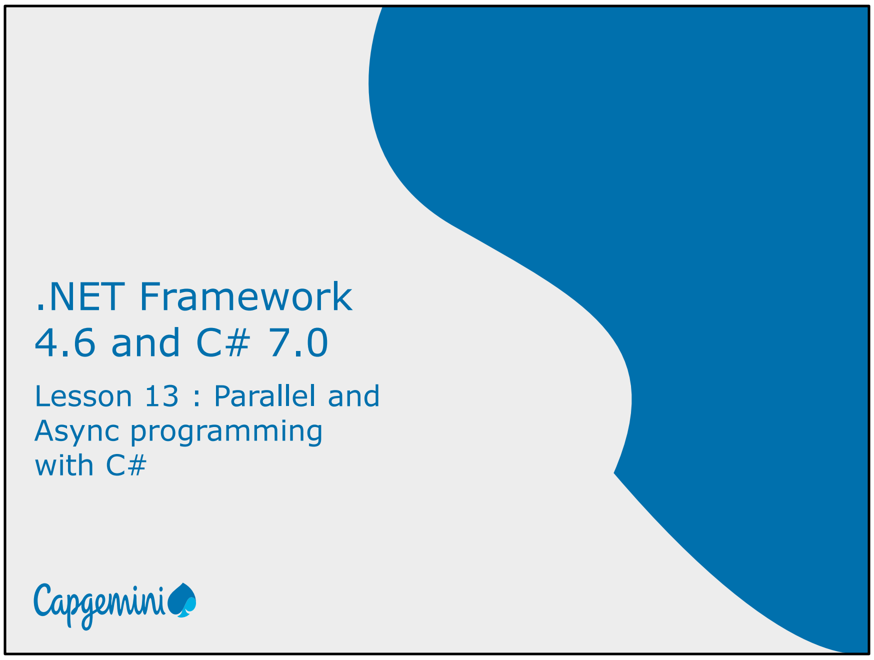


.NET Framework 4.6 and C# 7.0

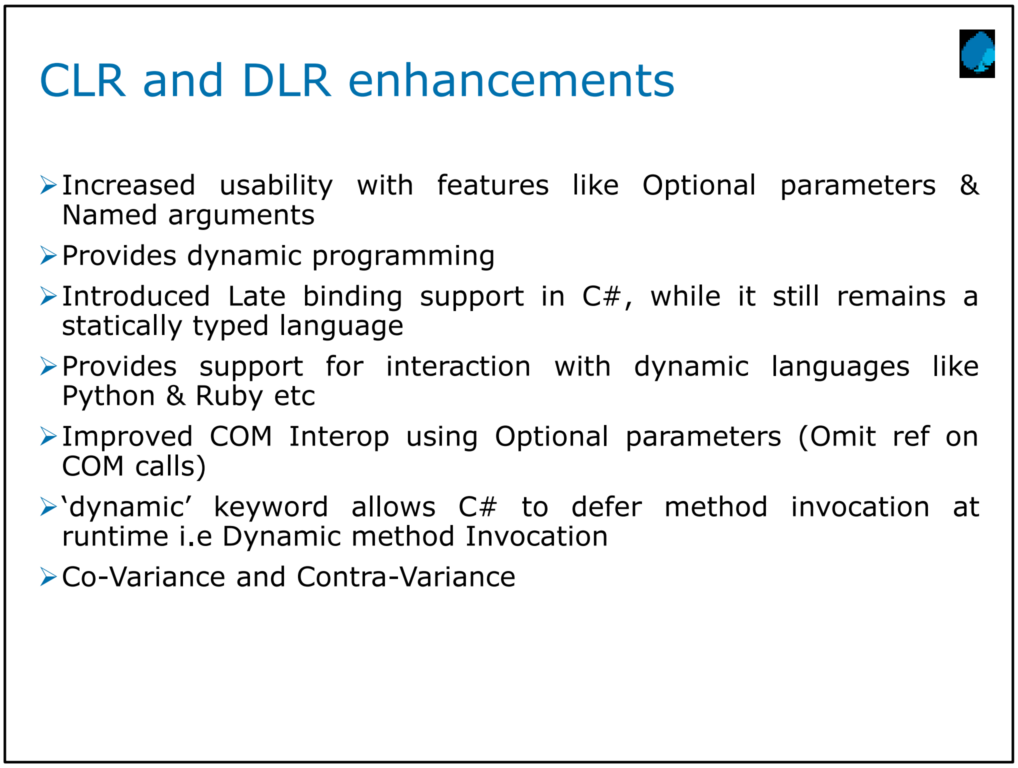
Lesson 13 : Parallel and Async programming with C#

Capgemini

Lesson Objectives

- Parallel Programming
- Async Keyword

Additional notes for instructor



CLR and DLR enhancements

- Increased usability with features like Optional parameters & Named arguments
- Provides dynamic programming
- Introduced Late binding support in C#, while it still remains a statically typed language
- Provides support for interaction with dynamic languages like Python & Ruby etc
- Improved COM Interop using Optional parameters (Omit ref on COM calls)
- ‘dynamic’ keyword allows C# to defer method invocation at runtime i.e Dynamic method Invocation
- Co-Variance and Contra-Variance

