DZone Refcardz

CONTENTS INCLUDE:

C#

By Jon Skeet

ABOUT THIS REFCARD

As C# has evolved over the past few years, there's more and more to remember. While it's still a compact language, it's helpful to have an *aide mémoire* available when you just can't remember that little bit of syntax which would be so handy right now. You'll find this reference card useful whatever type of C# project you're working on, and whichever version of C# you're using. It covers many topics, from the basics of string escape sequences to the brave new world of query expressions and LINQ in C# 3.

STRING LITERALS

C# has two kinds of string literals—the regular ones, and verbatim string literals which are of the form @"text". Regular string literals have to start and end on the same line of source code. A backslash within a string literal is interpreted as an escape sequence as per table 1.

Escape sequence	Result in string
\'	Single quote (This is usually used in character literals. Character literals use the same escape sequences as string literals.)
\"	Double quote
\\	Backslash
\0	Unicode character 0 (the "null" character used to terminate C-style strings)
\a	Alert (Unicode character 7)
\b	Backspace (Unicode character 8)
\t	Horizontal tab (Unicode character 9)
\n	New line (Unicode character 10 = 0xa)
\v	Vertical quote (Unicode character 11 = 0xb)
\f	Form feed (Unicode character 12 = 0xc)
\r	Carriage return (Unicode character 13 = 0xd)
\uxxxx	Unicode escape sequence for character with hex value xxxx. For example, \u20AC is Unicode U+20AC, the Euro symbol.
\xnnnn (variable length)	Like \u but with variable length—escape sequence stops at first non-hexadecimal character. Very hard to read and easy to create bugs—avoid!
\Uxxxxxxxx	Unicode escape sequence for character with hex value xxxxxxxx—generates two characters in the result, used for characters not in the basic multilingual plane.

Table 1. String/character escape sequences

Verbatim string literals can span multiple lines (the whitespace is preserved in the string itself), and backslashes are not interpreted as escape sequences. The only pseudo-escape sequence is for double quotes—you need to include the double quotes twice. Table 2 shows some examples of verbatim string literals, regular string literal equivalents, and the actual resulting string value.

Verbatim string literal	Regular string literal	Result
@"Here is a backslash \"	"Here is a backslash \\"	Here is a backslash \
@"String on two lines"	"String on\r\ntwo lines"	String on two lines
@"Say ""Hello,"" and wave."	"Say \"Hello,\" and wave."	Say "Hello," and wave.
@"ABC"	"\u0041\x42\U00000043"	ABC

Table 2. Sample verbatim and regular string literals

DELEGATES

Delegates show up in various different contexts in .NET—for event handlers, marshalling calls to the UI thread in Windows Forms, starting new threads, and throughout LINQ. A delegate type is known as a function type in other languages—it represents some code which can be called with a specific sequence of parameters and will return a specific type of value.

Delegate type declarations

Declaring a delegate type is like declaring a method, but with the keyword **delegate** before it. For example:

```
delegate bool StringPredicate(string x)
```

Any instance of the StringPredicate type declared above can be invoked with a string parameter, and will return a Boolean value.

Creating delegate instances

Over the course of its evolution, C# has gained more and more ways of creating delegate instances.

C# 1

In C# 1 only a single syntax was available to create a delegate instance from scratch .

```
new delegate-type-name (method-name)
```

→

Delegates, continued

The method name (known as a *method group* in the specification) can be prefixed by a type name to use a static method from a different type, or an expression to give the target of the delegate. For instance, to create an instance of StringPredicate which will return true when passed strings which are five characters long or shorter, and false otherwise, you might use code like this:

```
class LengthFilter
{
    int maxLength;
    public LengthFilter(int maxLength)
    {
        this.maxLength = maxLength;
    }

    public bool Filter(string text)
    {
        return text.Length <= maxLength;
    }
}
// In another class
LengthFilter fiveCharacters = new LengthFilter(5);
StringPredicate predicate =
    new StringPredicate(fiveCharacters.Filter);
```

C# 2

C# 2 introduced two important improvements in the ways we can create delegate instances.

1. You no longer need the new delegate-type part:

```
StringPredicate predicate = fiveCharacters.Filter;
```

2. *Anonymous methods* allow you to specify the logic of the delegate in the same statement in which you create the delegate instance. The syntax of an anonymous method is simply the keyword delegate followed by the parameter list, then a block of code for the logic of the delegate.

