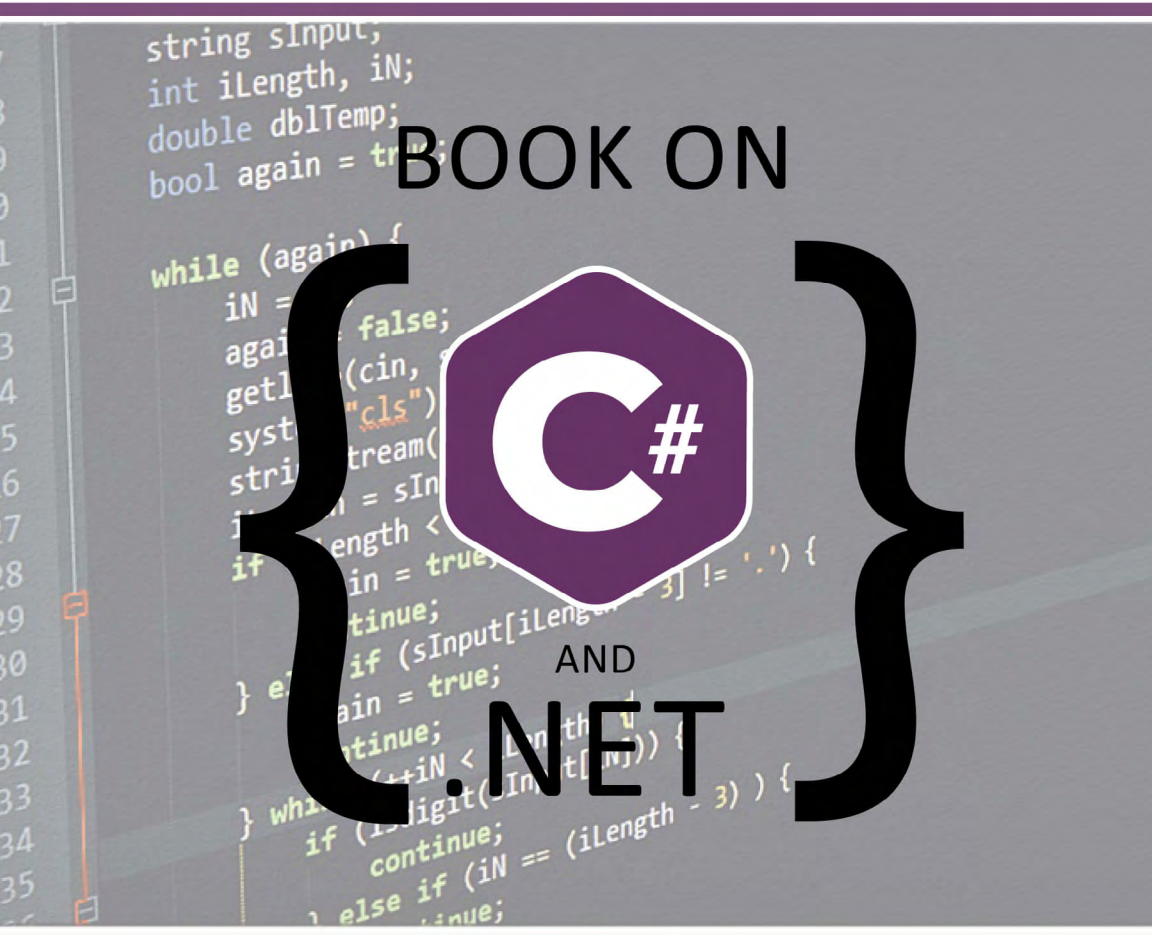
COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY AWESOME BOOK ON { C# AND .NET }



DAMIR ARH

The Absolutely Awesome Book on

C# and .NET

(Covers C# v6, v7, and .NET Core)

By Damir Arh

The Absolutely Awesome Book on C# and .NET

Bulk Sales

DotNetCurry.com offers bulk discount on this EBook for libraries and companies.

For more information, please contact A2Z Knowledge Visuals Pvt. Ltd at

ebooks@a2zknowledgevisuals.com

Cover Image By *Minal Agarwal* (minalagarwal@a2zknowledgevisuals.com).

Published in 2019 by A2Z Knowledge Visuals Pvt Ltd.

Cover created by **Minal Agarwal**

EBook created by **Minal Agarwal**

EBook Conceptualization by **Suprotim Agarwal**

Editing and Proofreading by **Suprotim Agarwal**

EBook Production and Distribution by *DotNetCurry.com* (A subsidary of A2Z Knowledge Visuals Pvt Ltd)

Written by Damir Arh and Reviewed by Yacoub Massad

About The Author

Damir Arh has many years of experience with software development and maintenance; from complex enterprise software projects, to modern consumer oriented mobile applications. Although he has worked with a wide spectrum of different languages, his favourite language remains C#. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, and by blogging, and writing articles.

He has received the prestigious Microsoft MVP award for developer technologies for 7 times in a row.

In his spare time, he's always on the move: hiking, geocaching, running, and sport climbing. You can follow him on twitter @damirarh or read his tutorials at www.dotnetcurry.com/author/damir-arh

About The Reviewer

Yacoub Massad is a software developer and architect who works primarily on Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software.

You can follow him on twitter @yacoubmassad and view his blog posts at criticalsoftwareblog.com.

Contents

SECTION I

.NET AND THE COMMON LANGUAGE RUNTIME (CLR)

SECTION II

THE TYPE SYSTEM

SECTION III

CLASSES AND INHERITANCE

SECTION VI

LANGUAGE INTEGRATED QUERY (LINQ)

SECTION VII

PARALLEL AND ASYNCHRONOUS PROGRAMMING

SECTION VIII

SERIALIZATION & REFLECTION

SECTION IX

C# 6 AND 7

SECTION X

A LOOK INTO THE FUTURE

SECTION I

.NET and the Common Language Runtime (CLR)

3

Q3. What is .NET Standard and why does it matter to me?

Since the original release of the .NET framework in 2002, the .NET ecosystem expanded with many alternative .NET runtimes: .NET Core, Universal Windows Platform (UWP), Xamarin, Unity Game Engine and others. Although the runtimes are similar to each other, there are some differences between their base class libraries (BCL), which makes it difficult to share code between projects targeting different runtimes.

Microsoft's first attempt at solving this problem were Portable Class Libraries (PCL), which allowed building assemblies that could be used with multiple runtimes without recompiling. Portable Class Libraries require the target runtimes to be selected when creating the library as based on this choice, the intersection of available APIs is calculated.

With an increasing number of runtimes and different versions, this approach was becoming ever more difficult to manage. Hence, Microsoft introduced .NET Standard as an alternative for easier cross-platform development. .NET Standard takes a reversed approach to determining the available set of APIs. Instead of calculating them based

on the targeted runtimes, it explicitly specifies the APIs which must be supported by a compatible runtime. Several different versions of .NET Standard are defined, each with an increasing number of supported APIs.

Any class library author can now select a .NET Standard version to target their library. A library targeting a specific .NET Standard version will work on any platform implementing this version of the standard. If a platform is updated to support a higher version of .NET Standard or if a new platform is released, all existing libraries targeting the newly supported version of .NET Standard will automatically work with it with no changes required by their respective authors.

Different versions of existing runtimes support different versions of .NET Standard.

Target Platform	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
UWP	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299
Windows Store	8.0	8.0	8.1					
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

Table 1: Supported runtimes for different .NET Standard versions (version 10.0.16299 of UWP was shipped as part of Windows 10 Fall Creators Update in October 2017)

What might not be obvious from the table is the big difference between .NET Standard 1.x and .NET Standard 2.0.

Compared to .NET Standard 1.6, .NET Standard 2.0 includes over *20,000 additional APIs*. These are APIs that are already a part of the .NET framework, but have been missing from .NET Standard before.

Since .NET Standard is focused on cross-platform compatibility, it still doesn't include all the .NET framework APIs, and it never will. By design, it excludes all application model APIs, such as WPF, Windows Forms, ASP.NET and others. It also excludes all Windows specific APIs that were included in the .NET framework, e.g. APIs for working

with the registry and event logs. Most of the .NET framework APIs that do not fall in these two categories are included in .NET Standard 2.0.

Some of the most notable additions that were missing from .NET Standard 1.6 are:

- System.Data namespace with DataSet, DataTable, DataColumn and other classes for local data manipulation that were commonly used in earlier versions of the .NET framework before the release of Entity Framework.
- XmlDocument and XPath based XML parsers in addition to XDocument, which was the only one available before .NET Standard 2.0.
- System.Net.Mail, System.Net.WebSockets and other previously missing network related APIs.
- Low level threading APIs such as Thread and ThreadPool.

There's also a namespace that's not in .NET Standard, but is worth mentioning as it might come as a surprise to most developers: **System.Drawing**.

The reason this namespace is not included in .NET Standard is because it is just a thin layer over the native GDI+ Windows component and therefore can't be easily ported to other runtimes. This is unfortunate as it is probably the most commonly used .NET framework API that's still missing from .NET Standard. There are currently no plans to include them in a future version of .NET Standard.

Although the System.Drawing APIs can't be used on all .NET Standard compliant runtime, they are made available to .NET Core as part of the Windows Compatibility Pack. Despite the name of this pack, many of its APIs aren't just supported on Windows, but on other platforms too. APIs from the System.Drawing namespace count among them and can also be used on Linux and macOS since .NET Core 2.1.

The increased API set in .NET Standard 2.0 makes it much easier to compile existing source code that was originally written for the .NET framework, and make it available to other runtimes. At least as long as it is not dependent on particular application models. This means that business logic could be shared across runtimes, but applications will still need to be written for each runtime.

However, our own source code, even with business logic, often depends on other third-party libraries, in addition to the base class library APIs that are now exposed via .NET Standard. As expected, .NET Standard libraries can reference other .NET Standard

libraries that target the same or lower version. They can also reference compatible portable class libraries.

Unfortunately, a large majority of existing third-party libraries do not target .NET Standard or PCL, but the .NET framework. Although many of those might be recompiled for .NET Standard with minimal or no code changes at all, this still needs to be done by their authors who might not actively maintain them anymore.

To avoid that requirement altogether, .NET Standard 2.0 includes a *special compatibility shim*, which makes it possible to reference existing .NET framework libraries from .NET Standard libraries, without recompilation. To allow that, any references to the .NET framework classes from the referenced .NET framework library are type forwarded to their counterparts in .NET Standard. Hence, the compiled code will also work on other runtimes, not only on the .NET framework.

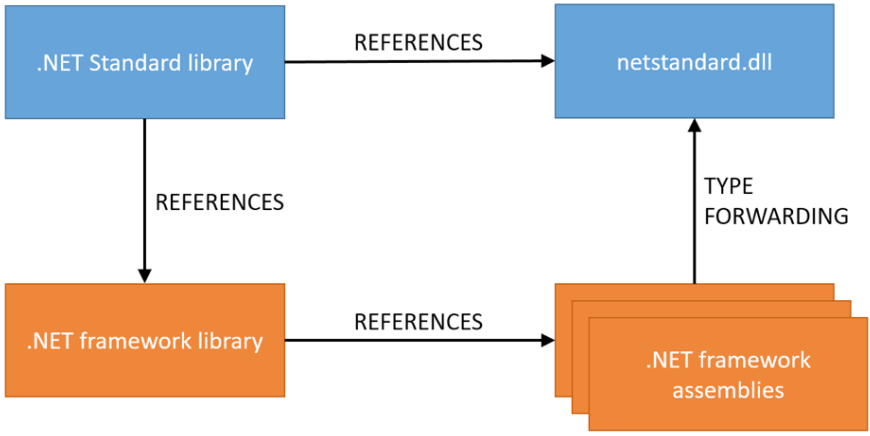


Figure 1: Type forwarding for referenced .NET framework libraries

Of course, this doesn't mean that any .NET framework library will just work with .NET Standard. If it uses any APIs that are missing from .NET standard, the call will fail.

