Session 3

- **Modular code using methods**
- **Object-Oriented Design**
- **Collections**
- **Exception Handling**

Quick aside: Modular code using “Methods”

- A.K.A. “Functions”
- A named block of code
- Reusable
- Must be defined within a class
- Can accept values to be passed in
- Consists of:
  - A return type (void if nothing is returned)
  - Parameters
  - Functionality (the code)

Code without methods

```
static void main()
{
  Console.Foreground = Color.Red;
  Console.WriteLine(“Welcome”);
  Console.ResetColor();
  console.WriteLine(“Here are the even numbers:”);
  for (int i = 0; i<100; i++)
  {
      if(i%2 == 0){
            Console.Writeline(i);
      }
  }
    Console.ReadKey(); //keep window open
}
```

Code using methods

```
static void main()
{
  PrintWelcome();
  console.WriteLine(“Here are the even numbers:”);
  PrintEvenNumbers();
  Console.ReadKey(); //keep window open
}
static void PrintWelcome()
{
  Console.Foreground = Color.Red;
  Console.WriteLine(“Welcome”);
  Console.ResetColor();
}
static void PrintEvenNumbers()
{
 for (int i = 0; i<100; i++)
  {
        if(i%2 == 0){
                Console.Writeline(i);
        }
  } }
```

Since this is a Console application, all methods are marked with the “static” keyword

Parts of a method: return type

- **What the method will return**
- **Use the “return” keyword**
- **If no return type, use “void”**

```
public int GetSecondHand()
{
  int result = DateTime.Now.Second;
  return result;
}

//calling the method
…
Int32 sec = GetSecondHand();
…
```

int is a c# alias for System.Int32

Method without return type

```
public void PauseScreen()
{
  Console.Write("(press any key to continue)");
  Console.ReadKey();
}

//calling the method
…
PauseScreen();
…
```

Parts of a method: parameters

- **What the method will accept**

```
public long Add(long num1, long num2)
{
   long result = num1 + num2;
   return result;
}

//calling the method
…
long sec = Add(552, 33);
…
```

Method Signature

- **Consists of:**
  - Name
  - Number, type & order of parameters

- **Enables method overloading**
- **Methods can have the same name as long as the signature is different**

```
public long Add(long num1, long num2){…}
public long Add(int num1, int num2) {…}
public long Add(string s1, string s2) {…}
```

Session 3

- Modular code using methods
- Object-Oriented Design
- Collections
- Exception Handling

What is Object-Oriented Design

- **Methodology for structuring code**
- **Allows for a software model of problem**
- **Manages complexity**
- **Model essential elements as software metaphore**
- **Basic structure is a class**
- **Data structure comprises**
  - Fields (private variables)
  - Attributes (or properties)
  - Behaviors (or actions)

A way of writing your code

What is a class

- A user-defined type
- Template for an object (or an instance)
- Defines everything an object will have
- "Instantiated" as an object

```
public class BankCustomer
{  //private fields
        private string _name;

   //public methods
   public string GetName()
    {
          return _name;
    }

   public void SetName(string newName)
    {
        _name = newName;
    }
}
```

The class is protecting its _name variable. The variable is only modified via the SetName () method.

This is called Encapsulation

An Object

- **Instance of a class at run-time**
- **Instantiate using the “new” keyword**
- **Self-contained module**
- **Contains data (state)**
- **Contains related functionality (behaviors)**

```
static void Main()
{
   BankCustomer customer;
   customer = new BankCustomer();
   customer.SetName(“Buzz”);
   ConsoleWriteLine(customer.GetName());
}
```

