

# Learning .NET High-performance Programming

For most of us, "performance" is a word with a single meaning. When it comes to software production, this meaning usually corresponds to something fast. A fast program is a good program. Although this is not a wrong assertion, the meaning of the word "performance" is wider and deeper.

Writing a responsive UI actually means writing a performing UI. Deploying a worker role across Microsoft Azure, which is able to scale up to 100 cores, and handling millions of messages per hour actually means writing a performing workflow. The two examples show two different kinds of performance, and more exist.

Other than multiple meanings, the word "performance" also refers to multiple implementation levels. For example, a developer has to keep the security aspect of his application in mind right from the outset, because using a simple X509 certificate does not make an insecure web application secure. The same is true when it comes to performance.

If we want to create a high-performance application, we have to design a high-performance architecture right from the start, implement performance-oriented strategies, and bring up a good performance-engineered project that is able to assist the wider development project to create valid performance requisites and tests.

As developers, we cannot avoid mastering all the techniques required to help us face day-to-day challenges. Asynchronous programming and parallel programming are two examples. Mastering such techniques helps us create good software in terms of responsiveness and scalability. A clear understanding of the .NET Framework has to become part of the knowledge arsenal for any .NET developer; understanding memory management, process isolation, and thread life cycle are examples.

"High Performance" is a big topic, it covers various aspects of software development. Due to the course duration constraint, we will only focus on two most essential aspects of High-Performance development, namely the programming language and programming techniques. These two aspects are covered here with enthusiasm and expertise.

Despite the course will entirely base on C# as programming language, some of the techniques introduced can be applicable equally when developing software using other programming languages.

## Performance Thoughts

In software engineering, the most misused word is *performance*. Although anyone may like a performing application or website, the word itself hides a lot of meanings, each with specific pros and cons.

A professional programmer must have a deep understanding of the various facets of the term *performance*, as the term assumes different meanings in different scenarios.

A well-performing application should comply with different kinds of performance requirements, which usually change according to the application's architecture and design. It should also focus on the market expectations and (sometimes) what the current development trend is.

As C# programmers, we must add to the generic knowledge about performance-oriented programming. All these skills let us achieve the best results from coding. Choosing the best architecture solution or design pattern will give a boost to long- or short-term performance results (explained later in this chapter). However, implementing these architectures with the wrong design, will nullify expectations of speed or quality we planned. This chapter will guide you on the meanings and facets of the term *performance*, as implied when programming for **Microsoft .NET Framework**:

- Understanding performance
- Performance as a requirement
- Performance engineering
- Performance aspects
- Class of applications
- Technical overview

### Understanding performance

When you talk about performance with respect to the results of an application being developed, it is a word that means *good results* for the *given expectations*.

Without diving into the details of the meaning, it is clear that the keyword here is not the search for *good results* but the comparison between those results with a specific reference value. No static, relative, or ranged value can have any significance without some kind of a legend associated with it.

Diving into the meaning of the phrase *good results for the given expectations*, there is another hidden important key concept: the availability to measure, in *technical terms*, any given aspect of our application. Such terms must be numerically defined, such as a time, a size expressed in Bytes (or multiples), and so on.

In other words, performance is associated with all the measurable aspects of an application.

As software developers, we have to understand client needs. We cannot be simply code writers or technical enthusiasts.

Although we have to mine technical requisites between use cases and user expectations, we have to guide the client to give us useful information regarding their expectations. I know they do not know anything about software engineering, but it is up to us to let them learn at least the basics here. Sometimes, we have to mine requisites by ourselves, while other times we can try to get the client to use the right requisite formula with suggestions, questions, and indications.

Any requisite with no relative or absolute reference value exposed is invalid.

Subtle to define as *not valid* is any performance requisite with some generic numeric needs, without specifying any context or value legend. An example will be a request like a web page response time, without specifying the server load or page computational complexity.

Taking some time to reflect on what we just read, we found another aspect of the term *performance*, which is a technical value and may become a performance indicator only if it's compared to a valid *expected range*.

Let's evaluate another client need. A client asks for a web page to be able to respond in less than 1 second in a low load time window (less than 1,000 active users) or not more than 10 seconds with heavy load equal to or more than 10,000 active users.

Here, we do have a valid request against a value range and environment, but there is still something missing, such as a *shared and documented test case*, which acts as a reference to everyone working on the project.

An example on a valid client's need for performance requirement would be that a client asks for a web application to execute `Test001` in less than one second with less than 1.000 active users online, or be able to execute the same test case in less than 10 seconds with no more than 10.000 active online users.

### Performance as a requirement

Talking about performance as a need for a client (or buyer), it is easy to infer how this is definitely a requirement and not a simple need that a client may have.

In software engineering, the requirement collection (and analysis) is called *Requirements engineering*. We found a couple of specific requirements that we should properly understand as software developers: the **functional** and **non-functional** requirements.

Under the functional requirement, we can find *what* a software must do, and this (and other specifications) codes what we call **software design**. While with a non-functional requirement, we focus on *how* the system (and hence, the software) has to work. This (and other specifications) codes what we call **system architecture**.

In other words, when a client asks for an application to compute something (and if the computation hides a proprietary logic, the formula is part of the requirement as a technical detail), they are asking for a function, so this is a functional requirement.

When a client asks for an application to work only if authenticated (a non-functional requirement), they are definitely asking that an application works in a specific manner, without asking the application to produce a target or goal in the results.

Usually, anything about security, reliability, testability, maintainability, interoperability, and performance guidelines, are all non-functional requirements. When the client asks the software what needs can be satisfied with respect to their business, it is actually a functional requirement.

Although a client may ask for a *fast* or *responsive* application, they are not actually asking for something related to their business or what to do with the software, they are simply asking for some generic feature; in other words, a wish. All such technical wishes are non-functional requirements. But what about a scenario in which the client asks for something that is a business requirement?

Let's say that a client asks for an application to integrate with a specific industrial bus that must respond in less than 100 milliseconds to any request made throughout the bus. This now becomes a

functional requirement. Although this is not related to their business, logically, this is a technical detail related to their domain, and has become a proper functional (domain-related) requisite.

## Performance engineering

Performance engineering is the structure behind the goal to succeed in respecting all the nonfunctional requirements that a software development team should respect.

In a structured software house (or enterprise), the performance engineering is within system engineering, with specific roles, skills, tools, and protocols.

The goal here is not only to ensure the availability of the expected performance requirements during the development stage, but also how these requirements evolve when the application evolves, and its lifecycle up to the production environment, when continuous monitoring of the current performance against the initial requirements gives us a direct and long-range analysis of the system running.

We live in a time when an IT team is definitely an asset for most companies. Although there are still some companies that don't completely understand the definition of IT and think of it as an unnecessary cost, they will at least see the importance of performance and security as the most easily recognizable indicators of a well-made application.

Performance engineering has objectives that cover the easy goal of how to write a *fast* application. Let's take a look at some of these objectives, as follows:

1. Reducing software maintenance costs.
2. Increasing business revenue.
3. Reducing hardware acquisition costs.
4. Reducing system rework for performance issues.

Here, the focus is on all aspects of software development that good performance engineering may optimize. It is obvious that a more powerful application leads to lesser hardware requirements, although it is still obvious that a well-made application needs less reworks for performance issues. The focus is not on the time or money saved, but the importance of thinking about performance from the beginning of the development project up to the production stage. Writing a performing piece of code is an easy task compared to a complete software development project, with *performance* in mind. I know that coding is loved by any developer, but as a professional, we have to do something more.

Reducing the work to fix issues and the cost of having developers working on performance optimization or system tuning after an application is deployed in the production stage enforces the contract with the client/buyer who commissioned the software development. This respects the performance requisites and builds trust with the customer as well as leading to a sensible reduction in maintenance costs.

In performance engineering, a formal performance requisite is coded at the beginning of the development stage, together with software and system architects. Multiple tests are then executed during the development lifecycle in order to satisfy requisites (first) and maintain the level of success at the time. At the end of the production stage, the performance test analysis will act as proof of the work done in programming, testing, releasing, and maintaining of the software, as well as an indicator for various kind of issues not related directly to performance (a disk failure, a DoS instance, a network issue, and so on).

## Performance aspects

When working on performance requirements, in the development stage or for the complete application lifecycle, we have to choose the performance aspects that influence our software and development project. Before writing the code, many decisions will be taken, such as what architecture the software must have, what design to implement, what hardware target will run our software, and so on.

As said previously, anything technically measurable and comparable with a valid value range may become a performance indicator and therefore a performance requirement. The more this indicator becomes specific to the application that is being developed, the more its requirements becomes domain related, while for all the others, they are generic non-functional requirements.

We have to always keep in mind that a lot of things may become performance indicators from the technical standpoint, such as the ability to support multithreading or parallel programming, also system-specific indicators, such as the ability to support multicore or a specific GPU's programming languages, but these are only details of a well-formed performance requisite.

A complete performance requisite usually covers multiple aspects of performance. Many aspects do exist. Think of this requirement as a map, as follows:

Latency	Magnitude					
	0%	20%	40%	60%	80%	100%
Latency						X
Throughput			X			
Resource usage					X	
Availability		X				
Scalability	X					
Efficiency				X		

A performance aspect map is a simple grid, exposing the importance of performance aspects

The first thing to keep in mind is that we cannot have every aspect that is shown in the preceding figure as our primary performance goal. It is simply impossible for hardware and software reasons. Therefore, the tricky task here is to find the primary goal and every secondary or less important objective that our application needs to satisfy. Without any regret, some aspect may become completely unnecessary for our application. Later in this chapter, we will cover a few test cases.



Putting extreme focus on a single kind of performance may lead to a bad performing application.

A desktop or mobile application will never scale out, so why focus on it? A workflow never interacts directly with a client; it will always work in an asynchronous way, so why focus on latency? Do not hesitate to leave some of this aspect in favor of other, more critical aspects.

Let's look at the most important and widely recognized performance aspects.

## Latency

The latency is the time between a request and response, or more specifically, the time between any action and its result. In other words, *latency is the time between a cause and its effect*, such that a user can feel it.

A simple example of latency issues is someone using an RDP session. What lets us feel that we are using an RDP session is the latency that the network communication adds to the usual keyboard and mouse iteration.

Latency is critical in web applications where any round-trip between the client's browser and server and then back to the browser is one of the main indicators about the website's responsiveness.

## Throughput

One of the most misused words, a synonym for power, or for the most part, the synonym for good programming, is throughput. Throughput simply means that the speed rate of anything is the main task of the given product or function being valued.

For instance, when we talk about an HDD, we should focus on a performance indicator to reassume all the aspects of HDD speed. We cannot use the sequential read/write speed, and we cannot use the seek time as the only indicator to produce a throughput valuation. These are specific performance indicators of the domain of HDD producers. The following guidelines are also mentioned at the beginning of the chapter. We should find a good indicator (direct, indirect, or interpolated) to reassume the *speed* of the HDD in the real world. Is this what a performance test suite does for a system and HDD? We can use a generic random 64K read/write (50/50 percent) test to produce a single throughput indicator.

Talking about software, the ability of a workflow to process transactions in a timely manner (such as per second or per hour) is another valid throughput performance indicator.

## Resource usage

This is another key performance indicator that includes everything about resource usage such as memory, CPU, or GPU (when applicable).

When we talk about resource usage, the primary concern is memory usage. Not because the CPU or GPU usage is less important, but simply because the GPU is a very specific indicator, and the CPU usually links to other indicators such as throughput.

The GPU indicator may become important only if the graphical computation power is of primary importance, such as when programming for a computer game. In this case, the GPU power consumption becomes a domain-specific (of game programming) technical indicator.

That being said, it is easy to infer that for the resource usage indicator, the most important feature is memory consumption. If we need to load a lot of data together (in the following chapters, we will see alternatives to this solution), we will have to set up hardware resources as needed.

If our application never releases unused memory, we will face a memory leak. Such a leak is a tremendous danger for any application. `OutOfMemoryException` is an exception, which in the .NET programming world means that no more memory is available to instantiate new objects.

The only chance to find a memory leak is by profiling the entire application with a proper tool to show us how an application consumes memory on a subroutine basis.

## Availability/reliability

This is a key performance indicator for any software serving multiple users, such as a web service, web application, workflow, and so on.

Availability is also the proof of how a performance indicator may also be something not directly related to speed or power, but simply the ability of the software being in up-time, actually running, without issues in any condition. Availability is directly related to reliability. The more a system is available, the more it is reliable. However, a system may become available using a good maintenance plan or a lot of rework. A reliable system is always a strong one that does not need special maintenance or rework because it was well developed at the beginning, and meets most of the challenges that the production stage can produce.

## Scalability

When talking about scalability, things come back to some kind of power—the ability of a single function or entire application to boost its performance—as the number of processors rise or the number of servers increases. We will focus a lot on this indicator by searching for good programming techniques such as multithreading and parallel programming in this and the following chapters, because at the time of writing this book, CPU producers have abandoned the path of single processor power, in favor of multicore CPU architectures. Today, we see smartphones with a CPU of four cores and servers with a single socket of twenty cores each. As software developers, we have to follow market changes, and change our software accordingly to take the most advantages possible.

Scalability is not too difficult to achieve because of the great availability of technologies and frameworks. However, it is not something we can always achieve and at any level. We can neither rely only on hardware evolution, nor on infinite scalability, because not all our code maybe scalable. If they are, it is always limited by the technology, the system architecture, or the hardware itself.

## Efficiency

Efficiency is a relatively new kind of performance indicator. The existence of mobile devices and computer-like laptops since 1975, with the release of **IBM 5100**, opened the way to a new performance indicator of efficiency. Absolute power consumption is a part of the meaning of efficiency, with a new technical indicator named **performance per watt**, an indicator that shows the computation level that consumes a single watt of power.

As software developers, we will never focus on hardware electrical consumption, but we have to reduce, at the most, any overhead. Our goal is to avoid wasting any computational power and consequently, electrical power. This aspect is critical in mobile computing, where battery life is never enough.

Speaking of cloud computing, efficiency is a critical indicator for the cloud provider that sells the virtual machines in a time-based billing method, trying to push as many billable VMs in the same hardware. Instead, for a cloud consumer, although efficiency is something outside of their domain, wasting CPU power will force the need to use more VMs. The disadvantage of this is to pay more to have the same results.

In my opinion, always take into consideration this aspect, at least a bit, if you want to reduce global electrical consumption.

## Class of applications

The performance requirement analysis is not easy to obtain.

A lot of aspects actually exist. As explained at the beginning of the last paragraph, we have to strike the right balance between all performance aspects, and try to find the best for our target application.

Trying to get the best from all the aspects of performance is like asking for no one at all, with the added costs of wasting time in doing something that is not useful. It is simply impossible reaching the best for all aspects all together. Trying to obtain the best from a single aspect will also give a bad overall performance. We always must make a priority table like the aspect map already seen in preceding paragraphs.

Different types of applications have different performance objectives, usually the same per type. Here are some case studies for the three main environments, namely desktop, mobile, and server-side applications.



## Case study: performance aspects of a desktop application

The first question we should ask ourselves, when designing the performance requirements of a desktop class application, is *to whom is this application going to serve?*

A desktop class application serves a single user per system.

Although this is a single-user application, and we will never need scalability at the desktop level, we should consider that the architecture being analyzed has a perfect scalability by itself.

For each new user using our application, a new desktop will exist, so new computational power will be made available to users of this application. Therefore, we can assume that scalability is not a need in the performance requisite list of this application kind. Instead, any server being contacted by this kind of application will become a bottleneck if it is unable to keep up with the increasing demands.

As written by Mr. Jakob Nielsen in 1993, a usability engineer, human users react as explained in the following bullet list:

- 100 milliseconds is the time limit to make sure an application is actually reacting well
- 1 second is the time limit to bring users to the application workflow, otherwise users will experience delay
- 10 seconds is the time limit to keep the users' attention on the given application

It is easy to understand that the main performance aspect composing a requisite for a desktop application is latency.

Low resource usage is another key aspect for a desktop application performance requisite because of the increasingly smaller form factor of mobile computing, such as the Intel Ultrabook®, device with less memory availability. The same goes for efficiency.

It is strange to admit that we do not need power, but this is the truth because a single desktop application is used by a single user, and it is usually unable to fulfil the power resources of a single desktop class system.

Another secondary goal for this kind of performance requirement is availability. If a single application crashes, this halts the users productivity and in turn might lead to newer issues such that, the development team will need to fix it. This crash affects only a single user, leaving other user application instances free by any kind of related issues.

Something that does not impact a desktop class application, as explained previously, is scalability, because multiple users will never be able to use the same personal computer all together.

This is the target aspect map for a desktop class application:

Latency	Magnitude					
	0%	20%	40%	60%	80%	100%
Latency						X
Throughput			X			
Resource usage					X	
Availability		X				
Scalability	X					
Efficiency				X		

The aspect map of a desktop application relying primary on a responding UI

### Case study: performance aspects of a mobile application

When developing a mobile device application, such as for a smartphone device or tablet device, the key performance aspect is resource usage, just after Latency.

Although a mobile device application is similar to a desktop class one, the main performance aspect here is not latency because on a small device with (specifically for a Modern UI application) an asynchronous programming model, latency is something overshadowed by the system architecture.

This is the target aspect map for a mobile device application:

Latency	Magnitude					
	0%	20%	40%	60%	80%	100%
Latency		X				
Throughput				X		
Resource usage						X
Availability			X			
Scalability	X					
Efficiency				X		

The aspect map of a mobile application relying primary on low resource usage

### Case study: performance aspects of a server application

When talking about a server-side application, such as a workflow running in a completely asynchronous scenario or some kind of task scheduler, things become so different from the desktop and mobile device classes of software and requirements.

Here, the focus is on throughput. The ability to process as many transactions the workflow or scheduler can process.

Things like Latency are not very useful because of the missing user interaction. Maybe a good state machine programming may give some feedback on the workflow status (if multiple processing steps occurs), but this is beyond the scope of the Latency requirement.

Resource usage is also sensible here because of the damage a server crash may produce. Consider that the resource usage has to multiply for the number of instances of the workflow actually running in order to make a valid estimation of the total resource usage occurring on the server. Availability is part of the system architecture if we use multiple servers working together on the same pending job queue, and we should always make this choice if applicable, but programming for multiple asynchronous workflow instances may be tricky and we have to know how to avoid making design issues that can break the system when a high load of work comes. In the next chapter, we will look at technologies we can use to write a good asynchronous and multithreaded code.

Let's see my aspect map for the server-side application class, shown as follows:

Latency	Magnitude					
	0%	20%	40%	60%	80%	100%
Latency	X					
Throughput						X
Resource usage					X	
Availability			X			
Scalability				X		
Efficiency		X				

The aspect map of a server-side application relying primary on high processing speed

When dealing with server-side applications that are directly connected to user actions, such as a web service responding to a desktop application, we need high computation power and scalability in order to respond to requests from all users in a timely manner. Therefore, we primarily need low latency response, as the client is connected (also consuming resources on the server), waiting for the result. We need availability because one or more application depends on this service, and we need scalability because users can grow up in a short time and fall back in the same short time. Because of the intrinsic distributed architecture of any web service-based system, a low resource usage is a primary concern; otherwise, the scalability will never be enough:

Latency	Magnitude					
	0%	20%	40%	60%	80%	100%
Latency						X
Throughput				X		
Resource usage						X
Availability			X			
Scalability				X		
Efficiency		X				

A user invoked server-side application aspect-map relying primary on latency speed

The aspect map of a server-side web service-based application carefully uses cloud-computing auto-scale features. Scaling out can help us in servicing thousands of clients with the right number of VMs. However, in cloud computing, VMs are billable, so never rely only on scalability.



It is not necessary to split the aspects trying to cover each level of magnitude, but it is a good practice to show the precedence order.

### Performance concerns as time changes

During the lifecycle of an application living in the production stage, it may so happen that the provisioned performance requisite changes.



The more focus we put at the beginning of the development stage in trying to fulfil any future performance needs, the less work we will need to do to fix or maintain our application, once in the production stage.

The most dangerous mistake a developer can make is underestimate the usage of a new application. As explained at the beginning of the chapter, performance engineering is something that a developer must take care of for the entire duration of the project. What if the requirement used for the duration of the development stage is wrong when applied to the production stage? Well, there is not much time to recriminate. Luckily, software changes are less dangerous than hardware changes. First, create a new performance requirement, and then make all brand new test cases that can be applied to the new requirements and try to execute this on the application as in the staging environment. The result will give us the distance from the goal! Now, we should try to change our code with respect to the new requirements and test it again. Repeating these two steps until the result becomes valid against the given value ranges.

Talking, for instance, about a desktop application, we just found that the ideal aspect map focuses a lot on the responsiveness given by low Latency in user interaction. If we were in 2003, the ideal desktop application in the .NET world would have been made on Windows Forms. Here, working a lot with technologies such as **Thread Pool** threads would help us achieve the goal of a complete asynchronous programming to read/write any data from any kind of system, such as a DB or filesystem, thus achieving the primary goal of a responsive user experience. In 2005, a `BackgroundWorker` class/component could have done the same job for us using an easier approach. As long as we used Windows Forms, we could use a recursive execution of the `Invoke` method to use any user interface control for any read/write of its value.

In 2007, with the advent of **Windows Presentation Foundation (WPF)**, the access to user controls from asynchronous threads needed a `Dispatcher` class. From 2010, the `Task` class changed everyday programming again, as this class handled the cross-thread execution lifecycle for background tasks as efficiently as a delegate handles a call to a far method.

You understand three things:

- If a software development team chose not to use an asynchronous programming technique from the beginning, maybe relying on the DBMS speed or on an external control power, increasing data over time will do the same for latency
- On the contrary, using a time-agnostic solution will lead the team to an application that requires low maintenance over time
- If a team needs to continuously update an old application with the latest technologies available, the same winning design might lead the team to success if the technical solution changes with time

## Technical overview

Until now, we have read about what performance requirement analysis means, how to work with performance concerns, and how to manage performance requirements against the full life cycle of a software development project. We will now learn more about the computing environment or architecture that we can leverage while programming for performance. Before getting into the details of the architecture, design, and C# specific implementations, which will be discussed in the following chapters, we will have an overview of what we could take as an advantage from each technique.

### Multithreaded programming

Any code statement we write is executed by a processor. We can define a processor as a stupid executor of binary logic. The same processor executes a single logic every time. This is why modern operating systems work in time-sharing mode. This means the processor availability is frequently switched from virtual processors.

A thread is a virtual processor that lives within a process (the `.exe` or any `.NET` application) that is able to elaborate any code from any logical module of the given application.

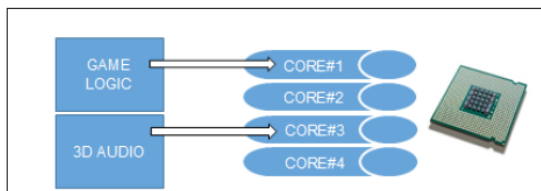
Multicore processors are physical processors, which are all printed in the same metallic or plastic package. This helps reducing some cost and optimizing some external (but still internal to the package) devices such as memory controller, system bus, and often a high-speed cache.

Multithreading programming is the ability to program multiple threads together. This gives our applications the ability to use multiple processors, often reducing the overall execution time of our methods. Any kind of software may benefit from using multithreaded programming, such as games, server-side workflows, desktop applications, and so on. Multithreading programming is available from `.NET 1.0` onward.

Although multithreading programming creates an evident performance boost by multiplying the code being executed at the same time, a disadvantage is the predictable number of threads used by the software on a system with an unpredictable number of processor cores available. For instance, by writing an application that uses two threads, we optimize the usage of a dual-core system, but we will waste the added power of a quad-core processor.

An optimization tries to split the application into the highest number of threads possible. However, although this boosts processor usage, it will also increase the overhead of designing a big hard-coded multithreaded application.

Gaming software houses update lot of existing game engines to address multicore systems. First implementations simply used two or three main threads instead of a single one. This helped the games to use the increased available power of first multicore systems.



A simple multithreaded application, like most games made use of in 2006/2007

### Parallel programming

Parallel programming adds a dynamic thread number to multithreading programming.

The thread number is then managed by the parallel framework engine itself according to internal heuristics based on dataset size, whether or not data-parallelism is used, or number of concurrent tasks, if task parallelism is used.

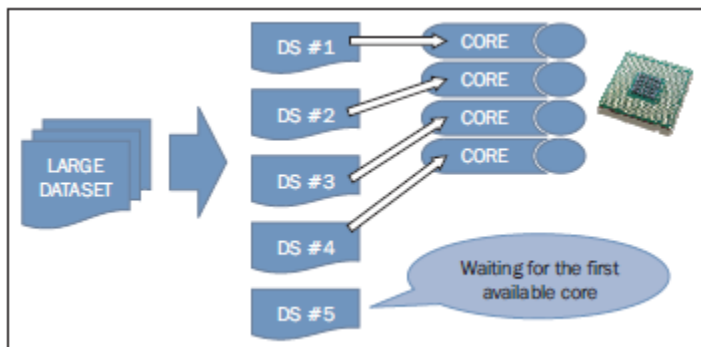
Parallel programming is the solution to all problems of multithreaded programming while facing a large dataset. For any other use, simply do not use parallelism, but use multithreading with a sliding elaboration design.

Parallelism is the ability to split the computation of a large dataset of items into multiple sub datasets that are to be executed in a parallel way (together) on multiple threads, with a built-in synchronization framework, the ability to unite all the divided datasets into one of the initial sizes again.

