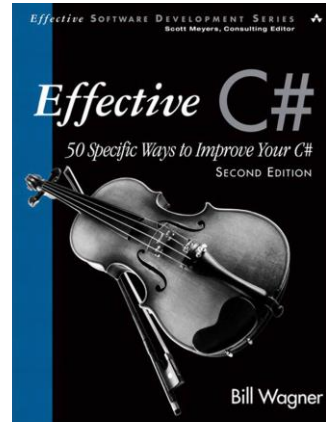
Effective C#



Presentation By: Brian Braatz

Derived from the book Effective C# 2nd Edition

Item 1:

Use Properties Instead of Accessible Data Me
mbers

Item1: Use Properties Instead of Accessible Data Members

```
public class Customer
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                  "Name cannot be blank",
                 "Name");
            name = value;
        }
        // More Elided.
    }
}
```

- Advantages
  - Write or Read only support
  - Multi-threading support
  - Implementation can change without breaking interface
  - Error handling for bad values
  - Properties can be virtual
  - Interface support
  - PropertyChange style notification possible
  - Get or Set can be private or protected
- Disadvantages
  - More typing?
  - Speed?
    - http://tinyurl.com/nuy95fn
    - Tested ints properties vs variables
    - 10 million R\W
    - 59 ms vs 97 ms in release mode

Item 2: Prefer readonly to const

```
public class UsefulValues
{
public static readonly int StartValue = 105;
public const int EndValue = 120;
}
```

- Const is “Inlined” like a C\C++ Macro
  - If Assembly X compiles against UsefulValues.EndValue, Assembly will get “120” ininlined in its IL
  - If UsefulValues.EndValue changes without recompile- Assembly X will have wrong value
- Readonly will act like a normal variable, but the compiler will enforce the readonly ability
- Readonly values are assigned at runtime
- Const only works with numbers or strings
- Const is (slightly) faster
- Readonly is safer and more flexible

Item 3: Prefer the is or as Operators to Casts

"As"

```
object o = Factory.GetObject();
// Version one:
MyType t = o as MyType;
if (t != null)
{
// work with t, it's a MyType.
}
else
{
// report the failure.
}
```

Cast

```
object o = Factory.GetObject();
// Version two:
try
{
        MyType t;
        t = (MyType)o;
        // work with T, it's a MyType.
}
catch (InvalidCastException)
{
// report the conversion failure.
}
```

- As \ Is
  - Do not perform any user-defined conversions
  - Succeed only if the runtime type matches the sought type;
    - they never construct a new object to satisfy a request
- Casts
  - May convert
    - Casting a long to a short can lose information
  - Will miss runtime conversions
    - compile time only!
    - I.e. if Rock is defined as convertible to Animal via "implicit operator" at runtime, the cast will fail as it is compile time only

Item 4: Use Conditional Attributes Instead of #if

```
public string LastName
{

  get
  {
   CheckState();
   return lastName;
  }
  set
  {
    CheckState();
    lastName = value;
    CheckState();
  }
}
```

- Good

```
[Conditional("DEBUG"),
Conditional("TRACE")]
private void CheckState()
{
…
}
```

- Bad

```
#if ( VAR1 && VAR2 )
#define BOTH
#endif

private void CheckStateBad()
{
  // The Old way:
   #if BOTH
   Trace.WriteLine("Entering CheckState");
   #endif
}
```

Item 5: Always Provide ToString()

“As”

```
public class Customer
{
    public string Name
    {
        get;
        set;
    }
    public decimal Revenue
    {
        get;
        set;
    }
    public string ContactPhone
    {
        get;
        set;
    }

    public override string ToString()
    {
        return Name;
    }
}
```

- Compiler supplied constructor will provide a ToString which returns “Customer”
- At the very least you could provide the customer’s name

```
public override string ToString()
{
    return Name;
}
```

Item 6: Understand the Relationships Among the Many Different Concepts of Equality

- C# provides four different functions that determine whether two different objects are “equal”
- Understand what each does before overriding

```
public static bool ReferenceEquals (object left, object right);
public static bool Equals (object left, object right);
public virtual bool Equals(object right);
public static bool operator ==(MyClass left, MyClass right);
```

Item 6: Understand the Relationships Among the Many Different Concepts of Equality (contd)

- Overriding:
  - Do not override static Object.ReferenceEquals() and static Object.Equals() because they provide the correct tests, regardless of the runtime type.
  - Always override instance Equals() and operator==() for value types to provide better performance.
  - You override instance Equals() for reference types when you want equality to mean something other than object identity.

