

# The Scoop on OOP in 90 minutes

# Objectives

- Fast-path
  - OOP fundamentals
  - Can't possibly cover everything
- Emphasis on practical use

# Whoa, There's a Lot to Know!!!

Where does one begin ?

Classes vs. Objects

Encapsulation

Inheritance

Reference Variables

Aggregation

Polymorphism

Fields

Instances

Properties

Abstract Classes

Methods

Overloading

Constructors

Interfaces

Overriding

# Classes

# Classes are Like Cookie Cutters

- Think of classes as being templates
- Classes **Encapsulate**
  - Data
  - Behavior (i.e. management of the data)

... in one location

# Examples of Classes

- ASP.Net Classes
  - Page, GridView, Button, DataSet
- Business Classes
  - Customer, Order, Product

# Why Bother ?

- When changes are required, we want to avoid ...
  - Shotgun-Surgery
  - Solution Sprawl
  
- Classes help to facilitate ...
  - Ease of maintenance, adaptability
  - Re-use of common logic ( avoid copy/paste )

# How Do We Achieve These Benefits?

- Centralization of
  - Data  
and
  - Logic that operates on that data
- **Cohesion**
  - The strength of the relationship between the data and logic
  - Remember the "Sesame Street Rule"
    - "One of these things is not like the other"

# Fields

- Variable data managed by the object
- Manipulation of data usually controlled by
  - Properties
  - Methods
- Goal is to ensure that field data remains in a valid state
  - The class defines these rules

# Fields

```
public class Location
{
    #region Fields
    private int _id = 0;
    private string _address = "";
    private string _city = "";
    private string _state = "";
    private string _zip = "";
    #endregion

    Properties

}

```

The *Private* access modifier ensures that field can't be updated from logic outside the class

# Properties

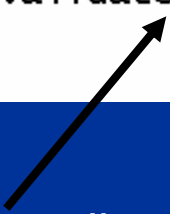
- Characteristics, attributes
- Control access to internal **Fields**
  - Ensures that internal data remains valid
- **Key Concepts**
  - Getters, Setters

# Read-Only Properties

```
public int Id
{
    get
    {
        return this._id;
    }
}
```

# Read-Write Properties

```
public string Address
{
    get
    {
        return this._address;
    }
    set
    {
        validateAndsetProperty(value, 5, 100, "Address", this._address);
    }
}
```



This setter calls another method

- Validates that length of the value in is between 5 and 100
- Sets the value of an internal field

# More Complex Properties

- Common misconception
  - Properties only interact with single fields
- This read-only property pulls data from multiple fields

```
public string FormattedAddress
{
    get
    {
        return this._address + "\n" +
               this._city + "\n" +
               this._state + "\n" +
               this._zip;
    }
}
```

# Methods

- Behaviors, Operations, Responsibilities
  - If objects are NOUNS
  - Methods are VERBS

# Methods That Don't Return Data

```
public class Customer
{
    Fields

    Constructors

    Properties

    #region Concrete Methods
    public void SetDefaultPaymentTerms(PaymentTerms terms)
    {
        // business logic goes here
    }

    public void AddShippingLocation(string address, string city,
                                    string state, string zip)
    {
        // business logic goes here
    }
}
```

```
#region Enumerated Data Types
public enum PaymentTerms
{
    Net30 = 30,
    Net60 = 60,
    Net90 = 90
}
#endregion
```

# Methods That Return Data

- Methods can return data as a single type
  - Similar to functions
    - May be simple primitive types (re: the first method returns an integer)
    - Or complex types (re: the second method returns a Location)
- Data can be returned via parameters as well

```
public int Save()
{
    // business logic goes here
}

public Location GetCorporateHQLocation()
{
    if (this._locations.Count == 0)
        return null;

    // Assume that the Headquarters will
    // always be the first location in
    // the list
    return this._locations[0];
}
```

# Overloading Methods

Can have many versions of a method

- Must have different signatures

```
public class Customer
{
    Fields

    Properties

    Methods

    More Methods

    #region And Some More Methods
    public int AddContact(string firstName, string lastName)
    {
        // business logic goes here

        // assume that this method returns a unique Id
        // for the new contact
    }

    public int AddContact(string firstName, string lastName, string email)
    {
        // business logic goes here
    }

    public int AddContact(string firstName, string lastName,
                          string email, string phone)
    {
        // business logic goes here
    }
}
```

