



Hanselminutes

Hanselminutes is a weekly audio talk show with noted web developer and technologist Scott Hanselman and hosted by Carl Franklin. Scott discusses utilities and tools, gives practical how-to advice, and discusses ASP.NET or Windows issues and workarounds.

## Text transcript of show #84

October 11, 2007

### Parallel Programming with .NET

Scott chats with Stephen Toub a Microsoft Developer working on new ways to make concurrency programming easier with .NET.

(Transcription services provided by [PWOP Productions](#))



#### Our Sponsors

 **telerik**  
*deliver more than expected*  
<http://www.telerik.com>

 **nsoftware**  
<http://www.nsoftware.com>

  
**NET DEVELOPER'S JOURNAL**  
<http://dotnet.sys-con.com>





**Lawrence Ryan:** From [hanselminutes.com](http://hanselminutes.com), it's Hanselminutes, a weekly discussion with web developer and technologist, Scott Hanselman, hosted by Carl Franklin. This is Lawrence Ryan, announcing show #84, recorded live Monday, October 8, 2007. Support for Hanselminutes is provided by Telerik RadControls, the most comprehensive suite of components for Windows Forms and ASP.NET web applications, online at [www.telerik.com](http://www.telerik.com), and by .NET Developers Journal, the worlds leading .NET developer magazine, online at [www.sys-con.com](http://www.sys-con.com). In this episode, Scott chats with Stephen Toub, a Microsoft developer working on new ways to make concurrency programming easier with .NET.

**Scott Hanselman:** Hi, this is Scott Hanselman and this is another episode of Hanselminutes and we're sitting here in Microsoft in Building 41 with Stephen Toub. How's it going Stephen?

**Stephen Toub:** It's going very well. How are you?

**Scott Hanselman:** Not too bad. What are you working on?

**Stephen Toub:** I am working on the Parallel Computing Platform Team at Microsoft and specifically on the Parallel Extensions to the .NET framework or PFX. This is sort of the grander name for a few different technologies, some of which your listeners may have already heard of, one of which is Parallel LINQ or PLINQ, one of which is the Task Parallel Library or TPL, and then there's the third piece we haven't really discussed much, but it's sort of rounding out this collection of libraries that enables developers to better take advantage of parallels and then concurrency in their applications.

**Scott Hanselman:** Wow. That was almost like a practice marketing speech. That was brilliant.

**Stephen Toub:** It wasn't. I'll have to write it down afterwards. It's recording.

**Scott Hanselman:** We'll record. I'll give you the transcript because that was really deep.

**Stephen Toub:** Thanks.

**Scott Hanselman:** I want to ask ignorant questions because this is a big topic.

**Stephen Toub:** Yeah, it's a huge topic.

**Scott Hanselman:** Yeah. By asking the ignorant questions, I want to kind of get my brain on exactly what's going on here. `System.Threading`, I know that's a silly thing to go and say -- you're talking about parallel extensions and concurrency and multi-`proc` and multi-`core` and I just said, "Hey, what about `System.Threading`?"

**Stephen Toub:** No, it's actually a fabulous question. In fact, we're trying to think through right now how to actually describe the relationship with `System.Threading` and whether we're actually part of `System.Threading` or not. At a core level, `System.Threading` is all about low level manipulation of work. There's a set of synchronization primitives in `System.Threading` and that includes, for example, `System.Threading.Monitor`, `System.Threading.ReaderWriterLock`, `System.Threading.ReaderWriterLockSlim`, and then a bunch of basically .NET wrappers around `win32` executor objects like `Mutex` and `Semaphore` and the like, `Event`, `Manual Reset Event`, `Auto Reset Event`, and `whatnot`. These are all relatively low level primitives that developers have a hard time working with them, rationalizing, and building higher level constructs around and we don't want forcibly to spend all their time kind of in the depths of the bowels of these programs. What they really want to do is to express what they want done, not specifically how they want it done and the bits and the bytes. So, what we're trying to do is provide those higher level constructs. For example, you've got `System.Threading.ThreadPool`, which allows you to say `ThreadPool.QueueUserWorkItem` and a few other kind of really advanced level scenarios, but some of the simplest things that people try and do with the `ThreadPool`, they have to write a lot of code to do it. For example, if you say `ThreadPool.QueueUserWorkItem` and then you want to wait on the thing that you queued, that's very difficult. You have to actually modify the delegate that you passed `QueueUserWorkItem` and then in that delegate you have to set some sort of event and outside you have to wait on that event to complete.

