Access Modifiers
Public: Any class or struct
Private: Same class or struct
Internal: Same assembly
Protected: Same or derived from class
Protected Internal: Same assembly *or* derived from class
Private Protected: Same assembly *and* derived from class.
Friend: Specify methods to access internal types from different assembly
Sealed prevents inheritance

Defaults
Class: internal
Member: private

Namespaces
Disambiguate identical function names.
Math.Max, Enumerable.Max
Often used to name DLLs.

Interfaces
ITweet = Search vs. Streaming Tweets
IEnumerable – iterates over a collection.
ICollection – iterate, modified
IList – iterate, modify, sort, access with subscript
IDictionary – Dictionary[key]
IQueryable – Bring filtered data to client.

Interface vs. Abstract Class
Int: Contract to contain a function.
Abs: Actually contains function. Can contain fields.
*Can derive from two interfaces but not two classes.

Collections (.Count)
System.Collections not strongly typed
ArrayList – variable length array.
HashTable
Stack
Queue

System.Collections.Generic strongly typed
Array
List
Dictionary

Object Oriented Programming
Inheritance
Encapsulation
    Binding data with methods.
Polymorphism
    Overloading
    Overriding
Abstraction
    Includes signature, not implementation.

SOLID
Single Responsibility
Open to extension / Closed to modification
Liskov Substitution: Should be able to substitute any subclass for a class. (Replace Animal with Dog.)
Interface Segregation: Keep interfaces as small as possible and implement multiple if needed.
Dependency Inversion: A class should not depend on a smaller class. Pass the smaller class into the subclass instead.

Linq

Sorting
```
// Use Linq to get an IOrderedEnumerable<City>
var items = from city in cities2
orderby city.CityName ascending
select city;
```

Selection
```
currentCities = items.Where(a =>
a.Interstates.Contains(Interstate)).ToList();
```

Entity Framework
Add Record
```
WebScrapingEntities : DbContext
WebScrapingEntities w = new WebScrapingEntities();
w.TickersAs.Add(t);
w.SaveChanges();
```

Get Record
```
TickersA t2 = w.TickersAs.SingleOrDefault(p => p.TickerID ==
54);
```

Strings
String immutable: use if data will not change.
StringBuilder: use if data will change

Threads
BackgroundWorker
Anonymous functions.

```
using System.Threading;
new Thread(() =>
{
   Thread.CurrentThread.IsBackground = true;
   /* run your code here */
   Console.WriteLine("Hello, world");
}).Start();
```

SQL
SELECT *
INTO *newtable* [IN *externaldb*]
FROM *table1;*

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);

INSERT INTO table2
(column_name(s))
SELECT column_name(s)
FROM table1;

DELETE w
FROM WorkRecord2 w
INNER JOIN Employee e
  ON EmployeeRun=EmployeeNo
Where Company = '1'
```

Frugality: Built an image load file validator out of a Word document.

Insist on the Highest Standards: Migration from Nuix NVA to compiled Java with UI (Engine API).

Invent and Simplify: Created a new time zone to replace a Click Robot used to select Daylight Saving time.

Customer Obsession, Are Right, A lot: Metalincs displayed imprecise counts. Explaining it was not good enough. Client demanded precision.

Unit Testing: Percentage CJK to select UTF-8 versus UTF-16.

System Testing: Filter. Checking query counts.

Most Proud Of:

- Dedupe – never need to open a compound case. Helps the lab.
- Proof-of-Concept capture text from print stream.
- Replace YTria with LotusTalker.

Design Patterns

Static Classes

1. Not an instance.
2. Contains only static members.
3. Non-static members result in error at compile time.

Static Classes vs. Singletons

1. Static classes have global scope.
2. Both have one instance.
3. Static classes contain only static members.

Singleton – Creates exactly one instance of a class.

1. **Concrete class**: Concrete class.
2. **Static Field**: Class has static field = new instance of class. (Greedy static instantiation guarantees thread safety)
3. **Private Constructor**: Class has private or protected constructor.
4. **Static Creation Method** Class has static method MakeInstance, which returns private field theInstance.

```
public class Program
{
    public static void Main(string[] args)
    {

        Logger logger = Logger.MakeInstance();
        Console.WriteLine(logger.a);  //1
        logger.a = 2;
        Console.WriteLine(logger.a);  //2
        Logger logger2 = Logger.MakeInstance();
        Console.WriteLine(logger2.a);  //2 because still returning theInstance
                                       //proves that it is the same instance.
        Console.WriteLine(Logger.TheInstance.a); //2
        logger.WriteOutput(); //Writing Output
        Console.ReadLine();
    }
}

public class Logger
{
    //FIELDS
    public int a = 1; // instance variable because does not use "static".
    static Logger theInstance = new Logger(); // Accesses the private constructor.

