

Instructor: Bruce Reynolds

Introduction to Applications in C#

Class 5

Concepts from Last Week

- Classes
 - Constructors
 - Methods
 - Properties
- Enumerations

Homework Review

- Use the debugger!
- Read the help and look up examples. .

Concepts for Week 5

- Classes
 - Methods –overloaded and static
 - ToString
- Structures
- Immediate Window
- Bonus content
 - Indexers

Today's Problem

- We're going to create a system of classes and structures that represents the student.
- The system will include:
 - Address structure
 - Student class

Analysis

- Before we create classes, we need to determine what our application needs to do:
 - Record homework grades for each student.
 - Print homework grades for each student.
 - Calculate final grades at the end of the quarter.

Tasks

- Input the students and their addresses.
- Input the grades for each student.
- Calculate the final grades.

Analysis

Student
Class

Fields

Address

mGrades

mName

Properties

Grades

Name

Methods

CalculateAverage (+ 1 overlo...

Student (+ 1 overload)

StudentTestCode

ToString

Address
Struct

Fields

City

State

Methods

Address

ToString

State
Enum

Washington

Oregon

Idaho

Montana

Wyoming

Structures

Structure

- A structure is similar to a class, but is a value type:
 - Structures can contain the same members as a class.
 - Assignment copies the structure.
 - No parameterless constructor can be declared.
 - And a few other limitations.
- Structures are good for small objects where a “value type” meaning makes sense.

DEMO: State Enum

- Add a new item to the project, a code file.
- Name it Address.cs.
- Add this code to the file:

```
using System;
```

```
public enum State  
{  
    Washington,  
    Oregon,  
    Idaho,  
    Montana,  
    Wyoming  
}
```

DEMO: Address Structure

■ Add this code to the file:

```
public struct Address
{
    // public fields
    public string City;
    public State State;

    // constructor
    public Address(string city, State state)
    {
        this.City = city;
        this.State = state;
    }

    // ToString
    public override string ToString()
    {
        string output = string.Format("{0}\n{1}\n", City, State);
        return output;
    }
}
```

TIP: Overriding ToString

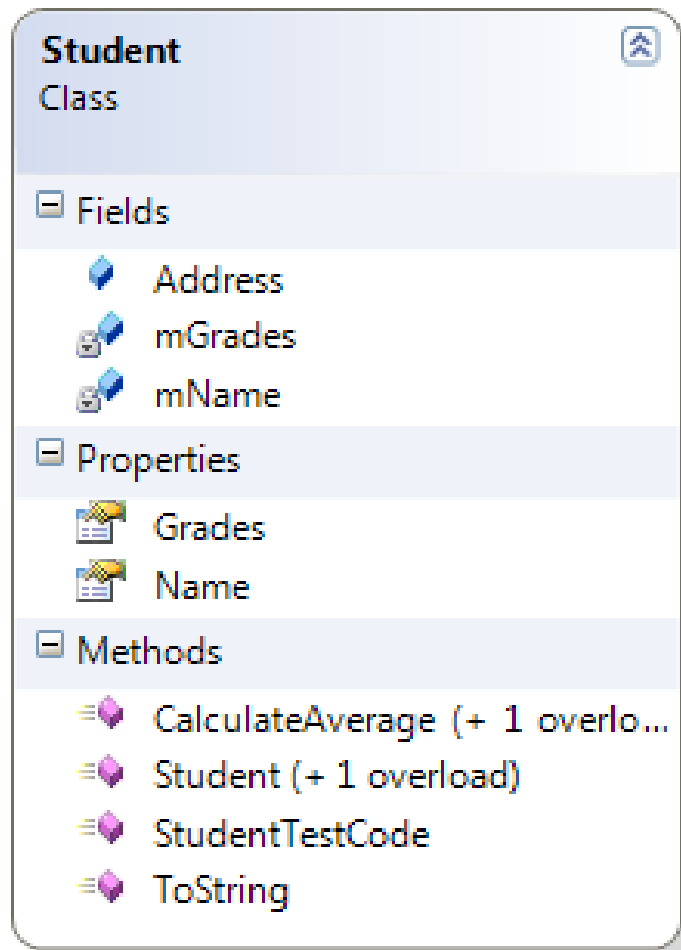
- Every structure and class has a ToString method that it inherits from its base class.
- If you don't override the base implementation, all you get is the class name if you print the class instance in `Console.WriteLine()`.
- You can override the ToString to print a custom string for your class or structure.