**Scott Hanselman:** But it's a really common pattern?

**Stephen Toub:** But it's a really common pattern. Similarly, we get a lot of requests from people to want to cancel work items that they've queued that may not have run yet, so one of the things that we're providing in this new `Task Parallel`



Library is the ability to get back a reference to that thing that you queued up and then be able to manipulate it in some way, so you can say `Task.Wait` or `Task.Cancel` and so forth. We're trying to take these abstractions to kind of the next level. Whether we're in `System.Threading` or in `System.Concurrency` or whatever namespace we happen to be in, our goal is that we're trying to take this beyond just threads and we're trying to say you're writing a parallel application. You're writing an application that takes advantage of concurrency, not you're writing an application that manipulates threads and waits on events and that sort of thing.

**Scott Hanselman:** Okay. So, let me try to put this into something concrete and I'll use an example from my own life. I was making a WinForms application that was going to talk to a blood sugar meter, a glucose meter for diabetics. It was going to talk to that and it was going to be a very long running thing, could have taken a couple of minutes, and I needed to have a progress bar update. I wanted also not just the progress bar updating, but I wanted to be informed when certain points, certain milestones had been reached within this kind of import process and I found that entire experience to be incredibly frustrating.

**Stephen Toub:** It's very difficult to do today.

**Scott Hanselman:** Because I wasn't clear if there was going to be just the main thread for the WinForm and then another thread for the getting of the data or whether that would be three or four threads, maybe one for the serial work, one for the more logical wrapper around talking to the actual serial port, and I have kind of get myself in quite a situation where basically I ended up just making the progress bar just go and spin, spin, spin. Doing this what I thought was a very simple thing, which was let me know how things are going, basically became a fire and forget operation and then when I eventually did come back and get information, I had to go and say `this.InvokeRequired` and make sure whether my button was being called under an appropriate thread or my progress bar is being actually sent to the appropriate thread. Are those the kinds of scenarios that people are bumping into that you're going to try to fix?

**Stephen Toub:** That's one set of scenarios that people are bumping into that we're going to try and fix.

**Scott Hanselman:** This is a much broader initiative. We're talking about fixing a much bigger problem.

**Stephen Toub:** We are. So, that's sort of the responsiveness, keeping your GUI thread from being blocked basically by background work that's happening, keeping your application responsive, that's sort of one big bucket. Another big bucket we're trying to solve is taking advantage of all of the processing power that's in your computer. In your typical desktop computer today, you've got maybe one, maybe two processors or cores, that fun Intel commercial where the guy is dancing and splits into two different people, the advertisement for their Dual Core. AMD obviously has Dual Cores as well. Next year, Quad Cores are going to be common and a couple of years after that...

**Scott Hanselman:** Yeah, I just built a Quad Core.

**Stephen Toub:** Exactly and it's a relatively high end today, but in the year that will be very affordable and two years after that 8-core and two years after that 16-core will be on everyone's desktop, Steve's predictions.

**Scott Hanselman:** Right. Is that the new thing now, a 16-core machine on everyone's desktop? That's the new mission statement?

**Stephen Toub:** That's my mission statement.

**Scott Hanselman:** Okay.

**Stephen Toub:** But most applications today just aren't written to take advantage of that even if the problem they're solving could. It's basically because developers aren't expressing their problems in a way that allows them to be parallelized and through no faults of their own, it's incredibly difficult to do. A lot of this stuff in PFX, in the Parallel Extensions of the .NET framework, be it PLINQ or TPL, are meant to address that particular problem of how do we scale whatever problem you're throwing at this technology, how do we scale it to all of the processors in your machine.

**Scott Hanselman:** How much of this is being done with -- forgive me if I use the incorrect language because I'm not a concurrency expert, but how much of this is being done from a programmatic perspective as opposed to a declarative perspective? I find that when I'm doing a lot of concurrency work, my head



starts to ache when I start thinking about it as a series of programmatic instructions, but if I just draw the tree and say, "Okay, we branch a part here, fork off here, and then we come back together at the end." That seems a very declarative thing. If I could just tell the system that, it would figure out the details.

