COIS1020H: Programming for Computing Systems

Chapter 9
Using Classes and Objects

Understanding Class Concepts

- Classes are the basic building blocks of Object-Oriented programming
- Two Types of classes
  - Classes that are only application programs with a `Main()` method
  - Classes from which you instantiate objects
    - Can contain a `Main()` method, but it is not required
- Everything is an object
  - Every object **is a** member of a more general class
- An object is an **instantiation** of a class

Understanding Class Concepts (cont'd.)

- **Instance variables (**also called **fields)**
  - Data components of a class
- **State**
  - Set of contents of an object's instance variables
- **Instance methods**
  - Methods associated with objects
  - Every instance of the class has the same methods
- **Class client** or **class user**
  - Program or class that instantiates objects of another prewritten class (such as **Console**)

3

Creating a Class from Which Objects Can Be Instantiated

- **Class header** or **class definition** parts
  - An optional access modifier
  - The keyword `class`
  - Any legal identifier for the name of your class
- **Class access modifiers**
  - `public`
  - `protected`
  - `internal`
  - `private`

4

Creating a Class from Which Objects Can Be Instantiated (cont'd.)

```
class Employee
{
   // Instance variables and methods go here
}
```

Figure 9-1 Employee class shell

```
public class Employee
{
      // Instance variables and methods go here
}
```

- Better to use a modifier as default is `internal`

5

Creating Instance Variables and Methods

- When creating a class, define both its fields and its methods
- Field access modifiers
  - `new`, **`public`**, `protected`, `internal`, **`private`**, `static`, `readonly`, and `volatile`
- Most class fields are nonstatic and `private`
  - Provides the highest level of security

6

Creating Instance Variables and Methods (cont'd.)

- Using `private` fields within classes is an example of **information hiding**
- Most class methods are `public`
- `private` data / `public` method arrangement
  - Allows you to control outside access to your data
    - Like using a gas gauge to "see" the level of the gas
  - The `private` data fields are manipulated by well-defined (and programmer-defined) interfaces provided by the `public` methods

7

Creating Instance Variables and Methods (cont'd.)

```
public class Employee
{
    private int idNumber;
    public void WelcomeMessage()
    {
        Console.WriteLine("Welcome from Employee #{0}", idNumber);
        Console.WriteLine("How can I help you?");
    }
}
```

- Better to use a modifier as default is `internal`

- `Employee` class with `idNumber` field and `WelcomeMessage()` method
- Notice how the method has access to the `idNumber` field without having to pass the information as a parameter
  - instance methods have direct access to instance variables
- Also notice that **static** is NOT part of the method header – instance method

8

Creating Objects

- Declaring a class does not create any actual objects
  - Just an abstraction (like a method until it is invoked)
- Two-step process to create an object
  - Supply a type and an identifier

    `Employee bob;`
  - Create the object, which allocates memory for it

    `bob = new Employee();`
  - When you create an object, you call its constructor

    `Employee()` is a method call
- **Reference type**
  - Identifiers for objects are references to their memory addresses

Creating Objects (cont'd.)

```
using System;
public class CreateEmployee
{
   public static void Main()
   {
      Employee myAssistant = new Employee();
      myAssistant.WelcomeMessage();
   }
}
```

Invoking constructor

Invoking an object's method

Figure 9-4 The `CreateEmployee` program

Passing Objects to Methods

- You can pass objects to methods
  - Just as you can simple data types

```
using System;
public class CreateEmployee
{
   public static void Main()
   {
      Employee aWorker = new Employee();
      Employee bWorker = new Employee();
      DisplayMessage("First", aWorker);
      DisplayMessage("Second", bWorker);
   }
   public static void DisplayMessage(string mess, Employee emp)
   {
      Console.WriteLine("\n{0} employee's message", mess);
      emp.WelcomeMessage();
   }
}
```

```
First employee's message
Welcome from Employee #0
How can I help you?

Second employee's message
Welcome from Employee #0
How can I help you?
```

11

Creating Properties

- **Property**
  - A member of a class that provides access to a field of a class (very helpful for `private` fields)
  - Defines how fields will be set and retrieved
- Properties have **accessors**
  - `set` **accessors** for setting an object's fields
    - setters (also called mutators)
  - `get` **accessors** for retrieving the stored values
    - getters (or just accessors)
- **Read-only property**
  - Has only a `get` accessor

12

Creating Properties (cont'd.)

```
public class Employee
{
   private int idNumber;
   public int IdNumber
   {
      get
      {
         return idNumber;
      }
      set
      {
         idNumber = value;
      }
   }
   public void WelcomeMessage()
   {
       Console.WriteLine("Welcome from Employee #{0}", IdNumber);
       Console.WriteLine("How can I help you?");
   }
}
```

Notice nomenclature

Figure 9-8 Employee class with defined property

13

Creating Properties (cont'd.)

```
using System;
public class UseEmployeeProperties
{
   public static void Main()
   {
      Employee myEmployee = new Employee();
      myEmployee.IdNumber = 9;
      Console.WriteLine("ID number is {0}", myEmp.IdNumber);
      myEmployee.WelcomeMessage();
   }
}
```

```
ID number is 9
Welcome from Employee #9
How can I help you?
```