DEMO: A Bit of Test Code

- Instead of writing code in Main and executing, we can use the Immediate Window.
- Open the Immediate Window, if it's not already open.
 - Debug | Windows | Immediate
- Copy this code to the Immediate Window, one line at a time.

```
Address b = new Address ("Seattle", State.Washington);  
b.ToString();
```

The Student Class



Student.Address: A Public Field of a Structure Type

- Add a Student class file to your project.
- Add this code to the Student class:

```
public Address Address;
```

- Address is a public field (variable) of the Address structure type.
- The compiler can distinguish the type Address from the field (class variable) Address.
- Note that you have no control over how the calling code changes Address.

DEMO: Student Name

- Add the Name property to the Student class:

```
private string name = "";  
public string Name  
{  
    get { return name; }  
    set { name = value; }  
}
```

DEMO: Constructors

- Add the overloaded constructors to the Student class:

```
public Student(string name, Address address)
{
    this.Name = name;
    this.Address = address;
}
```

```
public Student(string name)
{
    this.Name = name;
}
```

- Note that now you have no public parameterless constructor. The calling code must use one of these two constructors.

Some Test Code

- In the Immediate Window:

```
Student ada = new Student("Ada");
```

```
ada.Name;
```

Grades: Analysis of the Problem

- Each student has grades for 3 weekly assignments.
- We need to track the scores and print them out.
- We'll implement:
 - Grades – a readonly property of an array of three integer values
 - ToString – a method that creates a string of the grades

DEMO: Grades Property

- Add this code for the Grades property.

```
private int[] grades = new int[3];  
public int[] Grades  
{  
    get  
    {  
        return grades;  
    }  
}
```

- The Grades property is read-only, because it has a getter and no setter. This prevents you from calling “new int[3]” again.

DEMO: Override ToString

■ Add this code to the Student Class:

```
public override string ToString()
{
    string output = this.Name + "\n" + Address.ToString();
    for (int week = 0; week < grades.Length; week++)
    {
        output += string.Format(
            "Week {0}: {1}\n", week + 1, grades[week]);
    }
    return output;
}
```

DEMO: Test from Main

- Add this code to Main in Program.cs:

```
Student ada = new Student("Ada");  
ada.Address = new Address("Seattle", State.Washington);  
ada.Grades[0] = 100;  
ada.Grades[1] = 80;  
ada.Grades[2] = 90;  
Console.WriteLine(ada);
```

Static Class Members

- Classes have code and define data.
 - Each call to new creates a new instance of the data.
- Static members (code and data) can be called even if no class instance exists.
 - They are declared with the *static* access modifier.
 - They can be both field and methods.
 - Static methods cannot access member methods or member data.

Some Test Code

■ Add this method to the Student class:

```
public static string StudentTestCode()
{
    Student Grace = new Student("Grace Hopper",
        new Address("Portland", State.Oregon));
    Student Ada = new Student("Ada Lovelace",
        new Address("Seattle", State.Washington));
    Student Roberta = new Student("Roberta Williams",
        new Address("Missoula", State.Montana));

    string output = "";
    Student[] students = { Grace, Ada, Roberta };
    foreach (Student student in students)
    {
        output += student.ToString();
    }
    return output;
}
```

Run the Test Code

- Add this code call to the Program class:

```
static void Main(string[] args)
{
    Console.WriteLine(Student.StudentTestCode());
}
```