**Stephen Toub:** Yeah, and in fact we're sort breaking PLINQ and TPL apart into describing it as declarative data parallelism and procedural or imperative data and task parallelism. With PLINQ, you basically get to say what data you want and how you want it processed. You don't even need to go to the level of branch here or join here. You say, "From this data there, join with that, selected that, wear that, group by this," and under the covers PLINQ figures out, "Oh, you have 16 processors. Here's how I'm going to divide up the work. Here's how I'm going to partition it. Here's how I'm going to process it and parallel. Here's how I'm going to merge it." All you have to do to achieve that is to add `.AsParallel` onto your query.

**Scott Hanselman:** Nice.

**Stephen Toub:** So, very, very declarative and very much expressing what you want done, not how specifically you want it done.

**Scott Hanselman:** A really common use case that I run into at my last job where I worked in financial services was we wanted to do one logical operation like "Get Accounts." It turns out that those accounts are spread across 26 different mainframes, so you might have 26 mainframes and you say you've got three different classes of mainframes, so you've got 10 threads going off to those guys and 10 off these and 6 off to that. They're all different. They need to join up, but I don't care who's back at 30 seconds whatever happen at that point. That's what we've got and then we would come back with an incomplete dataset because user responsive was as important as getting data.

**Stephen Toub:** Absolutely.

**Scott Hanselman:** So, we would come back with records that would say things like "I'm unable to contact" or "Incomplete" or whatever. It was an okay experience because 99% of the time they would be okay. Is that a common scenario where someone wants to do something in parallel, but at some point give up?

**Stephen Toub:** Absolutely and for a variety of reasons. Sometimes you want to do something in parallel because you are trying a bunch of different approaches and any one of them is good enough. Sometimes you want to do things in parallel because you actually care about all the responses, but in your scenario you don't care enough to keep the user waiting for too long.

**Scott Hanselman:** Yeah.

**Stephen Toub:** Those are all scenarios we're trying to support with the technology we're providing.

**Scott Hanselman:** Wow. How do you model that? I mean there's this bell curve of all the different things that are out there and that might be any order of dozens of common scenarios depending on the tallest part of the bell curve that you want to hit. How do you manage the edge cases then? I'm sure that you'll nail the 80% solution, but how do you give the 20% person enough rope to hang themselves with. They don't just drop back into basic constructs.

**Stephen Toub:** That's a wonderful question. The cheap way we're doing this is through different levels of obstruction, so at the very top level we're providing, for example, a parallel class and the parallel class just has a handful of static methods on it, `Parallel.For`, `Parallel.ForEach`, `Parallel.Do`, whatever and with the `Parallel.For`, this is basically a drop and replacement for your `For` loop, so instead of saying `For i = 0, well, i < n; i++`, you say `Parallel.For 0 n`, here's my bot. With anonymous methods and lambda expressions, you actually get to express it by changing literally just that top four lines of your code and the rest of your body becomes the delegate that's passed to this method. That parallel class is under the covers built on top of this lower level of obstruction, which is basically a set of classes for doing this lower level work similar to what you would do with the thread pool, so you can say `Task.Create` and you create this asynchronous body of work, you get back a reference to it that you can either wait on or you could cancel and you can wait on with varying degrees of timeouts and varying degrees of controls and knobs and you control the number of threads under which it's running. You can create isolated pools. One of the number one requests that we have for the thread pool is I don't want this work to be part of the entire application's worth asynchronous work. I want to create a pool over there that just has five threads and then it can keep it separate from everything else. We support that sort of thing. We're



really trying to do it by providing that parallel class that targets the 80% of the cases and then the Task and the future classes and the Task Manager and everything else that sort of supports everything else and that the 80% case can be build entirely on top off.

**Scott Hanselman:** That's hot.

**Stephen Toub:** Yeah. In fact, PLINQ is kind of another top level of obstruction built on top of all that stuff.

**Scott Hanselman:** We're sitting here also with Howard Dierking, the Editor-in-Chief of MSDN Magazine. He just gave me a funny look when I said that's hot because this is the kind of stuff that we really get off on here at Hanselminutes.

**Howard Dierking:** Paris Hilton.

**Scott Hanselman:** Yeah, the Paris Hilton reference there, that's really hot, parallelism and concurrency in the .NET framework. Who else but our listeners and you guys can geek out with this stuff? I don't think my son cares about this. Let me take the conversation a little bit different direction.