- Notice how the property is used by an object

  myEmployee.idNumber = 9; // this would result in an error
- **Implicit parameter:** one that is undeclared and that gets its value automatically
  - `value` becomes 9 in this case

14

Using Auto-Implemented Properties

- **Auto-implemented property**
  - The property's implementation is created for you automatically with the assumption that:
    - The `set` accessor should simply assign a value to the appropriate field
    - The `get` accessor should simply return the field
- When you use an auto-implemented property:
  - You do **not** need to declare the field that corresponds to the property

15

Using Auto-Implemented Properties (cont'd.)

```
using System;
public class CreateEmployee3
{
   public static void Main()
   {
      Employee aWorker = new Employee();
      aWorker.IdNumber = 3872;
      aWorker.Salary = 22.11;
      Console.WriteLine("Employee #{0} makes {1}",
         aWorker.IdNumber, aWorker.Salary.ToString("C"));
   }
}
public class Employee
{
   public int IdNumber {get; set;}
   public double Salary {get; set;}
}
```

Notice no private fields

Figure 9-11 An `Employee` class with no declared fields and auto-implemented properties, and a program that uses them

16

More About `public` and `private` Access Modifiers

- Occasionally you need to create `public` fields or `private` methods
  - You can create a `public` data field when you want all objects of a class to be able to access it
- A named constant within a class is always static without having to declare it so
  - Belongs to the entire class, not to any particular instance

17

More About `public` and `private` Access Modifiers (cont'd.)

```
public class Carpet
{
   public const string MOTTO = "Our carpets are quality-made";
   private int length;
   private int width;
   private int area;
   public int Length
   {
      get
      {
         return length;
      }
      set
      {
         length = value;
         CalcArea();
      }
   }
   public int Width
   {
      get
      {
         return width;
      }
      set
      {
         width = value;
         CalcArea();
      }
   }
   public int Area
   {
      get
      {
         return area;
      }
   }
   private void CalcArea()
   {
      area = Length * Width;
   }
}
```

Figure 9-13 The Carpet class

18

More About `public` and `private` Access Modifiers (cont'd.)

```
using System;
public class TestCarpet
{
   public static void Main()
   {
      Carpet aRug = new Carpet();
      aRug.Width = 12;
      aRug.Length = 14;
      Console.Write("The {0} X {1} carpet ", aRug.Width, aRug.Length);
      Console.WriteLine("has an area of {0}", aRug.Area);
      Console.WriteLine("Our motto is: {0}", Carpet.MOTTO);
   }
}
```

```
The 12 X 14 carpet has an area of 164
Our motto is: Our carpets are quality-made
```

Notice how the constant MOTTO is accessed

Figure 9-14 The `TestCarpet` class

19

Understanding the `this` Reference

- You might eventually create thousands of objects from a class
  - Each object does not need to store its own copy of each property and method
- **`this` reference**
  - Implicitly passed reference
- When you call a method, you automatically pass the `this` reference to the method
  - Tells the method which instance of the class to use

20

Understanding the `this` Reference (cont'd.)

```
public class Book
{
   private string title;
   private int numPages;
   private double price;
   public string Title
   {
      get
      {
         return title;
      }
      set
      {
         title = value;
      }
   }
   public void AdvertisingMessage()
   {
      Console.WriteLine("Buy it now: {0}", Title);
   }
}
```

Without `this` reference

Figure 9-16 Partially developed Book class

21

Understanding the `this` Reference (cont'd.)

```
public class Book
{
   private string title;
   private int numPages;
   private double price;
   public string Title
   {
      get
      {
         return this.title;
      }
      set
      {
         this.title = value;
      }
   }
   public void AdvertisingMessage()
   {
      Console.WriteLine("Buy it now: {0}", this.Title);
   }
}
```

With `this` reference

Figure 9-17 Book class with methods explicitly using `this` references

22

Understanding the `this` Reference (cont'd.)

```
using System;
public class CreateTwoBooks
{
   public static void Main()
   {
      Book myBook = new Book();
      Book yourBook = new Book();
      myBook.Title = "Silas Marner";
      yourBook.Title = "The Time Traveler's Wife";
      myBook.AdvertisingMessage();
      yourBook.AdvertisingMessage();
   }
}
```

```
Buy it now: Silas Marner
Buy it now: The Time Traveler's Wife
```

Figure 9-18 Program that declares two Book objects

23

Understanding Constructors

- **Constructor**
  - Method that instantiates an object
- **Default constructor**
  - Automatically supplied constructor without parameters
  - The only time that C# provides a default constructor is when there are no programmer-defined constructors
- **Default value of the object**
  - The value of an object initialized with a default constructor
    - Numeric fields are set to 0
    - Character fields are set to '\0'
    - Boolean fields are set to false
    - References (strings and objects) are set to `null`

24

Passing Parameters to Constructors

- You can create a constructor that receives arguments

```
public Employee(double rate)
{
   PayRate = rate;
}
```

