



We use optional cookies to improve your experience on our websites, such as through social media connections, and to display personalized advertising based on your online activity. If you reject optional cookies, only cookies necessary to provide you the services will be used. You may change your selection by clicking "Manage Cookies" at the bottom of the page. [Privacy Statement](#) [Third-Party Cookies](#)

Accept

Reject

Manage cookies



Microsoft

DevBlogs

.NET Blog

Theme



## Azure Developers - .NET Day 2023

Close

Tune into the live event on Wednesday, April 5th, 2023 to hear the latest in cloud computing for .NET

Save the Date! >



pers with Azure.



# App Trimming in .NET 5



Sam Spencer

August 31st, 2020 | 29 | 1

## What is trimming

One of the big differences between .NET Core and .NET Framework is that .NET Core supports self-contained deployment – everything needed to run the application is bundled together. It doesn't depend on having the framework separately installed. From an application developer perspective, this means that you know exactly which version of the runtime is being used, and the installation/setup is easier. The downside is the size – it pulls along a complete copy of the runtime & framework.

To resolve the size problem, we introduced an option to trim unused assemblies as part of publishing self-contained applications. We first made [assembly trimming](#) available as part of .NET Core 3.0. It is also sometimes called the "assembly linker". Optionally during the publish process, a trim phase occurs which does an exhaustive walk of the code paths identifying the assemblies that are used by the code. It will then only package those assemblies into the app, thereby reducing the size of the application.

In .NET 5, we are taking this further, and cracking open the assemblies, and removing the types and members that are not used by the application, further reducing the size. For example, for a Hello World app (`dotnet new console`), the sizes of assemblies are:

Assembly	Trimmed (k)	Member Trimmed (k)
System.Collections.Concurrent.dll	184.9	23.5
System.Collections.dll	85.5	15.5
System.Collections.Immutable.dll	171.5	20.0
System.Console.dll	61.5	31.5
System.Diagnostics.StackTrace.dll	33.9	8.5
System.IO.Compression.dll	88.5	33.0

Feedback

System.IO.FileSystem.dll	84.5	18.5
System.IO.MemoryMappedFiles.dll	28.0	22.5
System.Linq.dll	128.0	–
System.Private.CoreLib.dll	9,127.4	1,781.5
System.Private.Uri.dll	–	64.5
System.Reflection.Metadata.dll	432.0	109.5
System.Runtime.CompilerServices.Unsafe.dll	7.0	–
System.Runtime.Serialization.Formatters.dll	112.5	84.5
Total	10,545.1	2,213.0

\* As the commercials say, “results not typical”. The size reductions above are probably better than can be expected for most real apps – the app simply outputs “Hello World!”, so makes minimal usage of most of the assemblies. The size above is just for the framework assemblies, it doesn’t account for the runtime which is a fixed cost that applies to every app.

Trimming sounds great, but as with most good things, there is a catch. The trimming does a static analysis of the code and therefore can only identify types and members when they are referenced from code. However .NET offers a great deal of dynamism, typically depending on reflection. For example, Dependency Injection in ASP.NET Core uses reflection to select appropriate constructors. This is largely transparent to the static analysis, so it needs to either be told about the required types or be able to detect common dynamism patterns – otherwise it will trim away code that is needed by the application which will result in runtime crashes..

## Assembly-level trimming

To assembly-level trim, use the dotnet publish command either without a trim mode, or use `<TrimMode>CopyUsed</TrimMode>` in the project file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
    <PublishTrimmed>true</PublishTrimmed>
    <TrimMode>CopyUsed</TrimMode>
  </PropertyGroup>
</Project>
```

Or on the command line:

```
dotnet publish -r win10-x64 -p:PublishTrimmed=True [-p:TrimMode=CopyUsed]
```

## Member-Level Trimming

.NET 5 can take it two levels further and remove types and members that are not used. This can have a big effect where only a small subset of an assembly is used – for example, the console application above. Member-level trimming has more risk than assembly level trimming, and so is being released as an experimental feature, *that is not yet ready for mainstream adoption*. With assembly level trimming, its more obvious when a required assembly is missing, with member level trimming you need to have exhaustive testing of the app to ensure that nothing has been trimmed that could be required.



The default for .NET 5 is to use the same conservative assembly level trimming as .NET Core 3. Further gains can be made by enabling Member-level trimming by putting TrimMode=Link in the project file

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RuntimeIdentifier>linux-x64</RuntimeIdentifier>
    <PublishTrimmed>true</PublishTrimmed>
    <TrimMode>Link</TrimMode>
  </PropertyGroup>
</Project>
```