The major theme for C# 4.0 is dynamic programming. Increasingly, objects are “dynamic” in the sense that their structure and behavior is not captured by a static type, or at least not one that the compiler knows about when compiling your program. Some examples include :-
Objects from dynamic programming languages, such as Python or Ruby
COM objects accessed through IDispatch
Ordinary .NET types accessed through reflection
Objects with changing structure, such as HTML DOM objects.
While C# still remains a statically typed language, it aims to vastly improve the interaction with such objects.

Co-Variance and Contra-Variance

- .NET Framework has introduced Co-variance and Contra-variance
  - to generic interfaces and delegates
- Suppose there is a class hierarchy, say Employee type and a Manager type that derives from it, then what will the following code do :

```
IEnumerable<Manager> ms = GetManagers();
IEnumerable<Employee> es = ms;
```

- In this example, we are trying to treat the sequence of managers as if they were the sequence of employees, but this will fail in C# 3.0 and the compiler will tell you that there is no conversion.
- In C# 4.0 this assignment works fine, because IEnumerable<T>, along with few other interfaces has changed, due to Co-variance of type parameters.

Covariance :

In .NET 4.0 the IEnumerable<T> and IEnumerator<T>interfaces will be declared in the following way:

```
public interface IEnumerable<out T> : IEnumerable
{
        IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IEnumerator
{
        bool MoveNext();
        T Current { get; }
}
```

The "out" in these declarations signifies that the T can only occur in output position in the interface – the compiler will complain otherwise. In return for this restriction, the interface becomes "covariant" in T, which means that IEnumerable<A> is implicitly reference convertible to IEnumerable<B> if A has an implicit reference conversion to B. As a result, any sequence of strings is also a sequence of objects.

Co-Variance and Contra-Variance

- Covariance means think “out”

  E.g. :

  ```
  public interface IEnumerable<out T> : IEnumerable
  {
          IEnumerator<T> GetEnumerator();
  }
  ```

- Here the ‘out’ keyword, actually modifies the definition of type parameter, T. Compiler sees this and marks type T as covariant
- The out keyword signifies that the generic type parameter, T can appear only in output positions like, as method return value and read-only properties
- Here the interface becomes covariant in ‘T’.
- It means that IEnumerable<A> is implicitly reference convertible to IEnumerable<B>, if A has an implicit reference conversion to B,

Co-Variance and Contra-Variance

- In case of Employee and Manager classes, since there is a relationship between these classes, there is an implicit reference conversion from Manager to Employee :
  - Manager -> Employee
- Also because now type T in IEnumerable is out i.e. IEnumerable<out T>,
- there is also an implicit reference conversion from IEnumerable<Manager> to IEnumerable<Employee> :
  - IEnumerable<Manager> -> IEnumerable<Employee>
- Here these new types convert the same way as the old ones.

Co-Variance and Contra-Variance

- Contra-Variance means think “in”
- In this case, ‘in’ keyword signifies that the generic type parameters can only be used at input positions like, as method parameters and write-only properties
- E.g. :- .NET framework has an interface IComparable<T>, which has a single method called CompareTo :

```
public interface IComparable<in T> {
 bool CompareTo(T other); }
```

- Here the ‘in’ keyword, actually modifies the definition of type parameter, T. Compiler sees this and marks type T as contravariant
- Now, the following code :

```
IComparable<Employee> ec = GetEmployeeComparer();
IComparable<Manager> mc = ec;
```

Contravariance :
Type parameters can also have an “in” modifier, restricting them to occur only in input positions. An example is IComparer<T>:

```
public interface IComparer<in T>
{
        public int Compare(T left, T right);
}
```

The result is that an IComparer<object> can in fact be considered an IComparer<string>. This may be surprising at first, but in fact makes perfect sense: If a comparer can compare any two objects, it can certainly also compare two strings. The interface is said to be “contravariant”.

Co-Variance and Contra-Variance

- Because a manager is an employee, putting manager in works due to contra-variance.
- In case of Employee and Manager classes, since there is a relationship between these classes, there is an implicit reference conversion from Manager to Employee :

    Manager -> Employee

- Also because type T in IComparable is in i.e. IComparable<in T>,
- there is also an reference conversion from
- IComparable<Employee> to IComparable<Manager> :
  - IComparable<Manager> <- IComparable<Employee>

Limitations

- Covariant and contravariant type parameters can be applied only on generic interfaces and delegates.

