Microsoft

.NET for Java Developers

Ted Neward, Principal
Neward & Associates, LLC

EDITION 1.0

DOWNLOAD available at: https://aka.ms/dotnet-forjavadevs

PUBLISHED BY

DevDiv, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way
Redmond, Washington 98052-6399

.NET for Java Developers

Prepared for Microsoft by Ted Neward, Principal, Neward & Associates, LLC

Contents

Introduction

When I was about eight years old, my parents took my sister and me to Paris, France. It was our first trip out of the country, and aside from the ridiculously long flight—eight hours on an airplane feels like forever to somebody who hasn't yet finished their first decade of life—I really had no idea of what to expect. In fact, in an amusing anecdote my father has yet to tire of telling, I remember having to sit down with my parents in front of a globe so that they could explain to me that "the United States" was not, in fact, synonymous with "the world."

It was entirely a trip of firsts: my first experience wandering around in a world where I couldn't understand any of the written signs, my first experience eating food I couldn't recognize, and my first experience feeling completely lost in the world around me. I found, to my eight-year-old horror, that I couldn't even figure out which bathroom to use—which, considering I had just the year before managed to convince my mother I didn't need to be escorted to the bathroom and back, was of particular concern. I didn't want her to start thinking I actually needed adult supervision. This was clearly not my home, and I was more than a little terrified of it.

Visiting a foreign land, particularly to someone who has cocooned themselves entirely in the concepts and cultures of "home," can be intimidating and overwhelming without a guide. On that trip to France I had my parents, and despite neither of them speaking anything close to fluent French, they managed to guide us through the tricky parts. (French-English guidebooks are a wonderful thing.)

For the typical Java developer, finding a guide to exploring the .NET ecosystem can be tricky, since the average Java developer can still hearken back to a time when expressing an interest in looking at .NET was tantamount to declaring a desire to betray everything good and right in the world (or so it seemed, at least at the time).

In this paper, I will serve as your guide to the .NET world. I've been a part of the .NET ecosystem almost as long as I've been a part of Java's. I started with Java in 1996; I started with what would become .NET in 2000, even before it was announced. I've built applications in both, written books for both, and explored the source for both. They're closer to one another than you might think, but each has been fundamentally influenced by the people using them. In fact, it's not a bad analogy to think of them as twin brothers, separated at birth, raised by different sets of parents. Same DNA, but shaped by their experiences and events to be different in a number of interesting places.

If you're a Java developer, looking to get started exploring the .NET ecosystem, then suspend what you think you know, take my hand, and let's take a trip to a foreign land. Doing so will require a little history to understand how things came to be, then a quick set of instructions on how to get the tooling set up (and which tooling you want, depending on your preferences). Once the environment is set up, we can pick up the critical differences between C# and Java, and start diving into some of the central frameworks in this world—ASP.NET, Entity Framework, and a few others. When we

finish up, if you're following along, you'll have everything you need to start your own exploration of the .NET community, and from there you'll get to draw your own conclusions and make your own opinions.

Most importantly, you'll know how to order off the menu and tell the bathrooms apart.

Ready?

History

If you'd like to skip the walk down memory lane, you can jump to the next section, Getting Started.

Ostensibly, the history of .NET begins in October of 2000, at Microsoft's Professional Developer Conference (PDC), when Microsoft announced to the world that they were working on a new programming platform, on which would rest the fate of all Microsoft development to follow. It was called .NET, it would be the backbone for their upcoming operating system (Windows Vista), it would be a part of their data management system (nicknamed "SQLServer.NET"), and it would form the core development platform for building interconnected applications into the future by leveraging emerging standards such as XML, SOAP, WSDL, and a few others the world had not yet really come to understand. It would be the principal competitor platform to their principal development rival at the time, Sun Microsystems, and its Java platform. And the world could see that the shot was fired, the line in the sand drawn, and the armies drawn up; clearly, this was Microsoft copying the efforts of its competitors, seeking dominance, and….

And most of the world completely misunderstood where .NET had come from, why it had come to exist, and what Microsoft really sought to obtain from it.

In fact, it's fair to suggest that many within the Microsoft ecosystem misunderstood what .NET was really about; once announced, the marketing machine took over. Much effort was wasted due to the entanglement of ".NET" with "web services," and it would take the better part of a decade before people could understand that the two really had nothing intrinsically to do with one another any more than "Perl" did with "databases." Microsoft management compounded all this confusion by re-branding the ".NET" suffix to a number of their products, which they later corrected but well after the damage had been done.

But despite all that, solid engineering rests at the core of .NET; all that branding and marketing made it harder to see, but it's always been there. To really understand .NET, it is necessary to see it in its proper historical context: that, in truth, it was the simple "next step" along an evolutionary chain that stretched back much further than that October keynote almost two decades ago.

Of components, managed environments, and code

In the beginning, there was C.

To be more accurate about it, in the beginning a number of programming languages were created to make it easier to produce programs that knew how to execute on a single chip. They integrated directly with the hardware, making use of the facilities there to obtain memory, interact with the disk, and present data to the display for the user to understand. In time, we as an industry found that this direct-hardware interaction had its drawbacks, and we sought to create a series of abstractions on top of that hardware, to help coordinate the use of the hardware across multiple simultaneously

executing programs. We called that first attempt at regulating and managing the hardware an operating system. And it was pretty good. So good, in fact, that numerous flavors of operating system were born, including Microsoft's Windows operating system, and Solaris, and macOS (though it was called "System 7" back then), and Linux, and a few others whose names exist now only in the history books and dusty corners of Usenet forums. (Bonus points if you recognize these names: Minix, Mach, BSD, OS/2, Amiga, or BeOS. Even more bonus points if you ever ran one of them. If you ever ran DOS, though, it just means you're old.)

The bigger, more important issue, however, lay with the underlying concept of the operating system itself—that programmers found it helpful and useful to have "something else" working with their programs. That "something else" would be responsible for certain aspects of how the programs executed and behaved, rather than assuming that every single nuance had to be the programmers' responsibility.

This concept, in large, is essentially that of a "managed environment." Although it has its roots in operating systems, it echoes all the way down through the ages, even into the modern era of containers and cloud: that there can be, within a program, a set of services and/or actors that are responsible for details that the programmer doesn't want to have to deal with. Operating systems are managed environments in that they take care of how to position data on the hard drive, how to schedule the execution of threads, and how to regulate access to various resources such as network ports or data files. Virtual machines such as the Java Virtual Machine are managed environments in that they regulate the access and reclamation of heap memory, transform an intermediate bytecode format into CPU-native instructions, and prevent the direct access of memory (except under very specific circumstances) in order to avoid process termination by careless programming. Containers such as Docker are managed environments in that they provide a context around an executable to mimic a precise environment—configuration files, environment variables, and shared network settings—so that developers can be assured that the environment outside their programs are consistent regardless of where they are deployed.

By the early '90s, Microsoft had come to understand the need for a managed environment for Windows developers to be able to take "the next step forward" in development evolution. The "GUI revolution" of the mid-'80s was largely finished; programs weren't just about "dialogs and data," but now needed to integrate more deeply with the GUI environment (right-click context menus, for example) and start taking advantage of networking in greater detail. (Old developers like me will remember that it was around 1995 that references to HTTP and "the World Wide Web" were just starting to appear in publications and technology journals.)

As is common when developers master a new concept, new challenges emerged. C++ developers had been speaking of "reusable code" for several years, but it was Visual Basic (and Delphi, along with several other "4GL" variants) that demonstrated actual practical code reuse, in the form of "controls": widgets that could be used across a variety of programs repeatedly without requiring in-depth knowledge about the controls' implementation. Simply obtain the library, put it into the right place within the development environment, drag-and-drop the control into the construction area, and lo, the developer was off and running. True, C++ had created the concept of "application frameworks," like MFC and OWL, but "GUI builders" had demonstrated that the concept of "controls" could be leveraged to create a marketplace of reusable atomic blocks of state, behavior, and events; now

Microsoft wanted to bring that concept, writ larger, into the bigger world beyond GUI development. More importantly, they also wanted to invert that relationship—make it possible for outside developers to build "things" that could be used by existing things (such as the operating system or, perhaps, the web browser) without detailed knowledge of their implementation.

To do that, they needed to create a managed environment that would work across a variety of different languages, but all operating under the same rules. That effort, called the Component Object Model (COM), was the first of several "managed platforms" that emerged during the early- and mid-'90s. (Ironically, several open-source efforts would go after this same space many years after Microsoft declared COM "finished," including Mozilla's XPCOM.) COM itself was successful in that it became the backbone of Windows development for a significant percentage of the Windows platform. But it still failed in that it left programs vulnerable to many of the same things that had plagued them before: memory leaks, mishandled pointers, and unexpected process termination at the first mistake. It was clear that, despite the managed platform, writing code in a "native" language (such as C/C++ or Delphi) still wasn't a particularly great place to be.

It was around this same timeframe, 1995, that Sun Microsystems demonstrated the HotJava browser to the world, the centerpiece of which was this language that could be used to write executable bits to run inside the browser. By 1996, Microsoft had come to the realization that this was what it was missing: a "managed language" to run on top of their managed platform. A managed language had exactly the right "distance" from the underlying operating system they were seeking—garbage collection would help prevent memory leaks and accidental crashes, and the virtual machine could help enforce stronger security controls that the world had just started to realize they needed. It was, in many ways, a perfect match. Microsoft invested resources into building their own JVM, extended their IDE to include writing Java code (Visual J++), and integrated COM into the runtime.

Unfortunately, in the last part of that sentence lay the seeds of its own destruction.

Hello, .NET

Divorces are never a pretty sight, and recounting the history of one is always fraught with opinion, revisionist history, and bad feeling. Suffice it to say that by 1997/1998, two facts were becoming clear: one, Sun didn't care for Microsoft's work increasing interoperability between their JVM and native COM components in Windows[1], and two, Microsoft and Java were going to part ways, one way or another. (Who got custody of the dog is still hotly debated.)