Or passing `-p:PublishTrimmed=True` and `-p:TrimMode=Link` to the dotnet publish command. For example the following sizes are for the [YARP sample project](#):

Trim Mode	Self Contained Linux x64 (MB)	Single File Linux x64 (MB)
No Trimming	78.9	65.3
Assembly Trimming	39.7	26.1
Member Trimming	31.5	17.9

Member level-trimming also strips the [ReadyToRun](#) (R2R) code from the assemblies – the results are smaller, but application startup will be slower. See further below for more details.

## Dynamic code pattern analysis

In the context of trimming applications, there are two main concerns:

- Any code and data that is *used* by the application *must be preserved* in the application.
- Any code or data that are *not used* by the application *should be removed* from the application.

Note the difference in “must” and “should” between those two concerns. An application that works is preferred over a smaller application that doesn’t. Therefore, it is critical that any necessary code must be preserved in the application. The problem with .NET is that it’s not always obvious as to what the necessary code is, particularly when it comes to dynamic code patterns – where the exact code that will executed is determined at runtime.

The big challenge for trimming is whether ILLink (the tool that does the trimming) can correctly discover all the code that can be reached by the application. The trimmer does a static analysis of the code, walking each of the code paths to determine the scope of what can be reached. There are a number of ways that dynamic code patterns can be used in the application that causes problems for trimming, the primary examples are:

- Using reflection – assemblies, types and members can be iterated over, or queried for by name, and then invoked. Eg  
`Type.GetType("Foo").GetMethod("Bar").ReturnType.GetMethod("Baz"),`
- Dynamic code loading – At runtime, assemblies are loaded and code within them executed. This is problematic if those assemblies have not been packaged within the app, or code that they depend on has been trimmed.

The problem with the dynamic patterns is that the trimmer cannot discover which types and members are going to be used at runtime, and so may be overly aggressive in trimming them out, which could result in catastrophic errors when they are used at runtime. For example, at runtime all the types that implement a specific interface could be queried for and instantiated. If those types are not used elsewhere in code that is reached, then they will be trimmed by the publish command. Later in the app’s



execution a type that was expected will not be there. This could happen at startup, or only in an infrequently used code path of the app – so the omission might not be caught in a simple test, and may not show up in unit tests unless they use the types in exactly the same pattern. Unit tests can also mask problems if they reference types or members directly, changing what the trimmer think is reachable.

This similar problem was faced by the .NET Native compiler used for Windows Store applications. The .NET Native compiler attempted to recognize examples of reflection, such as dynamic in C# and `Type.GetType("Foo").GetMethod("Bar").ReturnType.GetMethod("Baz")`, and try to make it just work. Despite significant effort, it was far from perfect, and the long tail of bugs meant that user assemblies could not be trimmed for reliability reasons. As .NET Native only did the native compilation as part of a retail build, issues did not show up for the “F5” inner loop of code-compile-run-debug, and only when the app was published, causing developer angst.

With .NET 5+, we want to take a different approach – we want the trimmer to provide predictable results to the developer: If the trimmer can’t be assured of the correctness of the trim, it will provide warnings based on the code used. The problem for dynamic code is not that it’s broken in a trimmed environment, but that the trimmer needs to know which assemblies/types/members will be used. A set of attributes have been added that enables code to be [annotated](#) to tell the trimmer what code should be included, or which API usage should prevent trimming.

<a href="#">[DynamicallyAccessedMembers]</a>	Applied to instances of System.Type (or strings containing a type name) to tell the trimmer which members of the type will be accessed dynamically.
<a href="#">[UnconditionalSuppressMessage]</a>	Used to suppress a warning message from the trimmer if the use case is known to be safe. For example, if an “Equals” method is written by retrieving all the fields on a type and looping over them comparing each field. If a field is unused, and therefore trimmed, there is no need to compare that field for equality.
<a href="#">[RequiresUnreferencedCode]</a>	Tells the trimmer that the method is incompatible with trimming and so warnings should be presented where the method is called. This will suppress warnings for the code paths that are called by this method reducing the noise, and making it clearer to the developer which methods they call are problematic.
<a href="#">[DynamicDependency]</a>	Specifies an explicit dependency from a method to other code, if the method is preserved the other code will also be preserved. Used in cases where the dependency is not expressed by the method, but rather by external factors, like native code.