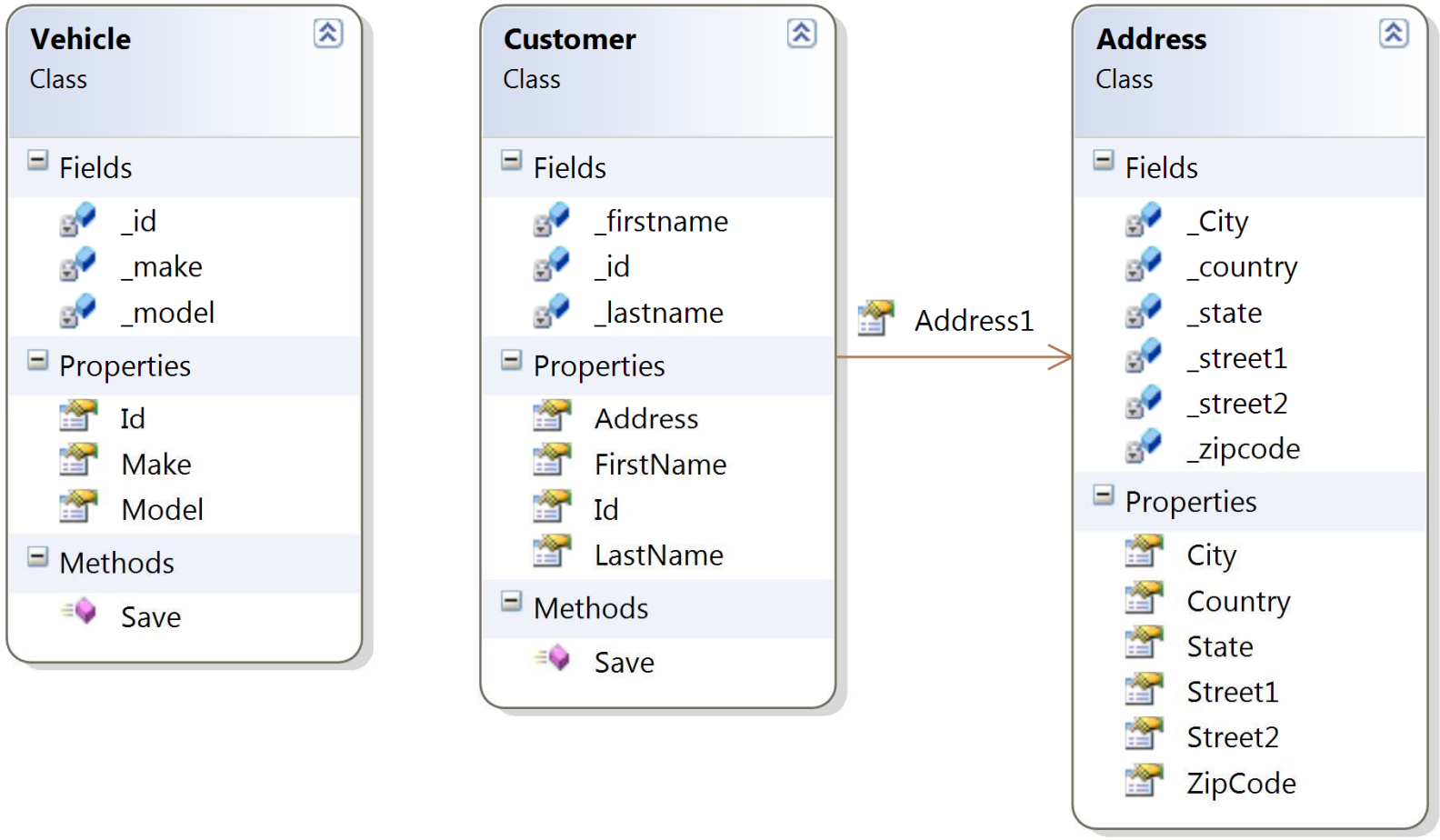
In the not too distant past…

```
/* customer info variables */
declare id as long
declare firstname as string
declare lastname as string
declare address as string
  …
/* vehicle info variables */
declare vehicleId as string
declare make as string
declare model as string
…
function main()
{
  //get customer info
  SaveCustomerInfo()
  //get vehicle info
  SaveVehicleInfo()
}
function SaveCustomerInfo()
{
  …
}
function SaveVehicleInfo()
{
  …
}
```

P.S. This is pseudo code

Think of a very complex project and imagine how complex things can get when programming in this way

Now days… OOP!