Either certain groups within Microsoft could see the writing on the wall or they had come to find issues (real or perceived) with the Java Virtual Machine; either way, by 1998, a new effort came to life inside of Microsoft, designed to bring a new managed platform and managed language into the world. Originally intended as the successor to COM, it went through a variety of names, including

[1] For the curious, Microsoft offered additional platform-specific APIs that permitted interoperation with native COM components. The irony of this is deep, considering that Sun had actually allowed for COM-specific native interfaces in their JNI specification—the first three methods in the interface table for a JNIEnv ("JNI environment") object were specifically left "reserved" for the necessary three methods (AddRef, Release, and QueryInterface) of every COM interface object. Those three slots are still reserved to this day.

"COM+," the "Component Object Runtime" (COR), the "Universal RunTime" (URT), and "Next-Generation Windows Services" (NGWS) before finally being introduced to the world as ".NET" at that October 2000 PDC. Along with it would come a new language, code-named "COOL" (supposedly to stand for the "Component-Oriented Object Language," although opinions vary now) and later renamed "C#."

While most observers considered C# and the CLR (the Common Language Runtime, the virtual machine executing the bytecode) to be direct competitors of Java, Microsoft took careful steps to ensure that it was "open" in ways that Java was not: where the JVM was designed from the beginning to be a virtual machine for a single language, the CLR embraced a more polyglot approach, with Microsoft even going so far as to recruit several universities and companies to create new languages for this platform and/or adapt existing ones. This project, nicknamed "Project 7" (for the 7 groups that participated), debuted several of these languages during .NET's introduction, and included CLR flavors of Eiffel, Perl, and even COBOL(!). Few of these languages gained any traction after the first few years of existence, and most no longer exist, but their presence had the effect on Microsoft of "keeping them honest" with respect to other languages beyond C# and Visual Basic as the platform grew and evolved. (Microsoft itself would later go on to introduce a number of new CLR-based languages, including a compiled-to-binary version of ECMAScript, two different efforts at bringing managed coding to C++, a hybrid object/functional language called F#, and several more research languages.)

However, Microsoft also sought to make sure that .NET was not a proprietary effort by investing thousands of man-hours into the creation of a series of international specifications—through the European Computer Manufacturer's Association (ECMA[2])—to ensure that C# and the underlying platform were both open industry standards. C# was accepted and ratified as ECMA-334, and the underlying runtime was given the name the "Common Language Infrastructure" (CLI), and accepted and ratified as ECMA-335. Intel, HP, and several other industry participants joined Microsoft in keeping these specifications up to date with changes to the language and the platform, the most recent updates[3] coming in 2012 for the CLI and 2017 for C#.

With the introduction of these specifications in 2001 came an unexpected side-effect: members of the open-source community began to examine building a "clean-room" implementation of them. While the CLI Specification didn't include any of the Windows-specific features (such as a deep COM integration) Microsoft wanted for backwards-compatibility and forwards-evolution for their existing developer base, the CLI Specification itself provided enough integration points to make it interesting to implement on other platforms. Miguel de Icaza, of GNOME fame, began one such project, calling it Mono.

The specification also gave Microsoft some room to do what some within the company had been looking to do for quite a while: provide a research and experimentation vehicle for the academic and research communities to explore. In 2002, Microsoft released a stripped-down version of the CLR source code, code-named "Rotor," distributed under an open-source license they created, called the

[2] ECMA, it should be noted, also owns the specification for the language commonly known as "JavaScript," but is more rightly called "ECMAScript," governed as ECMA-262.
[3] CLI: https://www.iso.org/standard/58046.html and http://www.ecma-international.org/publications/standards/Ecma-335.htm; C#: http://www.ecma-international.org/publications/standards/Ecma-334.htm

"Shared Source" license, giving it the formal name "Shared Source CLI" or "SSCLI." While Mono would remain the subject of only fleeting interest to some for many years, and Rotor of even less interest to those outside the .NET universe, the seeds of an open-source mindset were sown. Over time, Mono would be purchased by Novell, then spun out into its own company, Xamarin, which would faithfully parallel the work described by the specifications and periodically try out new ideas for the open-source community to consider.

Microsoft officially released the first version of C# and the CLR in 2002, and new versions came roughly three years apart after that: C# 2.0 in 2005, and C# 3.0 in 2008. Each was contained in a product release of Visual Studio, such that C# 2.0 and version 2.0 of the CLR runtime was released with Visual Studio 2005, and C# 3.0 and CLR 3.5 came with Visual Studio 2008. Microsoft then shifted to a two-year release cycle, with Visual Studio 2010, Visual Studio 2012, Visual Studio 2015, and most recently, Visual Studio 2017. However, the versions of C# and the CLR started falling out of lockstep with Visual Studio, deliberately, for a couple of reasons.

The first was that Microsoft had again seen the writing on the wall, so to speak, and was starting to unbundle its various parts of the .NET ecosystem into constituent, and independent, parts. F#, which first shipped "out of the box" in Visual Studio 2010, was the first of these to start down the open-source path, largely because having started life as a Microsoft Research project, it had never really embraced being commercialized (and therefore closed-source) in the first place. Microsoft started working shortly thereafter on an open-source rewrite of its entire compiler toolchain, code-named "Roslyn." By the time of Visual Studio 2015, many parts of the core ecosystem were starting to ship independently, with Visual Studio releases serving only as a vehicle for developers to gather the parts of the ecosystem they desired more easily. Open-source package management (called NuGet), which began life well outside of the Visual Studio organization, was incorporated into Visual Studio so that NuGet came "out of the box," much as support for Maven and Ant do in Java IDEs.

Mono, meanwhile, had started to gain some traction in an unusual place: gaming. In particular, the gaming toolkit Unity had expressed interest in Mono, and soon had the Mono framework running inside the Unity toolkit. Unity-based games could now be written in JavaScript (a distant fork of the JavaScript-on-.NET implementation that Microsoft had done back in 2001), C#, or even a Python-inspired .NET language called Boo. Additionally, the open-source community produced another .NET implementation, the .NET Micro Framework, specifically aimed at embedded software systems and small-scale hardware such as the Arduino platform.

All of this, however, was the calm before the storm. In 2008, Microsoft announced that the source for ASP.NET (the servlet/JSP equivalent) would be accessible to anyone who wanted to look at it. It wasn't full open-source yet, though, in that they weren't accepting pull requests. It was, however, an important step, and in 2014 Microsoft took that final step: It announced that it would completely open-source all of .NET, hosting it on GitHub, and thus making it available to any platform and accept pull requests from any developer so inclined. A new .NET implementation would be completely open-source and hosted on GitHub, called .NET Core. It also announced the creation of the .NET Foundation, an industry consortium of groups dedicated to the growth of the .NET platform, in

much the same fashion that the Java Community Process governs the continued growth of the Java platform[4].

To be sure, Microsoft made some missteps along the way, and definitely learned a few hard lessons about its relationship with the open-source community, but by late 2017, Microsoft was statistically the biggest contributor on GitHub (measured in terms of total numbers of contributors[5]), and all of its key components—language, runtime, libraries, and many of its tools—were available for download on GitHub.

Microsoft also started to get more serious about the cross-platform/portability aspects of the .NET ecosystem. Xamarin, which by 2009 had started producing tools that could cross-compile .NET bytecode into native applications for iOS and Android platforms (the catchy-named "MonoDroid" and less-catchy "Mono for iOS" toolkits), was flourishing, and Microsoft realized that a practical story for being able to share non-UI portions of code across multiple platforms was necessary if they wanted to avoid developer confusion.

While the ECMA Specifications had discussed different "profiles" for levels of .NET standardization, these profiles were in many ways similar to their "Java Standard Edition," "Java Micro Edition," and "Java Enterprise Edition" cousins—a little unwieldy in a number of ways, but most notably in that there wasn't really a good way to guarantee compatibility across platforms and implementation versions—if code is compiled by the Java6 compiler, it is safe to assume that it will continue to run on the Java6 virtual machine from whence it was compiled, but the Java APIs do change from release to release, and this has tripped up the unwary Java developer from time to time.

Thus, in 2016, Microsoft announced the ".NET Standard 1.0" profile, a collection not of relatively vague implementation documents, but an actual API standard, such that a developer who sought to build a library that could be used by a variety of different platforms (across a certain number of versions of that platform, even) could know that anything that implemented .NET Standard 1.0 would be able to consume that library, regardless of where that code is running—on a web server, on a mobile device, or even on somebody's VR helmet (running a 3D interactive application written with Unity).

Which brings us, now, to the current day.

---

[4] Ironically, Microsoft's move to the .NET Foundation comes at the same time Oracle is looking to move away from the JCP; both companies have experience with standards organizations, and neither wants to see innovation stifled in committee.

[5] http://www.businessinsider.com/microsoft-github-open-source-2016-9

Hello, strange new world

To the Java developer, this is a strange, strange world compared to the one they glimpsed into back in 2000.

For starters, the Java world looks wildly different than it once did. Sun no longer exists, acquired by Oracle. Oracle and Google squabble in the courtroom over the legal implications of Java on Android devices. Spring was acquired, then spun out into its own company, Pivotal. JBoss absorbed into Red Hat. IBM no longer really even tries to push Java much anymore. Does anybody even remember EJB? Or that it was once "a thing"? Who would've guessed any of this back in 2001?

But meanwhile, over in the Microsoft world, a new CEO runs the firm, and he has made it clear that Microsoft is an open-source player. A new player, to be sure, and still finding its feet in a variety of ways, but one committed to the open-source community. In addition to the strides it has made with .NET, Microsoft has opened Azure up to almost every mainstream language and platform—including Java. (Yes, you can run Tomcat and MySQL in an Azure cloud cluster.) SQL Server has been ported to Linux. Windows Server can be loaded into a Docker container, and Microsoft works with the Docker community to bring Docker-on-Windows up to the same level of sublime that it has on other platforms.

Within the .NET ecosystem, the .NET Standard, now at a 2.0 release, provides an umbrella of guaranteed compatibility across three major platforms of this writing: the Windows-based .NET Framework (which contains the necessary libraries for building Windows client applications, for example), .NET Core (intended for the cross-platform server world, including Docker containers and cloud-native targets), and Xamarin (principally aimed at the iOS and Android world, but also providing bindings for building macOS applications). Unity has joined the .NET Foundation and expects to be aligned with the various versions of .NET Standard by mid-2018.

