Uncommon C#

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

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

Jon Jagger

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

C#
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.

```csharp
static void Main()
{
    int value;
    unsafe { int * ptr = &value; }  // Corrected the unsafe block
    Console.WriteLine(value);
}
```
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.
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)
10.5.3 Protected access for instance members

```java
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 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...

```csharp
class Widget
```

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

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

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

```csharp
using Company.WidgetFramework;
...
namespace Tests.Company.OtherFramework
{
    [TestFixture]
    public class FubarTests
    {
        [Test]
        public void SomeTest()
        {
            Widget w = new Widget();
            ...
        }
    }
}
```
Default top-level class access is internal

- visible only inside the assembly...

```csharp
using NUnit.Framework;
namespace Company.WidgetLibTests {
    [TestFixture]
    class WidgetTests {
        [Test]
        public void SomeTest() {
            //...
        }
    }
}
```

Question: Does the test pass or fail?
Make sure test classes are public
- Otherwise NUnit won't see them!

```csharp
using NUnit.Framework;
namespace Company.WidgetLibTests {
    [TestFixture]
    public class WidgetTests {
        [Test]
        public void SomeTest() {
            //...
        }
    }
}
```

Question: What else should you do?
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!");
            }        
    }  
}  
}
What's this?

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
9.4.5 Operators and punctuators

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

class C

delegate void D();
class C

static void Main()
{
    delegate void D(); class C

    static bool b = F<F<int>>==d;
    static void F<T>()
    {
        ...
    }

    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
    {
        ...
    }

    delegate void D();
    class C
    
    static void F<T>()
    {
        ...
    }

    bool b = F<F<int>>==d;
    delegate void D();
    static void Main()
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
9.4.4.4 Character Literals

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

```
\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

```csharp
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
    }
}
```
Calling a method on a readonly struct?

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

```csharp
struct Mile
{
    private int value;

    public void Add(Mile rhs)
    {
        value += rhs.value;
    }
}

class Trap
{
    private readonly Mile distance;

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

- Make struct immutable – rely on assignment

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

    public static Mile operator+(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;
}
```
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);
    }
}
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.

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

- Must be evaluated at compile time
- And, by default, are checked

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

```
In retrospect a poor choice

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

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

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

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

Solutions

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

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

```csharp
public /*unsealed*/ class DoesntCompile
{
    protected sealed override ~Vulnerable() { ... }
}
```
virtual $\rightarrow$ **first** implementation

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

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

> Warning: Bottom.Foo() **hides** inherited member Middle.Foo()
override → another implementation

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

public sealed class Bottom : Middle
{
    public override void Foo()
    {
        ...
    }
}
```
ref/out overloading

- too easy to forget the ref/out

```csharp
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

```csharp
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
    }
}
```
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:*

```csharp
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);
    }
}
```
• Guidelines
  - Don't mix overloading and overriding
  - Don't overload solely on ref/out
  - Reference conversions → inheritance
  - Overloading does not happen in CIL
A class is implicitly convertible to an interface
- Only if it actually realizes the interface

```csharp
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
    }
```
14.9.6 Reference type equality operators
Every class type C implicitly provides the following predefined reference type equality operators:

```c
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.

```c
struct S { }
S s1, s2;
if (s1 == s2) ...
```
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();
How to tell if a struct instance is boxed?

```csharp
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;
}
```
A lock statement of the form

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

is precisely equivalent to:

```csharp
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!
The C# 1.0 Standard contained this example

```
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);
        }
    }
}
```
The C# 2.0 Standard the example is now...

- Much better...

```csharp
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);
        }
    }
}
```
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.

```csharp
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; } }
    }
}
```
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.

```csharp
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; }}
    }
}
```
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.

```csharp
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; }
    }
}
```
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]();
        }
    }
}
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]();
        }
    }
}
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();
    }
}
class Vulnerable {
    static void Main() {
        try {
            // ...
        } catch (Exception error) {
            // clean-up code. could result in
            // security hole if not run.
            // Ooops
        }
    }
}
C# 1.0

- General catch clause

```csharp
class Vulnerable {
    static void Main() {
        try {
            // ...
        } catch (Exception error) {
            CleanUp();
        } catch {
            CleanUp();
        }
    }
}
```
Unmanaged exception → RuntimeWrappedException

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

```csharp
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

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

[C#]

Jon Jagger

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

{ JSL }
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
...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

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

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

    static X()
    {
        ...
    }   // Static constructor not callable

    ~X()
    {
        ...
    }   // Finalizer not callable

    void IX.M()
    {
        ...
    }   // Explicit Impl. not callable*
}
```

- **Public not allowed**
- **Public required**
- Static constructor not callable
- Finalizer not callable
- Explicit Impl. not callable*
12.3.3.21 Invocation expressions...
For an invocation expression expr of the form:
    primary-expression(arg₁, arg₂, ..., argₙ)
...
For each argument argᵢ, the definite assignment state of v after argᵢ is determined by the normal expression rules, ignoring any ref or out modifiers.

```csharp
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);
    }
}
```