Vehicle
Class
Fields
_id
_make
_model
Properties
Id
Make
Model
Methods
Save
Customer
Class
Fields
_firstname
_id
_lastname
Properties
Address
FirstName
Id
LastName
Methods
Save
Address1
Address
Class
Fields
_City
_country
_state
_street1
_street2
_zipcode
Properties
City
Country
State
Street1
Street2
ZipCode

First Step – Data Modeling

- **Think about your problem domain**
- **What “things” you want to represent**
- **Generalize each as a class of objects (i.e. Plato’s ideal chair)**
- **What aspects of these classes are important?**
- **How do they relate to one another?**

Fields

- **A.K.A. variables**
- **Define an object's state**
- **Make them private**
- **Access them (read/write) via methods**

Methods

- **Actions or Behaviors of an object**
- **Can modify the internal state (value of fields)**
- **“Do Something”**
- **Special kinds of methods are:**
  - Constructor Method – gets called when you use “new”
  - Property Methods – syntactical sugar for getter/setter methods

Constructor Method

- **Name is same as class**
- **No return type**
- **May have more than 1 (different signatures)**
- **May accept parameters**
- **No parameter signature is the “default constructor”**

```
public class BankCustomer
{
    …
    public BankCustomer()
     {
          _name = “John Doe”;
     }
    public BankCustomer(string custName)
     {
          _name = custName;
     }
}
```

Properties

- **Object fields are traditionally access via methods know as "getter" and "setter" methods.**
  - GetCustomerName()
  - SetCustomerName(string newName)
- **C# provides a shortcut for this common pattern**
- **The compiler generates getter/setter methods**
- **Used to structure access to the classes fields**
- **Can be used to apply business rules:**
  - Is the current user allowed to see the value of the field
  - Is the value being set within range
- **Use "get" and "set"**
- **"value" has special meaning within a setter but is not a keyword**

Properties Example

```
public class BankCustomer
{
    …
    public string CustomerName
     {
          get
          {
            return _name;
          }
          set
          {
            if (!string.IsNullOrEmpty(value))
              _name = value;
          }
     }
}
```

Access Modifiers

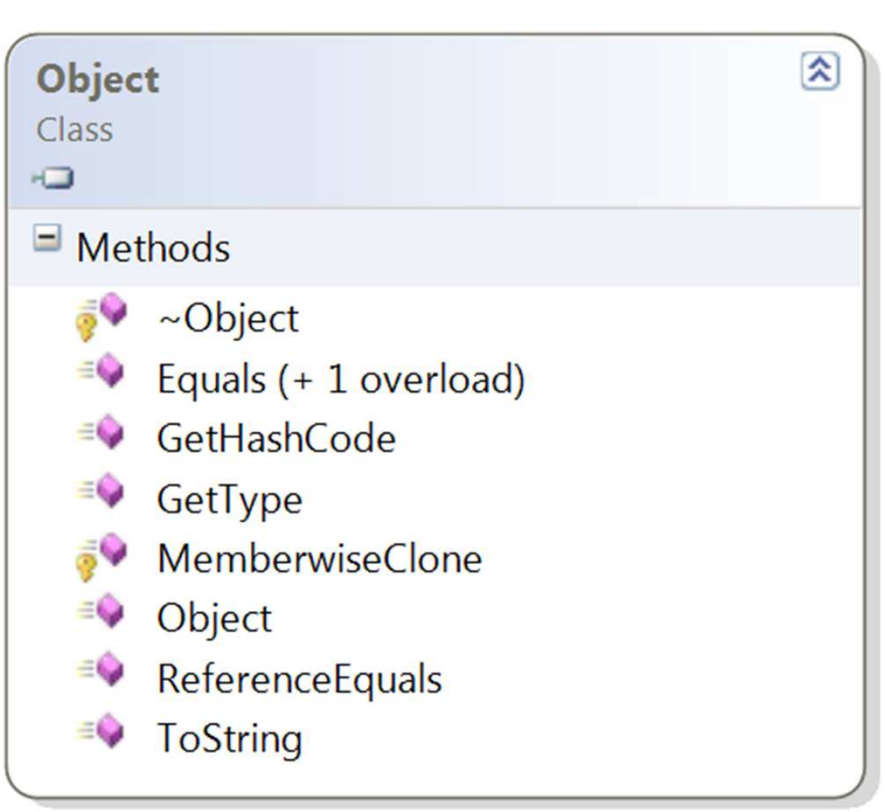
- **public - member is freely accessible inside and outside of the class in which it is defined.**
- **internal - member is only accessible to types defined in the same assembly.**
- **protected - member is accessible in the class in which it is defined and in classes which inherit that class.**
- **protected internal - member is accessible to types defined in the same assembly or to types in a derived assembly.**
- **private - member is only accessible in the class in which they are defined**