All of which means that it's not a bad time to start exploring this world in more depth.

Getting Started: Tooling

Getting started with .NET depends pretty heavily on your choice of operating system and, to a lesser degree, the development experience you want. As mentioned earlier, Visual Studio supports both Windows and macOS, as well as a variety of Linux distributions, which pretty much covers everybody. In this section, the goal is to get tools installed, a basic introduction to the major project artifacts, and, of course, build the traditional "HelloWorld." Once we've done that, we'll talk a little bit about what we're producing, and the package system for .NET (called NuGet).

Hello, web?

If you're just looking to flirt a little with the C# language but don't want to commit any disk space to the idea, one of the easiest ways to explore the language is the online interactive browser-based tutorial found at https://aka.ms/csharpquickstarts.

However, doing so will run into a whole slew of limitations to your exploration, so sooner or later you're probably going to want to commit some disk space.

Docker all the things!

If you're a Docker enthusiast, and you want to get the quickest path possible to working with .NET, fire up a Microsoft-maintained .NET Core Docker container by running:

```
docker run -it microsoft/dotnet bash
```

This will pull the latest version of Microsoft's Docker Hub–stored Docker images, and drop you into a shell prompt from which you can run the "dotnet" CLI tool. When you are ready to edit some files, run the Docker image with a volume mount to a host's local directory, and use your favorite editor—might we suggest Visual Studio Code?—to edit the .NET Core–generated files we're about to create.

Linux .NET?

As with most things Linux, installation instructions depend on which flavor of Linux you're running.

Red Hat is the easiest install, since Microsoft and Red Hat have partnered on a few things in the past, so the .NET Core packages are in the Red Hat distribution channels; assuming you are using the Red Hat Subscription Manager, two commands get .NET Core on your machine:

```
yum install rh-dotnet20 -y
scl enable rh-dotnet20 bash
```

At that point, the "dotnet" tool should be in your PATH and good to go.

If you're on Ubuntu, there's a bit more work required. While Microsoft is working to get the .NET Core packages into the standard Ubuntu distribution channels, at the time of this writing that hasn't happened completely, so Ubuntu users will need to add Microsoft's key and Ubuntu-release-specific URL into the list of sources the Ubuntu apt-get utilities check for packages. Full details are on the .NET Core website at www.dot.net/core; once that's done, however, apt-get can install the "dotnet-sdk-2.1.101" package onto the system, and this will get the "dotnet" tool in the PATH and good to go.

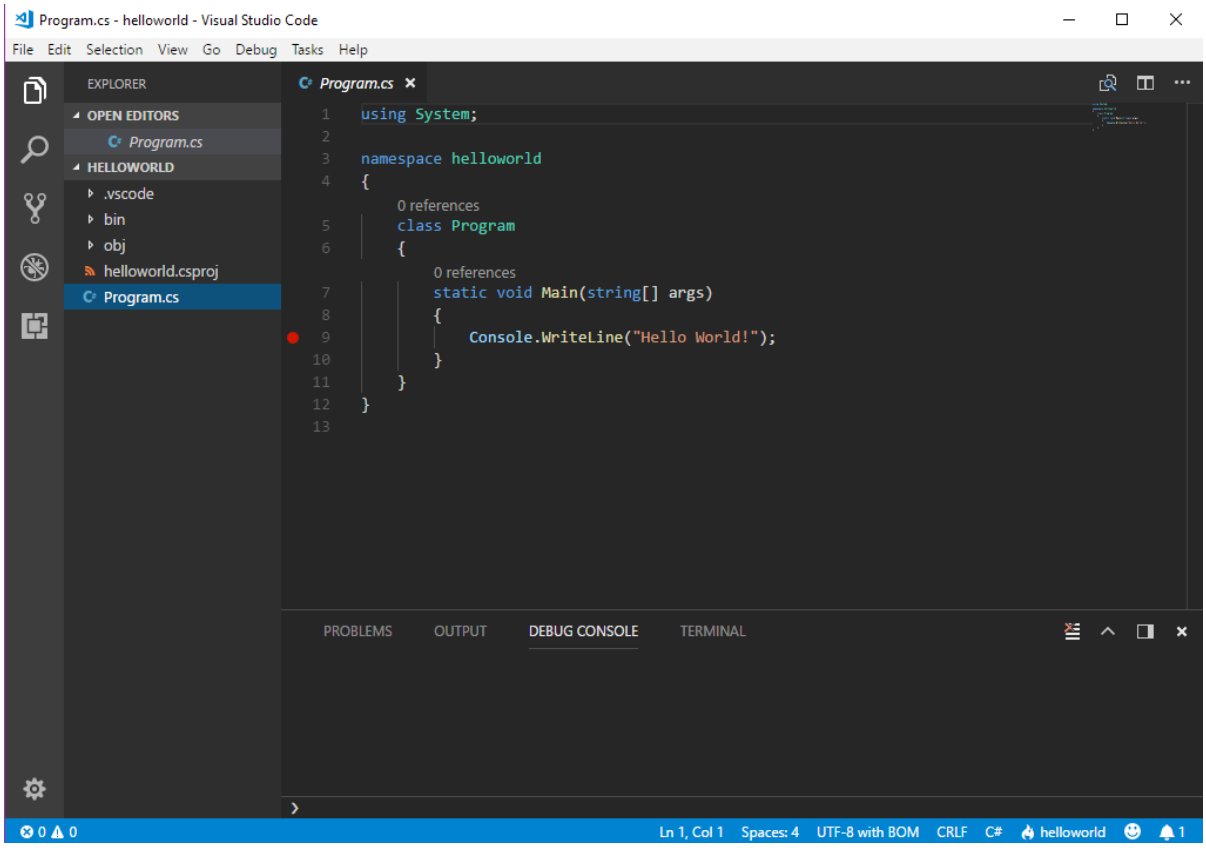
This "register the channel" followed by "use the standard package manager" is actually the same story for Fedora, openSUSE, CentOS, and Debian; again, full step-by-step instructions are on the .NET Core "Getting Started" page (https://aka.ms/dotnet-getstarted-linux), which has a drop-down to choose between the six flavors of Linux that .NET Core supports. Add the Microsoft feed to the package manager, use the package manager to do the install, and the "dotnet" tool should be available on the PATH.

Once that's done, .NET Core is ready to go.

Visual Studio Code

For almost any operating system you might find yourself using, including most of the popular distributions of Linux, Microsoft has a lightweight editor called Visual Studio Code. If you've been doing a lot of web front-end work, you may already have been using it. In many ways, it's the latest attempt at building an "editor-plus-plug-ins" tool, and it is gaining some traction with the world outside the Microsoft ecosystem. (It's actually the preferred tool for working with Google's open-source Angular project, among other things.) It's also available from http://code.visualstudio.com, it's available for Windows, macOS, and most of the major flavors of Linux, and it's a pretty quick download and install. And, honestly, it's not a bad editor for writing Java code, in case you were wondering; check out the Java extensions at https://code.visualstudio.com/docs/java/extensions the next time you're looking for a lightweight editor that understands Java projects.

Firing up Visual Studio Code from the command-line is often just a simple "code" if you're in the root directory of your project.



Mac .NET?

If you're like a lot of Java developers (myself included), you own a macOS machine. It might surprise you to know that many of the developers at Microsoft do, too. (Some of them repaved it with Windows, but a lot of them run native macOS.) Doing .NET on macOS falls into one of two options.

One is the command-line approach, which you can obtain via Homebrew. For the moment, using Homebrew requires tapping a cask, but after that, it's as simple as doing the classic "brew install" thing:

```
$ brew tap caskroom/cask #if you haven't already
$ brew cask install dotnet-sdk
```

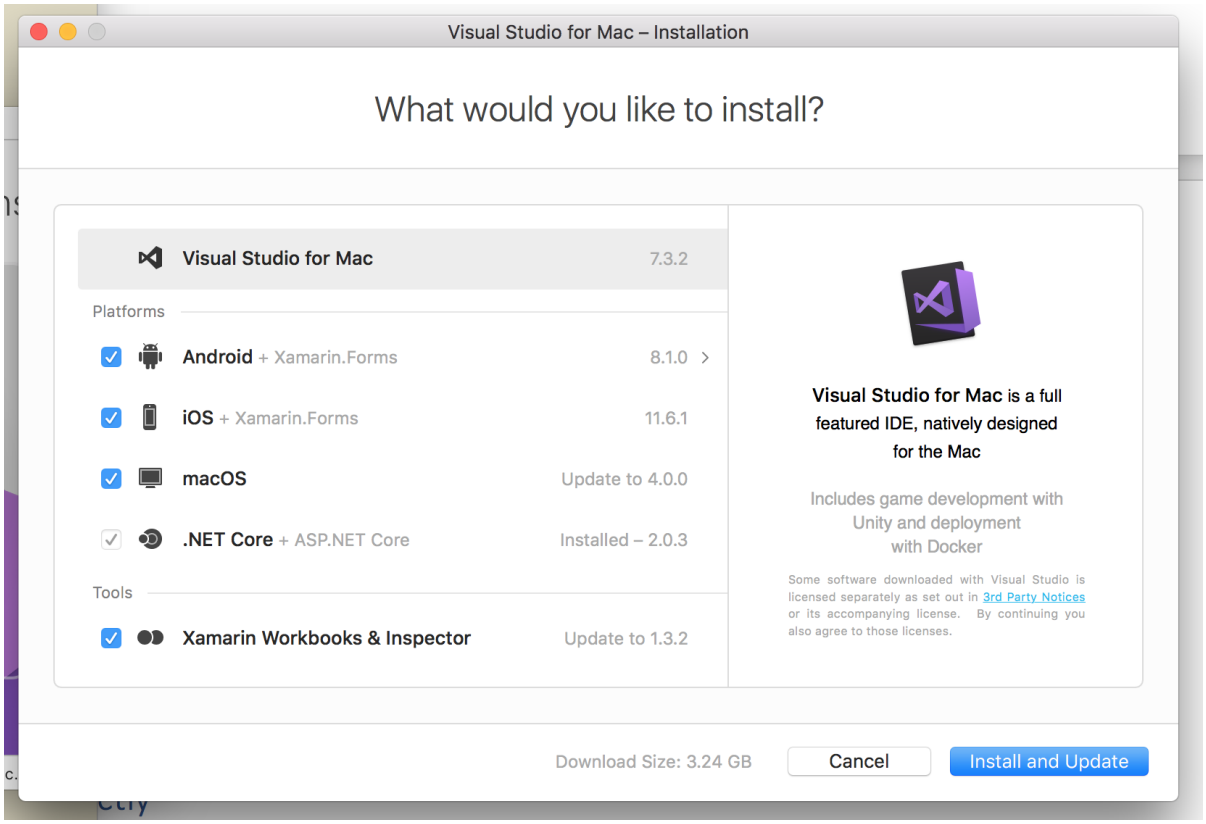
This will run a custom installer script that will put .NET Core onto your machine, so it's not completely Homebrew-friendly, but it "does the job." Once finished, fire up a new Terminal window (to pick up the environment changes that the installer makes) and run:

```
$ dotnet --version
2.1.1
```

Which should respond with "2.<something>," depending on which version is published as of the time you read this.