- Covariance and contravariance only applies when there is a reference (or identity) conversion between type arguments

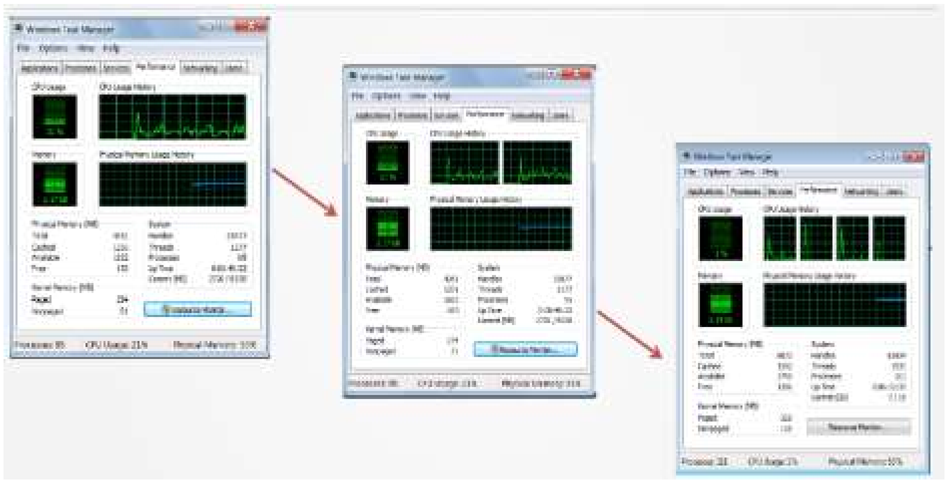
- For instance, an IEnumerable<int> is not an IEnumerable<object> because the conversion from int to object is a boxing conversion, not a reference conversion

Demo

- Covariance and Contravariance

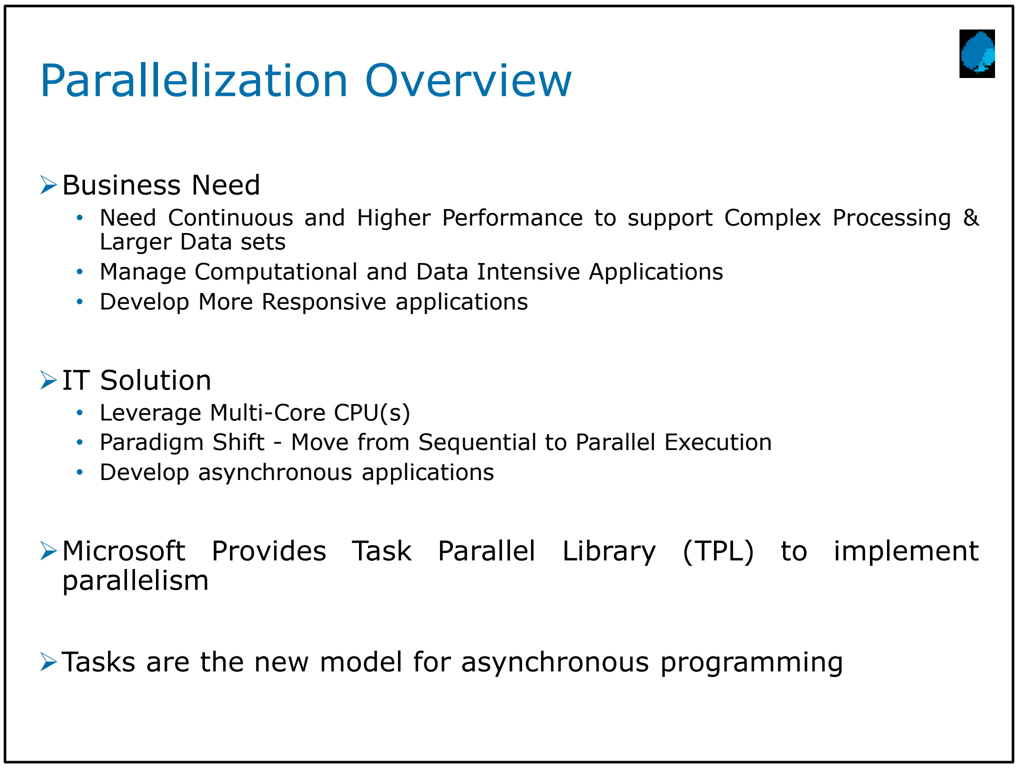
Add the notes here.