Item 6: Understand the Relationships Among the Many Different Concepts of Equality (contd)

- Equality Checking:
  - To check for reference equality, use ReferenceEquals.
  - To check for value equality, use Equals or Equals.
  - by Default the operator == tests for reference equality by determining if two references indicate the same object
  - reference types do not need to implement operator == in order to gain this functionality.
  - Equals checks data equality for value types,
    - and reference equality for non-value types (general objects).

Item 6: Understand the Relationships Among the Many Different Concepts of Equality

```
public static new bool Equals(object left, object right)
{
    // Check object identity
    if (Object.ReferenceEquals(left, right) )
        return true;

    // both null references handled above

  if (Object.ReferenceEquals(left, null) ||
        Object.ReferenceEquals(right, null))
        return false;

    return left.Equals(right);
}
```

- The default implementation of Equals
  - First checks by reference, then by calling the virtual Equals
- Guidance:
  - Override Equals() whenever you create a value type
  - Override Equals() on reference types when you do not want your referenc type to obey reference semantics
  - Anytime you create a value type, redefine operator==()
    - Default implementation uses reflection to do a value based comparison of all members- not very efficient

Equality Code Sample Cheat Sheet -1

```
    //odd os are null; evens are not null
    object o1 = null;
    object o2 = new object();
    object o3 = null;
    object o4 = new object();
    object o5 = o1;
    object o6 = o2;

    Demo d1 = new Demo(Guid.Empty);
    Demo d2 = new Demo(Guid.NewGuid());
    Demo d3 = new Demo(Guid.Empty);

    // adapted from
   // https://stackoverflow.com/questions/3869601/c-sharp-equals-referenceequals-and-operator
    Debug.WriteLine("comparing null with null always yields true...");
    ShowResult("ReferenceEquals(o1, o1)", () => ReferenceEquals(o1, o1)); //true
    ShowResult("ReferenceEquals(o3, o1)", () => ReferenceEquals(o3, o1)); //true
    ShowResult("ReferenceEquals(o5, o1)", () => ReferenceEquals(o5, o1)); //true
    ShowResult("o1 == o1", () => o1 == o1); //true
    ShowResult("o3 == o1", () => o3 == o1); //true
    ShowResult("o5 == o1", () => o5 == o1); //true

    Debug.WriteLine("...though because the object's null, we can't call methods on the object
(i.e. we'd get a null reference exception).");
    ShowResult("o1.Equals(o1)", () => o1.Equals(o1)); //NullReferenceException
    ShowResult("o1.Equals(o2)", () => o1.Equals(o2)); //NullReferenceException
    ShowResult("o3.Equals(o1)", () => o3.Equals(o1)); //NullReferenceException
    ShowResult("o3.Equals(o2)", () => o3.Equals(o2)); //NullReferenceException
    ShowResult("o5.Equals(o1)", () => o5.Equals(o1));  //NullReferenceException
    ShowResult("o5.Equals(o2)", () => o5.Equals(o1));  //NullReferenceException

    Debug.WriteLine("Comparing a null object with a non null object always yeilds false");