If you're a bit more of an IDE fan, however, you're still in luck. With its acquisition of Xamarin, Microsoft also acquired Xamarin's IDE for the Mac, which was just recently re-branded to be called "Visual Studio for Mac." Based on the open-source "MonoDevelop" project, it's not quite as extensive an IDE experience as Visual Studio, owing to a smaller set of plug-ins developed for it, but that can actually make it more enjoyable to use sometimes, particularly if all you're looking for is a good editor, project management system, and debugger.

Installing Visual Studio for Mac is a pretty straightforward experience—http://visualstudio.com has a link for an installer that, when opened, will go through the usual Mac installation process. There are a number of options here that aren't absolutely necessary for our purposes, but since a complete install is measured only in single-digit gigabytes of disk space, unless you're constrained for space, why not just keep it simple and install it all?

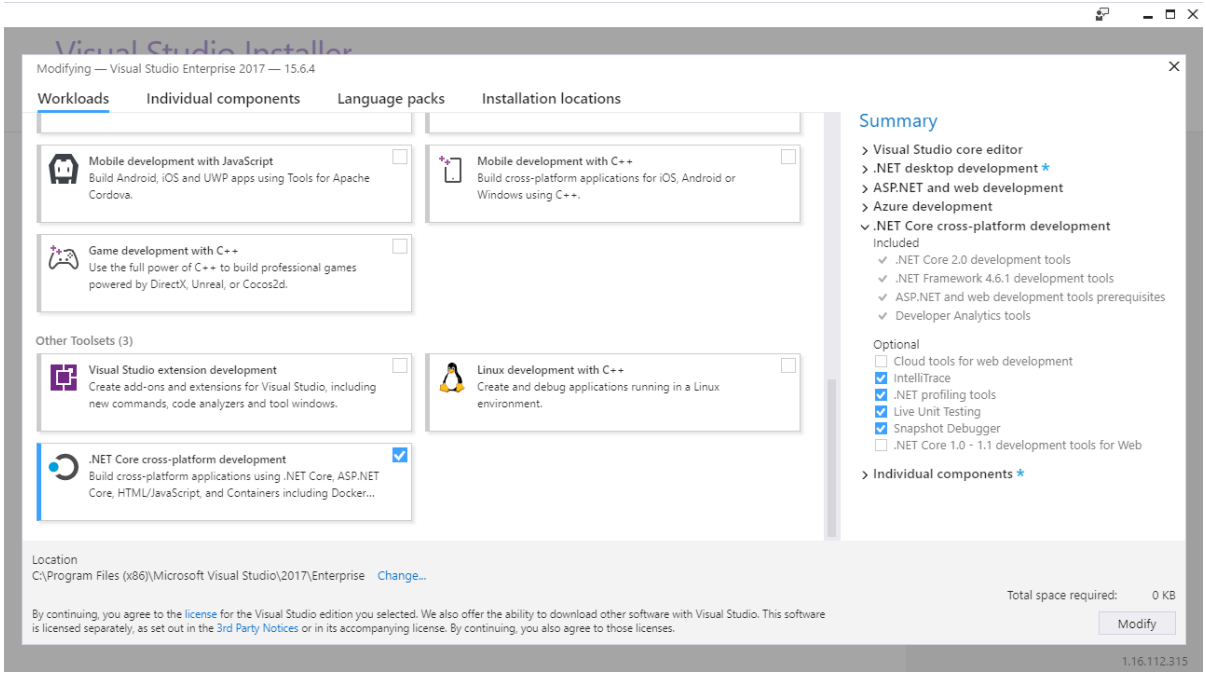


Once started, take a few minutes to grab some coffee; like its cousin on Windows, Visual Studio for Mac will download the components it needs and install them to disk. Don't walk away from the machine for too long, however—the installer will ask for the administrator password several times during the installation.

Windows: Visual Studio

Lastly, if you're running Windows, the most obvious option is installing Visual Studio—that combination has been the default for Windows-based development for close to two decades, and most Java developers know enough already to realize that. What many don't realize is that there is a free-to-download "Community Edition" that has all the core parts of the IDE and toolchain: compiler, debugger, editor, project management, and so on. You can find the Visual Studio Community Edition via your favorite search engine or wander on over to http://visualstudio.com to grab an installer. Make sure you're on a good Internet connection, since 98% of the install is coming from the Microsoft servers.

When the Visual Studio installer fires up, it immediately becomes obvious that this is every bit as large of an installation as Eclipse or IDEA—hundreds of options lie before you. While most of them are not necessary (many of them have to do with building unmanaged C++ code, for example, or for building game projects), selecting all of them will help ensure that anything you might look to explore will be available to you. The total install size is about the same as your favorite Java IDE with all the trimmings, when all is said and done. Alternatively, you can select just ".NET Core Cross-platform development" for a streamlined install of just .NET Core.



If all of this seems a little hard to remember, honestly, you're not wrong. Remember, Microsoft is fairly new to this "make it work on all platforms" story, and they're constantly trying to improve the whole experience, which in the past has resulted in some churn and confusion. One of the things they've done in 2017 is to try to collect all their .NET-related resources together into a single URL for easy reference. That URL, www.dot.net, is the place to start, and will end up redirecting to the main Microsoft site dedicated to the .NET ecosystem. It's an overview of all the different things one can do with .NET, as well as the different platform installers. Probably the most useful things are the button-links to "Get Started" and "Download"; the latter takes you to a collection of different download links, and the former takes you to a series of tutorial steps to getting started building a console "HelloWorld" application. (Sounds familiar, no?)

Hello, dotnet: CLI style

Regardless of how you've got the command-line tooling on your machine, it all boils down to one thing: The machine has a command-line tool called "dotnet" that's in the PATH and hooked up to an Internet connection. Away we go.

The "dotnet" tool, in many ways, is a re-examination of the thought process that Maven brought into the world: a command-line tool that can generate out a project based on customizable/extensible templates stored in the cloud, as well as act as a single point-of-control for building, running, testing, and any other commands that need to happen to a given project. The "dotnet" tool hides a fair amount of .NET Core's machinery, in the same way that Maven hides a fair amount of the Java machinery (CLASSPATH references, for example), so keep in mind that we're only looking at the tip of the iceberg as we work with this. (Make no mistake—the tooling here goes just as deep and just as wide as anything we've ever used in the Java world, and it's just as easy to get lost in it if we dive too deeply into it too quickly.)

To start a .NET Core application, we can use "dotnet" to generate a project based on the "console" application template, like so:

```
$ dotnet new console -o HelloWorld
```

This will generate a "console" project into the directory "HelloWorld"; for those who are curious, "dotnet new" by itself, or "dotnet new --list," will list all of the project templates that the "dotnet" command knows about. Out of the box, it has about a dozen or so project templates, including console applications, class libraries, a half-dozen flavors of web projects, and so on. This list is expected to grow and evolve as time goes on, just as the list of Maven archetypes grew after its release. Note that the list also allows generation of projects across different languages, so those who really want to sneak a peek at what an F# Web API project would look like can run:

```
$ dotnet new webapi -o HelloFSAPI -lang F#
```

ASP.NET Web API will be discussed in a little more detail later in this paper; however interesting, though, F# is not something we're going to explore here. You can learn more at fsharp.org.

Implicit in the "new" command is a request to download all of the dependencies that a project requires, which in "dotnet" parlance is a command to "restore." This requests all of the dependent assemblies—JAR file equivalents, essentially—from a Maven-like, cloud-based service called NuGet. NuGet was heavily inspired by Maven and Ruby gems, which means, for the most part, it works the same way Maven does—when doing a "dotnet restore," the machine must be connected to the Internet, and depending on the size of the project, could take a while to download all the dependencies.

Once the project has been "restore"d, all of the dependencies are downloaded, and the application can be run by using "dotnet run" from within the project directory. This will, not surprisingly, print "Hello World!" to the terminal. If outside of the project directory, tell "dotnet" which project to run by passing a reference to the project file (HelloWorld.csproj[6]) in via the "--project" command-line argument:

```
$ dotnet run --project HelloWorld/HelloWorld.csproj
Hello World!
```

If you're inside the project directory, this becomes a simple "dotnet run"; .NET assemblies know their entry point intrinsically.

