
            Console.WriteLine("Age: {0}", empAge);
            Console.WriteLine("SSN: {0}", empSSN);
            Console.WriteLine("Pay: {0}", curPay);
        }

        static void Main(string[] args)
        {
            Employee.Company = "Bogotobogo Inc.";
            Console.WriteLine("These folks work at {0} ", Employee.Company);

            Employee e = new Employee("Ben", 987, 30, 140000, "231988946");
            e.GiveBonus(20000);
            e.DisplayStats();

            e.Name = "Ken";
            Console.WriteLine("Employee is named: {0}", e.Name);
```

```
            Console.ReadLine();
        }
    }
}
```

These two files are compiled by the C# compiler, the end result is a single unified type. To this end, the **partial** modifier is purely a design-time construct.

Note that the names we give to the files that contain partial type definitions are entirely up to us. Here, **Employee.Internals.cs** was chosen simply to indicate that this file contains infrastructure code that most developers can ignore. The only requirement when defining partial types is that the type's name (**Employee** in this example) is identical and defined within the same .NET namespace.

CONTACT

BogoToBogo
contactus@bogotobogo.com (mailto:#)

FOLLOW BOGOTOBOGO

(https://www.facebook.com/KHongSanFrancisco) (https://twitter.com/KHongTwit) (https://plus.google.com/u/0/+KHongSanFrancisco/posts)

ABOUT US (/ABOUT_US.PHP)

contactus@bogotobogo.com (mailto:contactus@bogotobogo.co)

Golden Gate Ave, San Francisco, CA 94115

Golden Gate Ave, San Francisco, CA 94115