

Generics in C#, Java, and C++

A Conversation with Anders Hejlsberg, Part VII

by Bill Venners with Bruce Eckel

January 26, 2004

Summary

Anders Hejlsberg, the lead C# architect, talks with Bruce Eckel and Bill Venners about C# and Java generics, C++ templates, constraints, and the weak-strong typing dial.

Anders Hejlsberg, a distinguished engineer at Microsoft, led the team that designed the C# (pronounced C Sharp) programming language. Hejlsberg first vaulted onto the software world stage in the early eighties by creating a Pascal compiler for MS-DOS and CP/M. A very young company called Borland soon hired Hejlsberg and bought his compiler, which was thereafter marketed as Turbo Pascal. At Borland, Hejlsberg continued to develop Turbo Pascal and eventually led the team that designed Turbo Pascal's replacement: Delphi. In 1996, after 13 years with Borland, Hejlsberg joined Microsoft, where he initially worked as an architect of Visual J++ and the Windows Foundation Classes (WFC). Subsequently, Hejlsberg was chief designer of C# and a key participant in the creation of the .NET framework. Currently, Anders Hejlsberg leads the continued development of the C# programming language.

On July 30, 2003, Bruce Eckel, author of *Thinking in C++* and *Thinking in Java*, and Bill Venners, editor-in-chief of Artima.com, met with Anders Hejlsberg in his office at Microsoft in Redmond, Washington. In this interview, which is being published in multiple installments on Artima.com and on an audio CD-ROM to be released by Bruce Eckel, Anders Hejlsberg discusses many design choices of the C# language and the .NET framework.

- In [Part I: The C# Design Process](#), Hejlsberg discusses the process used by the team that designed C#, and the relative merits of usability studies and good taste in language design.
- In [Part II: The Trouble with Checked Exceptions](#), Hejlsberg discusses versionability and scalability issues with checked exceptions.
- In [Part III: Delegates, Components, and Simplicity](#), Hejlsberg discusses delegates and C#'s first class treatment of component concepts.
- In [Part IV: Versioning, Virtual, and Override](#), Hejlsberg explains why C# instance methods are non-virtual by default and why programmers must explicitly indicate an override.
- In [Part V: Contracts and Interoperability](#), Hejlsberg discusses DLL hell and interface contracts, strong names, and the importance of interoperability.
- In [Part VI: Inappropriate Abstractions](#), Hejlsberg and other members of the C# team discuss the trouble with distributed systems infrastructures that attempt to make the network transparent,

and object-relational mappings that attempt to make the database invisible.

- In this seventh installment, Hejlsberg compares C#'s generics implementation to Java generics and C++ templates, describes constraints in C# generics, and describes typing as a dial.

Generics in General

Bruce Eckel: Can you give a quick introduction to generics?

Anders Hejlsberg: Generics is essentially the ability to have type parameters on your type. They are also called parameterized types or parametric polymorphism. The classic example is a `List` collection class. A `List` is a convenient growable array. It has a `sort` method, you can index into it, and so on. Today, without parameterized types, there's a tension between either using arrays or `Lists` for your collections. If you use an array, you get strong typing, because you can declare an array of `Customer`, but you don't get growability or the convenience methods. If you use a `List`, you get all the conveniences, but you lose the strong typing. You can't say on a `List` what it's a `List` of. It's just a `List of Object`. And this gives you problems. Types must be checked at runtime, which by inference also means that types aren't checked at compile time. The compiler is perfectly happy to let you stick a `Customer` in a `List` and try to take out a `String`. You don't find out until runtime that it's not going to work. Also, when you put primitive types in a `List`, you have to box them. Because of all these issues, there's constantly this tension between `Lists` and arrays. You are always struggling to decide which you should pick.

The great thing about generics is now you can have your cake and eat it to, because you can define a `List<T>` [pronounced: `List of T`]. When you use a `List`, you can actually say what the `List` is a `List of`, and have the compiler strongly type check it for you. So that's the immediate benefit. And then there's all sorts of other benefits. Of course, you don't want to do this only with `Lists`. A `Hashtable`, or `Dictionary`—whatever you want to call it—maps keys to values. You may want to map `Strings` to `Customers`, or `ints` to `Orders`, and you want to be able to say that in a strongly typed fashion.