**Stephen Toub:** Sure.

**Scott Hanselman:** Help me understand when we say multi-core machines, if I have a single process that's running on a machine with four processors, is there agility? If I have multiple threads, are those threads jumping between? I don't quite understand how the hardware works. I always used to think it was one process gets pinned to a processor.

**Stephen Toub:** Only if you want it to. You'd actually have to express that either as the user in, for example, Task Manager or as a developing setting thread affinity and process affinity for particular processors, but by default, no. There is no affinity. If you have a process that uses only one thread, by default, that process will ever only be running on one processor at a time, but it can certainly move around from processor to processor. Ideally, it won't for a lot of subtle reasons like making sure that caches aren't invalidated and the memory that the -- well, that caches aren't invalidated is good enough, but if the process has in it multiple threads and let's say in your case it has four threads, each of those four threads can be running one on each processor at the same time.

**Scott Hanselman:** Who decides that?

**Stephen Toub:** For the most part, it's the operating system unless as a developer you go ahead and say, "This thread here, I really want to tie to processor 3," in which case you can set an affinity mask that says, "Never let the operating system only allow this thread to run on processor 3 and if processor 4 is available and there's absolutely nothing happening on it, sorry, you can't run this thread until processor 3 is available."

**Scott Hanselman:** Given the goals of what you've been talking about with these Parallel Extensions to the .NET framework, I shouldn't have to worry about that. An application that uses these extensions on a single proc machine will just simply run better on a multi-proc machine.

**Stephen Toub:** An application that uses these extensions on a single proc machine will run better ideally with as many processors as you can throw at it.

**Scott Hanselman:** Oh really?

**Stephen Toub:** So, the best applications we see are the ones that basically scale linearly in terms of performance with the number of processors. Now, that comes with a variety of caveats being that, for example, there's this law called Amdahl's Law that talks about the relative speedups you can expect in your application based on how much the application is serial and how much of it is parallel. So, if a very small portion of your application is parallel, even if we were to increase by 100X, the speed at which that parallel section runs, if it's still dominated by the serial portion, you're not going to overall see monstrous speedups. However, if that small portion of your application is working on data and over time you expect the amount of data your application is processing to grow exponentially, well, then the benefits of parallelizing that small portion is going to make a significant impact or theoretically could.

**Scott Hanselman:** That brings up in my mind two different questions. The first question is someone said that the optimal number of threads in any system is one; maybe that's the optimal number of threads in any processor is one.

**Stephen Toub:** It would make sense for with the optimal threads per processor.



**Scott Hanselman:** Okay.

**Stephen Toub:** Assuming that every thread is doing pure CPU.

**Scott Hanselman:** Right.

**Stephen Toub:** It's CPU-bound, it's doing pure computation, assuming you didn't care about a GUI that was responsible for whatnot...

**Scott Hanselman:** Sure.

**Stephen Toub:** There would be very little benefit to having more threads because then the operating system would have to contact switch them in and out of various processors, at which point each contact switch is expensive and it's expensive not just because the actual operations required to save the thread state from the registers to memory or disk or whatnot, it's also expensive in that there was some state that the thread was using in the cache, in the L1 cache, in the L2 cache, whatever, and if the thread is transitioned out and another thread comes in, it's going to start writing over those caches with data that it cares about and then when the original thread comes back in, that's going to have to read from memory rather than cache all the stuff. So, contact switches are expensive not just because of the actual cost, but because of the latent effects of them.

**Scott Hanselman:** So, certainly, that massive parallelism on a system that's hardware that didn't support massive parallelism would serve very little purpose?

**Stephen Toub:** It would and one of the things we're trying to achieve with these classes is we try and make sure that we scale the work according to the number of processors you have. So, at the Parallel.For, if you have a single CPU, we're going to basically do a For loop. We're not going to spin up a bunch of threads. If you have 16 processors, well, in that case, we'll nicely share the work amongst the processors.