Figure 9-22 Employee constructor with parameter

- Using the constructor

```
Employee partTimeWorker = new Employee(12.50);
```

25

Overloading Constructors

- C# automatically provides a default constructor
  - Until you provide your own constructor
- Constructors can be overloaded
  - You can write as many constructors as you want
    - As long as their argument lists do not cause ambiguity
    - Chooses constructor based on the signature

26

Overloading Constructors (cont'd.)

```
public class Employee
{
   public int IdNumber {get; set;}
   public double Salary {get; set;}

   public Employee()
   {
      IdNumber = 999;
      Salary = 0;
   }
   public Employee(int empId)
   {
      IdNumber = empId;
      Salary = 0;
   }
   public Employee(int empId, double sal)
   {
      IdNumber = empId;
      Salary = sal;
   }
   public Employee(char code)
   {
      IdNumber = 111;
      Salary = 100000;
   }
}
```

This parameterless constructor is the class's default constructor.

Figure 9-23 Employee class with four constructors

Overloading Constructors (cont'd.)

```
using System;
public class CreateSomeEmployees
{
   public static void Main()
   {
      Employee aWorker = new Employee();
      Employee anotherWorker = new Employee(234);
      Employee theBoss = new Employee('A');
      Console.WriteLine("{0,4}{1,14}", aWorker.IdNumber,
         aWorker.Salary.ToString("C"));
      Console.WriteLine("{0,4} {1,14}", anotherWorker.IdNumber,
         anotherWorker.Salary.ToString("C"));
      Console.WriteLine("{0,4}{1,14}", theBoss.IdNumber,
         theBoss.Salary.ToString("C"));
   }
}
```

```
999          $0.00
234          $0.00
111    $100,000.00
```

Figure 9-24 CreateSomeEmployees program

Classes and Instances

- Recall
  - Classes are defined to represent a single concept or service.
  - Each instance of the class contains different data (stored in the instance variables or fields)
  - The instances all share the same design and have access to the same properties and instance methods

Another Example: Building a `Rectangle` class

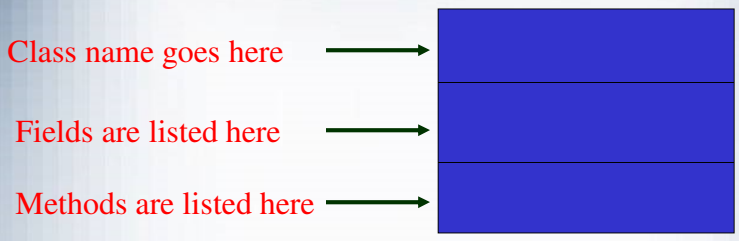
- A `Rectangle` object will have the following fields:
  - `length` - holds the rectangle's length.
  - `width` - holds the rectangle's width.

Building a `Rectangle` class

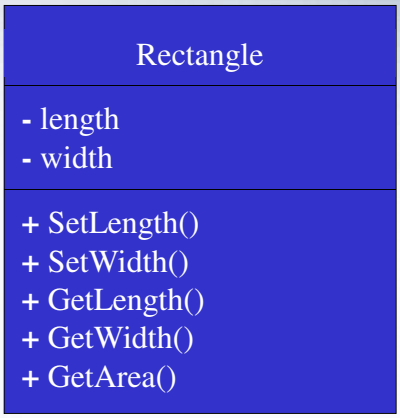
- The `Rectangle` class will also have the following methods (no Properties) yet:
  - `SetLength` - sets a value in an object's length field.
  - `SetWidth` - sets a value in an object's width field.
  - `GetLength` - returns the value in an object's length field.
  - `GetWidth` - returns the value in an object's width field.
  - `GetArea` - returns the area of the rectangle, which is the result of the object's length multiplied by its width.

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.



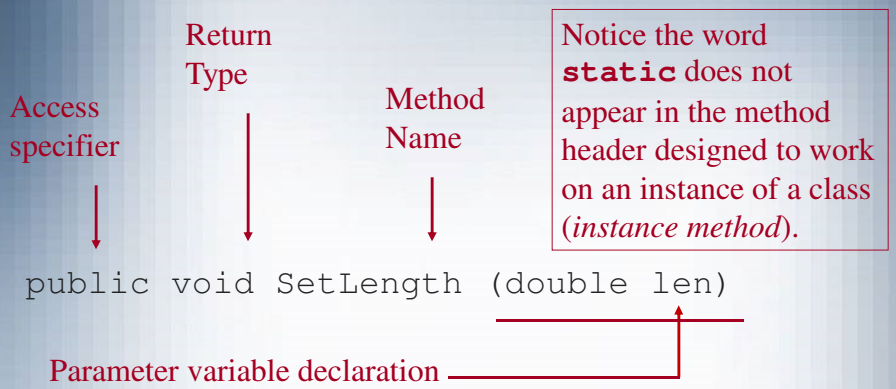
UML Diagram for `Rectangle` class



Writing the Code for the Class Fields

```
public class Rectangle
{
      private double length;
      private double width;
}
```

