Uncommon C#

- Authoring
- Consulting
- Crafting
- Designing
- Mentoring
- Training

{ JSL }

jon@jaggersoft.com
http://www.jaggersoft.com

Jon Jagger

phone +44 (0)1823 345 192
mobile +44 (0)7973 444 782

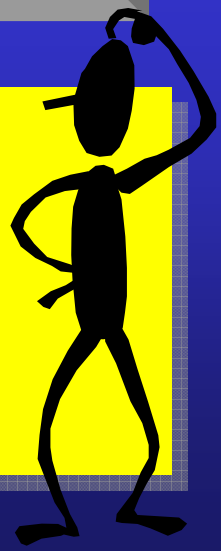
Definite Assignment

12.3 Definite Assignment

A variable shall be definitely assigned at each location where its value is obtained. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when:

- The variable is the left operand of a simple assignment.
- The variable is passed as an output parameter.
- The variable is a struct type variable and occurs as the left operand of a member access.

```
static void Main()
{
    int value;
    unsafe { int * ptr = &value; }
    Console.WriteLine(value);
}
```



Conformance

2 Conformance

The text in this International Standard that specifies requirements is considered normative. Normative text is further broken down into required and conditional categories. Conditionally normative text specifies a feature and its requirements where the feature is optional.

7 General Description

… all of clause 27 with the exception of the beginning is conditionally normative; …

27 Unsafe code

An implementation that does not support unsafe code is required to diagnose any usage of the unsafe keyword.

The remainder of this clause, including all of its subclauses, is conditionally normative.

Spot the bug

27.2 Pointer Types

In an unsafe context several constructs are available for operating on pointers:

- The unary * operator can be used to perform pointer indirection (27.5.1)
- The -> operator can be used to access a member of a struct through a pointer (27.5.2)
- The [] operator can be used to index a pointer (27.5.3)
- …
- The ==, !=, <, >, <=, and => operators can be used to compare pointers (27.5.7)
- The stackalloc operator can be used to allocate memory from the call stack (27.7)
- The fixed statement can be used to temporarily fix a variable so its address can be obtained (27.6)

Protected access

10.5.3 Protected access for instance members

```
public class A
{
    protected int x;

    static void F(A a, B b)
    {
        a.x = 1; // Ok
        b.x = 1; // Ok
    }
}

public class B : A
{
    static void F(A a, B b)
    {
        a.x = 1; // Error
        b.x = 1; // Ok
    }
}
```

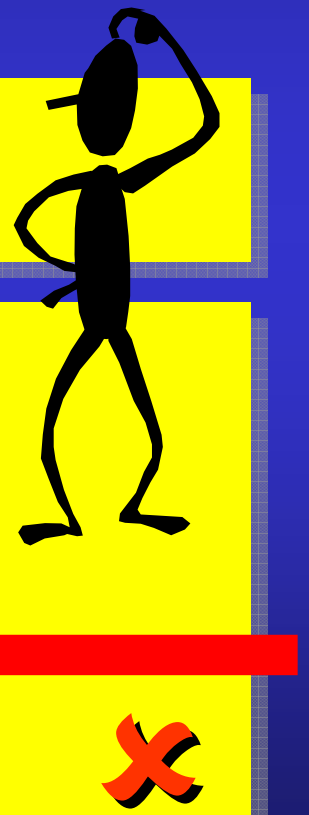
namespace pitfall

Namespace lookup is inside → outside

- Widget not in Company.Other.Framework.Tests
- Widget not in Company.Other.Framework
- Widget not in Company.Other
- Widget IS in Company…

```
namespace Company.Widget.Framework
{
    public class Widget { ... }
}
```

```
using Company.Widget.Framework;
...
namespace Company.Other.Framework.Tests
{   [TestFixture]
    public class FubarTests
    {   [Test]
        public void SomeTest()
        {
            Widget w = new Widget();
            ...
        }
    }
}
```



Solution

Guidelines

- avoid namespace-class name clashes
- prefer a namespace prefix style

```
namespace Company.WidgetFramework
{
    public class Widget { ... }
}
```