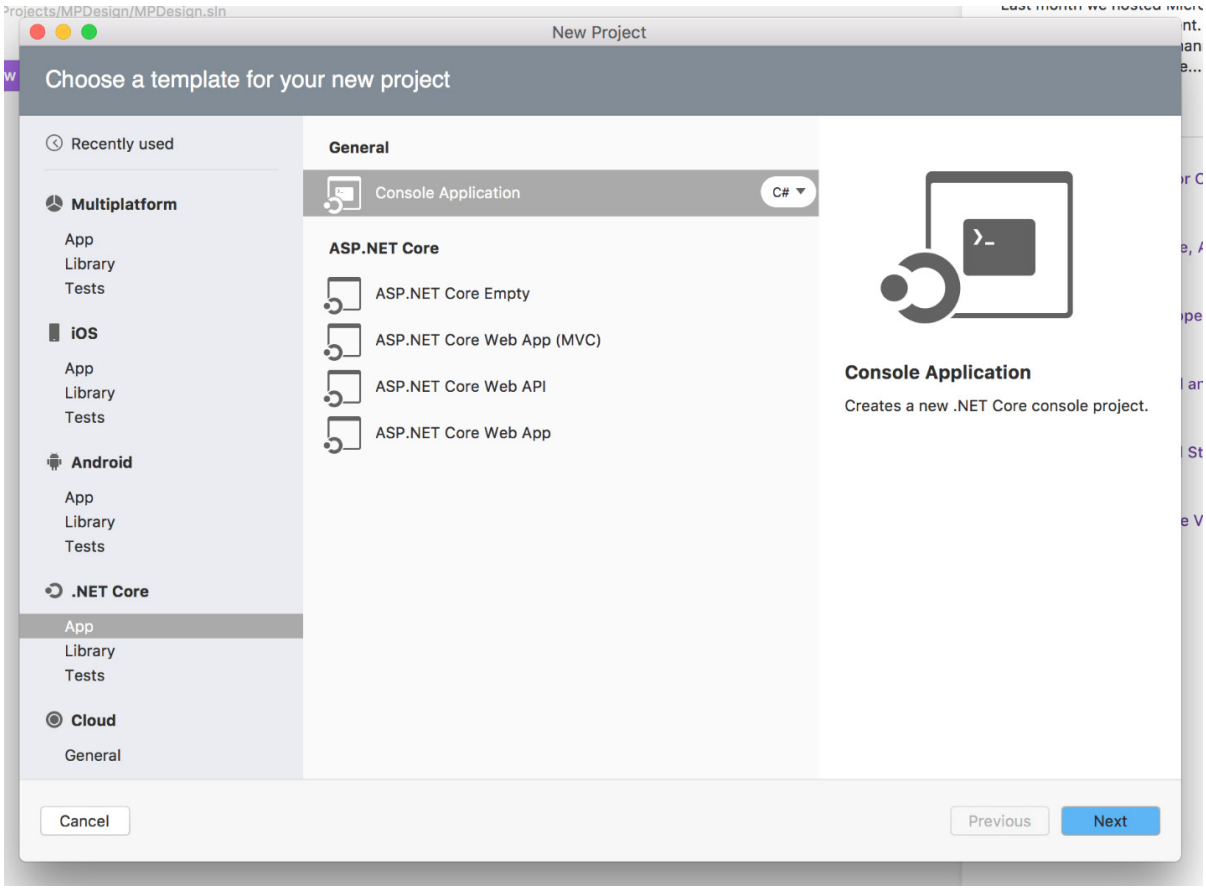
If you're feeling curious, poke into the "bin/Debug" directory underneath HelloWorld; in it resides the generated compiler artifacts, including the .NET assembly, "HelloWorld.dll," and its debugger support file, "HelloWorld.pdb." It may seem strange that one would "execute" a DLL, but an assembly, like a JAR file, is a repository of bytecode (but with much tighter cohesion than a JAR file provides), and the extension is (for the most part) almost entirely irrelevant. The dotnet launcher serves much the same purpose as the Java launcher does: to find the assembly specified on the command-line, load it, then execute the entry point method and get out of the way.

Hello, .NET: IDE style

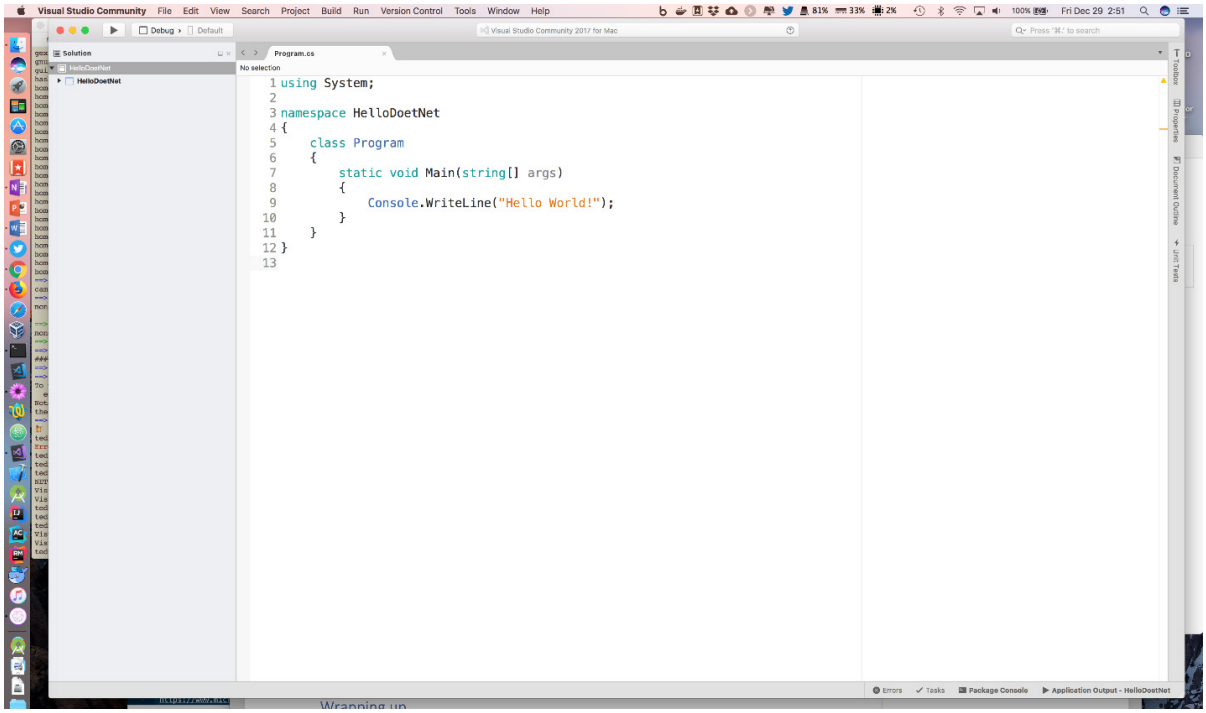
If Visual Studio (or Visual Studio for Mac) has finished installing, fire it up, and either select "File | New Project," or use the Start Page displayed in the center of the IDE to begin a new project by clicking one of the "New Project" templates. Depending on which Visual Studio you're running, you may be presented with two kinds of console applications: one that represents a native Console Application (for Windows, it will reference .NET Framework) and one that targets .NET Core (that's the open-source implementation). Choose either one (I prefer .NET Core), then do the usual IDE thing of choosing which directory in which the project should live, and its name. Note that we can designate this as a Git repository as well. Once finished, Visual Studio will think for a bit, then show the finished template, complete with C# code, ready for editing, building, and debugging.

---

[6] The .NET Core tooling took a bit of a detour recently through the world of JSON by working off of a "project.json" file, which is still mentioned in a number of .NET Core tutorials; that has been deprecated in favor of the MSBuild-based files, and is no longer supported. JSON is still used for a number of configuration files, but the .NET community reacted pretty strongly to the use of JSON for the project files, and Microsoft, looking to be a good open-source steward, responded to that feedback by backtracking and moving back to MSBuild.



Building the code (Ctrl-Shift-B on Windows, Command-B on macOS) produces a console application (a native Windows .exe for Windows or a .NET Core project for macOS), which can either be built by navigating to the "Debug/bin" subdirectory of the project and running it directly or by launching it from within Visual Studio.



The debugger works the same as it does within any IDE, and Visual Studio has its share of rich debugger tricks (which are really beyond the scope of this article, but easy to find https://aka.ms/vsdebugger). HelloWorld, achievement unlocked.

Solutions and projects

Within the Visual Studio ecosystem, code is arranged into "projects," which are in turn collected into "solutions." A project is language-specific, and sports an XML file whose extension reflects that language—C# projects are stored in ".csproj" files, for example. Each project produces one artifact—an assembly, which we'll discuss in more detail later. Solutions, then, are made up of any number of projects listed in a solution file (.sln), making it quite common for a solution to have some top-level "executable" (whether that is a website project, Web API project, or some form of GUI project) with one or more "library" projects to support it. It's extremely common, in fact, to see projects in separate subdirectories under the directory in which the solution file lives. This arrangement won't seem strange to the working Java developer; in many respects, this is no different from what we would see in a Java codebase, where some collection of code (project) produces a JAR file, and a Maven or Gradle build (solution) brings it all together into a bundle of some form.

Microsoft is working to keep all these file types in sync across all versions of Visual Studio, so that a project that builds on Visual Studio for Mac should also build on Windows, using an underlying build system called "MSBuild." Heavily Ant-inspired, MSBuild has been the build system for .NET since

Visual Studio 2005, and Xamarin was quick to adopt it for their project file system when they started building out their toolchain.

Goodbye, HelloWorld

Tooling is now installed. The Gods of Computer Science have been appeased by the creation of a successful HelloWorld application. It's time to take a pass across the C# language.

C#, Succinctly

While we can't cover the entirety of the C# language in half a dozen pages, we can cover the key concepts of C# pretty quickly. In fact, we can do it in one sentence: C# is an object-oriented language that borrows heavily from its lexical ancestors, C++ and Java, and from other "component"-oriented languages such as Delphi. In recent years, C# (like most languages) has begun to incorporate more functional programming concepts into its syntax, and as a result presents a pretty multi-paradigmatic set of capabilities.

In other words, take Java, throw in a number of the features Java developers have debated for a decade or more, then add a bunch of functional programming concepts, stir well, bake for 15 years, and serve. Microsoft's language designers have been much more aggressive than Sun's when it comes to introducing new features into their respective languages, so for the most part a Java developer's experience with C# will probably be characterized by "Oh, sure, that all looks pretty similar—wait, what is THAT?"

By the way, C# currently sits at version 7.2 at the time of this writing, and Microsoft's language designers have been just as aggressive in the latest revisions of the language as they were in the early ones. This is a language under constant evolution, for good or for ill. So there's much to discuss, and we will not be able to cover the entire surface area of the language—the focus here will be only on those parts that will be unintuitive to the Java developer looking at "average" C# code.

