High Performance Enterprise Apps using C# 10 and .NET 6

Hands-on Production-ready Clean Code, Pattern Matching, Benchmarking, Responsive UI and Performance Tuning Tools

OCKERT J. DU PREEZ

bpb

High Performance Enterprise Apps using C# 10 and .NET 6

Hands-on Production-ready Clean Code, Pattern Matching, Benchmarking, Responsive UI and Performance Tuning Tools

OCKERT J. DU PREEZ

bpb

High Performance Enterprise Apps Using C# 10 and .NET 6

Hands-on Production-ready Clean Code, Pattern Matching, Benchmarking, Responsive UI and Performance Tuning Tools

---

Ockert J. du Preez

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: August 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55510-181

www.bpbonline.com

Dedicated to

My beloved Family:

Elmarie, Michaela, Winton, *and* ***Björn***

About the Author

Ockert J. du Preez has been in the coding industry for 20+ years and has written hundreds of developer articles over the years detailing his programming quests and adventures. These can be found on CodeGuru, Developer.com and Database Journal.

He knows a broad spectrum of development languages including: C++, C#, VB.NET, JavaScript and HTML.

He has written the following books:

- Visual Studio 2019 In-Depth (BPB Publications)
- JavaScript for Gurus (BPB Publications)
- Building Cross-Platform Modern Apps using Visual Studio Code (VS Code)

He was the Technical Editor for the following books

- Professional C++, 5th Edition by Marc Gregoire
- C++ Software Interoperability for Windows Programmers by Adam Gladstone

He was a Microsoft Most Valuable Professional for .NET (2008–2017).

About the Reviewer

"**Richard Newcombe** has 25 years of experience in Big Data analysis and a strong background in Visual studio application development, as well as other packages. He has also studied in numerous fields. Software development has remained his strongest achievement. With a large collection of historical and current system and software books, reviewing is just the next step."

Acknowledgement

I want to thank my family for all their support, their belief in me and their support.

My gratitude also goes to the team at BPB Publications for their patience and support in writing this book.

Preface

As a developer you will encounter bottlenecks or many other performance related problems in your applications. This book aims to help you write preventative code and make use of the proper profiling technologies to ensure this doesn't happen.

This book takes a practical approach in a sense to help guide the develop better performing applications.

This book is divided into **12 chapters**. The details are listed below.

Chapter 1 introduces the reader to the history of .NET. It covers how .NET came into existence and how far it has come.

Chapter 2 explains the origins of C# and differentiates the various versions of C#

Chapter 3 delves into the features of C# 9 such as Records, new InterOp features, and Init-only setters.

Chapter 4 explains improvements in the C# language such as pattern matching and source generators.

Chapter 5 addresses the need for high performance code. It covers what application performance is all about, questions why performance tuning should be done and explains the levels of optimization

Chapter 6 explains why it is important to use the correct data types for the correct purpose

Chapter 7 goes into more details on pattern matching and covers Type patterns, conjunctive “AND” patterns, disjunctive “OR” patterns and relational patterns.

Chapter 8 explains why developers should use the correct collection for the correct purpose. It covers boxing cases, thread safety, and proper collection guidelines.

Chapter 9 deals about identifying performance problems with profilers, diagnosers and benchmarking

Chapter 10 works practically with some benchmarking tools.

Chapter 11 details the Memory Cache, how it works and why it is important

Chapter 12 details the Large Object Heap and how it can affect application performance.

Appendix A lists more resources to benchmarking and performance tools.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/h2i7yhh

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/High-Performance-Enterprise-Apps-Using-C--10-and-.NET-6**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Section - I: An Introduction to Code Performance and C#

1. The Need for High-Performance Code

- Introduction
- Structure
- Objective
- Application performance
  - *Application performance monitoring*
  - *Disadvantages of performance tuning*
  - *Levels of optimization*
- Conclusion
- Points to remember
- Questions
  - *Answers*
- Key terms
- References

2. Overview of C#

- Introduction
- Structure
- Objective
- The origins of C#
  - *Variables and constants*
  - *Data types*
  - *Creating/declaring variables*
  - *Naming conventions*
- Arrays and collections
  - *Arrays*
  - *Collections*
  - *Queues*
  - *Stack*
  - *Hashtable*

Enums
Selection statements
If
Switch
Iteration statements
for loop
foreach loop
do loop
while
Conclusion
Points to remember
Questions
Answers
Key terms
References

3. New and Improved Features in C#
Introduction
Structure
Objective
Records
Init only setters
Top-level statements
New Interop features
Native sized integers
Function pointers
Omitting the localsinit flag
Fit and finish features
Pattern matching enhancements
Null checks
Type tests
Specific values
C# pattern matching enhancements
Parenthesized patterns
Conjunctive and patterns
Disjunctive or patterns
Negated not patterns

Relational patterns
Conclusion
Points to remember
Questions
Answers
Key terms
References

4. Using Data Types
Introduction
Structure
Objective
Data types defined
Value types
Reference types
Object type example
String type example
Choosing the right data type for the right purpose
What is wrong here?
Conversion
Conclusion
Points to remember
Questions
Answers
Key terms
References

5. Enhancing Pattern Matching
Introduction
Structure
Objective
Parenthesized patterns
Conjunctive and patterns
Disjunctive or patterns
Negated not patterns
Relational patterns
Practical

Conclusion
Points to remember
Questions
Answers
Key terms
References

6. Using Collections Properly
Introduction
Structure
Objective
Collection types
ArrayList
BitArray
CaseInsensitiveComparer and CaseInsensitiveHashCodeProvider
Queue
SortedList
Stack
When to choose which Collection type
Boxing issues
Boxing
Unboxing
Thread safety in Collections
BlockingCollection
ConcurrentBag
Guidelines
Conclusion
Points to remember
Questions
Answers
Key terms
References

7. Identifying Performance Problems
Introduction
Structure
Objective

Using a profiler
Hardware interrupts
Code instrumentation
Debugging
Tracing
Writing to performance counters and event logs
Instruction set simulation
Operating system hooks
Rootkits
Keylogging
Keystroke logging
Keylogger tools
Using diagnostic tools
Analyze CPU usage
Analyze memory usage
Analyze asynchronous code
Conclusion
Points to remember
Questions
Answers
Key terms
References

8. Benchmarking Code with BenchmarkDotNet
Introduction
Structure
Objective
Benchmarking introduction
Benefits of benchmark testing
Phases of benchmark testing
Planning phase
Analysis phase
Integration phase
Action phase
Creating a benchmark test plan
Components of benchmark testing
Benchmarking architecture components

Successful benchmark testing
Benchmarking code with BenchmarkDotNet
BenchmarkDotNet explained
Benefits of using BenchmarkDotNet
Benchmark Config
Using BenchmarkDotNet
Conclusion
Points to remember
Questions
Answers
Key terms
References

9. Dealing with the Memory Cache
Introduction
Structure
Objective
Caching
Importance of caching
Benefits of caching
Disadvantages of caching
Caching mistakes
Cache replacement policies
In-memory cache
Persistent in-process cache
Distributed cache
When to consider implementing a distributed cache
Conclusion
Points to remember
Questions
Answers
Key terms
References

10. Working with the Large Object Heap
Introduction
Structure

Objective
Memory management in .NET
Garbage collection generations
Generation 0/Gen 0
Generation 1/Gen 1
Generation 2/Gen 2
Large object heap
The system is low on memory
The garbage collector's collect method is called
Allocation exceeds the Gen 0 or large object threshold
Common problems with the large object heap
Conclusion
Points to remember
Questions
Answers
Key terms
References

11. Creating a Responsive UI
Introduction
Structure
Objective
Creating a user interface
UX versus UI
Common UI design mistakes
Colors
Color symbolism
Color blindness
Conclusion
Points to remember
Questions
Answers
Key terms
References

12. Overcoming InterOp Challenges
Introduction

Structure
Objective
Defining Native InterOp
Platform Invoke (PInvoke)
C++ InterOp (It Just Works [IJW])
Native sized integers
Function pointers
Fixed-size buffers
Static delegates
Omitting the localsinit flag
Conclusion
Points to remember
Questions
Answers
Key terms
References

Appendix 'A'
Introduction
Structure
Objective
Reference list

Index

Section - I

An Introduction to Code Performance and C#

CHAPTER 1

The Need for High-Performance Code

Introduction

Any application can have issues, and some may have more than others. When mentioning application issues, developers and users tend to think of errors only. Although errors play a huge role in application performance, they are not all that influence application performance. It all starts with planning. When developing a large enterprise application, a lot of thought needs to go into the performance. When applications are sluggish and have timeout errors, for example, and these errors continue to happen, there is a problem that could have been prevented. In this chapter, we will learn all the performance metrics that need to be checked and planned for while developing an application.

Structure

Topics to be covered in this chapter are as follows:

- Application performance
- Disadvantages of performance tuning
- Levels of optimization

Objective

By the end of this chapter, the reader will know what application performance is and the different metrics to be tested for. Then we will have a look at the disadvantages of performance tuning. Finally, they will know the different levels of optimization, which explains where the optimization should start and where it should end.

Application performance

Ever had a program become unresponsive? Yes, most of us had to deal with a program that was not performing as it should. It takes a long time to do a simple procedure. It takes longer on a computation. There are times when the application performed fine on a day or so before, but all of a sudden, it is not performing well on another day.

In order to find these and more performance issues, first, we will have a quick look at application performance monitoring in general.

Application performance monitoring

Application performance monitoring or application performance management ensures that the application processes and performs in an expected manner and scope. Application performance monitoring continuously identifies, measures, and evaluates the performance of an application and then provides a way to isolate and fix any abnormalities or shortcomings.

These are measured by performance metrics. Performance metrics are defined as figures and data that represent an organization's actions, abilities, and overall quality. Some of the most important metrics to check for are as follows. Let us go through each of them quickly:

- **CPU usage**: This metric includes CPU usage, memory demands, and disk read and write speeds. If the CPU is under pressure, that is, many applications are open, and the memory is low, it can affect disk read and disk write speeds. A small CPU or not enough memory can influence applications to behave badly and lock or hang with a big possibility of losing data.
- **Number of instances**: It is quite easy to fall into the trap of re-opening an application multiple times. With a small program such as Notepad, for instance, this is not much of a problem, but with a bigger application such as Adobe Photoshop or CorelDraw, having multiple instances can cause applications to cause havoc. By testing this metric, it is easy to ascertain how many instances of an application are open and how they affect the memory and CPU processing.
- **Request rates**: The request rates metric determines how much traffic an application receives. There are periods during an application's running time when the request rates are higher because of more users

using the application during peak hours of work. Other times the application's request rates are low due to the fact of not being used by many concurrent users.

- **Application availability or uptime**: It is always a good idea to test if the application is currently available or not. If the application is not running, then there must be something wrong: a server could have crashed, or it could even be a network-related issue.
- **Garbage collection**: Although garbage collection has a plethora of advantages for overall application performance, it can also go a bit overboard with cleaning up resources; thus, creating potential deadlocks and performance issues. Not only should garbage collection be used sparingly, meaning when it is necessary, but it should also be monitored by a performance monitor.
- **Apdex scores**: **Apdex** (**Application Performance Index**) is an open standard that can report, rate, or benchmark application response times. It reports on user experiences into a zero-to-one score.
- **Throughput**: Throughput is the rate at which something is processed or the rate of successful message delivery over a communications channel. If the throughput is slow, there may be a problem with the network or resources allocated to the application.
- **Queue time**: Queue time can be equated the following way: The number of transactions in an application equals the throughput multiplied by the average response time in an application. Refer to the References section at the end of this chapter regarding Little's Law that helps equate queue time.
- **95th percentile response time**: Percentile response times are very important metrics to follow when looking at application response times. In general, percentile response times show the response time that the majority of users will encounter, excluding the major outliers. Average, min, and max are useful but do not provide a clear picture. Average response time is very misleading since this will show response times that only half of the users will experience. Min and Max are useful; however, they can include one-off response times.
- **Error rates**: Monitoring error rates is a crucial performance metric. Errors make or break any application's success. This is where proper

error handling and proper real-world testing need to happen. Unfortunately, there may be a few errors that may still creep in; for this reason, monitoring error rates is critical. Some error rates that can be tested for include the number of unhandled and logged errors from an application, the number of all exceptions that have been thrown, the number of Web requests that ended in an error, and hidden or swallowed application exceptions. For more information on swallowed exceptions, refer to the References section at the end of this chapter.

- **Memory**: Most programming languages allocate memory differently; understanding how and when the developer's chosen language both allocates and cleans up memory is critical to scaling.

Disadvantages of performance tuning

It may seem strange to see the heading of this section: *Disadvantages of performance tuning*. Obviously, performance tuning is a great and necessary skill to have; what could the cons or disadvantages even be?

The following are a few disadvantages:

- **Compulsive tuning disorder**: This happens when the developer or team constantly tweaks the application to perform better and better, even when it is already optimal. It is quite easy to fall into this trap, especially for inexperienced developers.
- **Time and money**: There is an old adage that says: time is money. Performance tuning takes time to do it properly. Setting up test cases and even possibly learning the performance tuning tools takes time and effort.
- **Too little too late**: This is quite an apt description of what developers tend to do as an afterthought. There is also an adage that says Too little too late. It helps nothing when the developers leave performance tuning until it is too late. The whole process of performance testing becomes rushed and is not done and tested properly, which actually defeats the whole purpose of performance tuning.
- **Knowledge**: There may be times when inexperienced developers need to do performance tuning. This comes back to an earlier point (time and money). When a developer does not know which tools to use and

how to use the provided tools, this can cause a huge delay in the release of the product. Again, time is money.

- **Over-optimization**: Over-optimization happens when the performance of applications is tuned too much, causing the application to still not behave correctly, if not worse. Developers tend to over-optimize code because it is either a new term that they have learned, or they are too scared the application will perform badly.

Levels of optimization

Optimization usually occurs at a number of levels. The higher the level, the greater the impact. Higher levels are harder to change near the completion of a project because it requires vast amounts of changes or even rewrites. This is why optimization should be done from a higher level to a lower level. Consideration should be given to the efficiency throughout a project. Longer-running projects usually contain cycles of optimization, where improving one area shows limitations on another.

We can break down the levels of optimization as follows:

- Design level
- Algorithms and data structures
- Source code level
- Build level
- Compile level
- Assembly level
- Run time
- Platform dependent and independent optimizations

Let us go into more detail on each:

Level	Description
Design level	This is the highest level. Here, the design should be optimized to make the best use of the resources (images, sounds, and so on), goals, any constraints, and the expected load. How a design is put together affects the project's performance greatly.
Algorithms and data structures	After the design level, the algorithms and data structures used in the project may affect efficiency more than any other aspect of the

	program because these are more difficult to change later on.
Source code level	The source code level depends on the source language, the target machine language, and the compiler. Because of this, it is a bit difficult to predict, and the language changes continuously.
Build level	This level is nested between the source code level and the compile level. Because of the build level, directives and build flags could be used to tune performance in the source code and compiler, respectively.
Compile level	At Compile level, it is best to make use of an optimizing compiler to ensure that the executable program is optimized properly for the target system.
Assembly level	Assembly level is the lowest level of optimization. Most code written these days is aimed at as many machines as possible because writing low-level assembly code is not a really common practice due to the time and costs involved. Assembly written code aids in understanding the platform it is written on perfectly as it is specifically tuned for a particular processor and machine instructions. Instead of making use of Assembly code, developers make use of disassemblers to analyze compiler output.
Run time	With the use of Just-in-time compilers, compilers produce customized machine code based on run-time data at the cost of compilation overhead.
Platform dependent and independent optimizations	Writing different versions of the same code for different processors might be needed.

Table 1.1: *Levels of optimization*

Conclusion

In this chapter, we learned about application performance in general. We learned why it is important to understand how performance metrics work and where the potential pitfalls are when checking performance issues. Finally, we had a look at the various levels of optimization, and we saw that the design level is the most critical. In the upcoming chapter, we will brush up on our C# skills.

Points to remember

- Application performance management ensures that an application processes and performs in an expected manner and scope.

- Compulsive tuning disorder happens when the developer or team constantly tweaks the application to perform better and better, even when it is already optimal.
- Performance metrics are defined as figures and data that represent an organization's actions, abilities, and overall quality.

Questions

1. Name three performance metrics to monitor.
2. Name the levels of optimization.

Answers

1. **Any of the following:** CPU usage, number of instances, request rates, application availability, garbage collection, Apdex scores, throughput, queue time, 95th percentile response time, error rates, and memory.
2. Design, algorithms, source code, build, compile, assembly, and run time.

Key terms

- Application performance metric
- Apdex scores
- CPU usage
- Request rates
- Queue time
- 95th percentile response time
- Cycles of optimization
- Levels of optimization

References

- **Little's Law:** https://toggl.com/track/littles-law/

- **Swallowed (Hidden) Exceptions: https://stackify.com/hidden-exceptions-prefix/**
- **Visual Studio In-Depth: https://www.amazon.com/Visual-Studio-2019-Depth-applications/product-reviews/9389328322/ref=cm_cr_dp_d_show_all_btm?ie=UTF8&reviewerType=all_reviews**

CHAPTER 2

Overview of C#

Introduction

The C# language has been around for a very long time. It has kept on evolving over the years. Now, with the advent of newer technologies such as IoT, the language had to adapt and grow to compensate for newer platforms.

But, in order to adapt to the C# language, we have to ensure that first, we know the basics of the language, and then, second, we see what all the new features can do. This chapter explores the C# language as well as looks at its new features.

Structure

The following topics are to be covered:

- The origins of C#
- Variables and constants
- Data types
- Creating/declaring variables
- Naming conventions
- Arrays and collections
- Queues
- Stack
- Hashtable
- Enums
- Selection statements
- Iteration statements

Objective

The objective of this chapter is to make sure your C# skills are up to date by going through the C# language's origins, then tuning up on the inner workings of the C# language.

The origins of C#

C# has come a long way since being shipped with the first version of the .NET Framework in the year 2000. Originally named **Cool** (**C-like Object-Oriented Language**), C# has morphed into one of the most popular and powerful programming languages in the world.

We need to understand what makes the language tick before we can use it productively. Let us explore the common language concepts.

Variables and constants

A variable in programming terms is a named memory location in which you can store information. This information can be words (strings), whole numbers (integers), decimal numbers (floats), dates, yes/no values (Boolean), and so on. We will go into detail on data types a bit later.

A variable gets its value through a process called an assignment, and it looks like the next statement:

```
VariableName1 = Value;
```

Better examples follow:

```
VariableName2 = 2;
VariableName3 = "Hello!";
```

`VariableName2` now contains a value of **`2`**, and **`VariableName3`** contains the string **`Hello!`** The values stored inside variables can be changed at any time. For example, the value inside the **`VariableName2`** variable can be changed to 50, 100, or even 10,000.

A constant's value cannot change after it has been assigned a value. Perfect examples of constants are as follows:

Constant	Value
The number of months in a year	12

Minutes in an hour	60
Seconds in an hour	3,600
Days in a week	7
Number of seasons in a year	4

Table 2.1: *Constants*

Whenever there is a value that might be repeated throughout the code, it is always a good idea to convert that value into a constant.

Data types

A Data type describes, as the name implies, the type of data that can be stored inside a variable or a constant. Without designating a data type, you cannot create a variable. The data type also determines how much space in memory to set aside for a variable or a constant. Earlier in this chapter, we spoke about strings, Booleans, integers, and floats. These are just some of the few data types the C# language has.

The following table explains the type of values that can be stored as well as the size and range in memory the variable or constant will occupy:

Data type	Size in memory	Value
Boolean	Depends on implementing platform	True or False
Byte	1 byte	0 through 255
Char	2 bytes	Codepoints 0 through 65535
Date	8 bytes	0:00:00 AM on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	16 bytes	0 through +/− 79,228,162,514,264,337,593,543,950,335 (+/−7.9...E+28) with no decimal point. 0 through +/−7.9228162514264337593543950335 with 28 places to the right of the decimal
Double (double-precision floating-point)	8 bytes	−1.79769313486231570E+308 through −4.94065645841246544E−324 for negative values
		4.94065645841246544E−324 through 1.79769313486231570E+308 for positive values

Integer	4 bytes	−2,147,483,648 through 2,147,483,647
Long (long integer)	8 bytes	−9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
Object	4 bytes on a 32-bit platform. 8 bytes on a 64-bit platform	Any type can be stored in a variable of type object
SByte	1 byte	−128 through 127
Short (short integer)	2 bytes	−32,768 through 32,767
Single (single-precision floating-point)	4 bytes	−3.4028235E+38 through −1.401298E-45 for negative values. 1.401298E−45 through 3.4028235E+38 for positive values
String (variable-length)	Depends on implementation platform	0 to approximately 2 billion Unicode characters
UInteger	4 bytes	0 through 4,294,967,295
ULong	8 bytes	0 through 18,446,744,073,709,551,615
UShort	2 bytes	0 through 65,535

Table 2.2: *Data types*

Creating/declaring variables

In order to create a variable or constant and provide the necessary space in memory for it, we need to declare it. We do it by supplying the data type first, then a decent name, and then optionally an initial value (known as initializing a variable or constant). Here is an example:

```
int Age;
```

The preceding code creates an integer variable named **`Age`**. Let us take the example further:

```
int Age = 40;
```

The preceding code creates the same variable, but it also gives it a starting value of **`40`**. **`Age`** is a variable because its value can change. However, when we create a constant, its value cannot change during the course of the program. Here is an example:

```
const string FirstName = "Ockert"
```

```
const string LastName = "du Preez";
```

Two constants are created in the preceding code segment, and values are populated inside them. These values cannot change while the program is running. The moment you try to assign a new value to a constant, an error will be thrown.

Naming conventions

When naming an object, keep in mind that you are not allowed to use special characters or symbols in the object's name. Remember as well that names cannot start with a number. In general, there are two rules for capitalization in names. These are as follows:

- camelCase
- PascalCase

With camelCasing the first letter of an object or variable or constant starts with a small letter, and every subsequent word in the name starts with a capital letter. With PascalCasing, the name starts with a capital letter, and every subsequent word also has a capital letter. Needless to say: names are not allowed to contain spaces.

Arrays and collections

Arrays and collections are known as grouping structures. Both arrays and collections are capable of holding more than one value; the difference comes in how we want to make use of these values and how we want to refer to them.

Arrays

An array can hold multiple variables of the same type. These variables should be logically related to each other. The individual items of an array are called elements. Each array element has an index that starts at 0 and ends at the highest element index value. Examples are as follows.

Declare a single-dimensional array and populate it with values.

```
int[] ArrayName1 = new int[5];
ArrayName1[0] = 0;
```

```
ArrayName1[1] = 1;
ArrayName1[2] = 1;
ArrayName1[3] = 2;
ArrayName1[4] = 3;
```

Declare and set array element values populated with the first 5 Fibonacci numbers

```
int[] ArrayName2 = new int[] { 0, 1, 1, 2, 3 };
```

Alternative syntax to set array element values with the first 10 Fibonacci numbers

```
int[] ArrayName3 = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
```

The Fibonacci sequence in mathematics is when the next number is the sum of the previous two numbers.

Jagged arrays, multi-dimensional arrays are beyond the scope of this book.

Collections

Another way to manage a group of similar information is by making use of Collections. Arrays are usually the best option for creating and working with a fixed number of strongly typed objects. Collections are more flexible. Collections can grow and shrink dynamically as the needs of the application change. For the most common collections (such as the **`Hashtable`** class), we can assign a key to any object that was put into the collection so that we can quickly retrieve the object by using the key.

The following classes form part of the **`Collections`** namespace:

- ArrayList
- BitArray
- CaseInsensitiveComparer
- CaseInsensitiveHashCodeProvider
- CollectionBase
- Comparer
- DictionaryBase
- Hashtable

- Queue
- ReadOnlyCollectionBase
- SortedList
- Stack
- StructuralComparisons

Queues

When visiting a bank, a grocery store, or fast food takeaway, we have to wait in a queue for service. The .NET **`Queue`** class replicates this behavior in memory. The **`Queue`** class is what is known as a **FIFO** list (**First in, first out list**). This means that the first person in the queue will be helped first, and the last person will be helped last. With the items inside a **`Queue`**, the first item in the queue will be processed first, and the last will be processed last.

Store values inside a **`Queue`** by creating a **`Queue`** object and using its **`Enqueue`** method as follows:

```
private Queue MyQueue = new Queue();
private void Button1_Click(object sender, EventArgs e)
    {
       MyQueue.Enqueue("Item 1");
       MyQueue.Enqueue("Item 2");
       MyQueue.Enqueue("Item 3");
    }
```

Remove items from a **`Queue`** by looping through the **`Queue`** object and using its **`Dequeue`** method:

```
    private void Button1_Click(object sender, EventArgs e)
    {
    while (MyQueue.Count > 0) {
    object obj = MyQueue.Dequeue;
    Console.WriteLine("from Queue: {0}", obj);
    }
```

Stack

The .NET Stack class can be compared to a stack of vinyl records, books, or papers on top of each other on a desk. A **`Stack`** processes the last item first, and the first item in the stack is processed last. This is what makes it a **LIFO** (**last in, first out**) list.

Store values inside a **`Stack`** by creating a **`Stack`** object and using its **`Push`** method:

```
private Stack MyStack = new Stack();
private void Button2_Click(object sender, EventArgs e)
{
   MyStack.Push("Item 1");
   MyStack.Push("Item 2");
   MyStack.Push("Item 3");
}
```

Remove items from a **`Stack`** by looping through the **`Stack`** object and using its **`Pop`** method:

```
private void Button2_Click(object sender, EventArgs e)
{
   while (MyStack.Count > 0) {
    object obj = MyStack.Pop;
    Console.WriteLine("from Stack: {0}", obj);
}
}
```

Hashtable

The **`Hashtable`** collection class enables us to store a collection of information that relates to a certain key. It is a collection of key and value pairs that gets organized based on the hash code of each key.

Store values inside a **`Hashtable`** by creating a **`Hashtable`** object and giving each item a value:

```
private Hashtable MyHashtable = new Hashtable();
private void Button3_Click(object sender, EventArgs e)
{
   MyHashtable("0") = "Item 1";
   MyHashtable("1") = "Item 2";
   MyHashtable("2") = "Item 3";
}
```

Loop through the **Hashtable** by iterating through each **DictionaryEntry** object in the **Hashtable**:

```
private void Button3_Click(object sender, EventArgs e)
{
 foreach (DictionaryEntry entry in MyHashtable)
 {
  Console.WriteLine("{0} = {1}", pair.Key, pair.Value);
 }
}
```

The other collection types are beyond the scope of this book.

Enums

An enumeration is a list of named constants. Create an enumeration by using the **enum** keyword shown as follows:

```
enum Day {
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6
};
```

Make use of an **enum** by optionally creating a new variable and set it to the desired **Enum** item. Also, ensure that the correct data types are used and the necessary conversions; otherwise, an error would occur. In the following code segment, the variable **FirstDay** is set to the value of Sunday inside the Day Enumeration. This equals 0. **LastDay** equals 6. Because we are dealing with whole numbers here, we have to cast the enumeration value to the correct data type in order to be stored inside the variables.

An example of casting variables to the correct data type as follows:

```
private void Button4_Click(object sender, EventArgs e)
{
 int FirstDay = (int)Day.Sunday;
  int LastDay = (int)Day.Saturday;
```

```
        Console.WriteLine("Sunday = {0}", x);
       Console.WriteLine("Saturday = {0}", y);
    }
```

Selection statements

A selection statement causes the program control flow to change depending on whether a certain condition is true or false. The condition is the value we test for. We will briefly examine the **`if`** statements as well as the **`switch`** statement.

If

An if statement determines which statement to run based on the value of a Boolean (true/false) expression.

In the following example, the if statement tests the value of the **`Age`** variable. If **`Age`** is equal to or greater than **`40`**, a message will be displayed stating **`Old!`**. If the value is smaller than **`40`**, a message will be displayed stating **`Young!`**:

```
int Age = 40;
if (Age >= 40)
{
    MessageBox.Show("Old!");
}
else
{
    MessageBox.Show("Young!");
}
```

In the next example, we test for more than one condition—which is the **`Password`**, and the program can branch in many different ways. All the passwords supplied in the else **`if`** statements are valid and will allow entry, whereas if the **`Password`** variable does not contain any of those values, the **`else`** clause kicks in and denies entry:

```
string Password = "BpB";
if (Password == "VB.NET")
{
    MessageBox.Show("Welcome!");
```

```
}
else if (Password == "C#")
{
    MessageBox.Show("Welcome!");
}
else if (Password == ".NET")
{
    MessageBox.Show("Welcome!");
}
else if (Password == "BpB")
{
    MessageBox.Show("Welcome!");
}
else if (Password == "OJ")
{
    MessageBox.Show("Welcome!");
}
else
{
    MessageBox.Show("No Entry!");
}
```

Switch

The next selection statement is the **`switch`** statement. In principle, it works the same as an **`if`** statement with multiple else **`if`** clauses, but it is just simpler and easier to write. Here is the same example as the previous, but just using a **`switch`** statement:

```
string Password = "BpB";
switch (Password)
{
    case "VB.NET":
     MessageBox.Show("Welcome!");
    break;
    case "C#":
     MessageBox.Show("Welcome!");
    break;
```

```
    case ".NET":
     MessageBox.Show("Welcome!");
    break;
    case "BpB":
     MessageBox.Show("Welcome!");
    break;
    case "OJ":
     MessageBox.Show("Welcome!");
    break;
    default:
     MessageBox.Show("No Entry!");
    break;
}
```

A lot less typing! Depending on the valid password, an entry will be given. If none of the cases match the variable's value, the default clause will step in and display the **`No Entry`** message. We can make this even simpler! Seeing the fact that the same statement has to run if a valid password is supplied, we can do this:

```
switch (Password)
{
    case "VB.NET":
    case "C#":
    case ".NET":
    case "BpB":
    case "OJ":
     MessageBox.Show("Welcome!");
    break;
    default:
     MessageBox.Show("No Entry!");
    break;
}
```

This saves even more time, as we can supply all the conditions and then the common statement to be run.