To estimate the severity of this issue, Microsoft did an analysis of all the libraries available on NuGet before releasing .NET Standard 2.0 and published the results in a video by Immo Landwerth. According to it, over two-thirds of libraries were API compatible with .NET Standard 2.0, i.e. they either targeted .NET Standard or PCL, or they targeted the .NET framework but only used APIs that are part of .NET Standard 2.0.

Almost two thirds of those that were incompatible, depended on APIs in specific application models (WPF, ASP.NET, etc.) and were therefore not appropriate for use

from .NET Standard. The rest mostly used APIs that were already considered for .NET Standard 2.0 but were excluded as 'too hard' to implement across all runtimes. It is therefore not expected that compatibility will significantly increase with future .NET Standard versions.

Compatible packages (68 %)		**Incompatible packages (32 %)**	
.NET framework (56 %)	PCL, .NET Standard (12 %)	Unsupported application models (20 %)	Missing APIs (12 %)

Table 2: Assembly compatibility in NuGet

A small number of API compatible libraries from the above analysis (less than 7%) used P/Invoke to call into native libraries. Although these assemblies can be used from .NET Standard, they will only work on Windows (when referenced from the .NET framework or .NET Core applications) but will fail on other operating systems because they depend on native binaries.

6

Q6. How did the .NET Framework evolve since version 1.0?

It's been over 15 years since the original release of .NET framework 1.0 and looking at it today, it seems much less impressive than it did. Most of the features we are used to, were not available back then and we would have had a hard time if we wanted to use it for development of modern applications today.

This is understandably the case with any framework. To keep up with new technology advancements, development frameworks have to evolve too. The .NET Framework has evolved and grown to keep up with these needs.

Let us take an overview of this evolution.

CLR version	.NET framework version	Most important changes
1.0	1.0	Initial release: Console Applications, Windows Forms, Windows Services, Web Forms, Web Services
1.1	1.1	IPv6, ASP.NET mobile controls, ADO.NET ODBC
2.0	2.0	Generics, edit and continue, 64-bit runtime
	3.0	WPF, WCF, WF
	3.5	LINQ, ASP.NET AJAX, WF services, cryptography
	3.5 SP1	.NET Client Profile, EF
4	4	DLR, PCL, TAP, ASP.NET MVC
	4.5	Async I/O, ASP.NET Web API, ASP.NET Web Pages
	4.5+	64-bit edit and continue, async debugging, HTTP/2, 64-bit JIT

Table 1: .NET framework version history

.NET framework 1.0 included a limited set of application frameworks, and supported the following types of applications:

- Console and Windows Forms applications for the desktop
- Windows Services
- ASP.NET Web Forms based web applications, and
- Web services using the SOAP protocol.

.NET framework 1.1 didn't bring a lot of changes. Although it featured a new version of the CLR, it was to enable side-by-side installation of both versions (v1.0 and v1.1) on the same computer. The most notable new features were support for IPv6, mobile controls for ASP.NET Web Forms and an out-of-the-box ADO.NET data provider for ODBC. Unless you had a specific need for any of these features, there was no reason convincing enough to migrate to the new version.

.NET framework 2.0 was a different story altogether. It fixed the weaknesses identified in the previous two versions and introduced many new features to boost developer productivity. Let's name only the ones with the biggest impact:

- The introduction of **generics** allowed creation of templated data structures that could be used with any data type while still preserving full type safety. This was most widely used in collection classes, such as lists, stacks, dictionaries, etc.

Before generics, all data types stored in the provided collection classes were treated as instances of System.Object. After retrieving them from the collection, the code had to cast them back to the correct type. Alternatively, custom collection classes had to be derived for each data type. Generics weren't limited to collections though. Nullable value types were also relying on them.

- Improved debugging allowed **edit and continue** functionality: when a running application was paused in the debugger, code could be modified on the fly in Visual Studio and the execution continued using the modified code. It was a very important feature for existing Visual Basic 6 programmers who were used to an equivalent functionality in their development environment.

- **64-bit runtime** was added. At the time, only Windows Server had a 64-bit edition, and the feature was mostly targeted at server applications with high memory requirements. Today, most of Windows installations are 64-bit and many .NET applications run in 64-bit mode without us even realizing it.

All of these, along with countless other smaller improvements to the runtime and the class library made .NET framework 2.0 much more attractive to Windows developers who were using other development tools. It was enough to make .NET framework, the de facto standard for Windows development.

.NET framework 3.0 was the first version of the .NET framework which couldn't be installed side-by-side with the previous version because it was still using the same runtime (CLR). It only added new application frameworks:

- **Windows Presentation Foundation (WPF)** was the alternative to Windows Forms for desktop application development. Improved binding enabled better architectural patterns, such as MVVC. With extensive theming support, application developers could finally move beyond the classic *battleship-gray* appearance of their applications.

- **Windows Communication Foundation (WCF)** was a new flexible programming model for development of SOAP compliant web services. Its configuration options made it possible to fully decouple the web service implementation from the transport used. It still features one of the most complete implementations of the WS-* standards on any platform, but is losing its importance with the decline of SOAP web services in favor of REST services.

- **Windows Workflow Foundation (WF)** was a new programming model for long

running business processes. The business logic could be defined using a graphical designer by composing existing blocks of code into a diagram describing their order of execution based on runtime conditions.

While WF wasn't widely used, we can hardly imagine the .NET framework without WCF and WPF (and other XAML based UI framework derived from it, such as Silverlight and UWP).

.NET framework 3.5 continued the trend of keeping the same runtime and expanding the class library. The most notable addition was **Language Integrated Query (LINQ**). It was a more declarative (or functional) approach to manipulating collections. It was inspired by SQL (Structured Query Language) widely used in relational databases. Thanks to the updates to C# and Visual Basic released during the same time, there was even a query syntax available as an alternative to the more conventional fluent API. By automatically converting the query into a tree-like data structure, it opened the doors to many LINQ providers, which could execute these queries differently based on the underlying data structure. Three providers were included out-of-the box:

- LINQ to Objects performs the operations on in-memory collections.
- LINQ to XML is used for querying XML DOM (document object model).
- LINQ to SQL is a simple object-relational mapping (ORM) framework for querying relational databases.

Web related technologies also evolved with this version of the .NET framework:

- **ASP.NET AJAX library** introduced the possibility of partially updating Web Forms based pages without full callbacks to the server by retrieving the updates with JavaScript code in the page. It was the first small step towards what we know today as single page applications (SPA).
- **WF services** exposed WF applications as web services by integrating WF with WCF.
- Managed support for **cryptography** was expanded with additional algorithms: AES, SHA and elliptic curve algorithms.

Looking at the name of .**NET framework 3.5 SP1**, one would not expect it to include features, but at least two of them are worth mentioning:

- **Entity Framework (EF)** was a fully featured ORM framework that quickly replaced

LINQ to SQL. It also heavily relied on LINQ for querying.

- **.NET Client Profile** was an alternative redistribution package for the .NET framework which only included the parts required for client-side applications. It was introduced as the means to combat the growing size of the full .NET framework, and to speed up the installation process on computers which did not need to run server-side applications.

.NET framework 4 was the next framework that included a new version of the CLR and could therefore be installed side-by-side with the previous version. The most notable new additions were:

- **Dynamic Language Runtime (DLR)** with improved support for late binding. This simplified certain COM interop scenarios and made it easier to port dynamic languages to the CLR. IronRuby and IronPython (Ruby and Python ports for the .NET framework) took advantage of that.

- **Portable Class Libraries (PCL)** were introduced for creating binary compatible assemblies that could run on different runtimes (not only the .NET framework, but also Windows Phone and Silverlight).

- **Task-based Asynchronous Pattern (TAP)** was a new abstraction for writing multithreaded and asynchronous applications which hid many of the error-prone details from the programmers.

- **ASP.NET MVC** was an alternative programming model to Web Forms for creating web applications. As the name implies, it implemented the model view controller (MVC) pattern which was gaining popularity on other platforms. It was first released out-of-band, but .NET framework 4 was the first version to include it out-of-the-box.

.NET framework 4.5 was the last major new release of .NET framework with several new important features:

- Support for **asynchronous I/O operations** was added as the new async and await syntax in C# made it much easier to consume them.

- **ASP.NET Web API** and **ASP.NET** Web Pages were included in the framework for the first time after they were originally released out-of-band. They allowed development of REST services and provided Razor, an alternative syntax for ASP.NET MVC views, respectively.

- PCL was expanded to support **Windows Store** applications for Windows 8.

Also, the .NET Client Profile was discontinued as optimizations to the size and deployment process of .NET framework made it obsolete.

Since .NET framework 4.5, the release cadence increased. The version numbers (4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2) reflected that. Each new version brought bug fixes and a few new features. The more important ones since .NET framework 4.5 were:

- Edit and continue support for 64-bit applications.
- Improved debugging for asynchronous code.
- Unification of different ASP.NET APIs (ASP.NET MVC, ASP.NET Web API and ASP.NET Web Pages).
- Support for HTTP/2.
- Improved 64-bit just-in-time (JIT) compiler for faster startup time.

The focus of new development has now shifted to .NET Core, therefore new major improvements to the .NET framework are not very likely. We can probably still expect minor releases with bug fixes and smaller improvements though.

10

Q10. What should I know about Garbage Collection in .NET?

Garbage collection makes automatic memory management in .NET framework possible. Thanks to it, the developer is only responsible for allocating memory by creating new instances of objects. Allocated memory is automatically released by the garbage collector once the created objects are not used any more.

All managed objects in the .NET framework are allocated on the managed heap. They are placed contiguously as they are created. If this process would continue like that for a long time, we would eventually run out of memory. To prevent that, garbage collection gets triggered when memory allocations reach a certain total size or when there's a lack of available memory.

The garbage collection consists of three phases:

- In the **marking phase**, a list of all objects in use is created by following the references from all the root objects, i.e. variables on stack and static objects. Any allocated objects not on the list become candidates to be deleted from the heap. The objects on the list will be relocated in the compacting phase if necessary, to

compact the heap and remove any unused memory between them.

- In the **relocation phase**, all references to remaining objects are updated to point to the new location of objects to which they will be relocated in the next phase.

- In the **compacting phase**, the heap finally gets compacted as all the objects that are not in use any more are released and the remaining objects are moved accordingly. The order of remaining objects in the heap stays intact.

To better handle different lifetimes of different objects, garbage collection introduces the concept of **generations**. A newly created object is placed in generation 0. If it is not released during a garbage collection run, it is moved to generation 1. If it is still not released when the next garbage collection run happens, it is moved to generation 2. It stays there for the remainder of its lifetime.

Based on the assumption that most objects are only used for a short time, while the objects that are used longer are less likely to be released soon if at all, the generations are used to optimize the garbage collection process. Most garbage collection runs will only process generation 0 and will therefore take the shortest time. Once more memory needs to be released, generation 1 will be included in garbage collection as well, making it take more time. When even generation 1 garbage collection doesn't release enough memory, generation 2 gets included as well. This is called full garbage collection and takes the longest time.