Inheritance

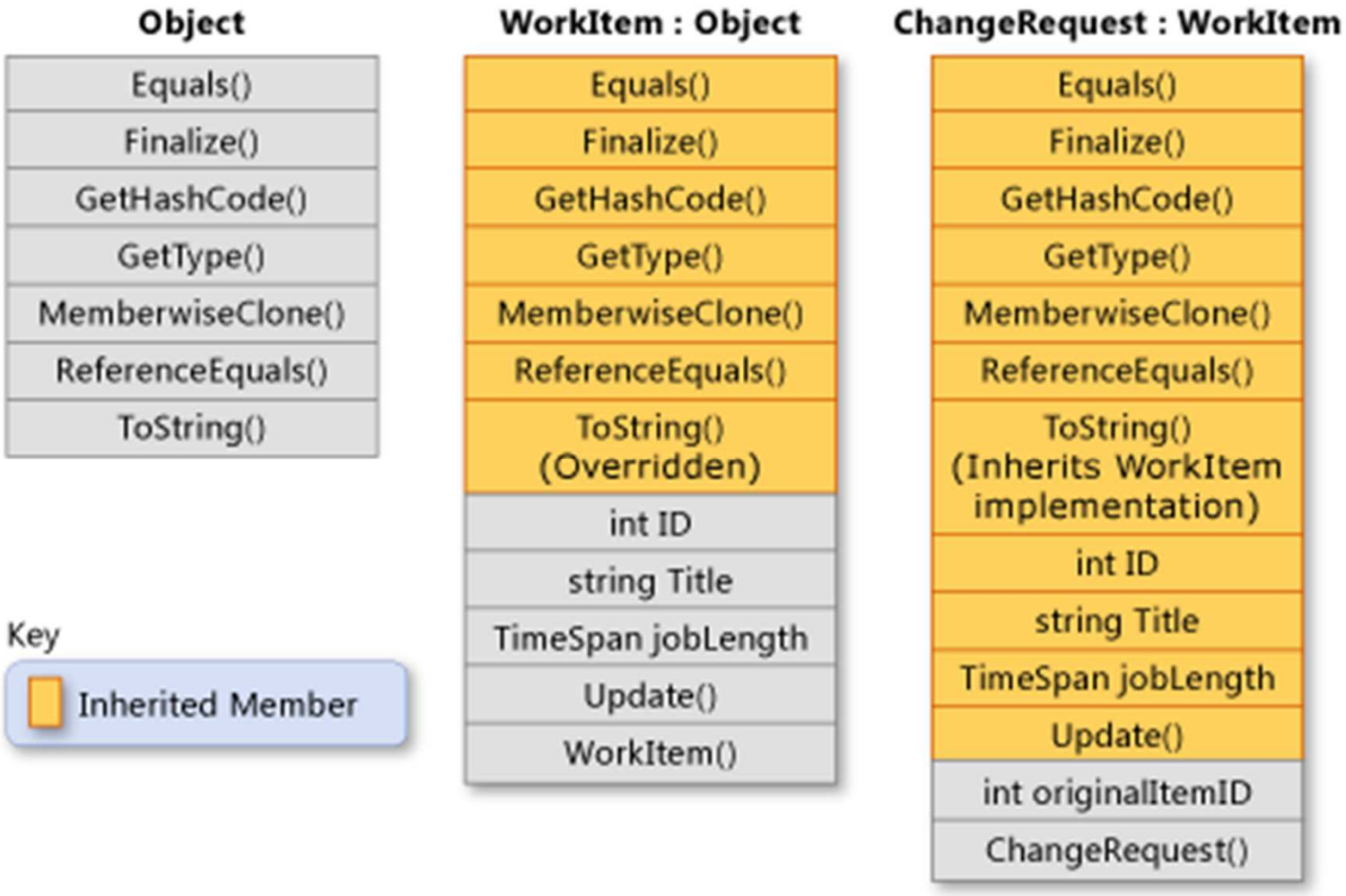
- **Ability to create a class that reuses, extends or modifies the behavior defined in another class.**
- **The *derived* class extends the *base* class**
- **Only one parent class (a.k.a. super class)**
- **Derived class represents different specialization**
- **In .NET, everything ultimately derives from “Object”**