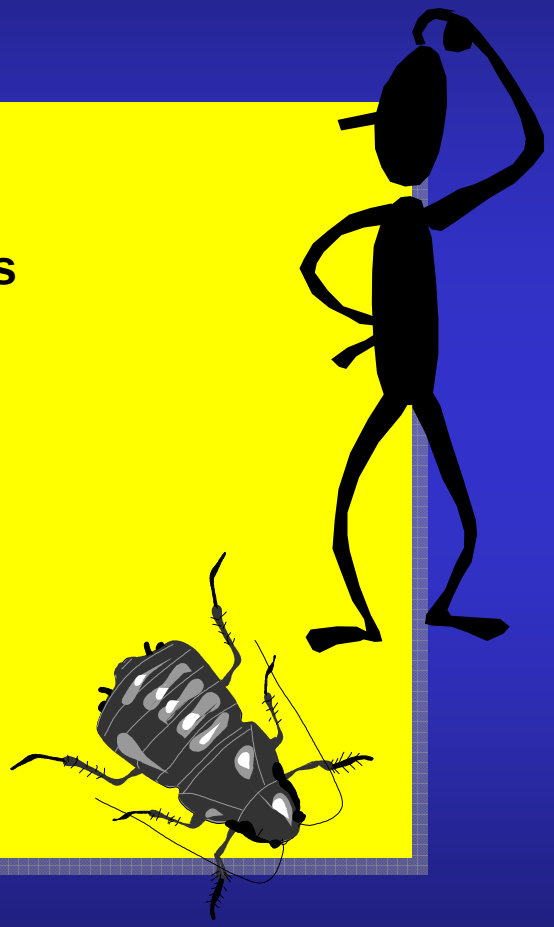
```
using Company.WidgetFramework;
...
namespace Tests.Company.OtherFramework
{
    [TestFixture]
    public class FubarTests
    {
        [Test]
        public void SomeTest()
        {
            Widget w = new Widget();
            ...
        }
    }
}
```

Default access

Default top-level class access is internal

- visible only inside the assembly…

```
using NUnit.Framework;

namespace Company.WidgetLibTests
{
    [TestFixture]
    class WidgetTests
    {
        [Test]
        public void SomeTest()
        {
            //...
        }
    }
}
```



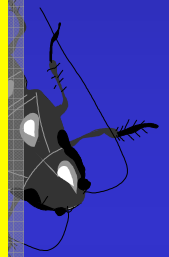
Question: Does the test pass or fail?

Default pitfall

Make sure test classes are public

- Otherwise NUnit won't see them!

```
using NUnit.Framework;

namespace Company.WidgetLibTests
{
    [TestFixture]
    public class WidgetTests
    {
        [Test]
        public void SomeTest()
        {
            //...
        }
    }
}
```



Question: What else should you do?

Testing the tests

Check if [TestFixture]'d classes are internal…

```
[TestFixture]
public class InternalFixtureTests
{
    [Test]
    public void AccidentalNonPublicTestFixture()
    {
        Type tfa = typeof(TestFixtureAttribute);
        Assembly self = this.GetType().Module.Assembly;
        foreach (Type type in self.GetTypes())
        {
            object[] attributes = type.GetCustomAttributes(tfa, false);
            if (attributes != null && attributes.Length > 0
                && type.IsNotPublic)
            {
                Assert.Fail(type.ToString() + " is not public!");
            }
        }
    }
}
```

Generic ambiguity

What's this?

```
F(G<A, B>(7));
```

A call to F with two arguments:

- G < A
- B > (7) viz redundant parentheses

A call to F with one argument:

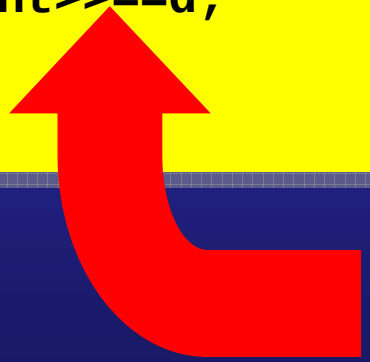
- G < A,B > (7)
  - Two type parameters: A,B
  - One regular argument: 7

>>= tokenization

9.4.5 Operators and punctuators

```
right-shift:
    >  >

right-shift-assignment:
    >  >=
```

```
delegate void D();

class C
{
    static void F<T>() { ... }
    static D d = F<F<int>>;
    static void Main()
    {
        bool b = F<F<int>>==d;
    }
}
```



Compile-time error

String Interning

9.4.4.5 String literals

When two or more string literals that are equivalent according to the string equality operator, appear in the same assembly, these string literals refer to the same string instance.

Unfortunate example of over-specification

- Implementation != specification
- Serves no useful purpose for C# users
- String reference equality does not mean strings are necessarily in the same assembly

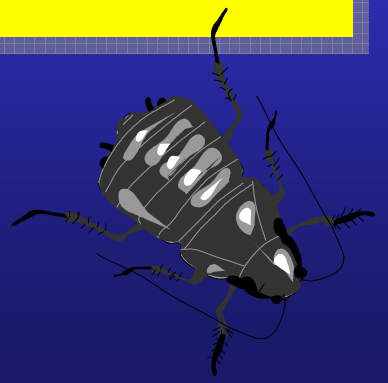
Hex Escape Sequence

9.4.4.4 Character Literals

- A hex-escape-sequence character literal has 1, 2, 3, or 4 hex-digits

```
hexadecimal-escape-sequence:
   \x hex-digit hex-digit? hex-digit? hex-digit?
```

```
s1 = "\x7Scotland"; // BEL Scotland
s2 = "\x7England";  // ~ ngland
```