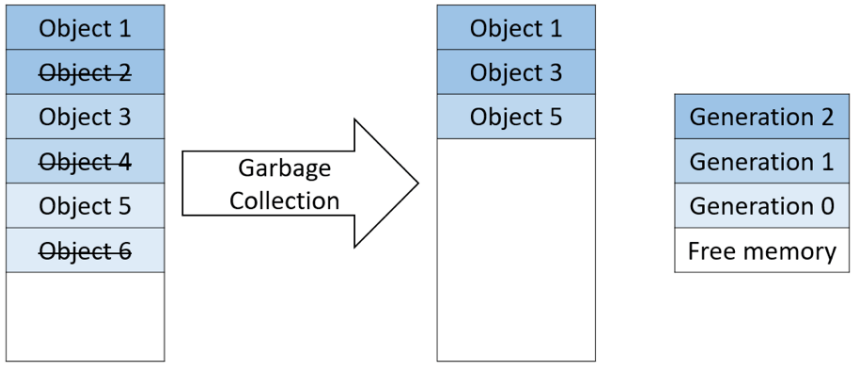


Figure 1: Garbage collection releases unused objects and promotes used objects through generations

To further optimize garbage collection, there is a separate object heap for objects larger than 85,000 bytes. Objects placed in it are handled differently:

- New objects are immediately put into generation 2, since larger objects (usually arrays) tend to live longer.

- Objects are not moved at the end of garbage collection to compact the large objects heap because moving larger objects would be more time consuming.

There are two different types of garbage collection:

- Workstation garbage collection runs on the normal priority thread, which triggered it by passing a memory threshold with its latest memory allocation or by explicitly calling `GC.Collect`

- Server garbage collection creates a separate managed heap and a corresponding garbage collection thread for each logical CPU. Threads run on highest priority. This approach to garbage collection makes it faster but more resource intensive.

By default, workstation garbage collection is used, but this can be changed with a setting in the application configuration XML file (named AssemblyName.config):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <gcServer enabled="true"></gcServer>
  </runtime>
</configuration>
```

If the computer only has a single logical CPU, the setting will be ignored and the workstation garbage collection type will be used.

Each approach to garbage collection is available in three different variations:

- **Non-concurrent garbage collection** suspends all non-garbage-collection threads for the full duration of the garbage collection, effectively pausing the application for that time.

- **Concurrent garbage collection** allows user threads to run for the most of generation 2 garbage collection. As long as there is still free space in the managed heap for new allocations, user threads are allowed to run and create new objects. This results in a shorter garbage collection pause, at the cost of higher CPU and memory requirements.

- **Background garbage collection** is the replacement for concurrent garbage collection. It was introduced in .NET framework 4 for workstation garbage collection, and in .NET framework 4.5 for server garbage collection. It is similar to concurrent garbage collection but allows generation 0 and generation 1 garbage collection to interrupt an ongoing generation 2 garbage collection and temporarily block program execution. After generation 0 and generation 1 garbage collection is completed, both the program execution as well as the generation 2 garbage collection, continues. This even further shortens the time the program execution is paused because of garbage collection.

By default, concurrent or background garbage collection will be used (depending on the .NET framework version), but this can be changed with a setting in the application configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <gcConcurrent enabled="false"></gcConcurrent>
  </runtime>
</configuration>
```

Garbage Collection in .NET Core

The same two garbage collection configuration options are available .NET Core. Since there's no application configuration XML file in .NET Core, the settings are placed in the runtime configuration JSON file (named AssemblyName.runtimeconfig.json) instead:

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Concurrent": false,
      "System.GC.Server": true
    }
  }
}
```

Typically, this file is not edited manually. Its contents are generated at compile time based on the MSBuild project (.csproj) file. To configure the garbage collector, you can

set the following properties in the project file:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
  <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>
```

Garbage Collection - Best Practices

Taking all this knowledge into account, application performance can be improved by following these recommendations when garbage collection is identified as the cause for bad performance:

- When a multithreaded application creating a lot of objects is running on a computer which doesn't host many other processes, switch from workstation to server garbage collection. The application will benefit from shorter pauses while not negatively affecting other applications with high priority garbage collection threads.

- Although garbage collection can be triggered manually by calling `GC.Collect`, it should usually be avoided as the CLR heuristics will be able to better schedule garbage collection based on all the information available. Probably the only use case when manual garbage collection could make sense would be when in a short time, a lot of objects were not in use, and the application can afford a deterministic garbage collection pause without negatively affecting the user.

- Allocation of large objects with short life time should be avoided as much as possible. It will take a long time before they are released because they are immediately placed in generation 2 and they will make the large object heap fragmented because it is not compacted.

- If possible, avoid allocating large numbers of objects with short lifetimes and try to reuse the objects instead. This will avoid frequent triggering of generation 0 garbage collection, which causes pauses in program execution.

SECTION II

THE TYPE SYSTEM

17

Q17. What are some of the less commonly used C# operators?

There's a large collection of operators available in C#. Many are common to most programming languages and we use them regularly, e.g. arithmetic, relational, logical, and equality operators. Others might not be as generally useful, but can make your code simpler and more efficient in specific scenarios.

The conditional operator is a great replacement for simple if-else statements used only to assign one or the other value to the same variable.

```
if (compactMode)
{
  itemsPerRow = 5;
}
else
{
  itemsPerRow = 3;
}
```

The same result can be achieved using a single expression with the conditional operator.

```
itemsPerRow = compactMode ? 5 : 3;
```

This operator is especially convenient when used with simple expressions. Checking if a value is null would be another good example:

```
middleName = middleName != null ? middleName : "";
```

In this special case, the null-coalescing operator can be used instead.

```
middleName = middleName ?? "";
```

Unfortunately, it only works when we want to directly use the value being checked for null. If we want to access one of its members, we still need the conditional operator.

```
count = list != null ? list.Count : 0;
```

To simplify such cases, the null conditional operator was introduced in C# 6.

```
count = list?.Count ?? 0;
```

It allows safe member access. If the left-hand side value is non-null, it accesses the member. Otherwise it just returns `null` avoiding the NullReferenceException which would be thrown when trying to access a member of a null-valued variable.

Its usage is not restricted to properties. It also works with fields, methods and even index access.

```
int? firstValue = list?[0];
```

The null conditional operator is also a great choice for thread-safe invoking of event handlers.

In earlier versions of C#, the event handler had to be stored in a local variable to be safe in multi-threaded scenarios. Otherwise its value could be changed by another thread after it was checked for null but before it was invoked, causing a NullReferenceException.

```
public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(string propertyName)
{
  var handler = PropertyChanged;
  if (handler != null)
  {
    handler.Invoke(this, new
    PropertyChangedEventArgs(propertyName));
  }
}
```

This whole block of code can be replaced with a single line if the null conditional operator is used:

```
public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(string propertyName)
{
  PropertyChanged?.Invoke(this, new
  PropertyChangedEventArgs(propertyName));
}
```

The call is still thread-safe, i.e. the generated code evaluates the variable only once. It then keeps the result in a temporary variable so that it cannot be changed afterwards.

Bitwise operators are specialized operators for bit manipulation of 32-bit (int and uint) and 64-bit (long and ulong) numeric values. Let's take advantage of binary literals and digit separators (both were added in C# 7.0) to see how they behave:

```
int a = 0b00000000_00000000_00000000_00001111;
int b = 0b00000000_00000000_00000000_01010101;
```

The result of the bitwise AND operator will include the bits which are set in both operands.

```
var and = a & b; // = 0b00000000_00000000_00000000_00000101
```

The result of the bitwise OR operator will include the bits which are set in at least one operand.

```
var or = a | b; // = 0b00000000_00000000_00000000_01011111
```

The result of the bitwise XOR (exclusive or) operator will include the bits which are set in exactly one operand.

```
var xor = a ^ b; // = 0b00000000_00000000_00000000_01011010
```

The result of the bitwise complement operator will include the bits which are not set in its (single) operand.

```
var complement = ~a; // = 0b11111111_11111111_11111111_11110000
```

The bitwise shift operators shift the bits in the binary representation to the left or to the right by the given number of places.

```
var shiftLeft = a << 1; // = 0b00000000_00000000_00000000_00011110
var shiftRight = a >> 1; // = 0b00000000_00000000_00000000_00000111
```

The bits which move past the end, do not wrap around to the other side. Hence, if we shifted any value far enough to the left or to the right, the result will be 0.

```
var multiShift = 0b00000000_00000000_00000000_00000001;
for (int i = 0; i < 32; i++)
{
  multiShift = multiShift << 1;
}
```

However, if we use the second operand to shift the bits by the same number of places in a single step, the result won't be the same.

```
var singleShift = 0b1 << 32; // = 0b1
```

To understand this behavior, we must look at how the operator is defined.

Before shifting the bits, the second operand will be normalized to the bit length of the first operand with the modulo operation, i.e. by calculating the remainder of dividing the second operand by the bit length of the first operand. In our example, the first operand is a 32-bit number, hence 32 % 32 = 0. The number will be shifted left by 0 places, not 32.

The bitwise AND, OR and XOR operators also serve as **logical operators** when applied to bool operands.

```
var a = true;
var b = false;
var and = a & b; // = false
var or = a | b; // = true
var xor = a | b; // = true
```

Although these AND and OR operators return the same result as the corresponding **conditional operators** (&& and ||), they don't behave identically. Operators & and | are eager, i.e. they will evaluate both operands even if the result can be determined after evaluating the first operand. Operators && and || are lazy, i.e. they won't evaluate the second operand when the result can be determined based on the value of the first operand. This can be an important difference when operands are method calls instead of simple variables:

```
var eager = true | IsOdd(1); // IsOdd will be invoked
var lazy = true || IsOdd(1); // IsOdd won't be invoked
```

All arithmetic and bitwise operators also have a corresponding shorthand assignment operator for the case when the calculated value is assigned to the first operand.

```
a = a + b; // basic syntax
a += b; // shorthand syntax
```

A special case of arithmetic operators is increment and decrement operators, which respectively increment or decrement the operand value by 1. They are available in two flavors: prefix and postfix. As the name implies, the former changes the operand before returning its value, while the latter first returns the operand value, and then changes it.

```
var n = 1;
var  prefixIncrement = ++n; // = 2, n = 2
var  prefixDecrement = --n; // = 1, n = 1
var postfixIncrement = n++; // = 1, n = 2
var postfixDecrement = n--; // = 2, n = 1
```

All four operators are most commonly used in `for` loops to change the value of the loop variable.

20

Q20. What does it mean that C# is a type safe language?

Type safety is a principle in programming languages which prevents accessing of variable values assigned inconsistently with their declared type. There are two kinds of type safety based on when the checks are performed:

- Static type safety is checked at compile time.
- Dynamic type safety is checked at runtime.

Without the checks, the values could be read from the memory as if they were of another type than they are, which would result in undefined behavior and data corruption.

C# implements measures for both kinds of type safety.

Static type safety prevents access to non-existent members of the declared type:

```
string text = "String value";
int textLength = text.Length;
```

```
int textMonth = text.Month; // won't compile

DateTime date = DateTime.Now;
int dateLength = date.Length; // won't compile
int dateMonth = date.Month;
```

The two lines marked with a comment won't compile, because the declared type of each variable doesn't have the member we are trying to access. This check can't be circumvented even if we try to cast the variable to the type which does have the requested member:

```
string text = "String value";
int textLength = text.Length;
int textMonth = ((DateTime)text).Month; // won't compile

DateTime date = DateTime.Now;
int dateLength = ((string)date).Length; // won't compile
int dateMonth = date.Month;
```

The two lines with comments still won't compile. Although the type we're trying to cast the value to has the member we're accessing, the compiler doesn't allow the cast because there is no cast operation defined for converting between the two types.