Another important advantage of parallel programming is that a parallel development framework automatically creates the right number of sub datasets based on the number CPU cores and other factors. If used on a single-core processor, nothing happens without costing any overheads to the operating system.

When a parallel computing engine splits the initial dataset into multiple smaller datasets, it creates a number, that is, a multiple of the processor core count. When the computation begins, the first group of datasets fulfills the available processor cores, while the other group waits for its time. At the end, a new dataset containing the union of all the smaller ones is created and populated with the results of all the processed dataset results.

When using parallel programming, threads flow to the cores trying to use all available resources:



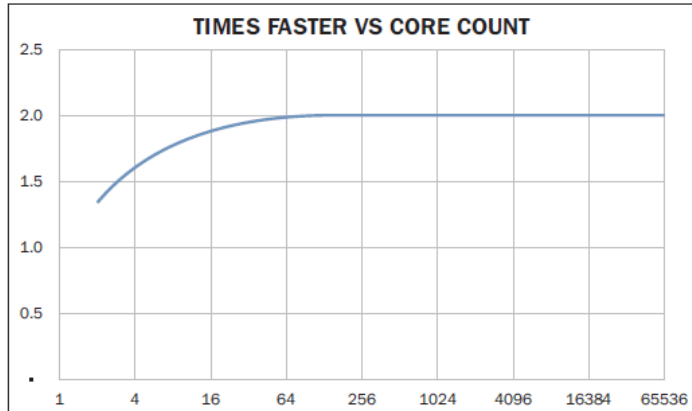
Differently from hardcoded thread usage with the parallelism of Task Parallel Library items flow to all available cores

In parallel programming, the main disadvantage is the percentage of the use of parallelizable code (and data) in the overall application.

Let's assume that we create a workflow application to read some data from an external system, process it, and then write the data back to the external system again. We can assume that if the cost of input and output is about 50 percent of the overall cost of the workflow, we can, at best, have an application that is twice as fast, if it uses all the available cores. Its the same for a 64-core CPU.

The first person to formulate this sentence was Gene Amdahl in his *Amdahl's law* (1967). Thinking about a whole code block, we can have a speed-up that is equal to the core count only when such code presents a perfect parallelizability; otherwise, the overhead will always become a rising bottleneck as the number of cores increases. This law shows a crucial limitation of parallel programming. Not everything is parallelizable for system limitations, such as hardware resources, or because of external dependencies, such as a database that uses internal locks to grant unlimited accesses limiting parallelizability.

The following image is a preview of a 50 percent parallelizable code across a virtually infinite core count CPU:



The execution speed increase of a 50 percent un-parallelizable code. The highest speed multiplication (2X) is achieved about at 100 cores.

A software developer uses the Amdahl's law to evaluate the theoretical maximum reachable speed when using parallel computing to process a large dataset.

Against this law, another one exists, by the name of *Gustafson-Barsis' law*, described by John L. Gustafson and Edwin H. Barsis. They said that because of the limits software developers put on themselves, software performances do not grow in a linear way. In addition, they said that if multiple processors work on a large dataset, we can succeed processing all data in any amount of time we like; the only thing we need is enough power in the number of processor cores.

Although this is partially true only on cloud computing platform, where with the right payment plan, it is possible to have a huge availability of processor count and virtual machines. The truth is that overhead always will limit the throttling multiplication. However, this also means that we have to focus on parallelizable data and never stop trying to find a better result in our code.



## Distributed computing

As mentioned earlier, sometimes the number of processor cores we have is never enough. Sometimes, different system categories are involved in the same software architecture. A mobile device has a fashionable body and may be very nice to use for any kind of user, while a server is powerful and can serve thousands of users, it is not mobile or nice.

Distributed computing occurs every time we split software architecture into multiple system designs. For instance, when we create a mobile application with the richest control set, multiple web services responding on multiple servers with one or more databases behind them, we create an application using distributed computing.

Here, the focus is not on speeding up a single elaboration of data, but serving multiple users. A distributed application is able to scale up and down on any virtual cloud farm or public cloud **IaaS** (infrastructure as a service, such as **Microsoft® Azure**). Although this architecture adds some possible issues, such as the security between endpoints, it also scales up at multiple nodes with the best technology its node can exploit.

The most popular distributed architecture is the **n-tier**; more specifically, the **3-tier** architecture made by a user-interface layer (any application, including web applications), a remotely accessible business logic layer (SOAP/REST web services), and a persistence layer (one or multiple databases). As time changes, multiple nodes of any layer may be added to fulfil new demands of power. In the future, technology will add updates to a single layer to fulfill all the requirements, without forcing the other layers to do the same.



Further reading:

[http://en.wikipedia.org/wiki/Distributed\\_computing](http://en.wikipedia.org/wiki/Distributed_computing)

## Grid computing

In grid computing, a huge dataset is divided in tiny datasets. Then, a huge number of heterogeneous systems process those small datasets and split or route them again to other small processing nodes in a huge **Wide Area Network (WAN)**, usually the Internet itself. This is a cheaper method to achieve huge computational power with widely distributed network of commodity class systems, such as personal computers around the world.

Grid computing is definitely a customization of distributed computing, available for huge datasets of highly parallelized computational data.

In 1999, the University of California in Berkeley released the most famous project written using grid computing named *SETI @ home*, a huge scientific data analysis application for extra-terrestrial intelligence search. For more details, you can refer to the following link:

<http://setiathome.ssl.berkeley.edu/>

## Summary

In this chapter, you read about the meaning and aspects of the term *performance*, the importance of performance engineering, and about the most widely used techniques available to fulfil any performance requirement.

## CLR Internals

This chapter will guide you into the knowledge and usage of the virtual machine in which any Microsoft .NET Framework-based language can actually run: the **Common Language Runtime (CLR)**. Good knowledge of such internal functionalities will help any programmer produce better code, avoiding bottlenecks and anti-patterns.

The important aspects that make .NET so easy to use and powerful at the same time have all been explained in depth in this chapter. Go through this exciting chapter and learn how to work with the most beautiful programming language framework there is.

In this chapter, we will cover the following topics:

- Memory management
- Garbage collection
- Working with AppDomains
- Threading
- Multithreading synchronization
- Exception handling

### Introduction to CLR

CLR is the environment that actually executes any .NET application. A widely used definition is that the CLR is the virtual machine running any .NET application. Although this simple explanation is somehow correct, we must take a step back and explain in depth what C#, Visual Basic, and CLR are.

.NET is a managed programming language that offers the ability to program any kind of application, target any platform, abstract what is usually said to be low-level programming, such as memory management, object initialization, and finalization, and access any operating system, and so on.

C#, VB.NET, F# and many other high-level languages from Microsoft for the .NET Framework and other non-Microsoft languages such as COBOL.NET are human-oriented languages with proper design pros and cons that are usually linked to historical trade area or scientific needs. For instance, management software was usually made in Visual Basic, while low-level programming in C/C++, scientific programming was done with FORTRAN, and banking programming in COBOL.

When dealing with .NET Framework, all compliant languages (such as C# and VB) are actually only the frontend a programmer uses to interact with real .NET language, such as Microsoft **Common Intermediate Language (CIL)**.

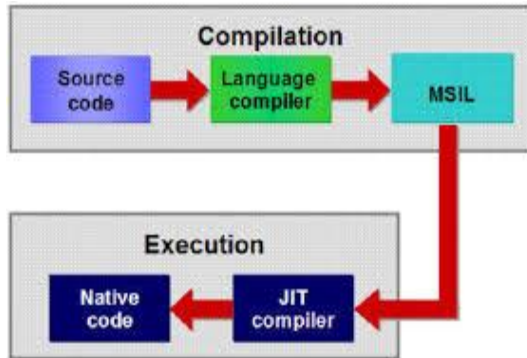
When a programmer builds code with Visual Studio, they trigger the compiler to produce the CIL from the source code. The compiler itself is also usable by any command prompt or script because of being a simple console application.

Together with the **Intermediate Language (IL)**, any compiler of .NET languages also produces relative metadata that is definitely required for datatype validation in class member invocation, to reflect types, members, and so on.

This module was made by IL and metadata, together with a Windows Portable Executable (PE32 or PE32+ for a 64-bit target platform) header, defines the kind of module (.dll or .exe) and the CLR header. This header defines the version of .NET used and relative options, produce a file package

known as Assembly, which also contains the eventually linked resources such as images, icons, and so on.

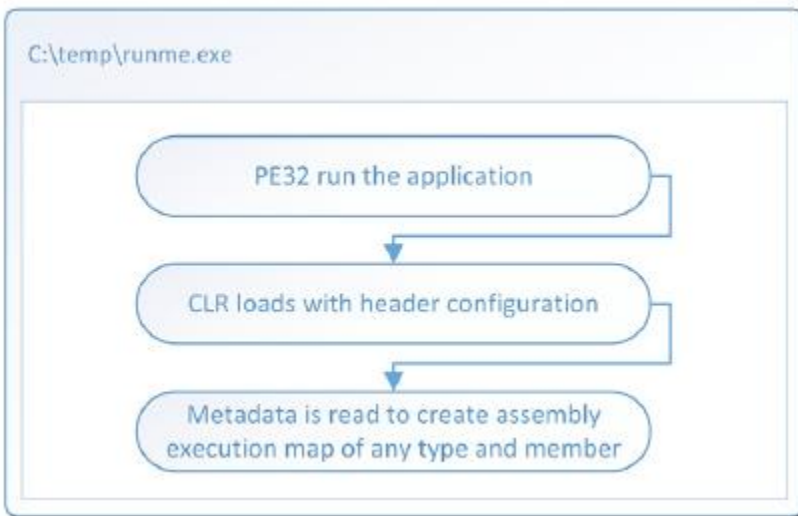
In the following diagram, we see an assembly with all its layers, showing the .NET physical file structure with the system headers, CLR header, code metadata, and body:



From C# to an Assembly

Once the compilation succeeds, we can launch the application (or link the DLL as a reference to other applications) with the usual double-click. The PE32(+) header will run the executable as an unmanaged application, which will try to load the .NET environment by launching the CLR with relative configuration as available in the assembly file, such as the .NET version, requested target platform, and others. On a system without the proper .NET framework runtime available, the whole application will break execution while on any valid system, the application will run normally.

The following is a simplified block diagram that shows the CLR process execution sequence:



The application startup lifecycle within the CLR

Once the metadata loads successfully, any method is ready to run within the **Just-in-Time (JIT)** compiler of the CLR. JIT compiles the platform-independent CIL language in a platform-specific optimized language that can be executed on the underlying system, method by method, in a lazy fashion. Once a method is actually compiled, this compiled code is injected into the in-memory metadata of the assembly, so as to not have to compile it again until the application remains loaded in the memory.

Although the best performance ever available is provided by using native coding, only experts are able to reach similar results. Otherwise, CLR and its JIT compilation produce great code that often performs fine (and sometimes better) than compared to any unmanaged application, if mid-level programmers are involved in the coding. This is because of the great optimization work done by Microsoft when converting CIL to native code.

This comparison is similar to what happens if we compare an OR/M (such as Entity Framework) querying performance with one of the stored procedures . Although the best-ever results are obtainable only by using specific DBMS features available with specific dialect-SQL coding (such as T-SQL, PL-SQL, and so on), only an expert SQL developer is able to provide such kind of querying.

A moderately experienced C# developer is more able at object querying (and such querying is more platform- and database-producer independent) than in specific SQL coding. This is why, for the most part, object-querying will always produce better performing queries, compared to SQL querying performances. In the future, relational databases will be superseded by NoSQL databases. So, for young developers, learning SQL coding is something actually secondary in their professional growth schedule.

## Memory management

When talking about memory management, any code programmer will remember how native languages opened their doors to any kind of issues and bottlenecks. This can also mean that the expert C++ programmer could have access to some customization to produce better memory management than CLR does. However, this relates only to very few people in very few cases.

Theoretically speaking, when a programmer needs to use some memory to store any value in an operation, they need to:

- Define a variable of the chosen type
- Allocate enough free memory to contain the variable:
  - Reserve some bytes in the operating system's memory stack to contain the variable
- Use the variable:
  - Instantiate the variable with the needed value
  - Do whatever is needed with such variable, for example - Define variable, allocate memory, use your variable, deallocate memory
- De-allocate the freed memory:
  - Once the variable becomes useless, free the related memory for further usage by this or other applications

Other than the usual generic programming issues with this step sequence, such as using the wrong type, wasting memory, or going against an overflow of the type, the trickiest memory management issues are *memory leak* and *memory corruption*:

- **Memory leak:** This occurs anytime we forget to de-allocate memory, or by letting the application always consume more memory, and offer an easy-to-predict result.
- **Memory corruption:** This occurs when we free memory by de-allocating some variable, but somewhere in our code, we still use this memory (because it is referred by another variable as a pointer), unaware of such de-allocation. This happens because when we de-allocate variables and relative memory, we must always be sure of updating (or de-allocating) all eventually related pointers that otherwise may still point to a freed memory area that could also contain other data.

CLR helps us by managing memory itself. Thus, in the .NET world, the previous list becomes the following:

- Declaring a variable of any type
- Instantiating the variable with a valid value:
  - CLR makes the difference between value-types and reference-types regarding initial values of variables before assignation. Reference-types have an initial value of `null` (`Nothing` in VB). Value-types (all primitive-types except `string` and `object`), instead, are always valued at the default value. Value-types may support a null value through the class `Nullable<T>` or by adding the character `?` at the type ending, like `int?` (only in C#).
- Using the variable

It is obvious here that memory management is done completely by the CLR, which allocates the needed variable memory plus some overhead (a pointer to a `type` instance and a `sync` block index) as soon as the variable is instantiated. A target-system sized integer pointer that points to an instance of `type` class represents the type of the variable and the another value of the same size used for synchronizing the variable usage. This means that on any 32-bit system, any variable will add 2 x 32-bit values, while for 64-bit systems, a variable will add 2 x 64-bit values. This explains the small additional memory usage that occurs on 64-bit systems. All those objects are arranged in sequence in a memory area called the **managed heap**.



C#/VB variable value assignation is a bit different. C# uses *early binding*, with a built-in type-safe validation for constant and (often) variable values. A down-casting (in terms of numeric type capability) must pass through a *cast* operation such as `int a = (int)longValue;`. When a value outside of the smaller type ranges enters the cast, `-1` becomes the new value. VB, instead, uses late binding that accepts any value assignation (with built-in support for conversions and parsing), by default. Because of the lazy approach, in VB, a numeric conversion must be compliant to the new type's value ranges. Here, a *cast* operation does not occur, so an eventual `OverflowException` is the result of a code like this: `Dim a As Integer = longValue.`

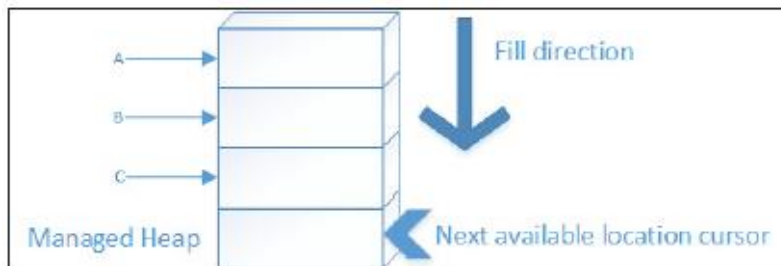
The CLR also manages another internal memory area called the **managed stack**. Each thread handles its own stack (this is why often we refer it as the **thread stack**) by storing all value-types values in a Last-In-First-Out (LIFO) manner. The purpose of CLR is to abstract memory allocation; thus, directly trying to impact that the kind of memory used is actually some kind of inference in CLR itself. To be honest, it's possible to use explicitly stack memory by switching to C# in unmanaged coding (with a proper keyword, such as `unsafe`) using C++ related techniques, or using only value-types such as integers, `double`, `chars`, and so on in managed C#. When using managed C#, the stack memory is available only until we program in a procedural way. This happens because any type within an object (a reference-type) will be stored in the managed heap. Although storing data in the stack will boost the value read/write speed in the memory, it is like programming in the 1960s.

An interesting read is an article by Eric Lippert, the Chief Programmer of C# compiler team at Microsoft. Find it at

<http://blogs.msdn.com/b/ericlippert/archive/2010/09/30/the-truth-about-value-types.aspx>.

The heap is a growing list of bytes that contains a First-In-First-Out (FIFO) collection. It is always slightly bigger than needed, as it quickly accepts new values, exactly the same as any `.NET List<T>` collection. The CLR also has a pointer or cursor that is always pointing to the newly available space for any future allocation.

Here is a diagram showing such FIFO-like memory handling with the new-item cursor:



The heap memory allocation model

This heap population job occurs on a portion of memory that is assigned to the application by the CLR, the address space, in which the Windows environment is actually a Virtual Address Space because it can span from physical memory to page files. This whole application's memory space is then divided into regions—small memory portions side by side to assemble table pages for the compiled CIL, plus metadata and other regions that are eventually created as more memory requests occur.

RAM	Microsoft Windows (32 64 bit)	Application Address Space (32 64 bit)
4GB	3.25GB   4GB	1.5GB   4GB
8GB	3.25GB   8GB	1.5GB   8GB
64GB	3.25GB   64GB	1.5GB   64GB

Memory availability in Microsoft Windows systems and CLR

Although for Windows-based systems the theoretic virtual memory available for application address space is 8 TB (64-bit) or 1.5 GB (32-bit), always remember that the address space may be fragmented. This will easily reduce real address space availability for simple variables like huge collections. This is why a CLR running at 32-bit usually raises an `OutOfMemoryException` error at around 1 GB of memory consumption if we simply populate a huge `List<T>`.

The difficult part of the job of CLR is freeing such a heap: instead of an unmanaged language in which this job is assigned to the programmer, here, the CLR de-allocates the memory just when the variable exits the scope (if it lives in a managed stack) or when there is no more reference by any other object and it exits the scope (if it lives in a managed heap). This job occurs in a lazy fashion with an internal heuristic that also looks for memory requests. This is why, on a system with high address space available, an application that consumes 100 MB of memory can simply continue consuming the same amount of memory, although it is not used anymore if the

application does nothing. However, as soon as possible, when the application needs to create new objects, it could trigger the memory cleanup of the heap by starting an operation named **garbage collection**.

## Garbage collection

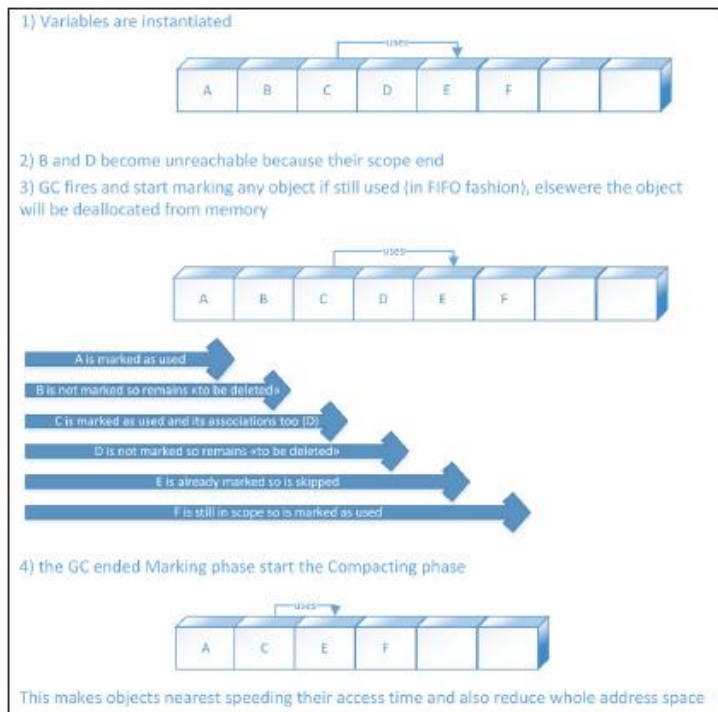
As mentioned, garbage collection (GC) is the engine that cleans up the memory of managed heap within the CLR with an internal algorithm and its own triggering engine. Although it is impossible to know exactly when the GC will fire, its algorithm is detailed in many articles on MSDN and relative blogs and also has known trigger points, for instance, when CLR needs lots of new memory. The GC memory cleanup operation is named **collect**.

Microsoft gives us the ability to trigger the collector manually, by invoking the `GC.Collect` method. Although this option is available, manually triggering the GC is something to avoid because every usage will interfere with CLR abstraction of the underlying system.

The GC collection occurs multiple times until the process is alive and running. Its execution has the goal of freeing the memory from objects that are not in use anymore by any code block, or that are not referred by any other living object.

Any surviving object is then marked as a survivor object. This marking phase is crucial in the GC logic. Each survival will increment the survival counter for such an object. The first time an object is analyzed by the GC is in generation zero of its mark counter. Multiple survivals will bring this counter to generation-1 or generation-2. In CLR, the most unchanging objects (survived through all GCs) are marked in generation-2.

Garbage collection always starts by pausing all threads of the application, and then the managed heap is scanned to find unused objects and can service them. Following is a graphical representation of such behavior:



The garbage collection algorithm

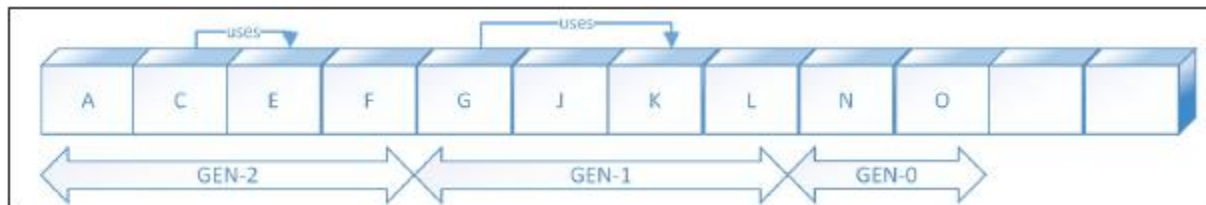
Always bear in mind that the variables seen in the preceding diagram are objects that can contain any number of variables, their self-like basic types, and links to other objects, such as the C item that is associated with the E item.

As mentioned, the GC can trigger its job any time that the application needs to instantiate new objects. This occurs because once started, the CLR defines a threshold in bytes, that is, new object breaks to trigger the GC algorithm. Newly-created objects are referred in the GC as gen-0 (generation-0) objects; they are never analyzed for marking. GC has a generational algorithm that focuses on newly-created objects because they are thought to be the most likely to exit the scope; instead, the objects first created when an application starts are thought to be the most enduring ones. Once an object passes the marking phase, it may be promoted to gen-1; thus, it becomes a long-living object. Any generation has its size limit, as defined by the CLR, so it may also happen that the GC analyzes all objects from gen-0 and gen-1. Usually, the GC only collects the generation that exceeds its size limit.

The choice of what generation to collect is ordered from the newer (gen-0) generation up to the older (gen-2) generation. Because of this, it may happen that if a generation always exceeds its limit, the following generations can never be collected, wasting some memory. Obviously, a manual collection trigger will start the collection of all generations. Although this may seem to be an issue, this algorithm is the result of an intensive study that proves this is generally the most efficient way to clean up memory usage.

Once an object survives two collections from gen-0, GC promotes it to gen-1. Once a gen-1 object survives four collections, it is promoted to gen-2. Gen-2 is the less-changing generation; it is also the less-collected one.

Here is a graphical representation of objects within the virtual address space of the managed heap showing different generations. Bear in mind that, as stated previously, physical fragmentation may occur, although virtual memory seems to be a straight collection of objects.



The managed heap with all available generations

When a process starts, the GC logic within the CLR assigns a size limit for each generation. During runtime of a process, the GC increments or decrements the generation size according to the execution of the application. This means that the GC somehow has a self-learning algorithm that tunes itself, based on how many objects it de-allocates or does not de-allocate.

Exceeding of the allocation threshold is not the only trigger for the GC to start collecting dead objects; it may also run when Windows signals low physical memory, when an `AppDomain` class exits (including the main one), or when the code fires `GC.Collect()` method.



The GC is unable to clean up objects somehow linked to static fields because their scope is the application itself. So, use this design carefully, or else a memory leak could happen.



## Large object heap

CLR divides objects in two sizes: small (less than 85,000 bytes) and large (equal to or greater than 85,000 bytes). All large objects are allocated in a specific heap, the **large object heap (LOH)**. The managed heap is valid for each heap, although the LOH has some limitations because of the size of objects contained within.

With the small object heap, the GC can avoid memory corruption, memory fragmentation, and memory leak because any object is only stored once. It's still possible to create thousands of non-useful items, but this is a behavior of the programmer that CLR cannot avoid. Instead of talking about LOH, the GC will avoid the compacting phase, reducing the thread suspend-time and avoiding costly CPU-intensive work, such as moving large objects in memory. This choice boosts the collection latency (time to finish) but obviously does not help memory consumption by never releasing the unused space between adjacent objects. Instead, the unused space at extremes is always released.

Another great limitation to dealing with using LOH is trying to reduce the collection time. All objects within LOH are marked as gen-2. This means that CLR expects that objects always live long. This causes a great impact on application performance if their real usage is short-lived because the great size will easily exceed the gen-2 size limit, starting the collection phase of such an internal, and usually never-changing, heap area.

## Collection tuning

By invoking the `GC.Collect` method (or when the CLR responds to the Windows low-memory event), it is possible to force start the collection algorithm of any generation. Although this may happen (I always suggest never invoking it manually), GC usually works in a triggered fashion, trying to balance the lowest application performance impact with the needed memory cleanup.