We’ll go into further detail on these attributes and how to use them in a subsequent blog post.

Due to the ambiguous nature of detecting issues, the trimmer tends towards reporting false positive issues when the code may be safe, rather than allowing for problems at runtime. The trimmer warns on all the (potential) problems it finds in the assemblies consumed by the app – this can yield hundreds of issues for the simplest of projects.

For .NET 5 we have a couple of mitigations:

- a. Issues are reported as warnings.
- b. By default, the trimmer will suppress the trim warnings as they are not actionable by app developers when they are coming from the .NET Framework or other libraries.



<SuppressTrimAnalysisWarnings>>false</SuppressTrimAnalysisWarnings>  
can be used to show these warnings.

## Testing Trimmed Apps

When an app is trimmed, it is essential to perform exhaustive end-to-end testing of the published version of the application. Unit tests are often not useful because they change the environment too much and the trimming will work differently. Testing of a trimmed app should be driven externally rather than by code within the app.

## Trimming and Ready2Run

Ready2Run (R2R) is an AOT technology that native compiles code at build time for faster app startup. The assemblies in the .NET Libraries include R2R native code to improve startup for all scenarios. R2R is a tradeoff favoring performance over size. Using R2R, methods don't need to be JIT before they are used for the first time. Therefore R2R has most effect on startup, but there should be little difference once the process is warmed up, for example when processing web requests.

When an application is trimmed, the assemblies may need to be modified – even for assembly level trimming. When trimming if type-forwarder assemblies are eliminated, the remaining assemblies need to be modified to redirect the forwards. When an assembly is re-written due to trimming, the pre-built R2R data is removed, thereby optimizing size over performance.

Using <TrimMode>Link</TrimMode> will remove R2R data, unless its specified to be added by using the <PublishReadyToRun>True</PublishReadyToRun> option as part of dotnet publish.

The relative sizes for the console app template with different modes of trimming, with and without R2R.

Trim Mode	Self Contained Linux x64 (MB)	Single File Linux x64 (MB)	Self Contained Win10 x64 (MB)	Single File Win10 x64 (MB)
No Trimming	78.9	65.3	64.1	57.6
Assembly Trimming	39.7	26.1	24.4	17.9
Assembly Trimming + R2R	41.7	28.1	26.4	19.8
Member Trimming	31.5	17.9	16.2	9.7
Member Trimming + R2R	34.5	20.8	19.4	12.9

As the application author, you should decide whether you want to favor size versus startup performance.

## Making .NET Trimmable

.NET has a long history, and dynamic code is prevalent throughout the framework and library ecosystem. The exercise to annotate the .NET libraries has begun, but will probably take multiple releases to complete – we are prioritizing based on usage.

In addition to using attribution, we are looking at using [source generators](#) to move functionality from runtime reflection to build-time code generation. This not only improves performance, but means that the generated code doesn't use reflection so the







trimmer can walk it to determine what needs to be kept. Good examples of where source generators can be used to aid trimming include:

- Dependency Injection (DI) – the source generator can figure out which types implement the given interfaces, and generate static code that does the hookup rather than relying on reflection at runtime.
- Object serialization – serializers typically use reflection to walk the properties and fields of objects to be serialized, examining annotations if necessary. Sometimes they even do dynamic method creation at runtime to improve the performance of many serializations. A source generator can create the equivalent code at build time, which means less runtime work, and it's linker friendly.

The library ecosystem, such as nuget package authors, will also have a role to play in making their assemblies trimmable. We'll go into more details in a subsequent post.



## Removing Framework Features

One of the differences between the Mono and .NET Core frameworks is that Mono was designed for mobile applications where package size is a concern; .NET framework is fuller featured and so larger. In the move to consolidate the two we needed a mechanism to retain the size benefits of Mono while using the larger framework.