Generics in C#

Bill Venners: How do generics work in C#?

Anders Hejlsberg: In C# without generics, you are basically able to say `class List {...}`. In C# with generics, you can say `class List<T> {...}`, where `T` is the type parameter. Within `List<T>` you can use `T` as if it were a type. When it actually comes time to create a `List` object, you say `List<int>` or `List<Customer>`. You construct new types from that `List<T>`, and it is truly as if your type arguments get substituted for the type parameter. All of the `Ts` become `ints` or `Customers`, you don't have to downcast, and there is strong type checking everywhere.

In the CLR [Common Language Runtime], when you compile `List<T>`, or any other generic type, it compiles down to IL [Intermediate Language] and metadata just like any normal type. The IL and metadata contains additional information that knows there's a type parameter, of course, but in

principle, a generic type compiles just the way that any other type would compile. At runtime, when your application makes its first reference to `List<int>`, the system looks to see if anyone already asked for `List<int>`. If not, it feeds into the JIT the IL and metadata for `List<T>` and the type argument `int`. The JITter, in the process of JITting the IL, also substitutes the type parameter.

Bruce Eckel: So it's instantiating at runtime.

Anders Hejlsberg: It's instantiating at runtime, exactly. It's producing native code specifically for that type at the point it is needed. And literally when you say `List<int>`, you will get a `List` of `int`. If the code in the generic type uses an array of `T`, that becomes an array of `int`.

Bruce Eckel: Does it garbage collect that class at some point?

Anders Hejlsberg: Yes and no, but that's an orthogonal issue. It creates the class in that app domain, and then the class lives forever in that app domain. If you kill the app domain, the class goes away, like any other class.

Bruce Eckel: But if I have an application that uses a `List<int>` and a `List<Cat>`, but it never goes down the branch that uses `List<Cat>`,...

Anders Hejlsberg: ...then the system won't instantiate a `List<Cat>`. Now, there are exceptions to that rule. If you're NGENing an image, that is, if you're generating a native image up front, you can generate instantiations early. But if you're running under normal circumstances, the instantiations are purely demand driven, and they are deferred to as late as possible.

Now, what we then do is for all type instantiations that are value types—such as `List<int>`, `List<long>`, `List<double>`, `List<float>`—we create a unique copy of the executable native code. So `List<int>` gets its own code. `List<long>` gets its own code. `List<float>` gets its own code. For all reference types we share the code, because they are representationally identical. It's just pointers.

Bruce Eckel: And you just need to cast.

Anders Hejlsberg: No, you don't actually. We can share the native image, but they actually have separate VTables. I'm just pointing out that we do fairly aggressive code sharing where it makes sense, but we are also very conscious about not sharing where you want the performance. Typically with value types, you really do care that `List<int>` is `int`. You don't want them to be boxed as `Objects`. Boxing value types is one way we could share, but boy it would be an expensive way.

Bill Venners: In the reference case, you actually have different classes. `List<Elephant>` is different from `List<Orangutan>`, but they actually share all the same method code.

Anders Hejlsberg: Yes. As an implementation detail, they actually share the same native code.

Comparing C# and Java Generics

Bruce Eckel: How do C# generics compare with Java generics?

Anders Hejlsberg: Java's generics implementation was based on a project originally called Pizza, which was done by Martin Odersky and others. Pizza was renamed GJ, then it turned into a JSR and ended up being adopted into the Java language. And this particular generics proposal had as a key design goal that it could run on an unmodified VM [Virtual Machine]. It is, of course, great that you don't have to modify your VM, but it also brings about a whole bunch of odd limitations. The limitations are not necessarily directly apparent, but you very quickly go, "Hmm, that's strange."

For example, with Java generics, you don't actually get any of the execution efficiency that I talked about, because when you compile a generic class in Java, the compiler takes away the type parameter and substitutes `Object` everywhere. So the compiled image for `List<T>` is like a `List` where you use the type `Object` everywhere. Of course, if you now try to make a `List<int>`, you get boxing of all the `ints`. So there's a bunch of overhead there. Furthermore, to keep the VM happy, the compiler actually has to insert all of the type casts you didn't write. If it's a `List` of `Object` and you're trying to treat those `Objects` as `Customers`, at some point the `Objects` must be cast to `Customers` to keep the verifier happy. And really all they're doing in their implementation is automatically inserting those type casts for you. So you get the syntactic sugar, or some of it at least, but you don't get any of the execution efficiency. So that's issue number one I have with Java's solution.