Garbage collection is divided into two different algorithms that fulfill different application needs. We can choose which *garbage collection type* to use within our application only once in the application configuration file (or Web Configuration File), under the `runtime` node, where we can switch from the *workstation* collection (default) to the *server* collection:

```
<runtime>
<gcServer enabled="true" /> <!-- enables Server mode -->
</runtime>
```

When the GC works in the workstation mode (default), the CLR tries to balance the overall execution time of the collection with a few resources, by using a single thread at normal priority to analyze and eventually release the unused memory blocks.

When the GC works in server mode (available only for multicore systems), it creates a thread per CPU core and divides the collection work across those threads that will clean up all managed heaps and LOHs related to all application threads executing on the same CPU core.

Using server collection, we can definitely boost memory cleanup throughput by using multiple cores and avoiding a single thread crossing all CPU cores available. The drawback is higher resource usage because of the increased thread count. The server collection should be configured only for applications that are specific to the server side (such as a database or web server), preferring single-application servers.

The `LatencyMode` property is another configuration available to optimize collection intrusiveness and triggering.

The default collect mode is the **interactive** (or concurrent) mode. With this mode, the collection marking phase works in a background thread (or multiple threads, if using server collection) and only the memory release and compact works by suspending all application threads. This mode is maybe the most balanced one, trying to have good throughput in memory release without consuming too many resources.

The opposite is the **batch** mode (or called as the non-concurrent mode). This mode is configurable within the configuration file, as shown earlier. It can be configured by disabling the concurrent mode, as seen in the following code—the configuration is combinable with the request for using server collection:

```
<runtime>
<gcConcurrent enabled="false"/> <!-- enables Batch mode -->
</runtime>
```

The batch mode is the most powerful in terms of throughput of memory release because it simply suspends all application thread execution and releases all unused memory. Obviously, this choice can break application latency because an application request must await the completion of the collection.

Other `LatencyMode` configurations are available only at runtime by setting the `GCSettings.LatencyMode` property with a value of the `GCConcurrencyMode` enum that contains the batch and interactive values, plus the `LowLatency`, `SustainedLowLatency` and `NoGCRegion` values.

By choosing the `LowLatency` mode (available only for workstation collection), gen-2 collection is suspended completely, while gen-0 and gen-1 are still collected. This option should be used only for short periods when we need a very low interference of the GC during a critical job; otherwise, an `OutOfMemoryException` error may occur. When manually triggering the collector with the related `GC.Collect` method, or when a system is low on memory, a gen-2 collection will occur, although in the `LowLatency` mode. One of the best benefit when using the `LowLatency` mode is increase in application responsiveness because of the collection of only small items. The GC itself uses resources minimally, but in the meanwhile, the process can still consume lots of memory because of the inability to collect long-visibility objects from gen-2.

The `LowLatency` mode is configurable, as shown in the following code:

```
var previousTiming = GCSettings.LatencyMode;
try
{
    //switch to LowLatency mode
    GCSettings.LatencyMode = GCLatencyMode.LowLatency;
    //your code
    //never use large short-living objects here
}
finally
{
    GCSettings.LatencyMode = previousTiming;
}
```

The `SustainedLowLatency` mode is similar to an optimized interactive mode that tries to have more memory retention than the interactive mode actually uses. A complete collection usually occurs only when Windows signals a low-on-memory state. Contrary to the `LowLatency` mode, which must be used only for a very short duration, the `SustainedLowLatency` mode can be chosen as an interactive or batch mode without the occurrence of an out-of-memory state. It is obvious that a system with more physical RAM is the best candidate for such a configuration



An LOH with short-living large objects (that is never collected) plus the interactive mode and the workstation mode usually equals a high memory-consuming application with great freeze time occurring, because of the slow mono-threading garbage collection.

Within *.NET 4.6*, a new mode is available for the extreme purpose of disabling the whole garbage collection process. This mode is named `NoGCRegion`. This choice gives all computational resources to application code, disabling any GC threads. Obviously, such behavior can easily create an `OutOfMemoryException` condition, and its usage should occur only for very short time periods in extreme cases.

This enumeration value (`GCSettings.LatencyMode`) is in read-only. This means that we cannot write the `GCSettings.LatencyMode` property specifying the `NoGCRegion` value. Instead, we need to signal such a critical section by invoking a couple of methods, one to enter and one to exit this section. Here's a code example:

```

try
{
    var neededMemoryAmount = 1000000000;
    //asks GC to stop collecting
    GC.TryStartNoGCRegion(neededMemoryAmount);
    //do your critical stuffs
}
catch (Exception)
{
    //handle the exception
}
finally
{
    //resume previous collect mode
    GC.EndNoGCRegion();
}

```

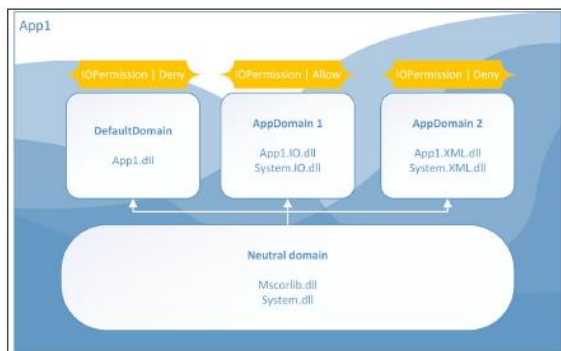
Bear in mind that in the .NET history, the GC algorithm has been updated multiple times, and this may occur again in future versions.

To know more about the fundamentals of garbage collection, visit [https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx).

### Working with AppDomains

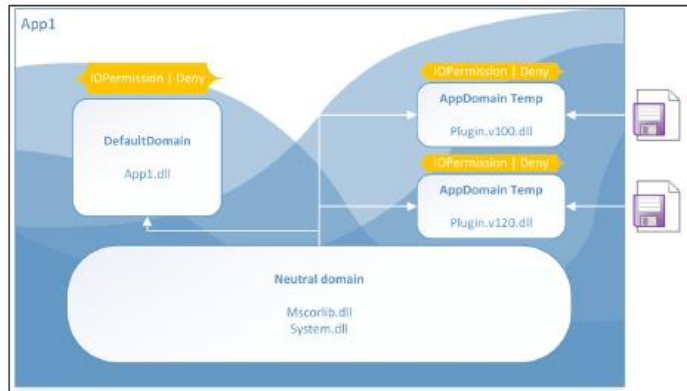
An application domain (AppDomain) is a kind of virtual application. It contains and runs code, starts multiple threads, and links to any needed reference, such as external assemblies or COM libraries.

Application domains can be created to isolate portions of an application and prevent them from directly contacting other portions; to configure different kinds of security authorizations, such as with **Code Access Security (CAS)** techniques to limit I/O access, network access, and so on; or to simply increment the whole security level of the application by isolating different application contexts from others.



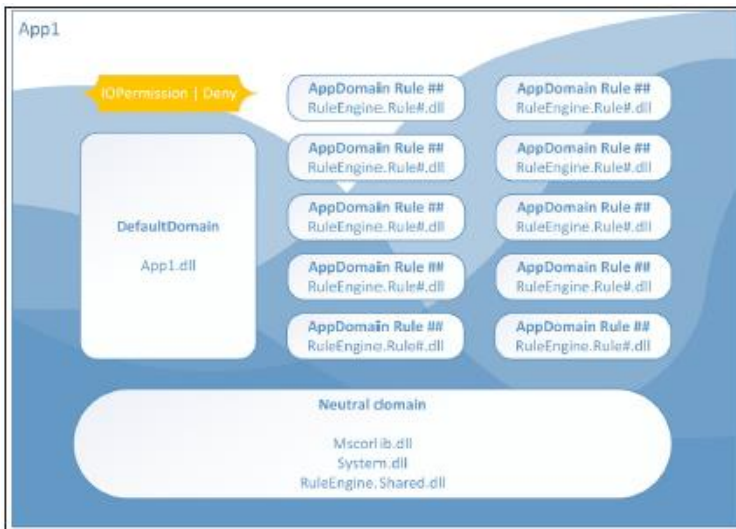
AppDomain usage for reference and/or CAS isolation

The application domains can be unloaded if needed, allowing us to work in a reliable way with multiple external plugins or extensions, like the ones from IoC designs, or simply because we need to load multiple versions of the same assembly all together.



External libs loaded at runtime in temporary AppDomains

Application domains also give us the ability to start multiple applications within a single Windows process that usually consumes several resources in multiple logical applications that are exposed as different application domains, each isolated by others as if they're different processes. The Windows Communication Foundation (WCF) handles its clients in a design similar to the following diagram:



Different assembly rules exposed as different DLL files, all loaded as different virtual processes in the same physical process as AppDomains

Keep in mind that the main goal of using an `AppDomain` class is always isolation. Thus, often some resource usage is incremented. An example is that when loading the same assembly in multiple AppDomains, it will produce multiple instances of the `Type` class for each type that is contained in the referred assembly by each AppDomain. Although the JIT is made only once, the compiled IL is copied across the multiple type-and-metadata tables that live within each AppDomain. The only objection is for the neutral AppDomain, which is a kind of shared AppDomain across all the processes; it does not waste resources and cannot be unloaded by invoking the `AppDomain.Unload` method.

The creation of an `AppDomain` class is straightforward, as shown in the following code snippet:

```
var d = AppDomain.CreateDomain("AppDomain1");
d.ExecuteAssembly("ConsoleApplication2.exe");
AppDomain.Unload(d);
```

The `Load` method of the `AppDomain` class always loads an assembly in the current application domain. Thus, the best way to load assemblies within a defined `AppDomain` class is within its code using the `Assembly.Load` method or by calling `ExecuteAssembly` method, as shown in the preceding example.

The `CreateDomain` method actually returns a proxy to the real objects, giving the other application domains the ability to invoke some method on the remote one (`AppDomain1`). Those proxies are part of .NET Remoting, a distributed programming framework derived from CORBA. Currently, WCF **TcpBinding** is compatible with Remoting, although it is heavily evolved and optimized to fulfill SOA requirements.

When multiple objects live in multiple `AppDomain`, some communication may occur between those domains. Other than the option of using an external component, such as a file or service such as any WCF binding, any instance in any `AppDomain` can produce a `Remoting` proxy to invoke distance methods. Such instances will be marshaled (copied between process boundaries) by value, serializing the object itself, or by reference, with a specific class heritage. Here's an example of this:

```
var domain1 = AppDomain.CreateDomain("domain1");
var domain2 = AppDomain.CreateDomain("domain2");
var byValueType=typeof(MarshalledByValueClass);
var byValue =
(MarshalledByValueClass) domain1.CreateInstanceAndUnwrap(byValueType.Assembly.
GetName().FullName, byValueType.FullName);
var byReferenceType=typeof(MarhalledByReferenceClass);
var byReference =
(MarhalledByReferenceClass) domain2.CreateInstanceAndUnwrap(byReferenceType.As
sembly.GetName().FullName, byReferenceType.FullName);
Console.WriteLine("MarhalledByValueClass -> domain: {0}\tisProxy: {1}",
byValue.DomainName, RemotingServices.IsTransparentProxy(byValue));
Console.WriteLine("MarhalledByReferenceClass -> domain: {0}\tisProxy: {1}",
byReference.DomainName, RemotingServices.IsTransparentProxy(byReference));
AppDomain.Unload(domain1);
AppDomain.Unload(domain2);
Console.ReadLine();
```

These new classes are available, although from another assembly:

```
[Serializable]
public sealed class MarshalledByValueClass
{
    public string DomainName { get; set; }
    public MarshalledByValueClass()
    {
        DomainName = AppDomain.CurrentDomain.FriendlyName;
    }
}
public sealed class MarhalledByReferenceClass : MarshalByRefObject
{
    public string DomainName { get; set; }
    public MarhalledByReferenceClass()
    {
        DomainName = AppDomain.CurrentDomain.FriendlyName;
    }
}
```

In this example, two different objects cross the `AppDomain` execution. The first object (named `byValueType`) used the *by-value* marshaling by being decorated with the `SerializableAttribute` class. This means that when the object crossed boundaries of two `AppDomains`, it got serialized/deserialized each time. Bear in mind that the caller is within the default `AppDomain` class.

The second object (named `byReferenceType`), instead, used the *by-reference* marshaling by inheriting the `MarshalByRefObject` class. Actually, such an object never crosses the boundaries of the two `AppDomains`. A remote proxy is available to remote the `AppDomains` classes to invoke remote methods and read/write remote properties.

The first difference is that for the marshaled-by-value instance, only the constructor actually works in the other domain. Thus, after it is unloaded, the object copy is still alive in the calling domain, ready to do anything else. For the marshaled-by-reference instance, the object actually lives in the remote domain; hence, any proxy usage after the domain unloads will raise an `AppDomainUnloadedException` event, proving that a single instance of such an object exists.

This is the execution console output:

```
MarshalledByValueClass -> domain: domain1 isProxy: False
MarhalledByReferenceClass -> domain: domain2 isProxy: True
```

## IDisposable interface

The `IDisposable` interface, when implemented in any class, informs CLR that such an object will handle some external resource or unmanaged handle. The best behavior here is to free up or disconnect from such costly resource as soon as possible, although the object collection will occur later with GC logics.

Once such an interface is implemented, the usage must tell CLR that cleanups of such resources must occur at a specific time, creating a local scope with usual parenthesis in C# or a specific `End` instruction in VB, by using the `using` keyword:

```

class Program
{
    static void Main(string[] args)
    {
        using (var instance = new ExternalResourceContainer())
        {
            } //here automatically CLR will invoke .Dispose method
        }
    }
public class ExternalResourceContainer : IDisposable
{
    private object externalResource;
    public void Dispose()
    {
        //release resource usage
    }
}

```

We may also invoke the `Dispose` method manually if we cannot use the `using` block.

## Threading

A thread is a virtual processor that can run some code from any AppDomain. Although at any single time a thread can run code from a single domain, it can cross domain boundaries when needed. Indeed, in the preceding example, there is only a single thread that did the entire job for all three AppDomains.

Without diving into the internals of Windows threading, we should know that a CLR thread is actually a Windows thread. This one has a high creation cost (even worse for 64-bit systems) as in any other virtualization technique, although in Windows, creating a process is even worse in terms of resource usage. This is why Microsoft has supported multi-threading programming in its operating systems since the age of Windows NT.

What is important to know is that a thread never has 100 percent of CPU's time because its CPU time is reassigned to any new pending-for-work threads every 30 milliseconds (a time-slice) by acting what we call in Windows a context switch.

This ensures that at the operating system level, no process can harm the system stability by locking every CPU forever and thus stopping critical OS tasks. This is why Windows is definitely a time-sharing operating system.

In the .NET world, a thread can be created by starting the `Run` method of the `Thread` class. Thus, the simple instantiation of the object does nothing more than instantiate any other class. A thread must always have an entry point: a starting method that can have an initialization parameter, usually referred to as state— that is actually anything within the .NET class hierarchy.

A `Priority` configuration is available and mapped to Windows' thread in order to alter the results in the context-switching search for new threads. Usually, priorities higher than normal are dangerous for system stability (in rare cases, letting the OS reach the *starvation* state that occurs when the highest-priority thread prevents context switching), while lower priorities are often used to process no CPU-time critical operations.

An `IsBackground` property is available to any `Thread` class instance. Setting this property to `True` will signal to the CLR that this is a non-blocking thread—a **background thread**—in that its execution does not keep a process in the running state. On the other hand, upon setting it to `False` (default



value), CLR will consider this thread as a **foreground thread**, in that its execution will keep the whole process in the running state.

Operations such as the animation of a clock may surely be made on a background thread, while the non-blocking UI operation of saving a huge file on a network resource is surely a candidate to run in a foreground thread, because although asynchronous against the UI, the foreground thread is also needed, and an eventual process premature exit should not kill such a thread. It is clear that CLR will automatically kill any background thread when a process ends without giving them any time to preserve any eventually needed data consistency.

Here's an example code on creating a background thread with low-priority CPU time:

```
static void Main(string[] args)
{
    //thread creation
    var t1 = new Thread(OtherThreadStartHere);
    //set thread priority at starting
    t1.Priority = ThreadPriority.Lowest;
    //set thread as background
    t1.IsBackground = true;
    //thread start will cause CLR asks a Thread to Windows
    t1.Start();
    //lock current executing thread up to the end of the t1 thread
    t1.Join();
}
private static void OtherThreadStartHere()
{
    //eventually change priority from innerside
    Thread.CurrentThread.Priority = ThreadPriority.Normal;
    //do something
}
```

The `Thread` class has specific methods to configure (as said) or handle thread lifetime, such as `Start`, to create a new OS thread and `Join` to kill a OS thread, and get back the remote thread status on the caller thread, such as any available exception.

Other methods are available, such as `Suspend`, `Resume` (both deprecated), and `Abort` (still not deprecated). It is easy to imagine that by invoking the `Suspend` or `Resume` method, the CLR will pause the thread from running or resuming work. Instead, the `Abort` method will inject a `ThreadAbortException` event at the current execution point of the thread's inner code, acting as a thread stopper. Although this will actually stop the thread from working, it is easy to infer that it is not an elegant solution because it can easily produce an inconsistent data state.

To solve this issue, CLR gives us the `BeginCriticalRegion` method to signal the beginning of an unable-to-abort code block and an `EndCriticalRegion` method to end such a code portion. Such methods will prevent any `ThreadAbortException` event being raised in such a portion of the atomic code. Here's an example code:

```
static void Main(string[] args)
{
    //thread creation
    var t1 = new Thread(OtherThreadStartHere);
    //thread start will cause CLR asks a Thread to Windows
    t1.Start();
    //do something
}
```

```

        t1.Abort();
    }
    private static void OtherThreadStartHere()
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(100);
            //signal this is an atomic code region
            //an Abort will never break execution of this code portion
            Thread.BeginCriticalRegion();
            //atomic code
            //atomic code
            //atomic code
            //atomic code
            Thread.EndCriticalRegion();
        }
    }
}

```



When dealing with iterated functions within a thread, instead of using `Abort` and `Critical` sections to gently signal the thread to exit, simply use a field as a flag (something like `canContinue`) to check within the iterated function, such as `while (canContinue)`. This choice will behave in a similar way to the previous example, without having to raise a useless exception.

Other interesting methods of the `Thread` class are `Sleep` (accepts a millisecond parameter) and `Yield`. The `Sleep` method suspends the thread for the given time; alternately, when `0` is used as a parameter, it signals a context switch to change the state to suspended, eventually causing higher-priority threads to use the thread time-slice as soon as possible. A better choice—when you want to recycle some of the time-slice time if a thread actually ended its job prematurely—is to use the `Yield` method that will give the remaining time-slice the next queued thread as soon as possible, waiting for the CPU time of the same processor. Here is an example code:

```

private static void OtherThreadStartHere()
{
    //change state to suspended and wait 1000 ms
    Thread.Sleep(1000);
    //change state to suspended
    Thread.Sleep(0);
    //give remaining time-slice to the next queued thread of current CPU
    Thread.Yield();
}

```

If we are in search of an alternative to create a thread from scratch with the `Thread` class, we could use an already created-thread preserved in CLR for any unimportant jobs that we can usually make in a background thread. These threads are contained in a collection named as `ThreadPool`. Many other CLR classes use threads from the `ThreadPool` collection, so if a lot of jobs are going to be queued in it, remember to increase the minimum and maximum pool size:

```

static void Main(string[] args)
{
    //set minimum thread pool size
    ThreadPool.SetMinThreads(32, 32);
    //set maximum thread pool size
    ThreadPool.SetMaxThreads(512, 512);
    //start a background operation within a thread from threadpool
}

```

```

//as soon as when a thread will became available
    ThreadPool.QueueUserWorkItem(ExecuteInBackgroundThread);
}
private static void ExecuteInBackgroundThread(object state)
{
//do something
}

```

## Multithreading synchronization

When dealing with multiple threads, data access in fields and properties must be synchronized, otherwise inconsistent data states may occur. Although CLR guarantees low-level data consistency by always performing a read/write operation, such as an atomic operation against any field or variable, when multiple threads use multiple variables, it may happen that during the write operation of a thread, another thread could also write the same values, creating an inconsistent state of the whole application.

First, let's take care of field initialization when dealing with multithreading. Here is an interesting example:

```

// a static variable without any thread-access optimization
public static int simpleValue = 10;
// a static variable with a value per thread instead per the whole process
[ThreadStatic]
public static int staticValue = 10;
//a thread-instantiated value
public static ThreadLocal<int> threadLocalizedValue = new ThreadLocal<int>(()
=> 10);
static void Main(string[] args)
{
// let's start 10 threads
    for (int i = 0; i < 10; i++)
        new Thread(IncrementVolatileValue).Start();
        Console.ReadLine();
}
private static void IncrementVolatileValue(object state)
{
// let's increment the value of all variables
    staticValue += 1;
    simpleValue += 1;
    threadLocalizedValue.Value += 1;
    Console.WriteLine("Simple: {0}\tLocalized: {1}\tStatic: {2}",
simpleValue, threadLocalizedValue.Value, staticValue);
}

```

Here is the console output:

```

Simple: 18 Localized: 11 Static: 1
Simple: 19 Localized: 11 Static: 1
Simple: 18 Localized: 11 Static: 1
Simple: 18 Localized: 11 Static: 1
Simple: 19 Localized: 11 Static: 1
Simple: 18 Localized: 11 Static: 1
Simple: 19 Localized: 11 Static: 1
Simple: 19 Localized: 11 Static: 1
Simple: 19 Localized: 11 Static: 1
Simple: 20 Localized: 11 Static: 1

```

The preceding code example simply incremented three different integer variables by 1. The result shows how different setups of such variable visibility and thread availability will produce different values, although they should all be virtually equal.

The first value (`simpleValue`) is a simple static integer that when incremented by 1 in all ten threads creates some data inconsistency. The value should be 20 for all threads—in some threads, the read value is 18, in some other 19, and in only one other thread is 20. This shows how setting a static value in multithreading without any thread synchronization technique will easily produce inconsistent data.

The second value (the `staticValue`) is outputted in the middle of the example output. The usage of the `ThreadStaticAttribute` legacy breaks the field initialization and duplicates the value for each calling thread, actually creating 10 copies of such an integer. Indeed, all threads write the same value made by 10 *plus* 1.

The most decoupled value is obtained by the third value (`threadLocalizedValue`), shown at the right of the example output. This generic compliant class (`ThreadLocal<int>`) behaves as the `ThreadStaticAttribute` usage by multiplying the field per calling thread with the added benefit of initializing such values with an anonymous function at each thread startup.



C# gives us the `volatile` keyword that signals to JIT that the field access must not be optimized at all. This means no CPU register caching, causing all threads to read/write the same value available in the main memory. Although this may seem to be a sort of magic synchronization technique, it is not; it does not work at all. Accessing a field in a volatile manner is a complex old-style design that actually does not have reason to be used within CLR-powered languages.

For more information, please read this article by Eric Lippert, the Chief Programmer of the C# compiler team in Microsoft, at <http://blogs.msdn.com/b/ericlippert/archive/2011/06/16/atomicity-volatility-and-immutability-are-different-part-three.aspx>.

More than the standard atomic operation given by CLR to any field, only for primitive types (often limited to `int` and `long`), CLR also offers a memory fence, such as field access utility named **Interlocked**. This can make low-level memory-fenced operations such as increment, decrement, and exchange value. All those operations are thread-safe to avoid data inconsistency without using locks or signals. Here is an example:

```
//increment of 1
Interlocked.Increment(ref value);
//decrement of 1
Interlocked.Decrement(ref value);
//increment of given value
Interlocked.Add(ref value, 4);
//substitute with given value
Interlocked.Exchange(ref value, 14);
```

## Locks

Different synchronization techniques and lock objects exist within CLR and outside of Windows itself. A lock is a kind of flag that stops the execution of a thread until another one releases the contended resources. All locks and other synchronization helpers will prevent threads from working on bad data, while adding some overhead.

In .NET, multiple classes are available to handle locks. The easiest is the `Monitor` class, which is also usable with the built-in keyword `lock` (`SyncLock` in VB). The **Monitor lock** allows you to lock access to a portion of code. Here is an example:

```
private static readonly object flag = new object();
private static void MultiThreadWork() {
    //serialize access to this portion of code
    //using the keyword
    lock (flag) {
        //do something with any thread un-safe resource
    }
    //this code actually does the same of the lock block above
    try{
        //take exclusive access
        Monitor.Enter(flag);
        //do something with any thread un-safe resource
    }
    finally{
        //release exclusive access
        Monitor.Exit(flag);
    }
}
```

## Signaling locks

All those locks that inherit the `WaitHandle` class are signaling locks. Instead of locking the execution code, they send messages to acknowledge that a resource has become available. They are all based on a Window kernel handle, the `SafeWaitHandle`, this is different from the `Monitor` class that works in user mode because it is made entirely in managed code from CLR. Such low-level heritage in the `WaitHandle` class hierarchy adds the ability to cross AppDomains by reference, inheriting from the `MarshalByRefObject` class.

More powerful than the `Monitor` class, the `Mutex` class inherits all features from the `Monitor` class, adding some interesting features, such as the ability to synchronize different processes working at the operating-system level. This is useful when dealing with multi-application synchronization needs.

Following is a code example of the `Mutex` class usage. We will create a simple console application that will await an operating-system level synchronization lock with the global name of `MUTEX_001`.