Advice: use unicode-escapes

are inadvisable in some places…

```
using Integer = int; // compile-time error

using Integer = System.Int32; // ok
```

```
Type ti = Type.GetType("int"); // runtime failure

Type ti = Type.GetType("System.Int32"); // ok

Type ti = typeof(int); // better
```

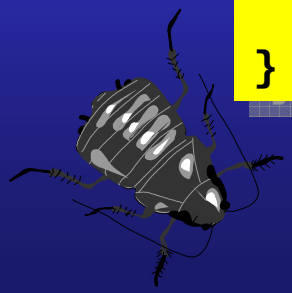
```
extern unsafe static void
    Process(int length, S * array); // ok

extern unsafe static void
    Process(Int32 length, S * array); // better
```

Properties are not variables

```
public struct Point
{   ...
    private int x, y;
    public int X
    {
        get { ... }
        set { ... }
    }
}
```

```
public struct Rectangle
{   ...
    private Point topLeft;
    public Point TopLeft
    {
        get { ... }
        set { ... }
    }
}
```

```
class App
{
    static void Main()
    {
        Rectangle r = new Rectangle();
        r.TopLeft.X = 42; // compile-time error
    }
}
```



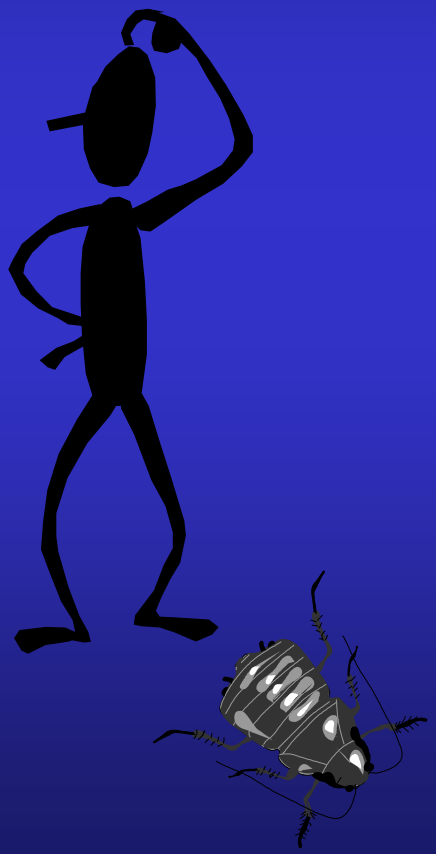
readonly + struct

Calling a method on a readonly struct?

- The method is called on a copy of the struct!

```
struct Mile
{
    ...
    public void Add(Mile rhs)
    {
        value += rhs.value;
    }
    private int value;
}
```

```
class Trap
{
    ...
    public void Ouch(Mile more)
    {
        distance.Add(more);
    }

    private readonly Mile distance;
}
```



readonly struct

Solution

- Make struct immutable – rely on assignment

```
struct Mile
{
    public Mile(int value)
    {
        this.value = value;
    }

    public static Mile
        operator+ Add(Mile lhs, Mile rhs)
    {
        return new Mile(lhs.value + rhs.value);
    }

    private readonly int value;
}
```

```
class Trap
{
    ...
    public void Ooops(Mile more)
    {
        distance += more; // compile-time error
    }

    private readonly Mile distance;
}
```

Checked pitfall

```
public class Eg
{
    public void Ok(int x)
    {
        x *= 2;
    }
}
```

```
public class Eg
{
    public void AlsoOk(int x)
    {
        checked { x *= 2; }
    }
}
```

```
public class Eg
{
    public void NotOk(int x)
    {
        checked(x *= 2);
    }
}
```

```
public class Eg
{
    public void OkAgain(int x)
    {
        x = checked(x * 2);
    }
}
```

Checking noses

15.11 The checked and unchecked statements

The checked statement causes all expressions in the block to be evaluated in a checked context, and the unchecked statement causes all expressions in the block to be evaluated in an unchecked context.