# Information Hiding

- Have you noticed that I haven't emphasized internal business logic so far?
- Best practice for class design
  - Consider the external interface first
    - Public Properties
    - Public Methods
  - Think about the use-cases
    - How is information used by the application?
    - Data may be persisted in a very different way
      - Don't jump to merely creating classes that mirror database tables

# Information Hiding

- Details of inner workings of class should be hidden from clients (or consumers)
  - e.g.
    - How the database is accessed
    - Algorithms used to manipulate data
- Why?
  - Allows freedom to change internal workings
    - E.g. optimizations, use of data structures, business rules
  - Clients of class never know the difference

# The Birth of an Object

# What are Objects ?

If classes are like cookie cutters  
objects are the actual cookies

**Class Type**

**Object**

Customer customer =

```
new Customer( txtCustomerName.Text );
```

**Instantiation**

# Object Instantiation

- Causes memory to be allocated for a class
- Creates an **Instance** of a class
  - A.k.a. **An Object**
  - Each instance has its own data

Therefore

**Classes != Objects**

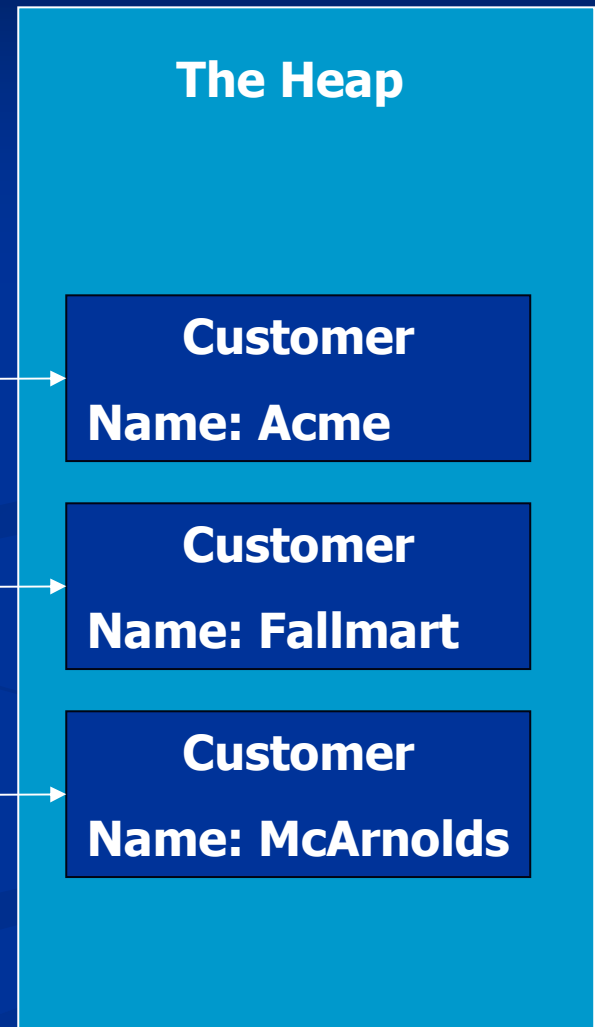
# Instances

- Object variables reference instances in **the Heap**
  - **Reference Variables**
- May be **Garbage Collected**

Customer **customer1** = new Customer( "Acme");

Customer **customer2** = new Customer( "Fallmart");

Customer **customer3** = new Customer( "McArnolds");



# Constructors

- Defines minimal data required to initialize object in a valid state
- Can have many versions of c-tors
  - Must have different signatures

```
public class Customer
{
    Fields

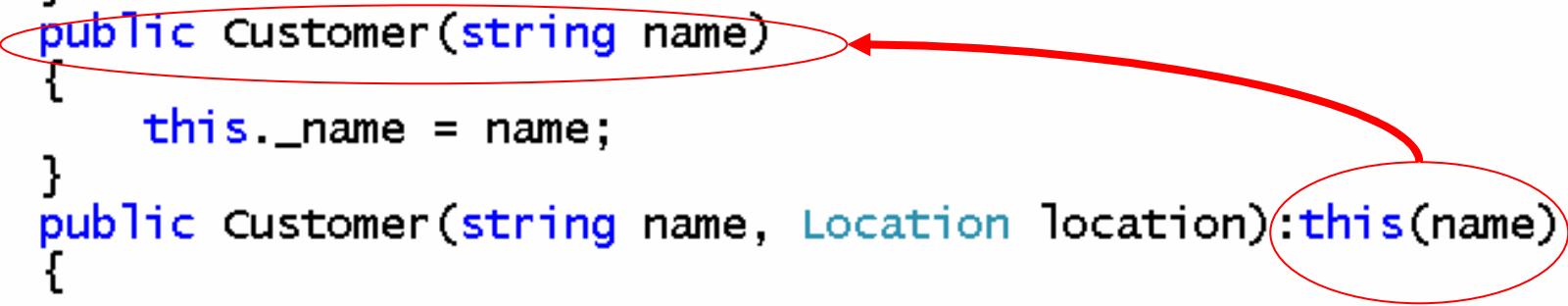
    #region Constructors
    public Customer()
    {
        // default c-tor; aka no-arg c-tor
    }
    public Customer(string name)
    {
        this._name = name;
    }
    public Customer(string name, Location location):this(name)
    {
        this._locations.Add(location);
    }
}
```