Please start multiple instances of the following application to test it out:

```
static void Main(string[] args) {
    Mutex mutex;
    try{
        //try using the global mutex if already created
        mutex = Mutex.OpenExisting("MUTEX_001");
    }
    catch (WaitHandleCannotBeOpenedException) {
        //creates a new (not owned) mutex
        mutex = new Mutex(false, "MUTEX_001");
    }
    Console.WriteLine("Waiting mutex...");
    //max 10 second timeout to acquire lock
    mutex.WaitOne();
    try{
        //you code here
    }
}
```

```

        Console.WriteLine("RETURN TO RELEASE");
        Console.ReadLine();
    }
    finally{
        mutex.ReleaseMutex();
        Console.WriteLine("Mutex released!");
    }
    mutex.Dispose();
}

```

Like the `Monitor` class, the `Semaphore` class enables us to lock a specific code portion access. The unique (and great) difference is that instead of allowing a single thread to execute such a code-block, the `Semaphore` class allows multiple threads all together. This class is a type of a limiter for limiting the resource usage.

In the following code example, we will see the `Semaphore` class is configured to allow up to four threads to execute all together – other threads will be queued until some allowed thread ends its job:

```

class Program{
    static void Main(string[] args){
        for (int i = 0; i < 100; i++){
            new Thread(AnotherThreadWork).Start();
            Console.WriteLine("RETURN TO END");
            Console.ReadLine();
        }
        //4 concurrent threads max
        private static readonly Semaphore waiter = new Semaphore(4, 4);
        private static void AnotherThreadWork(object obj){
            waiter.WaitOne();
            Thread.Sleep(1000);
            Console.WriteLine("{0}->Processed", Thread.CurrentThread.ManagedThreadId);
            waiter.Release();
        }
    }
}

```

Other widely used signaling lock classes are `ManualResetEvent` and the `AutoResetEvent` class. The two implementations simply differ in terms of the manual or automatic switch of the signal state to a new value and back to the initial value.

The usage of those two classes is completely different when compared to all classes seen before, because instead of giving us the ability to serialize thread access of a code-block, these two classes act as flags giving the signal everywhere in our application to indicate whether or not something has happened.

For instance, we can use the `AutoResetEvent` class to signal that we are doing something and let multiple threads wait for the same event. Later, once signaled, all such threads could proceed in processing without serializing the thread execution, for instance, when we use locks instead, like all others seen earlier, such as the `Monitor`, `Mutex`, or `Semaphore` classes.

Here is a code example showing two threads, each signaling its completion by the manual or the automatic wait handle, during which the main code will await the thread's completion before reaching the end:

```

static void Main(string[] args){
    new Thread(ManualSignalCompletion).Start();
    new Thread(AutoSignalCompletion).Start();
    //wait until the threads complete their job
    Console.WriteLine("Waiting manual one");
    //this method asks for the signal state, can repeat this row infinite times
    manualSignal.WaitOne();
    Console.WriteLine("Waiting auto one");
    //this method asks for the signal state and also reset the value back to
    //un-sigaled state, waiting again that some other code will signal the // //
    //completion. if I repeat this row, the program will simply wait forever
    autoSignal.WaitOne();
    Console.WriteLine("RETURN TO END");
    Console.ReadLine();
}
private static readonly ManualResetEvent manualSignal =
    new ManualResetEvent(false);
private static void ManualSignalCompletion(object obj){
    Thread.Sleep(2000);
    manualSignal.Set();
}
private static readonly AutoResetEvent autoSignal =
    new AutoResetEvent(false);
private static void AutoSignalCompletion(object obj){
    Thread.Sleep(5000);
    autoSignal.Set();
}
}

```

In this case, all such functionalities are overshoot by the `Task` class and the **Task Parallel Library (TPL)**.

Moreover, in .NET 4.0 or later, the `Semaphore` and the `ManualResetEvent` classes have alternatives in new classes that try to keep the behavior of the two previous ones by using a lighter approach. They are called `ManualResetEventSlim` and `SemaphoreSlim`.

Such new slim classes tend to limit access to the kernel mode handle by implementing the same logic in a managed way until possible (usually when a little time passes between signaling). This helps to execute faster than the legacy brothers do. Obviously, those objects lose the ability to cross boundaries of app domains or processes, as the `WaitHandle` hierarchy usually does. The usage of those new classes is identical to previous ones, but with some simple method renaming.

New classes are available in .NET 4 or greater: `CountdownEvent` and `Barrier`. Similar to the two slim classes we just saw, these classes do not derive from the `WaitHandle` hierarchy.

The `Barrier` class, as the name implies, lets you program a software barrier. A barrier is like a safe point that multiple tasks will use as parking until a single external event is signaled. Once this happens, all threads will proceed together.

Although the `Task` class offers better features in terms of continuation, in terms of more flexibility, the `Barrier` class gives us the ability to use such logic everywhere with any handmade thread. On the other hand, the `Task` class is great in continuation and synchronization of other `Task` objects. Here is an example involving the `Barrier` class:

```

private static readonly Barrier completionBarrier =
    new Barrier(4, OnBarrierReached);
static void Main(string[] args) {
    new Thread(DoSomethingAndSignalBarrier).Start(1000);
    new Thread(DoSomethingAndSignalBarrier).Start(2000);
    new Thread(DoSomethingAndSignalBarrier).Start(3000);
    new Thread(DoSomethingAndSignalBarrier).Start(4000);
    Console.ReadLine();
}
private static void DoSomethingAndSignalBarrier(object obj) {
    //do something
    Thread.Sleep((int)obj); //the timeout flowed as state object
    Console.WriteLine("{0:T} Waiting barrier...", DateTime.Now);
    //wait for other threads to proceed all together
    completionBarrier.SignalAndWait();
    Console.WriteLine("{0:T} Completed", DateTime.Now);
}
private static void OnBarrierReached(Barrier obj) {
    Console.WriteLine("Barrier reached successfully!");
}

```

The following is the console output:

```

17:45:41 Waiting barrier...
17:45:42 Waiting barrier...
17:45:43 Waiting barrier...
17:45:44 Waiting barrier...
Barrier reached successfully!
17:45:44 Completed
17:45:44 Completed
17:45:44 Completed
17:45:44 Completed

```

Similar to the `Barrier` class, the `CountdownEvent` class creates a backward timer to collect multiple activities and apply some continuation at the end:

```

private static readonly CountdownEvent counter = new CountdownEvent(100);
static void Main(string[] args) {
    new Thread(RepeatSomething100Times).Start();
    //wait for counter being zero
    counter.Wait();
    Console.WriteLine("RETURN TO END");
    Console.ReadLine();
}
private static void RepeatSomething100Times(object obj) {
    for (int i = 0; i < 100; i++) {
        counter.Signal();
        Thread.Sleep(100);
    }
}

```



## Drawbacks of locks

Use lock techniques carefully. Always try to avoid any **race condition** that happens when multiple different threads are fighting each other in trying to access the same resource. This produces an inconsistent state and/or causes high resource usage too. When a race condition happens in the worst possible manner, there will be **starvation** for resources.

Starvation happens when a thread never gets access to CPU time because different threads of higher priority take all the time, sometimes also causing an operating system fault if a thread in a loop-state is unable to abort its execution when running at highest priority level (the same of the OS core threads). You can find more details on resource starvation at [http://en.wikipedia.org/wiki/Resource\\_starvation](http://en.wikipedia.org/wiki/Resource_starvation).

With the wrong locking design, an application may fall in the **deadlock** state. Such a state occurs when multiple threads wait forever, each with the other, for the same resource or multiple resources without being able to exit this multiple lock state. Deadlock often happens in wrong relational database designs or due to the wrong usage of relational database inner lock techniques. More details on the deadlock state can be found at <http://en.wikipedia.org/wiki/Deadlock>.

Instead, with managed synchronization techniques such as spin-wait based algorithms (like the one within `SemaphoreSlim` class), an infinite loop can occur, wasting CPU time forever and bringing the application into a state called **livelock**, which causes the process to crash for the stack-overflow condition, at a time. For more details on livelock, visit <http://en.wikipedia.org/wiki/Deadlock#Livelock>.

## Exception handling

Exception handling is the black art of doing something to repair an unpredicted error or malfunction. Within CLR, anytime something happens outside our prevision, such as setting an `Int16` typed variable with a value outside valid ranges, the CLR will handle such an event by itself, creating an instance of an `Exception` class and breaking the execution of our code, trying instead to find some other code able to handle (a.k.a `catch`) such an exception.

Any `Exception` class is populated with all useful details regarding what just happened, like a simplified error text (within the `Message` property), the `StackTrace` that explains exactly the whole method call hierarchy, and other details. Often, instead of a simple `Exception` class, an inheritance child is instantiated to collect specific additional details or simply to define the kind of exception just raised. Indeed, setting an outranged value within an `Int16` typed variable will raise an `OverflowException` event in place of a simple `Exception` event.

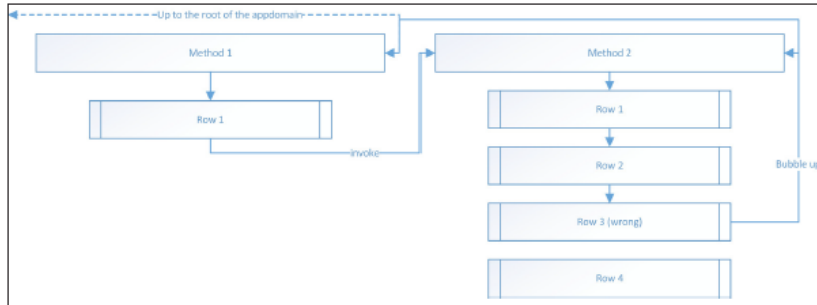
As just said, an exception is usually handled within a .NET-based application in response to an unpredicted error, although an exception is actually a special `GoTo` statement that will alter the control-flow of our application. This is the why its called as an *exception* instead of an *error*.

Anytime an error happens, or simply when the flow cannot proceed as normal, an exception is created and raised (raising an exception actually starts the control-flow alteration) to avoid completing a method run, maybe, because its data is inconsistent. We can also create our exceptions regarding our business, components, or helpers/frameworks if special parameters are needed to flow.

Carefully create and handle exceptions because of the cost the CLR incurs when any exception is raised. Although an exception will start a control-flow alteration, at the beginning, CLR will compute the full call stack of the executing thread. This is a CPU-intensive operation that actually stops the thread from running any other code. This is proof that Microsoft considers the entire exception-

handling framework as error management. Therefore, it is impossible to create a multiple control-flow application using exception handling without enabling great wastage of resources. This means that, if we still need to be creating multiple control-flow applications, we will still need to use the `goto` keyword.

Here is a graphical representation that shows the control-alteration made by any exception:



A flow diagram of an exception with its control-flow in search of a continuation

As seen in the preceding diagram, when the control-flow changes, the CLR searches for some `catch` code-block. This may be locally, or at any calling level, up to the program's `Main` method. Here is a classical implementation in C#:

```
int a = 0;
try{
//normal control-flow
    a = 10;
    a = a / int.Parse("0"); //this will raise a DivideByZeroException
}
catch (DivideByZeroException dx){
//altered control-flow if CLR raises DivideByZeroException
    Console.WriteLine(dx.Message);
}
catch (Exception ex){
//altered control-flow if CLR raises any other exception
    Console.WriteLine(ex.Message);
}
finally{
//usually used for cleanup resources
//or restore data state
    a = 0;
}
```

The CLR executes the code within the `try` block; then, when an exception is raised, CLR searches the best-fit altered control-flow by matching handled exceptions (with the `catch` keyword) with the exception raised. The search is from top to bottom and supports class hierarchy. This means that the less specific exception (the one handled with the generic `Exception` class) must be always the last one. Otherwise, other exceptions will never be matched.

In the preceding example, there is a `catch` block for such an exact exception raised, so the flow will continue in that block. After any `try` or `catch` block, the `finally` block is invoked, if present (optional).

A `try-catch` block can be nested in others if needed, although this may lead to a control flow that is tricky to understand with all those alterations. By nesting exceptions, CLR still goes in search of a `catch` block from the deeper code row where the exception has originated, flowing up to the process'

`Main` method. Such exceptions lift, like an air bubble in the water, it lets programmers work with exception-handling search as made by CLR from the deeper code to the most outer one (the `Main` method), the name of *exception bubbling*.

Raising new exceptions is actually simple; it is enough to use the `throw` keyword and pass the new exception:

```
throw new Exception("HI");
```

Any time an exception is raised somewhere, the related `AppPool` is notified of the `FirstChanceException` event that actually runs before the bubbling occurs—in other words, at the start of the bubbling.

During the bubbling, if the CLR cannot find any valid `catch` block, the related `AppDomain` (the one where the exception originated) is notified on the `UnhandledException` event. Although this cannot handle the exception as a super `catch` block can, it can somehow notify application users or the system administrator gracefully before the critical exit of the process. Here is an such an example:

```
static void CurrentDomain_UnhandledException(object sender,
UnhandledExceptionEventArgs e){
//contains exception details
    var ex = e.ExceptionObject;
//if true the process will terminate
    var willKillCLR = e.IsTerminating;
}
static void CurrentDomain_FirstChanceException(object sender,
FirstChanceExceptionEventArgs e){
//contains exception details
    var ex = e.Exception;
}
```

After the `UnhandledException` event occurs, the `AppDomain` class is unloaded by CLR.

A special case occurs when another compiler raises a non **Common Language Specification (CLS)** compatible exception (such as raising a `string` exception or `int` exception—classes that do not inherit from the `Exception` class). Although this is a rare opportunity, some external vendor language implementation could work with such behavior. In this case, the CLR will raise a `RuntimeWrappedException` event with the ability to read raw exception data, such as an `int` value or a `string` value, as an internal exception.

Another special case to be aware of is that a `finally` block—although it is usually called, anything that can happen within a `catch` block—cannot run if the executing thread is killed by unmanaged code, by invoking the `Win32 KillThread` or `KillProcess` method.

Obviously, in such cases, Windows will also kill the whole process with anything within. The only leak that will still survive will occur when you launch an external process driven by your application that somehow was killed by Windows. In this case, the `finally` block is avoided and the external process can remain in memory. A widely-used solution is to always check if zombie external processes are still alive from previous executions of the application, when we start a new instance of such an application.

## Summary

In this chapter, we looked into CLR internals with regard to compilation and memory management, the two most important abstractions that CLR offers. Thread management, synchronization, and event handling were discussed to give the developer the ability to interact with all specific tools and techniques CLR offers regarding the tricky aspects of programming.

In the next chapter, asynchronous programming techniques such as task creation, maintenance, executing, and tuning will be analyzed in depth.



Further reading:

Russinovich, Mark. *Windows Internals*, 6th edition, Microsoft Press, 2012

Richter, Jeffrey. *CLR via C#*, 4th edition, Microsoft Press, 2013

## Asynchronous Programming

This chapter will dive into the asynchronous elaboration techniques available within the .NET framework.

Here we will explain the following features, techniques, and frameworks:

- Asynchronous programming theory
- Asynchronous Programming Model (APM)
- Event-based Asynchronous Pattern (EAP)
- Task-based Asynchronous Pattern (TAP)
- `Async/await` operator
- Task optimization and CLR tuning
- Task tweaking
- Task UI synchronization

### Understanding asynchronous programming

Multi-threaded programming happens when we use multiple threads to execute our code. The added benefit is the increased CPU power available by using multiple threads.

Asynchronous programming happens anytime we move the execution of any our code from the main thread to another one and then back to the first one to catch any result or acknowledgement. Thus, the difference between multi-threaded and asynchronous programming is that the catching of the result happens within the asynchronous one. Otherwise, it's called multi-threaded programming. For instance, a background thread providing some data in a polling way is simply another multi-threaded one.

### The concept of asynchronous programming theory.

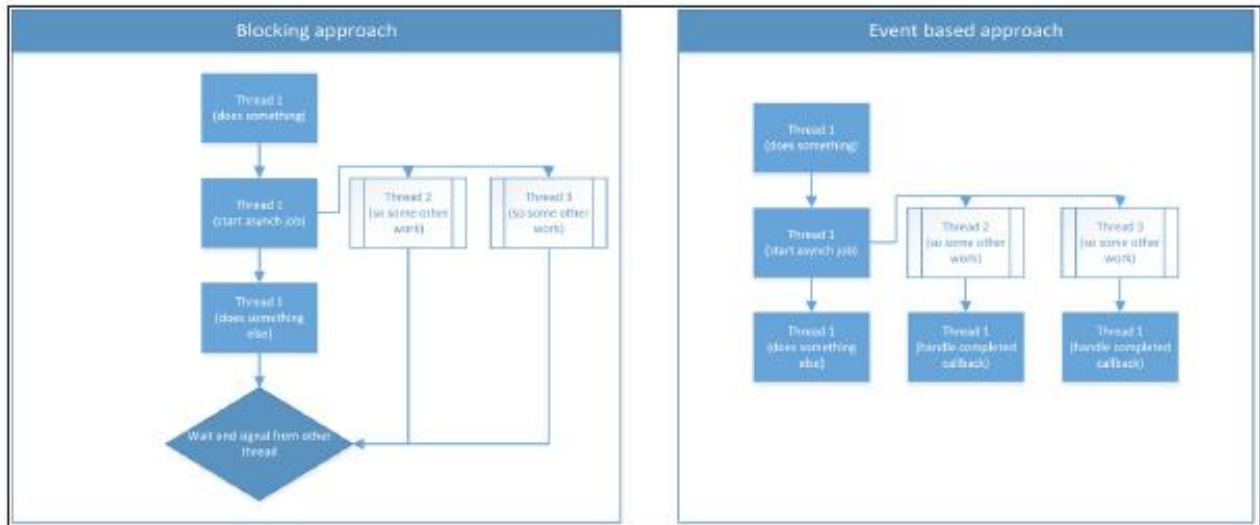
In multi-threaded programming, we create multiple virtual processors (threads) able to execute our code for *long-time* operations and without the need to participate in the same job. This means that different threads may do different things. In multi-threading, any thread does its job while trying to avoid any resource sharing with other threads, because of the cost of lock synchronization this sharing will imply, as seen in the *Multithreading synchronization* section in *Chapter CLR Internals*. It is like executing multiple applications in the same process or dividing macro features of the same application across available cores.

In asynchronous programming, we create multiple threads to execute a *single short-timed* job (usually involving different external systems) that must end (or continue) all together.

A chat application is an example of a multi-threaded program, an application that consists of two threads. A thread for read data from other participants, and another to write data back to the participants. The two threads have different goals, although they can sometimes exchange or share data. Those threads have a long life and behave as two different applications each integrating with the other only when needed.

An asynchronous programmed example application, instead, creates four threads to save data in four different CSV streams, later compressed into a single ZIP file. What makes such an example perfect for asynchronous programming is the short life of each single thread, the unified software barrier where all threads wait for each other to produce a single ZIP file, and the completely cohesive thread behavior.

Two main asynchronous programming designs are available to developers. A blocking one, which happens when the calling thread waits for all asynchronous threads to proceed all together, or an event signaling based one, where each asynchronous thread acknowledges the main thread by invoking CLR events.



Asynchronous programming approaches – blocking vs. event signaling

As visible in the preceding figure, asynchronous execution may happen in a blocking way with multiple threads that are waited for by the main thread, with signalers such as the `WaitHandle` hierarchy, as seen in the *Multithreading synchronization* section *Chapter CLR Internals*.

Obviously, the .NET framework's observer pattern implementation made with delegates and events is usable and thus, an asynchronous callback handler may be invoked in an operation-starting instance to complete the whole job. If desired, another signaling lock may be used here to continue all together in the blocking way, but again, on another thread.

Before .NET 4, Microsoft allowed asynchronous programming with two main different techniques, one for desktop class applications and another more generic one. Although when programming for .NET 4.5.x, the new frameworks do exist, a lot of SDKs, from Microsoft and other vendors, still support the legacy pattern. Thus, a good knowledge of those techniques is still needed for any programmer who wants to be compliant with all asynchronous designs from Microsoft and also wants to understand the architectural concerns that lie behind the mere technical skill.

Let's look at them in detail.

## Asynchronous Programming Model (APM)

The **Asynchronous Programming Model (APM)** is one of the oldest patterns introduced by Microsoft in .NET 1.0 back in 2001 for asynchronous programming handling.

The pattern is easy. To start a deferred job, you simply start such a job by using a **Delegate** (remote method invoker) and then get an object back of type `IAsyncResult` to know the status of such a remote operation. Here an asynchronous programmed application to compute file hashes. The application will add a "." to the starting data computation initial message to acknowledge to the user that the application is still processing. The following examples use the blocking approach:

```
static void Main(string[] args){
    //a container for data
    var complexData = new byte[1024];
    //a delegate object that gives us the ability to trigger the pointed method
    in async way
    var dataDelegate = new Action<byte[]>(ComputeComplexData);
    Console.WriteLine("Starting data computation...");
    //start retrieving complex data in another thread
    IAsyncResult dataStatus=dataDelegate.BeginInvoke(complexData, null, null);
    //waiting the completion
    while (!dataStatus.IsCompleted) {
        Console.WriteLine(".");
        Thread.Sleep(100);
    }
    Console.WriteLine(" OK");
    //instantiate a delegate for hash elaboration in async way
    var hashDelegate = new Func<byte[], string>(ComputeHash64);
    Console.WriteLine("Starting hash computation...");
    IAsyncResult hashStatus=hashDelegate.BeginInvoke(complexData, null, null);
    //waiting the completion
    while (!hashStatus.IsCompleted) {
        Console.WriteLine(".");
        Thread.Sleep(100);
    }
    //this time the async operation returns a value
    //we need to use the delegate again to catch this value from the other thread
    var hash = hashDelegate.EndInvoke(hashStatus);
    Console.WriteLine(" OK");
    Console.WriteLine("END");
    Console.ReadLine();
}

static void ComputeComplexData(byte[] data){
    var r = new Random();
    Thread.Sleep(3000);
    r.NextBytes(data);
}

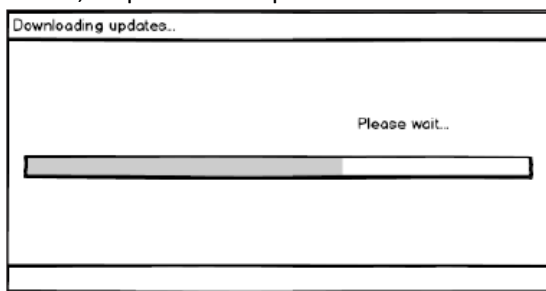
public static string ComputeHash64(byte[] data){
    using (var engine =
        new System.Security.Cryptography.MD5CryptoServiceProvider()){
        Thread.Sleep(3000);
        var hash = engine.ComputeHash(data);
        return Convert.ToBase64String(hash);
    }
}
```

The code is very easy. The class that helps make things asynchronous here is the `Delegate` class. Here, we use the pre-generated versions, `Action<T>` and `Func<T>`, compliant with the generic pattern, which helps us use any feature required of such `Delegate` objects without having to declare a specific one each time.



A `Delegate` object is an object-oriented method pointer with a lot of added features such as asynchronous support and multiple method handlers. Any CLR event is a `Delegate` too. Such a class gives us the ability to invoke any remote method in a synchronous way (with the usual `.Invoke` method), or in an asynchronous way with `BeginInvoke/EndInvoke`, as visible in the preceding example. As mentioned earlier, the `IsCompleted` property gives us feedback about the remote execution completion of all pointed remote methods.

The usage of such a blocking asynchronous operation, without the need to block the execution of the main thread, helps create respondent UXs as a download popup, or special import/export features.



A simple asynchronous download popup

There are a lot of SDKs that give us the ability to use APM patterns like in the preceding example. In .NET 4.0, many core-framework APM implementations have been updated to the new TAP framework (discussed later in this chapter in the *Task-based Asynchronous Pattern (TAP)* section). Here's another APM example showing network communication with the `IDisposable/using` pattern:

```
static void Main(string[] args){
//running in .NET 4.0
    var url = "http://www.google.com";
    Console.WriteLine("Asking for {0}", url);
//create a new web request for given url
    var request = WebRequest.Create(url);
//start collecting response in async way

    var responseStatus = request.BeginGetResponse(null, null);
//waiting the completion
    while (!responseStatus.IsCompleted){
        Console.WriteLine(".");
        Thread.Sleep(100);
    }
    Console.WriteLine(" OK");
//a size counter
    int size = 0;
//catch back on the main thread the response
    using (var response = request.EndGetResponse(responseStatus))
//open the stream to access the response data
    using (var stream = response.GetResponseStream())
//open a reader for such data
    using (var r = new StreamReader(stream, Encoding.UTF8))
//until data is valid
```



```

while (!r.EndOfStream && r.Read() >= 0) size++;
Console.WriteLine("Total size: {0:N1}KB", size / 1024D);
Console.WriteLine("END");
Console.ReadLine();
}

```

In the preceding second code example, we found the ability to use the APM with .NET assemblies. The usage is very straightforward. As visible in the two examples given, the `IAsyncResult` type gives us the ability to wait for completion in a polling way by repetitively checking the `IsCompleted` property value. The additional ability to wait for such completion with a `WaitHandle` class is interesting, as already seen in the *Multithreading synchronization* section in *Chapter CLR Internals*. Here's an example:

```

//alternative 1:
//waiting the completion
while (!status.IsCompleted)
    Thread.Sleep(100);
//alternative 2:
//waiting the completion with the signaling event lock
status.AsyncWaitHandle.WaitOne();

```

Although the `WaitHandle` class based alternative may seem to be more comfortable than the one that uses the polling property check, the difference is that the polling one also gives us the ability to update the UI while waiting. Instead, the `WaitHandle` class will simply stop the execution of the thread where such an object is being waited on. Bear in mind that multiple threads can wait together at the same time as the `IAsyncResult` completion status or wait handle, without any issues (until this brings some other resources to the race condition).