All of the earlier code to create a StringPredicate can be expressed in a single statement:

```
StringPredicate predicate = delegate (string text)
    { return text.Length <= 5; } ;
```

Note that you don't declare the return type of the anonymous method—the compiler checks that every return value within the anonymous method can be implicitly converted to the return type of the delegate. If you don't need to use any of the delegate's parameters, you can simply omit them. For instance, a StringPredicate which returns a result based purely on the time of day might look like this:

```
StringPredicate predicate = delegate
    { return DateTime.Now.Hour >= 12; } ;
```

One important feature of anonymous methods is that they have access to the local variables of the method in which they're created. This implements the notion of *closures* in other languages. There are important subtleties involved in closures, so try to keep things simple. See http://csharpindepth.com/Articles/Chapter5/Closures.aspx or chapter 5 of *C# in Depth (Manning, 2008)* for more details.

C# 3

C# 3 adds lambda expressions which are like anonymous methods but even more concise. In their longest form, lambda expressions look very much like anonymous methods, but with => after the parameter list instead of delegate before it:

```
StringPredicate predicate =
    (string text) => { return text.Length <=5;  };
```

However, lambda expressions have many special cases to make them shorter:

- If the compiler can infer the types of the parameters (based on the context) then the types can be omitted. (C# 3 has far more powerful type inference than C# 2.)
- If there is only a single parameter and its type is inferred, the parentheses around the parameter list aren't needed.
- If the body of the delegate is just a single statement, the braces around it aren't needed—and for single-statement delegates returning a value, the return keyword isn't needed.

Applying all of these shortcuts to our example, we end up with:

```
StringPredicate predicate = text => text.Length <=5;
```

DECLARING EVENTS

Events are closely related to delegates, but they are not the same thing. An event allows code to subscribe and unsubscribe using delegate instances as event handlers. The idea is that when an event is raised (for instance when a button is clicked) all the event handlers which have subscribed to the event are called. Just as a property is logically just the two operations get and set, an event is also logically just two operations: *subscribe* and *unsubscribe*. To declare an event and explicitly write these operations, you use syntax which looks like a property declaration but with add and remove instead of get and set:

```
public event EventHandler CustomEvent
{
    add
    {
        // Implementation goes here: "value" variable is the
        // handler being subscribed to the event
    }
    remove
    {
        // Implementation goes here: "value" variable is the
        // handler being unsubscribed from the event
    }
}
```

Hot Tip Don't forget that a single delegate instance can refer to many actions, so you only need one variable even if there are multiple subscribers.

Many events are implemented using a simple variable to store the subscribed handlers.

C# allows these events to be created simply, as field-like events:

```
public event EventHandler SimpleEvent;
```

This declares both the event and a variable at the same time. It's roughly equivalent to this:

```
private EventHander __hiddenField;
public event EventHandler SimpleEvent
{
    add
    {
        lock(this)
        {
            __hiddenField += value;
        }
    }
    remove
    {
        lock(this)
        {
            __hiddenField -= value;
        }
    }
}
```

Declaring Events, continued

Everywhere you refer to SimpleEvent within the declaring type, the compiler actually references __hiddenField, which is why you're able to raise the event by calling SimpleEvent(). Outside the type declaration, however, SimpleEvent only refers to the event. This duality has caused confusion for many developers—you just need to remember that fields and events really are very different things, and field-like events are just the compiler doing some work for you.

NULLABLE VALUE TYPES

Nullable value types were introduced in .NET 2.0 and C# 2, with support being provided by the framework, the language and the runtime. The principle is quite straightforward: a new struct System.Nullable<T> has been introduced which contains a value of type T (which must be another value type) and a flag to say whether or not this value is "null". The HasValue property returns whether or not it's a null value, and the Value property returns the embedded value or throws an exception if you try to call it on a null value. This is useful when you want to represent the idea of an unknown value.

The runtime treats nullable value types in a special manner when it comes to boxing and unboxing. The boxed version of a Nullable<int> is just a boxed int or a null reference if the original value was a null value. When you unbox, you can either unbox to int or to Nullable<int> (this follows for all nullable value types—I'm just using int as a concrete example).

C# support for nullable value types

C# adds a sprinkling of syntactic sugar. Firstly, writing Nullable<int> etc. can get quite tedious—so C# lets you just add a question mark to the normal type name to mean "the nullable version". Thus Nullable<int> and int? are exactly the same thing, and can be used entirely interchangeably.

The null-coalescing operator has been introduced to make working with null values (both of nullable value types and normal reference types) easier. Consider this expression:

```
left ?? right
```

At execution time, first left is evaluated. If the result is non-null, that's the result of the whole expression and right is never evaluated. Otherwise, right is evaluated and that's the result of the expression. The null-coalescing operator is right associative, which means you can string several expressions together like this:

```
first ?? second ?? third ?? fourth ?? fifth
```

The simple way of understanding this is that each expression is evaluated, in order, until a non-null result is found, at which point the evaluation stops.