However, static type safety in C# is not 100% reliable. By using specific language constructs, the static type checks can still be circumvented, as demonstrated by the following code:

```
public interface IGeometricShape
{
  double Circumference { get; }
  double Area { get; }
}

public class Square : IGeometricShape
{
  public double Side { get; set; }
  public double Circumference => 4 * Side;
  public double Area => Side * Side;
}
```

```
public class Circle : IGeometricShape
{
  public double Radius { get; set; }
  public double Circumference => 2 * Math.PI * Radius;
  public double Area => Math.PI * Radius * Radius;
}

IGeometricShape circle = new Circle { Radius = 1 };
Square square = ((Square)circle); // no compiler error
var side = square.Side;
```

The line marked with the comment will compile without errors, although we can see from the code that the `circle` variable contains a value of type `Circle` which can't be cast to the `Square` type. The compiler does not perform the static analysis necessary to determine that the circle variable will always contain a value of type Circle. It only checks the type of the variable. Since the compiler allows downcasting from the base type (the IGeometricShape interface) to a derived type (the Square type) as it might be valid for certain values of the variable, our code will compile.

Despite that, no data corruption will happen because of this invalid cast. At runtime, the compiled code will not execute. It will throw an InvalidCastException when it detects that the value in the circle variable can't be cast to the Square type. This is an example of dynamic type safety in C#. To avoid the exception being thrown at runtime, we can include our own type checking code and handle different types in a different way:

```
if (shape is Square)
{
  Square square = ((Square)shape);
  var side = square.Side;
}

if (shape is Circle)
{
  Circle circle = ((Circle)shape);
  var radius = circle.Radius;
}
```

Since `object` is the base type for all reference types in .NET, static type checking can

almost always be circumvented by casting a variable to the object type first, and then to the target type. Using this trick, we can modify the code from our first example to make it compile:

```
string text = "String value";
int textLength = text.Length;
int textMonth = ((DateTime)(object)text).Month;

DateTime date = DateTime.Now;
int dateLength = ((string)(object)date).Length;
int dateMonth = date.Month;
```

Of course, thanks to dynamic type safety in C#, an InvalidCastException will still be thrown. .NET took advantage of this approach to implement common data structures before the introduction of generics while still preserving type safety, albeit only at runtime. These data structures are still available today, although their use is discouraged in favor of their generic alternatives.

```
var list = new ArrayList();
list.Add("String value");
int length = ((string)list[0]).Length;
int month = ((DateTime)list[0]).Month; // no compiler error
```

`ArrayList` is an example of a non-generic data structure in the .NET framework. Since it doesn't provide any type information about the values it contains, we must cast the values back to the correct type on our own before we can use them. If we cast the value to the wrong type as shown above in the line marked with a comment, the compiler can't detect that. The type checking will only happen at runtime.

By using generic data structures instead, the type information is preserved at compile time, allowing full static type checking.

```
var list = new List<string>();
list.Add("String value");
int length = list[0].Length;
int month = list[0].Month; // compiler error
```

There's no need to cast the values anymore, which prevents programmer mistakes and allows the compiler to detect errors.

Looking at the examples so far, C# could be considered a type safe language with full dynamic type safety. However, it contains a language feature which can be used to also circumvent the dynamic type checking. It's called unsafe code.

```
unsafe {
  long longValue = 42;
  long* longPointer = &longValue;
  double* doublePointer = (double*) longPointer;
  double doubleValue = *doublePointer;
}
```

In code blocks marked as unsafe, pointers can be used to gain unrestricted direct access to memory. In the above code sample, we have managed to read a value of type `long` as if it were of type `double`. The resulting value doesn't make any sense in most cases.

Still, both types (long and double) are 64 bits in size. Therefore, we only accessed memory which was previously allocated. Nothing would have prevented us from gaining access to unallocated memory if we haven't taken that precaution ourselves (e.g. if we tried to read a 32-bit int value as a 64-bit double value).

Of course, all of this makes direct memory manipulation and unsafe code in general potentially much more dangerous than regular managed code. That's also the reason why we need to enable it explicitly with a special compiler flag on the *Build* tab of the project *Properties* window.

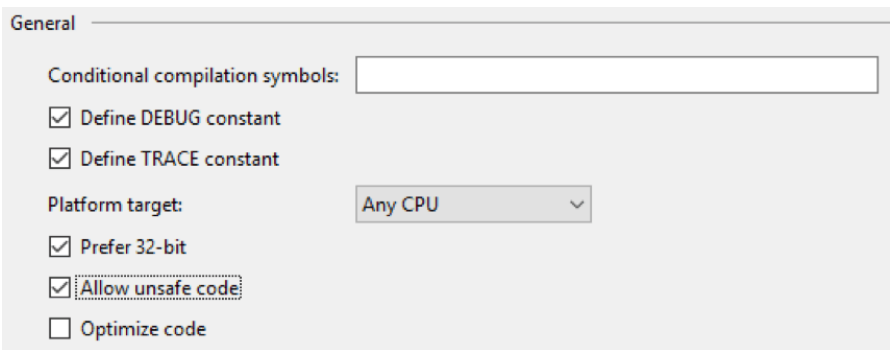


Figure 1: Allow unsafe code in project properties

Except in some rare cases when we need to interoperate with native code or have specific performance requirements, we should avoid using unsafe code.

SECTION III

CLASSES AND INHERITANCE

29

Q29. How do delegates and events differ?

Delegates are special types which describe a method signature, consisting of its parameters and the return type.

```
public delegate int Transform(int n);
```

When a variable of that delegate type is declared in code, any static or instance method matching its signature can be assigned to it.

```
int Increment(int n)
{
  return n + 1;
}

Transform incrementFunction = Increment;
var incremented = incrementFunction(1); // 2
```

In effect, a variable of a delegate type is a strongly-typed reference to a method which

can be invoked or passed around as a parameter to another method:

```
void TransformAll(List<int> list, Transform transform)
{
  for (var i = 0; i < list.Count; i++)
  {
    list[i] = transform(list[i]);
  }
}

var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, Increment); // list = { 2, 3, 4, 5, 6 }
```

The TransformAll method can be easily reused to apply different transformations to a list of numbers by passing it a different transform method:

```
int Decrement(int n)
{
  return n - 1;
}

var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, Decrement); // list = { 0, 1, 2, 3, 4 }
```

Creating a regular named method on a type for the sole reason to pass it as an argument to another method, or to assign to a variable without ever calling it otherwise, might seem unnecessary. That's why later versions of C# introduced an alternative syntax for writing code blocks which will only ever be assigned to a delegate type and never invoked directly.

In C# 2, *anonymous methods* were introduced to avoid the need for creating a named method as a type member. They allow a method to be directly defined inline, where it is assigned to a variable or passed as an argument:

```
Transform incrementFunction = delegate(int n)
{
  return n + 1;
};
```

```
var incremented = incrementFunction(1); // 2
var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, delegate(int n)
{
  return n - 1;
}); // list = { 0, 1, 2, 3, 4 }
```

In C# 3, lambda expressions were introduced as an even shorter alternative syntax for defining an inline block of code:

```
Transform incrementFunction = n =>
{
  return n + 1;
};
var incremented = incrementFunction(1); // 2
```

Using this syntax, the lambda expression could consist of multiple statements just like an anonymous or named method. Its only requirement is that it must return a value matching the return type declared by the delegate type.

When the inline code consists of only a single statement or an expression, an even shorter lambda expression syntax can be used:

```
Transform incrementFunction = n => n + 1;
var incremented = incrementFunction(1); // 2
```

To reduce the overhead of using delegates even further, a selection of generic delegate types was added to the base class library in .NET framework 3.5, mostly removing the need to create your own delegate types. Instead of the Transform delegate type, we can use Func<int, int>:

```
void ProcessAll(List<int> list, Func<int, int> processFn)
{
  for (var i = 0; i < list.Count; i++)
  {
    list[i] = processFn(list[i]);
  }
}
```

```
var list = new List<int> { 1, 2, 3, 4, 5 };
ProcessAll(list, x => x - 1); // list = { 0, 1, 2, 3, 4 }
```

Both anonymous methods and lambda expressions can be used to instantiate delegates. However, it is recommended to use lambda expressions instead of anonymous or named methods unless you are targeting an older version of C# and the .NET framework.

Anonymous methods and lambda expressions have another potential advantage over named methods. The code inside them has access to all variables outside it which are in scope i.e. where the anonymous method or lambda expression is declared:

```
var list = new List<int> { 1, 2, 3, 4, 5 };

var increment = 1;
ProcessAll(list, x => x + increment); // list = { 2, 3, 4, 5, 6 }

increment = 3;
ProcessAll(list, x => x + increment); // list = { 5, 6, 7, 8, 9 }
```

Delegates can be combined. Multiple methods can be invoked with a single invocation of delegate-typed variable. To add a method to a delegate variable or remove one from it, the + and - operators are used respectively:

```
Action firstDelegate = () =>
{
  Console.WriteLine("First delegate invoked.");
};
Action secondDelegate = () =>
{
  Console.WriteLine("Second delegate invoked.");
};

Action combinedDelegate = firstDelegate;
Console.WriteLine("First call.");
combinedDelegate();

combinedDelegate += secondDelegate;
Console.WriteLine("Second call.");
combinedDelegate();
```

```
combinedDelegate -= firstDelegate;
Console.WriteLine("Third call.");
combinedDelegate();
```

Between invocations of combinedDelegate in the above code, we are adding and removing lambdas. The resulting output will be as follows:

```
First call.
First delegate invoked.
Second call.
First delegate invoked.
Second delegate invoked.
Third call.
Second delegate invoked.
```

This capability of invoking multiple methods with a single delegate is used in events which provide an abstraction layer on top of delegates to give them publish and subscribe semantics. Events are exposed as public members of types. Other instances can subscribe to or unsubscribe from them.

To subscribe to an event, the same syntax is used as that for combining delegates. To unsubscribe from it, the syntax is the same as for removing a component delegate from a combined delegate:

```
var instance = new NotifyingClass();
EventHandler<MethodEventArgs> handler = (sender, eventArgs) =>
{
  Console.WriteLine($"Event handler called with {eventArgs.
Argument}.");
};
instance.MethodInvoked += handler; // subscribe to event
instance.MethodInvoked -= handler; // unsubscribe from event
```

Although not required, it is strongly recommended that all events use the `EventHandler` delegate type when no arguments are required, or its generic version `EventHandler<T>` when the delegate needs to be invoked with additional arguments. The generic type for the arguments should be derived from the EventArgs base class. Therefore, when implementing a custom event, a custom EventArgs-derived class will need to be implemented first:

```
public class MethodEventArgs : EventArgs
{
  public int Argument { get; private set; }
  public MethodEventArgs(int argument)
  {
    Argument = argument;
  }
}
```

The event can then be declared using the EventHandler<MethodEventArgs> delegate type:

```
public class NotifyingClass
{
  public event EventHandler<MethodEventArgs> MethodInvoked;
  public void Method(int argument)
  {
     MethodInvoked?.Invoke(this, new MethodEventArgs(argument));
  }
}
```

The `Method` method invokes the event handler using the null-conditional operator which was introduced in C# 6. If no event handler was subscribed to the event, the value of MethodInvoked would be null and invoking it without the null-conditional operator would therefore result in a NullReferenceException. The operator accesses the event only once, so that the value can't be changed from another thread between the null check and the invocation.