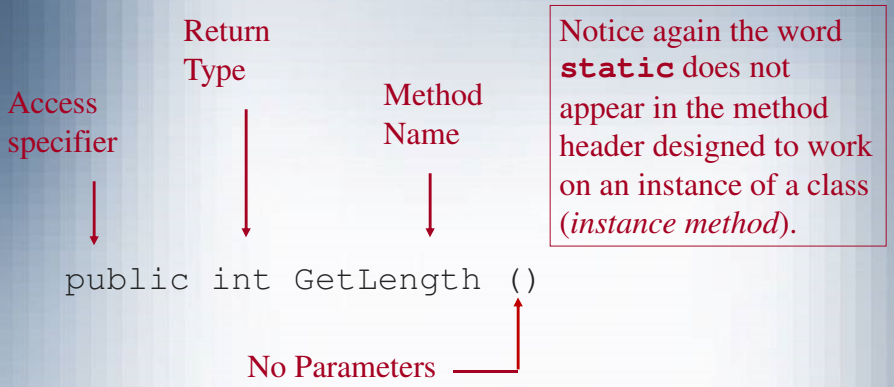
Header for the `SetLength` Method



Writing and Demonstrating the `SetLength` Method

```
// The method stores a value in the length field.
//
public void SetLength( double len )
{
    length = len;
}
```

Header for the `GetLength` Method



Notice again the word **`static`** does not appear in the method header designed to work on an instance of a class (*instance method*).

Writing and Demonstrating the `GetLength` Method

```
// The method returns the value in the length field.
//
public double GetLength()
{
    return length;
}
```

Rectangle.cs (Version 1)

```
public class Rectangle
{
  private double length;
  private double width;

// The SetLength method stores a value in the length field
  //   **param len The value to store in length.
  public void SetLength(double l)
  {
    length = l;
  }

  // The SetWidth method stores a value in the width field
  //   **param w The value to store in width.
  public void SetWidth(double w)
  {
    width = w;
  }
```

Rectangle.cs (cont'd)

```
// The GetLength method returns a Rectangle object's length
  //   *return The value in the length field.
  public double GetLength()
  {
    return length;
  }

// The GetWidth method returns a Rectangle object's width
//   **return The value in the width field.
  public double GetWidth()
  {
    return width;
  }

  // The GetArea method returns a Rectangle object's area.
  //   **return The product of length times width.
  public double GetArea()
  {
    return length * width;
  }
}
```

RectangleDemo.cs

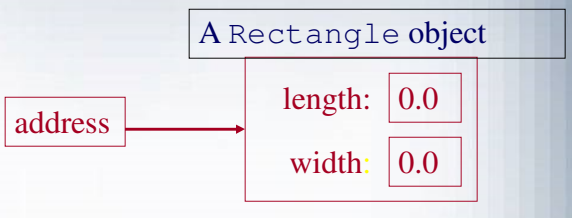
```
// This program demonstrates the Rectangle class'
// SetLength, SetWidth, GgetLength, GetWidth, and getArea methods.

using System;
public class RectangleDemo
{
  public static void Main()
  {
    Rectangle box = new Rectangle();

    box.SetLength(10.0);
    box.SetWidth(20.0);

    Console.WriteLine("The box's length is {0}", box.GetLength());
    Console.WriteLine("The box's width is {0}", box.GetWidth());
    Console.WriteLine("The box's area is {0}", box.GetArea());

    Console.ReadLine();
  }
}
```

Output:
The box's length is 10
The box's width is 20
The box's area is 200

Create a `Rectangle` class object

```
Rectangle box = new Rectangle();
```

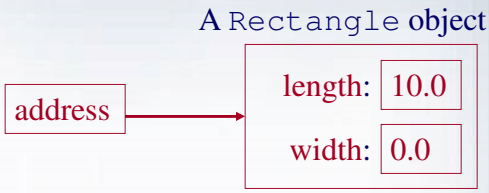
The `box` variable holds the address of the Rectangle object.



Calling the SetLength Method

```
box.SetLength(10.0);
```

The `box` variable holds the address of the Rectangle object.



This is the state of the box object after the `SetLength` *method executes.*

Accessor and Mutator Methods

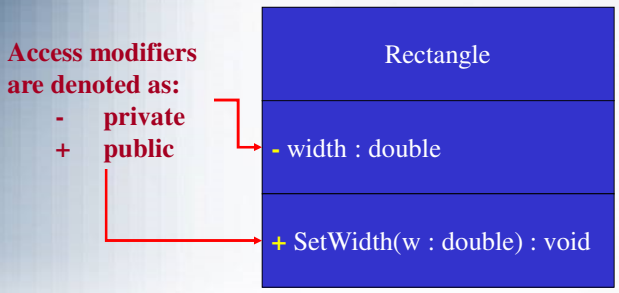
- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
  - Each field that the programmer wishes to be viewed by other classes needs an accessor.
- The methods that modify the data of fields are called *mutators*.
  - Each field that the programmer wishes to be modified by other classes needs a mutator.