One mechanism we are adding that will allow us to conditionally remove code from applications is [feature switches](#). Using this mechanism, an SDK (like Xamarin) or the developer can decide if a feature that would normally be available in the libraries can be removed, both at runtime and during linking. An example of an existing feature switch we have today is [InvariantGlobalization](#). We will add more switches to the libraries to allow large, optional pieces of code to be removed by the trimmer.

## Blazor Web Assemblies

One of the biggest needs for size reduction is in the context of Blazor web assemblies. The [3.2 release of Blazor](#) in May was based on the Mono framework. The .NET 5 release will use the .NET 5 framework, which is larger, and we didn't want to regress the size of Blazor Apps.

The following trimming approach is being taken for Blazor Apps in .NET 5:

- Assemblies from the shared framework/runtimepack get member-level trimming
- Microsoft.Extensions.\* assemblies get type-level trimming from TrimMode=Link
- Assemblies from Microsoft.AspNetCore.\* are trimmed via hand generated XML file
- All other assemblies are not trimmed

Blazor apps are also configured to disable the following framework features:

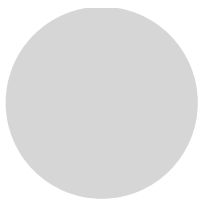
- EventSourceSupport
- EnableUnsafeUTF7Encoding
- HttpActivityPropagationSupport
- DebuggerSupport (for retail builds)

And enable:

- UseSystemResourceKeys

This results in Blazor apps with comparable sizes while using the fuller framework of .NET 5.

In the next post we will talk about how you can use attributes and xml files to further tweak the trimming operation.



Posted in .NET

Read next  
f

Twitter  
in

## Improvements in native code interop in .NET 5.0

In this post, we discuss interop improvements in .NET 5.0 and describe some of the work we are considering for the future. We also have a survey on GitHub, where we hope to hear about your experiences in the interop space.

Elinor Fung  
September 1, 2020

5 comments

## ARM64 Performance in .NET 5

ARM64 performance work in .NET 5

Kunal Pathak  
September 2, 2020

13 comments



## 29 comments

Comments are closed. [Login to edit/delete your existing comments](#)

1 2 >

James Wil August 31, 2020 5:11 pm 0



Wow the results are impressive!

JinShil August 31, 2020 7:05 pm 0



```
<RuntimeIdentifier>win10-x64</RuntimeIdentifier>  
<SelfContained>true</SelfContained>  
<PublishTrimmed>true</PublishTrimmed>  
<TrimMode>Link</TrimMode>  
<PublishReadyToRun>True</PublishReadyToRun>
```

This results in...

```
Application startup exception  
System.MissingMethodException: No parameterless constructor defined for type  
'System.ComponentModel.StringConverter'.
```

... for my ASP.Net hosted Blazor WASM app.

Changing to <TrimMode>CopyUsed</TrimMode> works fine.

Sam Spencer September 1, 2020 12:24 pm 0



This is a case where its probably being invoked via reflection – DI? The next post on trimming (almost ready) covers how to use attributes and/or xml to state the dependency. You could also explicitly reference from code, and that should force it to be included.

Feedback



**Eric Erhardt** September 1, 2020 12:29 pm 0



What version of .NET 5 are you using? We made ComponentModel [TypeConverters linker-safe](#) in .NET 5 RC1.



**JinShil** September 1, 2020 5:59 pm 0



I'm testing with 5.0.100-preview.8.20417.9. I'd like to test with .Net 5 RC1, but I don't see it at <https://dotnet.microsoft.com/download/dotnet/5.0>



**Daniel Hughes** August 31, 2020 9:38 pm 0



Why are the linux results almost twice the size of the windows ones?



**Daniel Hughes** August 31, 2020 9:54 pm 0



We shouldn't need to choose between startup time and size. CoreRT has proven you can have both. We just need to convince Microsoft to invest in it.



**Paulo Pinto** August 31, 2020 11:20 pm 0



Hopefully that will come with .NET 6.

Otherwise the best way to convince them is to jump to other managed languages with a better AOT story, eventually management will notice where everyone is going, like they had to do with WSL and Android (Surface Duo).



**James Wil** September 1, 2020 1:27 am 0



I agree with you



**Zero** September 1, 2020 2:47 am 0



I think just about every single recent .NET blog post has had a "give us CoreRT" comment under it, hah. I hope they'll listen.