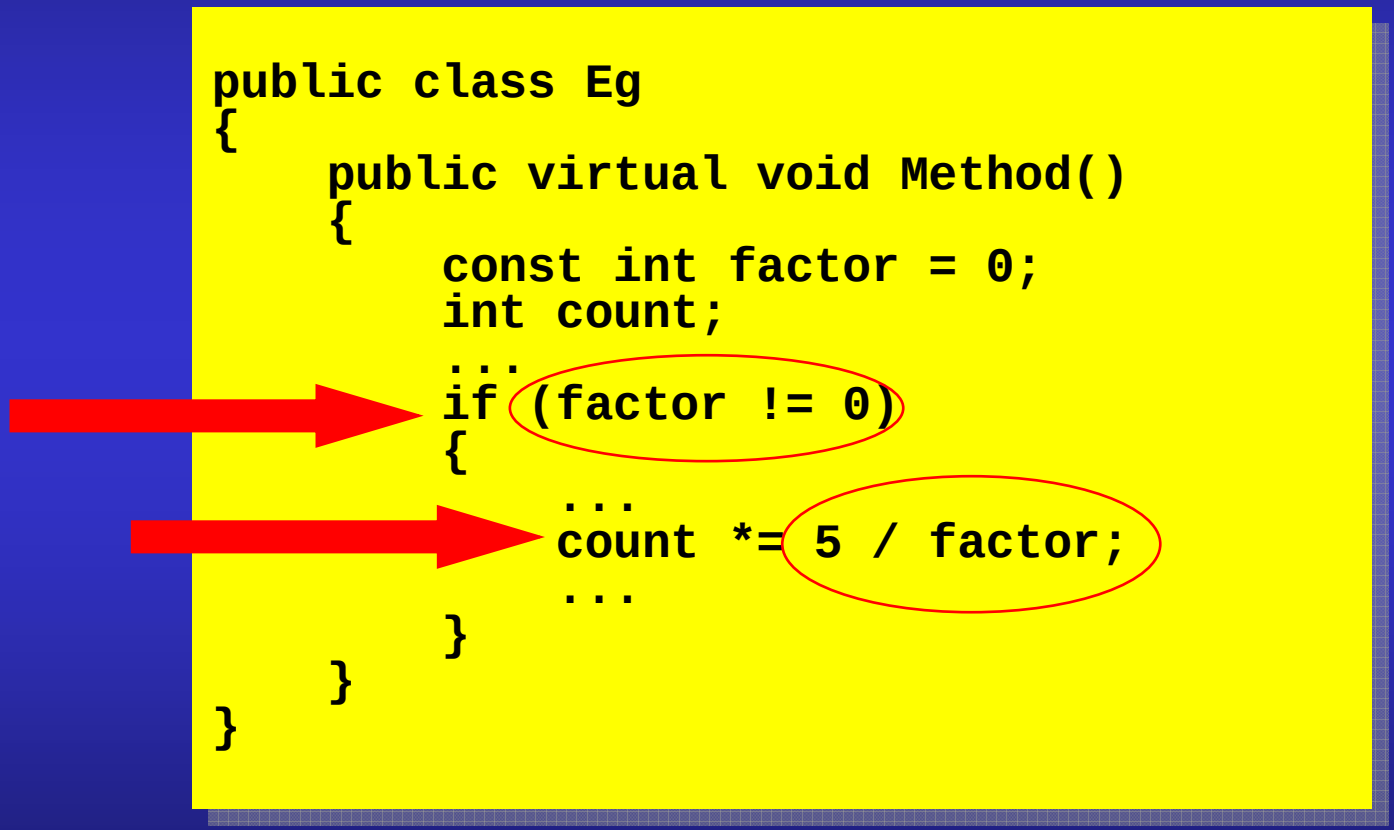
The checked and unchecked statements are precisely equivalent to the checked and unchecked operators (14.5.13) except that they operate on blocks instead of expressions.

```
checked
{
    int value;
    ...
    value = unchecked(checked(F() * G()) + 42);
}
```

Constant expressions

Must be evaluated at compile time

- And, by default, are checked



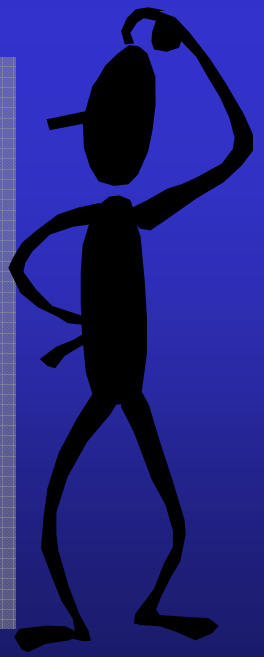
```
public class Eg
{
    public virtual void Method()
    {
        const int factor = 0;
        int count;
        ...
        if (factor != 0)
        {
            ...
            count *= 5 / factor;
            ...
        }
    }
}
```

Invariant meaning

14.5.2.1 Invariant meaning in blocks

For each occurrence of a given identifier as a *simple-name* in an *expression* or *declarator*, every other occurrence of the same identifier as a *simple-name* in an *expression* or *declarator* within the immediately enclosing *block* or *switch-block* shall refer to the same entity.