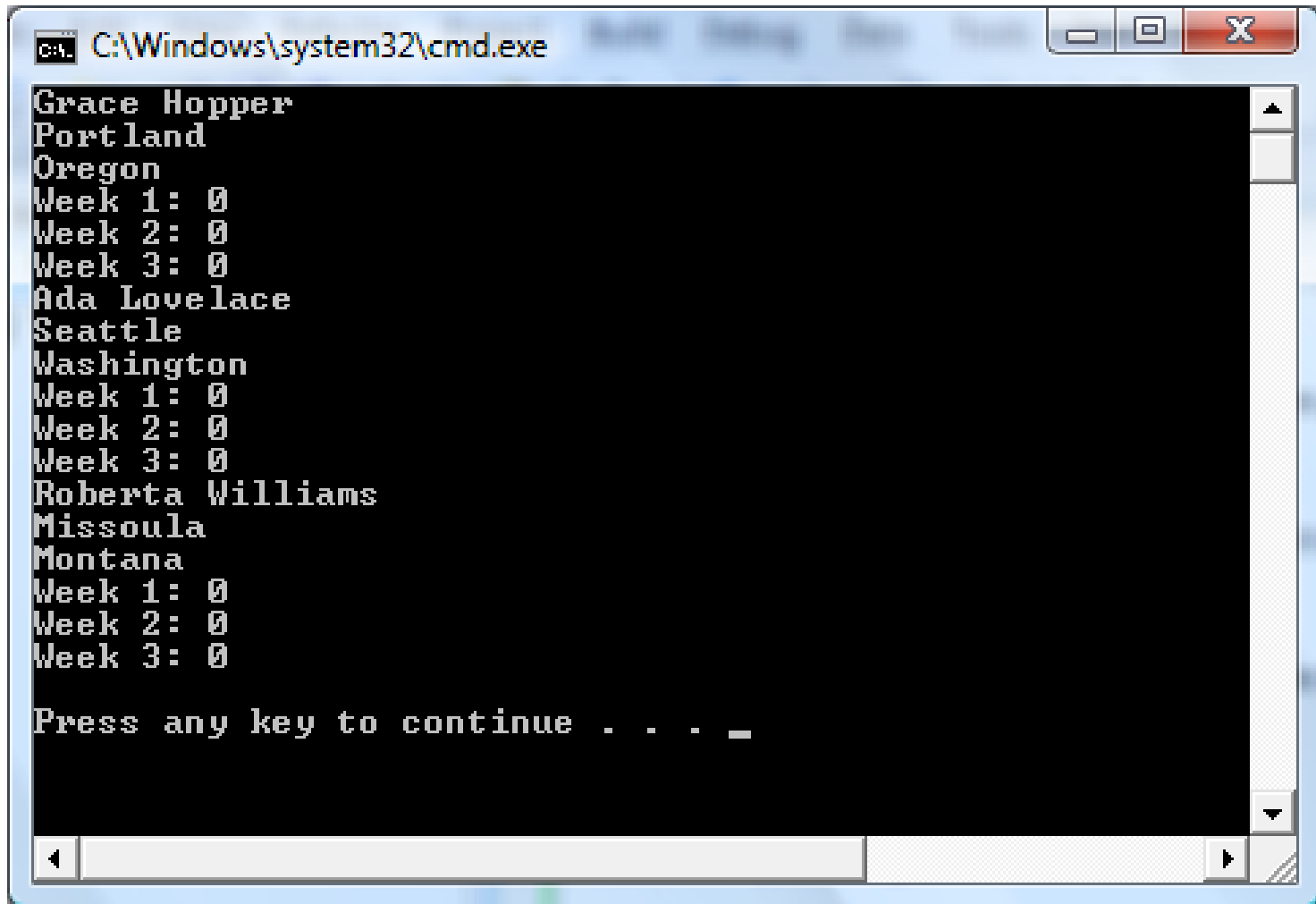
- Or, run this code in the Immediate window:

```
Student.StudentTestCode();
```