Evolution of Processors: The Multi Core Shift



The Multi Core Shift

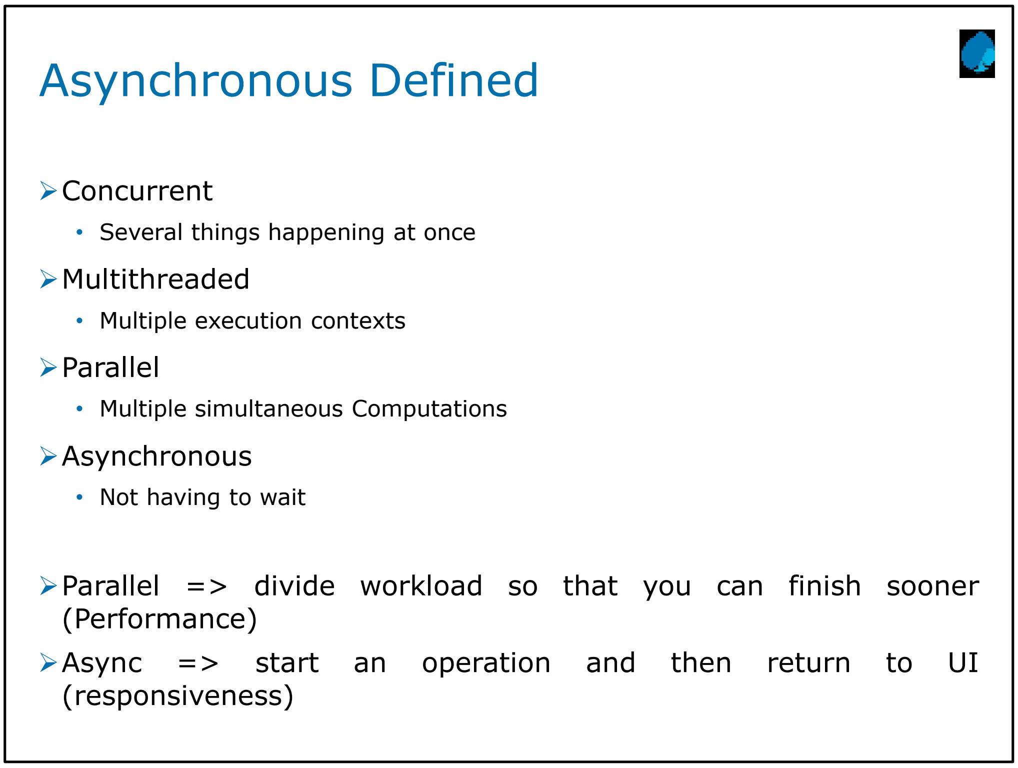
- New model for increasing hardware processing power
- Focus shifted from increasing PC CPU Clock speed
- Overall processing power increased due to additional CPUs, or “cores” on a single chip
- Most desktop and laptop systems now ship with multi-core microprocessors
- Developers should leverage these multi cores in the programs



Parallelization Overview

- Business Need
  - Need Continuous and Higher Performance to support Complex Processing & Larger Data sets
  - Manage Computational and Data Intensive Applications
  - Develop More Responsive applications
- IT Solution
  - Leverage Multi-Core CPU(s)
  - Paradigm Shift - Move from Sequential to Parallel Execution
  - Develop asynchronous applications
- Microsoft Provides Task Parallel Library (TPL) to implement parallelism
- Tasks are the new model for asynchronous programming

In the last 20-30 years we have seen a tremendous growth in processing power where the average speed of the processor has increased year by year. But over the last couple of years we are observing a shift in this trend. Instead of increase in processor speed the number of processors on a single chip are increasing. We are very fast moving from dual core to quad core to eight core processors. So we are gradually shifting towards a multicore processor architecture. But most of our present applications are still of sequential nature and not capable of leveraging the performance benefits offered by multicore processors. This is because parallel/concurrent programming involves threads,locks,mutexes etc. which are difficult to program and also hard to maintain because of lack of tools for profiling and debugging those applications.