In addition to the blocking wait, as said at the beginning of the paragraph, we have the ability to catch the completion of an asynchronous method execution, to another asynchronous method by passing a `Delegate` object, which represents a callback method. If multiple callbacks are caught in the same handler, a state parameter can help in its handling. Using a callback method gives us a more event-based approach. Consider that the callback executes in the same thread as the asynchronous processing. Here is an example with a single callback method:

```

static void Main(string[] args){
    var invoker = new Func<int>(OnCreateInteger);
    //trigger 3 invocations sending the same invoker as the state to the
    // ending handler
    invoker.BeginInvoke(OnHandleInteger, invoker); Thread.Sleep(100);
    invoker.BeginInvoke(OnHandleInteger, invoker); Thread.Sleep(100);
    invoker.BeginInvoke(OnHandleInteger, invoker);
    Console.WriteLine("MAIN THREAD ENDED");
    Console.ReadLine();
}
private static void OnHandleInteger(IAsyncResult ar){
    //the state contains the sent invoker var invoker = (Func<int>)ar.AsyncState;
    Console.WriteLine("Async operation returned {0}", invoker.EndInvoke(ar));
}
private static int OnCreateInteger() {
    Thread.Sleep(3000);
    return DateTime.Now.Millisecond; //returns a random integer
}

```

## Event-based Asynchronous Pattern (EAP)

The **Event-based Asynchronous Pattern (EAP)** is a specific design pattern to standardize event-based asynchronous programming features. Such a design is available in multiple classes from .NET itself and is available and suggested in all our implementations if applicable.

Unlike the previously seen APM, in this pattern, any method that supports synchronous execution will add an overloaded method for an asynchronous invocation. The result, instead, will be available only to a specific predefined callback method, one for each available method, within the class itself.

Here is an example showing the `WebClient` class downloading some web page data:

```
static void Main(string[] args){
//a simple web client
    var client = new WebClient();
//register for result callback
    client.DownloadDataCompleted += client_DownloadDataCompleted;
//invoke asynchronous request
    client.DownloadDataAsync(new Uri("http://www.google.com"));
    Console.WriteLine("MAIN THREAD ENDED");
    Console.ReadLine();
}
static void client_DownloadDataCompleted(object sender,
DownloadDataCompletedEventArgs e){
//this callback receives the whole response (data and status)
//data
    byte[] downloadDataResult = e.Result;
//eventual exception
    Exception ex = e.Error;
    Console.WriteLine("Downloaded {0:N1}KB", downloadDataResult.Length /
1024d);
}
```

The same features are available in any of our classes by implementing the same pattern. Here is an example:

```
class Program{
    static void Main(string[] args){
        var instance = new SimpleAsyncClass();
        Console.WriteLine("Sync value: {0}", instance.ProcessSomething());
//register an event handler to catch the result
        instance.ProcessSomethingCompleted +=
instance_ProcessSomethingCompleted;
//invoke async invoke
        instance.ProcessSomethingAsync();
        Console.WriteLine("MAIN THREAD ENDED");
        Console.ReadLine();
    }
    static void instance_ProcessSomethingCompleted(object sender, int e){
        Console.WriteLine("Async value: {0}", e);
    }
}
public class SimpleAsyncClass{
    public int ProcessSomething(){
        Thread.Sleep(3000);
//returns a random integer
        return DateTime.Now.Millisecond;
    }
}
```

```

    }
    public void ProcessSomethingAsync() {
//initialize a delegate object to make async call
        var invoker = new Func<int>(ProcessSomething);
//start async elaboration with callback
        invoker.BeginInvoke(InnerProcessSomethingCompleted, invoker);
    }
    private void InnerProcessSomethingCompleted(IAsyncResult ar) {
//catch the delegate object from async state
        var invoker = (Func<int>)ar.AsyncState;
//raise the event with computed value
        if (ProcessSomethingCompleted != null)
            ProcessSomethingCompleted(this, invoker.EndInvoke(ar));
    }
//the event that is usable for intercepting the computed value
    public event EventHandler<int> ProcessSomethingCompleted;
}

```

This simple implementation gives us the ability to understand how EAP works. The pattern asks us to add some naming conventions such as the syntax `Async` at the end of the method or the syntax `Completed` for the acknowledgement event.

The pattern itself, in its pure version, is more verbose than how it was just seen. It also requires a specific `Delegate` declaration for each method (although all with the same sign), cancellation support, process state notification (the percentage or completion), and a busy indicator. It is at the discretion of the programmer whether to implement a pure pattern or a simplified one, as visible in the example just given.

The `BackgroundWorker` is a component that supports a full EAP with the ability to run in an asynchronous way and synchronize the UI access by itself (discussed later in this chapter in the *Task UI synchronization* section).

### Task-based Asynchronous Pattern (TAP)

The **Task-based Asynchronous Pattern (TAP)** is the newly provided asynchronous programming framework born in .NET 4. TAP provides features of APM and EAP with an added signaling lock like an API that offers a lot of interesting new features.

#### Task creation

In .NET, this asynchronous job takes the name of a task. A task is also a class of the `System.Threading.Tasks` namespace. Here is a basic example:

```

var task = Task.Run(() =>{
    Thread.Sleep(3000);
//returns a random integer
    return DateTime.Now.Millisecond;
});
Console.WriteLine("Starting data computation...");
//waiting the completion
while (task.Status != TaskStatus.RanToCompletion) {
    Console.WriteLine(".");
    Thread.Sleep(100);
}
Console.WriteLine(" OK");
Console.WriteLine("END");
Console.ReadLine();

```

The preceding example is similar to the first one shown about the APM. Although a lambda expression is used here to create an anonymous method implementation, it is the same as creating a named method like we did in the previous example with the `ProcessSomething` instance.

The `Task.Run` method starts the asynchronous execution of the remote method provided by the `Delegate` object (the lambda syntax actually creates a `Delegate` object referring to an un-named method). It immediately returns a `Task` object that is usable to the query execution status and eventually waits with any wait handle, shown as follows:

```
task.Wait();
```

The preceding lambda-based syntax works on .NET 4.5, while another syntax is available from .NET 4 with more configurations available:

```
var task = Task.Factory.StartNew<int>(OnAnotherThread);
```

Although the two methods are actually the same because the `Task.Run` method executes the `StartNew` method of the default `TaskFactory` class, by invoking the `StartNew` itself, we can also specify customized options regarding task creation and continuation. In addition, we can create multiple factories, one for each specific group of tasks of a homogenous configuration with less effort and improved manageability.

A special feature of the `TaskFactory` class is the ability to marshal the result from APM's `EndInvoke` method in a specific task with the `FromAsync` method. In such cases, multiple overloads of the same method offer different options such as sending a state parameter or not sending one.

Let's look at a complete example:

```
static void Main(string[] args){
//the usual delegate for APM
    var invoker = new Func<int>(OnAnotherThread);
//a task catching the EndInvoke in another asynchronous method
    var fromAsyncTask = Task.Factory.FromAsync<int>(invoker.BeginInvoke,
invoker.EndInvoke, null);
//this usage of the result will internally invoke the Wait method
//blocking the execution until a result will become available
    Console.WriteLine("From async 1: {0}", fromAsyncTask.Result);
//this second overload wants the whole IAsyncResult
    var status = invoker.BeginInvoke(null, null);
//this will catch the EndInvoke in a task
    var fromAsyncTask2 = Task.Factory.FromAsync<int>(status,
invoker.EndInvoke);
    Console.WriteLine("From async 2: {0}", fromAsyncTask2.Result);
    Console.ReadLine();
}
private static int OnAnotherThread(){
    Thread.Sleep(500);
    return DateTime.Now.Millisecond; //a random int
}
```


Actually, the initial section of the code is the best regarding short coding because the second one also needs an `IAsyncResult` interface with the need for another variable.

Visit the following MSDN link to learn more about the `FromAsync` method:

[http://msdn.microsoft.com/en-us/library/dd321469\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd321469(v=vs.110).aspx)

At the end of the page, we can see the following remark:

### Remarks

 **Tip**

The `FromAsync` overloads that take an `asyncResult` parameter are not as efficient as the overloads that take a `beginMethod` parameter. If performance is an issue, use the overloads that provide the `beginMethod/endMethod` pattern.

This remark warns us about using the overload that wants the whole `IAsyncResult`. Instead, it suggests using the one that needs the couple `Begin/End` statements.

Another useful option that is available when using the `TaskFactory` method is the ability to configure how tasks are created; the following code shows an example:

```
var task1 = Task.Factory.StartNew(() =>{
//classic task creation with factory defaults
    var task2 = Task.Factory.StartNew(() =>{
//task that startup asap
    }, TaskCreationOptions.PreferFairness);
    var task3 = Task.Factory.StartNew(() =>{
//task that will run for a long time
    }, TaskCreationOptions.LongRunning);
```

The `TaskCreationOptions` enum helps us select between different choices (members) for task startups. The most interesting one here is the last one, that is, the `LongRunning` member. Although this does not change the task startup time, its creation will occur on a special background thread, without consuming a classic thread from the `ThreadPool` class, where the `TaskFactory` class usually takes background threads from, for its tasks.

handling task execution with the best performance. For the default task scheduler, high throughput is the primary concern. This is why we need to use the `PreferFairness` creation option to specify a low-latency startup scenario.

Generically talking, this is a good option because from .NET 4 onwards, good optimization has been applied on the `ThreadPool` engine that actually handles a single global FIFO queue of pending user jobs available for the whole process without any local or global lock. In addition to this global queue, any nested task will run on another queue instead, a local queue for each application thread that is running in a LIFO way is optimized for fast execution, CPU cache access optimization, and data locality in CLR memory; the following shows an example:

```
var task = Task.Factory.StartNew(() =>{
//will enqueue on global AppPool queue
    var inner = Task.Factory.StartNew(()=>{
//will enqueue on local AppPool queue
    });
});
```

An important aspect of the `System.Threading.ThreadPool` class usage is that some default limitation on thread availability does exist. Such limitations can be set both at the default pool size (minimum size) and at the maximum size. These defaults are the logical processor count for the minimum pool size, while the maximum size is dynamically set by the CLR itself (in older .NET frameworks, it was statically set as a multiple of the CPU count).

The `ThreadPool` class exposes static methods to get and set the minimum (`GetMinThreads`) and maximum (`GetMaxThreads`) pool size. Together they expose the `GetAvailableThreads` method, which gives us the actual remaining thread count number that equals the maximum size once it is subtracted from the currently used thread count. Here is a code example:

```
int min, minIO, max, maxIO;
//retrieve min and max ThreadPool size
ThreadPool.GetMinThreads(out min, out minIO);
ThreadPool.GetMaxThreads(out max, out maxIO);
//retrieve actually available thread count
int remaining, remainingIO;
ThreadPool.GetAvailableThreads(out remaining, out remainingIO);
//set up a new ThreadPool configuration
ThreadPool.SetMinThreads(64, 64);
ThreadPool.SetMaxThreads(2048, 2048);
```

Please bear in mind that all requests against the `ThreadPool` class will remain queued until some threads become available. This is why, if we create infinite tasks, in a little time, we will exceed the minimum thread pool size, and we will receive an `OutOfMemoryException` message. This exception happens because of the unavailability of adding other user tasks to the pool queue. Another important thing to know about the `ThreadPool` thread lifecycle is that CLR preallocates enough threads as the specified minimum size. When we continue adding threads, until we reach the maximum size, the CLR will add threads to the thread pool, as we might expect. The difference is that thread increase happens in a very slow way, adding only one thread per second. Here is a straightforward example:

```
int c = 0;
while (true){
    Task.Factory.StartNew(() =>{
        Console.WriteLine(++c);
        Thread.Sleep(-1);
    }, TaskCreationOptions.PreferFairness);
}
```

This example will print the number of logical threads on your CPU to a console output in a short amount of time. Later, a new thread per second count will be available (and never released), giving us a raw representation of pool increase timings. Please use this example, because as you learned before, without ever releasing such threads, the pool queue will reach its limit quickly, causing an `OutOfMemoryException` error.

Let's look at a more complete example:

```
static void Main(string[] args){
//creates a listener for TCP inbound connections
    var listener = new TcpListener(IPAddress.Any, 8080);
//start it
    listener.Start();
//accept any client
    while (true){
```

```

//get a task for client connection
    var client = listener.AcceptTcpClientAsync();
//wait for client connection
    client.Wait();
//once the connection succeeded, it starts a new task
//for handling communication with this new client
    Task.Factory.StartNew(HandleClientConnection, client,
TaskCreationOptions.PreferFairness); //run again to accept new clients
    }
}
private static void HandleClientConnection(object arg){
    var client = (TcpClient)arg;
//do something
}

```

This example shows you how to use asynchronous programming efficiently to handle thousands of client connections on the same port. This code has virtually no limits on the client connection count (but the limit set by Windows itself is somewhere around 65,000 connections per port). Obviously, as already said before, although the code is able to accept a virtually infinite number of clients, only a small number of them will be available to run on our CPU, because of the `ThreadPool` timings in its size increase.

The same example made with an old APM instead, will stop accepting clients as soon as it reaches the `ThreadPool` default limitations:

```

static void Main(string[] args){
//creates a listener for TCP inbound connections
    var listener = new TcpListener(IPAddress.Any, 8080);
//start it
    listener.Start();
//accept any client
    while (true){
//start waiting for a client
        var status = listener.BeginAcceptTcpClient(null, null);
//wait for client connection
        status.AsyncWaitHandle.WaitOne();
//catch the asynchronously created client
        var client = listener.EndAcceptTcpClient(status);
//once the connection happened, it start a new thread pool job
//for handling communication with such new client
        ThreadPool.QueueUserWorkItem(HandleClientConnection, client);
//run again to accept new clients
    }
}
private static void HandleClientConnection(object arg){
    var client = (TcpClient)arg;
//IMPLEMENTATION OMITTED
}

```

## Task synchronization

Back to the `TaskFactory` class, going deeper with regards to nested tasks and their execution in graph synchronization, we have to differ between attached and detached tasks. Any task may attach itself to its parent task, if any, although this is not the default behavior. With the default behavior, child tasks are detached from their parent tasks. This means that the parent does not care about its child tasks, shown as follows:

```
var parent = Task.Factory.StartNew(() =>{
    var child = Task.Factory.StartNew(() =>{
        Thread.Sleep(3000);
        Console.WriteLine("child: ending");
    });
    Thread.Sleep(1000);
});

parent.Wait();
Console.WriteLine("parent: ended before waiting for its child");
Console.ReadLine();
```

Here is the result:

```
parent: ended before waiting for its child
child: ending
```

The `TaskFactory` class lets us specify that a child task must attach to the parent one by passing the optional parameter `TaskCreationOptions.AttachedToParent`. In such a case, the parent will care about its child's status and exceptions by waiting for their completion times, shown as follows:

```
var parent = Task.Factory.StartNew(() =>{
    var child = Task.Factory.StartNew(() =>{
        Thread.Sleep(3000);
        Console.WriteLine("child: ending");
    }, TaskCreationOptions.AttachedToParent);
    Thread.Sleep(1000);
});

parent.Wait();
Console.WriteLine("parent: ended after waiting for its child");
Console.ReadLine();
```

As seen in the preceding code, such little differences in code produce a big difference in the result. In two words: *the opposite*:

```
child: ending
parent: ended after waiting for its child.
```

Always use the `Task.Factory.StartNew` method when dealing with child tasks because the `Task.Run` method prevents the child from attaching itself to the parent. The `Task.Run` method is only a shortcut for task creation with the default setup, while the `Task.Factory.StartNew` method gives us the ability to configure our task initialization options.

Such synchronization has its costs. Therefore, although not really useful, please use multiple outer tasks with the required synchronization techniques, such as waiting for the right number of tasks.

Regarding task synchronization, it is imperative that you understand the difference between all available waiting tasks. Waiting for a task is like waiting for a signaling lock, as already seen in the



*Multithreading synchronization* section in *Chapter CLR Internals*. The `Task` class gives us methods such as `Wait`, `WaitAll`, or `WaitAny` to accomplish jobs as shown in the following example:

```
//Make a Task wait forever
task1.Wait();
//wait for a task to timeout
if (task1.Wait(1000)) { //ms
//on time
}
else{
//timeout
}
if (task1.Wait(TimeSpan.FromMinutes(1))) { }
else { }
//Make tasks wait tasks forever, or timeout
Task.WaitAll(task1, task2, task3);
if (Task.WaitAll(new[] { task1, task2, task3 }, 1000)) { }
if (Task.WaitAll(new[] { task1, task2, task3 }, TimeSpan.FromMinutes(1))) { }
//wait the first one with or without timeout
//others will although complete their job
//wait any always returns the index of the fastest
Task.WaitAny(task1, task2, task3);
Task.WaitAny(new[] { task1, task2, task3 }, 1000);
Task.WaitAny(new[] { task1, task2, task3 }, TimeSpan.FromMinutes(1));
```

### Task exception handling

Slightly different from our usual exception handling, as already seen in the *Exception Handling* section in *Chapter CLR Internals*, when dealing with tasks, it is impossible to bubble up a raw exception. Any time an exception happens within a task, any tasks waiting, will receive an `AggregateException` error that acts as a container for all the exceptions that happened within the tasks being waited on. This behavior is similar to what happens in exceptions that originate in external threads. If we do not invoke the `Join` method to stop the external thread, such exceptions will never route to the main thread. Here's an example:

```
var task1 = Task.Factory.StartNew(() =>{
    throw new ArgumentException("Hi 1");
});
var task2 = Task.Factory.StartNew(() =>{
    throw new ArgumentException("Hi 2");
});
try{
//the wait will join the two threads exception routing
//all unhandled exceptions from external threads to the
//main one
    Task.WaitAll(task1, task2);
}
catch (AggregateException ax){
    foreach (var ex in ax.InnerExceptions)
        Console.WriteLine("{0}", ex.Message);
}
```

## Task cancellation

Another interesting feature when dealing with tasks is the ability to handle task cancellation. A slight similarity does exist between such a design (task cancellation) and the one from the `Thread.Abort` method. The difference is that for threads, an exception is raised by CLR itself, immediately stopping the thread's execution; while here, although the design may seem the same, all of the implementation is in our hands. The definition of critical section is something to be forgotten here. By the way, because we have cancellation handling in our hands, we can come up with all we need to accomplish a graceful exit from any critical code block. To accomplish state-of-the-art cancellation handling, we must create a `CancellationTokenSource` object to trigger the job cancellation. This source object will create a `CancellationToken` object representing a single-use cancellation token. Once used, a new source must be created and used.

To avoid tasks from being started after a cancellation has already been requested, we must pass this cancellation token to the `StartNew` method of the `TaskFactory` class. Together with this optimization, passing the token to the `StartNew` method that informed the **Task Parallel Library (TPL)** that eventually raised an `System.OperationCanceledException` from the token within the task code body, must become a `TaskCanceledException`. Here's a complete example:

```
static CancellationToken cancellationToken;
static void Main(string[] args){
//let us configure a minimal //threadpool size to slow down task execution
    ThreadPool.SetMinThreads(2, 2);
    ThreadPool.SetMaxThreads(2, 2);
//the cancellation token source able to trigger cancellation
    var cancellationTokenSource = new CancellationTokenSource();
//the cancellation token able to give a feedback on cancellation status
    cancellationToken = cancellationTokenSource.Token;
//let's create some task. the cancellationToken is here assigned to each
// task. this links such two objects avoiding a new task from starting if the
// token has already been triggered in addition, passing the token here will
// convert the OperationCanceledException thrown by the
// ThrowIfCancellationRequested method in the TaskCanceledException class
// that will inform TPL that such task has been kindly aborted
    var tasks = Enumerable.Range(0, 10).Select(i =>
Task.Factory.StartNew(OnAnotherThread, cancellationToken)).ToArray();
    Console.WriteLine("All tasks queued for running");
    Console.WriteLine("RETURN TO BEGIN CANCEL TASKS");
    Console.ReadLine();
    cancellationTokenSource.Cancel();
    Console.WriteLine("Cancel requested!");
    try {
//join back all tasks
        Task.WaitAll(tasks);
    }
    catch (AggregateException ax){
//all tasks will throw an OperationCanceledException
        foreach (var ex in ax.InnerExceptions)
            if (!(ex is TaskCanceledException))
                Console.WriteLine("Task exception: {0}", ex.Message);
    }
    foreach (var t in tasks)
        Console.WriteLine("Task status ID {0}: {1}", t.Id, t.Status);
    Console.WriteLine("END");
    Console.ReadLine();
}
```

```

}
[DebuggerHidden] //avoid visual studio from catching token exceptions
private static void OnAnotherThread(){
    Console.WriteLine("Task {0} starting...", Task.CurrentId);
    //do some CPU intensive job
    for (int i = 0; i < 100; i++){
    //prevent a cancelled task continuing doing useless job
        cancellationToken.ThrowIfCancellationRequested();
    //CPU job
        Thread.Sleep(500);
    }
    Console.WriteLine("Task {0} ending...", Task.CurrentId);
}
}

```

The following is the console output:

```

All tasks queued for running
RETURN TO BEGIN CANCEL TASKS
Task 2 starting...
Task 1 starting...
Cancel requested!
Task status ID 1: Canceled
Task status ID 2: Canceled
Task status ID 3: Canceled
Task status ID 4: Canceled
Task status ID 5: Canceled
Task status ID 6: Canceled
Task status ID 7: Canceled
Task status ID 8: Canceled
Task status ID 9: Canceled
Task status ID 10: Canceled
END

```

## Task continuation

Another useful feature of any task is the ability to attach (with an extension method) any other task.

When dealing with task continuation, the `Status` property of any task, and eventually the `Result` property with the internal return value may be valued to apply the right logic for each result.

In such operations, the task continuation helps us by providing a comfortable enumeration used to select the desired `Status` property when continuation occurs:

```

//a task
var task1 = Task.Factory.StartNew(() =>{
    Thread.Sleep(1000);
    //uncomment here for testing the error
    //throw new Exception("Hi");
    return 10;
});
//a continuation is attached to task1
task1.ContinueWith(task =>{
    //this continuation occur only when previous task will run without errors
    Console.WriteLine("OK: {0}", task.Result);
}, TaskContinuationOptions.OnlyOnRanToCompletion);
task1.ContinueWith(task =>{
    //this continuation will occur only if something goes wrong
    Console.WriteLine("ERR: {0}", task.Exception.InnerException); //the first
    inner exception
}, TaskContinuationOptions.NotOnRanToCompletion);

```

Without waiting for the `task1` completion, although a continuation occurs, it skips the need to handle eventual exceptions in the task-creating code.

With continuation and synchronization techniques used together, complex scenarios of asynchronous programming are available to programmers without having to face difficulties such as manual synchronization with signaling locks or by handling parent-child thread synchronization.

## Task factories

As seen previously, the `Task` and `TaskFactory` classes give us the ability to start tasks with special options. Although this is actually an interesting feature, we still use the default factory available, available throughout the `Task.Factory` property.

A `TaskFactory` class can also be instantiated with custom options that will work as the starting configuration for any task made with this factory. This is particularly useful when multiple instances of the same kind of task are going to be created. Here is an example:

```
static void Main(string[] args){
    var cancellation = new CancellationTokencSource();
    //a factory for creating int-returning tasks
    //all tasks created by this factory will share this default configuration
    //all tasks will support cancellation
    //all tasks will start in an attached-to-parent fashion
    //all tasks will accept a continuation occurring only on success
    //the default TaskScheduler will be used
    var factory = new TaskFactory<int>(cancellation.Token,
                                     TaskCreationOptions.PreferFairness,
TaskContinuationOptions.AttachedToParent, TaskScheduler.Default);
    //10 tasks
    var tasks = Enumerable.Range(1, 10)
        .Select(i => factory.StartNew(CreateRandomInt)
    //all tasks continue as attached (from factory) and skipping faulted results
        .ContinueWith(HandleRandomInt,
TaskContinuationOptions.NotOnFaulted))
    //always define such query materialization
    //differently, the foreach run could trigger infinite task creations
        .ToArray());
    bool canContinue = false;
    //a single continuation async task will start when all other tasks will
    // end such usage avoid calling WaitAll
    factory.ContinueWhenAll(tasks, allTasks =>{
        canContinue = true;
        return 0;
    });
    do{
        Console.Clear();
        foreach (var task in tasks)
            Console.WriteLine("Task n. {0}: {1}", task.Id, task.Status);
        Thread.Sleep(1000);
    }
    while (!canContinue);
    Console.WriteLine("END");
    Console.ReadLine();
}
//this method executes only if the previous task completed successfully
private static void HandleRandomInt(Task<int> task){
```

```

        Console.WriteLine("Handling value: {0}", task.Result);
    }
    static Random random = new Random();
    [DebuggerHidden] //this attribute disables exception debugging
    public static int CreateRandomInt() {
        //wait 1~10 seconds
        Thread.Sleep(random.Next(1000, 10000));
        //throw exception sometime
        if (random.Next(1, 100) % 10 == 0)
            throw new ArgumentException("Unable to produce a valid integer
value!");
        return random.Next(1, 100);
    }
}

```

This example launches multiple tasks to retrieve integers with some complex calculations (the random sleep time). Later, for the only successfully generated integers, a continuation task is assigned to handle these new values. Usually, this application would have used interaction (`for/foreach`), although in multiple tasks a whole re-join of all such asynchronous executions to catch all results is required. Instead, using continuations, everything is easier because such interaction does not occur.