Hot Tip One thing to be aware of is that there is also a nongeneric class **System.Nullable**, which just provides some static support methods for nullable types. Don't get confused between **Nullable** and **Nullable<T>**.

Hot Tip The C# compiler also lifts operators and conversions—for instance, because int has an addition operator, so does **Nullable<int>**. Beware of one conversion you might not expect or want to happen—a comparison between a normal non-nullable value type, and the null literal. Here's some code you might not expect to compile:

```
int i = 5;
if (i == null)
{
...
}
```

How can it possibly be null? It can't, of course, but the compiler is using the lifted == operator for **Nullable<int>** which makes it legal code. Fortunately the compiler issues a warning in this situation.

GENERICS

The biggest feature introduced in C# 2 was generics—the ability to parameterise methods and types by *type parameters*. It's an ability which is primarily used for collections by most people, but which has a number of other uses too. I can't cover generics in their entirety in this reference card—please read online documentation or a good book about C# for a thorough grounding on the topic—but there are some areas which are useful to have at your fingertips. Following are some references:

Reference	Resource
MSDN	http://msdn.microsoft.com/en-us/library/512aeb7t.aspx
C# in Depth (Manning Publications)	http://books.dzone.com/books/csharp

Syntax: declaring generic types and methods

Only types and methods can be generic. You can have other members which use the type parameter of declaring type (just as the Current property of IEnumerable<T> is of type T, for example) but only types and methods can introduce new type parameters.

For both methods and types, you introduce new type parameters by putting them after the name, in angle brackets. If you need more than one type parameter, use commas to separate them. Here are examples, both from List<T> (one of the most commonly used generic types). (In MSDN a lot of other interfaces are also listed, but they're all covered by ICollection<T> anyway.)

Generic type declaration:

```
public class List<T> : ICollection<T>
```

Generic method declaration:

```
public List<TOutput> ConvertAll<TOutput>
    (Converter<T, TOutput> converter)
```

A few things to note here:

- You can use the newly introduced type parameters for the rest of the declaration—the interfaces a type implements, or its base type it derives from, and in the parameters of a method.
- Even though ConvertAll uses both T and TOutput, it only introduces TOutput as a type parameter—the T in the declaration is the same T as for the List<T> as a whole.

→

Generics, continued

- The parameter to ConvertAll is another generic type—in this case, a generic delegate type, representing a delegate which can convert a value from one type (T in this case) to another (TOutput).
- Method signatures can get pretty hairy when they use a lot of type parameters, but if you look at where each one has come from and what it's used for, you can tame even the wildest declaration.

Type constraints

You can add type constraints to generic type or method declarations to restrict the set of types which can be used, with the where contextual keyword. For instance, Nullable<T> has a constraint so that T can only be a non-nullable value type—you can't do Nullable<string> or Nullable<Nullable<int>>, for example. Table 3 shows the constraints available.

Syntax	Notes
T : class	T must be a reference type—a class or delegate, an array, or an interface
T : struct	T must be a non-nullable value type (e.g. int, Guid, DateTime)
T : Stream T : IDisposable T : U	T must inherit from the given type, which can be a class, interface, or another type parameter. (T can also be the specified type itself – for instance, Stream satisfies T : Stream.)
T : new()	T must have a parameterless constructor. This includes all value types.

Table 3. Type constraints for generic type parameters

In both methods and type declarations, the constraints come just before the opening brace of the body of the type/method. For example:

```
// Generic type
public class ResourceManager<T> where T : IDisposable
{
    // Implementation
}
// Generic method
public void Use<T>(Func<T> source, Action<T> action)
    where T : IDisposable
{
    using (T item = source())
    {
        action(item);
    }
}
```

Type constraints can be combined (comma-separated) so long as you follow certain rules. For each type parameter:

- Only one of "class" or "struct" can be specified, and it has to be the first constraint.
- You can't force a type parameter to inherit from two different classes, and if you specify a class it must be the first inheritance constraint. (However, you can specify a class, multiple interfaces, and multiple type parameters—unlikely as that may be!)
- You can't force a type parameter to inherit from a sealed class, System.Object, System.Enum, System.ValueType or System.Delegate.
- You can't specify a "class" constraint and specify a class to derive from as it would be redundant.
- You can't specify a "struct" constraint and a "new()" constraint—again, it would be redundant.
- A "new()" constraint always comes last.