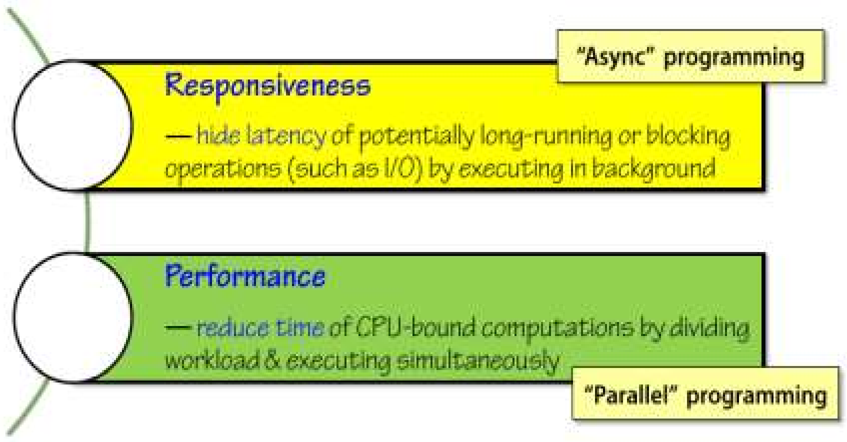


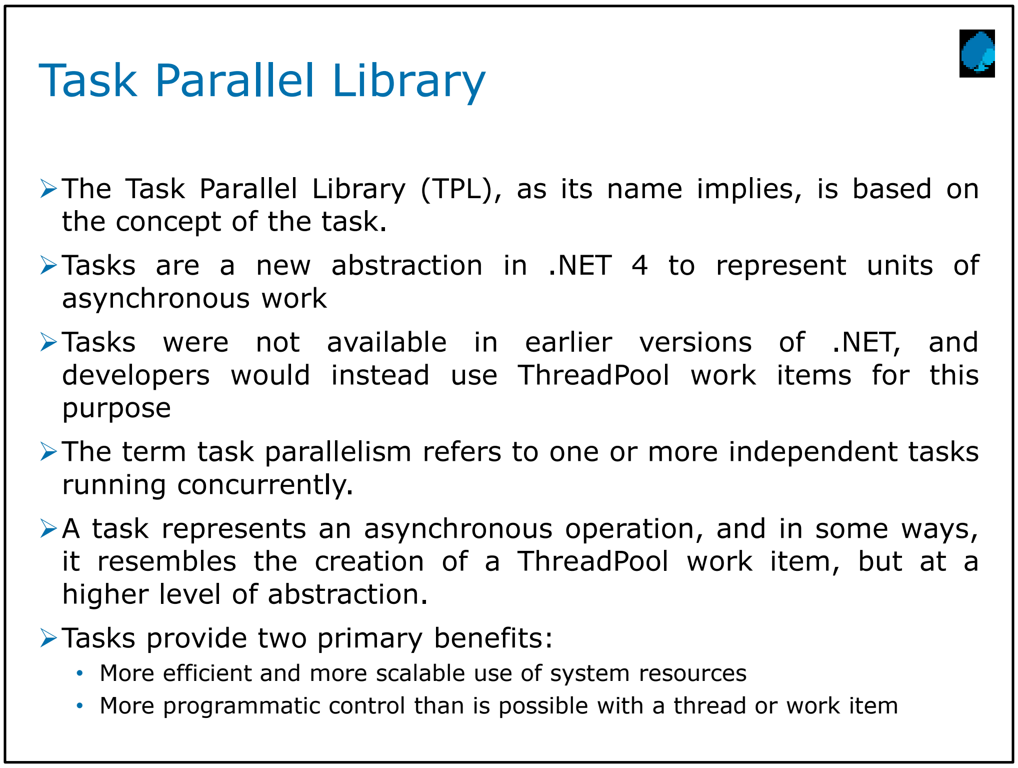
Asynchronous Defined

- Concurrent
  - Several things happening at once
- Multithreaded
  - Multiple execution contexts
- Parallel
  - Multiple simultaneous Computations
- Asynchronous
  - Not having to wait

- Parallel => divide workload so that you can finish sooner (Performance)
- Async => start an operation and then return to UI (responsiveness)

Async and Parallel


"Async" programming
Responsiveness
— hide latency of potentially long-running or blocking operations (such as I/O) by executing in background
Performance
— reduce time of CPU-bound computations by dividing workload & executing simultaneously
"Parallel" programming



Task Parallel Library

- The Task Parallel Library (TPL), as its name implies, is based on the concept of the task.
- Tasks are a new abstraction in .NET 4 to represent units of asynchronous work
- Tasks were not available in earlier versions of .NET, and developers would instead use ThreadPool work items for this purpose
- The term task parallelism refers to one or more independent tasks running concurrently.
- A task represents an asynchronous operation, and in some ways, it resembles the creation of a ThreadPool work item, but at a higher level of abstraction.
- Tasks provide two primary benefits:
  - More efficient and more scalable use of system resources
  - More programmatic control than is possible with a thread or work item

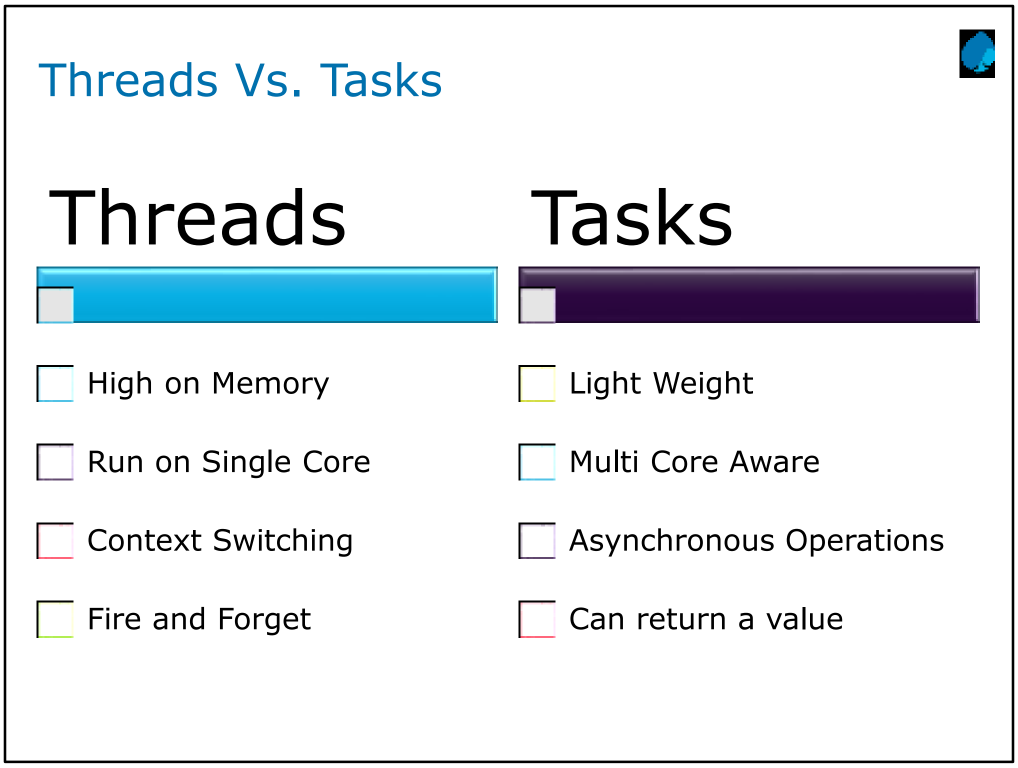
More efficient and more scalable use of system resources.
Behind the scenes, tasks are queued to the ThreadPool, which has been enhanced with algorithms (like hill-climbing) that determine and adjust to the number of threads that maximizes throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism. To complement this, widely-known work-stealing algorithms are employed to provide load-balancing.
More programmatic control than is possible with a thread or work item.
Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