Accessors and Mutators

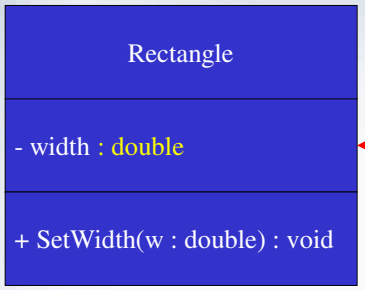
- For the rectangle example, the accessors and mutators are:
  - SetLength : Sets the value of the length field.
    public void SetLength(double len) …
  - SetWidth : Sets the value of the width field.
    public void SetWidth(double w) …
  - GetLength : Returns the value of the length field.
    public double GetLength() …
  - GetWidth : Returns the value of the width field.
    public double GetWidth() …
- Other names for these methods are *getters* and *setters*.

UML Data Type and Parameter Notation

- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.

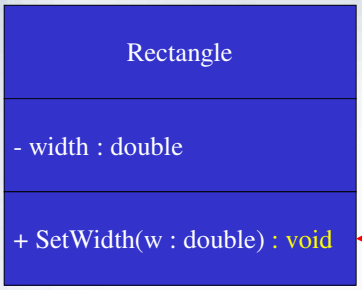


UML Data Type and Parameter Notation



Variable types are placed after the variable name, separated by a colon.

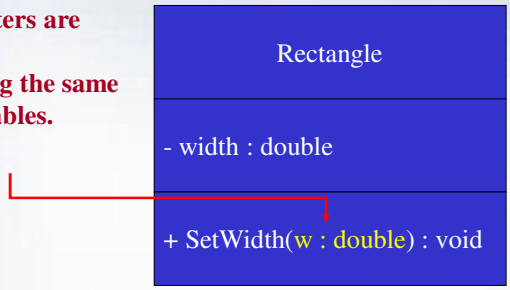
UML Data Type and Parameter Notation



Method return types are placed after the method declaration name, separated by a colon.

UML Data Type and Parameter Notation

Method parameters are shown inside the parentheses using the same notation as variables.



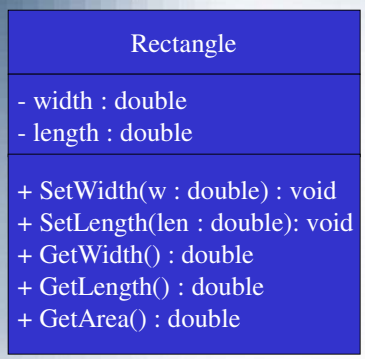
Converting the UML Diagram to Code

- Putting all of this information together, a C# class file can be built easily using the UML diagram.
- The UML diagram parts match the C# class file structure.

```
class header
{
   Fields
   Methods
}
```

Converting the UML Diagram to Code

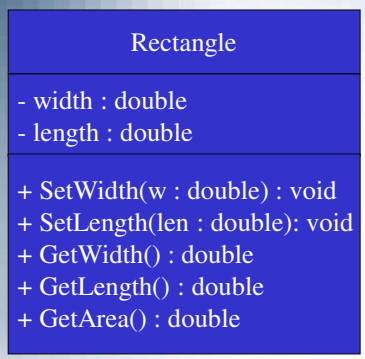
The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.




```
public class Rectangle
{
    private double width;
    private double length;

    public void SetWidth(double w) { }
    public void SetLength(double len) { }

    public double GetWidth() {
        return 0.0;
    }

    public double GetLength() {
        return 0.0;
    }
    public double GetArea() {
        return 0.0;
    }
}
```

Converting the UML Diagram to Code

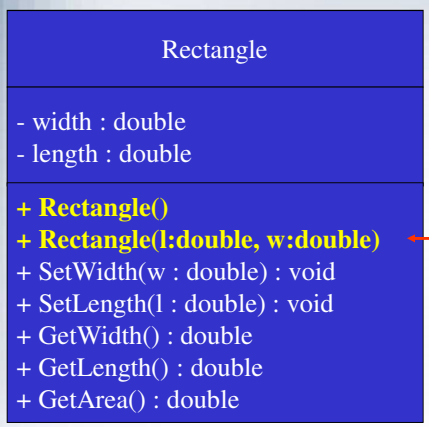
Once the class structure has been tested, the method bodies can be written and tested.




```
public class Rectangle
{
    private double width;
    private double length;

    public void SetWidth(double w) {
        width = w;
    }
    public void SetLength(double len) {
        length = len;
    }
    public double GetWidth() {
        return width;
    }
    public double GetLength() {
        return length;
    }
    public double GetArea() {
        return length * width;
    }
}
```

Constructors in UML

- In UML, the constructors are defined as follows:



Notice there is no return type listed for constructors.

