“C# (pronounced “See Sharp”) is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages, and will be immediately familiar to C, C++, and Java programmers.”

Garbage Collected
Value Types
Generics
Declarative
Functional
LINQ
Unified type system

Sane operator overloading
No pointers
No ADL
No templates
No copy constructors
No mutable-const
Simple with respect to what?

Key(word) differences

Table: Keywords

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>73</td>
<td>77</td>
</tr>
<tr>
<td>Alternate representations</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple with respect to what?

Key(word) differences

Table: Keywords

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>73</td>
<td>77</td>
</tr>
<tr>
<td>Alternate representations</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Context Keywords</td>
<td>2</td>
<td>23</td>
</tr>
</tbody>
</table>
Simple with respect to what?

Key(word) differences

Table : Keywords

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>73</td>
<td>77</td>
</tr>
<tr>
<td>Alternate representations</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Context Keywords</td>
<td>2</td>
<td>23</td>
</tr>
</tbody>
</table>

86 100

K&R C: 27

VB.Net 2008: 154 + 31 “unreserved” keywords
Unspecified behaviour

It’s not complex....

Do you know what a List<T> is?
Unspecified behaviour

It’s not complex....

Do you know what a List\(<T>\) is?

1. linked list?
2. array like?
Unspecified behaviour

It’s not complex....

Do you know what a List<T> is?

1. linked list?
2. array like?

No complexity guarantees
- Adding and removing...
  - at the end?
  - in the middle?
- Sorting?
- Counting?
- Set operations (except, intersection, union)?
Specified un-behaviour

How to generate the docs

...or how to write comments
Not just a... gin-fuelled rant

©Steve Love 2013
static class Lights
{
    enum TrafficLight
    {
        Red = 1,
        Amber = 2,
        Green = 4
    }

    public static void Main()
    {
        Console.WriteLine(light);
    }
}

NB: C++
enum class doesn't allow this.
static class Lights
{
    enum TrafficLight
    {
        Red = 1,
        Amber = 2,
        Green = 4
    }

    public static void Main()
    {
        Console.WriteLine(light);
    }
}

Output: 3

NB: C++ enum class doesn’t allow this.
static class Lights
{
    [Flags]
    enum TrafficLight
    {
        Red = 1,
        Amber = 2,
        Green = 4
    }

    public static void Main()
    {
        Console.WriteLine(light);
    }
}

Output: Red, Amber
SomeType someValue;

- If SomeType is a value type, someValue is definitely assigned.
- otherwise, someValue must be assigned before it’s used.
SomeType someValue;

- If SomeType is a value type, someValue is definitely assigned.
- otherwise, someValue must be assigned before it’s used.

What about incomplete types?

**Always a default**

Reference types can always be null
Value types can always be default-constructed
Whilst we’re on the subject...

The value of the reference
Whilst we’re on the subject...

The value of the reference
No const

- No const parameters
- No const methods

`const` means constant
No const

- No const parameters
- No const methods

`const` means `constant`
`readonly` means `const` (`initonly` in VB)
properties with no `set` cannot be assigned...

*but* that doesn’t work for collections you use `IEnumerable<T>` for that
All bets off

- Call a private method
- Access private data
- Circumvent restrictions on circular references
using System;
using Extensions;

class Extendable
{
    int Munge(int x)
    {
        return x;
    }
}

class Program
{
    static void Main()
    {
        var ext = new Extendable();
        Console.WriteLine(ext.Munge(10));
    }
}
using System;
using Extensions;

public class Extendable
{
    int Munge(int x)
    {
        return x;
    }
}

namespace Extensions
{
    public static class ExtendExtendable
    {
        public static int Munge(this Extendable ext, int x)
        {
            return x * 10;
        }
    }
}

static class Program
{
    static void Main()
    {
        var ext = new Extendable();
        Console.WriteLine(ext.Munge(10));
    }
}
The final word on safety

It must be safe because...
you can write explicitly unsafe code

c
public unsafe class Liberated
{
  public void ICanDoWhatILikeAndItsOk()
  {
  }
}
Remedies

Don’t sweat it

1. Use factories where objects *must* be complete
2. Make value and value-like types immutable
3. Put your extension methods in one place, or a few obvious ones
4. Be vigilant about use of reflection
Remedies

Don’t sweat it

1. Use factories where objects *must* be complete
2. Make value and value-like types immutable
3. Put your extension methods in one place, or a few obvious ones
4. Be vigilant about use of reflection

Corollary

*Unsafe code - Just don’t write any, right?*
...one size to fit them all...
I can’t get no specialisation
Where `T : new();`