Task Parallel Library

- The Task Parallel Library (TPL) is a set of APIs in the System.Threading and System.Threading.Tasks namespaces in the .NET Framework version 4.
- The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available.
- In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details.
- By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish

Task Parallel Library

- Parallel Extensions in .NET 4.0 provides a set of libraries and tools which
  - Makes concurrent programming easier where developers need not care much details of threads and locks
  - Makes profiling and debugging concurrent applications easier
  - Enables the applications to scale up without any code as number of cores increases
  - Allows existing applications to be easily parallelized

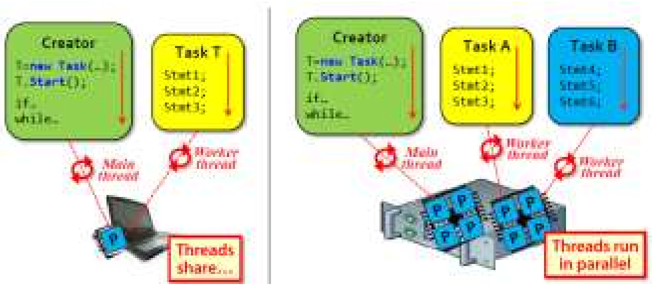


Threads Vs. Tasks

Threads	Tasks
☐ High on Memory	☐ Light Weight
☐ Run on Single Core	☐ Multi Core Aware
☐ Context Switching	☐ Asynchronous Operations
☐ Fire and Forget	☐ Can return a value

Threads Vs. Tasks

- Code based tasks are executed by a thread on some processor
- Thread is dedicated to task until task completes



Parallel Extensions in .NET

- Data Parallelism
  - This refers to dividing the data across multiple processors for parallel execution. e.g. we are processing an array of 1000 elements we can distribute the data between two processors say 500 each.
  - Using Parallel Foreach, For, Invoke, PLINQ
- Task Parallelism
  - This breaks down the program into multiple tasks which can be parallelized and are executed on different processors.
  - This is supported by Task Parallel Library (TPL) in .NET and the Task class
  - Implement the "Task Asynchronous Pattern" TAP

Data Parallelism

- When a parallel loop runs, the TPL partitions the data source so that the loop can operate on multiple parts concurrently.
- Behind the scenes, the Task Scheduler partitions the task based on system resources and workload.

```
// Sequential version
foreach (var item in sourceCollection)
{
    Process(item);
}

// Parallel equivalent
Parallel.ForEach(sourceCollection, item => Process(item));
```

Task Parallelism

- The term task parallelism refers to one or more independent tasks running concurrently.