# Constructors

- One constructor can call another
  - Allows reuse of initialization logic

```
public class Customer
{
    Fields

    #region Constructors
    public Customer()
    {
        // default c-tor; aka no-arg c-tor
    }
    public Customer(string name)
    {
        this._name = name;
    }
    public Customer(string name, Location location):this(name)
    {
        this._locations.Add(location);
    }
}
```



# How are We Doing?

**Classes vs. Objects**

**Encapsulation**

Inheritance

Polymorphism

Aggregation

**Reference Variables**

**Instances**

**Properties**

**Fields**

**Methods**

**Overloading**

Abstract Classes

**Constructors**

Interfaces

Overriding

# Object Relationships

# Containment

- “Has A” relationships
  - A Customer has
    - Locations
    - Contacts
    - Licenses
  - A Product has Pricing Rules
- One object **contains** another object
  - Sometimes called **Aggregation**

# Aggregation

Child objects are declared as **Instance Fields**

- Here we have an three child objects
  - 2 List objects that also contain objects (i.e. Locations, Contacts)
  - 1 object of the type Order
    - Marked as protected to ensure that can only be manipulated by subclasses

```
public class Customer
{
    #region Fields

    private int _id=0;
    private string _name = "";
    private List<Location> _locations = new List<Location>();
    private List<Contact> _contacts = new List<Contact>();

    protected Order _order = null;

    #endregion
}
```

# Accessing Contained Objects via Properties

## ■ Property Definition

- This looks like a read-only property BUT
- The contacts list can be update by outside statements because this is a reference variable

```
public List<Contact> Contacts
{
    get
    {
        return this._contacts;
    }
}
```

## ■ Accessing the contained objects from the client

```
foreach (Contact contact in customer.Contacts)
{
    Console.WriteLine("Name = " + contact.FirstName + " " + contact.LastName );
    Console.WriteLine("Email = " + contact.Email);
    Console.WriteLine("Phone = " + contact.Phone);
}
```

# Accessing Contained Objects via Methods

```
public class Customer
{
    Fields

    Properties

    Methods

    #region More Methods

    public Location GetCorporateHQLocation()
    {
        if (this._locations.Count == 0)
            return null;

        // Assume that the Headquarters will
        // always be the first location in
        // the list
        return this._locations[0];
    }

    #endregion

    Helper Methods
}
}
```