Another interesting feature visible in the example is the ability to have a continuation task for the whole group of tasks instead of a task-by-task basis. Such usage avoids the `WaitAll` invocation. Remember that `WaitAll` works like a `Thread.Join` method, which opens the door to exception bubbling of all joined tasks in the caller thread.

### Task UI synchronization

When dealing with asynchronous programming, the UI experience may achieve great improvement. The first look at any Windows Phone or Windows Store application will easily grant such feedback because of the obligation Microsoft made to SDKs for such platforms.

Dealing with an application that never waits for any external/internal resource or computation, and always remains responsive, is a great feature. The drawback is that Windows Forms and WPF controls are unable to easily update their user data using asynchronous threads (this limitation doesn't exist on ASP.NET).

Both frameworks, Windows Forms and WPF, implement their controls on a **Single Thread Apartment (STA)** with affinity. This means that all objects born on the starting thread will be available only throughout this thread. This affinity works like a firewall that prevents any other thread from accessing the resources behind it. So, although it is possible in asynchronous tasks/threads that do computations or that consume resource usage, when they are exited, any UI update must flow from the initial calling thread that created the UI controls, and maybe the same one that started the tasks.

This **is** a simple example that produces an exception **in** WPF:

```

//this is a no-MVVM WPF script
public partial class MainWindow : Window {
    public MainWindow() { InitializeComponent(); }
    private void Window_Loaded(object sender, RoutedEventArgs e) {
        Task.Factory.StartNew(() => {
            Thread.Sleep(3000);
            //asynchronously set the Content of a Label on UI
            Labell.Content = "HI";
        });
    }
}
}

```

When the `Content` property is set, `InvalidOperationException` type is raised with this description: The calling thread cannot access this object because a different thread owns it. As said previously, before the preceding example, all UI controls live within a single thread in STA mode. This is why we have such behavior in WPF (and in Windows Forms too).

The solution is easy here because this is a script in WPF code-behind. It is enough to use a `Dispatcher` class, already exposed by any WPF control that works as a bridge to ask the UI thread to do something we want. The preceding task becomes this:

```
Task.Factory.StartNew(() =>{
    Thread.Sleep(3000);
    var computedText = "HI";
    //asynchronously set the Content of a Label on UI
    Dispatcher.Invoke(() =>{
    //this code will execute on UI thread in synchronous way
        Label1.Content = computedText;
    });
});
```

When working in MVVM, a dispatcher is useless when a notification occurs. When any class that implements the `INotifyPropertyChanged` interface raises its `PropertyChanged` event, any control in binding with any property of this object, will read the underlying data again, refreshing the UI control state. This inverted behavior breaks the need for a direct cross-thread invocation with the dispatcher. The following is an example using the view model:

```
//a simple viewmodel
public sealed class SimpleViewModel : INotifyPropertyChanged{
    //supporting INotifyPropertyChanged is mandatory for data changes
    notification to UI controls with data binding
    public event PropertyChangedEventHandler PropertyChanged;
    //a simple helper method for such notification
    private void Notify([CallerMemberName] string name = null) {
        if (PropertyChanged != null && name != null)
            PropertyChanged(this, new PropertyChangedEventArgs(name));
    }
    public SimpleViewModel() {
    //the task is fired as view model is instantiated
        Task.Factory.StartNew(OnAnotherThread);
    }
    private string text;
    public string Text { get { return text; } set { text = value;
Notify(); } } //this is how notify fires
    private void OnAnotherThread() {
        Thread.Sleep(3000);
    //asynchronously set a text value
        Text = "HI";
    }
}
```

To avoid coupling between a `ViewModel` and a `View`, we need supporting cross-threaded operations within a `ViewModel` with the ability to execute some UI update in the proper thread (the UI thread one), although we need using the `Dispatcher` from within the `ViewModel` itself, we may use it without having the `View` passing it to the `ViewModel` in a direct way. The `ViewModel`, can simply store the initial creation thread in its constructor code, and later use such thread to ask for a dispatcher linked

to this thread. This choice gives the ability to make cross-threaded operations against object living within the UI thread, without having the ViewModel to directly interact with View. The following is an example:

```
//a simple ViewModel
public sealed class CrossThreadViewModel : INotifyPropertyChanged{
//supporting INotifyPropertyChanged is mandatory for data changes
notification to UI controls with data-binding
    public event PropertyChangedEventHandler PropertyChanged;
//a simple helper method for such notification
    private void Notify([CallerMemberName] string name = null)    {
        if (PropertyChanged != null && name != null)
            PropertyChanged(this, new PropertyChangedEventArgs(name));
    }
    Thread creatingThread;
    public CrossThreadViewModel(){
//in the constructor the caller thread is stored as field
        creatingThread = Thread.CurrentThread;
        Texts = new ObservableCollection<string>();
        Task.Factory.StartNew(OnAnotherThread);
    }
//an observable collection is a self-notifying collection
//that must be accessed by the creating thread
    private ObservableCollection<string> texts;
    public ObservableCollection<string> Texts { get { return texts; } set
{ texts = value; Notify(); } } //this is how notify fires
    private void OnAnotherThread(){
        Thread.Sleep(3000);
//asynchronously create a text value
        var text = "HI";

//add the value to the collection with a dispatcher
//for the creating thread as stored
        Dispatcher.FromThread(creatingThread).Invoke(() =>{
            Texts.Add(text);
        });
    }
}
```

Another solution is available to access collections from multiple threads within a WPF application in .NET 4.5 or greater. We can request the WPF that is synchronizing an STA object, by using a lock and simply invoking the `BindingOperations.EnableCollectionSynchronization` method. Here the previous example is modified using the `EnableCollectionSynchronization` class:

```
static object lockFlag = new object(); //the collection accessing lock
Thread creatingThread;
public CrossThreadViewModel(){
//in the constructor the caller thread is stored as a field
    creatingThread = Thread.CurrentThread;
    Texts = new ObservableCollection<string>();
    BindingOperations.EnableCollectionSynchronization(Texts, lockFlag);
    Task.Factory.StartNew(OnAnotherThread);
}
//an observable collection is a self-notifying collection
//that must be accessed by creating thread
private ObservableCollection<string> texts;
public ObservableCollection<string> Texts { get { return texts; } set { texts
= value; Notify(); } } //this is how notify fires
```

```

private void OnAnotherThread() {
    Thread.Sleep(3000);
    //asynchronously create a text value
    var text = "HI";
    Texts.Add(text); //no more dispatcher needed
}

```

Although this solution will avoid the need of collection synchronization, this will not avoid all of the cross-thread issues, and sometimes we still need to use the dispatcher using the synchronous ViewModel creation thread, as shown in the previous example.

When working in Windows Forms, although the dispatcher is unavailable, a solution always exists for such cross-thread issues. It is enough to ask the control (or Windows Form) to do the cross-thread operation for us, using a delegate identical to the one from the dispatcher.

The following is an example of a wrongly-made cross-thread operation that will generate the same `InvalidOperationException` exception already seen in the WPF example:

```

public Form1() {
    InitializeComponent();

    Task.Factory.StartNew(() =>{
//this code will fail
        label1.Text = "Hi";
    });
}

```

The following code example shows the right way to avoid the `InvalidOperationException` class:

```

public Form1() {
    InitializeComponent();

    Task.Factory.StartNew(() =>{
//this code will fail
        label1.Text = "Hi";
    });
}

```



## Async/await

Asynchronous programming was first released for Microsoft Visual Studio 2012 as an add-on, and is now natively available in Microsoft Visual Studio 2013. Asynchronous programming is also available with a special pattern called `async/await`, which is greatly optimized for cross-thread operations.

This pattern helps to achieve asynchronous programming in a simplified way and adds a transparent (to programmers) asynchronous/synchronous jump, row-by-row, with the ability to execute code on the UI, creating threads without having to use any dispatcher or a delegate. The following is an example from the legacy Windows Forms as seen earlier:

```
public Form1 () {
    InitializeComponent ();
    OnAsyncWay ();
}
private async Task OnAsyncWay () {
    //running in creating thread async elaboration
    //this starts a new task that returns the required value
    var text = await Task.Factory.StartNew (() =>{
    //running on another thread
        Thread.Sleep (1000);
        return "Hi";
    });
    //running again on creating thread
    labell1.Text = text;
}
```

As it is visible (although usually the `StartNew` syntax returns a task), this is executed by the `await` method that translates it in the asynchronous returned values as the `Result` property of the task itself.



Any `async` method must await something, else it is actually useless.

The `async` keyword specifies that the whole method contains asynchronous calls. The `await` keyword, instead, specifies that the next invocation will execute asynchronously, and when such a task ends, the code must continue again in a synchronous way on the caller thread. Unlike a dispatcher, that requests another thread that is doing something, with `async/await` methods we can write code that can work on multiple threads in a very simplified way.

We can also await multiple tasks with the `Task.WhenAll` method, as follows:

```
private async Task OnAsyncWay () {
    var allValues = await Task.WhenAll (
        Task.Factory.StartNew<int> (TaskRunner1),
        Task.Factory.StartNew<int> (TaskRunner2),
        Task.Factory.StartNew<int> (TaskRunner3)
    );
}
private int TaskRunner3 () {return 3;}
private int TaskRunner2 () {return 2;}
private int TaskRunner1 () {return 1;}
}
```

When dealing with `async/await` methods, if sleep-time is needed, instead of using the `Thread.Sleep` methods that occur on the main thread, use the `Task.Delay` method, which will wait for the same time without having to block the calling thread. The following is an example:

```
await Task.Delay (1000);
```

Keep in mind that `async/await` methods add the thread switching feature to the existing task-based asynchronous programming techniques already seen earlier. Everything is still available, from task continuation to factory configuration, task child synchronization, and so on. With such added features, it becomes very easy to deal with UI updates. Asynchronous programming is necessary to achieve a greater user experience and high throughput/scalability for any application.

For further reading, you can refer to the following link <https://msdn.microsoft.com/en-us/library/hh191443.aspx>.

## Summary

In this chapter, you saw how asynchronous programming is available to developers with different .NET techniques. Although with the last .NET editions, the TAP (with the `async/await` method) will be the main choice when dealing with such programming. A complete knowledge of the available solutions is mandatory anytime we cannot use the newest .NET edition. Moreover, it is actually a plus to know all the available techniques because such a wide knowledge opens the mind of any developer to the asynchronous theory problems and solutions.

In the next chapter, you will learn more about parallelism, another important aspect of high performance programming.

## Programming for Parallelism

Within the .NET world, **parallel programming** is the art of executing the same job on a collection of data or functions by splitting the desired elaboration over all available computational resources.

This chapter will focus on .NET **Task Parallel Library's (TPL)** implementation of parallel computing, together with the **Parallel Language Integrated Query (PLINQ)** language.

This chapter will cover the following topics:

- Parallel programming
- Task parallelism with TPL
- Data parallelism with TPL
- Integrated querying with LINQ
- Data parallelism with PLINQ

### Parallel programming

The goal of any parallel programming is to reduce the whole latency time of the operation by using all the available local resources, in terms of CPU computational power.

Two definitions of parallelism actually exist. **Task parallelism** happens when we execute multiple jobs all together, such as saving data against multiple database servers.

**Data parallelism**, instead, happens when we split a huge dataset elaboration across all available CPUs, like when we have to execute some CPU demanding method against a huge amount of objects in the memory, like hashing data.

In the .NET framework, we have the ability to use both parallel kinds. Despite that, the most widely used kind of parallelism within the .NET framework's programming is data parallelism, thanks to PLINQ being so easy to use.

The following table shows the comparison between Task parallelism and Data parallelism:

	Task parallelism	Data parallelism
<b>What does it parallelize?</b>	Parallelizable functions	Parallelizable data
<b>Performance boost</b>	Reduces overall execution time by executing multiple functions per time period	Reduces overall execution time by splitting the same algorithm's execution across all the available CPUs
<b>Starting constraint</b>	The same initial data state	The same data set
<b>Ending constraint</b>	Can end up all together in a synchronous or asynchronous way	Must end up all together in a synchronous way
<b>Messaging</b>	If required, any task can message others or can await others with signaling locks, as seen in the <i>Multithreading Synchronization</i> section in <i>Chapter CLR Internals</i>	Usually nonexistent

There is a tight coupling between **multithreading (MT)** programming and parallel programming. MT is actually a feature of programming languages that helps us by using low-level operating systems threads that give us the ability to run multiple code all at the same time.

Parallel programming, instead, is a high-level feature of programming languages, which will handle multiple operating-system threads autonomously, giving us the ability to split some jobs at a given time and later catch the result in a single point.

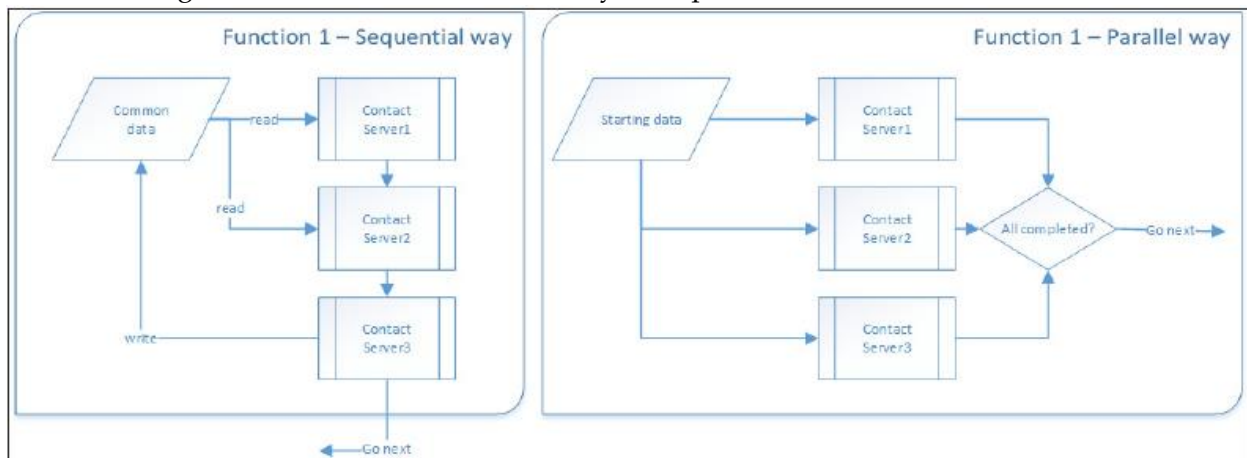
Multithreaded programming is a technique in which we work with multiple threads by ourselves. It is a hard-coded technique whereby we split the different logic of our applications across different threads. Opening two TCP ports to make a two-threaded network router is multithreading. Executing a DB read and data fix on a thread and a DB write on another thread is still multithreading. We are actually writing an application that hardly uses multiple threads.

In parallelism, instead, there is a sort of orchestrator, a chief of the whole parallel processing (usually the starting function or routine). The unified starting point, makes all parallel thread handlers share the same additional starting data. This additional data is obviously different from the divided main data that start up the whole parallel process, like a collection of any enumerable.

When dealing with multithreading programming, it is like executing multiple applications that live within the same process all together. They may also talk to each other with locking or signals, but they do not need to.

### Task parallelism

Task parallelism happens when we want to split different activities (functions or algorithms) that start from the same point with the same application state (data). Usually, these paralleled tasks end up all together in a task-group continuation. Although this common ending is not mandatory, it is maybe the most canonical definition for task parallelism within the .NET TPL. You should recognize that the choice of continuing with a single task or with multiple tasks, or waiting on another thread of tasks does not change the overall definition. It is always task parallelism.



How task parallelism changes a sequential communication with multiple external systems

Any time we need to do different things all together with the same starting data, it is task parallelism. For example, if we need to save data across three different DBs all together, it is task parallelism. If we need to send the same text throughout a mail a file log and a database, those three asynchronous tasks are task parallelism.

Usually, these different things do not need to talk to each other. If this is a requirement, usual locks or (better) signaling locks may give us the ability to drive such multiple asynchronous programming in order to avoid race conditions in resource usage or data inconsistencies with multiple read/writes. A

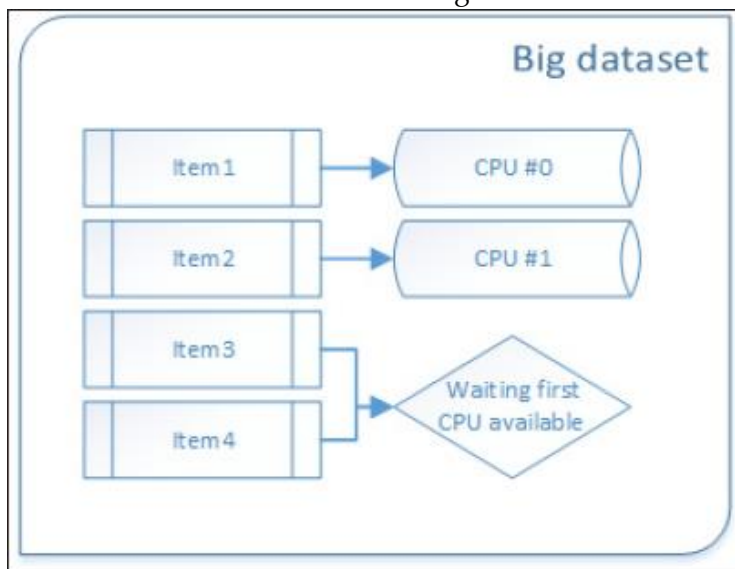
messaging framework is also a good choice when dealing with a multiple asynchronous task execution that needs some data exchanging outside the starting data state.

When using asynchronous programming with multiple tasks (refer to the *Asynchronous Programming Model* section in *Chapter Asynchronous Programming*), it may be that we actually use multiple threads. Although this is task-based multithreading, this is not task parallelism because it misses a shared starting point and overall shared architecture. Parallelism is made by another abstraction level above the abstraction level of Task-based Asynchronous Pattern (TAP).

When we query a **Delegate** object that is executing some remote method asynchronously, we are actually using a thread-pool thread (we may also use those threads by scratch); we are still using simplified multithreading tools. This is not parallelism.

### Data parallelism

The art of executing the same function/method against a single (usually huge) dataset is called data parallelism. When working with a huge dataset, parallel programming can bring about an impressive time reduction of the execution of algorithms.



How data parallelism splits data items across CPUs

Within data parallelism, there are more rules; this is different from task parallelism, in which we can actually implement any logic when creating our parallelized functions. Most important of all is that all data must come from a single (usually huge) dataset. This principle is directly coupled to the **set theory**.

A **set** is a uniform group of items of the same type. In the .NET world, a set is any typed array, collection, or the same data type. Like in any relational database, a single table may contain only a homogenous group of items; the same thing happens when we talk about a set. Indeed, a relational table is actually derived from the set theory.

It is not enough to have multiple items all together to create a set. Actually, a set must have any number of items that can be handled as a single unique super-entity. All items that compose a set must be structured as a whole. This means that to practice correctly with data parallelism, a single object type must fill the set once (no duplications), and no logic will ever be admitted to interact with a single item composing a set if the same logic will not be executed against all other items. Another principle about a set is that items do not have any order. Although, they do have an identifier; otherwise, duplication could occur.

## Task parallelism with TPL

As mentioned earlier, task parallelism happens when dealing with parallel invocations of multiple methods/function. Within the .NET Framework, this can be obtained with the invocation of the `Parallel.Invoke` method, which needs to have as a parameter all parallelizable actions as a whole. Most techniques applicable here are also applicable in asynchronous programming with the `Task` or the `TaskFactory` class. So reading *Chapter Asynchronous Programming* is mandatory to get the best of task parallelism.

The `Parallel.Invoke` method simply takes multiple remote methods to call procedures in a parallel way by accepting a `System.Action` array. Here is an example:

```
static void Main(string[] args){
//short form with named methods
    Parallel.Invoke(Method1, Method2, Method3);

//short form with anonymous methods
    Parallel.Invoke(
        () => { },
        () => { },
        () => { });
}
static void Method1() { }
static void Method2() { }
static void Method3() { }
```

In the following code example, we will process a picture resize in two different resolutions using task parallelism:

```
static void Main(string[] args){
//add reference to System.Drawing assembly
//an original image file
    byte[] originalImageData = File.ReadAllBytes("picture.jpg");
    byte[] thumb300x200 = null;
    byte[] thumb150x100 = null;
//resize picture to 300x200px and 150x100px for thumbprint needs
    Parallel.Invoke(
        new Action(() =>{
            thumb300x200 = ResizeImage(originalImageData, 300, 200);
        }),
        new Action(() =>{
            thumb150x100 = ResizeImage(originalImageData, 150, 100);
        })
    );
//save the resized images
    File.WriteAllBytes("pricture-300.jpg", thumb300x200);
    File.WriteAllBytes("pricture-150.jpg", thumb150x100);
}
static byte[] ResizeImage(byte[] original, int newWidth, int newHeight){
//creates a stream from a byte[]
    using (var sourceStream = new MemoryStream(original))
//load a bitmap from source stream
    using (var originalBitmap = new Bitmap(sourceStream))
//resize the original bitmap to a new size
    using (var resizedBitmap = new Bitmap(originalBitmap, newWidth,
newHeight))
//creates a new in-memory stream from resized image
```

```

        using (var targetStream = new MemoryStream())
        {
            //save resized image to the in-memory stream
            resizedBitmap.Save(targetStream, ImageFormat.Jpeg);
            //return a byte[] from the saved stream
            return targetStream.ToArray();
        }

```

The `Parallel.Invoke` method will do the most work for us by actually creating a task for each action we need to process; thus obtaining the parallelization needed.

As with any task creation by the `TaskFactory` class, here we have the ability to configure some task creation options such as the maximum concurrent task number, giving a `CancellationToken`, and so on:

```

Parallel.Invoke(new ParallelOptions{
    MaxDegreeOfParallelism = 2,
},
    () => Method1(),
    () => Method2(),
    () => Method3()
);

```

An important fact that we always have to deal with when working with parallel programming is that this result has no order. Because of parallelization, we cannot predict task execution time. We must simply wait for completion.

A similar result is available through the `WaitAll` behaviour:

```

Task.WaitAll(
    Task.Run(() => Method1()),
    Task.Run(() => Method2()),
    Task.Run(() => Method3())
);

```

Although this choice adds the ability to handle timeout as we wish, it provides a similar result because it lacks in task-group configuration, as what was offered by the `ParallelOptions` class. A solution is to use a custom class extending the `TaskFactory` class, but this choice will add nothing more than using the `Parallel.Invoke` method.

Please note that the focus when dealing with task parallelism is that the framework handles lot of things by itself; first of all, the task's creation and destruction. Because of this, the `WaitAll` method is a bit outside of the theory of task parallelism; it's only related to multiple asynchronous programming.

An interesting usage scenario for task parallelism is in **speculative execution**. This happens when we execute some task before it is actually needed, or in a more general way, when we do not need it. A canonical example is what happens when we execute multiple searches against our data source (or web) with different parameters. Here, only the fastest tasks win, so all other slower tasks are canceled. Here is an example:

```

static void Main(string[] args) {
    //a cancellation token source for cancellation signalling
    using (var ts = new CancellationTokenSource())
    //tasks that returns a value
    using (var task1 = Task.Factory.StartNew<int>(TaskWorker, ts.Token))
    using (var task2 = Task.Factory.StartNew<int>(TaskWorker, ts.Token))
    using (var task3 = Task.Factory.StartNew<int>(TaskWorker, ts.Token)) {
    //a container for all tasks

```

```

        var tasks = new[] { task1, task2, task3 };
//the index of the fastest task
        var i = Task.WaitAny(tasks);
//lets cancel all remaining tasks
        ts.Cancel();
        Console.WriteLine("The fastest result is {0} from task index {1}",
tasks[i].Result, i);
//bring back to the starting thread all task exceptions
        try{
            Task.WaitAll(tasks);
        }
        catch (AggregateException ax){
//let's handle all inner exceptions automatically
//if any not OperationCanceledException exist
//those will be raised again
            ax.Handle(ex => ex is OperationCanceledException);
        }
    }
    Console.ReadLine();
}
private static readonly Random random = new Random();
private static int TaskWorker(object token_as_object){
//the token is available as object parameter
    var token = (CancellationToken)token_as_object;
//do some long running logic
    var finish = DateTime.Now.AddSeconds(random.Next(1, 10));
    while (DateTime.Now < finish){
//if the cancellation has been requested
//an exception will stop task execution
        token.ThrowIfCancellationRequested();
        Thread.Sleep(100);
    }
    return random.Next(1, 1000);
}
}

```

Please note that although we can obtain task parallelism with by simply using the `Parallel.For/ForEach/Invoke` methods, complex scenarios are available only by manually handling task creation, continuation, and waiting. Please remember that a task is simply a deferred job. Nothing more or less. It is how we use it that makes our design using parallelism or asynchronous programming.