Apart from standardizing the publish and subscribe interaction, events also provide an abstraction over directly exposing a delegate. They make it possible to manually implement the logic for handling the addition and removal of event handlers:

```
public class InstrumentedNotifyingClass
{
  private EventHandler<MethodEventArgs> methodInvokedDelegate;
  public event EventHandler<MethodEventArgs> MethodInvoked
  {
    add
    {
```

```
      methodInvokedDelegate += value;
      Console.WriteLine("Added event handler.");
    }

    remove
    {
      methodInvokedDelegate -= value;
      Console.WriteLine("Removed event handler.");
    }
  }

  public void Method(int argument)
  {
    methodInvokedDelegate?.Invoke(this, new
    MethodEventArgs(argument));
  }
}
```

The code above adds logging to both operations. We can test it with the following code:

```
var instance = new InstrumentedNotifyingClass();
EventHandler<MethodEventArgs> handler = (sender, eventArgs) =>
{
  Console.WriteLine($"Event handler called with {eventArgs.
Argument}.");
};

instance.MethodInvoked += handler;
instance.Method(42);
instance.MethodInvoked -= handler;
```

The output to console will reflect the exact order of calls:

```
Added event handler.
Event handler called with 42.
Removed event handler.
```

Although the events can be customized in such a way, it is rarely done in production code and you should avoid it unless you have a good reason for it.

SECTION IV

STRING MANIPULATION

37

Q37. How do locale settings affect string manipulation?

Many string operations can behave differently based on the locale settings which are in use. In the .NET framework, all culture-based string operations use the current culture which is set for the current thread (as the CurrentCulture property of the Thread.CurrentThread static property) unless a different culture is specified for a specific call.

By default, the value of `CurrentCulture` property corresponds to the users' region and language settings in the operating system which is what the user typically expects. This makes it great in user-focused scenarios, i.e. when showing resulting strings to the user or parsing user input.

The fact that the behavior depends on the user settings makes these defaults unsuitable for internal string processing and programmatic communication between different components and systems.

In such scenarios, it is beneficial that the results are always and everywhere the same; deterministic and predictable. This can be achieved by changing the `CurrentCulture`

property on a thread if that thread never processes user-facing strings, or by specifying the desired culture for each operation.

The most obvious operations affected by culture settings, are formatting of dates and numeric values. When the `ToString` method is called on a value without a culture specified, the CurrentCulture is used (the samples in this chapter assume "en-US" locale settings are in use):

```
var date = new DateTime(2018, 9, 1);
var enDate = date.ToString("d"); // = "9/1/2018"
```

A different culture can be passed to the method to format the date differently:

```
var slCulture = CultureInfo.GetCultureInfo("sl-SI");
var slDate = date.ToString("d", slCulture); // = "1. 09. 2018"
```

The same approach can be used for numeric values. Although there is less variety in how the values are formatted, the differences are still important:

```
var pi = 3.14;
var enPi = pi.ToString(); // = "3.14"
var slPi = pi.ToString(slCulture); // = "3,14"
```

Of course, a fixed culture can also be specified when the `string.Format` method is used for composite formatting. The same culture will be used for formatting all the values injected into the format string:

```
var temperature = 21.5;
var timestamp = new DateTime(2018, 9, 1, 16, 15, 30);
var formatString = "Temperature: {0} °C at {1}";
var enFormatted = string.Format(formatString, temperature,
timestamp); // = "Temperature: 21.5 °C at 9/1/2018 4:15:30 PM"
var slFormatted = string.Format(slCulture, formatString,
temperature, timestamp); // = "Temperature: 21,5 °C at 1. 09. 2018
16:15:30"
```

Just like culture settings affect the formatting of dates and numeric values, they also affect the parsing of those values from a string. The same input string can be parsed very differently depending on what culture is in use:

```
var stringNumber = "3,14";
Double.TryParse(stringNumber, out var enNumber); // = 314
Double.TryParse(stringNumber, NumberStyles.Any, slCulture, out
var slNumber); // = 3.14
```

A less obvious operation affected by locale settings is string comparison:

```
var doubleS = "ss";
var eszett = "ß";
var equalOrdinal = string.Equals(doubleS, eszett); // = false
var equalCulture = string.Equals(doubleS, eszett, StringComparison.
CurrentCulture); // = true
```

The results of string comparison don't change only based on the culture. There are two additional important settings:

- Case sensitivity: are upper- and lower-case characters treated the same or not
- Ordinal or culture-based: ordinal comparison is based solely on the byte representation of the characters, while culture-based comparison adheres to linguistic rules.

Default string comparison in .NET is ordinal and case-sensitive. That's why the two different but equivalent ways to write the same character in German, are treated as different strings. When the strings are compared according to a culture, that linguistic rule is considered, and the strings are treated as equal even with non-German locale settings.

Ordinal case-insensitive string comparison is a special case which deserves additional explanation.

When strings are compared according to this rule, they are internally converted to upper case, using the invariant culture (a special culture based on the English language without any regional specifics). Then they are compared based on their byte representation, just like in the case of the default ordinal case-sensitive comparison.

String comparison settings are also important when sorting strings. Collections in the .NET framework base class library have an option to specify the sorting rules to use. Based on that setting, the strings will be sorted differently in the collection:

```
var czCulture = CultureInfo.GetCultureInfo("cs-CZ");
var words = new[] { "channel", "double" };
var ordinalSortedSet = new SortedSet<string>(words); // = {
"channel", "double" }
var cultureSortedSet = new SortedSet<string>(words, StringComparer.
Create(czCulture, ignoreCase: true)); // = { "double", "channel" }
```

As I used the Czech locale for the second collection, the order of the two strings was reversed in it. This happens because the Czech language treats the "ch" sequence as a different letter from C. It is positioned next to the letter H, i.e. several letters after the letter D, therefore it is considered "greater than" D.

There is another common string operation which can behave differently based on the culture in use: the change from lower- to upper-case letters or vice-versa:

```
var trCulture = CultureInfo.GetCultureInfo("tr-TR");
var lowerCaseI = "i";
var upperCaseIen = lowerCaseI.ToUpper(); // = "I"
var upperCaseItr = lowerCaseI.ToUpper(trCulture); // = "İ"
```

There aren't many languages where these rules are different, but that makes the bugs caused by them even more difficult to detect. Converting a string to upper or lower case is often used for case-insensitive string comparisons. If a fixed culture is not used for the operation, the results might be unexpected.

In the Turkish language, there are two different letters `I`: one with a dot and one without it. This property is preserved even when changing the case, i.e. the upper-case version of the "i" with a dot will still have a dot. That's different from the behavior in most other languages where the lower-case "i" has a dot and the upper-case one doesn't.

SECTION V

GENERICS AND COLLECTIONS

46

Q46. Why are there so many Collection classes and which one should I use?

The Base Class Library contains many different collection classes. Most of them are placed in the System.Collections and the System.Collections.Generic namespaces, but there are quite a few present in other namespaces too. This can make it difficult to decide when to use which collection.

In most cases, only collections in the System.Collections.Generic namespace should be considered.

The collections in System.Collections are all non-generic, originating from the time before Generics was added to C# with the release of the .NET framework 2.0 and C# 2.0. Except when maintaining legacy code, they should be avoided because they don't provide type safety, and when used with value types, display poor performance compared to their generic counterparts.

The collections in other namespaces are highly specialized for specific scenarios and are usually not applicable to code from other problem domains. E.g. the System.Dynamic.ExpandoObject class implements multiple collection interfaces but is primarily meant for implementing dynamic binding (you can read more about dynamic binding in Chapter 21 – "How can dynamic binding be implemented in C#?").

Even when only looking at generic collections in the System.Collections.Generic namespace, the choice can still seem overwhelming.

Fortunately, the collections can easily be grouped into a few categories based on the interfaces they implement. These determine which operations are supported by a collection and consequently in which scenarios they can be used.

The common interface for collections is the ICollection<T> interface. It inherits from the IEnumerable<T> interface which provides the means for iterating through a collection of items:

```
IEnumerable<int> enumerable = new List<int>() { 1, 2, 3, 4, 5 };

foreach(var item in enumerable)
{
    // process items
}
```

The ICollection<T> interface has the `Count` property and methods for modifying the collection:

```
ICollection<int> collection = new List<int>() { 1, 2, 3, 4, 5 };

var count = collection.Count;
collection.Add(6);
collection.Remove(2);
collection.Clear();
```

These members should suffice for implementing a simple collection. Three interfaces extend this base interface in different ways to provide additional functionalities.

The IList<T> interface describes collections with items which can be accessed by their index. For this purpose, it adds an indexer for getting and setting a value:

```
IList<int> list = new List<int> { 1, 2, 3, 4, 5 };

var item = list[2]; // item = 3
list[2] = 6; // list = { 1, 2, 6, 4, 5 }
```

It also includes methods for inserting and removing a value at a specific index:

```
list.Insert(2, 0); // list = { 1, 2, 0, 6, 4, 5 }
list.RemoveAt(2); // list = { 1, 2, 6, 4, 5 }
```

The most commonly used class implementing the IList<T> interface is List<T>. It will work great in most scenarios, but there are other implementations available for more specific requirements. For example, the SynchronizedCollection<T> class has a built-in synchronization object which can be used to make it thread-safe.

The ISet<T> interface describes a set, i.e. a collection of unique items that doesn't guarantee to preserve their order. It also provides an Add method for adding individual items to the collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

var added = set.Add(0);
```

However, its Add method will only add the item to the collection if it's not already present in it. The return value will indicate whether the item was added or not.

The rest of the methods in the interface implement different standard set operations from set theory. The following methods modify the collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

set.UnionWith(new[] { 5, 6 });              // set = { 1, 2, 3, 4,
5, 6 }
set.IntersectWith(new[] { 3, 4, 5, 6, 7 }); // set = { 3, 4, 5, 6 }
set.ExceptWith(new[] { 6, 7 });             // set = { 3, 4, 5 }
set.SymmetricExceptWith(new[] { 4, 5, 6 }); // set = { 3, 6 }
```

The others only perform tests on the collection and return a Boolean value:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

var isSubset = set.IsSubsetOf(new[] { 1, 2, 3, 4, 5 }); // = true
var isProperSubset = set.IsProperSubsetOf(new[] { 1, 2, 3, 4, 5 });
// = false
var isSuperset = set.IsSupersetOf(new[] { 1, 2, 3, 4, 5 }); // =
```

```
true
var isProperSuperset = set.IsProperSupersetOf(new[] { 1, 2, 3, 4, 5
}); // = false
var equals = set.SetEquals(new[] { 1, 2, 3, 4, 5 }); // = true
var overlaps = set.Overlaps(new[] { 5, 6 }); // = true
```

The most basic implementation of ISet<T> is the HashSet<T> class. If you want the items in the set to be sorted, you can use SortedSet<T> instead. Immutable versions of both are available as well.

IDictionary<TKey, TValue> is the third interface extending the ICollection<T> interface. In contrast to the previous two, this one is for storing key-value pairs instead of standalone values, i.e. it uses KeyValuePair<TKey, TValue> as the generic parameter in ICollection<T>. Its members are designed accordingly. The indexer allows getting and setting the values based on a key, instead of an index:

```
IDictionary<string, string> dictionary = new Dictionary<string,
string>
{
    ["one"] = "ena",
    ["two"] = "dva",
    ["three"] = "tri",
    ["four"] = "štiri",
    ["five"] = "pet"
};

var value = dictionary["two"];
dictionary["zero"] = "nič";
```

The `Add` method can be used instead of the indexer to add a pair to the collection. Unlike the indexer, it will throw an exception if the key is already in the collection.