C# Basics

To begin, let's take a look at what was generated as part of those "HelloWorld" console applications, since that's probably the easiest way to get started.

```
using System;

namespace HelloDotNet
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

It's pretty easy to spot a number of things right off the bat: C# is a C-family language, using curly-brackets to denote scope blocks and semicolons to terminate statements. Code is organized into

namespaces using the "namespace" keyword, and pretty clearly we have classes just as Java does. The Console class (part of the "System" namespace) represents the command-line input/output facilities, and WriteLine is obviously a synonym for "println." Square brackets are used to denote an array in the same way they do in Java, and arrays in C# work almost exactly the same way as they do in Java, save that they have been slightly more "formalized" in that there is a full-blown class, System.Array, that defines a number of methods and properties that all arrays will share. Core "primitive" types are actually defined as full-blown classes, so that System.Array is the "class" representing each array instance, System.String is the "string" primitive type, System.Int32 is the "int" primitive type, and so on. All types, whether "primitive" (what .NET calls "value types") or "class," all inherit from the base System.Object type, with value types being "boxed" to objects when necessary. (This was the inspiration for autoboxing rules in Java5, in fact.) As a result, "System.String s" and "string s" are actually entirely equivalent declarations. New value types can be defined, but it's pretty rare.

Let's start with some of the more straightforward parts of the language.

Starting exploration: classes, properties, and methods

C# 1.0 was, in large measure, a near-direct clone of Java, with some keyword changes and some new features that Java has debated for years: explicit properties and delegates (an early form of lambda expressions). Thus, we can write the traditional "Person" type in C# like so:

```
class Person
{
    public string FirstName
    {
        get { return this.firstName; }
        set { this.firstName = value; }
    }
    private string firstName;

    public string LastName
    {
        get { return this.lastName; }
        set { this.lastName = value; }
    }
    private string lastName;

    public int Age
    {
        get { return this.age; }
        set { this.age = value; }
```

```
        }
        private int age;
        public Person(string fn, string ln, int a)
        {
            this.FirstName = fn;
            this.LastName = ln;
            this.Age = a;
        }
    }
```

Properties can be read-only by eliminating the "set" clause, and write-only (if that makes sense, which it rarely does) by eliminating the "get" clause. The implicit parameter to a "set" clause is always called "value," and there are no restrictions to what a property-get or property-set clause can do within its body, although 95% of the time they simply read and write a field within the class.

So far, if anything, it seems more verbose than Java. However, that was the C# language as of 15 years ago—they've made a few changes along the way.

One very common change is to the requirement to actually provide a body to the property-get and property-set clauses; as is the case in Java, 95% of the time these clauses simply wrap around a field and provide encapsulated access and nothing else. Which means, frankly, that the time spent writing the implementation for these is pretty much a waste of time (and something that IDEA or other IDEs can generate for us). The C# language designers agreed, and in C# 3.0 added the concept of "auto-implemented properties," which asks the C# compiler to not only generate the property method implementations, but also synthesize the field used for the backing store. This simplifies the Person class down to:

```
    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }

        public Person(string fn, string ln, int a)
        {
            this.FirstName = fn;
            this.LastName = ln;
            this.Age = a;
        }
    }
```

That's…awfully terse. But what about those scenarios where we want to have the property available to be read by anyone (public access), but only accessible for mutation from within the class (private access)? The get and set clauses can have different levels of access, the most common idiom being:

```
class Person
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public int Age { get; private set; }

    public Person(string fn, string ln, int a)
    {
        this.FirstName = fn;
        this.LastName = ln;
        this.Age = a;
    }
}
```

This will effectively make the fields immutable from the outside yet still something we can change from the interior of the class. The field itself is synthesized by the compiler (usually with a straightforward name such as "__$<Person>$<FirstName>@5417," or something equally non-obvious), and won't be directly accessible by code. (Granted, one can always use System.Reflection—the .NET direct equivalent to Java's java.lang.reflect package—to access the field at runtime, but that's an awful lot of work to work around something that can be discarded at any time.) For those rare situations where the auto-generated properties don't do the right thing, C# developers can always "drop back" to writing the property clauses out longhand and wrapping a declared field.

Methods, meanwhile, work pretty much the same way they do in Java, in terms of their declaration and syntax. Methods can be instance methods or static (although static method invocations must always go through the class name, whereas Java will allow calling a static method through an instance variable—seriously, try it, it's weird), and parameters are declared in order with a type and a name:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    public Person(string fn, string ln, int a)
    {
```

```
            this.FirstName = fn;
            this.LastName = ln;
            this.Age = a;
        }
        public void SayHello(string recipient)
        {
            Console.WriteLine("Hello, {0}, I, {1}, say hello.",
                recipient, FirstName);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person p = new Person("Ted", "Neward", 46);
            p.SayHello("Fred");
        }
    }
```

As you can see, C# follows pretty much exactly the same rules that Java does for method declaration and invocation. But, again, C# allows for some interesting variations on this theme.

For starters, C# has had the concept of "method literals" since 1.0, though the syntax has changed over each successive revision of the language. The first step was "delegates," essentially the ability to declare a method signature and then assign methods-on-object-instances to a given delegate instance, like so:

```
    class Person
    {
        // . . .
        public delegate void HelloProc(string recipient);
        public HelloProc IndirectSayHello;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person p = new Person("Ted", "Neward", 46);
            p.SayHello("Fred");

            p.IndirectSayHello = p.SayHello;
            p.IndirectSayHello("Fred");
        }
    }
```

This was a pretty verbose way of doing method-reference-capture, and with successive versions of C#, they simplified it, particularly once the .NET platform introduced generics. (Yes, C# has generics, yes they look syntactically almost identical to Java generics, but no, they don't work the same way when compiled—where Java uses "type erasure," C# uses "reified generics" and persists the generic definition all the way into the compiled bytecode.) The "delegate" keyword here introduces a new type—a function pointer, if you will—and we can declare an instance of the delegate on the Person class, assign a method to it, and invoke the assigned method through that delegate.

Lambdas, type inference, and extension classes

By the time C# 3.0 came around, however, C# supported full-bore lambda syntax, complete with method parameter type inference, so that we could write method literals in-line:

```
p.IndirectSayHello = (recipient) => {
    Console.WriteLine("Hey, {0}. Whatever.", recipient);
};
p.IndirectSayHello("Shaggy");
```

Lambdas turned out to be a major turning point within the C# language, along with several other features of the C# 3.0 release, including local variable type inference, which simply allows for the keyword "var" to appear in place of the type in a local variable declaration, like so:

```
var x = 12;
var y = "Fred";
var z = p.IndirectSayHello;
z("Wilma");
```

The "var" keyword does not, as some in the Java world theorized at its introduction, create an untyped variable; the C# compiler simply infers the variable's type from the right-hand-side expression, and assigns that to the variable type. Thus, the "x" variable is strongly typed as an integer (since that is an integer literally on the right-hand side), "y" is a string, and "z" is a delegate that accepts a method reference that takes a single string parameter and returns void. All the "var" keyword does, simply, is allow the C# developer a brief respite from having to type out type signatures in places where the compiler could figure it out on its own.

On top of this, Microsoft also introduced a fascinating feature called "extension classes," which has found its way into other languages such as Swift. In short, a class can write "extension methods" that can effectively "inject" themselves as method declarations onto other classes, adding new functionality to classes already compiled. (It's vaguely reminiscent of aspects as described by AspectJ, but a strictly limited subset thereof.)

For example, assume that it's becoming really common to want to count the number of words in a given String. In the earliest days of Java, it was assumed that we could extend the String class to add that kind of functionality, but Sun very quickly clamped down on that, made String final, and

earned the (short-lived) wrath of Java developers everywhere. Java developers were forced to write "StringUtils" classes that had static methods on them to count the words in a string, similar to this C# equivalent:

```
public class StringUtils
{
    public static int WordCount(String str)
    {
        return str.Split(new char[] { ' ', '.', '?' },
                    StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Frankly, though, using this left a pretty bitter taste in everybody's mouth, since it smacked of doing procedural programming. Java developers just learned to live with it; C# language designers said, "We can go one step better." If you agree that the class will never have any instance data and never be instantiated as an instance (declare the class "static"), then it can use the first parameter to declare a "this" parameter, and essentially declare a method that will appear on the targeted class, like so:

```
public static class MyStringExtensions
{
    public static int WordCount(this String str)
    {
        return str.Split(new char[] { ' ', '.', '?' },
                        StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

(Note that "String" is shorthand for "System.String," which is a synonym for the earlier type "string." All of the "primitive types" in C# are aliases for the System-namespaced types, so "int" is the same as "System.Int32," and so on. This has to do with how C# allows the declaration of "value types," and while a fascinating discussion in its own right, it's a little out-of-scope from what we're doing here.)

This extension class means that now we can write the following:

```
var words = "One two three four";
var wordCount = words.WordCount(); // 4
```

Extension classes don't just magically apply everywhere, however—they must be brought into scope via the use of the "using" statements, so that if we don't care to use the MyStringExtensions, the WordCount() method won't appear on String types. And the original class remains unaffected, which means that the WordCount() extension (like all extensions) cannot have any impact on the internals of the extended class—in fact, it can't see any of the internal members, and so can only ever be able to use the public interface of the extended class. It's purely compiler syntactic sugar.

Language-INtegrated Query (LINQ)

All of this led up to one of the signature moments for the C# compiler with the 3.0 release: Language-INtegrated Query, or LINQ, for short. To many in the Java community, LINQ represented everything that was wrong with Microsoft, accusing them of "embedding SQL into the C# language" and turning C# into a 4GL that was proprietarily bound to SQL Server. As the Jedi Master put it, "Amazing—every single part of that sentence is entirely incorrect." LINQ wasn't integrating SQL, it isn't about SQL Server, and it's actually built up around core language principles that had nothing to do with relational databases, much less integrated with SQL Server.

Let's start by examining a simple collection of Person objects in a list:

```
var people = new List<Person>() {
    new Person("Ted", "Neward", 45),
    new Person("Charlotte", "Neward", 39),
    new Person("Michael", "Neward", 24),
    new Person("Matthew", "Neward", 18),
    new Person("Beth", "Massi", 39),
    new Person("Dustin", "Campbell", 43)
};
```

Suppose I want to give everybody within this list a beer. Sharp readers will notice, of course, that my youngest son is not yet eligible to receive a beer, and this is a family-friendly article, so we don't want to get into trouble with the local authorities.