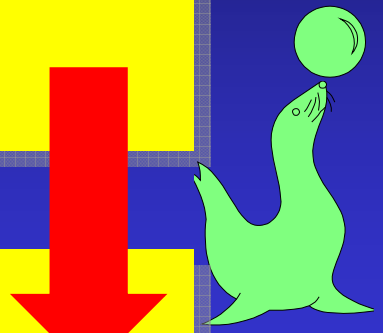
```
class Example
{
    void F()
    {
        if (true) {
            int v = 42;
        }
        int v = 1;
    }
}
```



Default == unsealed

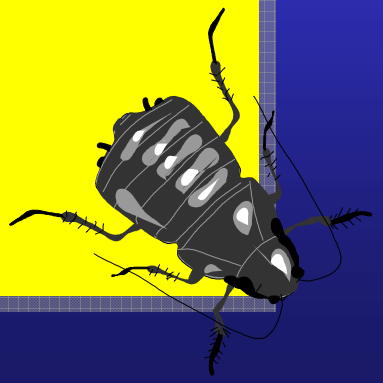
In retrospect a poor choice

```
public /*unsealed*/ class Vulnerable
{
    ~Vulnerable() { ... }
}
```



```
public /*unsealed*/ class Vulnerable
{
    protected override void Finalize()
    { ... }
}
```

```
public class Attacker : Vulnerable
{
    ~Attacker()
    {
        for(;;);
    }
}
```



Solutions

Guidelines

- Don't rely on defaults
- Make type and member access explicit
- Classes: static, sealed, abstract, or /*unsealed*/

```
public /*unsealed*/ class DoesntCompile
{
    protected sealed override ~Vulnerable()
    { ...
    }
}
```

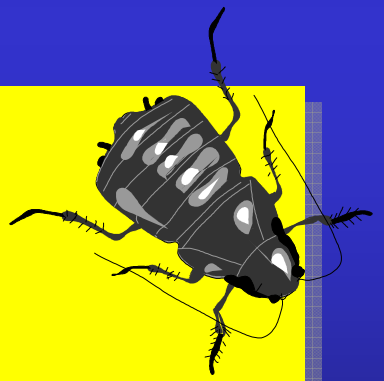
```
public sealed class Best
{
    ~Best()
    { ...
    }
}
```

virtual: C# != C++

virtual → first implementation

```
public abstract class Middle ...
{
    public virtual void Foo()
    { ...
    }
}
```

```
public sealed class Bottom : Middle
{
    public virtual void Foo()
    { ...
    }
}
```



>Warning: Bottom.Foo() hides inherited member Middle.Foo()

override → another implementation

```
public abstract class Middle ...
{
    public virtual void Foo()
    { ...
    }
}
```

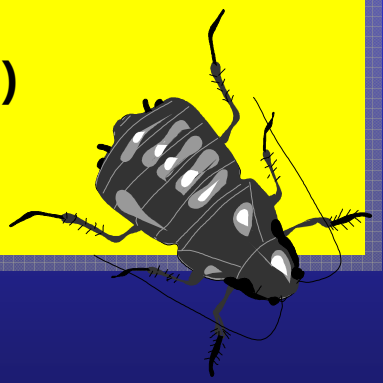
```
public sealed   class Bottom : Middle
{
    public override void Foo()
    { ...
    }
}
```

Solution

ref/out overloading

- too easy to forget the ref/out

```
public sealed class Dodgy
{
    public void Foo(Wibble value)
    { ...
    }
    public void Foo(ref Wibble value)
    { ...
    }

    public void Bar(Wibble value)
    { ...
    }
    public void Bar(out Wibble value)
    { ...
    }
}
```



Better conversion pitfall

```
public sealed class A
{
    public static implicit operator B(A from)
    { ...
    }
}

public sealed class B
{
}

public sealed class App
{
    static void Method(A a, B b) { ... }
    static void Method(B b, A a) { ... }

    static void Main()
    {
        A a = new A();
        Method(a, null); // ambiguous
    }
}
```

Overload pitfall - 3

14.5.5.1 Method invocations

The set of candidate methods for the method invocation is constructed. For each method F associated with the method group M:

…

The set of candidate methods is reduced to contain only methods from the most derived types:

```
public class Base
{
    public virtual void Method(int value) { }
}
public class Derived : Base
{
    public void Method(double value) { }
    public override void Method(int value) { }
}
public class Demo
{
    public static void Main()
    {
        Derived d = new Derived();
        d.Method(42);
    }
}
```