**John Tur** September 1, 2020 9:00 pm 0



The CoreRT folks are a vocal minority. I do not think they understand how poorly it works for many real-world apps. They tried hard to roll this out in the form of .NET Native for Store apps. It was theoretically the perfect match — store apps ran in a constrained environment without things like dynamic assembly loading, which hid many of the weaknesses of .NET Native. Store apps also had the brand new Windows.\* API surface, in addition to a significantly slimmed down System.\* API surface, both of which helped mask/avoid issues that would make .NET Native fall over for brownfield apps. Despite all of the significant investment in making .NET Native fit in well with .NET, people still hated it. The blog post even mentions this. And yet, the same people who are championing CoreRT as to the solution to all of .NET's problems will be the same ones who complain that trimming is a broken feature that only works well in theory.



Complaining in the comments about how the .NET team is stupid for doing ‘the wrong thing’ does not help fix any of the problems that make Native AOT difficult. On the other hand, this new trimming functionality is really a *massive step forward* (which .NET will need many more of) for Native AOT. CoreRT and .NET Native currently rely on package authors to hand-write cumbersome rd.xml files to make sure their apps don’t break after trimming. However, library developers ended up just passing the buck onto app authors to reverse-engineer the ways in which their library would break after trimming. Even with .NET Native’s complex static analysis, Microsoft had to cut trimming for non-system assemblies from .NET Native’s initial release. However, now that enabling trimming, as well as making libraries linker-safe, is so much more accessible in .NET 5, this feature really clears out more of the path for Native AOT to be more successful in .NET’s future. Bringing CoreRT to everyone tomorrow would be a disaster. It will take significant time and lots of re-engineering in order to make it turn out well.



**Zero** September 3, 2020 7:58 am 0



I don’t think the people championing CoreRT (myself included) are claiming that the *current state of it* is better than anything else – rather, it’s about asking for resources to be allocated to its development, precisely so that it CAN become a viable option to use, instead of being kept in a perpetual alpha for the next 5 years and then scrapped. The “championing” imo is the result of that uncertainty. Instead of native AOT being the next killer feature on the roadmap, coming in .NET 8 2023 once the prerequisites for it are done, current communication suggests that no one, Microsoft and the .NET team included, cares about it too much. Fill out this survey and we might revisit the issue one day (though credit where due, at least there IS a survey).

In an ideal world, Mono, .NET Native, UWP, IL2CPP and recent versions of the .NET framework wouldn’t have started development at all, and all of that work would’ve been spent on .NET Core and CoreRT. If it takes 10 more years of herculean effort for .NET to have AOT compilation the same way it took 10+ years for it to become cross platform, and the entirety of the .NET/C# ecosystem must be refactored first in order to make it happen, that is perfectly fine and more than understandable. I just hope that we’ll eventually get there.



**Timothy Ojo** September 22, 2020 2:13 pm 0



A company with the resources of Microsoft is not expected to be giving excuses why something can’t work. Are we suppose to applaud them for failing at what other relatively new programming languages can do?

C# is a beautiful language, powerful and easy to learn. It should not be crippled by a framework. Forget the framework and just work on a solution that compiles the language to native.



**Daniel Frank** September 1, 2020 2:48 am 0



It’s possible to use ilinker in framework dependant applications?



**Max Mustermueller** September 1, 2020 6:49 am 0





My experience with the trimming feature is that it only really works in very simple hello world applications. In bigger applications it almost immediately results in exceptions on application start, IF you get it to build, but most of the time, the trimming even fails on build.

Also, the size demonstrations are impressive but only for a console application. Try the same with WPF and you will get much, much bigger application sizes with much much less difference to non-trimmed.

Another reason why I consider this as a high experimental feature and rather looking forward for AOT, as in AOT only the code which is actually needed is integrated. At least on proper AOT solutions like Dephi.




[Sam Spencer](#)  September 1, 2020 12:32 pm  0



AOT has the same problem as trimming in knowing which members are pulled in from dynamic code. There is a lot of dynamic code sprinkled throughout .NET – DI, IoC, WPF data bindings, object serialization etc. This was the main cause of problems with .NET Native – it was trying to magic its way through reflection scenarios trying to understand the code paths. The analysis patterns and attribution we are doing for the framework for trimming will also be usable for AOT scenarios.