### Data parallelism with TPL

As already said, data parallelism with TPL happens only when we deal with a dataset (not the `DataSet` class). Within .NET, the easy way of doing this is with the `Parallel.For` and `Parallel.ForEach` methods from the `Parallel` module. The following example shows the basic usage of the `Parallel` module:

```

for (int i = 0; i < 10; i++){
//do something
}
Parallel.For(0, 10, i =>{
//do something
});

```



The first thing that catches our eye is the singularity of logic. While in task parallelism we deal with multiple instances of logic; here, there is always only one type of logic. We simply execute it on multiple CPUs.

This example is obviously incompatible with the Set Theory previously exposed, because there is neither a simple collection of objects. In other words, the parallel `For` is an iterative structure as the normal `For`.

To parallelize some logic regarding just a simple collection, the `Parallel` class gives us the `ForEach` method:

```
var invoiceIdList = new[] { 1, 2, 3, 4, 5 };
Parallel.ForEach(invoiceIdList, InvoiceID =>{
//do something
});
```

This alternative made with the `Parallel.ForEach` method outclasses the very simple result achieved by the `Parallel.For` method implementation, giving us the chance to work against a collection that is a `Set`.



Although a collection in .NET is not actually a `Set`, it is quite similar. The only missing requirement is that a collection does not guarantee the uniqueness of items.

Any collection of any size may be enumerated by the `Parallel.ForEach` method. Obviously, the best performance improvement is achieved by big collections because the more items there are, the more the TPL engine can split such items across multiple threads.

Parallelism in .NET executes on the TPL framework. This means that threads from `ThreadPool` are used to execute parallel jobs. Limitations and configurations of such behavior were exposed in the *Task-based Asynchronous Pattern* section in *Chapter Asynchronous Programming*.

An added feature available in parallelism is the throttling configuration within the `ParallelOptions` class. `Parallel.Invoke/For/ForEach` methods accept an instance of this class, giving the ability to specify a maximum amount of parallel executions. Here's an example:

```
//our throttling configuration
var throttling = new ParallelOptions{ MaxDegreeOfParallelism = 2};
//let's process the data in a parallel way
Parallel.ForEach(invoiceIdList, throttling, i =>{
//do something
});
```

## ThreadPool tuning

Please bear in mind that TPL uses threads from `ThreadPool` to execute a task's code. This is why tuning `ThreadPool` is so important when using parallel programming extensively. In addition to what we saw in the *Task-based Asynchronous Pattern* section in *Chapter Asynchronous Programming*, here we will try to show you what happens if we try to increase the thread pool size to an extreme, for example:

```
ThreadPool.SetMinThreads(256, 256);
ThreadPool.SetMaxThreads(4096, 4096);
```

The configuration shown asks the thread pool to increase its size from a minimum of 256 threads to a maximum size of 4096 (losing dynamic size management – the default value for the maximum size).

Increasing the thread pool size at such high values will cost some CPU usage and memory because the operating system needs such resources in thread creation.

Obviously, such a high thread availability will give TPL the ability to parallelize hundreds of tasks (at least 256, as set earlier). Carefully increment so extremely global thread availability because of the increased possibility of cross-thread issues that we will need to handle with locks and signals, as seen in the *Multithreading Synchronization* section in *Chapter CLR Internals*. In fact, with such an extreme concurrency level, when using locks to serialize specific code blocks, a huge overhead in terms of CPU time will occur because of the contest of the lock flag that all concurrent threads will try to obtain.

## Parallel execution abortion

Within a parallel iteration, we cannot use the `break` keyword in any classic `for/foreach` statement. If we need a similar behavior, we can use an overload of the `foreach` method that will execute inner parallel code by using an `Action<T, ParallelLoopState>` class that in addition to the iterating item will also inject a `ParallelLoopState` object available to the inner code. This state object will provide information about the overall parallel state or let us request a premature stop of the full parallel execution. Here's a complete example:

```
static void Main(string[] args) {
    //a big dataset
    var invoiceIdList = Enumerable.Range(1, 1000);
    int c = 0;
    Parallel.ForEach(invoiceIdList, (id, state) =>{
        //stop all ForEach execution if anything go wrong
        try{
            //execute some logic
            ExecuteSomeCode(id);
            //within the lambda/method we can know about stop signalling
            if (state.IsStopped)
                Console.WriteLine(
                    "Executed # {0} when IsStopped was true", c++);
            else
                Console.WriteLine("Executed # {0}", c++);
        }
        catch (Exception ex) {
            Console.WriteLine("Error: {0}", ex.Message);
        }
        //stop all parallel process
        state.Stop();
        Console.WriteLine("Requested a parallel state break!");
    }
});
Console.WriteLine("END");
Console.ReadLine();
}

private static readonly Random random = new Random();
private static void ExecuteSomeCode(int id) {
    //a random execution time
    Thread.Sleep(random.Next(1000, 2000));
    //an impredicted fail
    if (DateTime.Now.Millisecond >= 800)
        throw new Exception("Something gone wrong!");
}
```

In this example, we used the `Stop` method that actually requests a stop to all subsequent parallel interactions and together will signal the running iterations that a stop has been requested by the `IsStopped` flag. This is the output of such an example (the results can vary a lot):

```
Executed # 0
Executed # 1
Executed # 2
Executed # 3
Executed # 5
Executed # 4
Executed # 6
Executed # 7
Executed # 8
Error: Something gone wrong!
Requested a parallel state break!
Executed # 9 when IsStopped was true
Executed # 10 when IsStopped was true
Error: Something gone wrong!
Requested a parallel state break!
Error: Something gone wrong!
Requested a parallel state break!
Error: Something gone wrong!
Requested a parallel state break!
Executed # 11 when IsStopped was true
Executed # 12 when IsStopped was true
Executed # 13 when IsStopped was true
Executed # 14 when IsStopped was true
END
```

As shown, after the initial normal execution of parallel statements (from #0 to #8), an error has occurred; this invoked the `Stop` method of the `ParallelLoopState` class, which is available in the state parameter within the lambda code. This prevented new interactions of the `Parallel.ForEach` method. Within the already executing interactions, the `IsStopped` flag is given a value of `true`, so (eventually) a proper logic may be applied.

Similar to the `Stop` method, the `Break` method can also stop the execution of a parallel statement but it will stop executing only the items that will follow the calling item in the underlying collection order.

If we have a collection of integers from 1 to 100, and when processing the 14th we called the `Break` method, only items from 15 to 100 will actually receive the `IsStopped` flag or will not run at all.

## Partitions

Any time we deal with data parallelism, TPL will prepare data to flow in different tasks in small groups. Such groups are called **partitions**.

Two default partition logics are available within the .NET framework. **Range partitioning** happens against any finite collection. It divides the collection between all available threads, and any partition is then executed within its related thread. The following shows an example of the `Parallel.For` method that produces a finite indexer collection of values:

```
Parallel.For(1, 1000, item =>{
    Console.WriteLine("Item {0} Task {1} Thread {2}", item, Task.CurrentId,
        Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(2000);
});
```

This code produces the following output:

```
Item 1 Task 1 Thread 8
Item 125 Task 2 Thread 9
Item 249 Task 3 Thread 11
Item 373 Task 4 Thread 10
Item 497 Task 5 Thread 12
Item 621 Task 6 Thread 16
Item 745 Task 7 Thread 14
Item 869 Task 8 Thread 15
Item 993 Task 9 Thread 13
```

As visible, the index value collection has been divided by 9, as 8 is the number of the initial `ThreadPool` size, plus the one new thread created by `ThreadPool` is triggered by the huge pre-empted work. The same range partitioning logic is involved by using the `AsParallel()` method against any other finite collection such as an `Array`, `ArrayList`, `List<T>`, and so on.

Another built-in partition logic is the **chunk partitioning** logic, which takes place whenever we use the `Parallel.ForEach` method or the `AsParallel()` method against any enum without a finite length. This partitioning is based on an enumerator logic. It simply asks for some new item (usually the same amount as the number of CPU cores), creates a new task for this item group, and puts the execution on an available thread, and then waits for any new thread's availability to start its logic again. In chunk partitioning, the chunk size is known at start and totally handled by the TPL inner logic.

Chunk partitioning has a better balancing capability than the range partitioning because a chunk is often smaller than a partition.

If built-in partitioning logic is not enough for our needs, we can create a custom partitioner by inheriting from the `Partitioner<T>` class. A custom partition logic can avoid using locks, greatly improve overall resource usage, and lead to energetic efficiency within the whole solution. A complete guide is available on the MSDN website:

[https://msdn.microsoft.com/en-us/library/dd997411\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd997411(v=vs.110).aspx)

Although chunk partitioning supports dynamic chunk sizes, this size is invariant during a single enumeration. If we need full dynamic partitioning, we need to create a partitioner. An example is shown on the MSDN website:

<https://msdn.microsoft.com/en-us/library/dd997416%28v=vs.110%29.aspx>

Further details about partitioning are explained in the *Partitioning optimization* section later in this chapter.

### Sliding parallel programming

An interesting behavior takes place when we combine sliding programming, just like when using a cursor from a stream or an enumerable with parallel programming. In this scenario, we add high computation speed together with a very low footprint in the memory because of the tiny memory usage made by the few pieces of data currently loaded in each thread. Here is an example:

```
static void Main(string[] args) {
    var enumerable = GetEnumerableData();
    Parallel.ForEach(enumerable, new ParallelOptions {
        MaxDegreeOfParallelism = 2,
    }, i => {
        //process the data
        Console.WriteLine("Processing {0}...", i);
        Thread.Sleep(2000);
    });
}
```

```

    });
    Console.WriteLine("END");
    Console.ReadLine();
}
private static IEnumerable<int> GetEnumerableData() {
//let's produce an enumerable data source
//eventually use an underlying steam
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Yielding {0}...", i);
        yield return i;
    }
}
}

```

This scenario gives tremendous computational power without having to keep in memory all data altogether, thus, only actually processing objects resides in memory.

The previous examples shows a single method using the `yield` keyword for manually enumerated values. The example may be improved by implementing multiple methods using the `yield` operator invoking each one to the other. The obtained architecture, will be able to handle extremely complex logic without never having to keep more than needed data in memory.

### Integrated querying with LINQ

The **Language-Integrated Query (LINQ)** framework is what mostly changes the programming technique in the .NET world. LINQ is a framework that lives within the .NET language and helps us create complex queries against any data source, such as in-memory objects using the LINQ to object-provider or against an Entity Framework database using the LINQ to entities provider.

Almost anything that is enumerable in the world has its own provider within a simplified index, as shown at the following link:

<http://blogs.msdn.com/b/charlie/archive/2008/02/28/link-to-everything-a-list-of-linq-providers.aspx>

Any iteration for the `for` or `foreach` statements is now made with LINQ. The same is made against any relational database accessed by an O/RM or any non-relational database too. LINQ has its own pseudo SQL language that is accessible within any .NET language. Here is a LINQ statement example:

```

var query = from i in items
            where i >= 100
            orderby i descending
            select i;

```

By analyzing the preceding statement, you should understand that it represents a simple query that will make a filter for in-memory data, then will order the filtered data, and later will output the data. The key concept of LINQ is that LINQ creates queries. It does not executes such queries; it simply declares queries.

Any query has a proper type. The type of `query` mentioned earlier is `IOrderedEnumerable<int>`, which means that it is an ordered enumerable collection. Because of the verbosity of the query's result type, in conjunction with the frequent usage of anonymous types, the `var` keyword is widely used when typing a variable that will contain the query itself, as the previous example showed.

Please remember that anonymous types will be visible to **Intellisense** (Visual Studio's suggestion system of the text editor) only within the code block in which it was created. This means that outside

the method where we created the anonymous type, we will lose the Intellisense support. The anonymous type will be visible only to the assembly where it was created.

This means that if we need to use an anonymous typed object outside our assembly, we should instead use a statically typed type (not anonymous), or we need to use the `Reflection` namespace to read the `internal` marked properties that compose our anonymous type.

Back to LINQ, the query itself will be executed in a lazy fashion only when iterated by any `foreach` statement or when using specific **Extension methods** that will materialize (produce results) the query in the `ToArray` or `ToList` methods:

```
var concreteValues = query.ToArray();
```

LINQ has a lot of extension methods to access its huge library. Actually, the preceding syntax is very limited compared to all the extension methods available from any enumerable collection. Here's an example:

```
var values = items
    .Where(i => i >= 100)
    .OrderByDescending(i => i)
    .ToArray();
```

All LINQ methods will work in a fluent way, appending any new altering method to the previous one. The Lambda expression is ubiquitous and is used to define new values, value mapping, filter predicates, and so on, so its syntactical deep knowledge is mandatory when using LINQ. The two examples provide identical results, although the second one will not store the query itself but only its concrete result.

The magic of LINQ is the ability to work in any application and query any kind of data, from variables to XMLs with a single syntax. Of course, some limitations are present when dealing with external resources such as databases, because some data providers cannot handle all LINQ methods. In such a case, an exception will be thrown by the provider itself. In modern .NET programming, any iterating logic is usually executed within a LINQ statement.

Another magic aspect of LINQ is the ability to return queries and not always data. This adds the ability to create new features without having to retrieve the same data more than once, or without having to write complex `if/else` statements with possible copy/paste programming. The following example shows how to split the query similar to what we have already seen in the previous example:

```
var query1= items as IEnumerable<int>;
var query2= query1.Where(i => i >= 100);
var query3 = query2.OrderByDescending(i => i);
var values = query3.ToArray();
```

The following example shows how to add new filters to the already filtered query. The two `where` statements will execute just like an `and` clause.

```
var query = items.Where(i => i >= 100);
if (SomeLogic())
    //another where is added to the LINQ
    query = query.Where(i => i <= 900);
```

Please note that using this query result can trigger an undesirably query materialization (execution) multiple times. If you want to work in values, it is always easier to materialize the LINQ with an ending that consists of the `ToArray` or `ToList` method.

LINQ offers, by default, any set operation such as `Union`, `Distinct`, `Intersect`, and `Except`; any aggregate operation such as `Count`, `Sum`, `Max`, `Min`, and `Average`; and any relational data operation such as `Join`, `GroupJoin`, and so on.

Transformation methods such as `Select` or `SelectMany` are also interesting because they give us the ability to change the object that flows from one LINQ step to the other letting us add/remove/change data. By the way, one of the greatest features of LINQ is the ability to append LINQ queries to another LINQ query, including when dealing with multiple LINQ data providers.

Here's an overview of LINQ's features:

```
//a dataset
var items = Enumerable.Range(1, 1000);
//a simple filter
var filter1 = items.Where(i => i <= 100);
//takes until matches a specific predicate
var filter2 = items.TakeWhile(i => i <= 100); //same as Where above.
//shape the original data item in another one
var shape1 = items.Select(x => new{Value = items});
//shape multi-dimensional data in flattened data
//this will produce a simple array from 1 to 9
var shape2 = new[] { new [] { 1, 2, 3 }, new [] { 4, 5, 6 },
                    new [] {7, 8, 9 } }.SelectMany(i => i);
//group data by divisible by 2
var group1 = items.GroupBy(i => i % 2 == 0);
//take only x values
var take1 = items.Take(10);
//take only after skipped x values
var skip1 = items.Skip(10);
//paginate values by using Take and Skip together
var page3 = items.Skip(2 * 10).Take(10);
//join values
var invoices = new[]{
    new {InvoiceID=10, Total=44.50},
    new {InvoiceID=11, Total=34.50},
    new {InvoiceID=12, Total=74.50},
};
//join invoices with items array
//shape the result into a new object
var join1 = invoices.Join(items, i => i.InvoiceID, i => i, (a, b) => new{
    a.InvoiceID, a.Total, Index = b,
});
```

All LINQ methods are combinable with any other data collection made by any other data provider. This means that we can actually make a join between two different database values, or between a database value and a file, or an XML, a Web Service, or a control on our UI, or anything else we could want. Obviously, things are not so easy when we want merge data from multiple LINQ data providers, especially because of the Entity Framework data provider that will try to translate anything written in LINQ into SQL. To help us avoid LINQ to SQL translation issues, there are techniques such as small materializations within the LINQ steps (like putting a `ToArray()` method before performing the join

between different providers) or starting the query with an in-memory source instead of using an Entity Framework `DbQuery` class.

Here is a simple example of a cross LINQ provider query:

```
var localDataset = new[]{
    new { Latitude=41.88f, Longitude=12.50f, Location="Roma"},
    new { Latitude=45.46f, Longitude=9.18f, Location="Milano"},
    new { Latitude=59.32f, Longitude=18.08f, Location="Stockholm"},
};
//within the TestDB there is a simple table as
//CREATE TABLE [dbo].[Position] (
//[Latitude] [real] NOT NULL,
//[Longitude] [real] NOT NULL)
using (var db = new TestDBEntities()){
//this query starts from the local dataset
//and later creates a join with the table within the database
    var query = from p in localDataset
                join l in db.Position on new { p.Latitude, p.Longitude }
equals new { l.Latitude, l.Longitude }
                select new
    {
        l.Latitude,
        l.Longitude,
        p.Location,
    };
//materialize the query
    foreach (var position in query)
        Console.WriteLine("Lat {0:N2} Lon {1:N2}: {2}", position.Latitude,
            position.Longitude, position.Location);
}
Console.ReadLine();
```

By executing the example given, you will know how to join two different data providers by specifying the only coordinate couple that we want see in the database table.



Carefully use lambda expressions because any time we create an anonymous method with lambda, all variables available in the scope of the anonymous method are also available within the method itself. When we use some external variable within the anonymous method, those variables became *captured variables*, changing (extending) their lifecycle, and assuming the same lifecycle of the anonymous method that captures them.

## Data parallelism with PLINQ

PLINQ is the framework required to use LINQ within the TPL parallel framework. In .NET, it is straightforward to use parallelism against any LINQ query because we simply need to add the `AsParallel` method at the root of the query to switch the execution from the simple LINQ engine to PLINQ with TPL support.

The following example will execute two different where conditions against an enumerable using PLINQ:

```
static void Main(string[] args){
//a dataset
    var items = Enumerable.Range(1, 100);
//multi-level in-memory where executed as data parallelism
```



```

    var processedInParallel = items.AsParallel()
        .Where(x => CheckIfAllowed1(x))
        .Where(x => CheckIfAllowed2(x))
        .ToArray();

    Console.ReadLine();
}
private static bool CheckIfAllowed2(int x) {
    Console.WriteLine("Step 2 -> Checking {0}", x);
    //some running time
    Thread.Sleep(1000);
    return x % 3 == 0;
}
private static bool CheckIfAllowed1(int x) {
    Console.WriteLine("Step 1 -> Checking {0}", x);
    //some running time
    Thread.Sleep(2000);
    return x % 2 == 0;
}
}

```

This is a partial result:

```

Step 1 -> Checking 1 //first chunk starts
Step 1 -> Checking 6
Step 1 -> Checking 3
Step 1 -> Checking 2
Step 1 -> Checking 4
Step 1 -> Checking 7
Step 1 -> Checking 8
Step 1 -> Checking 5
Step 1 -> Checking 9
Step 1 -> Checking 10
Step 2 -> Checking 4 //second chunk starts
Step 2 -> Checking 8
Step 2 -> Checking 2
Step 2 -> Checking 6
Step 1 -> Checking 12
Step 1 -> Checking 11
Step 1 -> Checking 13

```

The execution of the preceding example will easily show how PLINQ performed using the chunk partitioning logic. The first chunk of items (10 items) reached Step 1 in a simple way. Just later, the second chunk of items were executed all together. This second chunk contains items of the first chunk that succeeded in passing Step 1 and now are ready for Step 2, and new items for the Step 1.

There is the ability to use the `AsParallel` method without returning values with the `ForEach` method, as shown in the following code example:

```

items.AsParallel().ForEach(i =>{
    //do something
});

```

After an `AsParallel` method invocation, the type of the enumerable changes in `ParallelQuery<T>`. This new type adds configurability for parallelism, such as forcing parallel-concurrency or forcing parallelism itself, although the heuristics of the TPL cannot be enabled if given an enumerable.

Forcing parallelism (with the `WithExecutionMode` method) is useful when the engine does not seem to understand that parallelism, it could add some execution time (latency time) reduction. This

happens because anytime we use the `AsParallel` method, the engine makes a prediction of the reduced execution time, and if this is not positive, the engine can decide to not use parallelism at all. Here is an example:

```
//multi-level in-memory where executed as data parallelism
var processedInParallel = items.AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .Where(x => CheckIfAllowed1(x))
    .Where(x => CheckIfAllowed2(x))
    .ToArray();
```

We can configure parallel concurrency by setting the maximum degree with the `WithDegreeOfParallelism` method. This method is useful for limiting (throttling) concurrency level, and for increasing it above the usual size as defined by the heuristic of the TPL. The maximum size is 64 for .NET 4 and 512 for .NET 4.5+, while the default value is the CPU core count. Here is an example:

```
var processedInParallel = items.AsParallel()
    .WithDegreeOfParallelism(100)
    .Where(x => CheckIfAllowed1(x))
    .Where(x => CheckIfAllowed2(x))
    .ToArray();
```

Another useful method of the `ParallelQuery<T>` class is `WithMergeOptions`, which gives us the ability to configure how the parallel engine will buffer (or not) data from the parallel partitions before collecting the result. The ability to disable buffering at all is interesting. This choice will give the parallel results to any enumerator consuming the parallel query as soon as possible, without having to wait for processing all parallel query items. The following shows an example that consists of parallel merge options:

```
items.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
```

## Partitioning optimization

The CLR gives us the ability to force a specific partitioning logic if the one exposed as the default is not optimal for our needs. The default partitioning logic is automatically chosen according to the data collection type given as an input through the `AsParallel` method. Here is an example:

```
//a dataset
var items = Enumerable.Range(1, 1000).ToArray();
//a customized partitioning logic
//range partitioning
var partitioner = Partitioner.Create<int>(items, false);
partitioner.AsParallel().ForAll(item =>{
    Console.WriteLine("Item {0} Task {1} Thread {2}", item, Task.CurrentId,
Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(2000);
});
```

The preceding example shows a range partitioning logic within the `AsParallel` execution of PLINQ. Here, the result shows the partition size:

```
Item 1 Task 2 Thread 6
Item 751 Task 8 Thread 13
Item 501 Task 6 Thread 16
Item 251 Task 4 Thread 12
Item 376 Task 5 Thread 15
Item 626 Task 7 Thread 14
Item 876 Task 9 Thread 9
Item 126 Task 3 Thread 10
Item 377 Task 5 Thread 15
Item 877 Task 9 Thread 9
```

Instead, the following example shows a load-balancing logic that is obtainable by using the `Partitioner` class:

```
//a dataset
var items = Enumerable.Range(1, 1000).ToArray();
//a customized partitioning logic, a load-balancing logic
var partitioner = Partitioner.Create<int>(items, true);
partitioner.AsParallel().ForAll(item =>{
    Console.WriteLine("Item {0} Task {1} Thread {2}", item, Task.CurrentId,
Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(2000);
});
```

The following is the output:

```
Item 2 Task 2 Thread 10
Item 7 Task 6 Thread 11
Item 5 Task 5 Thread 15
Item 8 Task 8 Thread 16
Item 6 Task 7 Thread 13
Item 3 Task 4 Thread 12
Item 4 Task 3 Thread 17
Item 1 Task 9 Thread 9
Item 10 Task 7 Thread 13
Item 15 Task 5 Thread 15
```

When no partitioning logic fits your needs, the only choice available is writing your own partitioner by extending the `Partitioner<T>` or `OrderablePartitioner<T>` class:

```
class Program{
    static void Main(string[] args){
        //a dataset
        var items = Enumerable.Range(1, 1000).ToArray();
        //my partitioner
        var partitioner = new MyChunkPartitioner(items);
        partitioner.AsParallel().ForAll(item =>{
            Console.WriteLine("Item {0} Task {1} Thread {2}", item,
                Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(2000);
        });
        Console.ReadLine();
    }
}
//only use for testing purposes
public class MyChunkPartitioner : Partitioner<int>{
    //underlying data collection
    public IEnumerable<int> Items { get; private set; }
    public MyChunkPartitioner(IEnumerable<int> items){ Items = items;}
    //partition elaboration
    public override IList<IEnumerator<int>> GetPartitions(
        int partitionCount){
        var result = new List<IEnumerator<int>>();
        //compute the page size in an easy way
        var pageSize = Items.Count() / partitionCount;
        for (int page = 0; page < partitionCount; page++){
            result.Add(Items.Skip(page * pageSize).
                Take(pageSize).GetEnumerator());
        }
        return result;
    }
}
```

Keep in mind that with custom partitioning logic, we have the opportunity to define partition size, and not partition count, because this count is passed as a parameter from outside.

## Summary

In this chapter, you saw how to use task parallelism and data parallelism with the .NET framework's features and techniques. Asynchronous programming and parallelism together give any .NET programmer the ability to reach impressive performance goals with an easy-to use approach and reliable environment.

## Programming for Math and Engineering

This chapter will focus on computation that is mathematical and engineering oriented, such as digital signal filtering, or any other mathematical computation that may apply to any Big Data of (usually) simple items.

A lot of the examples within this chapter will use libraries such as `Math.NET Numerics` or `AForge.Math`. These libraries are available for free through **NuGet Package Manager**.

In this chapter, we will cover the following topics:

- Evaluating the performance of data types
- Real-time applications
- Case study: the Fourier transform
- Sliding processing

### Introduction

Performance impact regarding complex computation is often a primary concern for mathematicians and engineers who deal with C# coding.

