CS 683 Emerging Technologies
Spring Semester, 2003
Doc 23 C# Object, Boxing, Operators & Struct
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 5-7

Object

Root of all classes

Methods in Object

- public Object()
- public virtual bool Equals(object obj)
- public static bool Equals(object objA, object objB)
- ~Object() (Finalize())
- public virtual int GetHashCode()
- protected object MemberwiseClone()
- public static bool ReferenceEquals(object objA, object objB)
- public Type GetType()

public Object()

Constructs a new instance of the System.Object class.

Usage

This constructor is called by constructors in derived classes

Can use directly to create an instance of the Object class.

Object instance can be used for synchronization

~Object() (Finalize())

Allows a System.Object to perform cleanup operations before the memory allocated for the System.Object is automatically reclaimed.

Default

The System.Object.Finalize implementation does nothing

System.Object.Finalize is automatically called after an object becomes inaccessible, unless the object has been exempted from finalization by a call to System.GC.SuppressFinalize

Conforming implementations of the CLI are required to make every effort to ensure that for every object that has not been exempted from finalization, the System.Object.Finalize method is called after the object becomes inaccessible. However, there may be some circumstances under which Finalize is not called. Conforming CLI implementations are required to explicitly specify the conditions under which Finalize is not guaranteed to be called.

For example, Finalize might not be guaranteed to be called in the event of equipment failure, power failure, or other catastrophic system failures.

.

public virtual bool Equals(object obj)

Returns true if obj is equal to the current instance

otherwise, false

Requirements

x, y, and z represent non-null object references

- x.Equals(x) returns true
- x.Equals(y) returns the same value as y.Equals(x)
- If (x.Equals(y) && y.Equals(z)) returns true, then x.Equals(z) returns true
- Successive invocations of x.Equals(y) return the same value as long as the objects referenced by x and y are not modified
- x.Equals(null) returns false for non-null x
- Implementations of System.Object.Equals should not throw exceptions

More Equals

Default

Returns true if the specified instance of Object and the current instance are the same instance; otherwise, it returns false.

Value types

Overriding Equals improves performance over default implementation

If you override Equals you should overload the equality operator

Reference types

Consider overriding Equals on a reference type if the semantics of the type are based on the fact that the type represents some value(s).

Most reference types should not overload the equality operator, even if they override Equals. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you should override the equality operator.

public virtual int GetHashCode()

It is recommended (but not required) that types overriding Equals also override GetHashCode.

Hashtables cannot be relied on to work correctly if this recommendation is not followed.

Returns

A System.Int32 containing the hash code for the current instance

Examples

```
public struct Point {
   int x;
   int y;

   public override int GetHashCode() {
   return x ^ y;
   }
}

public struct Int64 {
   long value;

   public override int GetHashCode() {
   return ((int)value ^ (int)(value >> 32));
   }
}
```

public static bool Equals(object objA, object objB)

Returns

true if one or more of the following statements is true:

- objA and objB refer to the same object,
- objA and objB are both null references,
- objA is not null and objA.Equals(objB) returns true;

otherwise returns false

If the Equals(object obj) implementation throws an exception, this method throws an exception.

public static bool ReferenceEquals(object objA, object objB)

Returns

True if a and b refer to the same object or are both null references; otherwise, false

Description

This static method provides a way to compare two objects for reference equality. It does not call any user-defined code, including overrides of System.Object.Equals.

protected object MemberwiseClone()

Performs a shallow copy

Does not call any constuctor

System.Object.MemberwiseClone is protected (rather than public) to ensure that from verifiable code it is only possible to clone objects of the same class as the one performing the operation (or one of its subclasses).

Cloning an object does not directly open security holes,
it does allow an object to be created without running any of its constructors.

public Type GetType() Example

```
using System;
public class MyBaseClass: Object {
}
public class MyDerivedClass: MyBaseClass {
}
public class Test {
  public static void Main() {
  MyBaseClass myBase = new MyBaseClass();
  MyDerivedClass myDerived = new MyDerivedClass();

  object o = myDerived;
  MyBaseClass b = myDerived;

  Console.WriteLine("mybase: Type is {0}", myBase.GetType());
  Console.WriteLine("myDerived: Type is {0}", myDerived.GetType());
  Console.WriteLine("object o = myDerived: Type is {0}", o.GetType());
  Console.WriteLine("MyBaseClass b = myDerived: Type is {0}",
b.GetType());
  }
}
```

output

```
mybase: Type is MyBaseClass
myDerived: Type is MyDerivedClass
object o = myDerived: Type is MyDerivedClass
MyBaseClass b = myDerived: Type is MyDerivedClass
```

Boxing

Value types are on the stack

Reference types are on the heap

Boxing wraps value types in a object and stored on the heap

Unboxing unwraps the value type and places it on the stack

```
using System;

public class BoxingExample
	{
	public static void Main()
		{
		int k = 123;

		// Parameter boxing
		Console.WrinteLine( k);

		// Variable boxing
		object boxed = k;

		// Cast is needed
		int unboxed = (int) boxed;
		}
	}
```

Nesting Classes

Nested class has access to out classed members

```
public class Outer
	{
	private int foo = 12;

	public class Inner
		{
		public int getFoo(Outer a )
			{
			return a.foo;
			}
		}
	}


public class Tester
	{
	public static void Main()
		{
		Outer a = new Outer();
		Outer.Inner  test = new Outer.Inner();
		int foo = test.getFoo( a);
		}
	}
```

