Instructor: Bruce Reynolds

Introduction to Applications in C# Class 4

Concepts from Last Week

- Arrays & Collections
- Preview of File I/O and Exceptions

Homework Review

Concepts for This Week

- Enumerations
- Classes
  - Constructors
  - Methods
  - Properties
  - Namespaces

Classes

Classes

- A class represents an object in the problem space.
- A class is often called an abstraction, because it only represents the aspects of the object that are relevant to the application.
- Classes are the building blocks of object-oriented programming.

Classes

- Encapsulation brings data and behavior together in an object. In a class definition, data and behavior become the properties and methods in a class.
  - Procedural programming creates data structures and functions that manipulate the data structures.
- Classes hide the data and offer methods that manipulate the data.

Classes

- Remember the nomenclature
  - *Classes* are types. The class is the blueprint that can be use to make numerous objects of that type.
  - *Objects* are instances of a type. Objects are created at run-time. Many objects created from the same class can exist at the same time. Each instance may have different values of data.

Members

- What is in a class
  - Class members are the elements that make up a class. Each element may be data or a method.
    - Data – information that is referred to as the state of the object
    - Methods – behaviors that the object can perform

Members

- Data members
  - Also called *fields*
  - May be instance or static
    - Instance members are different for each instance
    - Static members are shared among all instances
- Method members
  - May also be instance or static
    - Instance members may change their instance
    - Static methods only can affect static data members

DEMO: Add the Pet Class

- Right-click the project in the Solution Explorer (not the solution).
- Select Add then Class.
- Name the Class Pet. (We'll create our own digipet.)

Class Syntax

- The boilerplate code added is:

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

namespace Class01
{
    class Pet
    {
    }
}
```

Class Syntax

- Leaving out the optional parts:
  - `class` *identifier class-body*
- *identifier* is the class' name.
- *class-body* is one or more class members:
  - Constant
  - Field (variable)
  - Method
  - Property
  - Constructor
  - … and more

Class Syntax

- A class is not simply a collection of code. That code is organized in a particular way.
  - Property – Defines the data of the object. The identifier is often a noun.
    - Field – Also contains data of the object.
  - Method – Defines the behavior of the object. The identifier is often a verb.
  - Constructor – To set the properties and fields when the instance is created.
- Class name – is often a noun. Note that a noun can be a concept.

Namespace Syntax

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

namespace Class01
{
    class Pet
    {
    }
}
```

- The class is part of a namespace, whether you explicitly declare it or not.
  - By default, the namespace is the "root namespace" (the name of the project).
  - You can find this in the Project Designer (demo).
- Namespaces organize the classes into groups.

Using Syntax

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

- When we instantiate the Pet class, we'll use this code:

```
Pet fido = new Pet();
```

- If the Pet class were in some other namespace, we'd use this code:

```
OtherNamespace.Pet fido = new OtherNamespace.Pet();
```

- By putting the using statement at the top of the file, we can leave out the OtherNamespace.
  - You can always access classes in the root namespace without explicitly adding or having a using statement.

Properties

- Properties provide access to the data of a class. Properties look like fields to the user of the class.
- Properties describe your objects and modify their behavior.
- A property is a programming construct that provides a shortcut to a get method and a set method.

DEMO: Add the Name Property

- Add this code to Pet class:

```
class Pet
{
    private string name = "";
    /// <summary>
    /// The name of the pet.
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

The Property Construct

- The property is a get/set pair that sets and retrieves a piece of data. That data can take may forms, but in the Name example, is a string.
- The *name* variable, also called a field or a property backing variable is declared as `private`. The calling code will not be able to access field.
- The property Name is declared `public`.
- **value** is a keyword in C#. It carries the value when a property is set by the calling code.

DEMO: Instantiate the Pet Class

- In program.cs file, add this code to Main:

```
Pet dog = new Pet();
dog.Name = "Spot";
Console.WriteLine(dog.Name);
```

- Use the debugger to step into the code and into the property statement.
  - Set a breakpoint on the second line.
  - F5 to run.
  - F11 to step into (not F10 to step over).

DEMO: Add Age

- Add an Age property to the Pet class.
  - What type would age be?
  - Try it!

DEMO: Age

```
private int age = 0;
// The age of the pet, in years.
public int Age
{
    get
    {
        return age;
    }
    set
    {
        if (value < 0)
        {
            Console.WriteLine("Pet can't be less than 0 years old.");
            age = 0;
        }
        else
        {
            age = value;
        }
    }
}
```

Properties

- The set method allows you to guarantee the "internal consistency" of the class.
  - At every step in the execution of the application, the Pet instance only contains "reasonable" data.
  - You could also throw an exception:

```
if (value < 0)
{
    throw new Exception("Pet can't be less than 0 years old.");
}
```

Properties

- Read-only or write-only properties
  - A property must define at least a *get* or a *set* but does not require that both be defined
- Access modifiers are allowed
  - *get* and *set* may either be public or private
- Automatic properties
  - Exclude the backing field
  - Example

```
public int Age { get; set; }
```

Access Modifiers

- Access modifiers determine the visibility of a class member
  - Public are visible to any user of the class
  - Private are usable only by class members
  - Protected (and protected internal) are usable by members of the class and derived classes
  - Internal are usable by users within the same assembly

Access Modifiers

- By default
  - Methods and data are private
  - Classes are internal
- Partial classes and methods
  - The partial keyword allows you to implement the class or method across multiple source code files.

Access Modifiers

- The *readonly* keyword
  - Fields marked *readonly* can only be set as part of the declaration of an instance of the class or in a constructor in the same class
  - Once a value is assigned to a *readonly* field, the field becomes constant

Enumerations

- An enum type is a distinct value type that declares a set of named constants. (spec)
- Leaving out the optional parts:
  - `enum` *identifier* { *member-declarations* }
- Enumerations are used when you want to allow only a limited number of values for a variable, and you can name them.

DEMO: Species Enum

- Add this code in the namespace, before the Pet class:

```
enum Species
{
    Cat,
    Dog,
    Cow
}
```

Class Designer

- You can use the Class Designer to generate code for properties.
- Right-click the project name in the Solution Explorer and select View Class Diagram.
- Find the Pet class and click the down arrow to open the class.
- Right-click the class and select Add and Property.
  - Name the property Species.
- Right-click the property and select Properties. Set the Type to Species.
- Right-click the property and select View Code.
  - Fill in the code you need.

DEMO: Species Property

```
Species species = Species.Cat;
/// <summary>
/// The species of the pet.
/// </summary>
public Species Species
{
    get
    {
        return species;
    }
    set
    {
        species = value;
    }
}
```

Constructors

- You must instantiate a class before you can call any of the members of the class (except static members).
- When you allocate memory for an instance of your class, using new, the operator returns a reference to the location of the instance.
- The constructor is the class method that describes how the new object is created.

Constructors

- Constructors allow you to set class data when the instance is created.
- Constructors can be used to force the developer to set properties when the instance is created. In this way, you can enforce consistency of the object.

Initializers

- You can use object initializers to initialize objects without invoking a constructor

```
class Vehicle
{
    private int speed = 0;
}
```

Constructors and Object Identity

- When you call new, you get back a reference to the new instance of the class.
- The variable you declare contains the reference of the new instance.
- The reference uniquely identifies the new instance.

DEMO: Public Default Constructor

- Because you haven't declared any constructor at all, a default public constructor has been added by the compiler.
- Add this one explicitly:

```
public Pet()
{
}
```

- It doesn't add much value, though.
  - If you made it private, you wouldn't be able to instantiate your class.

DEMO: Constructor w/ Parameters

- Add this constructor to the class:

```
public Pet(string name, int age, Species species)
{
    Name = name;
    Age = age;
    Species = species;
}
```

- In Program.cs, add this code and run it:

```
Pet cat = new Pet("Fluffy", 4, Species.Cat);
Console.WriteLine("The cat's name is {0} and its
age is {1}.", cat.Name, cat.Age);
```

Copy Constructor

- C# does not provide a copy constructor automatically
- The copy constructor, if defined, allows the programmer to control how the object is copied

```
public Vehicle(Vehicle original)
{
    speed = original.speed;
}
```

this keyword

- The keyword *this* refers to the current instance or is a reference to the currently executing object
  - Use *this* in the debugger to examine the object
  - Use *this* with variables to resolve name ambiguities

Methods

- Methods provide the behavior of your class. They make your class do things.
- Typically, methods will have names that are verbs. Examples include Add, Delete, Show, Send, and Receive.

Method Syntax

- Methods take parameters and return values.
- At the most basic, the syntax is:
  - *return-type member-name(formal-parameter list$_{opt}$) method-body*
- As an example:

```
int AddTwoNumbers(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

Method Syntax

- A method can return a value or **void**. **void** is keyword and is used if the method does not return a value.
- The **return** statement returns execution to the caller of the function and returns the result of the method.
  - It is not required if the return type is void.
- You can call the function from the instance variable, or from within the class.
- There are numerous modifiers you can add before the *return-type*, to control which code in your application can call the function.

DEMO: MakeNoise Method

- Add this code to the class:

```
public void MakeNoise()
{
    switch (this.Species)
    {
        case Species.Cat:
            Console.WriteLine("meow");
            break;
        case Species.Dog:
            Console.WriteLine("woof");
          break;
        case Species.Cow:
            Console.WriteLine("moo");
            break;
    }
}
```

DEMO: MakeNoise Method

- Add this code to Program.cs and run it:

```
Pet fluffy = new Pet("Fluffy", 4, Species.Cat);
fluffy.MakeNoise();
```

DEMO: Calculate Birth Year

- Add this code to the class:

```
public int CalculateBirthYear()
{
    int birthyear = 2008 - Age;
    return birthyear;
}
```

- Add this code to Program.cs:

```
Pet fluffy = new Pet("Fluffy", 4, Species.Cat);
fluffy.MakeNoise();
int birthyear = fluffy.CalculateBirthYear();
Console.WriteLine("Fluffy was born in {0}.",
birthyear);
```

Scope and Accessibility

- We've declared variables in these "locations" or **scope**:
  - Class level – age, name, and species.
    - These are usable (*in scope*) to class instances if they are declared with **public**.
    - These are usable (*accessible*) to all class members, regardless of **public/private**.
  - Method level – birthyear.
    - These are not usable (*in scope*) outside of the method.
  - Block level – line in the File I/O problem.
    - These are not usable (*in scope*) outside the block ({}) they are declared in.

DEMO: Method with Parameters

- Add this code to the class:

```
public void MakeAnnoyingNoise(int howAnnoying)
{
    for (int i = 0; i < howAnnoying; i++)
    {
        MakeNoise();
    }

}
```

- Add this code to Program.cs:

```
fluffy.MakeAnnoyingNoise(3);
```

Overloading methods

- Several methods may have the same name
  - They must differ by the number, order and type of parameters
  - Constructors may also be overloaded

#Region Statement

- The #region statement is used to organize your code in the Code Editor.
- It creates collapsible code sections.
- Example:

```
#region "Constructors"
public Pet()
{
}
#endregion
```

Summary

- We created:
  - One **class**, *Pet*.
  - Three **properties**, *Name*, *Age*, and *Species*.
  - Two **methods**, *MakeNoise* and *CalculateBirthYear*.
  - One **enumeration**, *Species*.

Readings 4

- Spec: Chapter 10 (for this week and next)
- Deitel & Deitel
  - Chapter 7 – Methods: A Deeper Look
  - Chapter 10 – Classes and Objects: A Deeper Look

Assignment 4