Where `T : new( with, args );`

What is this doing?

```csharp
static bool Test<T>( T left, T right )
    where T : class
{
    return left == right;
}
```
There is no constraint which will allow you to use an arithmetic operator with generic parameters.

```c
static bool IsMultiple< T >( T x, T y )
{
    return x % y == 0;
}
```
Remedies

Clever tricks

1. Once again, factories help with the lack of constraints on constructor arguments

2. No operators - a redesign using lambdas and Func or Action arguments instead

e.g.

```csharp
static bool IsMultiple( Func<bool> isMult )
{
    return isMult();
}
static void Main()
{
    Console.WriteLine( IsMultiple( () => 10 % 5 == 0 ) );
}
```
4 Big Lies

1. GC is slow
2. Automatic Garbage Collection means not having to think about object ownership or lifetime any more.
3. A managed program cannot leak memory.
4. The language ensures that dangling references/pointers are not possible.
Create some objects with C a child of B.

B is GC'd, taking C with it, and free space compacted.

The object "A" hasn't actually moved.
Generation game

Create some objects with C a child of B.

B is GC’d, taking C with it, and free space compacted.

The object “A” hasn’t actually moved.
Disposing

It’s harder than it looks
You have to make sure
you’re using it correctly
It can interfere with the
efficient operation of the
garbage collector

...and don’t even think about recycling
Did you know...?

- All classes with custom finalizers make it to (at least) the generation 1 heap.
- It’s possible to prevent an object being finalized
Did you know...?

- All classes with custom finalizers make it to (at least) the generation 1 heap.
- It’s possible to prevent an object being finalized
  - ...without using the GC object
public class Hog
{
    public Hog( Expensive thing )
    {
        this.hog = thing;
    }

    public void Use()
    {
        var count = hog.Count;
        // do something with count
    }

    Expensive hog;
}
public class Hog {
    public Hog(Expensive thing) {
        this.hog = thing;
    }
    public void Use() {
        var count = hog.Count;
        // do something with count
    }
    Expensive hog;
}

public class Hog {
    public Hog(Expensive thing) {
        this.storedCount = hog.Count;
    }
    public void Use() {
        // do something with 
        // stored count instead
    }
    int storedCount;
}
Wotsit dangled;
using( dangled = new Wotsit() )
{
    dangled.Value = 10;
}
Console.WriteLine( dangled.Value );
class Wotsit
{
    public Wotsit()
    {
        mine.Append( "Dangled" );
    }
    ~Wotsit()
    {
        global = mine;
    }
    StringBuilder mine = new StringBuilder();
}

static void Main()
{
    var dangled = new Wotsit();
    GC.Collect();
    GC.WaitForPendingFinalizers();
    Console.WriteLine( global.ToString() );
}
 Remedies

**No magic here**

1. Understand the Dispose Pattern. Deeply and completely.
2. Don’t write finalizers unless you really really have to
   1. ...and if you do, have them invoke Dispose correctly
3. Understand how the GC works, and why
   1. ...which leads to avoiding `GC.Collect()` in code

©Steve Love 2013
Part 4
Equality

(yes, some of you may recognise some of what follows)
My favourite standard

5.10/4 Each of the operators shall yield true if the specified relationship is true and false if it is false.

Time for a quick pop quiz....
object o = "some string";
string s = string.Format( "{0} {1}" , "some", "string" );
Console.WriteLine( o == s );
object o = "some string";
string s = string.Format( "{0} {1}", "some", "string" );
Console.WriteLine( o == s );

False - identity comparison is used.

int i = 10;
object o = i;
Console.WriteLine( object.ReferenceEquals( o, i ) );

©Steve Love 2013
Values and references redux

```
object o = "some string";
string s = string.Format( "{0} {1}" , "some", "string" );
Console.WriteLine( o == s );

False - identity comparison is used.

int i = 10;
object o = i;
Console.WriteLine( object.ReferenceEquals( o, i ) );

False, boxed value-type.

string s = "some string";
object o = string.Format( "{0} {1}" , "some", "string" );
Console.WriteLine( s == o );
```
Values and references redux

```csharp
object o = "some string";
string s = string.Format( "{0} {1}", "some", "string" );
Console.WriteLine( o == s );

False - identity comparison is used.

int i = 10;
object o = i;
Console.WriteLine( object.ReferenceEquals( o, i ) );

False, boxed value-type.

string s = "some string";
object o = string.Format( "{0} {1}", "some", "string" );
Console.WriteLine( s == o );

False, RHS must be cast

Things aren't all they seem however....
```
Equality for all?

```csharp
object a = "10";
string b = "10";
object c = string.Format( "{0}" , a );

Console.WriteLine( a.Equals( b ) );

Console.WriteLine( b.Equals( c ) );

Console.WriteLine( object.ReferenceEquals( a, b ) );

Console.WriteLine( object.ReferenceEquals( a, c ) );
```

Confused?

