CS 683 Emerging Technologies
Spring Semester, 2003
Doc 21 C# Classes
Contents

References

C# Language Specification, http://download.microsoft.com/download/0/a/c/0acb3585-3f3f-4169-ad61-efc9f0176788/CSharp.zip

Programming C#, Jesse Liberty, O'Reilly, Chapters 4

2003 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Classes

Format for declaring a class

[Attributes] [class-modifiers] class identifier [class-base] class-body [;]

Class Modifiers

- new
- public
- protected
- internal
- private
- abstract
- sealed

Class Members

- constant
- field
- method
- property
- event
- indexer
- operator
- constructor
- destructor
- static-constructor
- type

Sample Class

```
public class BankAccount
	{
	string name;				// private is default access
	int balance = 0;

	public BankAccount(string customerName)
		{
		name = customerName;  //No chaining of constructors?
		}

	public BankAccount(string customerName, int initialbalance)
		{
		this.name = customerName;
		balance = initialbalance;
		}

  //Compiler complains if there is no main & don't give correct flags
	public static void Main()
		{
		BankAccount example = new BankAccount("Roger");
		}
	}
```

Access Modifiers for Members

- Public

  Accessible by any method of any class

- Protected

  Access limited to the containing class or types derived from the containing class

- Internal

  Access limited to this program (assembly)

- Protected internal

  Access limited to this program or types derived from the containing class

- Private

  Access limited to the containing type

  Default access level

Initializers

```
public class BankAccount
	{
	int balance = 0;
   }
```

Initializers done before constructors

Default Values

Type	Default Value
Numeric (int etc)	0
bool	false
char	‘\0’ (null)
enum	0
reference	null

this

pointer to current object

Static

Static methods
Static fields
Static initializers
Static Constructors

```
public class BankAccount
     {
     string name;
     int balance = 0;
     static string bank = "ComputerBank";
     static float interestRate;

    // Called sometime between start of program and
    // first time a BankAccount object is created

     static BankAccount()  //No access modifier allowed
          {
          interestRate = 1;
          }

     public override string ToString()
          {
          return string.Concat("Account for", name , " in bank " ,
      bank);
          }
     }
```

Destructors

```
using System;
class A
{
   ~A() {
      Console.WriteLine("A's destructor");
   }
}

class B: A
{
   ~B() {
      Console.WriteLine("B's destructor");
   }
}

class Test
{
   static void Main() {
      B b = new B();
      b = null;
      GC.Collect();           //Collect() method is not required by standard
      GC.WaitForPendingFinalizers();
   }
}
```

Output

```
B’s destructor
A’s destructor
```

Destructors

Used to release system resources held in an object

Called when the object is garbage collected

Cannot call directly

Compiler maps destructor to:

```
  override protected void Finalize() {}
```

A class with a destructor cannot implement Finalize()

You cannot implement override protected void Finalize() {}

```
class A
{
  override protected void Finalize() {}  // error
  public void F() {
    this.Finalize();          // error
  }
}

class A
{
  void Finalize() {}          // permitted
}
```

Dispose()

Since you can not call the destructor C# has Dispose()

You can call Dispose()

Example From Text

```
using System;

class Testing : IDisposable
    {
    bool isDisposed = false;
    protected virtual void Dispose(bool disposing)
        {
        if (isDisposed) return;
        if (disposing)
            {
            //Not in destructor so can access
            // other object here
            }
        // perform clean up here
        isDisposed = true;
        }

    public void Dispose()
        {
        Dispose(true);
        // tell GC not ot finalize
        GC.SuppressFinalize(this);
        }

    ~Testing()
        {
        Dispose(false);
        }
    }
```

Automatically calling Dispose with using

```
class Tester
	{
	public static void Main()
		{
		using (Font smallFont = new Font("Arial" , 10.0f))
			{
			// use smallFont here
			} // Dispose called on largeFont here

		Font largeFont = new Font("Courier", 12.0f);

		using (largeFont)
			{
			// use large font here
			} // Dispose called on largeFont here

		}
	}
```

Some Details

General Form

```
using ( resource-acquisition ) embedded-statement
```

When ResourceType is a value type, expanded to:

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      ((IDisposable)resource).Dispose();
   }
}
```

When ResourseType is a reference type, expanded to

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      if (resource != null) ((IDisposable)resource).Dispose();
   }
}
```

More Details

Variables declared in resource-acquisition are read only

```
 using (Font smallFont = new Font("Arial" , 10.0f))
```