**Scott Hanselman:** Are these library decisions? Are these managed code decisions?

**Stephen Toub:** These are managed code decisions.

**Scott Hanselman:** Cool. Cool. My second question was if I'm imagining kind of a typical web

application scenario where one client hits a website and when in the past that single thread would then go to an ASP.NET server, which would then call to a database, maybe I started doing parallelism and then one person hitting my web server now ends up being four connections to my database and that becomes a standard thing because I started being very familiar with these technologies, it seems to be like that would dramatically change the scale signature of the entire application and would potentially get me then disk-bound quicker. If I'm ultimately not going to become CPU-bound as much, I'm going to become disk-bound, but people are saying already that machines are fast and then CPUs are barely working and then ultimately we are IO-bound. So, what does this kind of parallelism mean when it comes to IO-intensive operations?

**Stephen Toub:** That's sort of two different questions and I'd like to address the first one, which is basically, what does this parallelism stuff have to do with the server? In general, server apps today like ASP.NET, they already have enough parallelism to satisfy the CPUs. So, you take a typical ASP.NET application, it's getting thousands of requests per second, if each of those requests can be isolated, if it's not accessing shared state or anything like that, you're already introducing a thousand different asynchronous pieces of work. So, unless you're expecting very few requests into your web server and unless each of those very few requests is doing a ton of computation work, using parallel class, for example, in your server application might not buy you all that much, which ties into your second question. Because, like you say, a lot of stuff is IO-bound in server world and so you take something like asynchronous pages in ASP.NET, it's basically meant to limit the number of thread pool resources you're consuming and maximize your throughput while at the same time basically making sure that you're not blocking other people from requesting your server resources just because you're waiting for the database to come back or you're waiting for a web service call to return or something like that. So, a lot of the technologies that we're working on right now, we're focusing largely at desktop or at backend data processing, not so much at the web server world because for the most part your front-end web applications have enough concurrency to go around.

**Scott Hanselman:** Interesting. I had my brain in kind of the middle part and you're hitting the front bumper and the back bumper. In the front, we always look at things like Outlook is a really good example of



a multi-threaded application. There's always stuff going on in the background in Outlook. On the backend, if you were doing some very computationally intensive batch processing, if you were doing protein folding or something like that, that would be a time to do that.

**Stephen Toub:** Absolutely.

**Scott Hanselman:** We used to do a lot of batch payment type stuff that we're trying to spin through millions and millions of bill payments on the backend.

**Stephen Toub:** Yup, exactly.

**Scott Hanselman:** That's very interesting. And all of this would be done within a single process? Not of this is parallelism through forking processes? Because a Windows process is so much heavier than a Unix process.

**Stephen Toub:** Right.

**Scott Hanselman:** You can't just fork willy-nilly.

**Stephen Toub:** You could, but it would be probably a bad idea.

**Scott Hanselman:** Yeah, yeah.

**Stephen Toub:** Yeah. A lot of stuff we're working on right now was within a single process, but with the goal that the resources are managed across the entire machine. For example, the types that we have underlying all of the stuff in PFX, we have this work-stealing scheduler. Let's say I was processing some image. I was doing a ray tracer, for example, and I had sort of a straightforward solution of how to parallelize my ray tracer processing because every pixel can be treated independently. I'm just shooting a ray of light basically from a particular pixel seeing what object it hits and comes back with a color. If I had two processors, I could say one processor is going to do the top half of the image and another processor is going to do the bottom half of the image or I have four process, I could space into four and be good to go, but this is actually going to take longer than some other methods because, for example, in the typical ray traced image, the top might be sky or might be all black. It might have no objects that hit and there's less work being spent on each of those rays at the top than, for example, on the bottom where there's floor that's reflecting a bunch of objects and whatnot. So, what a work-stealing scheduler

does is it basically assumes that different areas of what your processing are going to take more or less time than other areas and so it sort of dynamically manages itself and is able to steal work from other processors and say, "Oh, I see you've got a backlog. Let me take some of that off your case." The same thing can be done based on CPU utilization and everything else going on in the machine if other things happenings on the machine are causing a particular processor to run slower and that's not to be able to process work as fast as some other processor, those processors can steal work from that original thread and there's better balanced workloads across the machine.

**Scott Hanselman:** Very cool. That now forks -- it's interesting. We've forked a number of times here. I've split it into two questions there again, the first one being that it got me thinking about work partitioning. If I had an image that was that size, it sounds like I'm not going to have to think about how many processors and, therefore, how I'm going to chop this work up. I simply just kind of unleash the parallelism of this obstruction layer on that bit of work and what it's done, it's done.