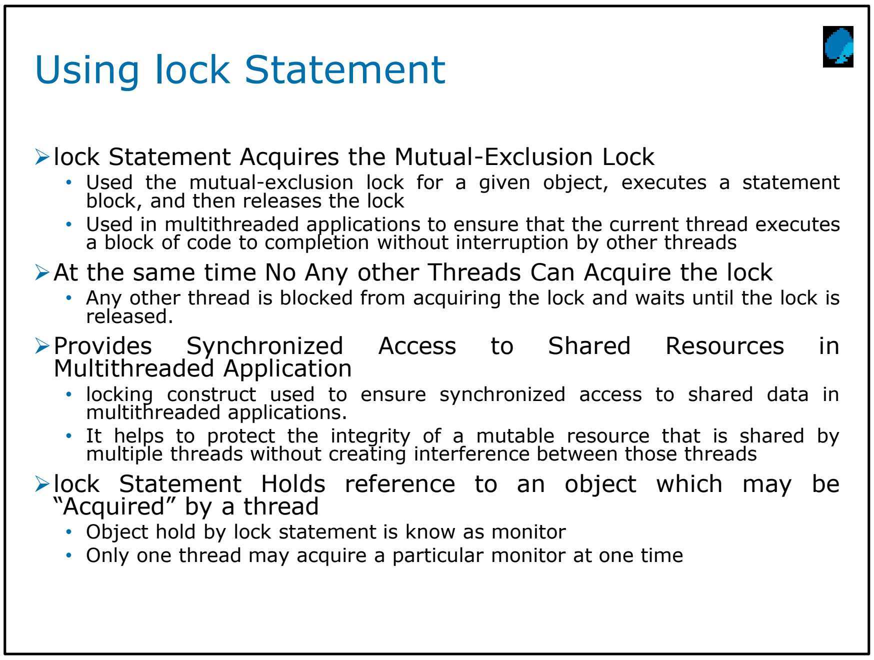
```
 // Create a task and supply a user delegate by using a lambda expression.

var taskA = new Task(() => Console.WriteLine("Hello from taskA."));

// Start the task.
taskA.Start();

// Output a message from the joining thread.
Console.WriteLine("Hello from the calling thread.");


              /* Output:
               * Hello from the joining thread.
               * Hello from taskA.
               */
```



Using lock Statement

- lock Statement Acquires the Mutual-Exclusion Lock
  - Used the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock
  - Used in multithreaded applications to ensure that the current thread executes a block of code to completion without interruption by other threads
- At the same time No Any other Threads Can Acquire the lock
  - Any other thread is blocked from acquiring the lock and waits until the lock is released.
- Provides Synchronized Access to Shared Resources in Multithreaded Application
  - locking construct used to ensure synchronized access to shared data in multithreaded applications.
  - It helps to protect the integrity of a mutable resource that is shared by multiple threads without creating interference between those threads
- lock Statement Holds reference to an object which may be "Acquired" by a thread
  - Object hold by lock statement is know as monitor
  - Only one thread may acquire a particular monitor at one time

The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

The lock statement is an exclusive locking construct used to ensure synchronized access to shared data in multithreaded applications. It helps to protect the integrity of a mutable resource that is shared by multiple threads without creating interference between those threads. The lock statement can be used by a singleton object to prevent concurrent access of its common data by multiple clients.

A lock statement takes a reference to an object which may be “acquired” by a thread; such an object is called a “monitor”. Only one thread may acquire a particular monitor at one time. If a second thread attempts to acquire a particular monitor while a first thread is holding it, and it will wait, block, until the object is released. Once the execution is completed it releases the lock and frees objects.

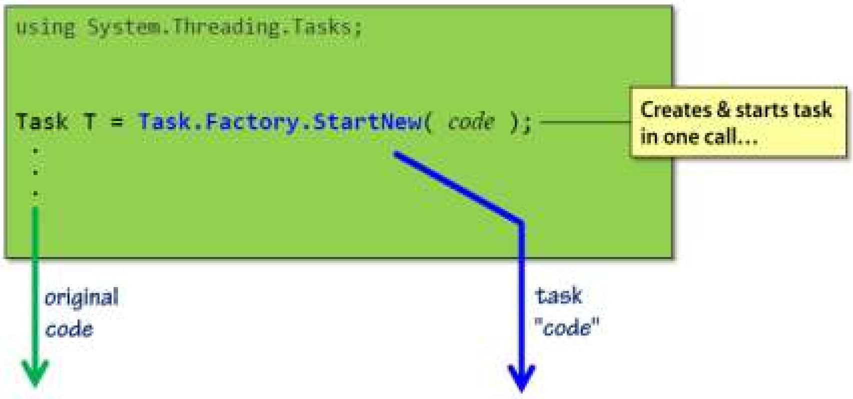
You can't use the await keyword in the body of a lock statement.

Demo

- Using lock Statement

Creating code Tasks – preferred approach

➢ More efficient way..

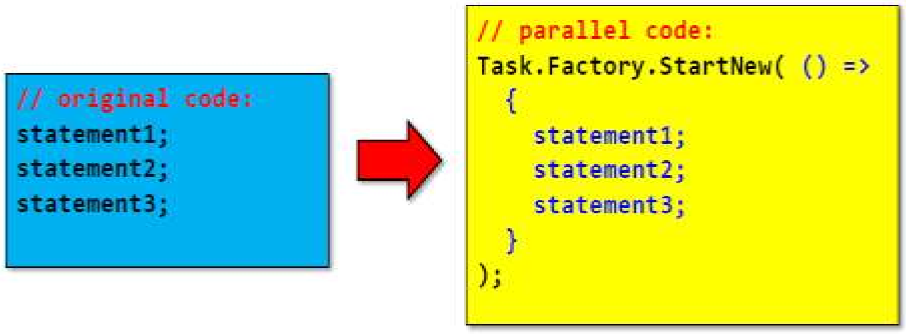


```
using System.Threading.Tasks;


Task T = Task.Factory.StartNew( code );
```