Object, The Superclass

- **Implicit base class for all types in .NET**
- **Common functionality to all**
- **Most generic**



Inheritance Example



```
public class WorkItem : Object
{
  protected int ID { get; set; }
  protected string Title { get; set; }
  protected string Description { get; set; }
  protected TimeSpan jobLength { get; set; }
}
public class ChangeRequest : WorkItem
{
  protected int originalItemID {get; set; }
  protected void ChangeRequest()
  {
    …
  }
}
```

Session 3

- Modular code using methods
- Object-Oriented Design
- Collections
- Exception Handling

Collections

- **Arrays**
- **ArrayList Object**
- **Generic List Object**

In the beginning… Array

- **The only intrinsic collection type**
- **When you know how many elements**
- **Immutable collection**
- **Type safe**

Intrinsic types are those defined in the Base-Class Libraries (BCL for short)

```
string[] words;
words = new string[3];
words[0] = "hello";
words[1] = "there";
words[2] = "world";

int[] nums = new int[4] { 1, 2, 3, 4 };
```

Take Two… The ArrayList Object

- **System.Collections namespace**
- **When you don't know how many elements**
- **Mutable collection**
- **Not Type Safe (any object will do)**

```
ArrayList list = new ArrayList();
list.Add("hi");
list.Add(1);
list.Add(1.572);
list.Add(true);
```

Ideal… The Generic List Object

- **All the advantages of the ArrayList**
- **But is type safe**
- **System.Collections.Generic namespace**
- **Specify the type of elements using <type-name>**

```
List<int> list;
list = new List<int>();
list.Add(1);
list.Add(2);
list.Add("hi"); //error
```

Session 3

- Modular code using methods
- Object-Oriented Design
- Arrays
- Exception Handling

Exceptions

- **Unexpected problems**
- **Occur at run time**
- **Can be logic errors or hardware problems**
- **If the user doesn't handle, CLR will terminate the application**
- **Encapsulated as objects.**

Exceptions

- Use the throw keyword to raise an exception
- try...catch…finally keywords enable structured handling
- Can catch many types of exception
- Don’t specify the exception type and catch everything
- Finally block always executes

Exception Classes

- **Object that encapsulates information related to abnormal run-time occurrence**
- **Use those defined in FCL or create your own**
- **Derive from System.Exception**
- **Custom Exceptions should derive from System.ApplicationException**

http://msdn.microsoft.com/en-us/library/aa664610(VS.71).aspx

Example

```
try
{
  int num = Convert.ToInt32(“one”);
  Console.WriteLine(num);
}
catch (FormatException ex)
{
}
catch (OverflowException ex)
{
}
catch
{
}
finally
{
  Console.WriteLine(“Always Executes”);
}
```