Nested Classes

Nested class does not automatically have access to an instance of the outer class

```
public class Outer
    {
    private int foo = 12;

    public class Inner
        {
        public int getFoo(Outer a )
            {
            return foo;  // Compile Error
            }
        }
    }
```

Nested Classes and Inheritance

```
public class Outer
        {
        private int foo = 12;

        public class Inner : Point
                {
                public int getFoo(Outer a )
                        {
                        return a.foo;
                        }
                }
        }

public class Point
        {
       // Code not shown
        }
```

Operator Overload

Unary Operators
Binary Operator
Conversion Operators

Conversion Operators

Implicit – no cast required
Explicit – requires a cast

Implicit Conversion Example

```
public class Foo  {
      int value = 4;
      public Foo(int value)  { this.value = value; }

      public static implicit operator int(Foo f) {
            return f.value;
      }
      public static implicit operator Foo(int x)   {
            return new Foo(x);
      }
}

public class Tester {
      public static void Main(){
            Foo value = new Foo(8);
            int test = value;
            Foo newValue = 8;
      }
}
```

Chaining Implicit Casts

```
public class Tester
	{
	public static void Main()
		{
		Foo value = new Foo(8);
		int test = value;
		Foo newValue = 8;
		intCast( value);
		floatCast( value);
		}

	public static void intCast(int x ) { }
	public static void floatCast( float x ) { }

	}
```

Explicit Conversion

```
public class Foo
	{
	int value = 4;
	public Foo(int value)
		{
		this.value = value;
		}

	public static explicit operator int(Foo f)
		{
		return f.value;
		}
	public static explicit operator Foo(int x)
		{
		return new Foo(x);
		}
	}

public class Tester
	{
	public static void Main()
		{
		Foo value = new Foo(8);
		int test =(int) value;
		Foo newValue =(Foo) 8;
		}
	}
```

Operator Example

```
using System;
public struct Digit
{
   byte value;
   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }

   public Digit(int value): this((byte) value) {}

   public static implicit operator byte(Digit d) {
      return d.value;
   }

   public static explicit operator Digit(byte b) {
      return new Digit(b);
   }

   public static Digit operator+(Digit a, Digit b) {
      return new Digit(a.value + b.value);
   }

   public static Digit operator-(Digit a, Digit b) {
      return new Digit(a.value - b.value);
   }

   public static bool operator==(Digit a, Digit b) {
      return a.value == b.value;
   }
```

Example Continued

```
   public static bool operator!=(Digit a, Digit b) {
      return a.value != b.value;
   }

   public override bool Equals(object value) {
      if (value == null) return false;
      if (GetType() == value.GetType()) return this == (Digit)value;
      return false;   }
   public override int GetHashCode() {
      return value.GetHashCode();
   }
   public override string ToString() {
      return value.ToString();
   }
}

class Test
{
   static void Main() {
      Digit a = (Digit) 5;
      Digit b = (Digit) 3;
      byte c = a;
      Digit plus = a + b;
      Digit minus = a - b;
      bool equals = (a == b);
      Console.WriteLine("{0} + {1} = {2}", a, b, plus);
      Console.WriteLine("{0} - {1} = {2}", a, b, minus);
      Console.WriteLine("{0} == {1} = {2}", a, b, equals);
   }
}
```

Struct

Like a class except:

- Struct is a value type (one the stack)
- No inheritance

Structs – use for small data structures with value semantics

- Complex numbers
- points in a coordinate system
- key-value pairs in a dictionary

Struct Example

```
using System;

struct Point
{
	public int x, y;
	public Point(int x, int y) {
		this.x = x;
		this.y = y;
	}
}

public class Tester
	{
	public static void Main()
		{
		Point a = new Point(10, 10);
		Point b = a;
		a.x = 100;
		System.Console.WriteLine(b.x);
		}
	}
```

Output

```
10
```

Assignment

Assignment to a variable of a struct type creates a copy of the value being assigned.

```
Point b = a;
```

When a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created.

A struct may be passed by reference to a function member using a ref or out parameter.

Inheritance

All struct types implicitly inherit from System.ValueType,

System.ValueType inherits from class object.

A struct may implement interfaces,

A struct cannot specify a base class.

Struct types are
- never abstract
- always implicitly sealed.

abstract & sealed modifiers are not permitted in a struct

A struct member cannot be protected or protected internal

struct function members in a cannot be abstract or virtual

Override modifier is allowed only to override methods inherited from the type System.ValueType.

Default Values

The default value of a struct is the value produced by setting

- all value type fields to their default value
- all reference type fields to null

Point a;
Point[] b = new Point[100];

The default constructor of a struct assigns default values

A struct cannot declare a parameterless instance constructor

Structs should be designed to consider the default initialization state a valid state.

```
using System;
struct KeyValuePair
{
   string key;
   string value;
   public KeyValuePair(string key, string value) {
      if (key == null || value == null) throw new ArgumentException();
      this.key = key;
      this.value = value;
   }
}
```

Boxing and Unboxing

```
object a = new Point(10, 10);
Point b = (Point)a;
```

Boxing and unboxing copies the values of a struct

Field Initializers

A struct cannot have variable initializers.

```
struct Point
{
   public int x = 1;  // Error, initializer not permitted
   public int y = 1;  // Error, initializer not permitted
}
```

Destructors

A struct is not permitted to declare a destructor.

Static constructors

Static constructors for structs follow most of the same rules as for classes.

Static constructor is called before:

- An instance member of the struct is referenced.
- A static member of the struct is referenced.
- An explicitly declared constructor of the struct is called.

The creation of default values of struct types does not trigger the static constructor

```
Point a;
Point[] b = new Point[10];
```