Were this a discussion of Java8 lambdas, we would talk about using lambdas and streams to filter the collection before applying a higher-order function across each to give each individual a beer. (If that's new to you, Venkat Subramaniam has some great conference lectures online that describe Java8 lambdas in more detail.) As far back as C# 3.0, Microsoft saw the need, and chose not only to layer functional principles into the language, but to put a syntax in place that would actually hide the fact that this was ever functional programming in the first place:

```
var drinkers =
    from p in people
    where p.Age > 21
    select p;
```

This is a LINQ query expression, and despite the fact that it looks similar to SQL, it's actually invoking several functional methods in sequence using a fluent interface; translated out of the SQL-like syntax, it looks more like:

```
drinkers = people.Where((p) => p.Age > 21);
```

All of the traditional functional operations are present here—map, filter, and the rest—but they "hide" behind this query expression syntax. We'll talk more about the interesting features and implications of LINQ when we get to Entity Framework in the last section, but for now, just think of

LINQ as a syntax that is designed to deliberately bring some of the set-oriented nature of SQL and the functional-oriented nature of languages such as Haskell into the C# language as a first-class citizen.

(For anybody who really wants to see LINQ taken to an absolute extreme, it can be instructional to look at Luke Hoban's "single-LINQ-expression-ray-tracer" blog post from 10 years ago, available at https://aka.ms/linqraytracer. It will send chills down your back, one way or another.)

Asynchronous programming with Async/Await

C# 4.0 introduced a number of interesting features, but C# 5.0 truly bucked the status quo by introducing language support for a form of asynchronous programming via the "async" and "await" keywords. While not a general-purpose, all-around concurrency feature, async/await added the ability to easily add "little concurrency" into common programming tasks such as issuing a request for a database query or executing a Web API request without having to manually manage the threads necessary to do so without blocking. Leveraging some work done elsewhere on the .NET platform (the Task Parallel Library, or TPL), C# essentially now allows developers to start some work and return a Task<T> object, which represents that additional work. At the source code level, "async" can decorate either a method signature or an anonymous method literal to indicate it should run asynchronously, and "await" tells the compiler to generate the code necessary to hold things in place until the asynchronous code completes. (Essentially, these are language-level implementations of "promises," which are permeating the JavaScript world and slowly starting to gather steam in the Java world as well, under the name "CompletableFuture" out of the java.util.concurrent package in Java8.)

```
async Task PlaySongsAsync(int[] ids)
{
    foreach (var id in ids)
    {
        try
        {
            var data = await service.GetEncodedDataAsync(id);
            var song = await Task.Run(() => Decode(data));
            await PlayAsync(song);
        }
        catch (Exception e) { /* just skip the song */ }
    }
}
```

There is much, much more we can discuss with respect to C#, but then we'd have no time to talk about anything else, and there is much, much more to go over on our tour. Fortunately, Microsoft has a great collection of web resources on the language, starting with a page that lists an overall history of the language, available at https://aka.ms/csharpwhatsnew, and a general (if abbreviated) overview of the language available at https://aka.ms/csharptour. And, of course, there're books. Lots and lots of books.

But once you know how to write code in this language, what can you do with it?

Frameworks, for the Win!

By far and away, the most popular thing to do with C# and the .NET runtime is to build a web application of some form, just as it is with the Java language and runtime. To be sure, the .NET ecosystem supports more than this—one can build desktop applications, mobile applications, and even games (using the aforementioned Unity framework, among others), but the lion's share of code written in C# is web applications.

Like its twin, .NET offers a comprehensive framework of web functionality, ranging all the way from drag-and-drop-built user interface, to MVC-based server-side rendering, to the construction of HTTP-based APIs for invocation by single-page applications written in JavaScript or mobile applications. Somewhat confusingly, however, all three are referred to by the name "ASP.NET."

(Note to those who recognize the "ASP" moniker from the early days of the Sun/Microsoft rivalry: In the COM days, Microsoft's web technology was called "Active Server Pages," ASP for short, named after ActiveX, one of the marketing names for COM. When Microsoft started working on the CLR internally, one of its most important consumers was the ASP team, who sought to create a system similar to JSP to replace ASP. Since the effort was to redo ASP in .NET, the name "ASP.NET" was coined. However, aside from the basic notion that "this is for the web," there's little to no relationship between ASP.NET and its predecessor.)

The ASP.NET "family" is fundamentally broken into three parts, two of which we'll examine here (in brief):

- ASP.NET WebForms, the first flavor of ASP.NET, released with .NET 1.0, featuring drag-and-drop user interface construction. It was designed to try to bring a VisualBasic-style design philosophy to the web, but by .NET 3.0, the Microsoft community had come to understand the design advantages of an MVC-style design approach, and all work on WebForms essentially ceased. Ironically, about the time that interest in WebForms was ending, the Java community was building up JSF, the equivalent to WebForms in Java.

- ASP.NET MVC: Based on the classic "Model-View-Controller" principles that the Sun "Model 2" design for servlets and JSP first laid out, Microsoft created ASP.NET MVC, released the source code for it shortly after its initial release, and has considered this to be its first-class server-side web framework ever since. ASP.NET MVC was rebranded (and refactored) to be "ASP.NET Core MVC" in 2015, in keeping with the ".NET Core" naming for the cross-platform runtime, and has made some conceptual shifts as new frameworks—such as Ruby-on-Rails—gained popularity and created some new patterns (such as routing tables).

- ASP.NET MVC Web API: While the name certainly gets no points for brevity, it does accurately reflect the idea that this framework is built off of the same concepts that underlie the ASP.NET MVC library, but with the intent of producing JSON data instead of HTML. When ASP.NET MVC was tagged with the "Core" moniker, the "MVC" was formally dropped from Web API projects, so that now they are simply "ASP.NET Core Web API," or more informally, "Web API" projects.

Let's have a look at a quick ASP.NET Core Web API application—the ubiquitous TODO list—and then we'll have a quick look at an MVC application before we look at Microsoft's data-storage framework, Entity Framework.

ASP.NET Core Web API

To get started with Core Web API, we first need to scaffold out the Web API template; this can be done either via "File|New" in one of the Visual Studio IDEs, or from the command-line using the .NET Core CLI tools by executing "dotnet new webapi -o todoapi." This will create a subdirectory called "todoapi" and install the .NET Core Web API project inside of it. Out of the box, it scaffolds out a simple API that has one controller to hand back arbitrary "value" objects, accessible at http://localhost:5000/api/values. Before we make any changes to the code, it's always good to make sure the scaffold is working, so execute a "dotnet run" from within the "todoapi" directory, and navigate to the given URL; the response should be a dirt-simple JSON array containing two strings.

A Web API project, like its cousin the MVC application, is broken into models and controllers, with the "view" assumed to be an automated conversion from a model object into JSON. Thus, in a Web API project, the goal will be to accept incoming requests, perform the necessary business logic, and return the appropriate model type, usually corresponding to the resource type in the URL, to be converted to JSON.

The Controllers directory contains the code for the ValuesController, which is the default-scaffolded code. It doesn't use any model objects, which means it's clearly inferior. Let's delete it, and replace it with something that's a little more "MV"-like. That means, first of all, that we need a model object to represent a TODO item, which is a pretty simple domain object:

```
public class Todo
{
    public long ID { get; set; }
    public string Description { get; set; }
    public bool Completed { get; set; }

    public Todo()
    {
        Completed = false;
    }
}
```

(Note that it would be a bit irregular to call this a POJO, since it's not a Java object, but that's basically exactly what this is. These are sometimes referred to as Plain Old C# Objects, POCOs, or, less often, Plain Old DotNet Objects or PODNOs. Definitely doesn't roll off the tongue like POJOs, but we do what we can.)

Having a domain object, it now remains to build a controller that will respond to the appropriate URL (which, by convention, will be prefixed by "/api," but is of course configurable). The Web API expectation is that a controller will be a class that has a number of public methods that obey some conventions, in terms of both method names and parameter sets, like Java has done with later versions of the Servlet and EJB specifications.

```
[Route("api/[controller]")]
public class TodosController : Controller
{
    private List<Todo> todos = new List<Todo>() {
      new Todo() { ID=0, Description="Get milk", Completed=true },
      new Todo() { ID=1, Description="Get pet food" },
      new Todo() { ID=2, Description="Learn .NET Core" },
      new Todo() { ID=3, Description="Write billion-dollar app" }
    };

    // GET api/todos
    [HttpGet]
    public IEnumerable<Todo> Get()
    {
        return todos;
    }

    // GET api/todos/5
    [HttpGet("{id}")]
    public Todo Get(int id)
    {
        return todos.First((t) => t.ID == id);
    }

    // POST api/todos
    [HttpPost]
    public void Post([FromBody]Todo value)
    {
        todos.Add(value);
    }
```

```
        // PUT api/todos/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody]Todo value)
        {
            var todo = todos.First((t) => t.ID == id);
            todo.Description = value.Description;
            todo.Completed = value.Completed;
        }

        // DELETE api/todos/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
```

To understand this class, it's important to understand a few things: First, the square-bracketed declarations outside each method are what .NET calls "custom attributes," which, true story, figured prominently in the design for JSR-175 annotations. The attributes provide information (via reflection, same way annotations do) about how HTTP verbs should match against the methods—so, for example, the ASP.NET runtime can know that the two Get() methods should both be invoked when an incoming GET request is made.

The other half of that binding, of course, is knowing the partial URL pattern that should trigger the method. Notice that the controller class itself has an attribute, "Routing," which contains the string "api/[controller]." The square-bracketed "controller" is to tell the ASP.NET runtime to use the name of the class, minus its "-Controller" suffix, so the URL pattern for all of these API endpoints is "api/todos." From here, standard REST conventions kick in—a GET to "api/todos" should return all of the TODO items, so the Get() method taking no parameters and returning an IEnumerable<Todo> (think of it as a high-level iterator interface) is the right one to invoke here. If, on the other hand, a caller wants a single TODO, they'll issue a request with a URL pattern of "api/todos/" plus the ID of the TODO requested. The second Get() method indicates that the URL pattern includes "{id}," so "api/todos/3" will extract "3" and pass it for the value of "id" to the method. Ditto for the Put and Delete methods.