Overload

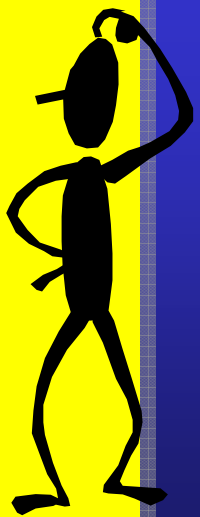
- Guidelines
  - Don't mix overloading and overriding
  - Don't overload solely on ref/out
  - Reference conversions → inheritance
  - Overloading does not happen in CIL

Conversion Oddity

A class is implicitly convertible to an interface

- Only if it actually realizes the interface

```
interface IWibble
{   ...
}

class Alpha : IWibble
{   ...
}

class Beta
{   ...
    public static implicit operator Alpha(Beta from)
    {
        return new Beta();
    }
}

class App
{
    static void Main()
    {
        Beta b = new Beta();
        IWibble iw = (IWibble)b; // cast-required
    }
}
```



Hidden boxing?

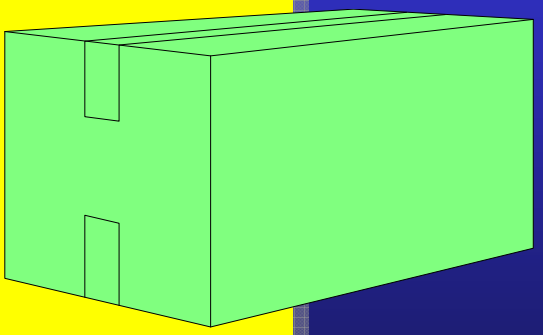
14.9.6 Reference type equality operators

Every class type C implicitly provides the following predefined reference type equality operators:

```
bool operator ==(C x, C y);
bool operator !=(C x, C y);
```

...there are special rules for determining when a reference type equality operator is applicable.

```
struct S { }

S s1,s2;
if (s1 == s2)
    ...
```

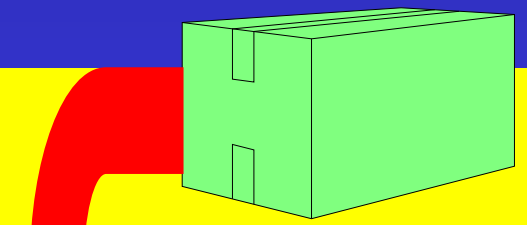


Hidden boxing

14.7.4 Addition operator

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

```
message = "Answer==" + 42;
```



```
message =
   operator+("Answer==", (object)42);
```

```
message = "Answer==" + 42.ToString();
```

Boxing Trick

How to tell if a struct instance is boxed?

```
interface IBoxable
{
    bool ? IsBoxed();
}

unsafe struct Eg : IBoxable
{
    public Eg(int value)
    {
        fixed (Eg * ptr = &this) {
            this.address = ptr;
        }
    }
    public bool ? IsBoxed()
    {
        if (address == null)
            return null;
        else
            fixed(Eg * ptr = &this) {
                return ptr != address;
            }
    }
    private readonly Eg * address;
}
```

The lock statement

A lock statement of the form

```
lock (x) ...
```

is precisely equivalent to:

```
object obj = x;
System.Threading.Enter(obj);
// comment: weak spot here...
try {
    ...
}
finally {
    System.Threading.Exit(obj);
}
```

Q: What happens if a Thread.Abort occurs at the comment?

A: The call to System.Threading.Exit is bypassed!

Event Thread Unsafely

The C# 1.0 Standard contained this example

- Ooops

```
public delegate
    void EventHandler(object sender, EventArgs e);

public class Button : Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e)
    {
        if (Click != null)
        {
            Click(this, e);
        }
    }
}
```

Event Thread safely

The C# 2.0 Standard the example is now…

- Much better…

```
public delegate
    void EventHandler(object sender, EventArgs e);

public class Button : Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e)
    {
        EventHandler toRaise = Click;

        if (toRaise != null)
        {
            toRaise(this, e);
        }
    }
}
```

Event Locking

The C# 1.0 Standard said this…

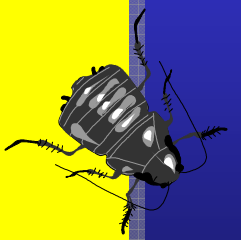
17.7.1 Field like events

In order to be thread safe, the addition and removal operations are done while holding the lock on the containing object for an instance event, or the type object for a static event.

```
public delegate void D();
```

```
class X
{
    public event D Ev;
}
```

```
class X
{
    private D __Ev;
    public event D Ev
    {
        add { lock(this) { __Ev += value; } }
      remove { lock(this) { __Ev -= value; } }
    }
}
```



Event Locking

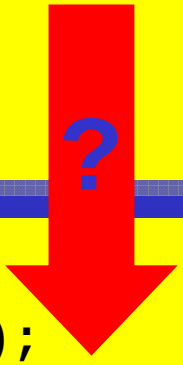
The C# 2.0 Standard says this…

17.7.1 Field like events

The addition and removal operations on all instance events of a class shall be done while holding the lock on an object uniquely associated with the containing object.

```
public delegate void D();
```

```
class X
{
    public event D Ev;
}
```




```
class X
{
    private readonly object __key = new object();
    private D __Ev;
    public event D Ev
    {
        add { lock(__key) { __Ev += value; }}
      remove { lock(__key) { __Ev -= value; }}
    }
}
```

Event Locking?