Iteration statements

Iteration statements (loops) allow statements inside the loop to be repeated a number of times or until a certain condition becomes invalid. We can set the number of times the loop should repeat by explicitly stating it (as done with a for next loop), or we can have a condition determine the number of times the loop should repeat (as done with a while loop).

for loop

The **`for`** statement executes its inner-statements a number of times, specified in the condition section of the **`for`** loop block. An example as follows:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

In the previous example, the **`for`** loop is supplied with an initializer, a condition, and an iterator. The initializer is the starting point for the loop. As in the previous example, an integer variable named **`i`** is set to **`0`**. The condition is where **`i`** is tested if it is less than **`5`**. If **`i`** is any number from 0 to 4, the loop will execute the statement or statements inside the **`for`** block. The iterator is the last part where **`i`** get incremented by 1 with the use of the **`++`** operator.

The **`for`** loop, also known as a counter loop, cannot loop more than what was specified, but with the help of the **`continue`**, **`break`**, and **`throw`** statements, we can exit it early.

foreach loop

The foreach statement executes statements for each element in an instance of a type that implements either the **`System.Collections.IEnumerable`** or the **`System.Collections.Generic.IEnumerable<T>`** interfaces (such as the **`Hashtable`** we spoke about earlier). It sounds more complicated than what it is. Here is a small example:

```
var lstFibonacci = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13,
21, 34, 55 };
foreach (int element in lstFibonacci)
{
```

```
    Console.WriteLine($"Element: {element}");
}
```

A list of Fibonacci numbers gets created, and a **foreach** loop is used to iterate through each of the list's elements to display them in the Console window.

do loop

The **do** loop executes statements while a specified condition is true or until the specified condition becomes false. The condition is re-evaluated after each iteration of the **do** loop. Very important to remember is that a **do-while** loop executes at least one time regardless of whether the condition checked for is true or false. Here is an example:

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while (i < 5);
```

An integer variable is created. Inside the do loop, the value of **i** gets printed inside the Console window. The value of **i** gets increased by 1 in each iteration. Finally, the **while** condition tests the value of **i**. Be careful not to create an infinite loop. An infinite loop is a loop that does not exist. Why? Well, because the value being tested never changes. If we removed the part where **i** gets incremented, this **do** loop will never exit.

In order to exit a **do-while** loop, the **goto**, return or throw statements can be used.

while

The **while** loop executes statements while a specified condition evaluates to true. The condition is evaluated before each execution of the **while** loop, causing it to execute zero or more times. An example follows:

```
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);
```

```
    i++;
}
```

An integer variable named `i` is created. The test for condition starts the `while` loop. If the condition is true, the statements inside the loop get executed. If the condition is initially false (`i` is equal to or greater than 5), the `while` loop will never get executed.

We can exit a `while` loop by using the `goto`, return, or throw statements.

Conclusion

In this chapter, we brushed up on our C# skills. We learned about variables, data types, and constants. We had a look at collections and arrays and branching statements like loops and conditional statements

In the upcoming chapter, we will explore the new and improved features of C# 9 and highlight coming changes in C# 10

Points to remember

- Selection statements cause the program control flow to change depending on whether a certain condition is true or false.
- Iteration statements (loops) allow statements inside the loop to be repeated a number of times or until a certain condition becomes invalid.
- An array can hold multiple variables of the same type.
- A constant's value cannot change after it has been assigned a value.

Questions

1. Define the term: Selection statement.
2. Define the term: Iteration statements.
3. What is an Enum?
4. How does a Hashtable work?

Answers

1. A selection statement causes the program control flow to change depending on whether a certain condition is true or false.
2. Iteration statements (loops) allow statements inside the loop to be repeated a number of times or until a certain condition becomes invalid.
3. An Enumeration is a list of named constants.
4. The Hashtable collection class enables us to store a collection of information that relates to a certain key.

Key terms

- Variables
- Data types.
- Constants
- Collections
- Arrays
- Iteration Statements
- Selection statements
- Cool (C-like Object-Oriented Language)

References

- **Visual Studio In-Depth: https://www.amazon.com/Visual-Studio-2019-Depth-applications/product-reviews/9389328322/ref=cm_cr_dp_d_show_all_btm?ie=UTF8&reviewerType=all_reviews**

CHAPTER 3

New and Improved Features in C#

Introduction

It is quite amazing to see how C# has grown through the years. With every iteration of .NET, the C# language becomes more powerful and intelligent. C# is a real step forward. The C# language keeps modernizing and reinventing so that it can keep up with more and more modern applications.

Structure

The following topics will be covered:

- Records
- Init only setters
- Top-level statements
- New Interop features
- Fit and Finish features
- Pattern matching enhancements

Objective

In this chapter, we will learn the new and improved features of the C# language. These changes and updates will help the developers to continuously improve code performance as well as to write enterprise applications. Highlights of this chapter include the new records feature, new fit and finish features, and enhancements to pattern matching.

Records

With C# comes record types. The keyword **`record`** defines a reference type that provides built-in functionality for encapsulating data. The **`record`** types

can be created with immutable properties by using positional parameters or by using the standard property syntax, which is given as follows:

Positional parameters example

```
public record Student(string FirstName, string LastName, int
Age);
```

Standard property syntax example

```
public record Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public int Age { get; init; }
};
```

Mutable property syntax example

```
public record Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
};
```

Keep in mind that although records can be mutable, as shown previous example, they are intended for use with immutable data models.

Positional property definition syntax example

```
public record Student(string FirstName, string LastName, int
Age);
public static void Main()
{
    Student student = new("Ockert", "du Preez", 43);
    Console.WriteLine(student);
    // output: Student{ FirstName = Ockert, LastName = du
    Preez, Age = 43 }
}
```

Positional parameters are used in the previous example to declare properties of a record and to initialize the values when an instance is created. For more

information regarding the differences between immutable and mutable, please refer to the *References* section at the end of this chapter.

Let us take a look at another example, this time demonstrating *value equality*:

```
public record Student(string FirstName, string LastName,
string[] Subjects);
public static void Main()
{
    var subjects = new string[3];
    Student student1 = new("Ockert", "du Preez", subjects);
    Student student2 = new("Ockert", "du Preez", subjects);
    Console.WriteLine(student1 == student2); //True
    student1.Subjects[0] = "High Performance Enterprise Apps
    using C# and .NET";
    Console.WriteLine(student1 == student2); //True
    Console.WriteLine(ReferenceEquals(student1, student2));
    //False
}
```

What is going on here? Let us have a closer look.

In the previous code segment, we check for value equality. This means that two variables of a record type are always equal if the types, all property values, and field values match.

The first check equates to true because the values of **`FirstName`**, **`LastName`**, and the string array are the same. The second check also equates to true. The final check equates to false because the student1 object contains a subject that the **`student2`** object does not have.

One last example on records

Let us have a look at the following code segment:

```
public abstract record Student(string FirstName, string
LastName);
public record Instructor(string FirstName, string LastName,
string Subject)
    : Student(FirstName, LastName);
public static void Main()
{
```

```
    Student instructor = new Instructor("Ockert", "du Preez",
    "High Performance Enterprise Apps using C# and .NET");
    Console.WriteLine(instructor);
}
```

In the previous example the **`Instructor`** record inherits from the **`Student`** record. Records cannot inherit from a class and vice versa. The output of the previous code is **`Instructor { FirstName = Ockert, LastName = du Preez, Subject = High Performance Enterprise Apps using C# and .NET }`**.

Init only setters

Init only setters provide consistent syntax to initialize members of an object. With C#, we can create **`init`** accessors instead of set accessors for properties and indexers. Callers can then use property initializer syntax to set these values in creation expressions. After construction has been completed, the properties become read-only.

Here is a small example:

```
public struct Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public int Age { get; init; }
    public override string ToString() =>
    $"Name: {FirstName} Surname: {LastName} Age: {Age}";
}
```

After the declaration, let us set its values:

```
var student = new Student
{
    FirstName = "Ockert",
    Lastname = "du Preez",
    Age = 43
};
```

The values are now initialized and set. This means that we cannot changes these values later on in the application as it would result in an error.

Top-level statements

Top-level statements remove not unnecessary code but more the showing of unnecessary code from applications. Let us have a closer look.

Most apps have a **`Program.cs`** file containing the application **`Main`** method. It usually looks like the following:

```
using System;
namespace NameOfNamespace
{
    class Program
    {
     static void Main(string[] args)
     {
      Console.WriteLine("Hello BpB!");
     }
    }
}
```

When we observe this code closely, there seems to be a lot of boilerplate code that *does nothing*, as the only real line of code that is of any value here is: **`Console.WriteLine("Hello BpB!");`**

With top-level statements, we can replace all the boilerplate code with the following:

```
using System;
Console.WriteLine("Hello World!");
```

We can even make it shorter by excluding the using directive:

```
System.Console.WriteLine("Hello World!");
```

The downside here is that only one file in an application may use top-level statements.

New Interop features

New Interop features, which will be covered in more detail in *Chapter 12: Overcoming InterOp Challenges*, include the following:

- Native sized integers
- Function pointers

- Omitting the localsinit flag

Let us go into detail.

Native sized integers

A native-sized integer is an integer with size specific to the platform. On 32-bit hardware, it is 32 bits; on 64-bit hardware, it is 64 bits. `nint` (represented by `System.IntPtr`) and `nuint` (represented by `System.UIntPtr`) are new keywords that represent native signed and native unsigned integers. Signed integers can contain negative whole numbers, whereas unsigned integers can only hold non-negative whole numbers.

Function pointers

Function pointers provide an easy syntax to access the **Intermediate Language** (**IL**) opcodes, an IL instruction, `ldftn`, and `calli`. `ldftn` pushes unmanaged pointers to native code. `call` calls methods that are indicated on the evaluation stack. You can declare function pointers using the new `delegate*` syntax.

For more information regarding unsafe code, refer to the *References* section at the end of this chapter.

Omitting the localsinit flag

By adding the `System.Runtime.CompilerServices.SkipLocalsInitAttribute` attribute, we can tell the compiler not to produce the `localsinit` flag. This flag tells the **Common language Runtime** (**CLR**) to initialize all local variables to 0. By initializing all local variables to 0 has had a performance impact in some scenarios, for example, using `stackalloc`, which allocates a block of memory on the stack.

Fit and finish features

With the fit and finish features, we can omit the type inside a new expression when the created object's type is known. Let us have a look at an example:

```
private List<Subjects> subjects = new();
```

Here, C# already knows that subjects are a list of **`Subjects`**, so there is no need to declare it as:

```
private List<Subjects> subjects = new List<Subjects>;
```

Next, let us have a look at target-typed new in the next example:

```
public Student GetGrades(int Year, Subjects subjects)
//Signature of GetGrades method
var grades = student.GetGrades(2021, new()); //Call
```

Here, the type **`Subjects`** is known, and there is no need to include the type expression. We can also combine this feature with **`init`**, only properties to initialize a new object as follows:

```
Student student = new() { FirstName = "Ockert" };
```

Pattern matching enhancements

Pattern matching is a technique where an expression is tested to determine if it has certain characteristics. These characteristics include the following:

- Null checks
- Type tests
- Specific values

An example and explanation of each follows.

Null checks

We do Null checks to ensure that values are not null (empty, nothing). Here is a short example:

```
int? Age = 43;
if (Age is int number)
{
    Console.WriteLine($"The nullable int 'Age' has the value
    {number}");
}
else
{
```

```
    Console.WriteLine("The nullable int 'Age' doesn't hold a
    value");
}
```

In this declaration pattern, we test the type of the variable **`Age`** and assign it to a new variable number. Here is another example:

```
string? Subject = "High Performance Enterprise Apps using C#
and .NET";
if (Subject is not null)
{
    Console.WriteLine(Subject);
}
```

In the preceding example, we make use of them is not null check. This means if the **`Subject`** contains a value, it will be written to the console; else, it will not.

Type tests

A type test is used to test a variable to see if it matches a given type. For example:

```
public static T Subjects<T>(IEnumerable<T> subjects)
{
    if (subjects is IList<T> subs)
    {
     return subs[subs.Count];
    }
    else if (subjects is null)
    {
     throw new ArgumentNullException(nameof(subjects),
     "Subjects can't be null.");
    }
    else
    {
     int NumberOfSubjects = subjects.Count();
      return subjects.First();
    }
}
```

The previous code segment checks if a variable, subjects, is non-null and implements **`System.Collections.Generic.IList<T>`**. If the test succeeds, it makes use of the **`Count`** method of **`ICollection<T>`** to return the number of subjects.

Specific values

We can test variables for matches on specific values, for example:

```
public Subject GetSubjects(string sub) =>
  sub switch
  {
     "High Performance Enterprise Apps using C# and .NET" =>
     CalcGradesForSubjectHigh(),
     "Building Cross-Platform Modern Apps using Visual Studio
     Code (VS Code)" => CalcGradesForSubjectBuilding(),
     "JavaScript for Gurus" => CalcGradesForSubjectJavaScript(),
     "Visual Studio 2019 In-Depth" =>
     CalcGradesForSubjectVisual(),
     _ => throw new ArgumentException("Invalid string value for
     sub", nameof(sub)),
  };
```

In this test, we check what the value is of the parameter named sub, and depending on its value, we call a different method.

C# pattern matching enhancements

C# includes quite a few enhancements. They include the following:

- Parenthesized patterns
- Conjunctive and patterns
- Disjunctive or patterns
- Negated not patterns
- Relational patterns

Let us have a closer look at them.

Parenthesized patterns

Parentheses inside patterns enriches the syntax for patterns and make conditions clearer. Take a look at the following statement:

```
public static string Grade(float percentage) =>
    percentage is >= 50 and <= 74 or >= 75 and <= 100;
```

This is a small test to test the value of the percentage. If the percentage is between **50** and **74** or between **75** and **100**, then something must happen. But this reads difficult, so let us make use of parentheses to make it read better:

```
public static string Grade(float percentage) =>
    percentage is (>= 50 and <= 74) or (>= 75 and <= 100);
```

Now, it is clear what gets tested for.

Conjunctive and patterns

This pattern matches an expression when both patterns match the expression. For example:

```
Console.WriteLine(Grades(10)); //Between 0 and 14
Console.WriteLine(Grades(41)); //Between 40 and 59
Console.WriteLine(Grades(72)); //Between 60 and 100
static string Grades(float percentage) => percentage switch
{
    < 0 => " 0",
    >= 0 and < 15 => "Between 0 and 14",
    >= 15 and < 30 => "Between 15 and 29",
    >= 40 and < 60 => "Between 40 and 59",
    >= 60 and <= 100 => "Between 60 and 100",
    double.NaN => "Not a Number",
};
```

In this example, we check if a given number is in a range, for instance, greater than or equal to **15** **AND** less than **30**.

Disjunctive or patterns

This pattern matches an expression when either of patterns matches the expression, for example:

```
Console.WriteLine(GetFoodType("Tomato")); //Fruit
Console.WriteLine(GetFoodType("Beef")); //Meat
```

```
Console.WriteLine(GetFoodType("Cabbage")); //Vegetable
static string GetFoodType(string food) => food switch
{
    "Orange" or "Apple" or "Tomato" => "Fruit",
    "Pork" or "Beef" or "Mutton" => "Meat",
    "Cabbage" or "Carrot" or "Pumpkin" => "Vegetable",
    _ => throw new ArgumentOutOfRangeException(nameof(food),
    $"food must contain a value."),
};
```

If either **`Orange`**, **`Apple`**, or **`Tomato`** has been entered, then the output would be **`Fruit`**. If either **`Pork`**, **`Beef`**, or **`Mutton`** has been entered, then **`Meat`** will be output and the same principle as **`Vegetable`**.

Negated not patterns

This is a pattern that matches an expression when the negated pattern does not match the expression. Here is an example:

```
if (Subject is not null)
{
    //Does something only if Subject is not null
}
```

Relational patterns

This pattern matches an expression result with a constant. For example:

```
Console.WriteLine(Grades(100)); //Greater than or equals to
100
Console.WriteLine(Grades(77)); // Less than or equals to 99
static string Grade(float percentage) => percentage switch
{
    >= 100 => "Greater than or equals to 100",
    <= 99 => "Less than or equals to 99",
    >= 60 and <= 100 => "Between 60 and 100",
    double.NaN => "Not a Number",
};
```

In this example, we test the value of a given number with the relational operators: <, >, <=, or >=.

Conclusion

In this chapter, we explored the exciting new and enhanced features of C#. We covered records define a reference type that provides built-in functionality for encapsulating data. We also looked at top-level statements to remove unnecessary overhead. Finally, we had a look at the Interop changes to integers and pattern enhancements.

In the upcoming chapter, we will start off the *Code performance* section, and we will jump right into data types and why they are crucial to building high-performance applications

Points to remember

- Relational patterns match expressions resulting in constants.
- Parentheses inside patterns enrich the syntax for patterns and make conditions clearer.
- A type test is used to test a variable to see if it matches a given type.
- A native-sized integer is an integer with size specific to the platform.

Questions

1. What do Init only setters do?
2. Define the term Positional Parameters.
3. What is the purpose of the localsinit flag?
4. Explain the purpose of Fit and Finish features.

Answers

1. Init only setters provide consistent syntax to initialize members of an object.
2. Positional parameters are used in the previous example to declare the properties of a record and to initialize the values when an instance is created.
3. This flag tells the Common language Runtime (CLR) to initialize all local variables to 0.

4. With the fit and finish features, we can omit the type inside a new expression when the created object's type is known.

Key terms

- Function pointers
- Null checks
- Conjunctive and patterns
- Disjunctive or patterns
- Negated not patterns

References

- **Immutable vs. Mutable: https://www.infoworld.com/article/3564161/how-to-use-immutability-in-csharp.html**
- **Unsafe code: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code**

CHAPTER 4

Using Data Types

Introduction

Knowledge of proper usage of data types is crucial, especially in larger applications. Knowing how much each variable holds in memory can help plan programs accordingly. If data types are not used correctly, they can cause bottlenecks (a bottleneck occurs when the capacity of an application or a computer is limited by a single component), which slows down performance and has the lowest throughput of all instructions of a program) or wrong results. This results in program failure and, ultimately, user adoption failure.

Structure

The following topics are to be covered:

- Data types defined
- Choosing the right data type for the right purpose

Objective

We will learn what data types are. We will also look into each data type accommodating range as well as their space occupation in memory.

Data types defined

Data types are attributes associated with pieces of data that tell a computer system how to interpret their values. Data types are necessary to ensure the desired object contains the correct value, so, in a sense, it describes the information that needs to go into the object and what can be done with it.

This is vital because when a wrong data type is used in a scenario, it can cause exceptions galore. For example, when an object requires a numeric

value, we cannot attempt to put in a string (set of characters) value, as it will produce an error. The same goes with an object expecting an integer (a whole number), but we attempt to put in a decimal value. This will result in the wrong result and even possibly an exception.

There are two main categories of C# data types. They are as follows:

- Value types
- Reference types

Let us have a look at their differences.

Value types

Variables of value types directly contain their data. This means that each value type variable has its own copy of the data, and any operations on one value type variable will not affect the other value type variables. Value type variables are copied into a value type variable when values are assigned when arguments are passed to methods and the return of the method results.

Here is a short example of showing value type variables in action:

```
using System;
public struct MonthsAndYearsStruct
{
    public int Months;
    public int Years;
    public ChangeValue(int m, int y) => (Months, Years) = (m,
    y);
    public override string ToString() => $"({Months},
    {Years})";
}
public class Program
{
    public static void Main()
    {
     var months = new MonthsAndYearsStruct(7, 1978);
     var years = months;
     MonthsAndYearsStruct.Years = 2021;
```

```
        Console.WriteLine($"{nameof(months)} after
        {nameof(years)} is modified: {months}");
        Console.WriteLine($"{nameof(years)}: {years}");
        Display(years);
        Console.WriteLine($"{nameof(years)} after passing to a
        method: {years}");
      }
      private static void Display(MonthsAndYearsStruct my)
      {
        my.Months = 12;
        Console.WriteLine($"Months mutated in a method: {my}");
      }
}
// Expected output:
// months after years is modified: (7, 1978)
// years: (7, 2021)
// Months mutated in a method: (12, 1978)
// years after passing to a method: (7, 2021)
```

Here, we continuously change the value of integer variables on different occasions, each not influencing the other. The following table shows the C# value types:

C# type keyword	type	Description
`bool`	`System.Boolean`	Value can either be true or false.
`byte`	`System.Byte`	Value can be any number between 0 and 255
`sbyte`	`System.SByte`	Value can be any number from –28 to 127
`char`	`System.Char`	Value can contain a Unicode UTF-16 character. If you are unfamiliar with Unicode or UTF, please refer to the *References* section at the end of this chapter.
`decimal`	`System.Decimal`	Value can be any number from ±1.0 × 10−28 to ±7.9228 × 1028
`double`	`System.Double`	Value can be any number from ±5.0 × 10−324 to ±1.7 × 10308
`float`	`System.Single`	Value can be any number from ±1.5 × 10−45 to ±3.4 x 1038

`int`	`System.Int32`	Value can be any number between –2,147,483,648 to 2,147,483,647
`uint`	`System.UInt32`	Value can be any number between 0 to 4,294,967,295
`nint`	`System.IntPtr`	Please refer to *Chapter 3: New and Improved features in C#*
`nuint`	`System.UIntPtr`	
`long`	`System.Int64`	Value can be any number between –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`ulong`	`System.UInt64`	Value can be any number between 0 to 18,446,744,073,709,551,615
`short`	`System.Int16`	Value can be any number between –32,768 to 32,767
`ushort`	`System.UInt16`	Value can be any number between 0 to 65,535

Table 4.1: *Value data types*

Now, let us have a look at reference types.

Reference types

Variables of reference type store references to their data. This means that they do not physically hold the value, just a reference. By being a reference type, two variables can reference the same object, and operations on one variable can affect the object referenced by the other variable.

Following are short examples showing some reference type variables in action:

Object type example

The following example demonstrates the usage of **`Object`** types and some of its most popular built-in methods:

```
using System;
// The Point class is derived from System.Object.
class Point
{
    public int ptX, ptY;
    public Point(int x, int y)
    {
     this.ptX = x;
```

```
      this.ptY = y;
    } //Point Class is Derived (inherited) from System.Object
    public override bool Equals(object objToTest) //Test if
    Objects are the same
    {
      if (objToTest.GetType() != this.GetType()) return false;
      var objXY = (Point) objToTest;
      return (this.ptX == objXY.x) && (this.ptY == objXY.y);
    }
    // Return the XOR of the x and y fields.
    public override int GetXOR()
    {
      return ptX ^ ptY; XOR ptX and ptY
    }
    public override String ToString() //Object's value as
    String
    {
      return $"({ptX}, {ptY})";
    }
    public Point Copy()
    {
      return (Point) this.MemberwiseClone(); //Define a
      ShallowCopy
    }
}
public sealed class MyApplication
{
    static void Main()
    {
      var ptPoint1 = new Point(78,7);
      var ptPoint2 = ptPoint1.Copy();
      var ptPoint3 = ptPoint1;
      Console.WriteLine(Object.ReferenceEquals(ptPoint1,
      ptPoint2));
      Console.WriteLine(Object.Equals(ptPoint1, ptPoint2));
      Console.WriteLine(Object.ReferenceEquals(ptPoint1,
      ptPoint3));
```

```
      Console.WriteLine($"ptPoint1's value is:
      {ptPoint1.ToString()}");
    }
}
// Expected output:
//
// False
// True
// True
// ptPoint1's value is: (78, 7)
//
```

Next, let us have a look at String types

String type example

The following example makes use of the **`String`** object's **`length`** property to determine the length of a string. This example shows three different ways to achieve this:

```
string strTemp = "Ockert du Preez"; //15
Console.WriteLine("1) The length of '{0}' is {1}", strTemp,
strTemp.Length);
Console.WriteLine("2) The length of '{0}' is {1}", "Ockert",
"Ockert".Length); //6
int intLength = strTemp.Length;
Console.WriteLine("3) The length of '{0}' is {1}", strTemp,
intLength);
// Expected output:
//    1) The length of 'Ockert du Preez' is 15
//    2) The length of 'Ockert' is 6
//    3) The length of 'Ockert du Preez' is 15
```

We will delve deeper into each type a bit later. The following table shows the C# reference types:

C# type keyword
class
`interface`
`delegate`

`record`
`dynamic`
`object`
`string`

Table 4.2: *Reference types*

Choosing the right data type for the right purpose

Choosing the right data type for the right purpose does not just involve knowing what each data type can do and hold, but it also entails how much memory can be saved by choosing the right purpose. This rings true, especially with value types. Developers (and beginners) can be lazy creatures and simply choose a numeric data type, not necessarily the correct one, just to get done with the current module or problem.

This causes the application, service, or website being written to use much more resources than what it should. Let us have a look at how we can possibly overcome this. First, let us have a look at the memory each data type takes up. The next table shows the value types along with the space in memory they occupy:

C# type keyword	.NET type	Space in memory
`bool`	`System.Boolean`	2 bytes
`byte`	`System.Byte`	1 byte
`sbyte`	`System.SByte`	1 byte
`char`	`System.Char`	2 bytes
`decimal`	`System.Decimal`	16 bytes
`double`	`System.Double`	8 bytes
`float`	`System.Single`	4 bytes
`int`	`System.Int32`	4 bytes
`uint`	`System.UInt32`	4 bytes
`long`	`System.Int64`	8 bytes
`ulong`	`System.UInt64`	8 bytes
`short`	`System.Int16`	2 bytes

ushort	System.UInt16	2 bytes

Table 4.3: *Space in memory for value types*

Table 4.3 provides a clear picture of how much space in memory gets allocated upon variable creation. In a very small application, it should not be much of a difference as the application is quite small, depending, however, on system resources.

In larger applications, memory will definitely become an issue. There are tools and profilers out there that can help detect these problems, which we will cover throughout the course of this book. Let us have a look at a small example:

```
using System;
  public class FactorialExample
  {
     public static void Main(string[] args)
      {
       int i;
       int Factorial = 1;
       int InputNumber;
       Console.Write("Enter a Number: ");
       InputNumber = int.Parse(Console.ReadLine());
       for(i = 1; i <= InputNumber; i++)
       {
      Factorial = Factorial * i;
       }
       Console.Write("Factorial of " + InputNumber + " is: " +
       Factorial);
     }
  }
```

In the previous example, we deal with factorials. A factorial is the product of all positive integers less than or equal to a given positive integer. For example:

So, for example: 4! (factorial) = 4 * 3 * 2 * 1 = 24

6! (factorial) = 6 * 5 * 4 * 3 * 2 * 1 = 720

7! (factorial) = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040

78! (factorial) = 1132428117 8206297831 4575211587 3204622873 1749579488 2519900489 6282566883 5325234200 7662450862 1317734400 0000000000 000000

For smaller numbers up to the number 5 will be well suited for a byte. A factorial of 8 is well suited for a short, but a factorial of 9 will have to be an integer data type. Then, have a look at the factorial of 78. That is a massive number with 116 digits!

In an example such as the previous, it is important to limit the input the user can give. If the user is able to enter any number, the program will throw an exception. Also, depending on the user's PC's resources, computing a large number such as the factorial of 78 will take a lot of processing time, CPU, and memory resources.

Let us take a look at another simple example:

```
using System;
 public class FactorialExample
 {
    public static void Main(string[] args)
     {
      long Age;
      int Months;
      Console.Write("Enter an Age: ");
      Age = long.Parse(Console.ReadLine());
      Console.Write("Enter a Month: ");
      Month = int.Parse(Console.ReadLine());
      Console.Write("Month = " + Month + " Age = " + Age);
    }
 }
```

What is wrong here?

The variable **`Age`** is declared as a long, and the variable **`Month`** is declared as an integer. These are the wrong data types. As seen in *table 4.1*, the ranges for a long and an *int* are much bigger than what was input. Also, as seen in *table 4.3*, the size of a long and an integer is 8 bytes and 4 bytes in memory, respectively. If we were to change the data types to a byte, in this case, the two variables would occupy 2 bytes in memory collectively, whereas they occupy 12 bytes in memory. Now, a saving of 10 bytes does not seem like

much; now, imagine this program has a hundred different classes, each making use of the wrong data types, by using the correct data types might save hundreds of bytes!

Conversion

Type conversion or casting happens when the value of one value of a data type is converted to another. When we convert, for example, numeric data types, C# does *Automatic Type Conversion*. If the two data types of values are not comparable, then an *Explicit Type Conversion* should happen; that is, they need to be converted explicitly.

An example of *Implicit Type Conversion* follows next:

```
using System;
namespace ImplicitCastingEx{
class CastImplicitly {
    public static void Main(String []args)
    {
     int intAge = 43;
     long lngAge = intAge; //This works because int is smaller
     as long and fits nicely in it
     float fltAge = lngAge; //Also works nicely as long and
     fits nicely in float
     Console.WriteLine("Int value " + intAge);
     Console.WriteLine("Long value " + lngAge);
     Console.WriteLine("Float value " + fltAge);
    }
    }
}
//Expected Output:
//Int value 78
//Long value 78
//Float value 78
```

An example of *Explicit Type Conversion* follows next:

```
using System;
namespace ExplicitCastingEx{
class CastImplicitly {
    public static void Main(String []args)
```