You can specify different sets of constraints for different type parameters; each type parameter's constraints are introduced with an extra where. All of the examples below would be valid type declarations (and the equivalent would be valid for method declarations too):

```
class Simple<T> where T : Stream, new()
class Simple<T> where T : struct, IComparable<T>
class Simple<T, U> where T : class where U : struct
class Odd<T, U> where T : class where U : struct, T
class Bizarre<T, U, V> where T : Stream, IEnumerable,
                      IComparable, U, V
```

The Odd class may appear to have constraints which are impossible to satisfy—how can a value type inherit from a reference type? Remember that the "class" constraint also includes interfaces, so Odd<IComparable,int> would be valid, for example. It's a pretty strange set of constraints to use though.

Using type parameters

We've seen that you can use type parameters in type and method declarations, but you can do much more with them, treating them much like any "normal" type name:

- Declare variables using them, such as:

```
T currentItem;
T[] buffer;
IEnumerable<T> sequence;
```

- Use typeof to find out at execution time which type is actually being used:

```
Type t = typeof(T);
```

- Use the default operator to get the default value for that type. This will be null for reference types, or the same result returned by new T() for value types. For example:

```
T defaultValue = default(T);
```

- Create instances of other generic classes:

```
sequence = new LinkedList<T>();
```

FYI Note that the typeof operator can be used to get generic types in their "open" or "closed" forms, e.g. typeof(List<>) and typeof(List<int>). Reflection with generic types and methods is tricky, but MSDN (http://msdn.microsoft.com/library/System.Type.IsGenericType.aspx) has quite good documentation on it.

Lack of variance: why a List<Banana> isn't a List<Fruit>

Probably the most frequently asked question around .NET generics is why it doesn't allow *variance*. This comes in two forms: *covariance* and *contravariance*—but the actual terms aren't as important as the principle. Many people initially expect the following code to compile:

```
List<Banana> bananas = new List<Banana>();
List<Fruit> fruit = bananas;
```

It would make a certain amount of sense for that to work – after all, if you think of it in human language, a collection of bananas *is* a collection of fruit. However, it won't compile for a very good reason. Suppose the next line of code had been:

```
fruit.Add(new Apple());
```

That would have to compile—after all, you can add an Apple to a List<Fruit> with no problems... but in this case our list of fruit is actually meant to be a list of bananas, and you certainly can't add an apple to a list of bananas!

→

Generics, continued

FYI

Unfortunately, when this becomes a problem you just have to work around it. That may mean copying data into the right kind of collection, or it may mean introducing another type parameter somewhere (i.e. making a method or type more generic). It varies on a case-by-case basis, but you're in a better position to implement the workaround when you understand the limitation.

EXTENSION METHODS

Extension methods were introduced in C# 3. They're static methods declared in static classes—but they are usually used as if they were instance methods of a completely different type! This sounds bizarre and takes a few moments to get your head round, but it's quite simple really. Here's an example:

```
using System;
public static class Extensions
{
    public static string Reverse(this string text)
    {
        char[] chars = text.ToCharArray();
        Array.Reverse(chars);
        return new string(chars);
    }
}
```

Note the use of the keyword **this** in the declaration of the **text** parameter in the **Reverse** method. That's what makes it an extension method—and the type of the parameter is what makes it an extension method on **string**. It's useful to be able to talk about this as the *extended type* of the extension method, although that's not official terminology.

The body of the method is perfectly ordinary. Let's see it in use:

```
class Test
{
    static void Main()
    {
        Console.WriteLine ("dlrow olleH".Reverse());
    }
}
```

There's no explicit mention of the **Extensions** class, and we're using the **Reverse** method as if it were an instance method on string. To let the compiler know about the **Extensions** class, we just have to include a normal using directive for the namespace containing the class. That's how **IEnumerable<T>** seems to gain a load of extra methods in .NET 3.5—the **System.Linq** namespace contains the **Enumerable** static class, which has lots of extension methods. A few things to note:

- Extension methods can only be declared in static non-nested classes.
- If an extension method and a regular instance method are both applicable, the compiler will always use the instance method.
- Extension methods work under the hood by the compiler adding the **System.Runtime.CompilerServices.ExtensionAttribute** attribute to the declaration. If you want to target .NET 2.0 but still use extension methods, you just need to write your own version of the attribute. (It doesn't have any behaviour to implement.)
- The extended type can be almost anything, including value types, interfaces, enums and delegates. The only restriction is that it can't be a pointer type.
- The first parameter of an extension method can't have any other modifiers such as **out** or **ref**.
- Unlike normal instance methods, extension methods can be called "on" a null reference. In other words, don't assume that the first parameter will be non-null.
- Extension methods are fabulous for allowing the result of one method to be the input for the next. Again, this is how LINQ to Objects works—many of the extension methods return an **IEnumerable<T>** (or another interface inheriting from it) which allows another method call to appear immediately afterwards. For example:

```
people.Where(p => p.Age >=18
             .OrderBy(p => p.LastName)
             .Select(p => p.FullName)
```