# A Note About Garbage Collection

When reference to parent is released

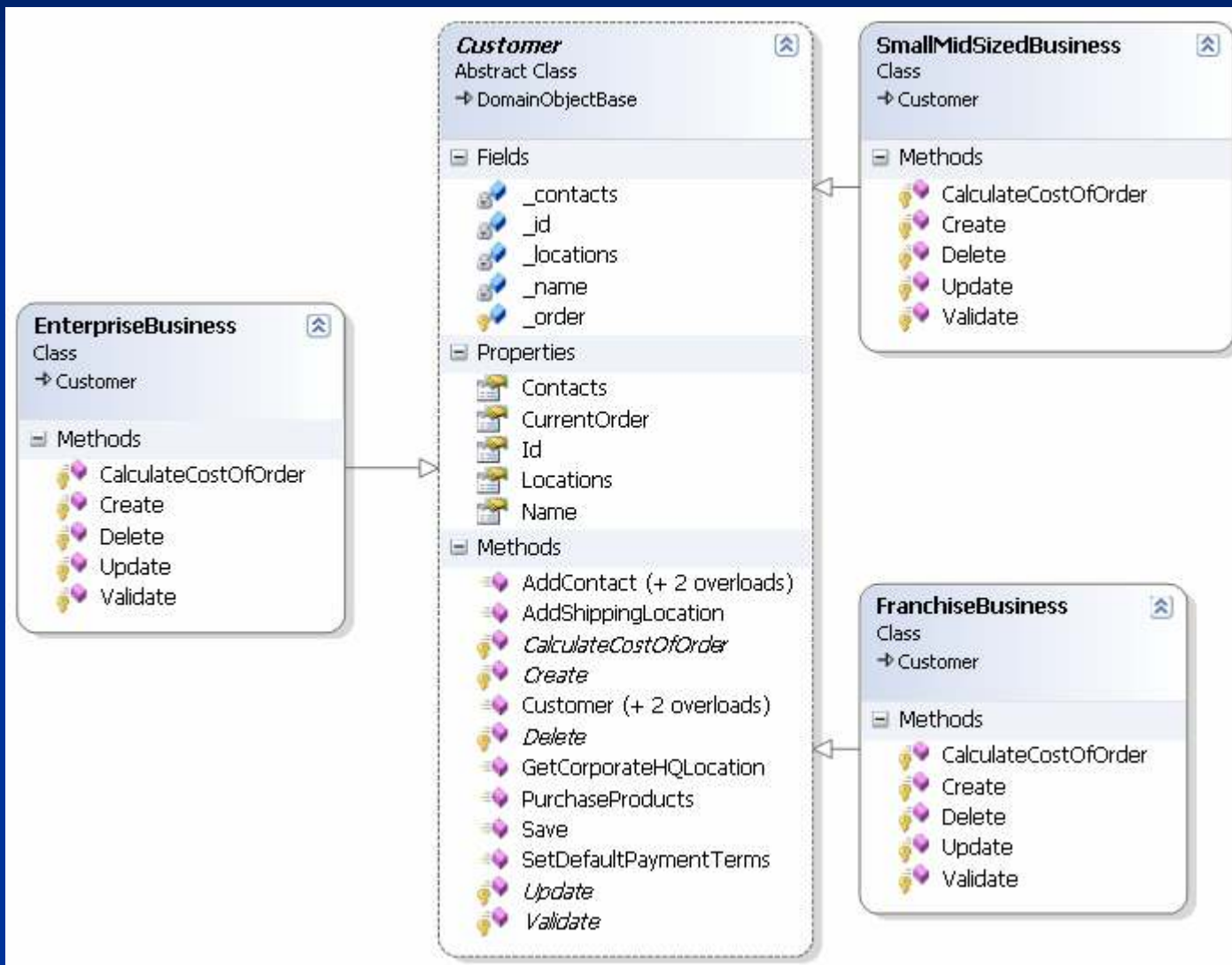
- Children will be marked for garbage collection IF they have no other references made to them
- If children are referenced, they may stay alive even if reference to parent is released

# Implementation Inheritance

- “Is A” relationships
- An overused example ...
  - A Dog is a Mammal
  - An Alligator is a Reptile

# Generalization vs. Specialization

Subtypes are specialized versions of their **Base Classes**



Each  
**Subtype  
(Subclass)**  
Inherits

- Fields
- Properties
- Methods

... from its  
parents

# Abstract Base Classes

- Sometimes it doesn't make sense to allow a base class to be created
- For example:
  - Might not want to create a plain old Customer
  - Might want to only create specializations ...
    - SmallMidSizedBusiness
    - FranchiseBusiness
    - EnterpriseBusiness

# Abstract Base Classes

- Abstract classes prohibit instantiation
  - Only allows subclass instantiation
- Abstract methods have no implementation
  - Subclasses must provide an implementation for abstract methods

```
public abstract class Customer
{
    Fields

    Constructors

    Properties

    Concrete Methods

    #region Abstract Methods
    protected abstract bool validate();
    protected abstract void Create();
    protected abstract void Update();
    protected abstract void Delete();
    protected abstract decimal CalculateCostOfOrder(decimal nonDiscountedCost);
    #endregion
}
```

# Creating a Subclass

In the current example, we can't create the base class, but we can create a subclass

```
Customer customer = CreateCustomer(custType);  
  
decimal orderCost = customer.PurchaseProducts(newOrder);  
  
Console.WriteLine("Customer type is : " + customer.GetType().ToString());  
Console.WriteLine("Order Cost is : " + orderCost);
```

The call to  
PurchaseProducts is  
actually a call to a  
Method on the base  
class  
... let's see how this  
works

```
// An example of a Factory Method  
public static Customer CreateCustomer(int custType)  
{  
    // Define variable of the base class type Customer  
    Customer customer = null;  
  
    // Instantiate a sub-class of Customer  
    switch (custType)  
    {  
        case 1:  
            customer = new EnterpriseBusiness();  
            break;  
        case 2:  
            customer = new FranchiseBusiness();  
            break;  
        case 3:  
            customer = new SmallMidSizedBusiness();  
            break;  
    }  
  
    return customer;  
}
```