All resources in using must implement Idisposable

You can declare more than one variable of the same type

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

Overloading Method

C# allows classes to overload a method name

Method signature includes

- Method name
- Argument types, order and number
- Does not include return type

Two methods/constructors implemented in the same class must have different signatures

Parameters

There are four kinds of formal parameters:

- Value parameters – default
- Reference parameters (ref)
- Output parameters (out)
- Parameter arrays (params)

Note that ref and out are part of method signature, but params is not

Value Parameters

A local copy of the actual parameter is made

```
using System;

class Tester
	{
	public static void Main()
		{
		int start = 0;
		sample(start);
		Console.WriteLine( start);
		}

	public static void sample(int value)
		{
		value = 10;
		}
	}
```

Output

```
0
```

Value Parameters and Reference Types

Reference types are passed as value parameters

Reference types are pointers to the heap

Method gets a pointer to an object on the heap

Method can change the state of the object

Reference Parameters

Both the method definition and the caller must declare the ref type

```
using System;

class Tester
	{
	public static void Main()
		{
		int start = 0;
		sample(ref start);
		Console.WriteLine( start);
		}

	public static void sample(ref int value)
		{
		value = 10;
		}
	}
```

Output

```
10
```

Output parameters

```
using System;

class Tester
    {
    public static void Main()
        {
        int start;
        sample(out start);
        Console.WriteLine( start);
        }

    public static void sample(out int value)
        {
        value = 10;
        }
    }
```

You must initialize a variable before passing it as a value or reference parameter

Params Parameter

Variable length parameter lists

```
using System;
class Test
{
   static void F(params int[] args) {
      Console.Write("Array contains {0} elements:", args.Length);
      foreach (int i in args)
          Console.Write(" {0}", i);
      Console.WriteLine();
   }

   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

Output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

Params and Overloading

```
using System;
class Test
{
   static void F(params object[] a) {
      Console.WriteLine("F(object[])");
   }

   static void F() {
      Console.WriteLine("F()");
   }

   static void F(object a0, object a1) {
      Console.WriteLine("F(object,object)");
   }

   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(1, 2, 3, 4);
   }
}
```

Output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

Params and object[] argument

```
using System;
class Test
{
   static void F(params object[] args) {
      foreach (object o in args) {
         Console.Write(o.GetType().FullName);
         Console.Write(" ");
      }
      Console.WriteLine();
   }

   static void Main() {
      object[] a = {1, "Hello", 123.456};
      object o = a;
      F(a);
      F((object)a);
      F(o);
      F((object[])o);
   }
}
```

Output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

Properties

```
using System;

class SampleProperty
    {
    int bar = 0;

    public int Foo
        {
        get
            {
            return bar;
            }
        set
            {
            bar = value;
            }
        }
    }

class Tester
    {
    public static void Main()
        {
        SampleProperty test = new SampleProperty();
        test.Foo = 12;
        Console.WriteLine( test.Foo++);
        }

    }
```

Properties Modifiers

new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

Read-Only Properties

```
class SampleProperty
	{
	int bar = 0;

	public int Foo
		{
		get
			{
			return bar;
			}

		}
	}
```

Write-Only Properties

```
class SampleProperty
	{
	int bar = 0;

	public int Foo
		{
		set
			{
			bar = value;
			}
		}
	}
```

Must Define a Property in one Property Declaration

```
class SampleProperty
	{
	int bar = 0;

	public int Foo
		{
		get
			{
			return bar;
			}
		}

	public int Foo			//Compile error
		{
		set
			{
			bar = value;
			}
		}
	}
```

Class Properties

```
class SampleProperty
	{
	static int bar = 0;

	public static  int Foo
		{
		get
			{
			return bar;
			}

		set
			{
			bar = value;
			}
		}
	}

class Tester
	{
	public static void Main()
		{
		SampleProperty.Foo = 12;
		Console.WriteLine( SampleProperty.Foo++);
		}

	}
```

Properties Reserve Names

```
using System;
class A
{
   public int P {
      get { return 123; }
   }
}
class B: A
{
   new public int get_P() {
      return 456;
   }
   new public void set_P(int value) {
   }
}
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      Console.WriteLine(a.P);
      Console.WriteLine(b.P);
      Console.WriteLine(b.get_P());
   }
}
```

Output

```
123
123
456
```

Readonly Fields

Assignment to a readonly field can be done in

- Declaration
- Constructor of same class

```
class SampleReadonly
	{
	public readonly int a = 12;
	public readonly int b;

	public static readonly int c = 0;
	public static readonly int d;

	public SampleReadonly()
		{
		b = 2;
		}

	static SampleReadonly()
		{
		d = 7;
		}
	}
```