```
dictionary.Add("zero", "nič");
```

Of course, there are also methods for checking if a key is in the collection and for removing an existing pair based on its key value:

```
var contains = dictionary.ContainsKey("two");
var removed = dictionary.Remove("two");
```

The latter will return a value indicating whether the key was removed or not. If not, it is because it wasn't in the collection in the first place.

There are also properties available for retrieving separate collections of keys and values:

```
ICollection<string> keys = dictionary.Keys;
ICollection<string> values = dictionary.Values;
```

For scenarios with no special requirements, the Dictionary<TKey, TValue> class is the go-to implementation of the IDictionary<TKey, TValue> interface. Two different implementations are available if keys need to be sorted (SortedDictionary<TKey, TValue> and SortedList<TKey, TValue>), each with its own advantages and disadvantages. Many more implementations of the interface are available in the Base Class Library.

There are two collection classes worth mentioning which unlike the ones we discussed so far don't implement ICollection<T> or any specialized interface derived from it.

The first one is the Queue<T> class which implements a FIFO (first in, first out) collection. Only a single item in it is directly accessible, i.e. the one that's in it for the longest time. The methods for adding and removing an item have standard names for such a data structure:

```
var queue = new Queue<int>(new[] { 1, 2, 3, 4, 5 });

queue.Enqueue(6);
var dequeuedItem = queue.Dequeue();
```

There's also the `Peek` method which returns the same item as the `Dequeue` method but leaves it in the collection:

```
var peekedItem = queue.Peek();
```

The second one is the Stack<T> class which is similar to Queue<T>, but it implements a LIFO (last in, first out) collection. The single item that's available in this collection is the one that was most recently added. The methods are named accordingly:

```
var stack = new Stack<int>(new[] { 1, 2, 3, 4, 5 });
```

```
stack.Push(6);
var poppedItem = stack.Pop();
var peekedItem = stack.Peek();
```

Although this overview doesn't cover all the available collection classes, it should serve as a very good guide for choosing the right collection class for the most common scenarios.

SECTION VI

LANGUAGE INTEGRATED QUERY (LINQ)

56

Q56. When and how is a LINQ query executed and how does this affect performance?

The key feature of LINQ is its universal querying API independent of the target data source. However, the way LINQ queries are executed depends on the data source being queried.

When querying local in-memory collections (commonly called **LINQ to Objects**), the LINQ extension methods for the IEnumerable<T> interface are used. The implementation of the extension methods somewhat depends on whether the method returns a value typed as IEnumerable<T>, or a scalar value.

The methods returning an IEnumerable<T> execute in a deferred way, i.e. at the point in code where the result is retrieved, not when the query is defined:

```
var list = new List<int> { 1, 2, 3, 4, 5 };
var query = list.Where(item => item < 3); // not executed yet
It's not too difficult to implement such an extension method
ourselves:
public static IEnumerable<T> Where<T>(this IEnumerable<T> list,
Func<T, bool> predicate)
{
  foreach (var item in list)
```

```
  {
    if (predicate(item))
    {
      yield return item;
    }
  }
}
```

For a `Where` method, we only need to iterate through all the items using a `foreach` loop and return those items which satisfy the supplied predicate.

To achieve deferred execution, the method is implemented as an iterator, i.e. the values are returned using the `yield` return statement (you can read more about that in Chapter 44 – "How to implement a method returning an IEnumerable?"). Whenever this statement is reached, the control of code execution is transferred back to the caller performing the iteration until the next item from the returned IEnumerable<T> is requested.

The methods returning a scalar value are in a way even simpler because they execute immediately:

```
var list = new List<int> { 1, 2, 3, 4, 5 };
var sum = list.Sum(); // executes immediately
Such extension methods don't even need to be iterators:
public static int Sum(this IEnumerable<int> list)
{
  var sum = 0;
  foreach (var item in list)
  {
    sum += item;
  }
  return sum;
}
```

For a `Sum` method, we need to iterate through the items with a foreach loop to calculate the total sum returned in the end.

The fact that LINQ to Objects extension methods accept IEnumerable<T> as their first argument makes them useful not only for querying collection classes, but in other scenarios as well. One such example is querying of XML documents, also known as

LINQ to XML.

XML documents must still be loaded in memory when they are queried, but they are not modelled with standard collection classes. Instead, a root `XDocument` class is used to represent an XML document. Usually the XML document will be read from a file or parsed from an XML string:

```
var xmlDoc = XDocument.Parse(xmlString);
```

The `Root` property will contain an instance of `XElement`, corresponding to the root element of the XML document. This class provides many methods for accessing the other elements and attributes in the document. The most commonly used among them are probably:

- Elements, returning the direct child elements as an instance of IEnumerable<XElement>
- Descendants, returning direct and indirect child elements as an instance of IEnumerable<XElement>
- Attributes, returning the attributes of the element as an instance of IEnumerable<XAttribute>

Since they all return IEnumerable<T>, they can easily be queried using LINQ. They don't provide strong type checking for element and argument names, though. Strings are used instead:

```
var elements = xmlDoc.Root.Elements()
  .Where(element => int.Parse(element.Attribute("age").Value) >
21);
```

For a certain structure of the XML, the above query would return all person elements with the age attribute above a specific value. Here's an example of such a document:

```
<persons>
  <person name="John" surname="Doe" age="33" />
  <person name="Jane" surname="Doe" age="42" />
  <person name="Johnny" surname="Doe" age="13" />
</persons>
```

To use LINQ to XML, we don't need to concern ourselves with the parsing of the XML documents. Since elements and attributes are exposed as instances of IEnumerable<T>, we can use the LINQ extension methods on them in combination with the properties available on the XElement and XAttribute classes.

The abstraction of underlying data source extends over to **LINQ to Entities** as well. This term represents querying of external databases using Entity Framework or Entity Framework Core. Although the data is not queried locally in this case, the querying experience is almost identical.

Sample code from here on will be based on Entity Framework Core. To use it with Microsoft SQL Server, the Microsoft.EntityFrameworkCore.SqlServer NuGet package must be installed in your project.

To specify the mapping between the code and the database tables, corresponding properties must be defined in a class deriving from the `DbContext` class. Entity Framework will take care of the connectivity to the remote database:

```
public class PersonContext : DbContext
{
  public PersonContext(DbContextOptions<PersonContext> options)
    : base(options)
  { }

  // maps to the Persons table in database
  public DbSet<Person> Persons { get; set; }
}
```

Rows in each database table are represented by a simple DTO (data transfer object) matching the table structure:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Surname { get; set; }
  public int Age { get; set; }
}
```

The connection string for the database can be specified in a DbContextOptions<T>

instance:

```
var optionsBuilder = new DbContextOptionsBuilder<PersonContext>();
optionsBuilder.UseSqlServer(connectionString);
var options = optionsBuilder.Options;
```

With this set up, a database table can be queried very similar to an XML document or a local collection:

```
using (var context = new PersonContext(options))
{
  var query = context.Persons.Where(person => person.Age > 21);
}
```

Because the DbSet<T> collections implement IQueryable<T>, not just IEnumerable<T>; the extension methods on the former interface are used instead. The IQueryable<T> interface makes it possible to implement a **LINQ query provider** which takes care of querying the data remotely. In our case, the provider is part of Entity Framework Core.

When writing queries, most of the times we aren't aware of the difference between the two interfaces because the same set of extension methods is available for both of them. They are just implemented differently.

For the above query, the LINQ query provider for SQL Server would send the following SQL query to the server:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].
[Surname]
FROM [Persons] AS[person]
WHERE [person].[Age] > 21
```

At least for simple cases when we are querying the data, it's not important what SQL query is generated and how. We can safely assume that the data will be retrieved correctly and in an efficient manner. We can even use certain common .NET methods in the query predicates:

```
var query = context.Persons.Where(person => Math.Abs(person.Age) >
21);
```

The LINQ provider will still generate a matching SQL query:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].
[Surname]
FROM [Persons] AS[person]
WHERE ABS([person].[Age]) > 21
```

However, this will only work for functions which have built-in equivalents in the target database and can therefore be implemented in such a way by the LINQ provider. If we start using our own custom methods, they won't be converted correctly any more:

```
private static int CustomFunction(int val) {
  return val + 1;
}

var query = context.Persons.Where(person => CustomFunction(person.
Age) > 21);
```

In this case, the following SQL query will be sent to the database:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].
[Surname]
FROM [Persons] AS[person]
```

Although all the rows from the `Persons` table will be retrieved, the LINQ query will still return the correct result. The filtering will be done locally *after* the data is retrieved. The results are still correct, but the performance will suffer when the number of rows in the table increases. Because of that, we should always make the necessary tests to discover such problems during development.

There are many more types of data sources which can be queried using LINQ. A large collection of third party LINQ providers are published on NuGet for querying different types of relational (NHibernate) and non-relational (MongoDB.Driver) databases, REST services (Linq2Rest), and more.

Depending on how they are implemented, there might be differences in how much of the query is executed remotely and which parts of it are processed locally after the data is received. Therefore it's important to always read the documentation and test the functionality and performance of the final code.

SECTION VII

PARALLEL AND ASYNCHRONOUS PROGRAMMING

67

Q67. What is the best way to start using Asynchronous methods in existing code?

Asynchronous methods can only be awaited using the `await` keyword in asynchronous methods with the `async` keyword in their signature:

```
public async Task AsyncMethod()
{
  var data = await GetDataAsync();
  // ...
}
```

In synchronous methods, without the `async` keyword in their signature, the use of the `await` keyword is not allowed and such code will not compile.

There are other ways to call asynchronous methods from synchronous ones, but they all have their own disadvantages:

- The call to the asynchronous method could be made synchronous by accessing the `Result` property or invoking the `Wait` method of its return value:

```
public void SyncMethodWithBlocking()
{
  var data = GetDataAsync().Result;
  // ...
}
```

However, this would block the invoking thread until the method returned. Preventing that is the main advantage of asynchronous methods.

- The asynchronous method could be invoked like a simple void-returning method without accessing the returned value in any way:

```
public void SyncMethodFireAndForget()
{
  GetDataAsync();
  // ...
}
```

This would be useless for methods returning a value (as in this case) because there would be no way to access that value. But even for methods not returning a value, this would mean that the execution of the calling method would continue immediately without waiting for the asynchronous method to return. Also, any exceptions thrown by the asynchronous method cannot be caught in the calling method.

Keeping this in mind, it only makes sense to call asynchronous methods from other asynchronous methods. This means that the requirement for methods to be asynchronous cascades up the call stack, all the way to an entry point method which is not explicitly called by another method in the application code. What exactly these entry point are depends on the type of the application.

In ASP.NET MVC based **web applications**, the entry points are action methods in controllers which handle incoming HTTP requests. By default, they are synchronous:

```
public void IActionResult Index()
{
  // ...
  return View();
}
```

In recent versions of ASP.NET MVC (as well as in ASP.NET Core), asynchronous action methods are fully supported. To make an action method asynchronous, only its signature needs to be changed:

```
public async Task<IActionResult> Index()
{
  var data = await GetDataAsync();
  // ...
  return View();
}
```

This is easy enough to do in any existing application. After changing the signature of any action method which needs to call an asynchronous method somewhere down its call stack, all the methods it calls until it reaches the one which has to call the asynchronous method, must also be made asynchronous.

Desktop applications (WPF and also Windows Forms) are all event driven. Their entry points are therefore event handlers. These are also synchronous by default:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  // ...
}
```

Because they don't return a value, they can't return a `Task` even when made asynchronous:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
  var data = await GetDataAsync();
  // ...
}
```

Although other asynchronous methods can be awaited inside it, such an event handler method when invoked, can't be awaited by the application framework. This means that any exceptions thrown by it can't be caught directly.

To prevent the application from crashing, they must either be handled with a try/catch block in each asynchronous event handler or by a global error handler as in the

following WPF example:

```
public App()
{
  DispatcherUnhandledException += App_DispatcherUnhandledException;
}

private void App_DispatcherUnhandledException(object sender,
DispatcherUnhandledExceptionEventArgs e)
{
  var exception = e.Exception; // get exception
  // ...                          show the error to the user
  e.Handled = true;            // prevent the application from
  crashing
}
```

So, when adding asynchronous calls to an existing application, in addition to making the event handler and all the intermediate methods in the call stack asynchronous, the error handling also needs to be revised and potentially enhanced.

In **console applications**, the only entry point is the static `Main` method:

```
static void Main(string[] args)
{
  // ...
}
```

Since C# 7.1, this method can simply be converted to an asynchronous one:

```
static async Task Main(string[] args)
{
  var data = await GetDataAsync();
  // ...
}
```

If you want the console application to exit with a return code, the asynchronous `Main` method can also return an integer value:

```
static async Task<int> Main(string[] args)
{
  var data = await GetDataAsync();
  // ...
  return result;
}
```

In older versions of C#, there's no support for the asynchronous `Main` method. You will need to write the necessary plumbing yourself:

```
static void Main(string[] args)
{
  MainAsync(args).GetAwaiter().GetResult();
}

static async Task MainAsync(string[] args)
{
  var data = await GetDataAsync();
  // ...
}
```

In this case, you'll use the new `MainAsync` method as the entry point for your code instead of the `Main` method.

Either way, after making the entry point asynchronous in an existing application, you can call asynchronous methods in it. In order to do that, all the intermediate methods must also be made asynchronous, i.e. all the methods between the new entry point method (Main or MainAsync) and the asynchronous method call you want to add.

No other changes are required to keep the application behavior unchanged.

SECTION VIII

SERIALIZATION & REFLECTION

75

Q75. What are the potential dangers of using Reflection?

In addition to being less performant than statically compiled code, reflection can also have a negative effect on the stability and security of the application. The exact impact depends on the scenario in which reflection is used.

A common use case for reflection is **application extensibility with dynamically loaded plugins**. In its simplest form, the application can specify a public interface that a class must implement to provide a specific functionality that needs to be extensible:

```
public interface IPluggable
{
  string GetString();
}
```

The application could have multiple built-in implementations for such an interface. But in addition to that, it could also include functionality to find implementations of the same interface in assemblies placed in a specific directory for plugins.

The assembly trying to extend this functionality could simply implement the interface in the same way as the application would:

```
namespace AddOns
{
  public class Pluggable : IPluggable
  {
    public string GetString()
    {
      return "Dynamically loaded";
    }
  }
}
```

All the additional work for dynamically loading these plugins would need to be done by the application. It would need to scan the plugins directory for assemblies, load them and search for suitable classes inside them:

```
var pluggables = new List<Type>();
var pluggableInterface = typeof(IPluggable);

var pluginsDir = new DirectoryInfo(pluginsPath);
var files = pluginsDir.GetFiles();

foreach (var file in files)
{
  try
  {
    var assembly = Assembly.LoadFile(file.FullName);
    var types = assembly.GetTypes();
    var pluggableTypes = types.Where(type => type.IsClass && !type.
    IsAbstract && pluggableInterface.IsAssignableFrom(type));
    pluggables.AddRange(pluggableTypes);
  }
  catch (Exception e)
  {
    // log errors
  }
}
```

The code you just saw does exactly that. It retrieves all the types declared in each assembly and checks whether each one of them can be assigned to the plugin interface (which effectively means that it implements that interface). It also makes sure that the type is a non-abstract class meaning that an instance of it can be created.

The application could then list all the plugins it found:

```
foreach (var pluggable in pluggables)
{
  Console.WriteLine(pluggable.FullName);
}
```

Once the user selected one of the implementations, the application could create an instance of it (using its parameterless constructor) and use it:

```
var instance = (IPluggable)Activator.
CreateInstance(selectedPluggable);
Console.WriteLine(instance.GetString());
```

The nice thing about this implementation is that the plugin class can be used in a statically bound manner from the point where it was instantiated and cast to the target interface.

Still, dynamically loading code from random assemblies can negatively affect reliability.

Of course, some of that can be avoided with a better implementation (e.g. I should have checked that there is a parameterless constructor available before using it to create the instance). But even then, the dynamically loaded code could have bugs causing the application to crash. Or it could be malicious and could try to gain access to users' files or try to damage them.

Potential risks should always be thoroughly analyzed before adding such functionality to an application.

Another possible use case for reflection is the ability to **extend functionality of a type or library by accessing and manipulating its internal members**. Of course, this only makes sense when someone else created the code and you can't simply implement the functionality in the library or make the required members public.

For example, in Entity Framework Core, there's no built-in method available for obtaining the SQL query which is going to be sent to the database server. Since the query is obviously generated by EF Core, it must contain the code that generates the query. It is just not accessible through the library's public interface.

With the project being open source, it's not too difficult to find the code responsible for query generation. With the use of reflection, this code could then be invoked even if it's not publicly exposed. Smit Patel, one of Microsoft's developers, even published such code in one issue on GitHub:

```
public static class IQueryableExtensions
{
  private static readonly TypeInfo QueryCompilerTypeInfo =
  typeof(QueryCompiler).GetTypeInfo();

  private static readonly FieldInfo QueryCompilerField =
  typeof(EntityQueryProvider).GetTypeInfo().DeclaredFields.First(x
  => x.Name == "_queryCompiler");

  private static readonly PropertyInfo NodeTypeProviderField =
  QueryCompilerTypeInfo.DeclaredProperties.Single(x => x.Name ==
  "NodeTypeProvider");

  private static readonly MethodInfo CreateQueryParserMethod
  = QueryCompilerTypeInfo.DeclaredMethods.First(x => x.Name ==
  "CreateQueryParser");

  private static readonly FieldInfo DataBaseField =
  QueryCompilerTypeInfo.DeclaredFields.Single(x => x.Name == "_
  database");

  private static readonly PropertyInfo DatabaseDependenciesField
  = typeof(Database).GetTypeInfo().DeclaredProperties.Single(x =>
  x.Name == "Dependencies");

  public static string ToSql<TEntity>(this IQueryable<TEntity>
 query) where TEntity : class
 {
   if (!(query is EntityQueryable<TEntity>) && !(query is
```

```
    InternalDbSet<TEntity>))
    {
      throw new ArgumentException("Invalid query");
    }

    var queryCompiler = (IQueryCompiler)QueryCompilerField.
    GetValue(query.Provider);
    var nodeTypeProvider = (INodeTypeProvider)NodeTypeProviderField.
    GetValue(queryCompiler);
    var parser = (IQueryParser)CreateQueryParserMethod.
    Invoke(queryCompiler, new object[] { nodeTypeProvider });
    var queryModel = parser.GetParsedQuery(query.Expression);
    var database = DataBaseField.GetValue(queryCompiler);
    var queryCompilationContextFactory = ((DatabaseDependencies)
    DatabaseDependenciesField.GetValue(database)).
    QueryCompilationContextFactory;
    var queryCompilationContext = queryCompilationContextFactory.
    Create(false);
    var modelVisitor = (RelationalQueryModelVisitor)
    queryCompilationContext.CreateQueryModelVisitor();
    modelVisitor.CreateQueryExecutor<TEntity>(queryModel);
    var sql = modelVisitor.Queries.First().ToString();

    return sql;
  }
}
```

The code above worked flawlessly with the version 2.0.0 of EF Core. However, it didn't work anymore with the next minor release, i.e. version 2.1.0.

That's the danger of using reflection in such a way.

Because the code is accessing internal members, there's no guarantee that these won't change in a future version. Private members are usually private for a reason. They give developers the freedom to refactor the code without having to keep them intact as they should with public members.

Because private members are accessed with the help of literal strings passed to the reflection APIs, the compiler also can't verify whether these members are still there. The code above will compile just fine even with the latest version of EF Core. It will

fail only at runtime when it won't find the missing private members which it wants to access.

With unit tests, the problem could at least be detected at build time when the tests would run and fail to access the requested members. But it would still require additional unplanned changes to the code in order to upgrade it to use the latest version of EF Core.

Unless such use of reflection achieves a critical functionality for an application, it should always be avoided as far as possible.

SECTION IX

C# 6 AND 7

78

Q78. Which C# 6 features should I really start using?

Changes in C# 6.0 make the language less verbose. Most of them affect declarations of class members and building of expressions.

But which ones have the most positive impact on code maintainability and readability?

Wherever applicable, you should start using the following new features in your code base immediately:

- the `nameof` operator,
- the null-conditional operator,
- and support for `await` in catch and finally blocks.

I would strongly suggest you apply them even to your existing code.

The Base Class Library sometimes expects you to put symbol names (such as property

and parameter names) as string literals in your code, e.g. when your class implements the `INotifyPropertyChanged` interface or when throwing an `ArgumentException`:

```
public int GetFibonacciNumberBefore(int n)
{
  if (n < 0)
  {
    throw new ArgumentException("Negative value not allowed", "n");
  }

  // TODO: calculate Fibonacci number
  return n;
}
```

When you rename the `n` parameter, you will need to remember to change the string literal as well – the compiler will not warn you about it. The **nameof** operator in C# 6.0 is a perfect solution to this problem: the `nameof(n)` call will still compile into a string literal but you don't need to use any string constants at the source code level:

```
public int GetFibonacciNumberAfter(int n)
{
  if (n < 0)
  {
    throw new ArgumentException("Negative value not allowed",
    nameof(n));
  }

  // TODO: calculate Fibonacci number
  return n;
}
```

Not only will the compiler now warn you if you forget to rename the parameter usage in the `nameof` operator, but also the rename refactoring in Visual Studio will automatically rename it for you.

Whenever you want to invoke a property or a method on a class, you must be sure that the instance is not null beforehand, otherwise the dreaded NullReferenceException will be thrown at run time. This can result in a lot of trivial code, just to check for values not being null:

```
public int GetStringLengthBefore(string arg)
{
  return arg != null ? arg.Length : 0;
}
```

The **Null-conditional operator** can replace all such null checking code:

```
public int GetStringLengthAfter(string arg)
{
  return arg?.Length ?? 0;
}
```

The `arg?.Length` expression in the above code will evaluate to `arg.Length` when `arg` is not null; otherwise, it will evaluate to null. You can even cascade multiple calls one after another:

```
public string FirstItemAsString<T>(List<T> list) where T: class
{
  return list?.FirstOrDefault()?.ToString();
}
```

The `list?.FirstOrDefault()?.ToString()` expression will evaluate to null if list or `list.FirstOrDefault()` are null and will never throw an exception.
The null-conditional operator is not limited to properties and methods. It also works with delegates, but instead of invoking the delegate directly, you will need to call its `Invoke` method:

```
public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChangedAfter(string propertyName)
{
  PropertyChanged?.Invoke(this, new
  PropertyChangedEventArgs(propertyName));
}
```

Implementing events like this also ensures that they are thread-safe, unlike their naïve implementation in previous versions of C#:

```
private void NotifyPropertyChangedBefore(string propertyName)
{
  if (PropertyChanged != null)
  {
    PropertyChanged(this, new
    PropertyChangedEventArgs(propertyName));
  }
}
```

In multithreaded scenarios, the above code could throw a NullReferenceException. This is in case if another thread removed the last remaining event handler after the first thread checked the event for being null, but before it called the delegate. To avoid this, `PropertyChanged` needs to be stored into a local variable before checking it for null. The null-conditional operator in C# 6.0 takes care of this automatically.

The `async` and `await` keywords introduced in C# 5.0 made asynchronous programming much easier. Instead of creating a callback for each asynchronous method call, all of the code can now be in a single `async` method, with the compiler creating the necessary plumbing instead of the developer.

Unfortunately, the `await` keyword was not supported in catch and finally blocks. This brought additional complexity when calling asynchronous methods from error handling blocks:

```
public async Task HandleAsyncBefore(AsyncResource resource, bool
throwException) {
  Exception exceptionToLog = null;
  try
  {
    await resource.OpenAsync(throwException);
  }
  catch (Exception exception)
  {
    exceptionToLog = exception;
  }
  if (exceptionToLog != null)
  {
    await resource.LogAsync(exceptionToLog);
  }
}
```

To await the asynchronous method, it had to be called outside the catch block. This was the only way to make sure it completed before executing any subsequent code (or exiting the calling method as in the example above).

Of course, if you wanted to rethrow the exception and keep the stack trace, it would get even more complicated.

With C# 6.0, none of this is required any more – async methods can simply be **awaited inside both catch and finally blocks**:

```
public async Task HandleAsync(AsyncResource resource, bool
throwException)
{
  try
  {
    await resource.OpenAsync(throwException);
  }
  catch (Exception exception)
  {
    await resource.LogAsync(exception);
  }
  finally
  {
    await resource.CloseAsync();
  }
}
```

The compiler will create all the necessary plumbing code itself, and you can be sure that it will work correctly.

SECTION X

A Look into the Future

86

Q86. What Functional programming features are supported in C#?

Functional programming is an alternative programming paradigm to the more popular and common object-oriented programming paradigm.

There are several key concepts that differentiate it from other programming paradigms. Let's start by providing definitions for the most common ones, so that we will recognize them when we see them applied throughout this chapter.

Basic building blocks of functional programs are **pure functions**. A pure function is defined by the two properties:

- Its result depends solely on the arguments passed to it. No internal or external state affects it.
- It doesn't cause any side effects. The number of times it is called will not change the program behavior.

Because of these properties, a function call can be safely replaced with its result. For example, to improve performance, we can cache the results of a computationally

intensive function for each combination of its arguments (a technique known as **memoization**).

Pure functions lend themselves well to function composition. This is a process of combining two or more functions into a new function, which returns the same result as if all its composing functions were called in sequence. If `ComposedFn` is a function composition of `Fn1` and `Fn2`, then the following assertion will always pass:

```
Assert.That(ComposedFn(x), Is.EqualTo(Fn2(Fn1(x))));
```

Composition is an important part of making functions reusable.

Having functions as arguments to other functions can further increase their reusability. Such **higher-order functions** can act as generic helpers. For example, they can apply another function passed multiple times as an argument, e.g. on all items of an array:

```
var anyMinors = Array.Exists(persons, IsMinor);
```

In the above code, `IsMinor` is a function defined elsewhere. For this to work, support for **first-class functions** is required, i.e. functions must be treated as first-class language constructs just like value literals are.

Data is always represented with **immutable objects**, i.e. objects that cannot change their state after they have been initially created. Whenever a value changes, a new object must be created instead of modifying the existing one. Because all objects are guaranteed to not change, they are inherently thread-safe, i.e. they can be safely used in multithreaded programs with no threat of race conditions.

As a direct consequence of functions being pure and objects being immutable, there is **no shared state** in functional programs. Functions can act only based on their arguments, which they cannot change and with that, affect other functions receiving the same arguments. The only way they can affect the rest of the program is through the result they return, which will be passed on as arguments to other functions. This prevents any kind of hidden cross-interaction between the functions.

Unless one function directly depends on the result of the other, they can be safely run in any order or even in parallel.

With the above basic building blocks, functional programs end up being more declarative than imperative, i.e. instead of describing how to calculate the result, the

programmer rather describes what to calculate.

The following two functions for making an array of strings lower case clearly demonstrate the difference between the two approaches:

```
string[] Imperative(string[] words)
{
  var lowerCaseWords = new string[words.Length];
  for (int i = 0; i < words.Length; i++)
  {
    lowerCaseWords[i] = words[i].ToLower();
  }
  return lowerCaseWords;
}

string[] Declarative(string[] words)
{
  return words.Select(word => word.ToLower()).ToArray();
}
```

Although you will hear about many other functional concepts, such as monads, functors, currying, referential transparency and others, these should suffice to give you the basic idea of what functional programming is and how it differs from object-oriented programming.

You can implement many of the functional programming concepts in C#.

Since the language is primarily object-oriented, the defaults don't always guide you towards writing functional code. But with intent and enough self-discipline, your code can become much more functional.

You are very likely used to writing mutable types in C#, but with very little effort, they can be made **immutable**:

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

```
  public Person(string firstName, string lastName, int age)
  {
    FirstName = firstName;
    LastName = lastName;
    Age = age;
  }
}
```

Properties without setters can't be assigned a different value after the object has been initially created. For the object to be truly immutable, all of the properties must also be of immutable types. Otherwise their values can be changed by mutating the properties instead of assigning a new value to them.

The `Person` type above is immutable because `string` is also an immutable type, i.e. its value cannot be changed as all its instance methods return a new string instance. However, this is an exception to the rule and most .NET framework classes are mutable. If you want your type to be immutable, you should not use any built-in types other than primitive types and strings as public properties.

To change a property of the object, e.g. to change the person's first name, a new object needs to be created:

```
public static Person Rename(Person person, string firstName)
{
  return new Person(firstName, person.LastName, person.Age);
}
```

When a type has many properties, writing such functions can become quite tedious. Therefore, it is a good practice for immutable types to implement a `With` helper method for such scenarios:

```
public Person With(string firstName = null, string lastName = null,
int? age = null)
{
  return new Person(firstName ?? this.FirstName, lastName ?? this.
LastName, age ?? this.Age);
}
```

This method creates a copy of the object with any number of properties modified.

Our `Rename` function can now simply call this helper method to create the modified person:

```
public static Person Rename(Person person, string firstName)
{
  return person.With(firstName: firstName);
}
```

The advantages might not be obvious with only two properties, but no matter how many properties the type consists of, this syntax allows us to only list the properties we want to modify as named arguments.

Making functions **pure** requires even more discipline than making objects immutable. There are no language features available to help the programmer ensure that a particular function is pure. It is your own responsibility to not use any kind of internal or external state, to not cause side effects and to not call any other functions that are not pure.

Of course, there is also nothing stopping you from only using the function arguments and calling other pure functions, thus making the function pure. The `Rename` function above is an example of a pure function: it does not call any non-pure functions or use any data other than the arguments passed to it.

Multiple functions can be **composed** into one by defining a new function, which calls all the composed functions in its body (let us ignore the fact that there is no need to ever call Rename multiple times in a row):

```
public static Person MultiRename(Person person)
{
  return Rename(Rename(person, "Jane"), "Jack");
}
```

The signature of the `Rename` function forces us to nest the calls, which can become difficult to read and comprehend as the number of function calls increases. If we use the With method instead, our intent becomes clearer:

```
public static Person MultiRename(Person person)
{
  return person.With(firstName: "Jane").With(firstName: "Jack");
}
```

To make the code even more readable, we can break the chain of calls into multiple lines, keeping it manageable, no matter how many method calls we compose into one function:

```
public static Person MultiRename(Person person)
{
  return person
    .With(firstName: "Jane")
    .With(firstName: "Jack");
}
```

When calls are nested, as in the case of calling the `Rename` function inside the `MultiRename` function, there is no good way to split the lines.

Of course, the `With` method allows the chaining syntax due to the fact that it is an instance method. However, in functional programming, functions should be declared separately from the data they act upon, like the `Rename` function is. While functional languages have a pipeline operator (|> in F#) to allow chaining of such functions, we can take advantage of extension methods in C# instead:

```
public static class PersonExtensions
{
  public static Person Rename(this Person person, string firstName)
  {
    return person.With(firstName: firstName);
  }
}
```

This allows us to chain non-instance method calls the same way as we can chain instance method calls:

```
public static Person MultiRename(Person person)
{
  return person.Rename("Jane").Rename("Jack");
}
```

To get a taste of functional programming in C#, you don't need to write all the objects and functions yourself. There are some readily available functional APIs in the .NET framework for you to utilize.

We have already mentioned string and primitive types as immutable types in the .NET framework. However, there is also a selection of **immutable collection types** available.

Technically, they are not really a part of the .NET framework, since they are distributed out-of-band as a stand-alone NuGet package (System.Collections.Immutable). On the other hand, they are an integral part of .NET Core, the new open-source cross-platform .NET runtime. You can read more about them in Chapter 48 – "How are immutable collections different from other collections?".

A much better-known functional API in the .NET framework is **LINQ** (the entire section VI of the book is dedicated to it). Although it is not commonly advertised as being functional, it manifests many previously introduced functional properties.

If we take a closer look at LINQ extension methods, it quickly becomes obvious that almost all of them are declarative in nature: they allow us to specify what we want to achieve, not how:

```
var result = persons
  .Where(p => p.FirstName == "John")
  .Select(p => p.LastName)
  .OrderBy(s => s.ToLower())
  .ToList();
```

The above query returns an ordered list of last names of people named John. Instead of providing a detailed sequence of operations to perform, we only described the desired result. The available extension methods are also easy to compose using the chaining syntax.

Although LINQ functions are not necessarily acting on immutable types, they are still pure functions, unless made impure by passing mutating functions as arguments. They are implemented to act on collections of type `IEnumerable` which is a read-only interface. They don't modify the items in the collection. Their result only depends on the input arguments and they don't create any global side effects, as long as the functions passed as arguments are also pure. In the example above, neither the persons collection, nor any of the items in it will be modified.

Many LINQ functions are higher-order functions: they accept other functions as arguments. In the sample code above, lambda expressions are passed in as function arguments, but they could easily be defined elsewhere and passed in instead of created inline:

```
public bool FirstNameIsJohn(Person p)
{
  return p.FirstName == "John";
}

public string PersonLastName(Person p)
{
  return p.LastName;
}

public string StringToLower(string s)
{
  return s.ToLower();
}

var result = persons
  .Where(FirstNameIsJohn)
  .Select(PersonLastName)
  .OrderBy(StringToLower)
  .ToList();
```

When function arguments are as simple as in our case, the code will usually be easier to comprehend with inline lambda expressions instead of separate functions. However, as the implemented logic becomes more complex and reusable, having them defined as standalone functions starts to make more sense.

Thank you for reading these sample chapters.
We hope you found it useful!

Click here to Purchase this EBook

What will you get?

500 Pages of .NET and C# Goodness.

PDF, ePUB (iPad), Mobi (Kindle) versions

Source Code for the Chapters

Join our .NET communities at Twitter and Facebook.