# Base Class Methods Can Invoke Methods on Subclasses

- This method exists on the Abstract Base Class
  - Receives an Order
  - Calls a method on the Order to get a Cost
  - Invokes method on child class, and returns result

```
public decimal PurchaseProducts(Order newOrder)
{
    this._order = newOrder;

    Decimal nonDiscountedCost = this._order.GetNonDiscountedCost();

    return this.calculateCostOfOrder( nonDiscountedCost );
}
#endregion

#region Abstract Methods
protected abstract decimal calculateCostOfOrder(decimal nonDiscountedCost);
#endregion
```

# Polymorphism

- This class inherits from the base class *Customer*
- Polymorphism is demonstrated by the different behaviors or responses for the same method

```
public class EnterpriseBusiness : Customer
{
    protected override decimal calculateCostOfOrder(decimal nonDiscountedCost)
    {
        return Decimal.Multiply(nonDiscountedCost, new decimal(.75));
    }
}
```

```
public class FranchiseBusiness : Customer
{
    protected override decimal calculateCostOfOrder(decimal nonDiscountedCost)
    {
        return Decimal.Multiply(nonDiscountedCost, new decimal(.80));
    }
}
```

# Interfaces

# Interfaces

- Define a contract for communications
  - Once you publish them ... Don't change them!
- Classes can implement multiple interfaces

# Interface Definition

- Types that identify Methods (and their parameters and return types), and Properties
- Classes that implement interfaces must also implement all methods or properties defined in the interface
- These interfaces are used by ASP.Net pages in [www.DesignPatternsFor.Net](http://www.DesignPatternsFor.Net)

```
#region Interfaces for Master Pages
public interface IListPage
{
    void ResetPageTab(ref int currentTab);
    void SetListContent(MasterPages_MasterPage2 masterPage, int currentTab,
        ref string pageOpenerOverride, ref string pageOpenerSuffix);
}

public interface IContentPage
{
    void SetPageContent(MasterPages_MasterPage2 masterPage, int pageId);
}
#endregion
```

# Implementing an Interface

This code snippet shows an ASP.Net page that implements the interface *IContentPage*

```
public partial class _Default : System.Web.UI.Page, IContentPage
{
    protected void Page_Load(object sender, EventArgs e) {...}
    public void SetPageContent(MasterPages_MasterPage2 masterPage, int pageId) {...}
}
```

Required method, see prior slide

# Working with Interfaces

## Example:

An ASP.Net Master Page that checks to see if the child page being loaded implements one of two different interfaces

```
protected void Page_Load(object sender, EventArgs e)
{
    GetQueryStrings();

    if (this.Page is IContentPage)
    {
        this.SetPageContent();
        return;
    }

    if (this.Page is IListPage)
    {
        this.SetListContent();
    }
}
```

# Working with Interfaces

Example continued:

- This method is called from the master page's Page\_Load
- Cast current page reference to the interface *IContentPage*
- Call the *SetPageContent* method implemented on the object referred to by the variable *currentPage*

```
private void SetPageContent()
{
    IContentPage currentPage = this.Page as IContentPage;

    try
    {
        this.LogPageRequest( Convert.ToInt32(this.TabId) );
        currentPage.SetPageContent(this, _qStrings.PageId);
    }
    catch (Exception ex)
    {
        ExceptionHelper.HandleException(ex, true, true);
    }
}
```

# Interfaces vs. Traditional Inheritance

## Interfaces

- Classes that use interfaces need not be of similar types
- You don't need code reuse
- You want to ensure a common means to communicate with any number of class types

## Traditional Inheritance

- All classes share a common ancestry and exist in a hierarchy
- You want code reuse
- You want some consistency across all classes in the hierarchy

# This is Just the Beginning of Your Journey !

**Classes vs. Objects**

**Encapsulation**

**Inheritance**

**Reference Variables**

**Aggregation**

**Polymorphism**

**Fields**

**Instances**

**Properties**

**Abstract Classes**

**Methods**

**Overloading**

**Constructors**

**Interfaces**

**Overriding**

# What's Next?

- Static data and operations
- Virtual Methods
- Delegates and Events
- Design Patterns
  - Best practices in class design
  - Mapping database tables to objects
- And much more!

# Resources

- [www.DesignPatternsFor.Net](http://www.DesignPatternsFor.Net)