    //PROPERTIES
    public static Logger TheInstance
    {
        get { return theInstance; } // returns the single instance.
    }

    //CONSTRUCTOR
    private Logger()
    {  }

    public static Logger MakeInstance()
    {
        return theInstance; //returns the single instance.
    }

    public void WriteOutput()
    {
        Console.WriteLine("Writing Output");
    }
}
```

Design Pattern – Factory

Why: You use the Factory design pattern when you want to define the class of an object at runtime. It also allows you to encapsulate object creation so that you can keep all object creation code in one place.

Creation

1. **Interface**: Interface for products.
2. **Concrete Classes**: One for each product. Create multiple classes, each of which implements the same interface.
3. **Factory Class (static)**: A concrete class responsible for creation of the product.
4. **Factory Method**: A method in the factory class with conditionals.
5. **Conditions**: IInvoice = factory.CreateInvoice(int i)

```
class Program
    {
        static void Main(string[] args)
        {
            int intInvoiceType = 0;
            IInvoice objInvoice;
            Console.WriteLine("Enter Invoice Type");
            intInvoiceType = Convert.ToInt32(Console.ReadLine());
            objInvoice = clsFactoryInvoice.getInvoice(intInvoiceType);
            objInvoice.print();
            Console.ReadLine();
        }
    }

    public static class clsFactoryInvoice
    {
        static public IInvoice getInvoice(int intInvoiceType)
        {
            IInvoice objinv;
            if(intInvoiceType == 1)
            {   objinv = new clsInvoiceWithHeader(); }
            else if (intInvoiceType == 2)
            {   objinv = new clsInvoiceWithOutHeaders(); }
            else
            {   return null; }
            return objinv;
        }
    }

    public class clsInvoiceWithHeader : IInvoice
    {
        public void print()
        {  Console.WriteLine("With Headers");  }
    }

    public class clsInvoiceWithOutHeaders : IInvoice
    {
        public void print ()
        {  Console.WriteLine("WITHOUT Headers"); }
    }

    public interface IInvoice
    {
        void print();
    }
```

Design Pattern – Factory Method

Why: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Creation

- Create an abstract base class.
- Create an abstract method in the base class. This is the factory method.
- Create multiple concrete descendants of the base class. Make sure to override the factory method in concrete classes.
- Abstract / overriden methods return a concrete class of the desired type.
- Call abstract method in abstract class to get an object of the desired type.
- Call a method of the object.

```
//namespace DoFactory.GangOfFour.Factory.Structural
//{
/// <summary>
/// MainApp startup class for Structural
/// Factory Method Design Pattern.
/// </summary>
class MainApp
{
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  public static void Main()
  {
    // An array of creators
    Creator[] creators = new Creator[2];

    creators[0] = new ConcreteCreatorA();
    creators[1] = new ConcreteCreatorB();

    // Iterate over creators and create products
    foreach (Creator creator in creators)
    {
      Product product = creator.FactoryMethod();
      Console.WriteLine("Created {0}",
        product.GetType().Name);
    }

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Product' abstract class
/// </summary>
abstract class Product
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ConcreteProductA : Product
{
}

/// <summary>
/// A 'ConcreteProduct' class
```

```
/// </summary>
class ConcreteProductB : Product
{
}

/// <summary>
/// The 'Creator' abstract class
/// </summary>
abstract class Creator
{
  public abstract Product FactoryMethod();
}

/// <summary>
/// A 'ConcreteCreator' class
/// </summary>
class ConcreteCreatorA : Creator
{
  public override Product FactoryMethod()
  {
    return new ConcreteProductA();
  }
}

/// <summary>
/// A 'ConcreteCreator' class
/// </summary>
class ConcreteCreatorB : Creator
{
  public override Product FactoryMethod()
  {
    return new ConcreteProductB();
  }
}
//}

MainApp.Main();
```

Design Pattern – Abstract Factory Method

Why: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Create:

- Create abstract factory (class).
- Create two or more abstract methods in abstract factory.
- Pass factory to client class.
- Call factory in client class.

```
namespace DoFactory.GangOfFour.Abstract.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Abstract Factory Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>

        public static void Main()
        {
            // Create and run the African animal world
            ContinentFactory africa = new AfricaFactory();
            AnimalWorld world = new AnimalWorld(africa);
            world.RunFoodChain();

            // Create and run the American animal world
            ContinentFactory america = new AmericaFactory();
            world = new AnimalWorld(america);
            world.RunFoodChain();

            // Wait for user input
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'AbstractFactory' abstract class
    /// </summary>
    abstract class ContinentFactory
    {
        public abstract Herbivore CreateHerbivore();
        public abstract Carnivore CreateCarnivore();
    }

    /// <summary>
    /// The 'ConcreteFactory1' class
    /// </summary>
    class AfricaFactory : ContinentFactory
    {
        public override Herbivore CreateHerbivore()
        {
            return new Wildebeest();
        }

        public override Carnivore CreateCarnivore()
        {
            return new Lion();
        }
```

```
}

/// <summary>
/// The 'ConcreteFactory2' class
/// </summary>
class AmericaFactory : ContinentFactory
{
    public override Herbivore CreateHerbivore()
    {
        return new Bison();
    }

    public override Carnivore CreateCarnivore()
    {
        return new Wolf();
    }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{

}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
    public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
    public override void Eat(Herbivore h)
    {
        // Eat Wildebeest
        Console.WriteLine(this.GetType().Name +
          " eats " + h.GetType().Name);
    }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
```

```
    class Wolf : Carnivore
    {
        public override void Eat(Herbivore h)
        {
            // Eat Bison
            Console.WriteLine(this.GetType().Name +
              " eats " + h.GetType().Name);
        }
    }

    /// <summary>
    /// The 'Client' class
    /// </summary>
    class AnimalWorld
    {
        private Herbivore _herbivore;
        private Carnivore _carnivore;

        // Constructor
        public AnimalWorld(ContinentFactory factory)
        {
            _carnivore = factory.CreateCarnivore();
            _herbivore = factory.CreateHerbivore();
        }

        public void RunFoodChain()
        {
            _carnivore.Eat(_herbivore);
        }
    }
}
```