    ShowResult("ReferenceEquals(o1, o2)", () => ReferenceEquals(o1, o2)); //false
    ShowResult("ReferenceEquals(o2, o1)", () => ReferenceEquals(o2, o1)); //false
    ShowResult("ReferenceEquals(o3, o2)", () => ReferenceEquals(o3, o2)); //false
    ShowResult("ReferenceEquals(o4, o1)", () => ReferenceEquals(o4, o1)); //false
    ShowResult("ReferenceEquals(o5, o2)", () => ReferenceEquals(o3, o2)); //false
    ShowResult("ReferenceEquals(o6, o1)", () => ReferenceEquals(o4, o1)); //false
    ShowResult("o1 == o2)", () => o1 == o2); //false
    ShowResult("o2 == o1)", () => o2 == o1); //false
    ShowResult("o3 == o2)", () => o3 == o2); //false
    ShowResult("o4 == o1)", () => o4 == o1); //false
    ShowResult("o5 == o2)", () => o3 == o2); //false
    ShowResult("o6 == o1)", () => o4 == o1); //false
    ShowResult("o2.Equals(o1)", () => o2.Equals(o1)); //false
    ShowResult("o4.Equals(o1)", () => o4.Equals(o1)); //false
    ShowResult("o6.Equals(o1)", () => o4.Equals(o1)); //false

}
```

```
Debug.WriteLine("(though again, we can't call methods on a null object:");
    ShowResult("o1.Equals(o2)", () => o1.Equals(o2)); //NullReferenceException
    ShowResult("o1.Equals(o4)", () => o1.Equals(o4)); //NullReferenceException
    ShowResult("o1.Equals(o6)", () => o1.Equals(o6)); //NullReferenceException

    Debug.WriteLine("Comparing 2 references to the same object always yields true");
    ShowResult("ReferenceEquals(o2, o2)", () => ReferenceEquals(o2, o2)); //true
    ShowResult("ReferenceEquals(o6, o2)", () => ReferenceEquals(o6, o2)); //true <-- Interesting
    ShowResult("o2 == o2", () => o2 == o2); //true
    ShowResult("o6 == o2", () => o6 == o2); //true <-- Interesting
    ShowResult("o2.Equals(o2)", () => o2.Equals(o2)); //true
    ShowResult("o6.Equals(o2)", () => o6.Equals(o2)); //true <-- Interesting

    Debug.WriteLine("However, comparing 2 objects may yield false even if those objects have the
same values, if those objects reside in different address spaces (i.e. they're references to
different objects, even if the values are similar)");
    Debug.WriteLine("NB: This is an important difference between Reference Types and Value
Types.");
    ShowResult("ReferenceEquals(o4, o2)", () => ReferenceEquals(o4, o2)); //false <-- Interesting
    ShowResult("o4 == o2", () => o4 == o2); //false <-- Interesting
    ShowResult("o4.Equals(o2)", () => o4.Equals(o2)); //false <-- Interesting

    Debug.WriteLine("We can override the object's equality operator though, in which case we
define what's considered equal");
    Debug.WriteLine("e.g. these objects have different ids, so we treat as not equal");
    ShowResult("ReferenceEquals(d1,d2)", () => ReferenceEquals(d1, d2)); //false
    ShowResult("ReferenceEquals(d2,d1)", () => ReferenceEquals(d2, d1)); //false
    ShowResult("d1 == d2", () => d1 == d2); //false
    ShowResult("d2 == d1", () => d2 == d1); //false
    ShowResult("d1.Equals(d2)", () => d1.Equals(d2)); //false
    ShowResult("d2.Equals(d1)", () => d2.Equals(d1)); //false
```