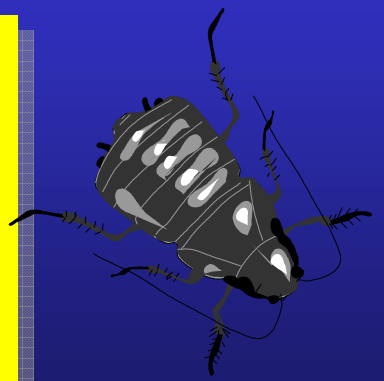
Events inside structs?

17.7.1 Field like events
The addition and removal operations on all instance events of a class shall be done while holding the lock on an ***object*** uniquely associated with the containing object.

```
public delegate void D();
```

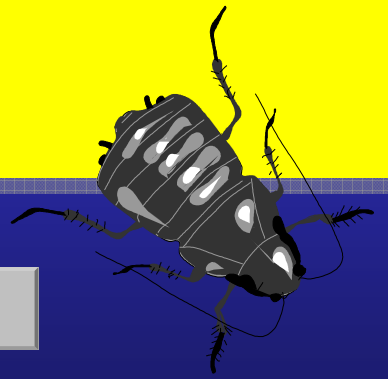
```
struct S
{
    public event D Ev;
}
```

```
struct S
{
    private D __Ev;
    public event D Ev
    {
        add { __Ev += value; }
      remove { __Ev -= value; }
    }
}
```



Captured Variables

```
delegate void F();

class CapturedPitfall
{
    static void Main()
    {
        F[] array = new F[5];
        for (int at = 0; at != 5; at++)
        {
            array[at] = delegate {
                Console.Write(at);
            };
        }
        for (int cat = 0; cat != 5; cat++)
        {
            array[cat]();
        }
    }
}
```



```
55555
```

Captured Variables

```
delegate void F();

class CapturedPerLoop
{
    static void Main()
    {
        F[] array = new F[5];
        for (int at = 0; at != 5; at++)
        {
            int value = at;
            array[at] = delegate {
                Console.Write(value);
            };
        }
        for (int at = 0; at != 5; at++)
        {
            array[at]();
        }
    }
}
```

```
01234
```

Captured Variables

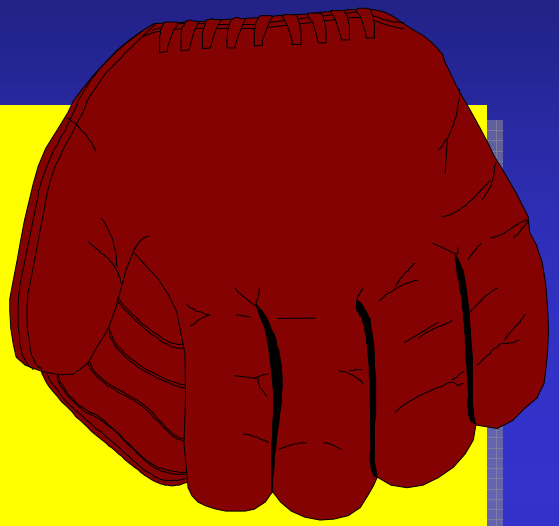
Local variable shared between threads!

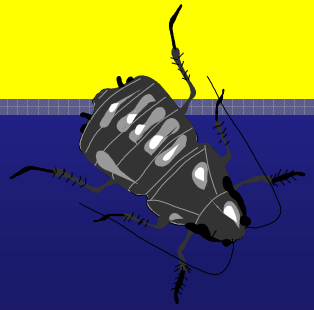
```
using System.Threading;

class Demo
{
    static void Main()
    {
        int i = 0;

        ThreadStart t1 = delegate {
            for (;;) {
                Thread.Sleep(102);
                i++;
            }
        };
        ThreadStart t2 = delegate {
            for(;;) {
               Thread.Sleep(500);
               System.Console.Write("{0} ", i);
            }
        };
        new Thread(t1).Start();
        new Thread(t2).Start();
    }
}
```

```
4 9 13 18 23 27 32
```