The full details of routing are in the ASP.NET documentation, but this should be sufficient to understand what's happening here. To see the new controller in action, just do a "dotnet run" from the "todoapi" directory, and point an HTTP client (curl or Postman) at http://localhost:5000/api/todos to see the JSON representations handed back.

ASP.NET Core MVC

HTTP-based APIs are nice, but when we need a traditional server-side rendered application, ASP.NET provides a traditional MVC framework called (not surprisingly) ASP.NET Core MVC. It's the same MVC we find in most modern web frameworks; views are rendered using a particular template syntax, called "Razor" (or "Razor pages") that is designed to be expressive yet terse. It's a syntax inspired from other similar frameworks, particularly JSP and Ruby-on-Rails, but also tried to be reasonably familiar to generations of ASP and ASP.NET WebForms developers. Controllers look almost the same as they do in the API scenario, and models, of course, are just C# classes representing the domain.

Razor templates use a .cshtml extension, and like JSP, they use different syntax to indicate which parts of the page are to be interpreted in a code-aware manner, as opposed to being taken literally as output format. Razor syntax uses the "@" symbol to set off code blocks or other processing statements in several different ways; for example, to do a simple "for" loop within a page, Razor uses an "@for" to set off the loop. Unlike JSP, however, the Razor syntax is clear enough that there isn't a need to require explicit symbols to set off between code and HTML output:

```
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < 3; i++)
    {
        <li>Beetlejuice</li>
    }
</ul>
```

Views, of course, can make use of objects and data provided by a controller, often by making use of a predefined key-value map called ViewData. If we want the message and the number of times it should be printed to be set up by a controller (perhaps the controller is obtaining it from HTML query parameters or a database), then the controller can set those by using the keys "Message" and "NumTimes" in the ViewData dictionary, respectively. The view can then display them like this:

```
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

In essence, ViewData is the ASP.NET equivalent to Servlet "request" scope.

The controller perspective that drives this looks remarkably similar to what a servlet controller would do, with a few differences:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

Notice that the Welcome method takes two parameters, which will be filled in from HTML query parameters, fills in the ViewData dictionary, and then calls "View," which then looks for a Razor page file whose name matches that of the method—in this case, it will be looking for "Welcome.cshtml." ASP.NET Core MVC has a number of ways to adjust and change these conventions, but out of the box, the convention-based approach makes it simpler and easier than the configuration descriptor approach of older Servlet versions.

ASP.NET Core MVC obviously consists of a great deal more than just what we've discussed here, but the underlying message should be clear: ASP.NET Core (both the MVC and the Web API versions) learned a great deal from its contemporaries, including those in the Servlet world (JSP, the Play framework, and more), and as such, it will be very straightforward for a Java developer to pick up and explore.

For those interested in a deeper dive into the code above, consider having a look into this tutorial: https://aka.ms/first-web-api.

Entity Framework Core

Of course, being able to access the server and pass JSON back and forth is only half the story; the other half is taking that data and storing it. Or rather, storing it, indexing it, searching it, collating it, sorting it, drawing pretty graphs with it, and more. When working with a relational database, the technology Microsoft offers its .NET constituents is called Entity Framework, or EF, and in many respects it works like any other object/relational mapping framework does. Thanks to some of the language features in C#, however, in many cases working with EF is actually easier than working with JPA or Hibernate.

Like much in the .NET universe, Entity Framework is undergoing a bit of a reshuffle; what used to be a closed, proprietary source base is now open, and as EF was opened to the world, the project owners thought a (slight) rebrand seemed appropriate. As a result, the open-source version of EF is called "Entity Framework Core," and to begin using it, we need to install it into the project via NuGet, either by using the Package Manager in Visual Studio, or by using the command-line for .NET Core by entering:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

This is, of course, assuming we are using SQLServer—since not everybody uses SQL Server, however, a number of different EF database providers are available, including MySQL, Postgres, SQLite, Firebase, and a few more. (Third-party vendors are constantly introducing new ones, in a manner similar to the JDBC driver ecosystem, so bear in mind that Microsoft may not even know all the EF providers in the world.)

There're a few "management" tasks that will be easier to do from the command-line, and EF has a command-line package that installs "inside" the standard .NET Core command-line tool, but as of this writing, making use of it requires adding a line by hand to the project's .csproj file:

```
<ItemGroup>
  <DotNetCliToolReference
     Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
     Version="2.0.0" />
</ItemGroup>
```

In addition, the EF Core command-ling tooling requires another dependent package:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

At this point, EF should be installed and ready to go. More detailed instructions can be found on the Microsoft docs site at https://aka.ms/ef-install.

Fundamentally, the most critical element to using EF from a project is the DbContext. This is a class from which a project will create a derived class that represents the "context" for this project against the database—it will be a repository, session, and gateway, all rolled into one. Like many object/relational tools, it also provides some core behavior, including some migration capabilities (database versioning).

Like Hibernate or JPA, EF uses classes to represent entities in the database, and these classes are often referred to as "model" objects, since they typically serve that purpose in an MVC and/or Web API application. These will usually look like "plain old C# objects," with the odd custom attribute to help give EF some additional hints about how to map to the database table they represent.

For example, if the application is a blog, then we will want classes representing a blog, and blog post, representing a single blog entry. But there's also some blog metadata we might want to track that wouldn't be appropriate to store in the database (such as whether the blog was stored in the database), which we will want Entity Framework to ignore entirely. One way to do this is to use a custom attribute, or the EF Fluent API can indicate that the blog metadata class should be ignored using the "OnModelCreating" method, invoked when the database schema is created.

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }

    public BlogMetadata Metadata { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

The "Id" fields, of course, are typically primary key integer columns in the database, and the List of Posts is a classic one-to-many relationship. All of this will be eminently familiar to any user of Hibernate, JPA, or other object-relational modeling system. What will be different will be the way in which queries and modifiers are run; thanks to LINQ, Entity Framework uses an entirely new style of query starting from the DbSet fields modeled on the DbContext-inheriting class. To do a query to fetch all the blogs from the system, we simply create an instance of the context inside of a "using" block (which is the inspiration for the try-with-resources block in Java7), and use that instance to obtain the set of Blog instances and iterate through them:

```
using (var context = new BloggingContext())
{
    foreach (var blog in context.Blogs) {
        // do something with the blog
    }
}
```

Contrary to what we experienced during the heyday of EJB Entity Beans, however, this isn't a case of early loading or late loading; under the hood, Entity Framework is making use of "enumerator objects" (very similar to Java8 streams and the Java Iterable<T> interface) to fetch objects in batches, just as a typical database cursor would do.

In the Java world, however, filtering that list of blogs to include only those that have "dotnet" in the URL presented problems—either the filtering happens on the database side, in which case the query has to be translated to SQL, or the filtering happens on the client side, which means the entire collection has to be brought over the network. With LINQ, however, this looks like this:

```
using (var context = new BloggingContext())
{
    foreach (var blog in context.Blogs
        .Where(b => b.Url.Contains("dotnet"))) {
        // do something with the blog
    }
}
```

But this has the same drawbacks as before; instead, we can use an "expression tree," a feature of LINQ that compiles into a runtime structure that reflects the compiled code:

```
using (var context = new BloggingContext())
{
    var allDotnetBlogUrls = from blog in context.Blogs
        where blog.Url.Contains("dotnet")
        select blog.Url;
```

```
        foreach (var url in allDotnetBlogUrls) {
            // do something with the blog URL
            // note we don't have the entire Blog, just the Url
        }
    }
```

This "from-where-select" syntax that looks so much like SQL isn't; it's a syntax that's designed to look similar to SQL and offer many of the same kinds of operators and options. However, this isn't an SQL query—it will not be sent to the database. Instead, the expression tree is analyzed—and optimized—by the EF plumbing to generate an efficient SQL statement, on a driver-by-driver basis. This means that the query filters and restrictions are analyzed on the client side, but sent to the database to be executed—for example, the above turns into a "SELECT url FROM blog WHERE…" using whatever primitives to do a string-containment check that the database and EF provider offers. This is really the best-of-both-worlds scenario. Couple that with the compile-time type checking that the C# compiler provides against the model objects, and you have a solution that is entirely new to the Java world. (Not impossible to provide, mind you, but still something the Java world currently lacks.)

Updates and modifications entail making modifications on the model object, and those changes will be sent to the database when the context object goes out of the using block's scope. To add a new blog, construct the object and add it to the DbSet on the context object; to remove it, use the context's DbSet again. And should you really, really want to write raw SQL, the context object provides that capacity.

There is, of course, much more to discuss about Entity Framework—it's every bit as rich and powerful as JPA, and as such, definitely larger than just what we can cover in a single paper. For more examples of LINQ syntax, check out https://aka.ms/101LinqSamples. For 101 LINQ samples, and for more information on Entity Framework itself, head over to the EF documentation home at https://docs.microsoft.com/ef/core/.

Wrapping Up

To suggest that this is somehow the end of the story is ludicrous. The .NET ecosystem holds a rich tapestry of open-source projects and commercial tools and frameworks, and that means that a developer could, if they wished, spend their entire career just discovering the boundaries there. We haven't even begun to discuss the alternative languages for the .NET platform, either from Microsoft (F#, an object/functional hybrid that fills the same role as Scala, or good old Visual Basic, one of the original dynamic languages) or from the community (check out Boo, a Python-inspired dynamic language). We haven't begun to explore the different open-source frameworks that do dependency injection. We haven't begun to explore the different frameworks for doing unit testing, or the online tooling from Microsoft to provide agile project support. We haven't even mentioned Azure.

This is a brave new land, intrepid adventurer, but rest easy: Much of what you know from your time in the world of Java will serve you well here. The concepts are similar, but different enough to be interesting. The platforms are powerful, and the communities open and inviting. The tools are easily obtained, and the tutorials plenty. You might even recognize a few familiar names, those who have come before you across the pond and gone on this adventure before you. Be the tourist, if you like, but know that there's plenty of room to build your own home and settle down here, even if it's only a summer home. Come in, and be welcome.

And, unlike a certain young boy on his first trip to France, you won't have to worry about somebody's father trying to convince you that "escargot" is French for "tater tots."

Get started today!
www.dot.net