Rectangle.cs (Version 2)

```
public class Rectangle
{
  private double length;
  private double width;

  // Parameterless constructor
  public Rectangle()
  {
     length = 0;
     width = 0;
  }

  // Parameter Constructor
  //   ** l  The length of the rectangle.
  //   ** w    The width of the rectangle.
  public Rectangle(double l, double w)
  {
     length = l;
     width = w;
  }
```

Rectangle.cs (cont'd)

```
// The SetLength method stores a value in the length field
//   **param len The value to store in length.
public void SetLength(double l)
{
    length = l;
}

// The SetWidth method stores a value in the width field
//   **param w The value to store in width.
public void SetWidth(double w)
{
    width = w;
}

// The GetLength method returns a Rectangle object's length
//   *return The value in the length field.
public double GetLength()
{
  return length;
}
```

Rectangle.cs (cont'd)

```
// The GetWidth method returns a Rectangle object's width
//   **return The value in the width field.
public double GetWidth()
{
  return width;
}

// The GetArea method returns a Rectangle object's area.
//   **return The product of length times width.
public double GetArea()
{
  return length * width;
}
}
```

Class Layout Conventions

- The layout of a source code file can vary by employer or instructor.
- Typically the layout is as follows:
  - Fields are listed first.
  - Methods are listed second.
    - The main method is sometimes first, sometimes last.
    - Accessors and mutators are typically grouped.
    - Constructors tend to go first in the methods
- There are tools that can help in formatting layout to specific standards.

Instance Fields and Methods

- Instance fields and instance methods require an object to be created in order to be used.
- Note that each room represented in this example can have different dimensions.

```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

RoomArea.cs

```
/ This program creates three instances of the Rectangle class.
using System;

public class RoomAreas
{
  public static void Main()
  {
     double number,       // To hold a number
           totalArea;    // The total area

     // Create three Rectangle objects.
     Rectangle kitchen = new Rectangle();
     Rectangle bedroom = new Rectangle();
     Rectangle den = new Rectangle();

     // Get and store the dimensions of the kitchen.
     Console.Write("What is the kitchen's length?");
     number = Convert.ToDouble(Console.ReadLine());
     kitchen.SetLength(number);
```

RoomArea.cs (cont'd)

```
Console.Write("What is the kitchen's width?");
 number = Convert.ToDouble(Console.ReadLine());
 kitchen.SetWidth(number);

 // Get and store the dimensions of the bedroom.
 Console.Write("What is the bedroom's length? ");
 number = Convert.ToDouble(Console.ReadLine());
 bedroom.SetLength(number);

 Console.Write("What is the bedroom's width? ");
 number = Convert.ToDouble(Console.ReadLine());
 bedroom.SetWidth(number);

 // Get and store the dimensions of the den.
 Console.Write("What is the den's length? ");
 number = Convert.ToDouble(Console.ReadLine());
 den.SetLength(number);
```

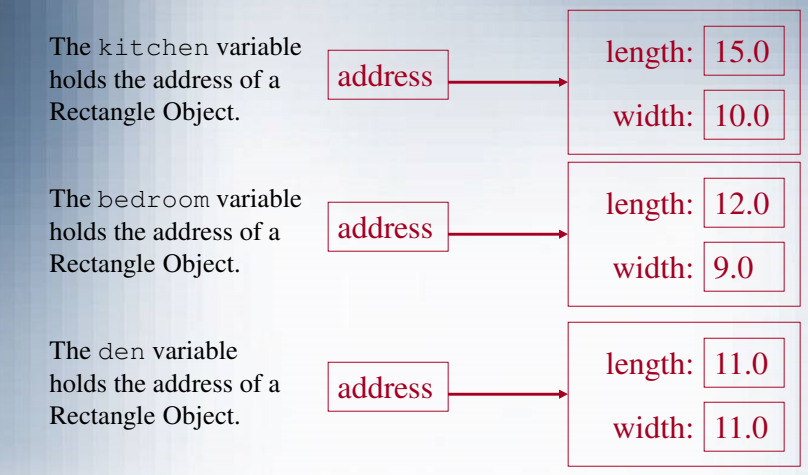
RoomArea.cs (cont'd)

```
    Console.Write("What is the den's width?");
    number = Convert.ToDouble(Console.ReadLine());
    den.SetWidth(number);

    // Calculate the total area of the rooms.
    totalArea = kitchen.GetArea() + bedroom.GetArea() + den.GetArea();

    // Display the total area of the rooms.
    Console.WriteLine("\nThe total area of the rooms is {0}", totalArea);

    Console.ReadLine();
  }
}
```

Output:

```
What is the kitchen's length?15
What is the kitchen's width?10
What is the bedroom's length? 12
What is the bedroom's width? 9
What is the den's length? 11
What is the den's width?11

The total area of the rooms is 379
```

States of Three Different Rectangle Objects



Properties

- C# has an alternative to using accessor and mutator methods, called *properties.*
- Properties allow us to modify and access private data members like accessor and mutator methods without using public methods.

Properties

- We use the key words `get`, `set` and `value` when defining properties.
- The `get` and `set` blocks replace the function of the accessor and mutator methods.
- The `get` block uses a variable named `value` that returns the actual value of the instance field.
- The `set` block uses the `value` to set the value of the instance field.

Properties

```
public double Length
{
   get
   {
        return length;
   }
   set
   {
       length = value;
   }
 }
```

Rectangle.cs (Version 3)

```
public class Rectangle
{
  private double length;
  private double width;

// No-Arg constructor
  public Rectangle()
  {
     length = 0;
     width = 0;
  }
  // Parameter Constructor
  public Rectangle(double l, double w)
  {
     length = l;
     width = w;
  }
```

Rectangle1.cs (cont'd)