Equality Code Sample Cheat Sheet -2

```
    //odd os are null; evens are not null
    object o1 = null;
    object o2 = new object();
    object o3 = null;
    object o4 = new object();
    object o5 = o1;
    object o6 = o2;

    Demo d1 = new Demo(Guid.Empty);
    Demo d2 = new Demo(Guid.NewGuid());
    Demo d3 = new Demo(Guid.Empty);

    ShowResult("d1 == d2", () => d1 == d2); //false
    ShowResult("d2 == d1", () => d2 == d1); //false
    ShowResult("d1.Equals(d2)", () => d1.Equals(d2)); //false
    ShowResult("d2.Equals(d1)", () => d2.Equals(d1)); //false
    Debug.WriteLine("...whilst these are different objects with the same id; so we treat as
equal when using the overridden Equals method...");
    ShowResult("d1.Equals(d3)", () => d1.Equals(d3)); //true <-- Interesting (sort of;
different to what we saw in comparing o2 with o6; but is just running the code we wrote as we'd
expect)
    ShowResult("d3.Equals(d1)", () => d3.Equals(d1)); //true <-- Interesting (sort of;
different to what we saw in comparing o2 with o6; but is just running the code we wrote as we'd
expect)
    Debug.WriteLine("...but as different when using the other equality tests.");
    ShowResult("ReferenceEquals(d1,d3)", () => ReferenceEquals(d1, d3)); //false <--
Interesting (sort of; same result we had comparing o2 with o6; but shows that ReferenceEquals
does not use the overridden Equals method)
    ShowResult("ReferenceEquals(d3,d1)", () => ReferenceEquals(d3, d1)); //false <--
Interesting (sort of; same result we had comparing o2 with o6; but shows that ReferenceEquals
does not use the overridden Equals method)
    ShowResult("d1 == d3", () => d1 == d3); //false <-- Interesting (sort of; same result we
had comparing o2 with o6; but shows that ReferenceEquals does not use the overridden Equals
method)
    ShowResult("d3 == d1", () => d3 == d1); //false <-- Interesting (sort of; same result we
had comparing o2 with o6; but shows that ReferenceEquals does not use the overridden Equals
method)
```

```
    Debug.WriteLine("For completeness, here's an example of overriding the == operator
(wihtout overriding the Equals method; though in reality if overriding == you'd probably want
to override Equals too).");
    Demo2 d2a = new Demo2(Guid.Empty);
    Demo2 d2b = new Demo2(Guid.NewGuid());
    Demo2 d2c = new Demo2(Guid.Empty);
    ShowResult("d2a == d2a", () => d2a == d2a); //true
    ShowResult("d2b == d2a", () => d2b == d2a); //false
    ShowResult("d2c == d2a", () => d2c == d2a); //true <-- interesting
    ShowResult("d2a != d2a", () => d2a != d2a); //false
    ShowResult("d2b != d2a", () => d2b != d2a); //true
    ShowResult("d2c != d2a", () => d2c != d2a); //false <-- interesting
    ShowResult("ReferenceEquals(d2a,d2a)", () => ReferenceEquals(d2a, d2a)); //true
    ShowResult("ReferenceEquals(d2b,d2a)", () => ReferenceEquals(d2b, d2a)); //false
    ShowResult("ReferenceEquals(d2c,d2a)", () => ReferenceEquals(d2c, d2a)); //false <--
interesting
    ShowResult("d2a.Equals(d2a)", () => d2a.Equals(d2a)); //true
    ShowResult("d2b.Equals(d2a)", () => d2b.Equals(d2a)); //false
    ShowResult("d2c.Equals(d2a)", () => d2c.Equals(d2a)); //false <-- interesting
```

Item 7: Understand the Pitfalls of GetHashCode()

- In .NET, every object has a hash code, determined by System.Object.GetHashCode().

- Any overload of GetHashCode() must follow these three rules:
  - 1. If two objects are equal (as defined by operator==), they must generate the same hash value.
    - Otherwise, hash codes can't be used to find objects in containers.
  - 2. For any object A, A.GetHashCode() must be an instance invariant.
    - No matter what methods are called on A, A.GetHashCode() must always return the same value.
    - That ensures that an object placed in a bucket is always in the right bucket.
  - 3. The hash function should generate a random distribution among all integers for all inputs.
    - That's how you get efficiency from a hash-based container.