Throughput is usually the main performance goal when dealing with scientific data because the faster the application can do its job, the faster the result will be available to the user. This high computational speed improves updating a UI in a higher FPS or processing more asynchronous data for a non-UI application. This always affects the end user considerably.

As opposed to an enterprise world in which big datasets of complex data and logics with simple operations exist, in the mathematical or engineering world, these datasets are usually huge but always of primary types, such as floating-point numbers or sometimes timestamps.

Saving a single millisecond per item when processing 100,000 items per second means saving lot of time, so, fixing usually unintended mistakes or computations that are not optimized has become a main task of any programmer.

### Evaluating the performance of data types

Years ago, when CPUs were very slow, the choice of the type of variable was actually an important choice. With modern managed programming languages, even the most primitive data types perform quite the same, but in some cases specific and different behaviors still exist regarding data type performance, in terms of throughput and resource usage.

The following is a sample application that is useful for checking data type speed in randomly generating and sum ten million data items of primitive values. Obviously, such sample code is not able to give any kind of absolute speed rating. It is a simple demonstration application that is useful for giving an idea of different data-type performance behavior:

```

//a random value generator
var r = new Random();
//repeat 10 times the test to have an averaged value
var stopwatchResults = new List<double>();
//a stopwatch for precision profiling
var w = new Stopwatch();
w.Start();
//change type here for testing another data type
var values = Enumerable.Range(0, 10000000).Select(i => (float)(r.NextDouble()
* 10))
    .ToArray();
w.Stop();
Console.WriteLine("Value array generated in {0:N0}ms",
w.ElapsedMilliseconds);
for (int j = 0; j < 10; j++){
    w.Reset();
    w.Start();
//change type here for testing another data type
    float result = 0;
//sum all values
    foreach (var f in values)
        result += f;
    w.Stop();
    Console.WriteLine("Result generated in {0:N0}ms", w.ElapsedMilliseconds);
    stopwatchResults.Add(w.ElapsedMilliseconds);
}
Console.WriteLine("\r\n-> Result generated in {0:N0}ms avg",
stopwatchResults.Average());
Console.ReadLine();

```

When executed, this application gives an average execution time. Here are some results per data type that were executed on my laptop, which has a quad-core Intel i7-4910MQ, running at 3.9Ghz in turbo mode.



Please note that the following values are only useful in relation to each other and they are not valid speed benchmark results.

Type	Average Result
Int32	37ms
Int16	37ms
Int64	37ms
Double	37ms
Single	37ms
Decimal	328ms

As expected, all data types with a footprint of 32 or 64 bits executed at the same on my 64-bit CPU within the CLR execution environment, but the `Decimal` data type, which is actually a 128 floating-point number, executed almost 10 times slower than all the other data types.

Although in most financial or mathematical computation the increased precision of a decimal is mandatory, this demonstrates how using a lower precision data type will really boost throughput and latency of any of our applications.

The `Decimal` datatype is able to drastically reduce rounding errors that often happen when using standard 64-bit or 32-bit (double or float) floating point data types, because most of the decimal memory footprint is used to increase precision instead of minimum/maximum numeric values. Visit the following link for more details: <https://msdn.microsoft.com/en-us/library/system.decimal.aspx>.

## BigInteger

A special case is when dealing with arbitrary-sized data types, such as the `BigInteger` of the `System.Numerics` namespace (add reference to `System.Numerics` assembly in order to use the related namespace). This structure can handle virtually any numeric signed integer value. The size of the structure in memory will grow together with the internal value numeric size, bringing an always-increasing resource usage and bad throughput times.

Usually, performance is not impacted by the numeric values of two (or multiple) variables when involved in a mathematical computation. This means that computing  $10*10$  or  $10*500$  costs the same with regards to CPU usage. When dealing with arbitrary sized typed variables, this assertion becomes false, because of the increased internal size of the data being computed, which brings a higher CPU usage at each value increase.

Let us see how a `BigInteger` multiplication speed changes with the same multiplier:

Multiplier	Average Result	Difference
10	332ms	
100	348ms	+5%
1000	441ms	+26%
10000	640ms	+45%
100000	658ms	+3%

Things change a lot here. The `BigInteger` numeric structure performs in a similar way to the `Decimal` type, when the number is actually a small value. Compared with other data types, this type always performs worse as the value increases, although only by a small amount, because of the intrinsic implementation of the type that internally contains an arbitrary amount of small integer values that compose the full value. A `BigInteger` type has no bound limitations in the numeric value range.

Compared to a `Decimal` type that has very high precision with a good numeric value range, the `BigInteger` type has no precision (as an integer value) with no value range limitation. This differentiation simply states that we should only use the `BigInteger` type when we definitely need storing and computing calculations against a huge numeric value, possibly with no known upper/lower numeric range boundaries.

This behavior should discourage users from using such a data type, except when it is absolutely necessary. Consider that an arbitrary size numeric structure is hard to persist on any relational database without strong customizations or by using serialization features, with the high costs of lot of data extraction whenever we need to read/write such a value.

## Half-precision data type

Although not available by default within the CLR, when dealing with unmanaged code, often referred to as unsafe code, this old data type becomes available.

A native implementation for .NET is made in this link: <http://csharp-half.sourceforge.net/>.

This implementation, as expected, uses native unsafe code from C++. To enable unsafe coding within C#, we need to select the `Allow unsafe code` flag within the **Build** pane of project property page.

Here, a 16-bit floating-point precision example is given with unsafe coding:

```
var doublePrecision = double.Parse("10.987654321");
Console.WriteLine("{0}", doublePrecision);
var singlePrecision = float.Parse("10.987654321");
Console.WriteLine("{0}", singlePrecision);
var halfPrecision = Half.Parse("10.987654321");
Console.WriteLine("{0}", halfPrecision);
```

```
//result:
10.987654321
10.98765
10.98438
```

When you lose precision by using a smaller data type, you reduce the characteristic digit count (the number of digits different from zero) of the contained value. As seen, the 64-bit example (the first row) clearly shows more digits than the other two types.

Another great disadvantage of using small precision datatypes is that because of the numeric characterization reduction, it may be possible that a specific value won't exist. In this case, the nearest value available is used in place of the original value. This issue also exists in 32-bit and 64-bit floating points. But when working with a very tiny 16-bit value, such as the half-precision data type, this issue becomes more evident.



All floating-point values have a characteristic number and a 10-based multiplier. This means that the more characteristic numbers we use upside the decimal separator, the fewer will remain available downside the decimal separator. And the opposite is also true.

Regarding performance, maybe because of this specific implementation, performances are actually poor with high computation times that become a visible issue on big datasets.

Using a 16-bit floating-point data type is discouraged because it is unable to give any performance improvement, but only in case if we definitely need such a 16-bit data type, that may be because of some legacy-application integration.

## Real-time applications

Real-time computing happens when a system is able to guarantee its latency time, regardless of the system load. Obviously, low latency times are mandatory, but it is the ability to guarantee the same latency time as load increases that makes a fast system actually a real-time system. A more detailed definition is available here: [http://en.wikipedia.org/wiki/Real-time\\_computing](http://en.wikipedia.org/wiki/Real-time_computing).

A canonical example is the ABS (anti-lock braking system) logic ubiquitously implemented in any automobile. Such logic must give results within a specific deadline (in a span of milliseconds), otherwise the system will go into a failed state.

Sometimes real-time systems may run at an acceptable service level, although with soft constraint specifications such as adding some tolerance to the deadline requirement. With such a near real-time requirement, we can code in .NET for Microsoft Windows as easily as we usually do for any other application type.

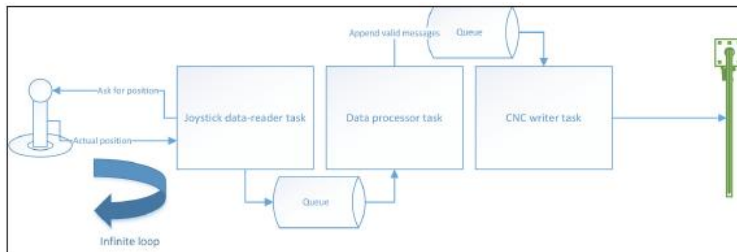
Bear in mind that within Microsoft Windows we cannot have full real-time computation, mainly because of the unavailability of any application to claim the 100 percent time of a CPU that is handled in time-share by the OS itself.



This does not mean that Windows or the CLR cannot run code fast enough for real-time, as real-time programming does not mean fast – it means with deterministic times. Windows cannot run real-time applications, simply because it cannot guarantee specific timing for (system – Win32) method invocations or thread start-up/stop times. The same happens regarding the CLR that cannot guarantee fixed timings about method execution and object destruction, for instance as has already been described in the *Garbage Collection* section in *Chapter CLR Internals*.

When dealing with specific applications such as industrial systems for automations or robotics, it may be that we need near-real-time execution in C# to drive such automations. Although we cannot create a Computer Numerical Control (CNC) with C#, we can drive a CNC with remote invocations made in any .NET language. When driving a CNC, the best performing architecture in C# is made using task-parallelism or multi-thread based design.

If we have to create an application that reads a joystick position value, and then moves a robotic arm to a specific position, we should make at least a three-threaded application that can run at 60 FPS (Frame Per Second). This means that all C# code must execute in less than 16ms per cycle.



A real-time queued driver processor made with C#

The application consists of a task, or a thread in charge of asking the position of the joystick with an infinite-loop in a polling design. No logic can be placed here because if any kind of logic is placed here, it would reduce the read speed, which, in turn, would reduce the overall FPS rates of the application.

Successively, any value will be queued in a valid thread-safe in-memory queue that will propagate the value without having to couple the two tasks processing speeds.

The second task will eventually check for data integrity and avoid data duplication by knowing the actual CNC state and thus avoid sending the same position to the next step multiple times.

The last task will read messages from the previous step by reading another queue and will later send any queued message to the CNC in the right sequence, because the whole queued architecture guarantees the sequential transmission of all messages.

For instance, you can run the real time application and later run a telnet client by executing such command: [telnet localhost 8080]. When the telnet will establish the connection to the real time application we can simply test it by writing some text in the telnet client one. All the text will be sent to the real time application and is later shown in the console.

```
static void Main(string[] args){
//create all needed tasks and wait until any will exit
//any task exit will be considered an error and will cause process exit
    Task.WaitAny(
        Task.Factory.StartNew(
            OnDataReaderTask, TaskCreationOptions.LongRunning),
        Task.Factory.StartNew(
            OnDataProcessorTask, TaskCreationOptions.LongRunning),
        Task.Factory.StartNew(
            OnDataWriterTask, TaskCreationOptions.LongRunning));
}
```

```

        Console.WriteLine("Abnormal exit!");
    }
    //a stopwatch for testing purposes
    static readonly Stopwatch stopwatch = new Stopwatch();
    //this task will read data from the reader source
    //we will use a simple tcp listener for testing purposes
    private static void OnDataReaderTask() {
        //a listener for opening a server TCP port
        var listener = new TcpListener(IPAddress.Any, 8080);
        listener.Start();
        //the server client for communication with remote client
        using (var client = listener.AcceptTcpClient())
        using (var stream = client.GetStream())
        while (true) {
            //try reading next byte
            var nextByte = stream.ReadByte();
            //valid char
            if (nextByte >= 0) {
                AllMessagesQueue.Enqueue((char)nextByte);
            }
            //start stopwatch
            stopwatch.Reset();
            stopwatch.Start();
            Thread.Sleep(1);
        }
    }
    //this queue will contains temporary messages going from reader task to
    processor task
    static readonly ConcurrentQueue<char> AllMessagesQueue = new
    ConcurrentQueue<char>();
    //this task will process data messages
    //no data repetition will be admitted
    private static void OnDataProcessorTask() {
        char last = default(char);
        while (true) {
            char c;
            //if there is some data to read
            if (AllMessagesQueue.TryDequeue(out c))
            //only new values are admitted when sending coordinates to a CNC
            if (c != last) {
                last = c;
                ValidMessagesQueue.Enqueue(c);
            }
            else
            //stop stopwatch
            stopwatch.Stop();
            Thread.Sleep(1);
        }
    }
    //this queue will contains temporary messages going from processor task to
    writer task
    static readonly ConcurrentQueue<char> ValidMessagesQueue = new
    ConcurrentQueue<char>();
    //this task will push data to the target system
    //instead of a CNC we will use the Console for testing purposes
    private static void OnDataWriterTask() {
        while (true) {

```

```

        char c;
//if there is some data to read
        if (ValidMessagesQueue.TryDequeue(out c)) {
//stop stopwatch
            stopwatch.Stop();
            Debug.WriteLine(string.Format("Message crossed tasks in
{0:N0}ms", stopwatch.ElapsedMilliseconds));
//we will send such data to the CNC system
//for testing purposes we will use a Console.Write
                Console.Write(c);
            }
            Thread.Sleep(1);
        }
    }
}

```

The preceding example shows how to process data without ever making any tasks wait for another. This solution will guarantee the message flow and executes in not more than 10ms on my laptop, so its speed is actually higher than 60 FPS, as required.

The usage of `Thread.Sleep` at 1ms will force CLR to pause the execution of the thread. On Windows, this stop-and-resume time is variable.

Obviously, we cannot guarantee that under load, the system will process in the same time, so this is definitely a near-real-time application with a soft constraint on deadlines specification; although optimistically, for 99.99% of the time, it works just fine.

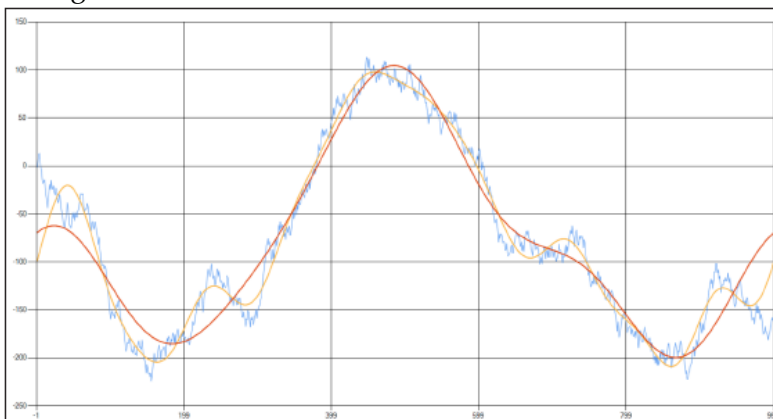
### Case study: Fourier transform

Fourier transform has several usages in engineering programming. The easiest usage is producing a rolling average value for a dataset by applying a digital filter on the given values as being frequency-domain values.

A low-pass filter is the one that stops high frequency values from passing. In audio engineering, it is used to drive a sub-woofer or any low-frequency speaker. When dealing with any other numerical value, such filters become useful to have an averaged value or to cut away any interference or parasite signal in our values.

### Rolling average

The application of a Fast Fourier Transform (FFT) on any numerical value will produce a rolling average result like this:



A rolling average with a FFT at 10hz (orange) and 4hz (red) cut frequency

A typical feature of a FFT filter is at the edges, where the filter follows the trend of the whole dataset instead of the local data. In the preceding picture, this error is visible on the right-hand side, where the FFT produces an increasing averaged value while the raw one is going down.

By using the `Math.NET Numerics` package from NuGet, the following is the code to make a low-pass with FFT:

```

/// <summary>
/// Makes a low-pass digital filter against any floating point data
/// </summary>
private static IEnumerable<float> LowPass(IEnumerable<float> values,
    int cutHz){
//convert raw data to Complex frequency domain-data
    var complex = values.Select(x => new Complex(x, 0)).ToArray();
//start a fast Fourier transform (FFT)
//this will change raw data in frequency data
    Fourier.Forward(complex);
//low data is at edges so we clean-up
//any data at the centre because we want
//only low data (is a low-pass filter)
    for (int i = 0; i < complex.Count(); i++)
        if (i > cutHz && i < complex.Count() - cutHz)
            complex[i] = new Complex();
//convert back data to raw floating-point values
    Fourier.Inverse(complex);
    return complex.Select(x => (float)x.Real);
}

```

The following example shows how to create the preceding chart in Windows Forms. The application starts with an empty `Form1` file.

```

//for data initialization
private void Form1_Load(object sender, EventArgs e){
    var r = new Random();
    double d = 0;
//randomly generated data
    var data = Enumerable.Range(1, 1000)
        .Select(i => (float)(r.Next() % 2 == 0 ? d += (r.NextDouble()
* 10d) : d -= (r.NextDouble() * 10d)))
        .ToArray();
//namespace System.Windows.Forms.DataVisualization.Charting
    var chart1 = new Chart();
//add the chart to the form
    this.Controls.Add(chart1);
//shows chart in full screen
    chart1.Dock = DockStyle.Fill;
//create a default area
    chart1.ChartAreas.Add(new ChartArea());
//create series
    chart1.Series.Add(new Series{
        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "RawValue",
        ChartType = SeriesChartType.Line,
    });
    chart1.Series.Add(new Series{

```

```

        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "AveragedValue10",
        ChartType = SeriesChartType.Line,
        BorderWidth = 2,
    });
    chart1.Series.Add(new Series{
        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "AveragedValue4",
        ChartType = SeriesChartType.Line,
        BorderWidth = 2,
    });
//apply a digital low-pass filter with different cut-off frequencies
    var lowPassData10hz = LowPass(data, 10).ToArray();
    var lowPassData4hz = LowPass(data, 4).ToArray();
//do databinding
    chart1.DataSource = Enumerable.Range(0, data.Length).Select(i => new
    {
        Index = i,
        RawValue = data[i],
        AveragedValue10 = lowPassData10hz[i],
        AveragedValue4 = lowPassData4hz[i],
    }).ToArray();
    chart1.DataBind();
//window in full screen
    WindowState = FormWindowState.Maximized;
}

```

## Low-pass filtering for Audio

Low-pass filtering has been available since 2008 in the native .NET code. **NAudio** is a powerful library helping any CLR programmer to create, manipulate, or analyze audio data in any format.

Available through NuGet Package Manager, NAudio offers a simple and .NET-like programming framework, with specific classes and stream-reader for audio data files.

Let's see how to apply the low-pass digital filter in a real audio uncompressed file in WAVE format. For this test, we will use the Windows start-up default sound file. The chart is still made in a legacy Windows Forms application with an empty `Form1` file, as shown in the previous example.

```

private async void Form1_Load(object sender, EventArgs e){
//stereo wave file channels
    var channels = await Task.Factory.StartNew(() =>{
//the wave stream-like reader
        using (var reader = new WaveFileReader("startup.wav")){
            var leftChannel = new List<float>();
            var rightChannel = new List<float>();
//let's read all frames as normalized floats
            while (reader.Position < reader.Length){
                var frame = reader.ReadNextSampleFrame();
                leftChannel.Add(frame[0]);
                rightChannel.Add(frame[1]);
            }
            return new{
                Left = leftChannel.ToArray(),
                Right = rightChannel.ToArray(),
            };
        }
    });
}

```

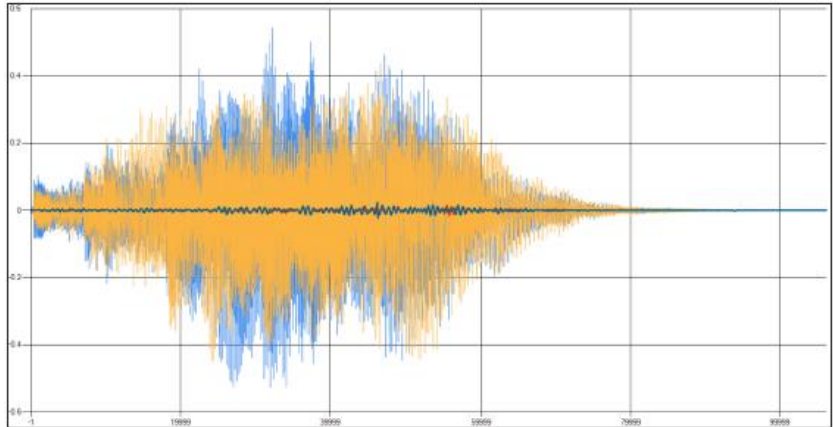
```

        };
    }
});
//make a low-pass digital filter on floating point data
//at 200hz
    var leftLowpassTask = Task.Factory.StartNew(() => LowPass(channels.Left,
200).ToArray());
    var rightLowpassTask = Task.Factory.StartNew(() =>
LowPass(channels.Right, 200).ToArray());
//this let the two tasks work together in task-parallelism
    var leftChannelLP = await leftLowpassTask;
    var rightChannelLP = await rightLowpassTask;
//create and databind a chart
    var chart1 = CreateChart();
    chart1.DataSource = Enumerable.Range(0, channels.Left.Length). Select(i
=> new{
        Index = i,
        Left = channels.Left[i],
        Right = channels.Right[i],
        LeftLP = leftChannelLP[i],
        RightLP = rightChannelLP[i],
    }).ToArray();
    chart1.DataBind();
//add the chart to the form
    this.Controls.Add(chart1);
}
private static Chart CreateChart(){
//creates a chart
//namespace System.Windows.Forms.DataVisualization.Charting
    var chart1 = new Chart();
//shows chart in fullscreen
    chart1.Dock = DockStyle.Fill;
//create a default area
    chart1.ChartAreas.Add(new ChartArea());
//left and right channel series
    chart1.Series.Add(new Series{
        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "Left",
        ChartType = SeriesChartType.Line,
    });
    chart1.Series.Add(new Series{
        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "Right",
        ChartType = SeriesChartType.Line,
    });
//left and right channel low-pass (bass) series
    chart1.Series.Add(new Series{
        XValueMember = "Index",
        XValueType = ChartValueType.Auto,
        YValueMembers = "LeftLP",
        ChartType = SeriesChartType.Line,
        BorderWidth = 2,
    });
    chart1.Series.Add(new Series{
        XValueMember = "Index",

```

```
XValueType = ChartValueType.Auto,  
YValueMembers = "RightLP",  
ChartType = SeriesChartType.Line,  
BorderWidth = 2,  
});  
return chart1;  
}
```

Let's see the graphical result:



The Windows start-up sound waveform. In bolt, the bass waveform with a low-pass filter at 200hz.

The usage of parallelism in elaborations such as this is mandatory. Audio elaboration is a canonical example of engineering data computation because it works on a huge dataset of floating points values. A simple file, such as the preceding one that contains less than 2 seconds of audio sampled at (only) 22,050 Hz, produces an array greater than 40,000 floating points per channel (stereo = 2 channels).

Just to have an idea of how hard processing audio files is, note that an uncompressed CD quality song of 4 minutes sampled at 44,100 samples per second \* 60 (seconds) \* 4 (minutes) will create an array greater than 10 million floating-point items per channel.

Because of the FFT intrinsic logic, any low-pass filtering run must run in a single thread. This means that the only optimization we can apply when running FFT based low-pass filtering is parallelizing in a per channel basis. For most cases, this choice can only bring a 2X throughput improvement, regardless of the processor count of the underlying system.

### Sliding processing

As already seen in *Chapter CLR Internals*, CLR has some limitations in memory management.

Working with engineering data means having to deal with a huge dataset of more than a million records.

Although we can load a simple integer array with millions of items in memory, the same thing will be impossible when the number rises by a lot, or the data type becomes heavier than a simple integer value.

The .NET has a complete enumerator-like execution model that can help us handle a billion items without ever having to deal with all such items in memory, all together. Here is an example on sliding processing:

```

static void Main(string[] args){
//dataset
//this dataset will not be streamed until needed
    var enumerableDataset = RetrieveHugeDataset();
//start using the enumerable
//this will actually start executing code within RetrieveHugeDataset method
    foreach (var item in enumerableDataset)
        if (item % 12 == 0)
            Console.WriteLine("-> {0}", item);
//parallel elaboration is also available
    enumerableDataset
        .AsParallel()
        .Where(x => x % 12 == 0)
        .ForAll(item => Console.WriteLine("-> {0}", item));
    Console.ReadLine();
}
static readonly Random random = new Random();
//return an enumerable cursor to read data in a sliding way
static IEnumerable<int> RetrieveHugeDataset(){
//easy implementation for testing purpose
    for (int i = 0; i < 10000; i++){
//emulate some resource usage
        Thread.Sleep(random.Next(50, 200));
//signal an item available to the enumerator
        yield return random.Next(10, 100000);
    }
}
}

```

Although this is a simple example, the ability to process a huge dataset with data-parallelism, without storing the whole dataset in memory, is often mandatory when dealing with special data elaboration such as what is produced by CNC systems or audio ADCs. When dealing with a high frequency sampler dataset, it is easy to store more than a billion of items. Because dealing with such a huge dataset in memory may easily cause an `OutOfMemoryException` issue, it is easy to see that sliding elaboration is the only design that can avoid memory issues altogether, with the ability to process in a parallel manner.

Keep in mind that LINQ queries against in-memory objects work with exactly the same implementation as the preceding code. Most LINQ methods, such as an altering method, or a filtering method, will internally execute in a sliding way. By executing a LINQ expression against another enumerator, such as our `RetrieveHugeDataset` method, we start a completely new world of programming in which the data flows between enumerator steps without having to be stored somewhere in memory in a fixed-length container.

A canonical example of such sliding elaboration also uses a source or target (or together) a stream-based class, as a `FileStream` or `NetworkStream`. The combination of all such sliding processor classes is infinite and greatly powerful.

## Summary

In this chapter, we discussed classical mathematic and engineering concerns about data processing using practical examples and real-world solutions, such as the Fast Fourier Transform and near-real-time elaboration.