```
// the get and set for the Width  property
  public double Width
  {
     get
     {   return width; }
     set
     {
        if (value < 0)
           width = 0;
        else
           width = value;
     }
  }
```

```
// the get and set for the Length property
  public double Length
  {
     get { return length; }
     set
     {
        if (value < 0)
           length = 0;
        else
           length = value;
     }
  }

  //  GetArea method
  public double GetArea()
  {
      return Length * Width;

  }
}
```

PropertyTest.cs

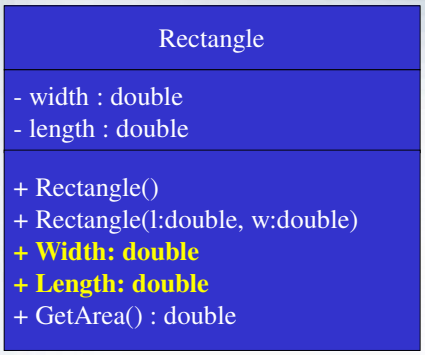
```
// This program demonstrates Rectangle1.cs.
// This tests C# properties.
using System;

public class PropertyDemo
{
  public static void Main()
  {
    Rectangle box = new Rectangle();

    box.Length = 5.0;
    box.Width = 15.0;

    Console.WriteLine("The box's length is {0}", box.Length);
    Console.WriteLine("The box's width is {0}", box.Width);
    Console.WriteLine("The box's area is {0}", box.GetArea());

    Console.ReadLine();
  }
}
```

Output:

The box's length is 5
The box's width is 15
The box's area is 75

UML Diagram with Properties

- In UML, the constructors are defined as follows:



Overloading Operators

- Overload operators
  - Enable you to use arithmetic symbols with your own objects
- Overloadable unary operators:

  `+ - ! ~ ++ -- true false`
- Overloadable binary operators:

  `+ - * / % & | ^ == != > < >= <=`
- Cannot overload the following operators:

  `= && || ?? ?: checked unchecked new typeof as is`
- Cannot overload an operator for a built-in data type
  - Cannot change the meaning of `+` for `ints`

70

Overloading Operators (cont'd.)

- When a binary operator is overloaded and has a corresponding assignment operator:
  - It is also overloaded
- Some operators must be overloaded in pairs:
  == with !=, and < with >
- Syntax to overload unary operators:
  type `operator` *overloadable-operator* (*type identifier*)
- Syntax to overload binary operators:
  type `operator` *overloadable-operator* (*type identifier, type operand*)

71

Overloading Operators (Class)

```
public class Book
{
   private string title;
   private int numPages;
   private double price;

   public string Title
   {
      get
      { return title; }
      set
      { title = value; }
   }
   public int NumPages
   {
      get
      { return numPages; }
      set
      { numPages = value; }
   }
   public double Price
   {
      get
      { return price; }
      set
      { price = value; }
   }
```

```
   public Book()
   {
      Title = "";
      NumPages = 0;
      Price = 0;
   }
   public Book(string title, int pages, double price)
   {
      Title = title;
      NumPages = pages;
      Price = price;
   }
   public static Book operator +(Book first, Book second)
   {
      const double EXTRA = 10.00;
      Book third = new Book();
      third.Title = first.Title + " and " + second.Title;
      third.numPages = first.NumPages + second.NumPages;
      if (first.Price > second.Price)
         third.Price = first.Price + EXTRA;
      else
         third.Price = second.Price + EXTRA;
      return third;
   }
}
```

72

Overloading Operators (Driver)

```
using System;
public class AddBooks
{
   public static void Main()
   {
      Book book1 = new Book("Visual C#", 840, 75.00);
      Book book2 = new Book("Moby Dick", 250, 16.00);
      Book book3;
      book3 = book1 + book2;
      Console.WriteLine("The new book is \"{0}\"", book3.Title);
      Console.WriteLine("It has {0} pages and costs {1:C}", book3.NumPages, book3.Price);
      Console.ReadLine();
   }
}
```

Output:
The new book is "Visual C# and Moby Dick"
It has 1090 pages and costs $85.00

Overloading Operators (Textbook version)

```
class Book
{
   public Book(string title, int pages, double price)
   {
      Title = title;
      NumPages = pages;
      Price = price;
   }
   public static Book operator+(Book first, Book second)
   {
      const double EXTRA = 10.00;
      string newTitle = first.Title + " and " +
         second.Title;
      int newPages = first.NumPages + second.NumPages;
      double newPrice;
      if(first.Price > second.Price)
         newPrice = first.Price + EXTRA;
      else
         newPrice = second.Price + EXTRA;
      return(new Book(newTitle, newPages, newPrice));
   }
   public string Title {get; set;}
   public int NumPages {get; set;}
   public double Price {get; set;}
}
```

Figure 9-32 Book class with overloaded + operator

Overloading Operators (Textbook version)

```
using System;
public class AddBooks
{
   public static void Main()
   {
      Book book1 = new Book("Silas Marner", 350, 15.95);
      Book book2 = new Book("Moby Dick", 250, 16.00);
      Book book3;
      book3 = book1 + book2;
      Console.WriteLine("The new book is \"{0}\"",
         book3.Title);
      Console.WriteLine("It has {0} pages and costs {1}",
         book3.NumPages, book3.Price.ToString("C"));
   }
}
```