©Steve Love 2013
Equality for all?

```csharp
object a = "10";
string b = "10";
object c = string.Format( "{0}", a );

Console.WriteLine( a.Equals( b ) );
True - Equals is virtual

Console.WriteLine( b.Equals( c ) );
True - Equals is overridden for strings

Console.WriteLine( object.ReferenceEquals( a, b ) );
True! Equal string constants are interned.

Console.WriteLine( object.ReferenceEquals( a, c ) );
False...of course.

Confused?
```

©Steve Love 2013
Equality for all?

```csharp
object a = "10";
string b = "10";
object c = string.Format("{0}", a);

Console.WriteLine(a.Equals(b));

True - Equals is virtual

Console.WriteLine(b.Equals(c));

True - Equals is overridden for strings

Console.WriteLine(object.ReferenceEquals(a, b));
```

Confused?
Equality for all?

object a = "10";
string b = "10";
object c = string.Format("{0}", a);

Console.WriteLine(a.Equals(b));

True - Equals is virtual

Console.WriteLine(b.Equals(c));

True - Equals is overridden for strings

Console.WriteLine(object.ReferenceEquals(a, b));

True! Equal string constants are interned.

Console.WriteLine(object.ReferenceEquals(a, c));
Equality for all?

```csharp
object a = "10";
string b = "10";
object c = string.Format("{0}", a);

Console.WriteLine(a.Equals(b));
True - Equals is virtual

Console.WriteLine(b.Equals(c));
True - Equals is overridden for strings

Console.WriteLine(object.ReferenceEquals(a, b));
True! Equal string constants are interned.

Console.WriteLine(object.ReferenceEquals(a, c));
False...of course.

Confused?
```

©Steve Love 2013
You can use *anything you like* as a key in a Dictionary. It *might* not behave as you’d expect.

Let’s see an example....
Remedies

Avoid the worst

3 simple rules

1. Don’t muck about with equality of reference types - you’ll regret it
2. Always override equality for value types you create - it’ll be quicker
3. Always, but always, make value types immutable.
Part 5
If it’s so easy...

...it could write itself
Syntax sugar

The `using` statement is a straightforward shortcut

```csharp
static void Main()
{
    {
        var u = new Used();
        try
        {
            // use u
        }
        finally
        {
            u.Dispose();
        }
    }
}
```
Syntax sugar

The `using` statement is a straightforward shortcut

```csharp
static void Main()
{
    {
        var u = new Used();
        try
        {
            // use u
        }
    }
    finally
    {
        u.Dispose();
    }
}
```

```csharp
static void Main()
{
    using(var u = new Used())
    {
        // use u
    }
}
```
Some refinement

The `foreach` is a bit more sophisticated.

- If the iterator implements `IDisposable` you get a `try...finally` added
- `IEnumerator<T> extends IDisposable`
- `IEnumerable<T> returns IEnumerable<T>`
Some refinement

The `foreach` is a bit more sophisticated.

- If the iterator implements `IDisposable` you get a `try...finally` added
- `IEnumerator<T> extends IDisposable`
- `IEnumerable<T> returns IEnumerator<T>`

Modern C# will probably almost *always* have a finally block and a call to a - probably redundant - Dispose method.

- Oh yes...`Dispose` is a virtual call, by the way
The grand-daddy of give-away code is...

The yield statement

For using this keyword, you are rewarded with a complete implementation of IEnumerator<T>
Delegates and lambdas

Lambdas and anonymous delegates also provoke the compiler into writing whole classes for you. Why?

A captured variable lives for at least as long as any delegate instance referring to it.

Jon Skeet, C# In Depth

Read this book!
Nothing for free

The canonical captured trap

```csharp
static void Main()
{
    var items = new List<Func<int>>();
    for (int item = 0; item != 3; ++item)
    {
        items.Add(() => item);
    }
    foreach (var item in items)
    {
        Console.WriteLine(item());
    }
}
```
More pet peeves

1 Why can’t you say
   \[
   \text{var } f = () => x;
   \]

2 Why must I always write `operator!==` if I write `operator===`? There is only one sensible implementation of it!
Remedies

TANSTAAFL

1. Understand what code the compiler is writing for you
2. Be consistent with Dispose and using
3. At the risk of advocating Tool Driven Development, learn to listen to complaints from Resharper, FxCop and similar tools
C# is a doddle

Too easy

We shall have no better conditions in the future if we are satisfied with all those which we have at present.

Thomas Edison