- The default GetHashCode() implementation creates sequential numbers
  - Not well distributed
  - Implementation is safe but not efficient

- Best to create your own and use some immutable property of the object

- Item only applies for issues which are indexed in containers

Item 8: Prefer Query Syntax to Loops

Imperative Loops:

```
int[] foo = new int[100];

for (int num = 0; num < foo.Length; num++)
    foo[num] = num * num;

foreach (int i in foo)
  Console.WriteLine(i.ToString());
```

- Declarative

```
int[] foo = (from n in Enumerable.Range(0, 100)
            select n * n).ToArray();

foo.ForAll((n) => Console.WriteLine(n.ToString()));
```

Item 9: Avoid Conversion Operators in Your APIs

```
public class Circle : Shape
{
    private PointF center;
    private float radius;
    public Circle() :
        this(PointF.Empty, 0)
    {}
    public Circle(PointF c, float r)
    {
        center = c;
        radius = r;
    }
    public override void Draw()
    {
        //...
    }
    static public implicit operator Ellipse(Circle c)
    {
        return new Ellipse(c.center, c.center,
          c.radius, c.radius);
    }
}
```

- Conversion Happens Automatically

```
public static double ComputeArea(Ellipse e)
{
    // return the area of the ellipse.
    return e.R1 * e.R2 * Math.PI;
}

// call it:
Circle c1 = new Circle(new PointF(3.0f, 0), 5.0f);
ComputeArea(c1);
```

In this case we are safe because ComputeArea is NOT MODIFYING Elipse e;

But if it DID, it would be modifying a temporary

Conversion operators can lead to subtle side effects that are tricky to track down

Item 10: Use Optional Parameters to Minimize Method Overloads

```
class Program
{
    static void Main()
    {
     // Omit the optional parameters.
     Method();

     // Omit second optional parameter.
     Method(4);

     // You can't omit the first but keep the second.
     // Method("Dot");

     // Classic calling syntax.
     Method(4, "Dot");

     // Specify one named parameter.
     Method(name: "Sam");

     // Specify both named parameters.
     Method(value: 5, name: "Allen");
    }

    static void Method(int value = 1, string name = "Perl")
    {
       Console.WriteLine("value = {0}, name = {1}", value, name);
    }
}
```

Output

value = 1, name = Perl

value = 4, name = Perl

value = 4, name = Dot

value = 1, name = Sam

value = 5, name = Allen

Item 11: Understand the Attraction of Small Functions

- Basic idea
  - Break your code into smaller functions
  - Will allow for better use of LINQ
  - Compiler optimizer will have more options
    - Optimize \ possibly pipeline better for multi-core
    - JIT compiler performance increases
  - Breaking each function down to a single responsibility

Item 11: Understand the Attraction of Small Functions (contd)

- Supports practicing SOLID Principles in design

- SOLID
  - Single Responsibility
    - Function should have a single responsibility

  - Open \ Closed Principle
    - Class is open for extension, but closed for underlying conceptual design modification
      - Having each function with single responsibility helps achieve this goal

  - Liskov Substituion
    - If X is a type of Y, X should be able to substitute for Y without unexpected side effects
    - I.e. x::OnDraw() should NOT also re-sort a database table as a unexpected side effect

  - Interface Segregation
    - IAnimal should not have a Bark() and Meow() functions
    - Instead you should have a ICanBark and ICanMeow interfaces

  - Dependancy Inversion
    - Parts of the system should not depend on explicit parts of the system
    - Instead, depend on Interfaces
    - (True whether or not you are using an Inversion of Control (IOC) framework)

Item 12: Prefer Member Initializers to Assignment Statements

```
// inline member initialization
public class MyClass2
{
    //  declare  the  collection,  and  initialize  it.
    private List<string> labels = new List<string>();

    MyClass2()
    {
    }
}

// Member intialized in constructor
public class MyClass
{
    //  declare  the  collection,  and  initialize  it.
    private List<string> labels ;
    MyClass()
    {
        try
        {
        labels = new List<string>();
        }
        catch() {.....}
    }
    MyClass(int isize)
    {
        try
        {
        labels = new List<string>(isize);
        }
        catch() {.....}
    }
}
```

- Its preferable to initialize the members in the constructor and not inline
- Improved Exception handling
- Reduction of potentially redundant initialization code