```
The new book is Silas Marner and Moby Dick
It has 600 pages and costs $26.00
```

Figure 9-33 The AddBooks program

75

Overloading Operators (more)

- Let's add a unary operator (-) to the Book class
  - Assume we define this operator to cut the title and number of pages in half and reduce the price to 75% of the original
  - We have complete control over how the operators behave

```
public static Book operator -(Book tome)
{
   Book newBook = new Book();
   newBook.Title = tome.Title.Substring(0,tome.Title.Length/2);
   newBook.NumPages = tome.NumPages/2;
   newBook.Price = tome.Price * 0.75;
   return newBook;
}
```

76

Overloading Operators (Driver)

```
using System;
public class Addßooks2
{
   public static void Main()
   {
      Book book1 = new Book("Visual C#", 840, 75.00);
      Book book2 = new Book("Moby Dick", 250, 16.00);
      Book book3;
      book3 = book1 + book2;
      Console.WriteLine("The new book is \"{0}\"", book3.Title);
      Console.WriteLine("It has {0} pages and costs {1:C}", book3.NumPages, book3.Price);
      book3 = -book1;
      Console.WriteLine("The new book is \"{0}\"", book3.Title);
      Console.WriteLine("It has {0} pages and costs {1:C}", book3.NumPages, book3.Price);
      Console.ReadLine();
   }
}
```

Output:

```
The new book is "Visual C# and Moby Dick"
It has 1090 pages and costs $85.00
The new book is "Visu"
It has 420 pages and costs $56.25
```

77

Overloading Operators (one more)

- Add relational operators (>= and <=) to the Book class
  - They have to be added in pairs and unlike the first two which create a new Book, these return true or false
  - Notice that we can define the operators anyway we want

```
public static bool operator <=(Book b1, Book b2)
{
   if ((b1.NumPages / b1.Price) <= (b2.NumPages / b2.Price))
      return true;
   else
      return false;
}
public static bool operator >=(Book b1, Book b2)
{
   if ((b1.Title.Length > b2.Title.Length)
      return true;
   else
      return false;
}
```

78

Overloading Operators (Driver)

```
using System;
public class AddBooks3
{
   public static void Main()
   {
      Book book1 = new Book("Visual C#: For Fun and Profit", 840, 75.00);
      Book book2 = new Book("Moby Dick", 250, 16.00);
      if (book1 >= book2)
         Console.WriteLine("Purchase Book 1");
      else
         Console.WriteLine("Purchase Book 2");
      if (book1 <= book2)
         Console.WriteLine("Purchase Book 1");
      else
         Console.WriteLine("Purchase Book 2");
      Console.ReadLine();
   }
}
```

Output:
Purchase Book1
Purchase Book 1

Declaring an Array of Objects

- You can declare arrays that hold elements of any type
  - Including objects
- Example

```
Employee[] empArray = new Employee[7];

for(int x = 0; x < empArray.Length; ++x)
   empArray[x] = new Employee();
```

Understanding Destructors

- **Destructor**
  - Contains the actions you require when an instance of a class (object) is destroyed
- Most often, an instance of a class is destroyed when it goes out of scope (program terminates)
  - Can be used to clean up when the object is destroyed (release memory)
- Explicitly declare a destructor
  - Identifier consists of a tilde (~) followed by the class name

Understanding Destructors (cont'd.)

```
public class Employee
{
   public int idNumber {get; set;}
   public Employee(int empID)
   {
      IdNumber = empID;
      Console.WriteLine("Employee object {0} created", IdNumber);
   }
   ~Employee()
   {
      Console.WriteLine("Employee object {0} destroyed!", IdNumber);
   }
}
```

Figure 9-39 Employee class with destructor

Understanding Destructors (cont'd.)

```
using System;
public class DemoEmployeeDestructor
{
   public static void Main()
   {
      Employee aWorker = new Employee(101);
      Employee anotherWorker = new Employee(202);
   }
}
```

```
Employee object 101 created
Employee object 202 created
Employee object 202 destroyed
Employee object 101 destroyed
```

Figure 9-40 DemoEmployeeDestructor program

Summary

- You can create classes that are only programs with a `Main()` method and classes from which you instantiate objects
- When creating a class:
  - Must assign a name to it and determine what data and methods will be part of the class
  - Usually declare instance variables to be `private` and instance methods to be `public`
- When creating an object:
  - Supply a type and an identifier, and you allocate computer memory for that object

Summary (cont'd.)

- A property is a member of a class that provides access to a field of a class
- Class organization within a single file or separate files
- Each instantiation of a class accesses the same copy of its methods
- A constructor is a method that instantiates (creates an instance of) an object
- You can pass one or more arguments to a constructor

85

Summary (cont'd.)

- Constructors can be overloaded
- You can pass objects to methods just as you can simple data types
- You can overload operators to use with objects
- You can declare arrays that hold elements of any type, including objects
- A destructor contains the actions you require when an instance of a class is destroyed

86