QUERY EXPRESSIONS (C# 3)

If you've seen any articles at all on C# 3, you'll almost certainly have seen a query expression, such as this:

```
from person in people
where person.Age >= 18
orderby person.LastName
select person.FullName
```

Query expressions are the "LIN" part of LINQ—they provide the language integration. The query expression above looks very different from normal C#, but it is extremely readable. Even if you've never seen one before, I expect you can understand the basics of what it's trying to achieve.

Query expressions are translated into "normal" C# 3 as a sort of pre-processor step. This is done in a manner which knows nothing about LINQ itself—there are no ties between query expressions and the **System.Linq** namespace, or **IEnumerable<T>** and **IQueryable<T>**. The translation rules are all documented in the specification—there's no magic going on, and you can do everything in query expressions in "normal" code too.

The details can get quite tricky, but table 4 gives an example of each type of clause available, as well as which methods are called in the translated code.

Clause	Full example	Methods called for clause
First "from" (implicit type)	**from p in people** select p	n/a
First "from" (explicit type)	**from Person p in people** select p	Cast<T> (where T is the specified type)
Subsequent "from"	from p in people **from j in jobs** select new { Person=p, Job=j }	SelectMany
where	from p in people **where p.Age >= 18** select p	Where
select	from p in people **select p.FullName**	Select
let	from p in people **let income = p.Salary +** **p.OtherIncome** select new { Person=p, Income=income}	Select
orderby	from p in people **orderby p.LastName,** **p.FirstName,** **p.Age descending**	OrderBy OrderByDescending ThenBy ThenByDescending (depending on clauses)
join	from p in people **join job in jobs** **on p.PrimarySkill** **equals job.RequiredSkill** select p.FullName + ": " + job.Description	Join

Table 4. Clauses used in query expressions

Query Expressions (C#3), continued

Clause	Full example	Methods called for clause
join ... into	from p in people **join job in jobs** **on p.PrimarySkill** **equals job.RequiredSkill** **into jobOptions** select p.FullName + ": " + jobOptions.Count()	GroupJoin
group ... by	from p in people **group p by p.LastName**	GroupBy

Table 4. Clauses used in query expressions

If a "select" or "group ... by" clause is followed by "into", it effectively splits the query into two. For instance, take the following query, which shows the size of all families containing more than 4 people:

```
var result = from p in people
             group p by p.LastName into family
             let size = family.Count()
             where size > 4
             select family.Key + ": " + size
```

We can split the above into two separate query expressions:

```
var tmp = from p in people
          group p by p.LastName;

var result = from family in tmp
             let size = family.Count()
             where size > 4
             select family.Key + ": " + size;
```

Splitting things up this way can help to turn a huge query into several more manageable ones, and the results will be exactly the same.

This is only scratching the surface of LINQ. For further details, I recommend reading *LINQ in Action* (Manning, 2008).

ABOUT THE AUTHOR

Jon Skeet

Jon Skeet is a software engineer with experience in both C# and Java, currently working for Google in the UK. Jon has been a C# MVP since 2003, helping the community through his newsgroup posts, popular web articles, and a blog covering C# and Java. Jon's recent book *C# in Depth* looks at the details of C# 2 and 3, providing an ideal guide and reference for those who know C# 1 but want to gain expertise in the newer features.

Publications

Author of *C# in Depth* (Manning, 2008), co-author of *Groovy in Action* (Manning, 2007)

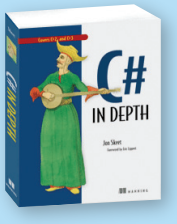
Blog

http://msmvps.com/jon.skeet

Web Site

http://pobox.com/~skeet/csharp

RECOMMENDED BOOK



C# in Depth is designed to bring you to a new level of programming skill. It dives deeply into key C# topics—in particular the new ones in C# 2 and 3. Make your code more expressive, readable and powerful than ever with LINQ and a host of supporting features.

BUY NOW

books.dzone.com/books/csharp

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

Available:

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server
Silverlight 2

Visit **refcardz.com** for a complete listing of available Refcardz.

Design Patterns
Published June 2008

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-15-8
ISBN-10: 1-934238-15-5
50795
9 781934 238158
$7.95