Item 13: Use Proper Initialization for Static Class Members

- Its preferable to use a static constructor to initialize a static
- Instead of an inline assignment
- Improved debugging \ Exception handling

```
// static initiailized inline
public  class  MySingleton
{
    private  static  readonly  MySingleton theOneAndOnly=
    new  MySingleton();

    public  static  MySingleton  TheOnly
    {
        get  {  return  theOneAndOnly;  }
    }

    private  MySingleton()
    {
    }
}
```

```
// static initiailized inline
public class MySingleton
{
    static MySingleton theOneAndOnly;
    static MySingleton()
    {
        try
        {
            theOneAndOnly = new MySingleton();
        }
        catch
        {
            //  Attempt  recovery  here.
        }
    }
}
```

Item 14: Minimize Duplicate Initialization Logic

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Number for this instance
    private int counter;
    // Name of the instance:
    private readonly string name;

    public MyClass()
    {
        commonConstructor(0, string.Empty);
    }

    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, string.Empty);
    }

    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string Name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        // ERROR changing the name outside of a constructor.
        this.name = Name;
    }
}
```

Item 15: Utilize using and try/finally for Resource Cleanup

```
try
{
    myConnection = new
SqlConnection(connString);
    myConnection.Open();
}
finally
{
    myConnection.Dispose();
}
```

Item 16: Avoid Creating Unnecessary Objects

```
//  Bad for the GC!
protected  override  void  OnPaint(PaintEventArgs  e)
{
    //  Bad.  Created  the  same  font  every  paint  event.
    using  (Font  MyFont  =  new  Font("Arial",  10.0f))
    {
        e.Graphics.DrawString(DateTime.Now.ToString(),
        MyFont,  Brushes.Black,  new  PointF(0,  0));
    }
    base.OnPaint(e);
}

// Much better- no GC issues
private  readonly  Font  myFont  =
new  Font("Arial",  10.0f);

protected  override  void  OnPaint(PaintEventArgs  e)
{
    e.Graphics.DrawString(DateTime.Now.ToString(),
    myFont,  Brushes.Black,  new  PointF(0,  0));
    base.OnPaint(e);
}
```

Item 17: Implement the Standard Dispose Pattern

```
public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    private bool alreadyDisposed = false;

    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    // Virtual Dispose method
    protected virtual void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (alreadyDisposed)
        return;
        if (isDisposing)
        {
            // elided: free managed resources here.
        }
        // elided: free unmanaged resources here.
        // Set disposed flag:
        alreadyDisposed = true;
    }

    public void ExampleMethod()
    {
        if (alreadyDisposed)
            throw new ObjectDisposedException(
        "MyResourceHog",
        "Called Example Method on Disposed object");
        // remainder elided.
    }
}
```

```
// If a derived class needs to perform additional cleanup,
// it implements the
// protected Dispose method:
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool disposed = false;

    protected override void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (disposed)
        return;
        if (isDisposing)
        {
        // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.

        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);

        // Set derived class disposed flag:
        disposed = true;
    }
}
```

Item 18: Distinguish Between Value Types and Reference Types

- Value types – “struct”
  - Copied by value
- Reference types “class”
  - Copied by reference
- In C++ struct and class are really just about defaulting to private or public exposure of methods
- In C#, a struct is a value type will be copied with a compiler generated copy constructor
- In C# a class is more like a Java reference or a C++ smart pointer

Item 19: Ensure That 0 Is a Valid State for Value Types

- In C#, all initialized values are set to 0
- If you have an enum and do NOT have a value for 0, it’s possible someone using your code could have an invalid enum value

- Guidance
  - Always have a 0 value in enums
  - Even if its ERROR or INVALID

Item 20: Prefer Immutable Atomic Value Types

```
public struct Address2
{
    public string Line1
    {
        get;
        private set;
    }
    public string Line2
    {
        get;
        private set;
    }
    public string City
    {
        get;

        private set;
    }
    public string State
    {
        get;
        private set;
    }
    public int ZipCode
    {
        get;
        private set;
    }
    // .... etc
}
```

- Basic idea is to always make immutable types
- Idea borrowed from “Functional” languages like ML, Haskell, Lisp etc
- Approach allows for a reduction of side effects
- The drawback is to change an object you have to create a new one from a previous one
- C# strings are immutable in this way, which is why we have StringBuilder to allow for stepping around this idea for heavy string concatenation