**Stephen Toub:** Exactly. You say `Parallel.For, = 0; x < width; x++` and then you have your inner `For` loop for the height and whatnot. Now, we say that you don't have to think about it, but in reality you still do because there are still lots of problems that developers are going to have to think about when introducing concurrency into their applications. For example, the ray tracer example is a good one because every pixel is independent of every other pixel and that's what we refer to as an embarrassingly parallel application because you don't have to think about what's happening in the body of the loop, but if, for example, one of those pixels depended on one of the other pixels unless there was a sort of loop-carried dependencies and enforced ordering that needed to happen, we're now doing these things in parallel so you have to be very, very careful. You have to manage that shared state yourself. The third portion of the PFX technology is a set of data structures and coordination types that we hope will make that sort of thing easier, but it's still something that people will need to think about. There are other deeming parallelism blockers that people wanted to be aware of. Obviously, manipulation of shared stated and mutability is one of them, but there are other things, for example, let's say you are writing a Windows Forms application, you wanted to loop over a huge number of controls and modify something on



each of the controls, do some computation to come up with a new name for the control or something and you decide to go ahead and parallelize that because the loop is taking a few seconds and you'd love for it to take a second or less, but in Windows Forms and most of GUI programming on Windows, you have this STA concepts, single-threaded apartment, and controls can't be modified by threads that didn't create the control.

**Scott Hanselman:** Right.

**Stephen Toub:** So, as soon as you say `Parallel.For`, the loop over all these controls now background threads are going to be executing the body of this loop. They're going to be modifying this foreground controls and you're going to start getting exceptions all over the place.

**Scott Hanselman:** And that's where I have to go and basically I found myself in there. Before I touched the control, I have to ask myself the one thread safe property on the control, "Is it cool to be here?"

**Stephen Toub:** Exactly, `InvokeRequired`?

**Scott Hanselman:** And if it's not, basically it's like come back later and come through the back door.

**Stephen Toub:** Exactly. So, there are certainly cases where the stuff isn't going to be as applicable or as appropriate or additional thought will need to be added to figure out what exactly to do. There are other scenarios. One of the big ones we run into is exception handling and what does it mean for a processor, some operation to fail. If you were, for example, multiplying every value in an array by some number and computing something and that computation may possibly throw an exception and let's say it may possibly throw an exception on the third and the sixth item, well if you're processing this thing serially, you go 0, 1, 2, you get to 3 and it throws an exception in 4, 5, 6, 7, etc., they haven't been modified, but in a concurrent world if you had 8 processors, for example, 0 through 7 might all be processed at the same time and thus you might actually even if 3 throws an exception and 6 throws an exception 4 and 5 may have already been modified. Similarly, now, rather than just getting this one exception back, now you're getting the set of exceptions back, which may not have been expected if you are coding your code from an inherently sequential world.

**Scott Hanselman:** How do you handle those? I mean do you collect them all up and then make a decision about whether the whole thing was a failure or not?

**Stephen Toub:** That's our current plan.

**Scott Hanselman:** And then you get into the concept of a transaction. Can you roll back to those things? Can you wrap all this in `systems.transactions` code?

**Stephen Toub:** Transactional memory is a whole another topic.

**Scott Hanselman:** Okay. My second question from our previous forking about 10 minutes ago was what about parallelism across machines. There's always this notion of build farms and render farms and these are the common things. This is an ignorant question, but do they call these Beowulf clusters?

**Stephen Toub:** These are clusters in general. Beowulf clusters are specific to Linux.

**Scott Hanselman:** Okay.

**Stephen Toub:** But yeah.

**Scott Hanselman:** But that concept of like "hey, well, these boxes aren't doing anything." I mean I've got four or five machines at home that aren't working too hard."

**Stephen Toub:** Yup.

**Scott Hanselman:** Is it conceivable that this kind of concept could be extended kind of like outwards?

**Stephen Toub:** Absolutely. There are various teams at Microsoft working to address that. One of the biggest ones at the moment is the HPC Team, which is doing the compute cluster server, which allows you to do exactly that. You have some task that you want split across a whole bunch of machines and you would install basically compute cluster server. It is the equivalent of the Beowulf clusters you were mentioning, but for Windows basically.