Not all exceptions are managed


```
class Vulnerable
{
    static void Main()
    {
        try
        {
            // ...
        }
        catch (Exception error)
        {
            // clean-up code. could result in
            // security hole if not run.
            // Ooops
        }
    }
}
```



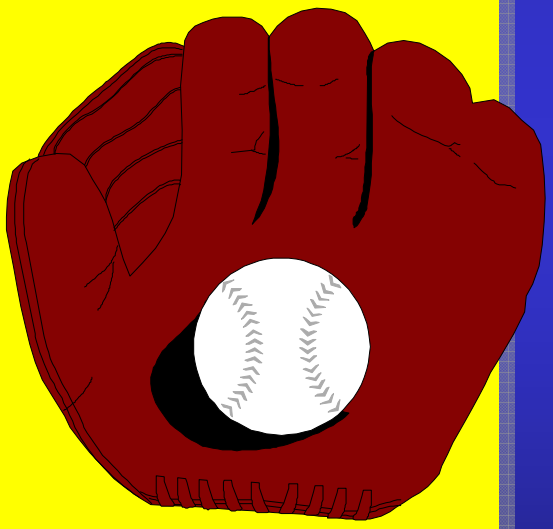


Exceptions

C# 1.0

- General catch clause

```
class Vulnerable
{
    static void Main()
    {
        try
        {
            // ...
        }
        catch (Exception error)
        {
            CleanUp();
        }
        catch
        {
            CleanUp();
        }
    }
}
```

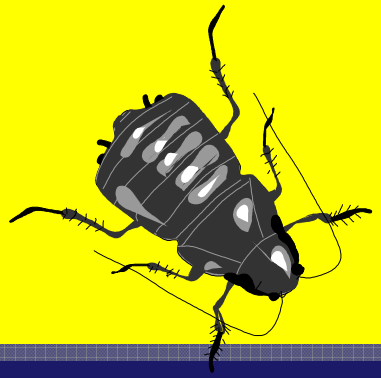


Exceptions

CLR 2.0

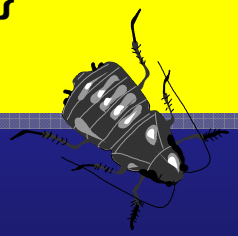
- Unmanaged exception → RuntimeWrappedException

```
class Vulnerable
{
    static void Main()
    {
        try
        {
            // ...
        }
        catch (Exception error)
        {
            CleanUp();
        }
        catch
        {
            // new compiler warning
            // now unreachable...
        }
    }
}
```



Spot the bug…

```
using System.Threading;

class App
{
    static void Main()
    {
        bool firstInstance;
        Mutex key =
            new Mutex(@"Global\App", out firstInstance);
        if (firstInstance)
        {
            // we're the only instance running
            // ...
        }
        else
        {
            // another instance detected
            // ...
        }
    }
}
```



One solution…

```
using System.Threading;

class App
{
    static void Main()
    {
        bool firstInstance;

        using (new Mutex(@"Global\App", out firstInstance))
        {
            if (firstInstance)
            {
                // we're the only instance running
                // ...
            }
            else
            {
                // another instance detected
                // ...
            }
        }
    }
}
```

That's all Folks!

Any Questions?

- Authoring
- Consulting
- Crafting
- Designing
- Mentoring
- Training

{ JSL }

jon@jaggersoft.com
http://www.jaggersoft.com

Jon Jagger

phone +44 (0)1823 345 192
mobile +44 (0)7973 444 782

What's this?

```
1.D
```

1 an int literal
. the member access operator
D the field/property called D

or

1. a double literal*
D a double type suffix confirming 1. as a double

1.D?

Plug…

…portions of this presentation are from the forthcoming book…

Annotated C# Standard

by Jon Jagger, Nigel Perry, Peter Sestoft
published by Morgan Kaufmann
copyright 200? Elsevier Inc

Access Modifiers

Absence, defaults, forbidden, compulsory

```
public interface IX
{
    void M();
}

public class X : IX
{
    public static operator ...
    { ...
    }

    static X()
    {   ...
    }

    ~X()
    {   ...
    }

    void IX.M()
    {   ...
    }
}
```

public not allowed

public required

Static constructor not callable

Finalizer not callable

Explicit Impl.

not callable*

Uninitialized variables

12.3.3.21 Invocation expressions…

For an invocation expression *expr* of the form:

primary-expression(arg_1, arg_2, …, arg_N)

…

For each argument arg_i, the definite assignment state of v after arg_i is determined by the normal expression rules, ignoring any ref or out modifiers.

```
public class Eg
{
    static void Method(out int x, int y)
    {
        Console.WriteLine(y);
        x = 42;
    }

    static void Main()
    {
        int x;
        Method(out x, x);
    }
}
```