- Or, run this code in the Immediate window:

```
Console.WriteLine(Student.StudentTestCode());
```

Output



```
C:\Windows\system32\cmd.exe
Grace Hopper
Portland
Oregon
Week 1: 0
Week 2: 0
Week 3: 0
Ada Lovelace
Seattle
Washington
Week 1: 0
Week 2: 0
Week 3: 0
Roberta Williams
Missoula
Montana
Week 1: 0
Week 2: 0
Week 3: 0
Press any key to continue . . . _
```

Method Overloading

- We will write two versions of CalculateGrades.
 - They have different parameter lists.
- Method overloading simplifies the class interface, making it easier on the developer to find the right method to use.
 - First decision is to pick the method.
 - Second decision is to pick the overload.
- Overloaded methods cannot differ only on return type.

DEMO: CalculateGrades

- Add this code to the Student class:

```
public int CalculateAverage()  
{  
    int sum = 0;  
    for (int i = 0; i < grades.Length; i++)  
    {  
        sum += grades[i];  
    }  
    int average = sum / grades.Length;  
    return average;  
}
```

DEMO: Overload CalculateGrades

- Add this code to the Student class:

```
public int CalculateAverage(int howManyWeeks)
{
    int sum = 0;
    for (int i = 0; i < howManyWeeks; i++)
    {
        sum += grades[i];
    }
    int average = sum / howManyWeeks;
    return average;
}
```

File I/O

- Or course, you don't want to type in the class list every time.
- Load and Save methods are defined but not implemented.
- Load is a good motivation for using databases. 😊

DEMO: Test Code

- Add this code to Main in Program.cs:

```
Console.WriteLine(ada.CalculateAverage());
```

```
Console.WriteLine(ada.CalculateAverage(1));
```


Method Parameters

- Value parameters – AKA “pass by value”
- Reference parameters – AKA “pass by reference”
- Output parameters – AKA “out parameters”
- Parameter array – an array of parameters (not a parameter that is an array)

Reference Parameters

- When you create a method with parameters, you cannot (by default) change the value of the parameters.
 - BUT – if the parameter is a reference type, you can change the members that the reference value points to.
- To change the value of parameters, you need to use reference parameters.
 - Example on next slide.

By Reference Example

```
public static void Main()
{
    int one = 1;
    int two = 2;

    WrongSwap(one, two);
    Swap(ref one, ref two);
}

public static void WrongSwap(int a, int b)
{
    int c = a;
    a = b;
    b = c;
}

public static void Swap(ref int a, ref int b)
{
    int c = a;
    a = b;
    b = c;
}
```

TIP: String Input for Enumerations

- Consider this code:

```
public Address(string city, string state)
{
    this.City = city;
    this.State = (State)Enum.Parse(typeof(State), state, true);
}
```

- The Enum structure provides several methods for manipulating enumerations.

Indexer

- Indexers let you access an instance of a class as though it were an array.
- The indexed value does not return an instance of a class. It returns some object that's an internal collection of a class.
 - A Bag object would return Marble.
 - A Book object would return Page.
 - The ArrayList object returns Object.
- The code would look like this:

```
Student ada = new Student("ada");  
ada[0] = 1;  
int grade = ada[0];
```

Indexer for Student

```
// with error checking
public int this[int homeworkNumber]
{
    get
    {
        if ((homeworkNumber >= 0) && (homeworkNumber < grades.Length))
        {
            return grades[homeworkNumber];
        }
        else
        {
            throw new ArgumentOutOfRangeException("There is no homework for week " +
homeworkNumber);
        }
    }
    set
    {
        if ((homeworkNumber >= 0) && (homeworkNumber < grades.Length))
        {
            grades[homeworkNumber] = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException("There is no homework for week " +
homeworkNumber);
        }
    }
}
```

Reading 5

- Spec: Chapter 10
- Deitel & Deitel
 - Chapter 7 – Methods: A Deeper Look
 - Chapter 10 – Classes and Objects: A Deeper Look

Assignment 5