**Scott Hanselman:** My last question about PLINQ specifically even though as we've mentioned before PLINQ is just kind of a member of a larger family of parallel technologies that we're talking about, it is truly



a LINQ-based thing? I mean is it based on the basic ideas that were put forth when LINQ was introduced?

**Stephen Toub:** PLINQ, our Parallel LINQ, which is the current codename for it is a parallel implementation of the .NET standard query operators. LINQ to objects is basically an implementation of a bunch of methods on the Enumerable class, Enumerable.Select, Enumerable.Where, Enumerable.Range, whatever. PLINQ introduces parallel Enumerable, so you have ParallelEnumerable.Select, ParallelEnumerable.Where, ParallelEnumerable.range, etc., and it implements all those using kind of all aspects of parallel implementations under the covers. When you compile your application in say C# or VB and you're using, for example, the query comprehensions where in C# you say var q = from p and p and people where p.name = Steve select p. Under the covers, the C# compiler compiles that down to invocations of these static methods on the Enumerable class. Because of how LINQ works and how the binding works, it searches for these extension methods and it finds them in the only place that is currently there, which is System.Linq and the System.Linq.Enumerable class, but if you say .AsParallel, that basically causes the C# compiler to also find the PLINQ methods and to bind to those tighter than to the Enumerable ones and so if you were to look at Reflector, for example, in .NET Reflector, the compiled code, you would see that rather than invoking Enumerable.Select, Enumerable.Where, it was invoking ParallelEnumerable.Select and ParallelEnumerable.Where. The semantics of these things are almost identical with the exception being where we really can't make them identical or where we by default don't make them identical because of performance reasons. So, for example, with PLINQ, our current thinking is that by default we won't preserve ordering.

**Scott Hanselman:** You have to sort the results at the end because there are no guarantees on how things are going to come back.

**Stephen Toub:** Yeah. So, if you were to go with the default, that's true. We allow you to turn on orderpreservation and it's possible that by the time we actually ship the technology, we'll decide that it's in the best interest of everyone to enable order preservation by default because too many people would get hung up on it not being there.

**Scott Hanselman:** Sure, and you and I, we're just talking. I mean to be clear to the listeners, as with typical Hanselminutes, this is just kind of a scoop.

**Stephen Toub:** Exactly.

**Scott Hanselman:** No warranty expressed or implied...

**Stephen Toub:** Thank you.

**Scott Hanselman:** In this particular podcast, which brings me to my last forked question of the evening, which was you were talking about I Parallel Enumerable. Does that mean that I'm not going to be able to get free kind of parallelism on data structures like XElement and things? Are they going to have to support parallelism inherently?

**Stephen Toub:** No, they will not have to support parallelism implicitly. Basically, PLINQ works with anything that implements Enumerable of T.

**Scott Hanselman:** Wow. How's it going to keep them from hurting themselves? I'm always enumerating through collections and getting in trouble because someone got mad halfway through because they modified their collection.

**Stephen Toub:** When you're working with IEnumerable of T, if we don't know anything more specific about it, we have to internally lock as we're accessing elements from the collection, but we try and do it in an efficient way, for example, batching a request from it, so we might pull 128 elements from it and then allow another thread to come in and pull 128 elements from it and so forth.

**Scott Hanselman:** So, you kind of do like a checkout/check-in process. That's really creative because when I first heard that, I was like, "Oh no. I'm gonna have to go and implement something to make my classes parallel."

**Stephen Toub:** Now, we work with any data structures that implement IEnumerable, so that will include LINQ to XML.

**Scott Hanselman:** That's fantastic. My last question was when you say AsParallel at the end of the C# compiler I think you used the words "binds more tightly," it kind of puts parallel on top because we know when we put in mix-ins and we put in



extension methods, it's about kind of who gets there first. Did that involved a change in the C# compiler?

**Stephen Toub:** No.

**Scott Hanselman:** This is all playing with how LINQ was initially designed?

**Stephen Toub:** Exactly.

**Scott Hanselman:** This is really cool. Wow.

**Stephen Toub:** Great.

**Scott Hanselman:** Yeah. Gosh. I have to drink this in. This is pretty exciting. I really appreciate you taking the time out of your day to talk to us, Stephen.

**Stephen Toub:** Oh, it's my pleasure. Thanks for coming.

**Scott Hanselman:** And this has been another episode of Hanselminutes. We'll see you again next week.