Issue number two, and I think this is probably an even bigger issue, is that because Java's generics implementation relies on erasure of the type parameter, when you get to runtime, you don't actually have a faithful representation of what you had at compile time. When you apply reflection to a generic `List` in Java, you can't tell what the `List` is a `List` of. It's just a `List`. Because you've lost the type information, any type of dynamic code-generation scenario, or reflection-based scenario, simply doesn't work. If there's one trend that's pretty clear to me, it's that there's more and more of that. And it just doesn't work, because you've lost the type information. Whereas in our implementation, all of that information is available. You can use reflection to get the `System.Type` for object `List<T>`. You cannot actually create an instance of it yet, because you don't know what `T` is. But then you can use reflection to get the `System.Type` for `int`. You can then ask reflection to please put these two together and create a `List<int>`, and you get another `System.Type` for `List<int>`. So representationally, anything you can do at compile time you can also do at runtime.

Comparing C# Generics to C++ Templates

Bruce Eckel: How do C# generics compare with C++ templates?

Anders Hejlsberg: To me the best way to understand the distinction between C# generics and C++ templates is this: C# generics are really just like classes, except they have a type parameter. C++ templates are really just like macros, except they look like classes.

The big difference between C# generics and C++ templates shows up in when the type checking occurs and how the instantiation occurs. First of all, C# does the instantiation at runtime. C++ does it at compile time, or perhaps at link time. But regardless, the instantiation happens in C++ before the program runs. That's difference number one. Difference number two is C# does strong type checking when you compile the generic type. For an unconstrained type parameter, like `List<T>`, the only methods available on values of type `T` are those that are found on type `Object`, because those are the

only methods we can generally guarantee will exist. So in C# generics, we guarantee that any operation you do on a type parameter will succeed.

C++ is the opposite. In C++, you can do anything you damn well please on a variable of a type parameter type. But then once you instantiate it, it may not work, and you'll get some cryptic error messages. For example, if you have a type parameter `T`, and variables `x` and `y` of type `T`, and you say `x + y`, well you had better have an `operator+` defined for `+` of two `T`s, or you'll get some cryptic error message. So in a sense, C++ templates are actually untyped, or loosely typed. Whereas C# generics are strongly typed.

Constraints in C# Generics

Bruce Eckel: How do constraints work in C# generics?

Anders Hejlsberg: In C# generics, we have the ability to put constraints on type parameters. Take for example our `List<T>`. You could then say, `class List<T> where T: IComparable`. And that means `T` must implement `IComparable`.

Bruce Eckel: Interesting. In C++, the constraints are implied.

Anders Hejlsberg: Yes. And in C#, you can of course do the same thing. Say we have a `Dictionary<K, V>` that has an `add` method that takes `K key` and `V value`. The `add` method implementation will likely want to compare the passed key to the keys already in the `Dictionary`, and it might want to do that using an interface called `IComparable`. One way it could do this is cast the `key` parameter to `IComparable` and then invoke the `compareTo` method. And of course, when you do that you have created an implicit constraint on the `K` type and on the `key` parameter. If the passed key doesn't implement `IComparable`, you get a runtime error. But it's nowhere stated really in any of your methods or your contracts that `key` must implement `IComparable`. And of course you have to pay the overhead of runtime type tests, because you're actually doing a dynamic type check at runtime.

With a constraint, you can hoist that dynamic check out of your code and have it be verifiable at compile time or load time. When you say `K` must implement `IComparable`, a couple of things happen. On any value of type `K`, you can now directly access the interface methods without a cast, because semantically in the program it's guaranteed that it will implement that interface. Whenever you try and create an instantiation of that type, the compiler will check that any type you give as the `K` argument implements `IComparable`, or else you get a compile time error. Or if you're doing it with reflection you get an exception.

Bruce Eckel: You said the compiler *and* the runtime.

Anders Hejlsberg: The compiler checks it, but you could also be doing it at runtime with reflection, and then the system checks it. As I said before, anything you can do at compile time, you can also do at runtime with reflection.