[MgSam](#) September 1, 2020 9:38 am  0




I've had extremely unreliable results with the trimming feature so far. I shipped a self-contained, trimmed application which ran fine on my local machine, and ran fine on most machines it was deployed to, but failed to start on one specific machine. All machines were using Windows 10 and the had the .NET Core runtimes installed (although that shouldn't have even been necessary). Disabled the trimming and the app worked fine everywhere.

Solution- don't use trim every again. It's really not worth saving a few MBs at the cost of an unreliable application.

I sympathize with the Blazor/mobile app conundrum here. Hopefully you guys can eventually make something that works really well.





[Mike Perez](#) September 1, 2020 12:05 pm  0



<

p>I hope .net 5 makes it easier to identify what needs to be included when trimming takes something it shouldnt. I have a wpf app and it seems to shrink the wpf assemblies (especially presentationcore), but the error messages are so deeply nested in the exceptions it's not really possible to tell what I am supposed to include. I was willing to try and figure it out, but I didn't even know where or what to try include.




[Sam Spencer](#)  September 1, 2020 12:40 pm  0



Trimming is a journey – we need to work through the framework to annotate all the dynamic cases, and that is going to take some time. We started with Blazor as the scenario we focused on as size concerns are significant in the WASM environment.




[toolgood@qq.com](#) September 21, 2020 12:19 am  0



How to package a single file after trimming and obscure.




[Ramon de Klein](#) October 23, 2020 3:28 pm  0



Isn't the development for small ARM devices a much bigger use-case compared to Blazor that is hardly used in production? .NET Core is great, but performance on low-end ARM processors is slow and file sizes are huge. We need good trimming for these platforms to make .NET Core a success for IOT devices.



[ALIEN Quake](#) September 1, 2020 3:33 pm  0



42 MB – that's how small the Net 5.0 Preview 8 Avalonia self-contained app is! Many thanks for the improvements!



Why is Blazor trimming limited to System and Microsoft assemblies? As the author of Bolero, being forced to serve the full FSharp.Core.dll is... painful :/



.NET Feature Blogs

- [.NET MAUI](#)
- [ASP.NET Core](#)
- [Blazor](#)
- [Entity Framework](#)
- [ML.NET](#)
- [NuGet](#)
- [Xamarin](#)

Languages

- [C#](#)
- [F#](#)
- [Visual Basic](#)



Archive

- [March 2023](#)
- [February 2023](#)
- [January 2023](#)
- [December 2022](#)
- [November 2022](#)
- [October 2022](#)
- [September 2022](#)
- [August 2022](#)
- [July 2022](#)
- [June 2022](#)
- [May 2022](#)

More .NET

- [Download .NET](#)
- [.NET Community](#)
- [.NET Documentation](#)
- [.NET API Browser](#)

Learn

- [.NET Learning Hub](#)
- [Architecture Guidance](#)
- [Beginner Videos](#)
- [Customer Showcase](#)

Stay informed



What's new

- Surface Pro 9
- Surface Laptop 5
- Surface Studio 2+
- Surface Laptop Go 2
- Surface Laptop Studio
- Surface Go 3
- Microsoft 365
- Windows 11 apps

Microsoft Store

- Account profile
- Download Center
- Microsoft Store support
- Returns
- Order tracking
- Virtual workshops and training
- Microsoft Store Promise
- Flexible Payments

Education

- Microsoft in education
- Devices for education
- Microsoft Teams for Education
- Microsoft 365 Education
- Education consultation appointment
- Educator training and development
- Deals for students and parents
- Azure for students

Business

- Microsoft Cloud
- Microsoft Security
- Dynamics 365
- Microsoft 365
- Microsoft Power Platform
- Microsoft Teams
- Microsoft Industry
- Small Business

Developer & IT

- Azure
- Developer Center
- Documentation
- Microsoft Learn
- Microsoft Tech Community
- Azure Marketplace
- AppSource
- Visual Studio

Company

- Careers
- About Microsoft
- Company news
- Privacy at Microsoft
- Investors
- Diversity and inclusion
- Accessibility
- Sustainability