```
    {
     double dblPie = 3.14159265359;
     int intPie = (int)dblPie; //Explicit Conversions with
     (int)
     Console.WriteLine("Value of intPie is " + intPie);
    }
  }
}
//Expected Output:
//Int value 3, because casting from double to integer removes
the decimal
```

Conclusion

In this chapter, we explored data types. We learned why size does matter, especially in memory, and what each data type's range is. We also had a quick look into the conversion process of data types: implicit casting and explicit casting.

In the upcoming chapter, we will continue where we left off in *Chapter 3: Improved features in C#* and concentrate specifically on pattern matching.

Points to remember

- Data types have different ranges to suit any need.
- Data types occupy different amounts of space in memory.
- Automatic type conversion happens when two data types are compatible.
- Automatic type conversion happens when two data types are not compatible.

Questions

1. Name three value data types.
2. What is the difference between Reference data types and Value data types?
3. Define the term uint?

Answers

1. Refer to *table 4.1*
2. Variables of value types directly contain their data, and variables of reference type store references to their data.
3. Value can be any number between 0 to 4,294,967,295

Key terms

- Value types
- Reference types
- Implicit conversion
- Explicit conversion

References

- **Unicode: https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-encoding-introduction**
- **ShallowCopy: https://www.c-sharpcorner.com/UploadFile/56fb14/shallow-copy-and-deep-copy-of-instance-using-C-Sharp/**
- **Bitwise and Shift Operators: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators**

CHAPTER 5

Enhancing Pattern Matching

Introduction

When working with code, it is essential to know precisely what the particular language can do and what features it has. C# has always been capable of doing good pattern matching, but C# has taken pattern matching to the next level.

Structure

The following topics are to be covered:

- Parenthesized patterns
- Conjunctive and patterns
- Disjunctive or patterns
- Negated not patterns
- Relational patterns

Objective

The objective of this chapter is to introduce Pattern Matching in C#. We will have a look at various patterns, including parenthesized, relational, and "*and*, *or*, and *not*" patterns.

Parenthesized patterns

Parentheses inside patterns enriches the syntax for patterns and make conditions clearer. Take a look at the following statement:

```
public static string Grade(float percentage) =>
    percentage is >= 50 and <= 74 or >= 75 and <= 100;
```

This is a small test to test the value of the percentage. If the percentage is between `50` and `74` or between `75` and `100`, then something must happen. But this reads difficult, so let us make use of parentheses to make it read better:

```
public static string Grade(float percentage) =>
    percentage is (>= 50 and <= 74) or (>= 75 and <= 100);
```

Now, it is clear what gets tested for.

Conjunctive and patterns

This pattern matches an expression when both patterns match the expression. For example:

```
Console.WriteLine(Grades(10)); //Between 0 and 14
Console.WriteLine(Grades(41)); //Between 40 and 59
Console.WriteLine(Grades(72)); //Between 60 and 100
static string Grades(float percentage) => percentage switch
{
    < 0 => " 0",
    >= 0 and < 15 => "Between 0 and 14",
    >= 15 and < 30 => "Between 15 and 29",
    >= 40 and < 60 => "Between 40 and 59",
    >= 60 and <= 100 => "Between 60 and 100",
    double.NaN => "Not a Number",
};
```

In this example, we check if a given number is in a range, for instance, greater than or equal to 15 AND less than 30.

Disjunctive or patterns

This pattern matches an expression when either of patterns matches the expression, for example:

```
Console.WriteLine(GetFoodType("Tomato")); //Fruit
Console.WriteLine(GetFoodType("Beef")); //Meat
Console.WriteLine(GetFoodType("Cabbage")); //Vegetable
static string GetFoodType(string food) => food switch
{
    "Orange" or "Apple" or "Tomato" => "Fruit",
    "Pork" or "Beef" or "Mutton" => "Meat",
    "Cabbage" or "Carrot" or "Pumkin" => "Vegetable",
    _ => throw new ArgumentOutOfRangeException(nameof(food),
    $"food must contain a value."),
};
```

If either **`Orange`**, **`Apple`**, or **`Tomato`** has been entered, then the output would be **`Fruit`**. If either **`Pork`**, **`Beef`**, or **`Mutton`** has been entered, then the output will be **`Meat`** and the same principle for **`Vegetable`**.

Negated not patterns

This is a pattern that matches an expression when the negated pattern does not match the expression. Here is an example:

```
if (Subject is not null)
{
    //Does something only if Subject is not null
}
```

Relational patterns

This pattern matches an expression result with a constant. For example:

```
Console.WriteLine(Grades(100)); //Greater than or equals to 100
Console.WriteLine(Grades(77)); // Less than or equals to 99
static string Grade(float percentage) => percentage switch
{
    >= 100 => "Greater than or equals to 100",
    <= 99 => "Less than or equals to 99",
    >= 60 and <= 100 => "Between 60 and 100",
    double.NaN => "Not a Number",
};
```

In this example, we check the value of a given number with the relational operators: **<**, **>**, **<=**, or **>=**.

Practical

Let us apply all of these practically. We will start off by creating a Console application, then add each section's code inside of it. Follow these steps:

1. Open Visual Studio 2019.
2. Create a **`New Project`**, as shown in the following *figure 5.1*:

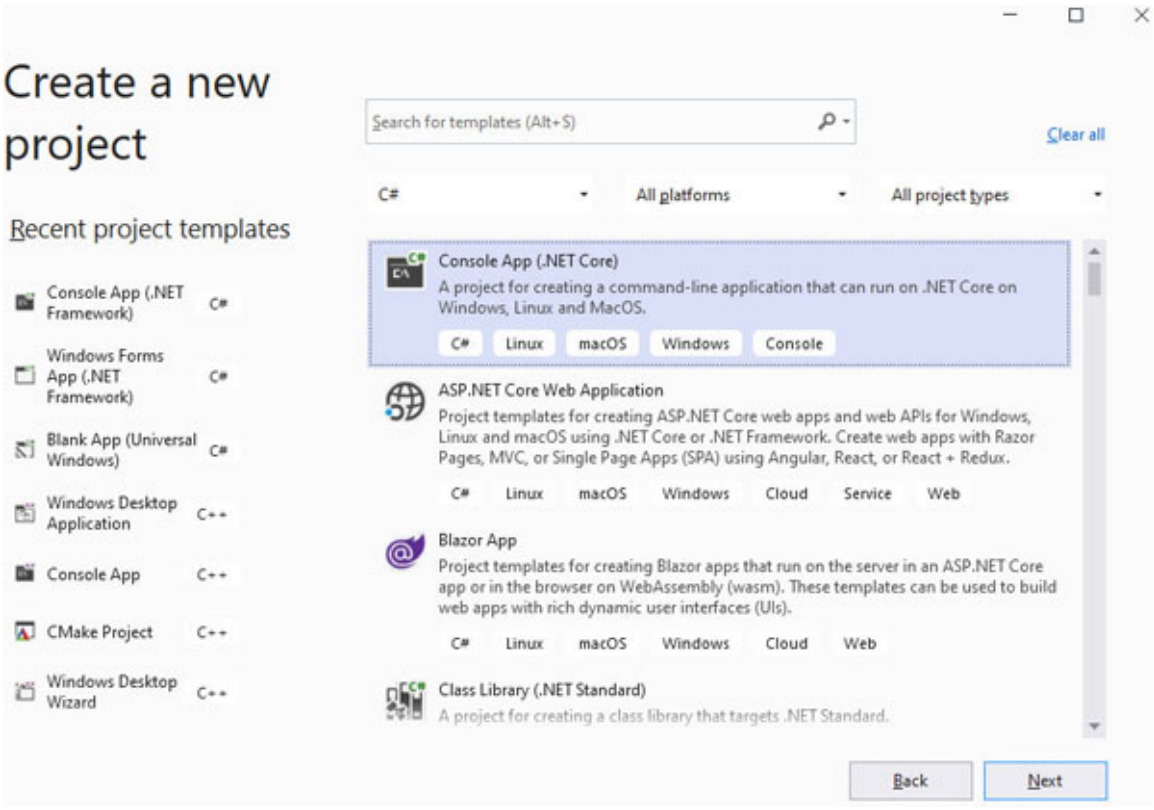


Figure 5.1: *Create a new Project*

3. Name the new project **`Pattern_Ex`**.
4. Edit the **`Program.cs`** file to resemble the following code:

```
using System;
namespace Pattern_Ex
{
 class Program
 {
  static void Main(string[] args)
  {
   char c = 't';
   if ((c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z')))
    Console.WriteLine("Alphabetical Letter");
  }
 }
}
```

There may be errors due to the fact that we are using .NET 5 or higher to be compatible with C#.

To fix this, follow these steps:

1. Open the **`Properties`** page for the project by clicking **`Project`** | **`Project Name`** (in our case **`Pattern_Ex`**) | **`Properties`**.
2. Ensure that the **`Application`** tab is selected.
3. Click on the **`Target Framework`** dropdown. Notice that .NET Frameworks 5 and 6 are not shown in the list.
4. Click on **`Install other Frameworks`**:

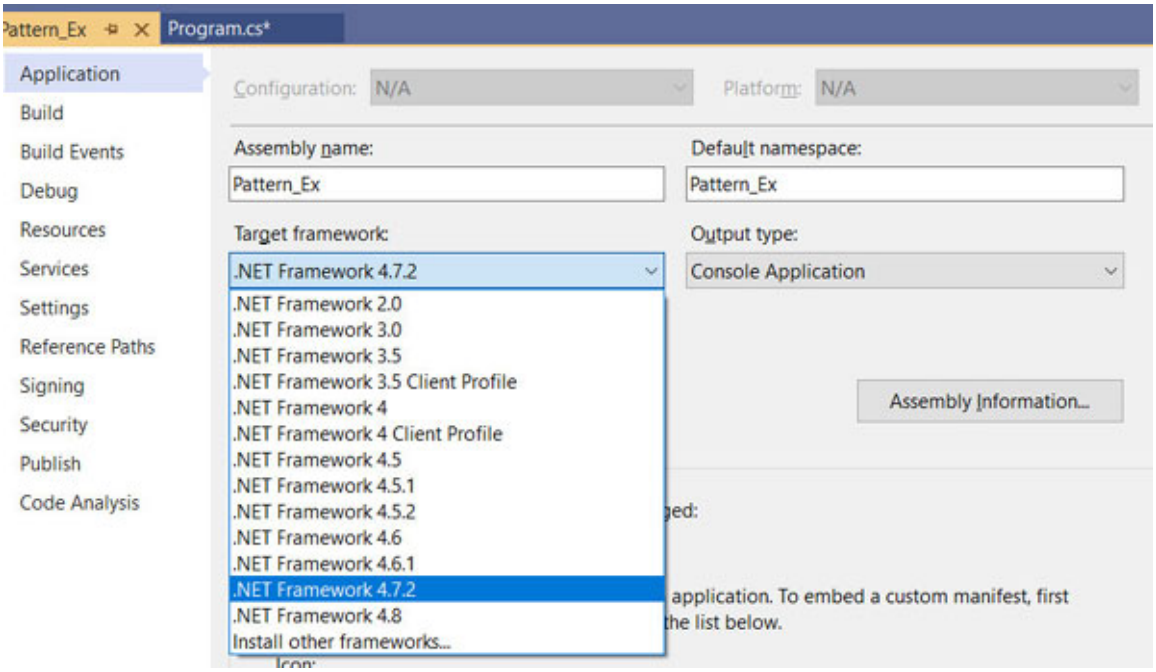


Figure 5.2: *Install other frameworks*

This will open up a browser page with the URL: **https://dotnet.microsoft.com/download/visual-studio-sdks?utm_source=getdotnetsdk&utm_medium=referral** allowing us to download the latest version of the.NET SDK (Software Development Kit):

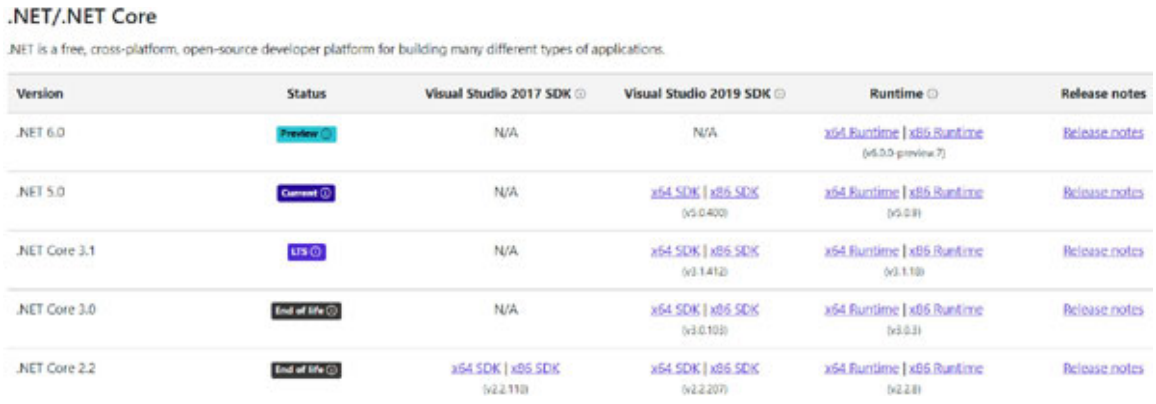


Figure 5.3: *.NET SDK download page*

5. Choose the version (for example: .NET 5 or .NET 6) to install and click on it to install it.
6. After the download has been completed, double-click on it to start the installation process:

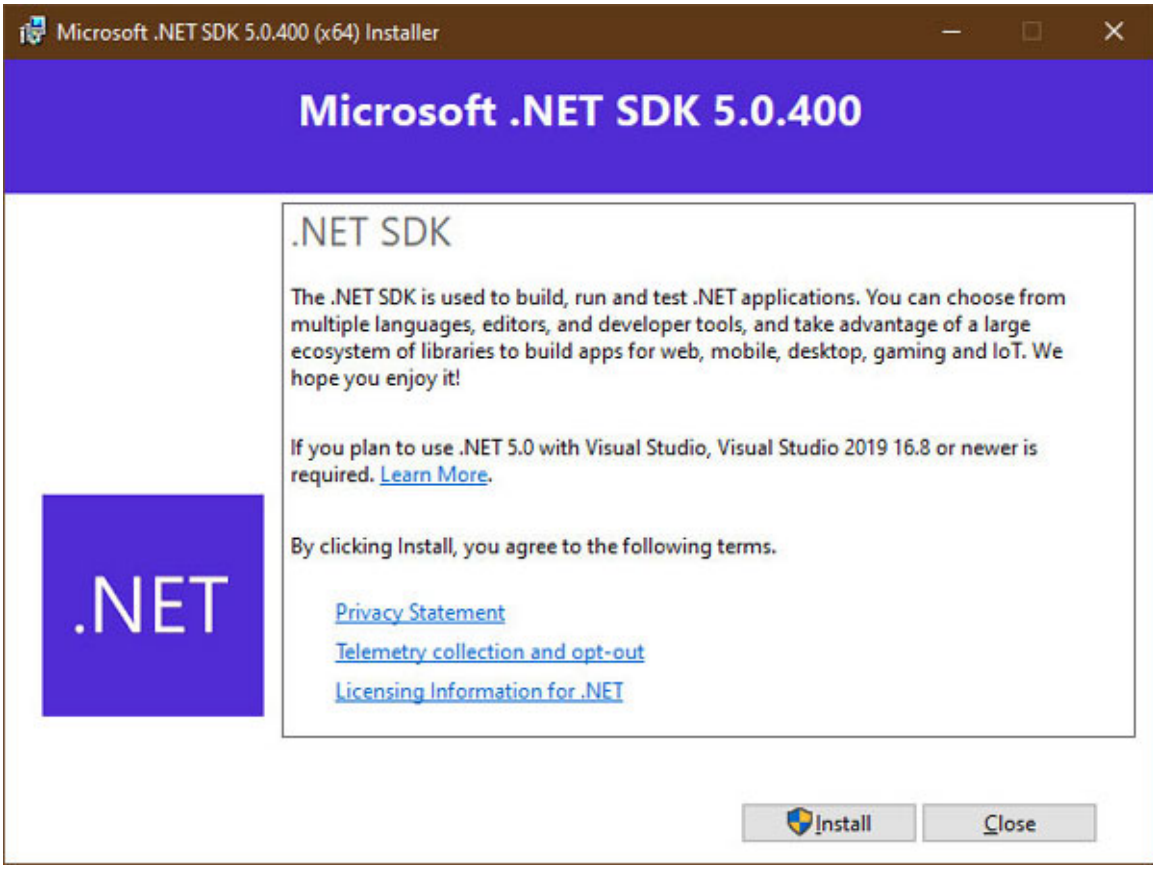


Figure 5.4: *Install .NET SDK*

7. Click **`Install`**. A screen similar to *figure 5.5* will display:

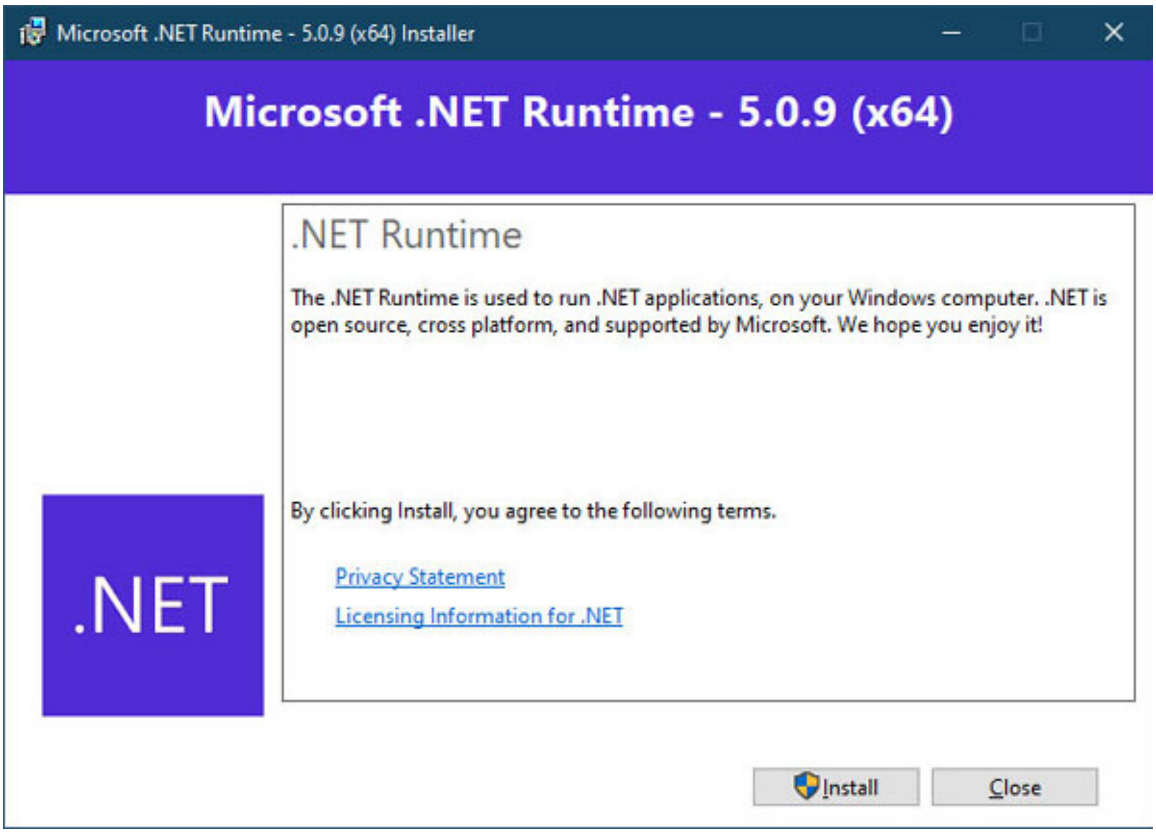


Figure 5.5: *Install .NET runtime*

8. After the installation, click **`Close`**.
9. Back on the **`Downloads`** page, download the Runtime as well.
10. Click **`Install`**.
11. If necessary, restart Visual Studio.

In Visual Studio 2019, ensure that the **`Options`** that enable preview versions of the .NET Core SDK are enabled by selecting: **`Tools`** | **`Options`** | **`Preview Features`**.

In Visual Studio 2019 preview, the previous step can be skipped as all the preview features are already installed.

Conclusion

In this chapter, we concentrated on the most anticipated pattern-matching features of C#. They are quite powerful and versatile, as we saw. In the upcoming chapter, we will learn about the various types of Collections, boxing, unboxing, and thread safety.

Points to remember

- Pattern matching is a technique where an expression is tested in order to determine if it has certain characteristics.
- Parentheses inside patterns enrich the syntax for patterns and make conditions clearer.
- Conjunctive and pattern match an expression when both patterns match the expression.
- .NET Framework 5 and higher is needed to be able to achieve these types of Pattern Matching.

Questions

1. Explain the term: Disjunctive or patterns.
2. Define the term: Relational patterns.
3. Explain: Negated not patterns

Answers

1. This pattern matches an expression when either of the patterns matches the expression.
2. This pattern matches an expression result with a constant.
3. This is a pattern that matches an expression when the negated pattern does not match the expression.

Key terms

- Parenthesized patterns
- Conjunctive and patterns
- Disjunctive or patterns
- Negated not patterns
- Search paths

References

- **C#: Pattern Matching in Switch Expressions: https://www.thomasclaudiushuber.com/2021/02/25/c-9-0-pattern-matching-in-switch-expressions/**
- **Pattern matching overview: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching**

CHAPTER 6

Using Collections Properly

Introduction

Collections are essential for any application. Collections can store lists of information in various powerful ways, depending on needs. The question though is to know when to use them and why to use them.

Structure

The following topics are to be covered:

- Collection types
- Boxing issues
- Thread safety
- Guidelines

Objective

The objective of this chapter is to learn how to use Collections properly. We will learn about the various types of collections and their purpose. We will then have a look at some guidelines and thread-safety and boxing in collections.

Collection types

There are quite a few different Collection types available in .NET. *Table 6.1* explains them shortly, and then later, we will delve deeper into each a bit later.

Collection type	Short description
`ArrayList`	An array that can increase size dynamically
`BitArray`	An array of Boolean values representing bits

`CaseInsensitiveComparer`	Determines if two objects, irrespective of the casing, are equal
`CaseInsensitiveHashCodeProvider`	Uses a hashing algorithm to ignore the case of strings
`CollectionBase`	Strongly typed Collection
`Comparer`	Case-sensitive two-object comparer
`DictionaryBase`	Strongly typed Collection of key/value pairs
`Hashtable`	Collection of key/value pairs based on each key's hash code
`Queue`	First-in and first-out Collection
`ReadOnlyCollectionBase`	Strongly typed non-generic read-only Collection
`SortedList`	Collection of key/values, which are accessible by key or index
`Stack`	Last-in-first-out collection
`StructuralComparisons`	Performs structural comparisons of two collections
`List`	Strongly typed list accessible by index
`LinkedList`	Doubly linked list
`Dictionary`	Collection of keys and values
`ConcurrentQueue`	Thread-safe first in-first out Collection
`ConcurrentStack`	Thread-safe last in-first out Collection
`ImmutableQueue`	Immutable (once the queue is created, it cannot change) Queue.
`ImmutableStack`	Immutable Stack
`StringCollection`	Collection of Strings
`ImmutableArray`	Immutable Array
`ListDictionary`	Specialized list. More on this a bit later in this chapter
`StringDictionary`	Hashtable with strongly typed keys and values of type string
`SortedDictionary`	Sorted key and value pairs
`ImmutableHashSet`	Immutable Hash set

`ImmutableDictionary`	Immutable Dictionary
`ImmutableSortedSet`	Immutable Sorted Set
`ImmutableSortedDictionary`	Immutable Sorted Dictionary
`NameObjectCollectionBase`	Base for string keys and object values
`NameValueCollection`	Collection of keys and values of type string
`KeyedCollection`	Base for a collection whose keys are embedded in the values
`ImmutableList`	Immutable List

Table 6.1: *.NET Collection types*

Some small examples of the most frequently used collections follow. Note that all these examples can be done in Visual Studio by creating a Console application.

ArrayList

An example of an **`ArrayList`** is as follows:

```
using System;
using System.Collections;
public class ArrayList_Ex {
 public static void Main() {
     ArrayList aList = new ArrayList(); //Create and
     initialize new ArrayList.
     aList.Add("Summer");
     aList.Add("Autumn");
     aList.Add("Winter");
     aList.Add("Spring");
     DisplayValues(aList);
 }
 public static void DisplayValues( IEnumerable tmpList ) {
     foreach (Object obj in tmpList)
    Console.Write("{0}", obj);
     Console.ReadLine();
 }
}
```

An **ArrayList** is created, and items are stored into it. Finally, with the help of the **DisplayValues** method, the **Console** prints the **ArrayList**'s contents.

BitArray

An example of a **BitArray** is as follows:

```
using System;
using System.Collections;
public class BitArray_Ex {
 public static void Main() {
     int[] arrInts = new int[5] {3, 4, 5, 6, 7};
     BitArray bArray = new BitArray(arrInts);
 }
}
```

We create an array of integers, then populate the **BitArray** with the Integer array.

CaseInsensitiveComparer and CaseInsensitiveHashCodeProvider

An example of the usage of these two collections is as follows:

```
using System;
using System.Collections;
using System.Globalization;
public class Comparer_Ex {
 public static void Main() {
     Hashtable hashNormal = new Hashtable(); //Create a
     Hashtable
     hashNormal.Add("ONE", "Summer");
     hashNormal.Add("TWO", "Autumn");
     hashNormal.Add("THREE", "Winter");
     hashNormal.Add("FOUR", "Spring");
     //Hashtable with case-insensitive code provider and case-
     insensitive comparer
     Hashtable hashCompare = new Hashtable( new
     CaseInsensitiveHashCodeProvider(), new
     CaseInsensitiveComparer());
```

```
        hashCompare.Add("ONE", "Summer");
        hashCompare.Add("TWO", "Autumn");
        hashCompare.Add("THREE", "Winter");
        hashCompare.Add("FOUR", "Spring");
        // Search for a key in both Hashtables
        Console.WriteLine("ONE is in hashNormal: {0}",
        hashNormal.ContainsKey("one"));
        Console.WriteLine("ONE is in hashCompare: {0}",
        hashCompare.ContainsKey("one"));
    }
}
```

We create two Hashtables and store the same information in both. Then, we search for a key inside both Hashtables. The results would be false and true, respectively. The reason for this is that the first Hashtable expects a key with the exact case, whereas the second ignores the casing of the key being searched for and returns true.

Queue

An example of how to store items in a queue and access them is as follows:

```
using System;
 using System.Collections;
 public class Queue_Ex {
    public static void Main() {
       Queue qQueue = new Queue(); //Create and initialize a
       new Queue
       qQueue.Enqueue("Summer");
       qQueue.Enqueue("Autumn");
       qQueue.Enqueue("Winter");
       qQueue.Enqueue("Spring");
       DisplayValues(qQueue);
    }
    public static void DisplayValues(IEnumerable tmpQue) {
       foreach (Object obj in tmpQue)
     Console.Write("{0}", obj);
       Console.WriteLine();
    }
```

```
  }
```

We use the **`Enqueue`** method to add items to a **`Queue`**, then a for loop to loop through the items and display them with the help of the **`DisplayValues`** method.

SortedList

A quick example of a **`SortedList`** is as follows:

```
using System;
using System.Collections;
public class SortedList_Ex {
 public static void Main() {
      SortedList sList = new SortedList(); //Create and
      initialize a new SortedList.
      sList.Add("Third", "Summer");
      sList.Add("First", "Autumn");
      sList.Add("Fourth", "Winter");
      sList.Add("Second", "Spring");
     DisplayValues(sList);
 }
 public static void DisplayValues(SortedList tmpList) {
     for ( int i = 0; i < tmpList.Count; i++ ) {
      Console.WriteLine( "\t{0}:\t{1}", tmpList.GetKey(i), t
      mpList.GetByIndex(i) );
     }
     Console.ReadLine();
 }
}
```

What we do here is to set up the keys and values of the list because this is a sorted list; the results would be as follows:

- Summer
- Winter
- Spring
- Summer

Stack

A Stack is a Last-in-First-out list, similar to a stack of books. Here is a small example of how to store information inside it:

```
using System;
using System.Collections;
public class Stack_Ex {
  public static void Main() {
      Stack sStack = new Stack(); //Create and initialize a new
      Stack.
      sStack.Push("Summer");
      sStack.Push("Autumn");
      sStack.Push("Winter");
      sStack.Push("Spring");
      DisplayValues(sStack);
  }
  public static void DisplayValues(IEnumerable tmpCollection) {
      foreach (Object obj in tmpCollection)
     Console.Write( " {0}", obj );
      Console.ReadLine();
  }
}
```

The **`Console`** will display the results after we have stored and loop through the items in the Stack. The **`Console`** would display the items in the following order:

- Spring
- Winter
- Autumn
- Summer

For all other Collection types, the *References* section at the end of this lesson provides more insight.

When to choose which Collection type

Using collections for the right purpose is crucial to performance. In this topic, we are quickly going to explore which Collection type is correct for

which purpose:

Consideration	Type of Collection to use
List where an element is discarded after the retrieval of a value	Any `Queue` or `Stack` classes
Accessing of elements in a certain order	Any `Queue`, `Stack`, or `LinkedList` classes
Accessing of elements by index	Any of the following classes: `ArrayList`, `StringCollection`, `Hashtable`, `SortedList`, `ListDictionary`, `StringDictionary`, `NameObjectCollectionbase`, and `NameValueCollection` classes
Contain only one value	Any collection based on the `Ilist` interface
One key and one value	Any collection based on the `IDictionary` interface
One key and multiple values	`NameValueCollection` class
Sorting elements by Hash codes	`Hashtable` class
Sorting by key	`SortedList` and `SortedDictionary` class
List that provides a Sort method	`ArrayList` class
Fast searches for small collections	`ListDictionary` class
Fast searches and fast retrieval	Any `Dictionary` class
String only collection	`StringCollection` class

Table 6.2: *Proper collection usage*

Another important topic on utilizing collections and memory properly.

Boxing issues

Boxing (implicit) is simply the process of converting a value type to the type object or to any interface type implemented by this value type. Unboxing (explicit) extracts the value type from the object. An example of both is as follows:

Boxing