Bruce Eckel: Will I be able to do a template function, in other words, a function where the argument is the unknown type? You are adding stronger type checking to the containers, but can I also get a kind of weaker typing as I can get with C++ templates? For example, will I be able to write a function that takes parameters `A a` and `B b`, and then in the code say, `a + b`? Can I say that I don't care what `A` and `B` so long as there's an `operator+` for them, because that's kind of a weak typing.

Anders Hejlsberg: What you are really asking is, what can you say in terms of constraints? Constraints, like any other feature, can become arbitrarily complex if taken to their ultimate extreme. When you think about it, constraints are a pattern matching mechanism. You want to be able to say, "This type parameter must have a constructor that takes two arguments, implement `operator+`, have this static method, has these two instance methods, etc." The question is, how complicated do you want this pattern matching mechanism to be?

There's a whole continuum from nothing to grand pattern matching. We think it's too little to say nothing, and the grand pattern matching becomes very complicated, so we're in-between. We allow you to specify a constraint that can be one class, zero or more interfaces, and something called a constructor constraint. You can say, for example, "This type must implement `IFoo` and `IBar`," or "This type must inherit from base class `X`." Once you do that, we type check everywhere, at both compile and run time, that the constraint is true. Any methods implied by that constraint are directly available on values of that type parameter type.

Now, in C#, operators are static members. So, an operator can never be a member of an interface, and therefore an interface constraint could never endow you with an `operator+`. The only way you can endow yourself with an `operator+` is by having a class constraint that says you must inherit from, say, `Number`, and `Number` has an `operator+` of two `Numbers`. But you could not in the abstract say, "Must have an `operator+`," and then we polymorphically resolve what that means.

Bill Venners: You did the constraints by type, not by signature.

Anders Hejlsberg: Yes.


Bill Venners: So the type must either extend a class or implement interfaces.

Anders Hejlsberg: Yes. And we could have gone further. We did give thought to going further, but it gets very complicated. And it's not clear that the added complexity is worth the small yield that you get. If something you want to do is not directly supported in the constraint system, you can do it with a factory pattern. You could have a `Matrix<T>`, for example, and in that `Matrix` you would like to define a dot product method. That of course that means you ultimately need to understand how to multiply two `Ts`, but you can't say that as a constraint, at least not if `T` is `int`, `double`, or `float`. But what you could do is have your `Matrix` take as an argument a `Calculator<T>`, and in `Calculator<T>`, have a method called `multiply`. You go implement that and you pass it to the `Matrix`.

Bruce Eckel: And `Calculator` is a parameterized type also.

Anders Hejlsberg: Yes. It is sort of a factory pattern. So there are ways of doing these things. It's maybe not quite as nice as you'd like, but everything comes at a price.

Bruce Eckel: Well, yes, I started seeing C++ templates as sort of a weak typing mechanism. And when you start adding constraints on top of that, you're going from the weak typing to the strong typing. It always gets more complex when you add strong typing. It's a spectrum.

Anders Hejlsberg: The thing you realize about typing is that it's a dial. The higher you place the dial, the more painful the programmer's life becomes, but the safer it becomes too. But you can turn that dial too far in either direction. 

Next Week

Come back Monday, February 2 (Ground Hog's Day!) for the final installment of this conversation with C#'s creator Anders Hejlsberg. If you'd like to receive a brief weekly email announcing new articles at Artima.com, please subscribe to the [Artima Newsletter](#).

Talk Back!

Have an opinion about the design principles presented in this article? Discuss this article in the Articles Forum topic, [Generics in C#, Java, and C++](#).

Resources

Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg:
http://windows.oreilly.com/news/hejlsberg_0800.html

A Comparative Overview of C#:
http://genamics.com/developer/csharp_comparative.htm

Microsoft Visual C#:
<http://msdn.microsoft.com/vcsharp/>

Dan Fernandez's Weblog:
<http://blogs.gotdotnet.com/danielfe/>

Eric Gunnerson's Weblog:
<http://blogs.gotdotnet.com/ericgu/>

[Interviews](#) | [Discuss](#) | [Print](#) | [Email](#) | [Screen Friendly Version](#) | [Previous](#) | [Next](#)

Copyright © 1996-2017 Artima, Inc. All Rights Reserved. - [Privacy Policy](#) - [Terms of Use](#) - [Advertise with Us](#)