```
int iAge = 43;
```

```
object oAge = iAge; //iAge is boxed
```

Unboxing

```
oAge = 43;
iAge = (int)oAge; //oAge is unboxed
```

Here is an example of both

```
class Boxing_Ex
{
    static void Main()
    {
    int iAge = 43;
    //Copy value of iAge into object oAge.
    object oAge = iAge;
    //Change value of iAge.
    iAge = 78;
    System.Console.WriteLine("Value of iAge = {0}", iAge);
    System.Console.WriteLine("Value of oAge = {0}", oAge);
    }
}
```

The output will be:

```
Value of iAge = 78
Value of oAge = 43
```

Keep in mind that boxing works with value types.

Thread safety in Collections

If we refer back to *Table 6.1*, we will see a few concurrent collections, namely, **`ConcurrentQueue`** and **`ConcurrentStack`**. **`ConcurrentQueue`** and **`ConcurrentStack`**, among others form part of the **`System.Collections.Concurrent`** namespace.

Thread safety applies to multi-threaded code. Multi-threaded means programs may execute code in several threads simultaneously in a shared address space. Code running across multiple threads, and accessing the shared address space, may cause deadlocks and erratic behavior if not managed properly.

The next table shows the available thread-safe collection classes:

Thread-safe Collection class	Description
`BlockingCollection`	Provides blocking and bounding capabilities
`ConcurrentBag`	Unordered collection of objects
`ConcurrentDictionary`	Key and value pairs accessed by multiple threads
`ConcurrentQueue`	First in-first out Collection
`ConcurrentStack`	Last in-first our Collection
`OrderablePartitioner`	Splits orderable data into multiple partitions
`Partitioner`	Provides common partitioning strategies

Table 6.3: *Thread-safe Collection classes*

Some examples are as follow.

BlockingCollection

A **`BlockingCollection`** is a thread-safe collection. With it we can have multiple threads add and remove data concurrently. Create a new **`BlockingCollection`** instance:

```
BlockingCollection<int> bColl = new BlockingCollection<int>
(new ConcurrentBag<int>(), 10); //Set maximum items to 10 for
boundedCapacity parameter
```

Adding new items can be done in two ways:

1. Add
2. TryAdd

An example of **`Add`**:

```
BlockingCollection<int> bColl = new BlockingCollection<int>
(2);
bColl.Add(1);
bColl.Add(2); //Add two items to collection
```

An example of **`TryAdd`**:

```
BlockingCollection<int> bColl = new BlockingCollection<int>
(2);
```

```
bColl.Add(1);
bColl.Add(2);
if (bColl.TryAdd(3, TimeSpan.FromSeconds(1)))
{
    Console.WriteLine("Added");
}
else
{
    Console.WriteLine("Not Added");
}
```

The **`TryAdd`** method takes an additional **`TimeSpan`** object as a parameter. Here, we try to add a third item. Then waits for 1 second if it is unable to add the third item and outputs **`Not Added`**. Taking items from the Collection also involves two methods, which are basically the same as the previous.

ConcurrentBag

Represents a thread-safe, unordered Collection of objects and supports duplicate entries.

Initializing a **`ConcurrentBag`**:

```
ConcurrentBag<int> conBag = new ConcurrentBag<int>();
```

Adding items example:

```
conBag.Add(1);
conBag.Add(9);
conBag.Add(7);
conBag.Add(8);
```

Taking items form the **`ConcurrentBag`**:

```
int iItem;
bool output = conBag.TryTake(out iItem);
```

Finally, let us have a look at the guidelines for proper Collection usage.

Guidelines

The next table lists some dos and don'ts and considerations when using collections:

Don't	Avoid	Consider	Do

Use `List` in public APIs	Using `ICollection<T>` as a parameter to access properties	Use subclasses of generic base collections	Use the least-specialized type as a parameter
Use `ArrayList` in public APIs	Using `ICollection` as a parameter to access properties	Use keyed collections if the items have unique keys	Use `Collection<T>` for properties that represent read and write collections
Use `Hashtable` in public APIs	Using suffixes that imply the particular implementation	Use arrays in low-level APIs	Use `Collection` for properties that represent read and write collections
Use `Dictionary` or `ArrayList` in public APIs	Implement collection interfaces on types with complex APIs	Use `ReadOnly` in the name of Read-only collections	Use `ReadOnlyCollection<T>` for properties that represents read-only collections
Use types that implement `IEnumerator`			Use `ReadOnlyCollection` for properties that represent read-only collections
Implement both `IEnumerable` and `IEnumerator`			Use collections over arrays
Provide settable properties for Collections			Use byte arrays instead of byte collections
Return NULL values from properties and methods			Use `Collection` suffix in `Collection` names
			Use `Dictionary` suffix in Collection names

Table 6.4: *Guidelines for creating Collections*

Conclusion

In this chapter, we gave an overview of Collections in .NET. We learned why it is important to use the right Collection type for the right job. We learned why it is important to ensure the PC's memory is used optimally.

Finally, we had a look at boxing/unboxing and Collections capable of handling multithreading.

In the upcoming chapter, we will learn how to identify performance problems.

Points to remember

- For accessing elements in a certain order, use any `Queue`, `Stack`, or `LinkedList` classes.
- For fast searches for small Collections, use the `ListDictionary` class.
- Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type.
- Unboxing extracts the value type from the object.

Questions

1. What should we use instead of in-public APIs?
2. Explain the term multi-threaded.
3. What is the difference between boxing and unboxing?

Answers

1. Use the least-specialized type as a parameter.
2. Multi-threaded means programs may execute code in several threads simultaneously in a shared address space.
3. Boxing (implicit) is simply the process of converting a value type to the type object or to any interface type implemented by this value type. Unboxing (explicit) extracts the value type from the object.

Key terms

- Thread-safety
- Boxing
- Unboxing

- Immutable
- Strongly Typed
- First in-Last out
- Last in-First out

References

- **Collection types: https://docs.microsoft.com/en-us/dotnet/standard/collections/commonly-used-collection-types**

CHAPTER 7

Identifying Performance Problems

Introduction

If we know how to check the performance issues of an application, it will save a lot of headaches and heartache. Developers do not spend enough time on performance testing of their apps, and then they wonder why their applications are misbehaving.

Visual Studio provides a lot of functionality and tools out of the box to aid in determining where the problems are in an application other than code.

Structure

The following topics are to be covered:

- Using a profiler
- Using diagnostic tools

Objective

This chapter focuses on the performance and diagnostic tools available in Visual Studio. We will learn how to use them and what each of their various settings does.

Using a profiler

Profiling or software profiling is simply a form of dynamic program analysis. A few things a profiler measures are as follows:

- Program complexity concerning memory and time
- Instruction usage
- Function calls frequency and duration

The preceding list (and more) helps to identify program performance problems. They also aid in performance engineering and optimization.

Profilers gather the following program events:

- Hardware interrupts
- Code instrumentation
- Instruction set simulation
- Operating system hooks

Let us look at each.

Hardware interrupts

Hardware interrupts are input signals with the highest priority for hardware events that require immediate processing. Now, why is this important? Responding to interrupts prevents the computer's CPU from running your code while it processes the hardware request. This consumption of clock cycles causes poor performance.

Hardware interrupts are grouped into two types:

- **Maskable interrupts (MI)**: With maskable interrupts, processors have an interrupt mask register (a settable/resettable register within the CPU). Each interrupt has a bit in the mask register. When set, the interrupt is enabled. When not set, the interrupt is disabled.
- **Non-maskable interrupts (NMI)**: The NMIs are the highest priority activities that need to be processed immediately regardless of the situation.

Code instrumentation

Code instrumentation is the task of including code in programs to monitor the performance of .NET applications. Code instrumentation can display messages or write to event logs in case of a failure during the execution of an application at run time. Some tasks that Code Instrumentation performs are as follows:

- Debugging
- Tracing
- Writing to performance counters and event logs

Debugging

Debugging enables developers to troubleshoot, for example, logical errors in code.

Tracing

With tracing, we can gather information about code execution at runtime, which can be used to troubleshoot applications after deployment.

Writing to performance counters and event logs

Counters and logs collect and analyze performance-related data to monitor the performance of applications.

Instruction set simulation

Instruction set simulators (**ISS**) are simulation models that are usually coded in a high-level programming language. These simulation models copy the behavior of mainframes by reading instructions as well as maintaining variables that represent the processor's registers.

Instruction set simulation can be used for several reasons; they include the following:

- To simulate the machine code of another hardware device or entire computer for upward compatibility.
- To monitor and execute machine code instructions as an input stream on the same hardware for test and debugging purposes.
- To improve the speed performance compared to a slower cycle-accurate simulator—of simulations

Operating system hooks

Operating system hooks are used to alter the behavior of an operating system, application, or any other software components by intercepting function calls, messages, or events passed between software components.

Hooking can be used for the following *good* purposes:

- **Include debugging functionality**: The following custom debug hook functions, which can be inserted into code at predefined points, include Client Block hook functions, Allocation hook functions, CRT memory allocations, and Report hook functions. These functions are far beyond

the scope of this book, so please have a look at the *References* section at the end of this chapter for more information.

- **Intercept keyboard or mouse event messages before they reach an application**: These hooks are usually known as low-level hooks. As the name implies, these types of hooks allow developers to override these events with their own functionality.
- **Intercept operating system calls**: Similar to the previously mentioned hook, developers can intercept calls directly from and to the Operating System, for example, by adding items to the system context menu upon right-clicking or automating mundane system tasks.
- **Used in benchmarking programs**, which we will cover in *Chapter 8: Benchmarking Code with BenchmarkDotNet*.

Now, the *bad* purposes that are used by malicious code include the following:

Rootkits

Rootkits are sets of tools that maintain root privileged access to an operating system; depending on the type of rootkit, they may carry malicious code that is deployed secretly into the target system. Rootkits exploit background system processes at various privilege levels, such as user mode or kernel mode in Windows operating systems.

Rootkits do not compromise the system security on their own, but they do provide backdoor access for injecting harmful code or software. Examples of these are as follows:

- Sending Trojans or infected files as email attachments. For more information regarding Trojans, refer to the *References* section at the end of this chapter
- Creating malware applications posing as harmless pop-ups or banners on websites
- Using phishing attacks and keyloggers can give hackers root access to inject rootkits. We will speak about keylogging shortly.

To understand how rootkits work, we must understand how Windows executes code. There are following two ways for Windows to execute code:

- **User mode**: In user mode, lower-level functions and lower-level privileges are handled because not all programs and processes require the

full range of processing power and hardware.

This is where **Application Programming Interface** (**API**) hooking comes in. API hooking modifies or changes the flow of Windows API calls and injects harmful code into the program's memory. This is usually done inside Windows **Dynamic Link Libraries** (**DLLs**) because they are called by most programs on the Operating system.

- **Kernel-mode**: In kernel mode, the operating system runs high-level code directly to handle the managing tasks of the operating system.

  Kernel-mode rootkits exist on the same level as most malware detection software and can alter parts of the kernel. Kernel-mode rootkits use the following hooking techniques, which are beyond the scope of this book:

  - Inline function hooking
  - IDT hooking
  - SSDT hooking

Keylogging

Keylogging hardware or software records the keystrokes of the user, sometimes for illegal purposes and other times not.

Keystroke logging

Keystroke logging tracks and records every keystroke the user performs to communicate with the computer. The keyboard commands, which get recorded, include the following:

- Length of the keypress
- Time of keypress
- Velocity of keypress
- Name of the key used

User behaviors and private data can easily be assembled from logged keystrokes. These include online banking access, identity numbers, social media, emails, websites visited, and text messages.

Keylogger tools

Keyloggers automate keystroke logging, or even cut-copy-paste operations, calls, GPS information, and other input devices such as cameras and microphones to record the data into text files. There are following two forms of keyloggers:

- **Software keyloggers**: Software keyloggers are installed on a device. Some forms of keylogger include:
    - **API-based**: These intercept the signals sent from keyboards and log them.
    - **Form-grabbing**: Intercepts text entered into website forms. The data is first recorded locally before being transmitted.
    - **Kernel-based**: Kernel-based keyloggers sneak their way into the operating system's core so that it can obtain administrator permission, which allows them to get unrestricted access to all input signals sent to the system.
- **Hardware keyloggers**:
    - Hardware keyloggers are physical components built in or connected to a device. Some of these components include the following:
        - Keyboard keyloggers that can be built into the keyboard or placed in line with the keyboard's connection cable.
        - Hidden cameras that can be placed in public spaces that can visually track keystrokes
        - USB disk-loaded keyloggers that, once connected to a device, deliver a Trojan horse that delivers a keystroke logger.

Using diagnostic tools

Diagnostic tools help identify problems in your applications. These can include Memory usage, CPU usage, GPU performance, and database performance:

Analyze CPU usage

To analyze CPU usage, follow these steps:

1. Open the **`Debug`** main menu.
2. Select **`Windows`**.
3. Select **`Show Diagnostic Tools`**. This is shown next:

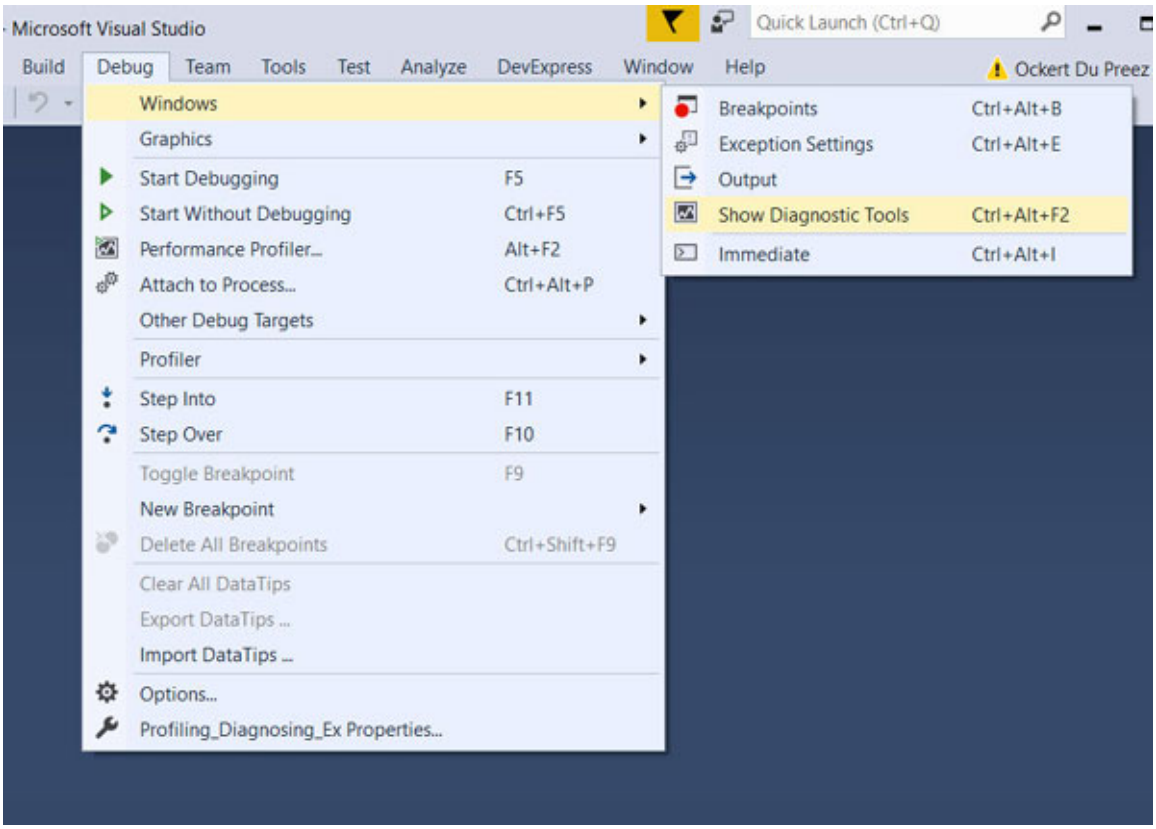


Figure 7.1: *Diagnostic tools*

4. Click on select **`Tools`** and ensure **`CPU usage`** is selected. This is shown in the following screenshot:

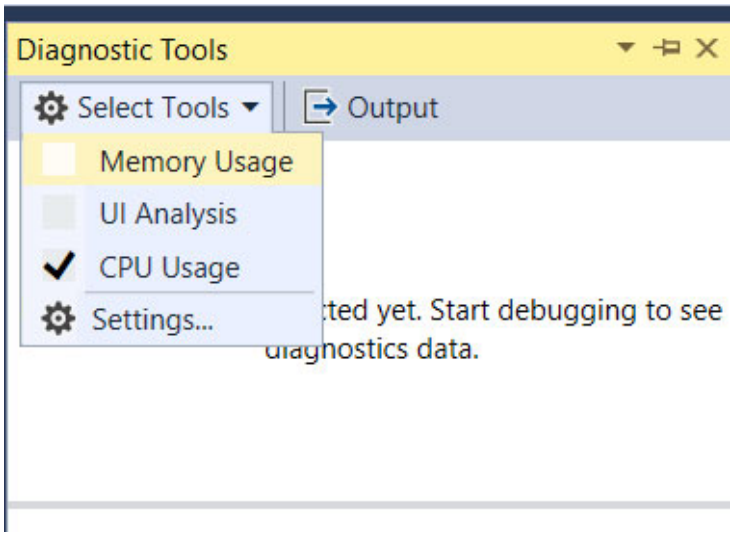


Figure 7.2: *CPU usage*

5. If we were to click on the **`Settings`** item, it would open a dialog box where we can adjust additional settings such as **`CPU usage`**. This is shown

in the following screenshot:

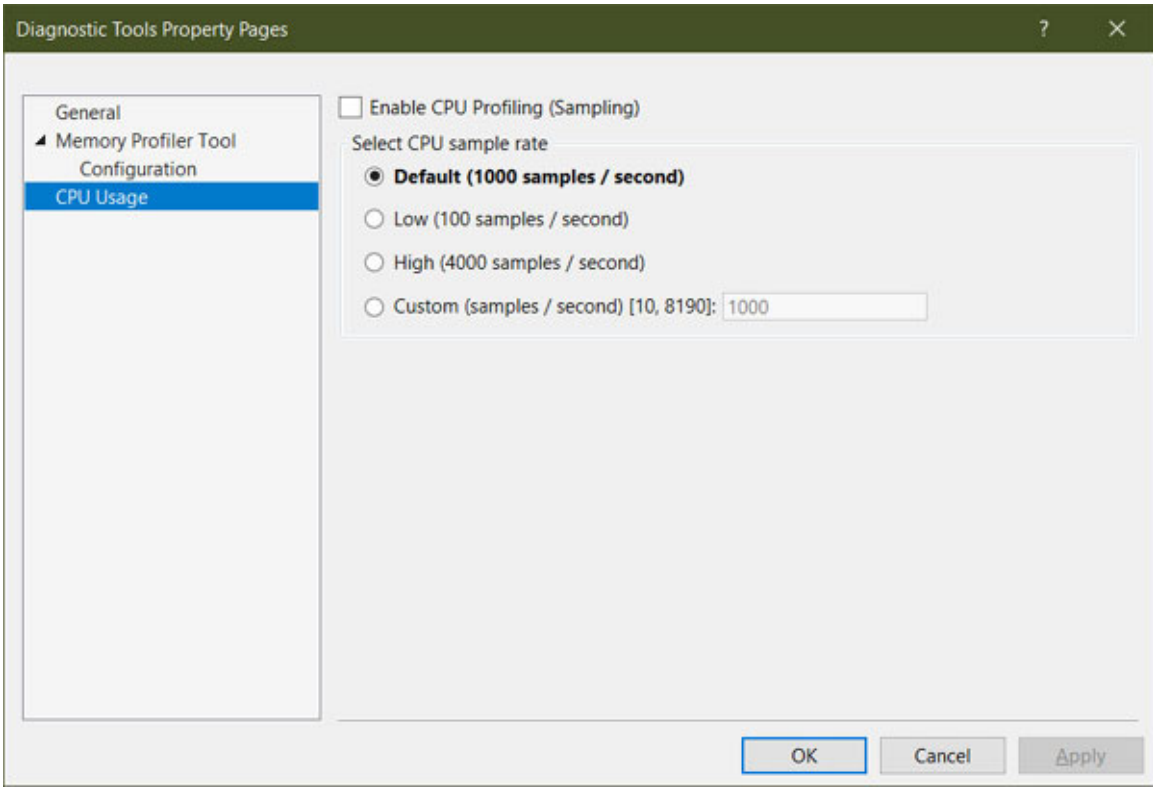


Figure 7.3: *CPU usage settings*

We can set the frequency of sampling per second and enable CPU profiling. When the application is run, the `Diagnostic` window will show the details of the tests; here is an example screenshot:

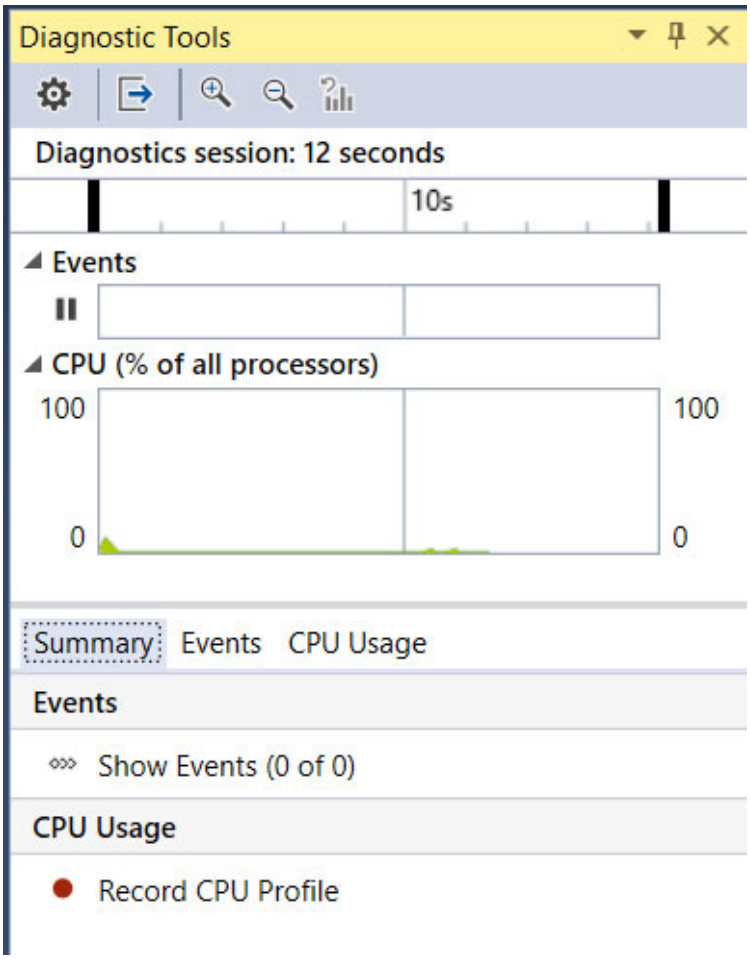


Figure 7.4: *Diagnostic window upon running the application*

To monitor CPU usage properly, we need to set two break points in our program. One at the beginning and one at the end of the function that needs to be monitored. Select **`Record CPU Profile`**. When the application reaches the second break point, the profiler will collect the data, and then a screen similar to *figure 7.5* will appear:

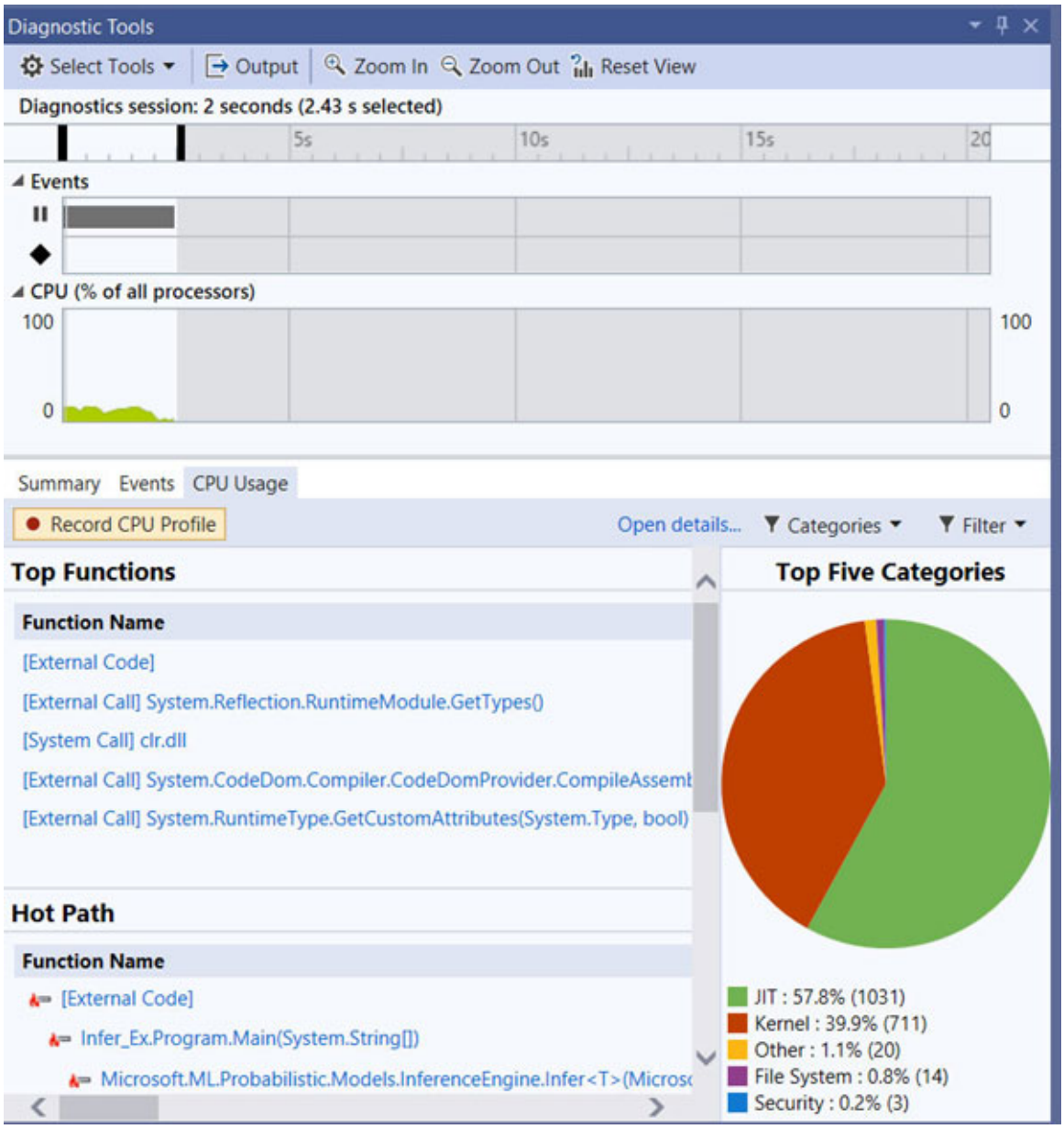


Figure 7.5: *CPU usage Windows*

Let us break the sections down quickly

- **Top functions** show which functions (internal and external).
- **Hot Path** displays the execution path the current function followed.

The *pie chart* displays which categories take up the most and least CPU processing power. Let us take it a step further! Select the categories drop-down just above the Pie chart. It should look like *figure 7.6*:

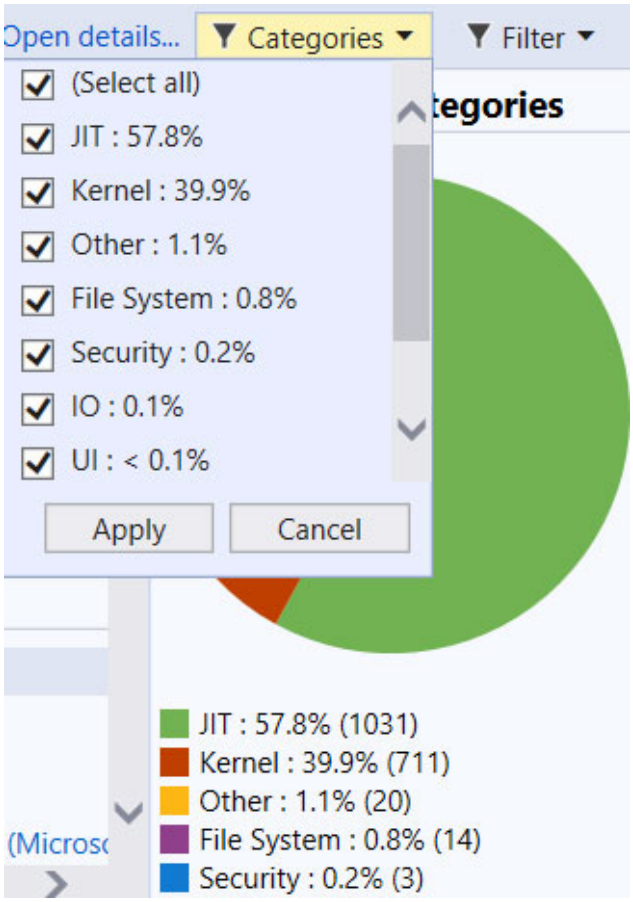


Figure 7.6: *Categories*

Here, we can select the different categories we had like to monitor. The button next to **`Categories`** is **`Filter`** (shown next), which allows us to filter the threads we had like to monitor. Now how do we see these threads?

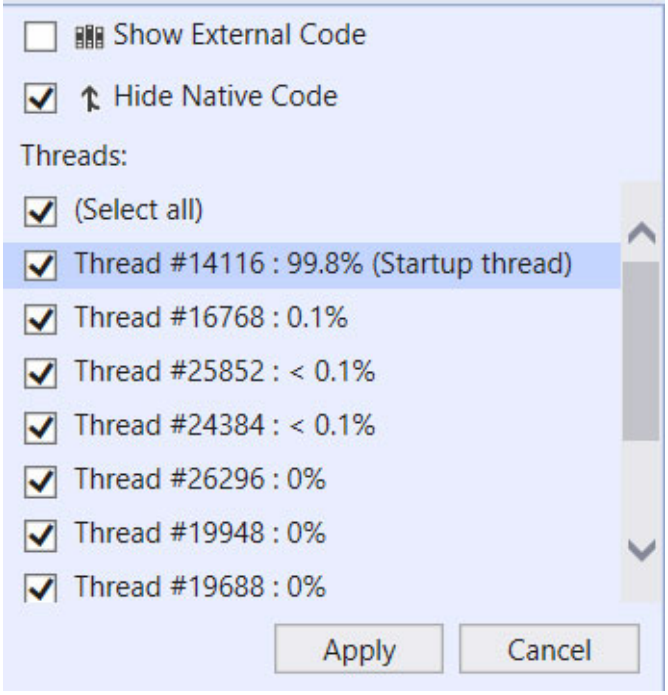


Figure 7.7: *Filter threads*

On the left side of **`Categories`** is a link saying: **`Open details`**…. Click this link to open details of each function and thread. This window is shown in *figure 7.8*:

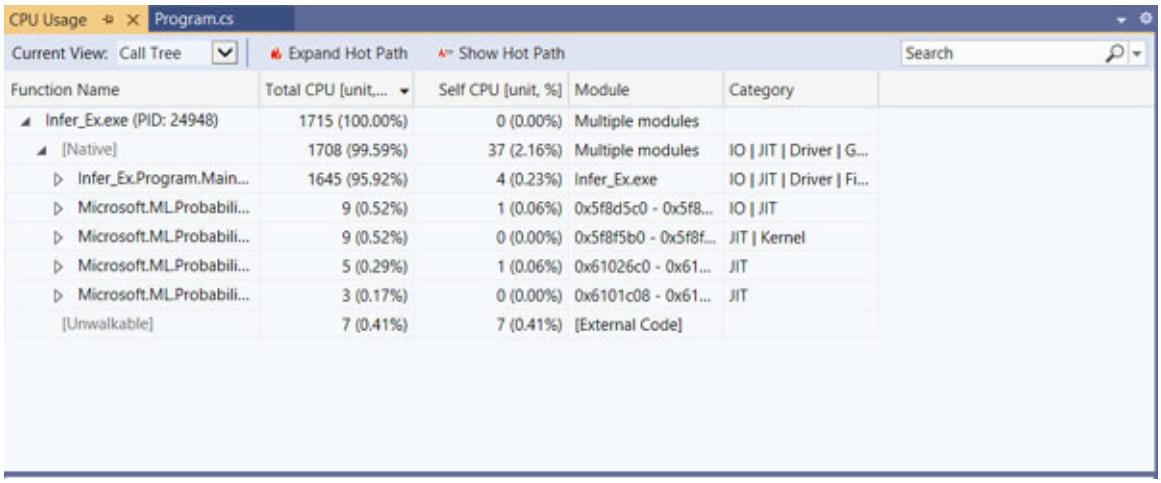


Figure 7.8: *Details*

Note, this window may look different, as this depends on the **`Current View`** selected from the **`Current View`** drop-down, which includes call tree, caller/callee, modules, and functions. Double-clicking on a program function, such as the program's **`Main`** function, will open the following function and show the CPU stats:

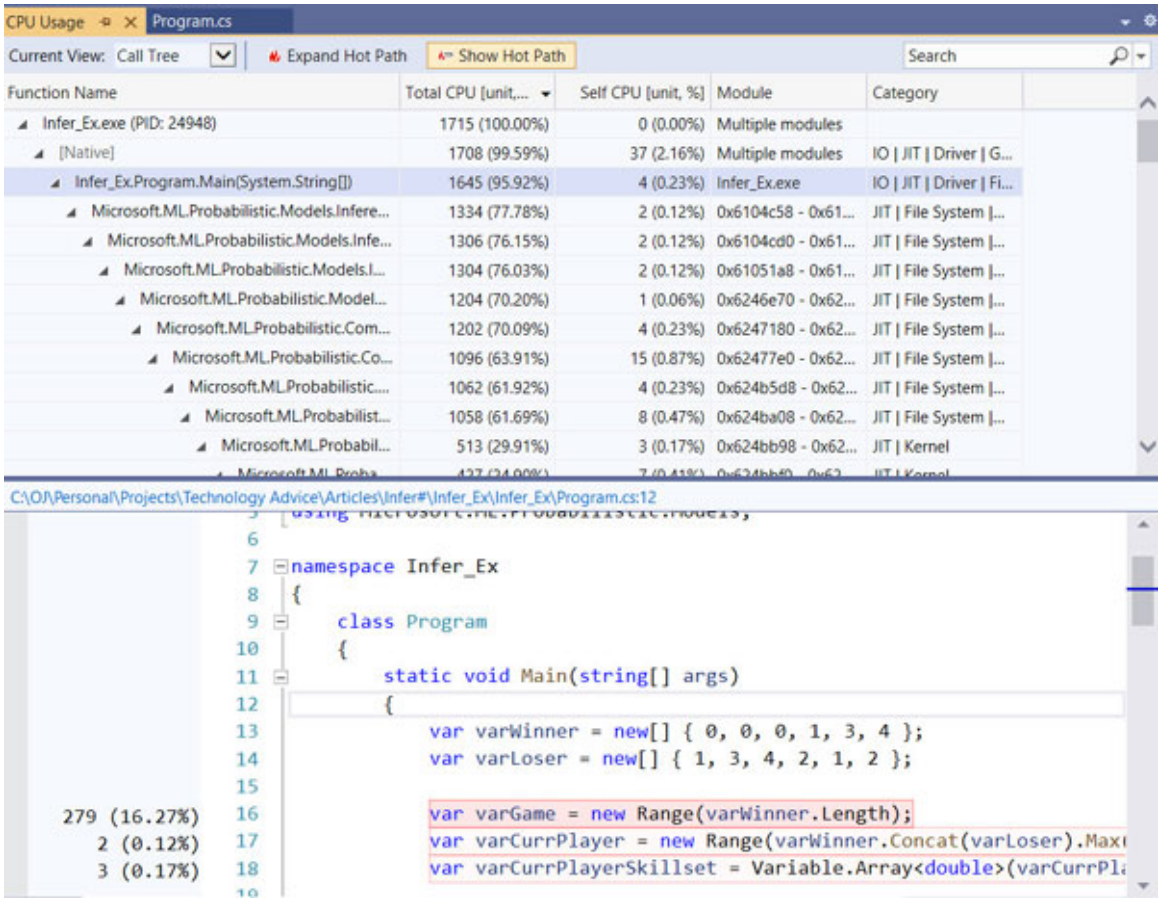


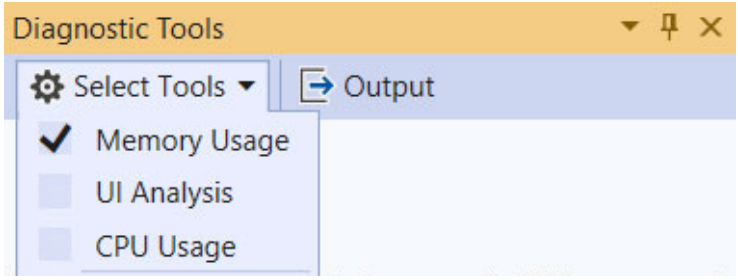
Figure 7.9: *Main method statistics*

The same options are available inside the **`Performance Profiler`**, found at **`Debug`** | **`Performance Profiler`**.

Analyze memory usage

To analyze the memory usage of an application, we can use either the **`Performance Profiler`** or the **`Diagnostic Tools`** window. In this little exercise, we will use the **`Diagnostic Tools`** window:

1. Click on the **`Select Tools`** drop-down and ensure that only the **`Memory Usage`** option is selected.



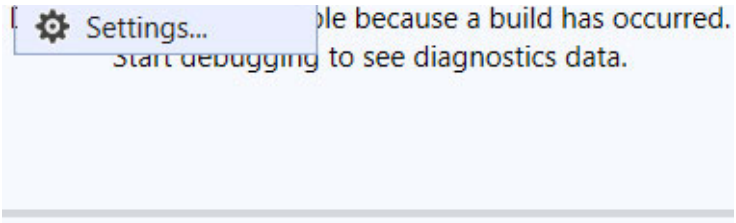


Figure 7.10: *Memory usage drop-down option*

2. Run the application. The **`Diagnostics Tools`** window should display a bar displaying the memory allocation graphically.
3. At the bottom of this bar, in the **`Memory Usage`** tab, click **`Take Snapshot`**. Wait a second or two and click it again. The next images show the **`Process Memory`** bar in action as well as display two snapshots:

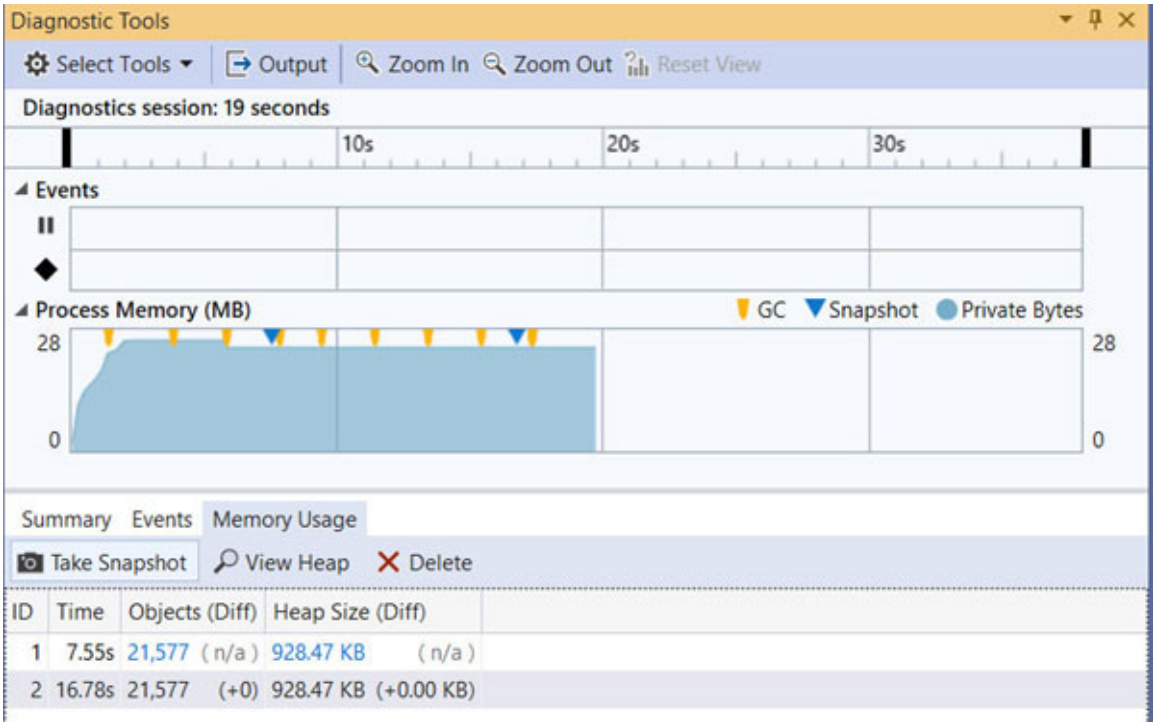


Figure 7.11: *Memory usage*

The columns display the time the snapshots were taken, the differences between the memory allocation at that time, and the Space the objects

occupy on the heap.

4. Click on the **`View Heap`** button. A new profiling window will open, displaying the Memory Management of all the objects, as shown in *figure 7.12*:

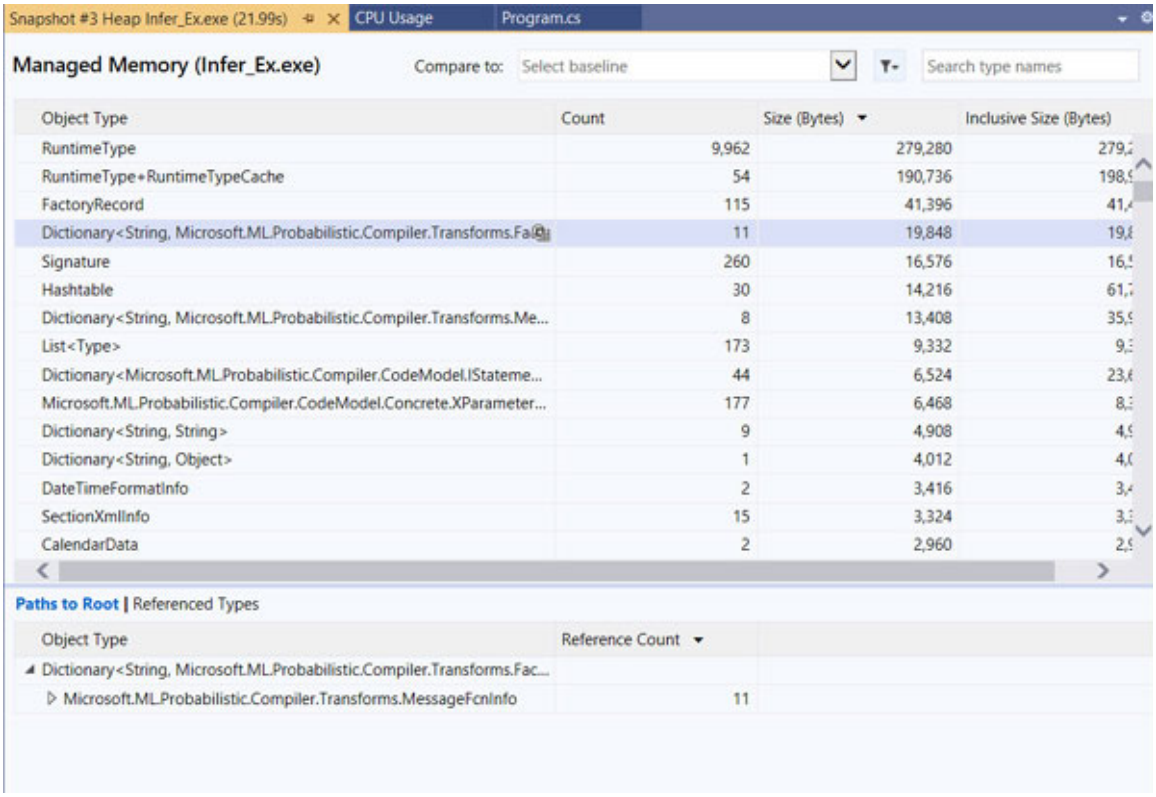


Figure 7.12: *Managed memory*

Analyze asynchronous code

We can investigate async and await usage in our .NET applications with the .NET Async tool. This tool is found in the performance profiler. Let us follow these steps to have a look at async calls.

1. Click **`Debug`**.
2. Click **`Performance profiler`**.

> **A quick note: When the configuration is set to Release instead of Debug, the results may be more reliable.**

3. Select **`.NET Async`**. This is shown in the following screenshot:

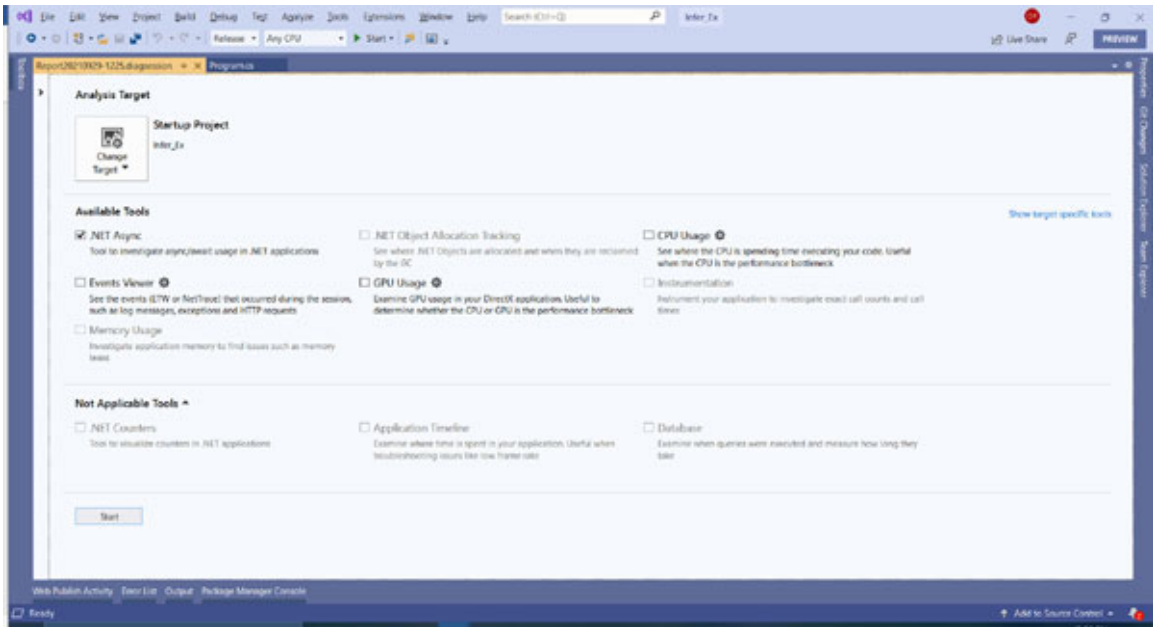


Figure 7.13: *.NET async*

4. Click the `Start` button at the bottom of this window

It takes a while, depending on the size of the program. There are plenty more tools for diagnosing and profiling apps, but we will look into them throughout the course of this book. For a list of profiling and diagnosing tools, please have a look at the *References* section at the end of this chapter.

To find out which performance tool should be used, please refer to the *References* section at the end of this chapter.

Conclusion

In this chapter, we have looked at how performance tuning can help programs to perform better. This is critical. We also had a look at a few very important aspects of a profiler, which included safety, security, and vulnerabilities programs that can be exposed to.

In the upcoming chapter, we will start using external tools to help programs perform better.

Points to remember

- Keylogging hardware or software records the keystrokes of the user.

- Hardware interrupts are input signals with the highest priority for hardware events that require immediate processing.
- Code instrumentation is the task of including code in programs to monitor the performance of .NET applications.
- We can investigate async and await usage in our .NET applications with the .NET Async tool.

Questions

1. Explain what the Hot Path CPU Usage window does
2. What happens when we click “take Snapshot” in the `Memory Usage` tab?
3. How do we investigate async and await usage in our .NET applications?
4. Explain the term Rootkit.

Answers

1. Hot Path displays the execution path the current function followed.
2. The columns display the time the snapshots were taken, the differences between the memory allocation at that time, and the Space the objects occupy on the heap
3. With the .NET Async tool.
4. Rootkits exploit background system processes at various privilege levels, such as user mode or kernel mode in Windows operating systems, for example.

Key terms

- Tracing
- Interrupts
- Instruction set simulation
- Operating system hooks
- Diagnostic tools
- CPU usage
- Memory usage

- Performance profiler

References

- **Debug hook functions: https://docs.microsoft.com/en-us/visualstudio/debugger/debug-hook-function-writing?view=vs-2019**
- **Trojans: https://in.norton.com/internetsecurity-malware-what-is-a-trojan.html**
- **Profiling tools: https://docs.microsoft.com/en-us/visualstudio/profiling/?view=vs-2019**
- **Which tools to use and where: https://docs.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2019#which-tool-should-i-use**

CHAPTER 8

Benchmarking Code with BenchmarkDotNet

Introduction

Benchmarking code is vital, even more than profiling. When looking at writing high-performance code, profiling and benchmarking should go hand in hand. In benchmarking code, we can pinpoint exactly where problems are and even when they will occur.

Structure

The following topics are to be covered:

- Benchmarking introduction
- Benchmarking code with BenchmarkDotNet

Objective

This chapter explains the why is and how is of Benchmarking code.

Benchmarking introduction

We have talked about performance profiling in previous chapters; many software developers confuse benchmarking with profiling, but they are not the same. They work together, in a sense, to indicate performance problems.

Benchmarking enables software developers to see how slow their code is, for example, a process that took 30 seconds instead of being much faster. Profiling tells the software developer why that process took so long and what causes it to be slow.

Benchmarking allows us to also test stability, responsiveness, and effectiveness. Benchmarking can also be used to investigate, measure, validate, and verify other parts of code, such as scalability or usage. When proper

benchmarking is in place, we can then use it to measure new developments, updates, or changes in the future.

As mentioned earlier, a benchmark must be repeatable. This means that if the response times vary too much with each iteration of a test, the system performance should be benchmarked. Response time needs to be stable amongst different load conditions.

Some guidelines to follow when benchmarking include the following:

- Running a benchmark test at least 10 times instead of just once.
- Ensure response times are stable in all load conditions.
- Never trust small performance improvements, like 5%. This will cause the benchmarking test to be performed more than is necessary.

Let us have a look at some benefits of benchmarking.

Benefits of benchmark testing

The benefits include the following:

- How well an application or a website is performing with respect to competitors' applications or websites?
- How do different customers experience the response time and availability of a site or app?
- It ensures that applications and websites comply with standards and best practices.
- Allows to figure out the mistakes to be avoided.

Phases of benchmark testing

There are ultimately four different phases involved in benchmark testing. They are as follows:

- Planning phase
- Analysis phase
- Integration phase
- Action phase

Let us have a look at each, respectively.

Planning phase

In the planning phase, we should identify and prioritize the various standards and requirements, decide on benchmark criteria and define the benchmark test process.

Analysis phase

In this phase, we should identify the root causes of errors in order to improve quality; then, we should set goals for test processes.

Integration phase

Here, we should share the outcomes with the concerned parties to get approval and establish functional goals.

Action phase

As the name implies, in the action phase, we have to develop a test plan and documentation. We should also implement the actions specified in the previous phases and monitor their progress. Finally, we have to run the processes continuously.

Creating a benchmark test plan

A benchmark test plan follows the next few steps given as follows:

- Scale the workload
- Invoking the workload
- Collect measures for benchmark testing
- Store measures for benchmark testing
- Define the time span required for a test process
- Prepare a backup plan in the event of test case failure

Components of benchmark testing

There are essentially three main components of benchmark testing, and these are as follows:

- **Workload specifications**: This determines the type and frequency of requests to be submitted to the system being tested.
- **Specifications of metrics**: This determines what element must be measured, for example, download speed.
- **Specification of measurement**: This determines how to measure the specified elements to find appropriate values.

Benchmarking architecture components

The following is a list of the components of any benchmarking architecture:

- **Benchmarks**: An application or website that has various scenarios to benchmark.
- **Server**: An application or website which queues jobs that can run custom websites or applications to be benchmarked.
- **Client**: An application or website that queues jobs to create custom client loads on an application or website.
- **Driver**: A command-line application that can enqueue server and client jobs and display the results locally.

Successful benchmark testing

In order to run a successful benchmark test, consider the following facts:

- Ensure that all software components are in working condition
- Ensure that the operating system and its supporting drivers work accurately
- Close all unneeded processes and applications running in the background before testing

There are a few different types of benchmarks that you may want to establish; they include the following:

- **Load Benchmarking**: These tests are used to establish how a system will behave under a specific load. This would be the amount of traffic that the programs or sites anticipate seeing on a certain day. Load benchmarking will help to establish the benchmark for application performance.

- **Stress benchmarking**: These tests allow developers to understand the limits of a system and help them to determine whether it is built to work with the current load and the expected load in the future.
- **Endurance benchmarking**: Endurance benchmarking looks for any leaks that occur. This is usually due to memory usage or performance degradation. Endurance benchmarking helps to model a significant load on a system for an extended period of time and test performance into the future.
- **Spike benchmarking**: Spike benchmarking tests how a system will perform if there is an unexpected change in loads.
- **Breakpoint benchmarking**: These tests look at how far a system can be pushed until it breaks.

Benchmarking code with BenchmarkDotNet

First, what is BenchmarkDotNet?

BenchmarkDotNet explained

BenchmarkDotNet is a powerful, lightweight, and user-friendly open-source .NET library that transforms methods in a program or website into benchmarks. It then tracks those methods and provides insights into the performance data captured.

BenchmarkDotNet works in both .NET Framework and .NET Core applications and is used by Mono, ASP.NET Core, ML.NET, Entity Framework Core, .NET Core runtime, and libraries, Roslyn, .NET Docs, and TensorFlow.NET, to name a few projects.

Benefits of using BenchmarkDotNet

Some benefits of using BenchmarkDotNet are as follows:

- Simplicity
- Automation
- Reliability

Here is a quick overview of each of the points:

- **Simplicity**: Performance experiments can be designed using the simple APIs of BenchmarkDotNet. For instance, to compare benchmarks with each other, mark one of the benchmarks as the baseline, causing it to be compared to all the other benchmarks.
- **Automation**: In order to have reliable benchmarks, they require a lot of benchmark code, for example, repeating sections of code. Mistakes happen, especially with repetitive code may end up messing up the measurements. This is where BenchmarkDotNet comes in. Besides this, BenchmarkDotNet can also perform advanced tasks such as measuring the managed memory traffic and printing disassembly listings of benchmarks.
- **Reliability**: The BenchmarkDotNet library protects us from the most common benchmarking pitfalls, such as deciding the number of method invocations and the number of actual iterations. This is due to its statistical metrics, including numerous heuristics, checks, hacks, and tricks, making the results more reliable.

Benchmark Config

BenchmarkDotNet Config comprises the following:

- **Jobs**: Set of characteristics that describe the way to run the benchmarks.
- **Columns**: Columns in the summary table
- **Exporters**: Enables exporting results of the benchmark in various formats such as CSV, HTML, and markdown.
- **Loggers**: Enable logging of the results of benchmarks.
- **Diagnosers**: Attaches to the benchmarkers to retrieve useful information.
- **ToolChains**: Contains the app generator, builder, and executor; these can include Roslyn and DotNet CLI.
- **Analyzers:** Analyzes summaries of each benchmark and produces appropriate warnings wherever necessary.
- **Validators**: Validates each benchmark before its execution and produces validation errors.
- **Filters**: Gives the choice of choosing only some and not all of the benchmarks specified.

- **Orderers**: Enables customization of the order of benchmark results in summary.

Using BenchmarkDotNet

To use BenchmarkDotNet in Visual Studio, make use of the following steps:

1. Open Visual Studio.
2. Create a new Console App (.NET Core) in C#.
3. Name the application **`Benchmarking_Ex`**, for example.
4. Install the BenchmarkDotNet NuGet package by selecting **`Project`** | **`Manage NuGet`** packages.
5. Search for **`BenchmarkDotNet`** and install the package, as shown in *figure 8.1*:

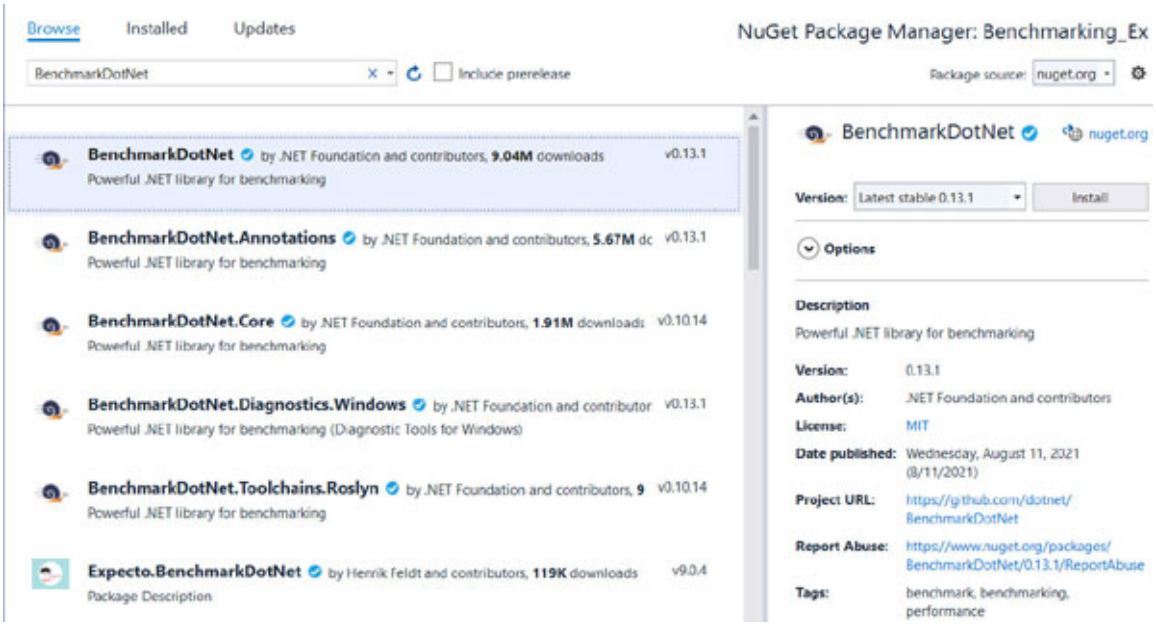


Figure 8.1: *BenchmarkDotNet NuGet package*

6. Add a new class and name it **`TestClass`**.
7. Edit the **`TestClass`** to resemble the following code listing:

```
using System;
using System.Collections.Generic;
using System.Linq;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
namespace Benchmarking_Ex
```

```
{
 public class TestClass
 {
  private readonly List<string> lstMain = new List<string>
  ();
  private readonly int intListSize = 1000000;
  private readonly string strItem = "Test";
  public TestClass()
  {
   //Add a large number of items to list
   Enumerable.Range(1, intListSize).ToList().ForEach(x =>
   lstMain.Add(x.ToString()));
   //Insert 'Test' right in the middle.
   lstMain.Insert(intListSize / 2, strItem);
  }
  [Benchmark]
  public string SingleItem() => lstMain.SingleOrDefault(x =>
  x == strItem);
  [Benchmark]
  public string FirstItem() => lstMain.FirstOrDefault(x => x
  == strItem);
 }
}
```

With the **`TestClass`**, we build up a large list containing random items, and then we insert the phrase **`Test`** in the middle of the list. The items marked with the **`[Benchmark]`** attribute is going to be benchmarked and compared.

8. Edit the **`Main`** method inside the **`Program.cs`** file to look like the following:

```
using BenchmarkDotNet.Running;
using System;
namespace Benchmarking_Ex
{
 public class Program
 {
  static void Main(string[] args)
  {
   var summary = BenchmarkRunner.Run<TestClass>();
```

```
        Console.ReadLine();
      }
    }
  }
```

This runs the benchmark with the help of the **`BencmarkRunner`** class included at the top of the **`Program.cs`** file.

9. Build this project in **`Release mode`** by selecting **`Release`** in the **`Solution Configurations`** dropdown, as shown in the following figure:

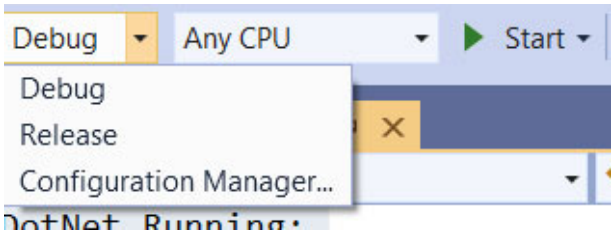


Figure 8.2: *Solution configurations dropdown*

10. Build the Solution.
11. Open the Command Prompt.
12. Navigate to the release folder inside the bin folder of the **`Benchmarking_Ex`** application.

    For more information on the Command prompt, please have a look at the *References* section at the end of this chapter.

13. Run the application by entering run DotNet **`BenchmarkDotNet.dll`**.

    The output results would look similar to the following:

```
Method |      Mean |    StdDev |    Median |
-------|----------:|----------:|----------:|
Single  |15.591 ms|0.4429 ms|15.507 ms|
  First   | 7.638 ms | 0.4399 ms|7.475 ms |
```

`SingleItem` seems to be twice as slow as **`FirstItem`**. When **`FirstItem`** finds an item, it immediately returns. When **`SingleItem`** finds an item, it still needs to traverse the entire list checking for duplicate items in order to throw an exception.

Conclusion

In this chapter, we taught us all about benchmarking and why it is necessary to help improve code performance. We looked at the benefits of benchmarking,

the justification of benchmarking, and the different types of benchmarking. We then learnt about a benchmarking tool named BenchmarkDotNet—how to set it up, set its configuration, and make use of it practically.

In the upcoming chapter, we explore the cache and the common problems people encounter by not writing proper code.

Points to remember

- In the planning phase of benchmarking, we should identify and prioritize the various standards and requirements, decide on benchmark criteria and define the benchmark test process.
- Specifications of Metrics determine what element must be.
- BenchmarkDotNet is a powerful, lightweight, and user-friendly open-source .NET library that transforms methods in a program or website into benchmarks.

Questions

1. What are the benefits of using BenchmarkDotNet?
2. What are the phases involved in benchmark testing?

Answers

1. They are as follows:
    a. Simplicity
    b. Automation
    c. Reliability
2. They are
    a. Planning Phase
    b. Analysis Phase
    c. Integration Phase
    d. Action Phase

Key terms

- Benchmarking
- Scaling
- Local benchmarking

References

- **Command prompt**: https://www.lifewire.com/list-of-command-prompt-commands-4092302

CHAPTER 9

Dealing with the Memory Cache

Introduction

Understanding how important it is to work with a cache is vital for any developer and application. However, it is also not a case of implementing a cache but rather knowing the risks involved in it.

Structure

The topics to be covered are as follows:

- Caching
- Caching mistakes
- Cache replacement policies
- In-memory cache
- Persistent in-process cache
- Distributed cache

Objective

By the end of this chapter, the readers will have a good understanding of the cache and the common problems people encounter by not writing a proper code.

Caching

Caching is a software development pattern that stores data in a cache container so that continuous requests for the data can be served to the client quicker. Implementation of caching should occur when performing memory or CPU operations are performed. For example, when a request is sent to a database to obtain a piece of information, the piece of information is stored inside a memory cache, so for the next time we need it, the application extracts it

directly from the cache container in memory (cache hit) instead of doing a round trip to the database again, which would cause performance issues and possible wrong results or no results (cache miss).

A cache hit happens when a program makes a request in the cache for the content, and the content has been saved inside the cache. A cache miss occurs when the requested content is not stored in the cache.

When a cache hit occurs, the content inside the cache will load much faster as the cache can immediately deliver the content to the end user. When a cache miss occurs, the information should first be sourced and then saved inside the cache.

Importance of caching

Caching allows developers to achieve performance improvements. Developers want to deploy the best-performing version of their applications, and users want the best-performing applications. Users should not waste their time on processes taking painstakingly long to complete.

- **Competition**: If an application wants to compete with the myriad of applications on the market, the application should implement caching properly.
- **Frustrated users**: Explaining to users why an application is slow can represent unforeseen problems.
- **Reduce overhead**: With caching the applications, we avoid making new requests or reprocessing data every time, which reduces CPU overhead and network overhead, prolonging the life of the computer systems running the application and reducing the costs of adding more infrastructure.

Benefits of caching

Some benefits of caching include the following:

- **Performance**: Storing data in a cache allows a computer to run faster.
- **Offline work**: Caches also let applications function without an internet connection.
- **Resource efficiency**: Caching helps physical devices conserve resources.

Disadvantages of caching

Some disadvantages of caching include:

- **Corruption**: When caches are corrupted, which can happen, applications such as browsers can crash or display data incorrectly.
- **Performance**: When caches get too large, they cause performance to degrade and consume memory that other applications might need
- **Outdated information**: Sometimes, an app cache displays old or outdated information, which causes the application to glitch or returns misleading information.

Caching mistakes

The following are some common problems occurring when implementing a cache:

- **Source of truth**: By implementing caching, the program no longer reads from the application's source of truth. This causes possible unexpected results in the data. Every read or write to a piece of cached data is potentially subject to not match with the source of the truth and must be taken into account when debugging.
- **Perspective-dependent values**: One of the most common caching mistakes is caching perspective-dependent values and serving them to the wrong users that have a different perspective. This leads to potentially serious privacy and security issues.
- **Different behavior expectations**: With caching, new race conditions are potentially possible; for example, items can unexpectedly expire from the cache. Objects that are cached depend on access patterns that may vary, so when issues appear, it is not obvious how to reproduce the problems and fix them.
- **Access patterns**: When access patterns change, the performance also changes. This increases cache misses; thus, latency increases, and throughput drops.

Cache replacement policies

Cache replacement policies, also called cache algorithms or cache replacement algorithms, are optimizing instructions that applications can use to manage a

cache of stored data on the computer by choosing which items to discard to make room for the new ones.

Some of these policies include the following:

- **First in first out (FIFO)**: This algorithm works similar to a normal queue where the cache evicts the blocks in the order they were added—the first one added is the first one removed irrespective of how often or how many times the blocks were accessed before.
- **Last in first out (LIFO) or first in last out (FILO)**: This algorithm works similar to a normal stack where the cache evicts the block added most recently the last one added is the first one removed irrespective of how often or how many times the blocks were accessed before.
- **Least recently used (LRU)**: This page replacement algorithm keeps track of page usage over a short period. It checks what pages have been most heavily used in the past few instructions and concludes that they are most likely to be used heavily in the next few instructions too.
- **Time aware least recently used (TLRU)**: The time aware least recently used algorithm is a variant of LRU and is designed for the situation where the stored contents in the cache have a valid lifetime.
- **Most recently used (MRU)**: The most recently used page replacement algorithm discards the most recently used items first. These algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.
- **Random replacement (RR)**: The random replacement algorithm randomly selects a candidate item and discards it to make space when necessary.
- **Segmented LRU (SLRU):** The segmented LRU cache is divided into two segments:
    - **Probationary segment**: Missed data is added to the cache at the most recently accessed.
    - **End of the probationary segment**: Hits are added to the most recently accessed end of the protected segment.
- **Least-frequently used (LFU)**: This algorithm Counts how often an item is needed. Those that are used least often are discarded first.
- **Least frequent recently used (LFRU)**: The *least frequent recently used* cache replacement algorithm combines the least-frequently used

algorithm and least recently used algorithm.

- **LFU with dynamic aging (LFUDA)**: This variant adds a cache age factor to the reference count when a new object is added to the cache or when an existing object is re-referenced.
- **Low inter-reference recency set (LIRS)**: The low inter-reference recency set page replacement algorithm has improved performance over the least recently used algorithm. This is achieved by using reuse distance as a metric for dynamically ranking accessed pages to make a replacement decision.
- **Adaptive replacement cache (ARC)**: This balance between the least recently used algorithm and the least-frequently used algorithm improves the combined result.
- **AdaptiveClimb (AC)**: AdaptiveClimb uses recent hits and misses to adjust the jump where in climb, any hit switches the position one slot to the top, and in the least recently used algorithm, hit switches the position of the hit to the top.

A small example of LRU follows:

```
using System;
using System.Collections.Generic;
class LRU_Ex
{
//Find page faults
static int ObtainPageFaults(int []Pages, int PageLength, int
PageCapacity)
    {
    //Page present?
    HashSet<int> CurrentPages = new HashSet<int>(PageCapacity);
    Dictionary<int, int> LRUIndexes = new Dictionary<int, int>();
    int NoofPageFaults = 0;
    for (int i = 0; i < n; i++)
    {
     if (CurrentPages.Count < PageCapacity)
     {
      if (!CurrentPages.Contains(Pages[i]))
      {
       CurrentPages.Add(Pages[i]); //Store recently used page
       NoofPageFaults++;
```

```
    }
    if(LRUIndexes.ContainsKey(Pages[i]))
     LRUIndexes[Pages[i]] = i;
    else
     LRUIndexes.Add(Pages[i], i);
   }
   Else //Remove least recently used page
   {
    if (!CurrentPages.Contains(Pages[i]))
    {
     int PagesInSet = int.MaxValue, CurrIndex = int.MinValue;
     foreach (int j in CurrentPages)
     {
      int temp = j;
      if (LRUIndexes [temp] < PagesInSet)
      {
       PagesInSet = LRUIndexes[temp];
       CurrIndex = temp;
      }
     }
     CurrentPages.Remove(CurrIndex);
     LRUIndexes.Remove(CurrIndex);
     CurrentPages.Add(Pages[i]);
     NoofPageFaults++;
    }
    if(LRUIndexes.ContainsKey(Pages[i])) //Update page index
     LRUIndexes[Pages[i]] = i;
    else
     LRUIndexes.Add(Pages[i], i);
   }
 }
 return NoofPageFaults;
 }
 public static void Main(String []args)
 {
 int []Pages = {8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3};
 int PageCapacity = 4;
 Console.WriteLine(ObtainPageFaults(Pages,
      Pages.Length, PageCapacity));
```

```
    }
}
```

In the previous code segment, we make feed the following into memory:

```
8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3
```

When the app starts, all four slots are empty. Why four slots? Because we set the **`PageCapacity`** object to 4. Now, let us break this process down. We allocate the first 4 numbers to the empty slots (8, 1, 2, and 3). At this stage, we sit with 4-page faults.

In the process of allocating the fifth number, 1, it ends up being a 0-page fault because the number 1 is already present in one of the allocated slots. This is where it gets interesting: In the process of allocating the sixth number, 4, the number 4 will take the place of the first memory slot, which currently occupies the number 8. This is due to the fact that the number 8, in this case, is the least recently used page fault.

The next number, number 1, is already in memory, so it is a 0-page fault. The number 5 will take the place of the third memory slot, which currently occupies the number 2. The rest of the numbers would result in 0-page faults as they are all already in memory.

A small example on FIFO follows:

```
using System;
using System.Collections;
using System.Collections.Generic;
class FIFO_Ex
{
    static int ObtainPageFaults(int []Pages, int PageLength, int
    PageCapacity)
    {
    HashSet<int> CurrentPages = new HashSet<int>(PageCapacity);
    Queue FIFOIndexes = new Queue() ;
    int NoofPageFaults = 0;
    for (int i = 0; i < PageLength; i++)
    {
     if (CurrentPages.Count < PageCapacity)
     {
      if (!CurrentPages.Contains(Pages[i]))
      {
       CurrentPages.Add(Pages[i]);
```

```
            NoofPageFaults++;
            FIFOIndexes.Enqueue(Pages[i]);
          }
        }
        else
        {
          if (!CurrentPages.Contains(Pages[i]))
          {
            int val = (int)FIFOIndexes.Peek();
            FIFOIndexes.Dequeue();
            CurrentPages.Remove(val);
            CurrentPages.Add(Pages[i]);
            FIFOIndexes.Enqueue(Pages[i]);
            NoofPageFault++;
          }
        }
      }
      return NoofPageFault;
      }
      public static void Main(String []args)
      {
      int []Pages = {8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3};
      int PageCapacity = 4;
      Console.Write(ObtainPageFaults(Pages, Pages.Length,
      PageCapacity));
      }
}
```

When the numbers 8, 1, 2, and 3 are allocated, each occupies a slot and thus a page fault each. When the next 1 is allocated, it is a 0-page fault as it is already in memory. When the number 4 arrives, it replaces the number 8, which currently occupies the first slot. This happens again when the number 5 arrives, but it replaces number 1.

In-memory cache

An in-memory cache is a data storage layer that sits between applications and databases. This helps to deliver responses quicker because it stores data from earlier requests or copied directly from databases. With the help of an in-

memory cache, reading data from memory is quicker than from the disk. It improves online application performance.

Let us quickly do an example demonstrating in-memory caching with C#. Follow these steps:

1. Create a new **`Console (.NET Framework)`** application named **`InMemoryCache_Ex`**.
2. Right-click on the projects inside the **`Solution Explorer`**.
3. Select **`Add`**.
4. Select **`Reference`** this is shown in the following figure:

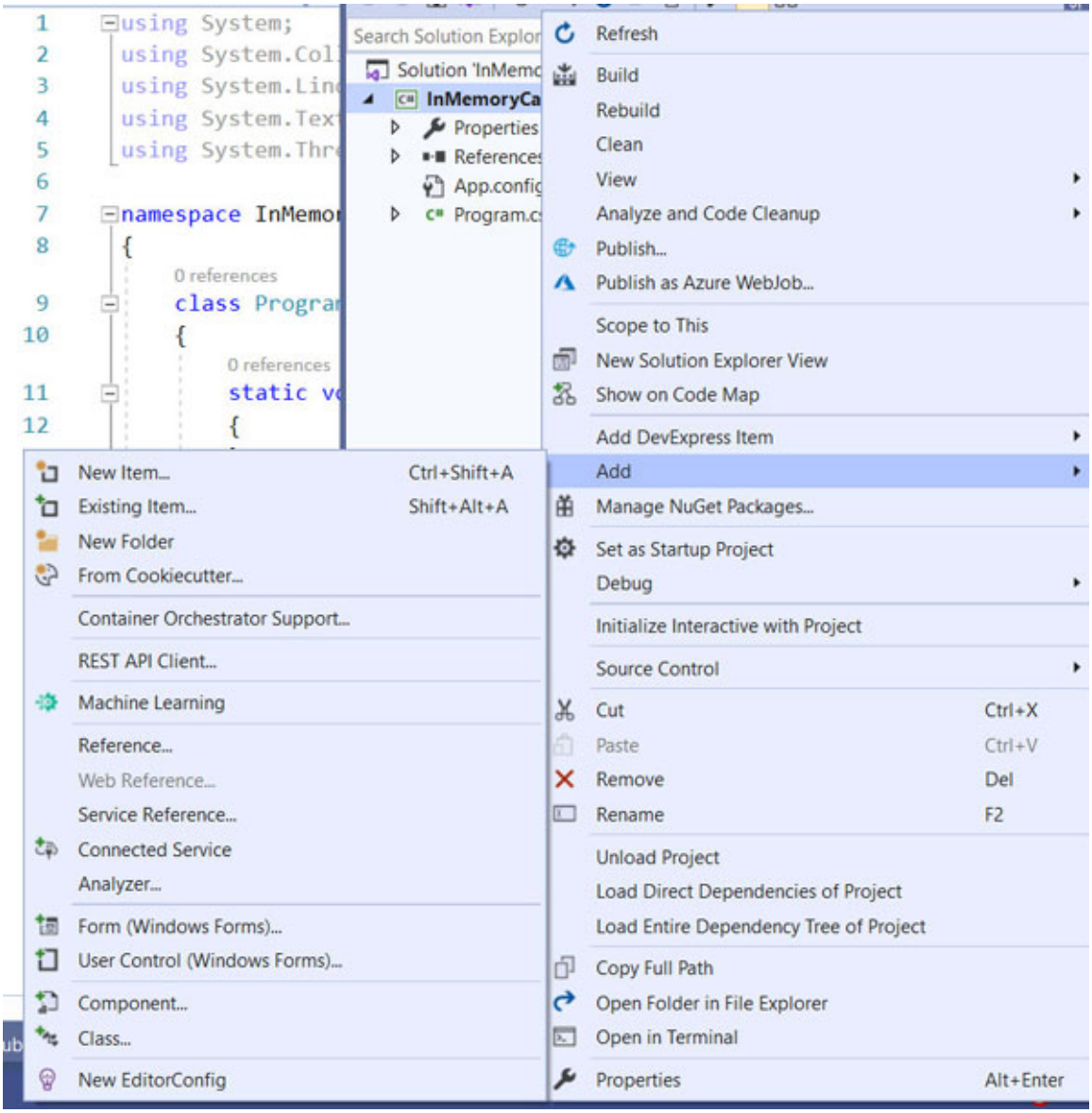


Figure 9.1: *Add references*

5. In the dialog box that appears, select **`Assemblies`**.
6. Scroll down to find **`System.Runtime.Caching`**.
7. Select the checkbox next to it to select it.

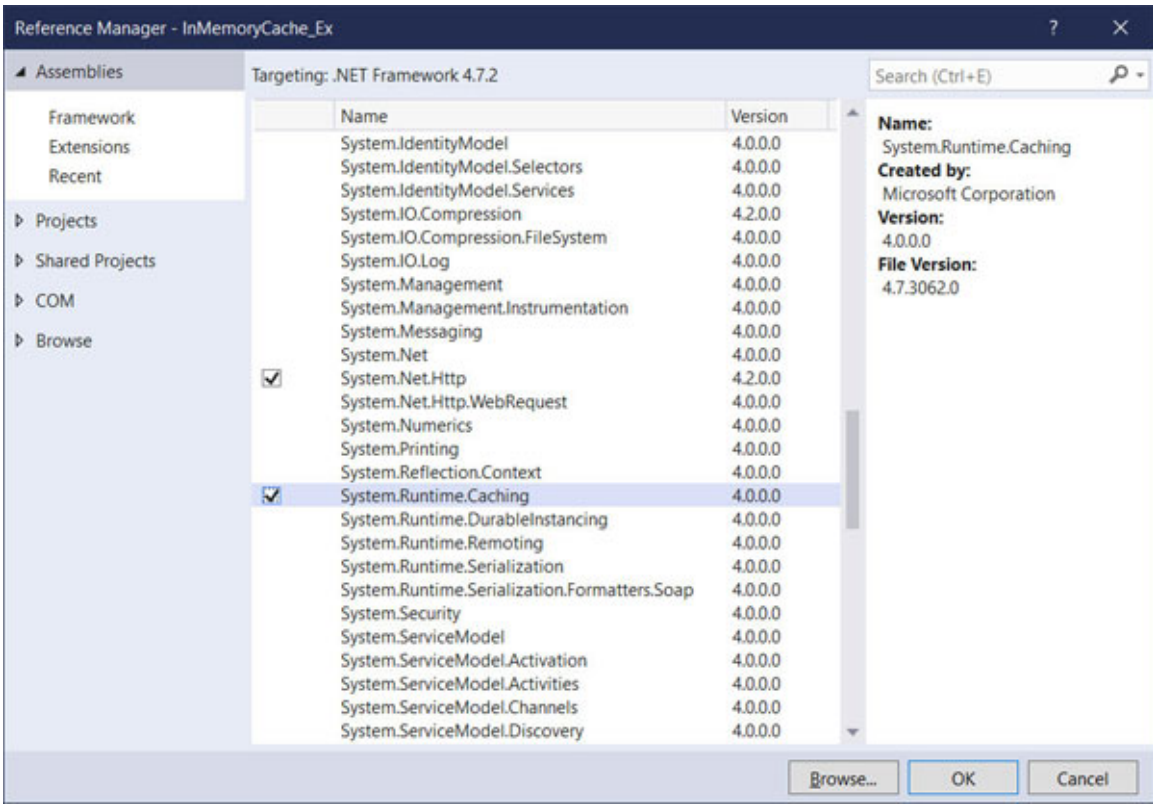


Figure 9.2: *System.Runtime.Caching*

8. Click **OK**.
9. Edit the code to look like the next code segment:

```
using System;
using System.Runtime.Caching;
namespace InMemoryCache_Ex
{
 class Program
 {
  static void Main(string[] args)
  {
   ObjectCache oCache = MemoryCache.Default;
   oCache.Add("FirstCache", " FirstValue", null);
   oCache.Add("SecondCache", 2, null);
   var cipPolicy = new CacheItemPolicy
```

```
            {
             AbsoluteExpiration =
             DateTimeOffset.Now.AddSeconds(90.0),
            };
            oCache.Add("ThirdCache", "Lives in Memory for 90
            Seconds", cipPolicy);
            var cItem = new CacheItem("FourthCache", "
            Fourthvalue");
            oCache.Add(cItem, cipPolicy);
            Console.WriteLine("FourthCache - " +
            oCache.Get("FourthCache"));
            oCache.Remove("Firstcache");
            oCache.Set("SecondCache", 20, null);
            Print(oCache);
          }
          public static void Print(ObjectCache oCache)
          {
            foreach (var c in oCache)
            {
              Console.WriteLine("Cache Key and Value:" + c.Key + " -
              " + c.Value);
            }
            Console.ReadLine();
          }
        }
    }
```

In the previous example, we set various cache objects at various times and display them afterward by looping through them.

Persistent in-process cache

A persistent in-process cache is a cache that is backed up outside of process memory when you back up your cache outside of process memory. The backup could be a file or a database. It might be in a file or in a database. This means that if the process is restarted, the cache is not lost.

Distributed cache

A distributed cache pools together **Random Access Memory** (**RAM**) of networked computers into a single in-memory data store and is then used as a data cache that provides fast access to data. A distributed cache grows beyond the memory limits of a single computer by linking together multiple computers (distributed architecture or distributed cluster), which allows incremental scaling by adding more computers to the cluster, allowing the cache to grow in sync with the data growth.

When to consider implementing a distributed cache

Some considerations to keep in mind when thinking of implementing a distributed cache are as follows:

- **Application acceleration**: Dramatically reduces bottlenecks of disk-based systems, causing the application to run much faster.
- **Web session data**: We can load balance web traffic over several application servers so that we do not lose session data when an application server fails.
- **Decreasing network usage and costs**: By caching data in multiple places in your network, we can reduce network traffic and leave more bandwidth available for other applications that depend on the network.
- **Extreme scaling**: Leverage more resources across multiple machines.

Conclusion

In this chapter, we learned about caching in applications. We explored the various types of caching. We covered the advantages and disadvantages of caching. Finally, we had a look at common caching mistakes.

In the upcoming chapter, we will look at the Large Object Heap. Common problems of the LOH and where we can improve on garbage collection.

Points to remember

- A cache hit happens when a program makes a request to the cache for content, and the content has been saved inside the cache.
- A cache miss occurs when the requested content is not stored in the cache.

- When a cache hit occurs, the content inside the cache will load much faster because the cache can immediately deliver the content to the end user. When a cache miss occurs, the information should first be sourced and then saved inside the cache.
- Caching allows developers to achieve performance improvements.
- Cache replacement policies are optimizing instructions that applications can utilize to manage a cache of stored data on the computer.

Questions

1. Explain the importance of caching.
2. Name the benefits of caching.

Answers

1. Caching allows developers to achieve performance improvements.
2. They are :

   (a) Performance. (b) Offline work. (c) Resource efficiency.

Key terms

- Caching
- In-memory cache
- Persistent in-process cache
- Distributed cache

References

- **Ten Caching Mistakes that Break your App:** http://highscalability.com/blog/2014/7/16/10-program-busting-caching-mistakes.html

CHAPTER 10

Working with the Large Object Heap

Introduction

Memory is not straightforward. Any object gets placed on a heap. A heap is simply not a singular continuous object but a series of chunks. Understanding how this works is crucial in understanding memory manipulation

Structure

The topics to be covered are as follows:

- Memory management in .NET
- Large object heap
- Common problems with the large object heap

Objective

By the end of this chapter, the reader will explore the memory and the garbage collector. It covers the large object heap and the small object heap, and all the associated collection generations.

Memory management in .NET

Before we can dissect the **large object heap** (**LOH**) in .NET, we must first understand how .Net handles memory in several ways. It has to cater to small .NET objects belonging to the **Small Object Heap** (**SOH**) and larger .NET objects belonging to the LOH, which we will delve into in the next topic. There is also the need to cater to weak references and the garbage collector.

Garbage collection generations

Small .NET objects (less than 85,000 bytes/85KB) are allocated onto the SOH, which is divided into three generations:

1. Generation 0
2. Generation 1
3. Generation 2

Objects move up these generations based on their age.

Generation 0/Gen 0

Generation 0 is the youngest generation and contains short-lived objects such as temporary variables, which causes garbage collection to happen the most frequently. A new generation of objects is formed when new objects are allocated. This generation of newly generated objects is Gen 0 collections unless they are large objects and go on the large object heap in a Gen 2 collection.

In the event of Gen 0 becoming full, the .NET Garbage Collector will run and will dispose of objects which are no longer needed and move everything else up to Gen 1. When Gen 1 becomes full, the GC runs again but moves objects in Gen 1 up to Gen 2 instead.

Generation 1/Gen 1

Generation 1 also contains short-lived objects, but it acts as a buffer between short-lived objects and long-lived objects. Objects that survive a Gen 1 garbage collection are promoted to Gen 2.

When the garbage collector performs a collection of Generation 0, the GC compacts the memory for the reachable objects and promotes them to Gen 1. Objects get promoted to Gen 1 because they have a longer lifetime, and they survive the collection of Gen 0 objects.

When a collection of Gen 0 does not reclaim enough memory for the application to create a new object, the garbage collector performs a collection of Gen 1, then Gen 2. Just as in Gen 0, objects in Gen 1 that survive collections are promoted to Gen 2.

Generation 2/Gen 2

Generation 2 contains long-lived objects such as static objects or data that lives for the duration of the process. When objects survive a Gen 2 garbage collection (full garbage collection which reclaims all objects in all generations), they remain in Gen 2.

Collecting a generation means collecting objects in that generation and all its younger generations. A Gen 2 garbage collection is also known as a full garbage collection because it reclaims all objects in all generations.

Large object heap

As we saw in the previous topic, an object that is greater than or equal to 85 KB in size is considered a large object and is allocated onto the large object heap. This number was determined by rigorous performance tuning. Performance tuning was discussed in *Chapter 7: Identifying Performance Problems*.

Large objects belong to Generation 2. When objects survive a Gen 2 garbage collection (full garbage collection, which reclaims all objects in all generations), they remain in Gen 2. When a generation is collected, all its younger generation(s) are also collected. All objects live in managed heap segments. Managed heap segments are chunks of memory that the Garbage Collector reserves from the Operating System by calling certain functions on behalf of managed code.

When the **Common Language Runtime** (for more information on **CLR**, refer to the references section at the end of this chapter) is loaded, the Garbage Collector allocates two initial heap segments: one for small objects (the small object heap, or SOH), and one for large objects (the large object heap).

In the event that a garbage collection is triggered, it scans through all the live objects on the heap and compacts them. Compacting the objects is expensive, so the GC sweeps the LOH. A list of dead objects is made, which can be reused later to satisfy large object allocation requests. Any other dead objects are made into one free object.

When there is not enough free space to accommodate large objects, the GC attempts to acquire more segments (space) from the operating system. If this fails, it triggers a Gen 2 GC to try to free up some space under one of the following three conditions:

- The system is low on memory
- The Garbage Collector's Collect method is called
- Allocation exceeds the Gen 0 or large object threshold

The system is low on memory

The operating system would send a high memory notification to the GC. Based on this, the GC determines if doing a Gen 2 collection will be productive; it triggers one.

The garbage collector's collect method is called

When the **`Collect`** method of the GC is called without parameters or by passing an overload with the **`MaxGeneration`** property, the LOH is collected along with the rest of the managed heap.

Here is an example of the garbage collector's **`Collect`** method:

```
using System;
using System.Runtime;
namespace GarbageCollect_Ex
{
    class Program
    {
    private const int iLimit = 11000;
    static void Main(string[] args)
    {
     Generate();
     Console.WriteLine("Before: {0:N0}",
     GC.GetTotalMemory(false));
     GCSettings.LargeObjectHeapCompactionMode =
     GCLargeObjectHeapCompactionMode.CompactOnce;
     GC.Collect();
     Console.WriteLine("After: {0:N0}",
     GC.GetTotalMemory(true));
     Console.ReadLine();
    }
    static void Generate()
    {
```

```
        Version vUnusedObjects;
        for (int i = 0; i < iLimit; i++)
        {
          vUnusedObjects = new Version();
        }
      }
      }
}
```

In the previous code example, we generate 11,000 random objects to populate memory. **`GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;`** compacts the memory. The Common Language Runtime grows the large object heap incrementally via the **`VirtualAlloc`** method as the objects are allocated. For more information on the **`VirtualAlloc`** method, please refer to the end of this chapter. The large object heap is not a huge contiguous chunk of address space, so there is not really a way for these chunks to be merged. When run, the results may not show a big difference between the initial memory and the compacted memory obtained by the Garbage Collector's **`GetTotalMemory`** method. *Figure 10.1* shows a dramatic difference.

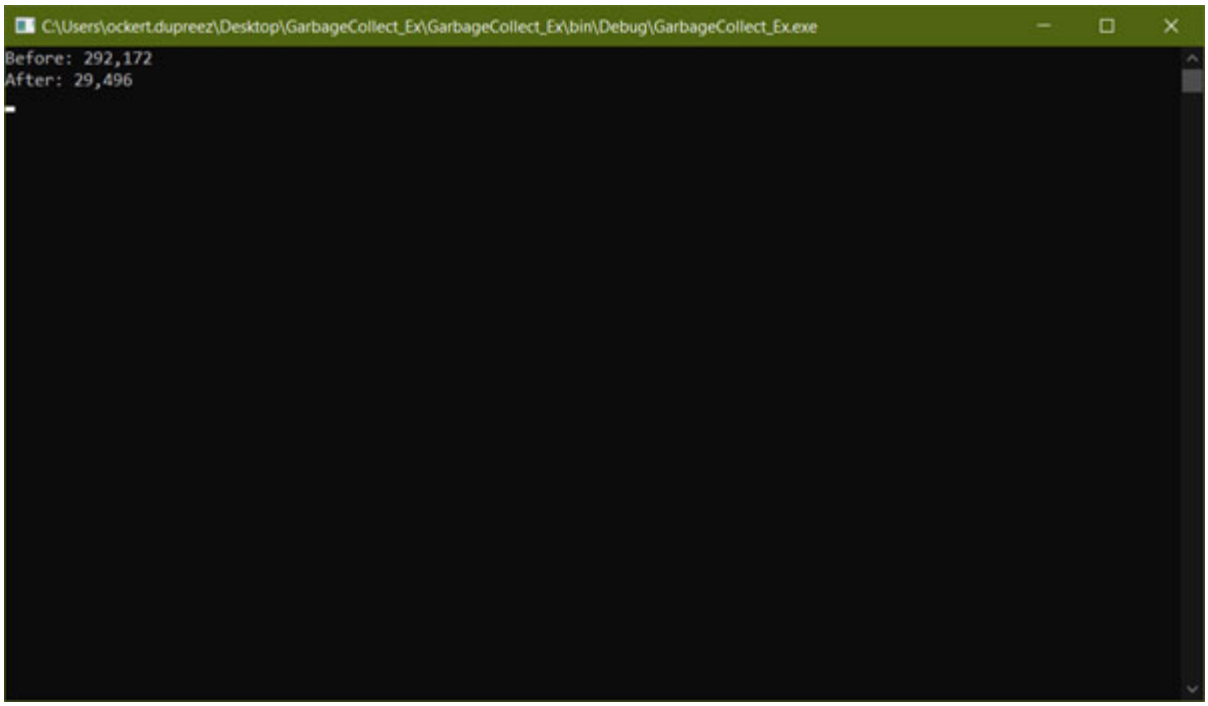


Figure 10.1: *GCSettings.LargeObjectHeapCompactionMode*

For more information on the Version object, refer to the references section at the end of this chapter. Here is an example of the Garbage Collector's **`MaxGeneration`** method:

```
using System;
namespace GarbageCollectMaxGeneration_Ex
{
    class Program
    {
    private const long iLimit = 11000;
    static void Main()
    {
     Program objGenNumber = new Program();
     Console.WriteLine("Maximum Number of Iterations
     Supported: {0}", GC.MaxGeneration);
     Program.Generate();
     Console.WriteLine("Determine Current Collection: {0}",
     GC.GetGeneration(objGenNumber));
     Console.WriteLine("Total Memory: {0}",
     GC.GetTotalMemory(false));
     GC.Collect(0);
     Console.WriteLine("Current Generation: {0}",
     GC.GetGeneration(objGenNumber));
     Console.WriteLine("Total Memory: {0}",
     GC.GetTotalMemory(false));
     GC.Collect(2);
     Console.WriteLine("Generation: {0}",
     GC.GetGeneration(objGenNumber));
     Console.WriteLine("Total Memory: {0}",
     GC.GetTotalMemory(false));
     Console.Read();
    }
      static void Generate()
    {
     Version vUnusedObjects;
     for (int i = 0; i < iLimit; i++)
     {
      vUnusedObjects = new Version();
```

```
        }
      }
      }
}
```

The previous code is a bit similar to the previous code listing. The difference here is that we determine the number of generations that are supported by the garbage collector. We then show each generation's memory. This is shown in the following figure:

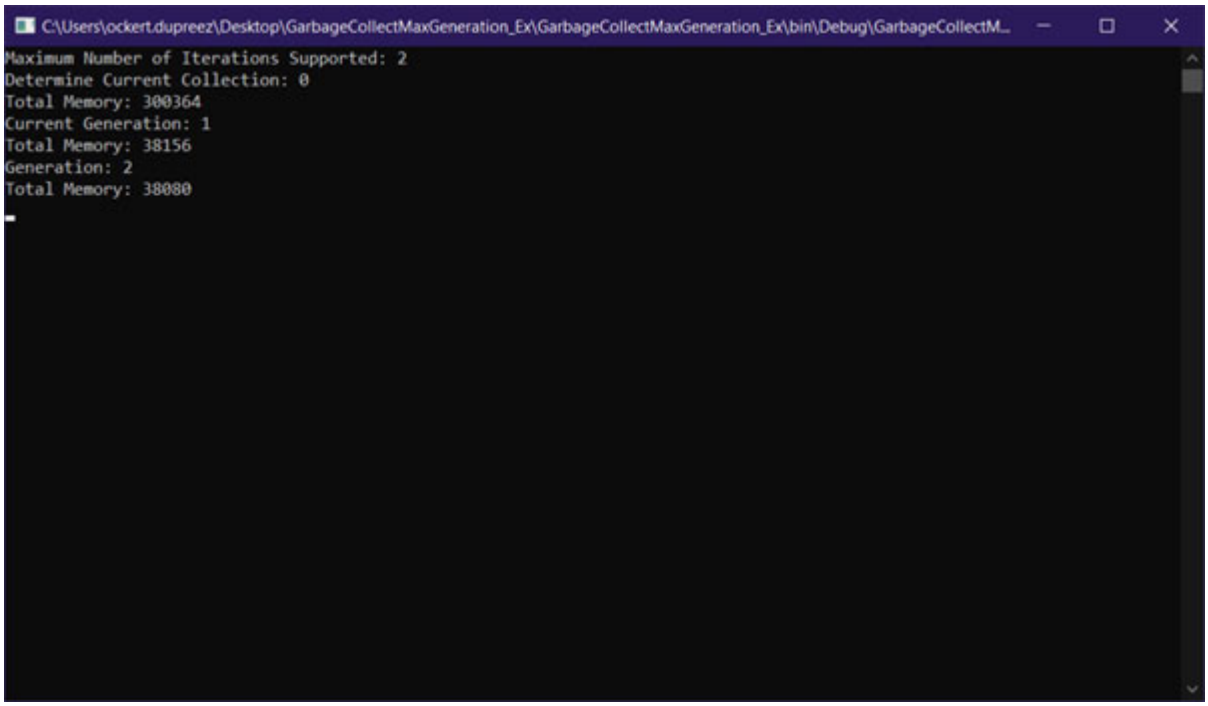


Figure 10.2: *Memory of different generations*

Allocation exceeds the Gen 0 or large object threshold

The generation's threshold property gets set when the garbage collector allocates objects into it. When this threshold gets exceeded, the GC is triggered for that specific generation. These thresholds are dynamically tuned as the program runs.

Common problems with the large object heap

Allocations on the large object heap can impact performance in the following ways:

- **The cost of large object allocation**: With the common language runtime making the guarantee that the memory for every new object it gives out is cleared, the allocation cost of a large object is dominated by memory clearing.
- **The cost of large object collection**: The large object heap and Gen 2 are collected, so if one's threshold is exceeded, a Generation 2 collection is triggered. If a Gen 2 collection is triggered because of the large object heap, Gen 2 might not be much smaller after the garbage collection. If there is not much data on Gen 2, this has minimal impact, but if it is large, it causes performance problems when many Gen 2 collections are triggered.

An example is as follows:

```
class BinaryTree
{
  Data dat;
  BinaryTree left;
  BinaryTree right;
};
BinaryTree[] btTree = new BinaryTree[nodes];
```

If the object nodes are large, the garbage collector will need to go through at least two references per element. Another approach would be to store the index of the right and the left nodes:

```
class BinaryTree
{
  Data dat;
  uint left_index;
  uint right_index;
} ;
```

Conclusion

In this chapter, we learned about how memory heaps are used in applications. We explored the various types of heaps—large object heap, small object heap, and the generations of the garbage collector.

In this chapter, we have looked at how to create a responsive user interface. It will concentrate on threads and background code.

Points to remember

- Generation 0 is the youngest generation and contains short-lived objects such as temporary variables, which causes garbage collection to happen the most frequently.
- Generation 1 also contains short-lived objects, but it acts as a buffer between short-lived objects and long-lived objects.
- Generation 2 contains long-lived objects such as static objects or data that lives for the duration of the process.

Questions

1. Explain the term SOH.
2. Explain the term LOH.

Answers

1. Small .NET objects (less than 85,000 bytes/85KB) are allocated onto the small object heap (SOH).
2. An object that is greater than or equal to 85KB in size is considered a large object and is allocated onto the large object heap.

Key terms

- Common Language Runtime
- Garbage Collector
- GCSettings.LargeObjectHeapCompactionMode

References

- **Common Language Runtime:** https://forums.codeguru.com/showthread.php?369619-NET-

Framework-General-CLR-What-is-the-Common-Language-Runtime

- **Version class: https://docs.microsoft.com/en-us/dotnet/api/system.version?view=net-5.0**
- **VirtualAlloc: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc**

CHAPTER 11

Creating a Responsive UI

Introduction

Designing Web applications is easy. Designing Web applications properly, well, that is on a totally different level. When designing any application, consideration should be made for what type of audience the application is made. Do the users have any afflictions, and do the designers have enough information or too much information?

Structure

The topics to be covered are as follows:

- Creating a user interface
- Common pitfalls

Objective

By the end of the chapter, the reader will know how to write proper code to cater for all users. We will cover how to create a user interface properly and go through all the pitfalls that can be encountered.

Creating a user interface

A User Interface can make or break applications, irrespective of what the application can do or not. Creating a user interface should be planned out properly and tested afterwards, not only for functionality but also for user experience. If the user does not have a good understanding of what the application is about or how to navigate it, the user will leave the site within seconds.

Before we get to the rights and wrongs of application design, let us first understand what goes on in the background.

UX versus UI

User interface (**UI**) and **user experience** (**UX**) are commonly confused with one another. It is especially important to understand the subtle differences and similarities between the two. The UI usually works with a device's features, including the screen, buttons, scrolling, and sound. The user experience deals with everything a user experiences from the start to the end. Let us take an online shopping site or/and application, for example.

The designers responsible for the UI should ensure the users can search and see all the various products nicely, indicate specials or quick delivery options and indicate how to order a product. This is done with visual elements, icons, buttons, colors, text, and application responsiveness.

Conversely, UX designers should ensure the customer can find the product they were looking for and ensure the correct product was delivered when it was supposed to. So, the UX, as mentioned earlier, does not solely concentrate on the user interface features and ease of use but on the customer's overall experience with the brand, which is the online store. This opens more possibilities. What if the customer did not get the product when he/she was supposed to? How does an order get cancelled? Is there an after-sales service? In the end, any brand needs satisfied customers, and that is what the UX assists in doing.

Any customer experience is a pleasant experience, right?

No, user experience can be positive, negative, or neutral. The important ones to note are positive and negative. To ensure positive experiences with an application, a few elements come into play, such as information architecture, project management, visual design, user research, content strategy, accessibility (more on this later in this chapter), analytics, and UI. Therefore, understanding the customer's journey from start to finish is essential in developing good interactions with the customers.

A UX designer must be able to do the following tasks to facilitate a good customer experience:

- Plan
- Research
- Information organization
- User flow

- Skeleton/wireframe
- Testing
- Final analysis

Common UI design mistakes

Making UI mistakes, albeit small ones, can hinder your site's popularity, and ultimately your products or services will get lost in translation. People can be tricky, and finding the true balance of what most people find acceptable takes a lot of effort and experience. This topic will cover some of the most common UI mistakes.

Some of the mistakes include the following:

- **Fonts**: Fonts can quickly become a huge problem. If not used properly, your design will not flow as it should or, worse, confuse users. Most notable font-related problems that should be avoided are:
    - **Distracting fonts**: Fonts can be really distracting when used in the wrong environment. Take a resume as an example. A resume will not look good and will not be taken seriously if it is in a font such as Lucida Handwriting, Viner Hand ITC, or Comic Sans Serif. The same goes for an application. In general, fonts should complement all the other UI elements, not interfere with any of the application's functionality and must work well with each page's mission.
    - **Using too many fonts**: Fonts, such as the application, need to convey a message or make any text easy and more legible. Using too many fonts confuses the user and messes up the whole flow of the application. When using more than one font cannot be avoided, try to make use of fonts of similar styles.
- **Cluttered layouts**: It is difficult for designers not to get too overly creative with UI designs. Some designers tend to forget the singular principle of focusing on the customer's demands and not on theirs. Having a cluttered app frustrates users because there are too many elements on it, and the elements are competing for the customer's attention.

This causes confusion for the customers or users; they may end up losing the intended message of the application or website and perhaps also miss potential specials and discounts.

When an application is cluttered, visitors spend too much time searching for what they need, even if they do not exactly know what they are searching for, and then become uninterested, overwhelmed, frustrated, and irritated. They will not come back.

Some guidelines to avoid cluttering include the following:

  - Do not have too much information on the home page (landing page); rather, consider only including a page banner (hero image), engaging headlines, and examples of the benefits of the service/products.
  - Other pages should only focus on their respective actions. This allows the customers or users to intuitively know what step to take next and where that step will lead them.
  - Product pages should ensure that there is sufficient whitespace between descriptions, images, forms, and Call to Actions (CTAs).
  - Use colors and fonts sparingly. More on colors a bit later.

- **Too much information**: Over half of users spend less than 15% on a page. If the user does not promptly get to where he/she wants to be, they will exit. If the page has too much text, the users will struggle to find what they are looking for. Most users skim through a page instead of reading all the content on it, especially when looking for specific information.

  To prevent people from leaving the site because of unorganized information, try to use less text by using fewer words. Proofread the pages, review pages, test pages, and try to be more concise, expressive, and to the point.
- **Long forms**: The purpose of any form, whether it be on an application or website, is to collect information. Some designers tend to forget the importance of a form, so let us get back to basics. Being a designer, a developer should be able to maintain a fine balance between knowing which information to collect, auto-saving features, input prompts, and

anything else a form needs to do to accomplish what it was created to do.

Users, in general, do not want to enter information, which is not essential, so why put it on a list? In the same breath, users want to control the data they do enter. Users also like to navigate within forms with ease. Colors and accessibility, as mentioned later, also play a huge role in designing forms. If a developer does not understand the user's needs, they should try to get in the users' shoes.

When designing forms for applications or Web forms, things should not get lost in translation, so here are a few issues to avoid.

- The size of the form is too big, and smaller items on the application page may get overshadowed.
- The form can be inaccessible by the user's keyboard.
- The form might be too difficult for the users to scroll down or to select specific options.
- The form might try to collect too much information or useless information.

- **Contrast**: Contrast is the state of being strikingly different from something else in juxtaposition or close association. To avoid dull and low contrast applications, keep the following in mind:
  - Make use of contrasting elements so that the user can navigate through the application's interface more smoothly.
  - Use contrast to separate the different sections of the application. This could be to use different background colors showing clear differences between headers, content, and footers.
  - Use contrast to separate elements from the background. This may be because there is a sharp contrast between the text (which should be legible) and associated photos.
  - Using different font types and/or font sizes on different elements. But do not overuse fonts; as explained earlier in the Distracting fonts section, use different saturations (bold and normal) instead.
  - Use whitespace to show contrast.

    - Do not highlight narrow page elements with color. Instead, make use of a single-color background for an entire block on the application's design.

  To find out about a tool that can be used to identify contrasts between colors, please have a look at the references section at the end of this chapter.

- **Navigation**: Navigation allows users to flow through the applications. Navigation should not be complicated for the users—it should be easy and self-explanatory. Navigation can be broken up into seven distinct parts, some good, some bad, as you will see, and they are:
    - **Page hierarchy**: Organization is a skill. When the navigation and text on an application are unorganized, it is difficult to navigate through it, and users will give up trying to find what they are looking for and go and look for it somewhere else.

      A lack of organizational hierarchy confuses visitors and gives off an unflattering impression of the brand it is trying to advertise or inform about. Navigation should be easy to understand and follow to page visitors.

      Page hierarchy can be formed via the use of headings, subheadings, and paragraph font sizes to indicate the text's various levels. Descriptions should have a logical order. Headings must convey the company or product's mission statement and tell all customers what is on offer. Conversely, subheadings must explain how customers can benefit from it.

      Navigation should also be placed on the site or application so that users can easily find and understand it, and it should be clear, and it should always meet the expectations of visitors.

      Make the menu obvious to find by placing it where users expect to find it, for example, horizontally on top or on the left as a vertical sidebar. Also, try to keep the number of menu items limited to five or six.
    - **Page real-estate**: It is all good and well to have some sort of navigation on the websites or applications, but more often than not, they include too many navigational options. This not only takes up too much page real estate but also tends to confuse users

because they are not sure which option to take to find what they are looking for. One cardinal rule about navigation is that users should find what they are looking for quickly from any of the site's entry points.

Cluttering a website with too much of, for example, search boxes, main navigation, sub-navigation, parallax navigation, multi-media-based menus, quick links, table of contents, and footers in one site is an overkill. Even using just some of these options together.

These are not the only types of navigation that is available; here are some of the navigation options used currently:

- Interactive Navigation
- Sticky Nav Bars
- Vertical Sidebar Static Navigation
- Vertical Sliding Navigation
- Parallax Navigation
- Hamburger Menu
- Multimedia-based Menu
- Table of Contents
- Related Post Navigation
- Mega Drop-Down Menus
- Single Page Dot Navigation
- Universal Navigation
- Breadcrumb Navigation
- Fat Footers Navigation
- Responsive Subnav Menus
- Top Story Carousels
- Slide-out Menu

We will not go into detail about these navigation options as it is beyond the scope of this book, so for more information on these, please refer to the references section at the end of this chapter.

- **Directional cues**: A directional cue is any element of UI that gives visitors a visual hint that lets the navigation options faster and easier. Directional cues guide visitors to the key elements, text lines, and call-to-action elements; thus, ensuring the conversion is reachable, and visitors' problems are solved quicker.

  Directional cues provide the following advantages to navigation:

  - Enhance the page or screen scannability
  - Strengthen the visual hierarchy
  - Improve navigation
  - Increase conversion rates.

  Some types of directional cues include the following:

  - **Arrows**: An arrow is probably the most obvious visual cue. Arrows on interfaces ensure that people of any age, culture, level of education, and tech literacy find their way through the page easily.
  - **Pointers**: As the name implies, pointers point at something. On interfaces, pointers can be used to direct the visitor's attention to an important call to action or valuable information elegantly.
  - **Eye line/gaze direction**: Peoples' eyes usually focus on some points or objects; because of this, people have a natural urge of curiosity to check what it is. Eye line cues are often used with visual content like photos or illustrations integrated into UI layout.
  - **Visual prompts**: Besides arrows, icons and illustrations can symbolize the type of interaction

- **Mystery meat navigation (MMN)**: MMN is a form of navigation UI whereby the links and image maps are set up so that target of the links is not visible until the user/visitor points the cursor at them. This means that users and visitors are not able to distinguish links from text and have to guess where each link points to. For more information on MMN, please refer to the references section at the end of this chapter.

- **Newsletters**: As the name implies, making use of newsletters is usually a clever idea to further engage the desired audience, but many websites fail to do the following:
    - **Failing to provide engaging letters**: Remember, with a newsletter, the application can reach far more users. Some users tend to not visit a site too regularly because the newsletter provides more information, but they are still active in participating with the business. This happens only if the newsletter provides compelling information to keep the users engaged enough.
    - **Forcing users to sign up**: Many sites and applications force newsletter sign-ups. This is usually done by displaying an annoying "Sign up" banner each time a user visits. If the content on the application is not good enough, chances are that the visitor or user will not waste their time signing up. Some websites display a huge banner on every single page of their application, whereas some websites only provide a link in the site's footer.
    - **Optimizing Newsletters**: Content should stay relevant with the times. A newsletter containing more or less the same content will cause users to unsubscribe very quickly. This is where the creators of the website should know their audience. If, for example, the website is about technology, the content needs to be regularly updated as the technology landscape changes quite quickly.

      Here, as well, a technology website should cover the entire spectrum of technology, not just computers and hardware. Breakthroughs such as technology in medicine, technology in space, and technology in schools could be covered, just to name a few. Know the audience.
    - **Promoting newsletters further**: Only current visitors to the site know about the newsletter. Chances are fifty-fifty in them signing up. Is this enough? Probably, depending on the popularity of the website, but in most cases, it is not.

      So, what are the next steps?

Make use of social media to advertise the website, for example. Team up with companies covering the same topic and propose to promote their products in exchange; revise the content of the website and/or newsletter so that it is more engaging to the users.

- **Sending out too many newsletters**: Visitors are busy. This could be with their work or personal time. Sending newsletters twice or three times a day can become problematic to the users; sending out daily newsletters can also become irritating, especially if the newsletter does not contain a nice lead-in to the topics covered. It is a fine balance to know how often a newsletter should be sent, but as mentioned earlier: know the audience!

- **Error pages**: A relatively new trend is to have funny error pages. Funny, in this case, means a picture with a message. Authors put too much time and energy into making a funky error page, but the question remains: Why is there an error in the first place? Why is the link being clicked on not navigating to where it should?

  This begs the question: Is the site regularly maintained?

  Obviously, adding a nice error page can be quicker to get the error message through, so it is ultimately the author's decision to include one.

- **Browser navigation**: A browser's purpose is to display Web applications and websites. It provides navigation options such as the back button and the forward button, the address bar, and even window capabilities.

  Many designers rely too much on these instead of having a decent site map planned out. It is frustrating to click the browser's back button every time to go back to the original page. For instance, if the site is an online shop. The landing page displays all the products. Clicking on a product loads the new page over the older page instead of a new tab or a custom popup [page. Imagine how frustrating it can be to continuously click Back on the browser.

  On the other side of the coin, designers do not have a nice site map in place but also not good historical navigation in the

browser. If a site does not have proper navigation as well as not making enough use of the tools the browser provides, it is a problem, and the user might get frustrated and leave the site.

- **Design inconsistency**: We may have covered design inconsistency before—or parts of it, but there are two more particularly important things to consider when it comes to buttons. They are:
    - **Icons**: Many modern websites and apps make use of icons, or pictures, instead of ordinary buttons to add to the design, and these icons are easier on them as well as easier for the user to determine the buttons' purpose.

      When the icons are not in the same style, it confuses users, and the CTA's may not be met. Some pointers to consider are as follows:

        - **Be consistent:** This means that the design of the icons should look similar. Outlines of the icons and the color-filling of the icons should be carefully planned.
        - **Make use of vectors:** Vectors or SVG (scalable vector graphics) give the advantage of all devices would display the icons similarly by not affecting the UI.
    - **Primary and secondary buttons**: A primary button is the main call of action on the page. A secondary button is usually a button that should be, well, a secondary option. Look at any app with a login button. The login button is the main button on the screen because most users will click on there first. The reset password or forgot password buttons are the secondary buttons because they are not clicked on as much as the login button.

So, it should make sense that the login button or icon should stand out more than its counterparts. The secondary buttons should not look like the primary button. They should not look the same, as this confuses users and is considered a sloppy design.

Colors

Colors are quite a difficult subject to understand. Why? Because of color blindness, colors have positive and negative connotations in certain cultures. Another thing to consider when working with colors is the psychology behind certain colors. Let us have a look.

Color symbolism

Colors affect each of the world's cultures differently. The following is the list of some colors and what they mean to assorted colors:

- **Pink**: Pink is synonymous with love, femininity, caring, and tenderness and things for little baby girls in western and several eastern cultures. In Japan, pink is more associated with men's clothes. In South Korea, pink symbolizes trust, and in Latin America, it represents architecture.
- **White**: In western cultures, white is a symbol of purity, peace, and elegance. Korea and China; however, associate white with mourning, death, and bad luck. In Peru, white symbolizes angels, health, and time.
- **Orange**: Western cultures usually associate orange with autumn, harvest, and warmth. Hindus believe that a variant of orange and saffron is sacred. Other eastern cultures associate orange with health, love, and happiness. Buddhist monks associate orange with positive virtues, and the Dutch royal family considers orange as their personal color. For Egypt, orange represents mourning. Japan and China consider orange for health, love, courage, and happiness. India associates it with fire, and in Ukraine, orange represents strength and bravery and is a symbol of people's unity.
- **Black:** Most cultures associate black with sophistication and, depending on the circumstances, mystery, and mourning. Death, evil, and magic. In Africa, black is a symbol of masculinity, and in the Middle East, black means rebirth and mourning.
- **Purple:** For Britain, India, Brazil, and Thailand, purple represents mourning. In Japan, only the highest-ranked Buddhist monks are allowed to wear purple robes. In western cultures, purple means honor, and Catholics wear purple for sorrow or repentance.

- **Yellow**: Yellow represents pornography in Chinese cultures; in Germany, it is a symbol of jealousy. For France, yellow represents jealousy, weakness, contradiction, and betrayal. Egypt associates yellow with gold and mourning. Some Latin American cultures associate yellow with sorrow, mourning, and death. Yellow represents refinement, wealth, and bravery. Thailand considers yellow to be lucky.
- **Red**: Russia connotates red with revolution and communism. The West considers red to be dangerous, love, action, passion, and energy. In Asian colors, red is conductive to success, long life, joy, celebrations, and prosperity. Red represents wealth, fire, fear, love, beauty, and spirituality. Some African cultures consider red to be a symbol of death. In Nigeria, red stands for aggression and vitality. Iran considers red to be a symbol of courage; for Egypt, it is believed that red is a lucky charm.
- **Blue**: Blue represents authority, trust, and security in North America and Europe. For Greece, Albania, Iran, Afghanistan, and Turkey, blue is regarded as protection. Many Eastern cultures consider blue a symbol of immortality and spirituality. For Ukraine, blue represents health. Hindus associate blue with divine joy and love. In Judaism, blue represents holiness. In China, blue is associated with women, and in Latin America, blue represents wealth, health, and hope.
- **Green**: Green represents corruption in North Africa. It is the national color of Mexico. Green is banned in Indonesia. In the Middle East, green symbolizes luck, wealth, and fertility and is also the traditional color of Islam. In the West, green means a lot of things, including greed, jealousy, freshness, luck, nature, wealth, spring, and inexperience. For Spain, green means independence. For most Asian cultures, green represents youth, fertility, and new life. For China, green means infidelity, and for Japan, green is a symbol of eternal life.

Color blindness

In total, 95.5% of the world's population can see colors normally. Normal vision is trichromatic. This means that most of the population can match any color or tone with a mixture of the three primary colors (red, green, and blue).

The problem here is that designers tend to not forget but only design applications from their color perspective. Nothing wrong with that. However, when an application is designed well enough, color blindness should not even be a problem. Before we go further, let us learn a bit more about color blindness.

There are three types of color blindness; they are as follows:

- **Protanopia (red deficiencies)**: For people suffering from protanopia, reds seem darker than the actual hues and appear beiger-like. Certain shades of green are also confused with reds. The color receptors or cones in the eyes of protanopia sufferers are not sensitive to reds, i.e., long wavelengths.
- **Deuteranopia and deuteranomaly (green deficiencies)**: The eyes of deuteranopia and deuteranomaly sufferers are insensitive to greens, and they perceive shades or hues of blue and yellow. Red, with people with this affliction, are not as dark as people suffering from protanopia.
- **Tritanopia (blue deficiencies)**: In tritanopia, the eyes are marked by the absence of blue cones. Blues and greens might be confused, and yellows may appear as lighter shades of red or may even disappear completely.

Now that we have a better understanding of color blindness, it is easier to see the bigger picture on how to design our applications. Here are some ideas:

- Use both colors and symbols
- Keep it minimal
- Use patterns and textures to show contrast
- Avoid bad color combinations
- Utilize colors and symbols
- Use text labels
- Underline links
- Make primary buttons standout
- Mark required form fields

- Do not use colors to convey valuable information
- Increase contrast between similar colors
- Lighten light colors and darken the dark ones
- Increase saturation of colors
- Use thicker lines
- Problematic colors should not be used side-by-side

Conclusion

In this chapter, we learned how to approach the design of applications. We touched on quite a few topics here, including the differences between UX and UI designers, common UI mistakes and the importance of colors in our applications.

In the upcoming chapter, we will look at how to overcome InterOp challenges.

Points to remember

- Fonts can quickly become a huge problem. If not used properly, your design will not flow as it should or, worse, confuse users.
- A UI usually works with a device's features, including the screen, buttons, scrolling, and sound.
- UX designers should ensure the customer can find the product they were looking for and ensure the correct product was delivered when it was supposed to.
- Colors affect each of the world's cultures differently.

Questions

1. Explain the term directional cue.
2. Name the three types of color blindness.

Answers

1. A directional cue is any element of UI that gives visitors a visual hint that lets the navigation options faster and easier.
2. They are as follows:
    - Protanopia
    - Deuteranopia and deuteranomaly
    - Tritanopia

Key terms

- UX
- UI
- User flow
- Page real-estate
- Interactive navigation

References

- **Identify contrasts between colors: https://colorable.jxnblk.com/**
- **Navigation options: https://fatstacksblog.com/types-of-blog-website-navigation/**
- **Mystery Meat navigation: https://en.wikipedia.org/wiki/Mystery_meat_navigation**
- **Color blindness simulator: http://www.color-blindness.com/coblis-color-blindness-simulator/**

CHAPTER 12

Overcoming InterOp Challenges

Introduction

Native Interoperability (InterOp) is part and parcel of .NET and most operating systems. Without Interoperability, programming languages would not be able to communicate directly to the OS's functions and properties.

Structure

The following topics are to be covered:

- Defining Native InterOp
- Native sized integers
- Function pointers
- Omitting the localsinit flag

Objective

By the end of this chapter, the reader will know about some obstacles we encounter when working with Native InterOp.

Defining Native InterOp

Interoperability allows us to take full advantage of unmanaged code. Any code running under the **Common Language Runtime** (**CLR**) is called managed. Any code such as COM, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code as they do not run under the CLR.

There are mainly two ways to make use of unmanaged code in .NET; they are as follows:

- Platform Invoke (PInvoke)

- C++ InterOp (It Just Works [IJW])

Platform Invoke (PInvoke)

Platform Invoke enables managed code to call unmanaged code; in other words, Platform Invoke enables .NET code to work with unmanaged code, such as COM, C++ components, ActiveX components, and Microsoft Windows API. This works by locating the exported function(s) and marshalling the function(s)' arguments. These components and Windows API function typically reside in **Dynamic Link Libraries** (**DLL**) files on computers.

Dynamic Linked Libraries, also known as shared libraries, exist, as mentioned previously, in files that are separate from the program's executable. This allows the program to make only one copy of the dynamic library's files at compile time. For more information on Dynamic Linked libraries, please refer to the *References* section at the end of this chapter.

Just as there are dynamically linked libraries, there are statically linked libraries which function in an opposite way to the dynamic libraries. Static Libraries can be used by multiple programs, but they are locked into the programs at compile time. For more information on Static Libraries, please refer to the *References* section at the end of this chapter.

Marshalling transforms the memory representation of an object into a data format suitable for storage or transmission. For more information on Marshalling, please refer to the *References* section at the end of this chapter.

An example of **`PInvoke`** in action follows:

```
using System;
using System.Runtime.InteropServices;
class PInvoke_Example
{
    // Import the Win32 MessageBox with DLLImport.
    [DllImport("user32.dll", CharSet = CharSet.Unicode)]
    public static extern int MessageBox(IntPtr hWnd, String
    text, String caption, uint type);
    static void Main()
    {
    // Call system MessageBox function using platform invoke.
```

```
    MessageBox(new IntPtr(0), "Hello BpB!", "Example title",
    0);
    }
}
```

The magic of **PInvoke** happens through the **DLLImport** function. In the previous example, which is common to use, we call the system **MessageBox** from our .NET application.

For more examples on **PInvoke**, have a look at the *References* section at the end of this chapter.

C++ InterOp (It Just Works [IJW])

With C++ InterOp, we can wrap native C++ classes so that they can be consumed by .NET code. Wrapper classes encapsulate resources or other classes to simplify or restrict the interface, such as a COM object, and provide .NET access methods and properties and translate them internally to the function calls to the actual COM interface.

An example on **PInvoke** in Visual C++ and an example on Visual C++ making use of IJW follows. The following two examples are equal but make use of different methods of making use of managed code:

Sample 1

```
// PInvoke_Ex.cpp
using namespace System;
using namespace System::Runtime::InteropServices;
[DllImport("msvcrt", CharSet=CharSet::Ansi)]
extern "C" int puts(String ^);
int main() {
 String ^ pStr = "Hello BpB!";
 puts(pStr);
}
```

Sample 2

```
// IJW_Ex.cpp
using namespace System;
using namespace System::Runtime::InteropServices;
#include <stdio.h>
int main() {
```

```
  String ^ pStr = "Hello BpB!";
  char* pChars =
  (char*)Marshal::StringToHGlobalAnsi(pStr).ToPointer();
  puts(pChars);
  Marshal::FreeHGlobal((IntPtr)pChars);
}
```

With IJW Interoperability, we can not only use our own unmanaged libraries, but all the unmanaged libraries that already exist, including Microsoft Foundation Classes (MFC), Active Template Library (ATL), and Standard Template Library (STL). For more information on MFC, ATL, and STL, please refer to the *References* section at the end of this chapter.

When interacting with COM objects, C++ is not limited to the restrictions of the .NET Framework Tlbimp.exe (Type Library Importer), including limited support for data types and the mandatory exposure of every member of every COM interface. COM objects can be accessed by C++ at will (via **`CoCreateInstance`** and **`QueryInterface`**), and they do not need any separate interOp assemblies.

With C++ InterOp, COM components can be used normally, or they can be wrapped inside C++ Classes, known as custom runtime callable wrappers (CRCWs). These wrapped classes can then be used instead of native types, and data Marshalling is performed transparently. CRCW also causes the resulting class to be accessible to other languages.

Native sized integers

Native-sized integers (both signed and unsigned), depending on the Operating System, can be 32-bit on a 32-bit OS and 64-bit on a 64-bit OS. The next table demonstrates all the differences between signed and unsigned integer types in .NET:

Keyword	Size	Range
`sbyte / System.SByte`	−128 to 127	Signed 8-bit integer
`byte / System.Byte`	0 to 255	Unsigned 8-bit Integer
`short / System.Int16`	−32,768 to 32,767	Signed 16-bit integer
`ushort / System.UInt16`	0 to 65,535	Unsigned 16-bit Integer

`int / System.Int32`	−2,147,483,648 to 2,147,483,647	Signed 32-bit integer
`uint / System.UInt32`	0 to 4,294,967,295	Unsigned 32-bit Integer
`long / System.Int64`	−9,223,372,036,854, 775,808 to 9,223,372,036, 854,775,807	Signed 64-bit integer
`ulong / System.UInt64`	0 to 18,446,744,073, 709,551,615	Unsigned 64-bit Integer
`nint / System.IntPtr`	Depends on platform	
`nuint / System.UIntPrt`	Depends on platform	

Table 12.1: *Integer differences.*

Let us have a look at what the table shows. Here, it shows that Unsigned integers do not have a negative value, thus, increasing their positive range. Signed integers do have both negative and positive values.

The bottom two works a bit different as they are Native sized Integers. As mentioned earlier, native-sized integers can be 32-bit on a 32-bit OS and 64-bit on a 64-bit OS. A small example follows:

```
int a = 4;
nint b = 4;
nint c = b + 1;
c--;
var obj_1 = typeof(nint); //Returns System.IntPtr
var obj_2 = typeof(nuint); //Returns System.UIntPtr
var obj_3 = (a + 1).GetType(); //Returns System.Int32
var obj_4 = (b + 1).GetType(); //Returns System.IntPtr
var obj_5 = (a + y).GetType(); //Returns System.IntPtr
long d = 12;
var obj_6 = (a + d).GetType(); //Returns System.Int64
var res_1 = nint.Equals(a, b); //False
var res_2 = nint.Equals((nint)a, b); //True
var res_3 = b + 1 > a; //True;
var res_4 = b - 1 == a; //False
```

When int is added to a **`nint`**, the result would be **`nint`**. If a long is added to a **`nint`** the result would be long because the **`nint`** could be either 32-bit or 64-bit as it depends on the OS.

Arrays also supports **`nint`** as an index, as shown next.

```
string[] months = { “January”, “February”, “March”, “April”,
“May”, “June”, “July”, “August”, “September”, “October”,
“November”, “December” };
for (nint i = 0; i < 12; i++)
{
    Console.WriteLine(months[i]);
}
```

Constant expressions can also be of type **`nint`** or **`nuint`**. Constant folding is supported for all unary operators **`{ +, -, ~ }`** and binary operators **`{ +, -, *, /, %, ==, !=, <, <=, >, >=, &, |, ^, <<, >> }`**. These operations gets evaluated with Int32 and UInt32 operands instead of native **`ints`**. This provides for consistent behavior on all compiler platforms.

Constant folding is a compiler optimization used by modern compilers. It is the process of recognizing and evaluating constant expressions at compile time instead of computing these expressions at runtime. Constants can hold hard-coded values, or calculated values, i.e., a sum or combination of variables whose values are known at compile time.

A small example of constant folding follows:

```
int f (void)
{
  return 7 + 4;
}
```

In the previous example, the expression (7 + 4) is evaluated at compile time and replaced with the constant 11. This example of constant folding is shown as follows:

```
int f (void)
{
  return 11;
}
```

Function pointers

C# allows us to write unsafe code, meaning we can write unverifiable code in an unsafe context. When dealing with code inside an unsafe context, we can write code that makes use of pointers. We can allocate and free memory blocks and call methods through the use of function pointers.

Properties of unsafe code include the following:

- Methods, types, and code blocks can be defined as unsafe.
- Unsafe code could increase an application's performance.
- Unsafe code is required when a native function that requires pointers is called.
- Using unsafe brings with it security and stability risks.
- Must be compiled with the **`AllowUnsafeBlocks`** compiler option.

The first bullet point previously mentioned that a type could be defined as unsafe. This is in addition to a value type or a reference type. Here is a short example:

```
type* identifier;
void* identifier;
```

`type*` (called a referent type) in this case can be substituted with the desired type. A function pointer is a variable that stores the address of a function that can later be called through it. Function pointers can be invoked and passed arguments just like a normal function call.

In C# function pointers are declared by using the **`delegate*`** syntax, which is similar to the syntax used by delegate declarations, as shown in the following code:

```
unsafe class BpB_Example
{
  void BpB_Example(Action<int> a, delegate*<int, void> b)
  {
     a(44);
     b(44);
  }
}
```

`delegate*` types are pointer types. This means they have all of the capabilities and all of the restrictions of a standard pointer type. Let us explore them quickly.

Capabilities are as follows:

- They are only valid in an unsafe context.
- Only the methods which contain delegate* parameters can be called.

- They cannot be converted to objects.
- They cannot be used as generic arguments.
- They can implicitly convert from **`delegate*`** to **`void*`**.
- They can explicitly convert from **`void*`** to **`delegate*`**.

Restrictions are as follows:

- Custom attributes cannot be applied to a **`delegate*`** or any of its elements.
- **`delegate*`** parameters cannot be marked as params.
- **`delegate*`** types have the same restrictions as normal pointer types.
- Pointer arithmetic cannot be performed directly on function pointer types.

Have a look at the next code snippet:

```
unsafe class BpB_delegate_Ex
{
    public static void Del_Method()
    {
    Console.WriteLine("Del_Method with no parameters.");
    }
    public static void Del_Method(int i)
    {
    Console.WriteLine("Del_Method with int parameter");
    }
    public static void Use()
    {
    delegate*<void> var1 = &Del_Method; // Del_Method()
    delegate*<int, void> var2 = &Del_Method; // Del_Method(int
    i)
    var1();
    var2(7);
    }
}
```

Overload resolution rules work in conjunction with the address-of operator, as shown previously. Method groups will be allowed as arguments to an address-of (**`&`**) operator.

Consider the following example on the fixed statement (which we will discuss in the next topic):

```
int[] arrNum = new int[5] { 50, 60, 70, 80, 90 };
unsafe
{
    fixed (int* ex = &arrNum[0])
    {
    // Create another pointer as ex is fixed.
    int* ex2 = ex;
    Console.WriteLine(*ex2);
    // ex2 occupies four bytes because it is fixed.
    ex2 += 1;
    Console.WriteLine(*ex2);
    ex2 += 1;
    Console.WriteLine(*ex2);
    Console.WriteLine("--------");
    Console.WriteLine(*ex);
    // Dereference.
    *ex += 1;
    Console.WriteLine(*ex);
    *ex += 1;
    Console.WriteLine(*ex);
    }
}
Console.WriteLine("--------");
Console.WriteLine(arrNum[0]);
/*
Expected Output:
50
60
70
--------
50
51
52
--------
52
```

```
*/
```

In the previous example, we made use of both the unsafe keyword and the fixed statement. The example demonstrates how to increment an interior pointer. For more information on interior pointers, have a look at the *References* section at the end of this chapter.

Fixed-size buffers

The fixed statement in C# prevents the garbage collector (GC) from relocating a movable variable in an unsafe context. The fixed statement sets a pointer to a managed variable and "pins" that variable during the execution of the statement. The fixed keyword can also be used to create fixed-size buffers. Fixed-size buffers are particularly useful with methods that interoperate with external data sources.

A fixed array has all the attributes and modifiers that are allowed for regular struct members. But the type of the array must be **`bool`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`sbyte`**, **`ushort`**, **`uint`**, **`ulong`**, **`float`**, or **`double`**.

We can also embed a fixed-sized array in a struct when it is used in an unsafe code block. An example follows:

```
public struct Array_Ex
{
    public char[] chrWord;
    private int intCount;
}
```

In the previous example, **`chrWord`** is a reference, so the size of the **`Array_Ex`** struct does not depend on the number of elements in the array.

There are some differences between regular arrays and fixed-size buffers; here are some differences from a fixed-size buffer's perspective:

- Fixed-size buffers can only be used in unsafe contexts.
- Fixed-size buffers can only be instance fields of structs.
- Fixed-size buffers are always vectors or one-dimensional arrays.
- A fixed size buffer declaration must include the length.

Let us have a look at the next example where fixed fields without pinning are accessed.

```
internal unsafe struct Words
{
    public fixed char chrWords[256];
}
internal unsafe class BpB_Example
{
    public chrWords words = default;
}
private static void GetArray()
{
    var example = new BpB_Example();
    unsafe
    {
    fixed (char* charPtr = example.words.chrWords)
    {
     *charPtr = 'Z';
    }
    char a = example.words.chrWords[0];
    Console.WriteLine(a);
    example.words.chrWords[0] = 'Y';
    Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

The previous example demonstrates how a fixed-size buffer can exist in an unsafe struct. The size of the 256-element char array is 512 bytes. This is because fixed-size char buffers always take 2 bytes per character. The **`bool`** array elements are always 1 byte in size because they are not really appropriate for creating bit arrays or buffers.

Fixed-size buffers are compiled with the **`System.Runtime.CompilerServices.UnsafeValueTypeAttribute`**. This attribute instructs the CLR that a type contains an unmanaged array that can potentially overflow. Memory gets allocated using **`stackalloc`**, which automatically enables features in the CLR that can detect buffer overruns. For more information on buffer overrunning, please have a look at the *References* section at the end of this chapter.

The following example uses pointers to copy bytes from one array to another:

```
static void Copy1()
{
    int arrLength = 100;
    byte[] arrByte1 = new byte[arrLength];
    byte[] arrByte2 = new byte[arrLength];
    for (int i = 0; i < arrLength; ++i)
    {
    arrByte1[i] = (byte)i;
    }
    System.Console.WriteLine("Original");
    for (int i = 0; i < 10; ++i)
    {
    System.Console.Write(arrByte1[i] + " ");
    }
    System.Console.WriteLine("\n");
    Copy(arrByte1, 0, arrByte2, 0, arrLength);
    System.Console.WriteLine("Copy");
    for (int i = 0; i < 10; ++i)
    {
    System.Console.Write(arrByte2[i] + " ");
    }
    System.Console.WriteLine("\n");
    int arrOffset = arrLength - 10;
    Copy(arrByte1, arrOffset, arrByte2, 0, arrLength -
    arrOffset);
    System.Console.WriteLine("Copy");
    for (int i = 0; i < 10; ++i)
    {
    System.Console.Write(arrByte2[i] + " ");
    }
    System.Console.WriteLine("\n");
}
static unsafe void Copy2(byte[] arrSource, int
arrSourceOffset, byte[] arrTarget, int arrTargetOffset, int
intCount)
```

```
{
    if ((arrSource == null) || (arrTarget == null))
    {
    throw new System.ArgumentException();
    }
    if ((arrSourceOffset < 0) || (arrTargetOffset < 0) ||
    (intCount < 0))
    {
    throw new System.ArgumentException();
    }
    if ((arrSource.Length - arrSourceOffset < intCount) ||
    (arrTarget.Length - arrTargetOffset < intCount))
    {
    throw new System.ArgumentException();
    }
    fixed (byte* bSource = arrSource, bTarget = arrTarget)
    {
    for (int i = 0; i < count; i++)
    {
     bTarget[arrTargetOffset + i] = bSource[arrSourceOffset +
     i];
    }
    }
}
```

The previous example makes use of the unsafe keyword that enables the use of pointers in the Copy method. Declaring pointers to the source and destination arrays is done through the fixed statement.

Static delegates

Normal C# delegates contain function pointers and links to chained delegates. A delegate structure is heap-allocated as with reference types with the same memory pressure. When a delegate is used with unmanaged code, it has to be Marshalled.

Now, a static delegate does not have such a complex structure as a normal delegate does. Static delegates are implemented as structs containing only one field, which is a function pointer of type **`IntPtr`**. This ensures that it

has exactly the same memory layout when used in managed and unmanaged code; in other words, blittable. For more information on blittable and non-blittable types, please refer to the *References* section at the end of this chapter.

Omitting the localsinit flag

By adding the `System.Runtime.CompilerServices.SkipLocalsInitAttribute` attribute, we can tell the compiler not to produce the `localsinit` flag. This flag tells the Common Language Runtime (CLR) to initialize all local variables to 0. By initializing all local variables to 0 has had a performance impact in some scenarios, for example, using `stackalloc`, which allocates a block of memory on the stack.

Conclusion

In this chapter, we learned saw some possible scenarios where Native InterOp can give problems. We went into detail about what makes Interoperability tick and why and how to make use of its powerful functionalities

In the Appendixes section, we will get some detail about the third-party tools we can use for high performance and some new C# language features.

Points to remember

- Platform Invoke enables .NET code to work with unmanaged code, such as COM, C++ components, ActiveX components, and Microsoft Windows API.
- Native-sized integers (both signed and unsigned), depending on the Operating System, can be 32-bit on a 32-bit OS and 64-bit on a 64-bit OS.
- The fixed statement in C# prevents the garbage collector (GC) from relocating a movable variable in an unsafe context.
- Fixed-size buffers are compiled with the `System.Runtime.CompilerServices.UnsafeValueTypeAttribute`.

Questions

- Explain what does the CompilerServices.SkipLocalsInitAttribute attribute do?
- What does the fixed statement do?
- Explain the term function pointer

Answers

1. This flag tells the Common Language Runtime (CLR) to initialize all local variables to 0.
2. The fixed statement in C# prevents the garbage collector (GC) from relocating a movable variable in an unsafe context.
3. A function pointer is a variable that stores the address of a function that can later be called through it.

Key terms

- Native InterOp
- Platform Invoke
- Dynamic Linked Libraries
- Static Libraries
- It Just Works (IJW)
- Constant folding
- Function pointers

References

- **Dynamic Libraries: https://www.techopedia.com/definition/27133/dynamic-library**
- **Static Libraries: https://medium.com/@StueyGK/what-is-c-static-library-fb895b911db1**
- **Marshaling: https://docs.microsoft.com/en-us/dotnet/standard/native-interop/type-marshaling**

- **PInvoke: https://github.com/dotnet/pinvoke**
- **ATL: https://docs.microsoft.com/en-us/cpp/atl/active-template-library-atl-concepts?view=msvc-170**
- **STL: https://www.cplusplus.com/reference/stl/**
- **MFC: https://docs.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=msvc-170**
- **Buffer Overrun: https://docs.microsoft.com/en-us/windows/win32/secbp/avoiding-buffer-overruns**
- **Interior Pointers: https://docs.microsoft.com/en-us/cpp/extensions/how-to-declare-and-use-interior-pointers-and-managed-arrays-cpp-cli?view=msvc-170**
- **Blittable vs. non-blittable types: https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types**

Appendix 'A'

Introduction

There are never enough tools to check for application performance. We have covered quite a few throughout this book. In this Appendix, we will quickly run through other tools that can be used to check the performance of applications.

Structure

The following topics are to be covered:

- Visual Studio Diagnostic Tools
- JetBrains DotMemory
- Red Gate ANTS
- dotTrace

Objective

By the end of this Appendix, the reader will have a good resource for some additional performance testing tools. These tools can come in quite handy in any situation, and they are a good fit for any developer's arsenal.

Reference list

- **Visual Studio Diagnostic Tools https://docs.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2022**
- **JetBrains DotMemory https://www.jetbrains.com/dotmemory/**
- **Red Gate ANTS https://www.red-gate.com/products/dotnet-development/ants-performance-profiler/**
- **dotTrace https://www.jetbrains.com/profiler/**

Index

Symbols

95th percentile response time 6

A

AdaptiveClimb (AC) 118
adaptive replacement cache (ARC) 118
Apdex (Application Performance Index) 5
application performance 4
  disadvantages of tuning 6, 7
  monitoring 4
  optimization levels 7-9
  performance metrics 4
Application Programming Interface (API) 87
ArrayList
  example 69, 70
arrays 16, 17
asynchronous code
  analyzing 97

B

BenchmarkDotNet 106
  automation 106
  benefits 106
  reliability 106
  simplicity 106
  using 107-110
BenchmarkDotNet Config
  Analyzers 107
  Columns 107
  Diagnosers 107
  Exporters 107
  Filters 107
  Jobs 107
  Loggers 107
  Orderers 107
  ToolChains 107
  Validators 107
benchmarking 102
benchmarking architecture
  benchmarks 104
  client 104

driver 104
server 104
benchmark testing
benefits 102, 103
components 104
considerations 105
phases 103
test plan, creating 104
benchmark types
breakpoint benchmarking 105
endurance benchmarking 105
load benchmarking 105
spike benchmarking 105
stress benchmarking 105
BitArray
example 70
BlockingCollection 77, 78
boxing (implicit) 75, 76
breakpoint benchmarking 105

C

cache algorithms 116
cache hit 114
cache replacement policies 116
AdaptiveClimb (AC) 118
adaptive replacement cache (ARC) 118
first in first out (FIFO) 116
first in last out (FILO) 116
last in first out (LIFO) 116
least-frequently used (LFU) 117
least frequent recently used (LFRU) 117
least recently used (LRU) 116
LFU with dynamic aging (LFUDA) 117
low inter-reference recency set (LIRS) 117
most recently used (MRU) 117
random replacement (RR) 117
segmented LRU (SLRU) 117
time aware least recently used (TLRU) 117
caching 114
benefits 115
common mistakes 115, 116
disadvantages 115
significance 114
CaseInsensitiveComparer
example 71
CaseInsensitiveHashCodeProvider
example 71
C++ InterOp 157, 158
C# language 11

constants 13
data types 13
naming conventions 16
origin 12
variables 12, 13
variables, creating/declaring 15, 16
code instrumentation 85
debugging 85
performance counters and event logs 85
tracing 85
Collections 17
classes 17
guidelines 79, 80
thread safety 76
Collection type 68, 69
ArrayList 69, 70
BitArray 70
CaseInsensitiveComparer 71
CaseInsensitiveHashCodeProvider 71
queue 72
selecting 74, 75
SortedList 72, 73
Stack 73, 74
color blindness
deuteranopia and deuteranomaly 151
protanopia 150
tritanopia 151
colors 148
color blindness 150
symbolism 148-150
Common Language Runtime (CLR) 35, 129
common UI mistakes 139
clustered layouts 140
colors 148
contrast 141, 142
design inconsistency 147, 148
directional cues 144, 145
fonts 139
long forms 141
mystery meat navigation (MMN) 145
navigation 142, 143
newsletters 145-147
page real-estate 143, 144
too much information 140, 141
components, of benchmark testing
specification of measurement 104
specifications of metrics 104
workload specifications 104
ConcurrentBag 78
conjunctive and patterns 38, 39, 58

constants 13
conversion 53
C# pattern matching enhancements 38
  applying 60-64
  conjunctive and patterns 38, 39
  disjunctive or patterns 39
  negated not patterns 40
  parenthesized patterns 38
  relational patterns 40
CPU usage 4
  analyzing 89-94
  Hot Path 92
  top functions 92

D

data types 13-15, 44
  correct data types 53
  correct data type, selecting 50-52
  reference types 47
  value types 44, 46
  wrong data types 53
delegate* types 163
  capabilities 163
  restrictions 163
diagnostic tools
  asynchronous code, analyzing 97
  CPU usage, analyzing 89-94
  memory usage, analyzing 95, 96
  using 89
disjunctive or patterns 39, 59
distributed cache 125
  considerations 125
do loop 25, 26
Dynamic Link Libraries (DLL) 87, 156

E

endurance benchmarking 105
enums 20, 21
error rates 6

F

first in first out (FIFO) 116
first in last out (FILO) 116
fit and finish features 35
  pattern matching 35, 36
fixed-size buffers 165-169
foreach loop 25

for loop 24, 25
function pointers 34, 162-165

G

garbage collection 5
garbage collection generations
  Generation 0 128
  Generation 1 128, 129
  Generation 2 129

H

hardware interrupts 84
  maskable interrupts (MI) 84
  non-maskable interrupts (NMI) 85
hardware keyloggers
  hidden cameras 89
  keyboard keyloggers 89
  USB disk-loaded keyloggers 89
Hashtable 19, 20

I

if statement 21, 22
implicit type conversion
  example 53, 54
init only setters 32, 33
in-memory cache 122-124
instruction set simulators (ISS) 85, 86
Intermediate Language (IL) opcodes 34
iteration statements 24
  do loop 25, 26
  foreach loop 25
  for loop 24, 25
  while loop 26

K

keylogger tools 88
  hardware keyloggers 89
  software keyloggers 88
keylogging 88
keystroke logging 88

L

large object heap (LOH) 128-130
  conditions 130-133

issues 133, 134
last in first out (LIFO) 116
least-frequently used (LFU) 117
example 118-122
least frequent recently used (LFRU) 117
least recently used (LRU) 116
LFU with dynamic aging (LFUDA) 117
load benchmarking 105
localsinit flag
omitting 170
low inter-reference recency set (LIRS) 117

M

maskable interrupts (MI) 84
memory management, in .NET
garbage collection generations 128
memory usage
analyzing 95, 96
most recently used (MRU) 117
mutable property syntax 30

N

naming conventions 16
Native Interoperability (InterOp) 155
C++ InterOp 157-159
defining 156
Platform Invoke (PInvoke) 156, 157
native-sized integers 34, 159-161
negated not patterns 40, 59
New Interop features 34
function pointers 34
localsinit flag, omitting 35
native-sized integer 34
non-maskable interrupts (NMI) 85

O

operating system hooks 86
keylogger tools 88
keylogging 88
keystroke logging 88
rootkits 87
usage 86
optimization levels 7, 8
algorithms and data structures 8
assembly level 9
build level 8
compile level 8

design level 8
platform dependent and independent optimizations 9
run time 9
source code level 8

P

parenthesized patterns 38, 58
pattern matching 35, 36
Null checks 36
specifics values 37, 38
type tests 36, 37
pattern matching enhancements. *See* C# pattern matching enhancements
performance metrics 4
95th percentile response time 6
Apdex scores 5
application availability or uptime 5
CPU usage 4
error rates 6
garbage collection 5
memory 6
number of instances 4, 5
queue time 5
request rates 5
throughput 5
persistent in-process cache 124
distributed cache 125
phases of benchmark testing
action phase 103
analysis phase 103
integration phase 103
planning phase 103
Platform Invoke (PInvoke) 156
example 157-159
positional parameters 30
positional property definition syntax 31
profiler
code instrumentation 85
diagnostics tools 89
hardware interrupts 84
instruction set simulators (ISS) 85, 86
operating system hooks 86
using 84
profiling 84

Q

queue 18
example 72
queue time 5

R

random replacement (RR) 117
records 30
reference type variables 47
  Object type example 47, 49
  String type example 49
relational patterns 40, 59, 60
request rates metric 5
rootkits 87
  examples 87
  kernel mode 87, 88
  user mode 87
  working 87

S

segmented LRU (SLRU) 117
  end of probationary segment 117
  probationary segment 117
selection statements 21
  if statement 21, 22
  switch statement 23, 24
small object heap (SOH) 128
software keyloggers
  API-based 88
  form-grabbing 88
  kernel-based 89
software profiling 84
SortedList
  example 72, 73
spike benchmarking 105
Stack 19, 73, 74
standard property syntax 30
static delegates 169, 170
stress benchmarking 105
switch statement 23, 24

T

thread safety, in Collections 76, 77
  BlockingCollection 77, 78
  ConcurrentBag 78
throughput 5
time aware least recently used (TLRU) 117
top-level statements 33, 34
type* 162
type conversion 53

U

unboxing (explicit) 75, 76
User Interface (UI)
  common UI mistakes 139-148
  creating 138
  versus, User Experience (UX) 138, 139

V

value types 44
value type variables 44-46
variables 12, 13
  creating/declaring 15, 16

W

while loop 26