Lambda Expressions

- TPL accepts lambda expressions as parameters
  - Making it easy to create tasks..



```
// original code:
statement1;
statement2;
statement3;
```

```
// parallel code:
Task.Factory.StartNew( () =>
  {
    statement1;
    statement2;
    statement3;
  }
);
```

Parallel LINQ - PLINQ

- PLINQ provides a parallel implementation of LINQ and supports all the standard query operators.
- This is achieved by the System.Linq.ParallelEnumerable class which provides a set of extension methods on IEnumerable interface.
- Developers can opt-in for parallelism by invoking the AsParallel method of ParallelEnumerable class on the data source.
- Using PLINQ we can combine sequential and parallel queries

Task Based Asynchronous Model in .NET 4.5

- We can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming.
- However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.
- C# 5.0 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime

Async and Await - .NET 4.5

- Async programming is recommended in the following scenarios:
  - To keep your application responsive while waiting for something that is likely to take a long time to complete
  - To make your application as scalable as possible (particularly important for services and ASP.NET that have limited threads/resources to service requests)
- However, not all problems are suited to be run asynchronously
- Consider using async functionality in the following situations
  - You are waiting for a response from a remote service
  - The task you want to perform is computationally expensive and doesn't complete very quickly

Async

- async: This modifier indicates the method is now asynchronous.
- It provides a simpler way to perform potentially long-running operations without blocking the callers thread.
- The caller of this async method can resume its work without waiting for this asynchronous method to finish its job
- Reduces developer’s efforts for writing an additional code
- The method with async modifier has at least one await
- This method now runs synchronously until the first await expression written in it

Await

- await: Typically, when a developer writes some code to perform an asynchronous operations using threads, then he/she has to explicitly write necessary code to perform wait operations to complete a task.
- In asynchronous methods, every operation which is getting performed is called a Task.
- The ‘await’ keyword is applied on the task in an asynchronous method and suspends the execution of the method, until the awaited task is not completed.
- During this time, the control is returned back to the caller of this asynchronous method

Await

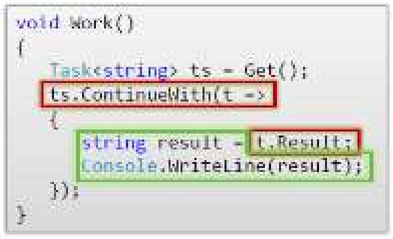
- Most important behavior of the await keyword is it does not block the thread on which it is executing.

- What it does is that it signals to the compiler to continue with other async methods, till the task is in an await state.

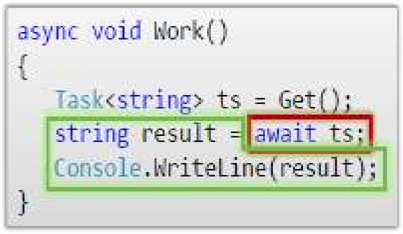
- Once the task is completed, it resumes back where it left off from.

.NET 4.5 async and await

- Async
  - Enables use of asynchronous features
  - Applied to methods
- Await



```
void Work()
{
    Task<string> ts = Get();
    ts.ContinueWith(t =>
    {
        string result = t.Result;
        Console.WriteLine(result);
    });
}
```

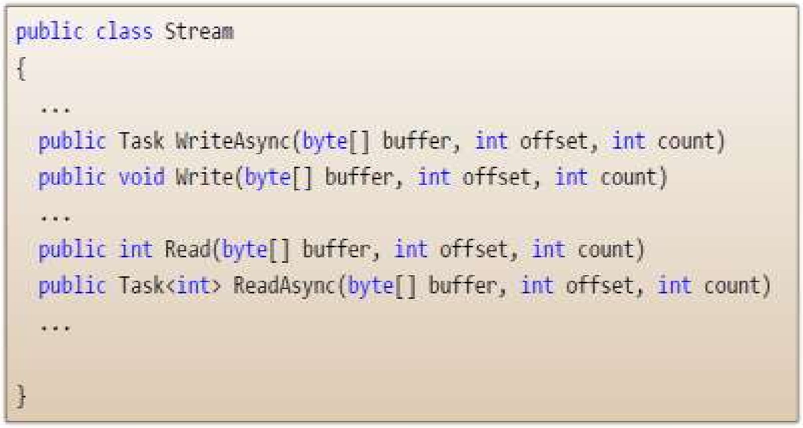


```
async void Work()
{
    Task<string> ts = Get();
    string result = await ts;
    Console.WriteLine(result);
}
```

Async Rules

- The compiler will give you a warning for methods that contain async modifier but no await operator
- A method that isn’t using the await operator will be run synchronously
- Microsoft suggests a convention of post-fixing an Async method with the words Async e.g. DoSomethingAsync
- Async methods can return any of the following:
  - **Void**
  - **Task**
  - **Task<T>**
- A number of BCL methods now have an Async version for you to use with the await keyword

The BCL Support for Async in .NET 4.5



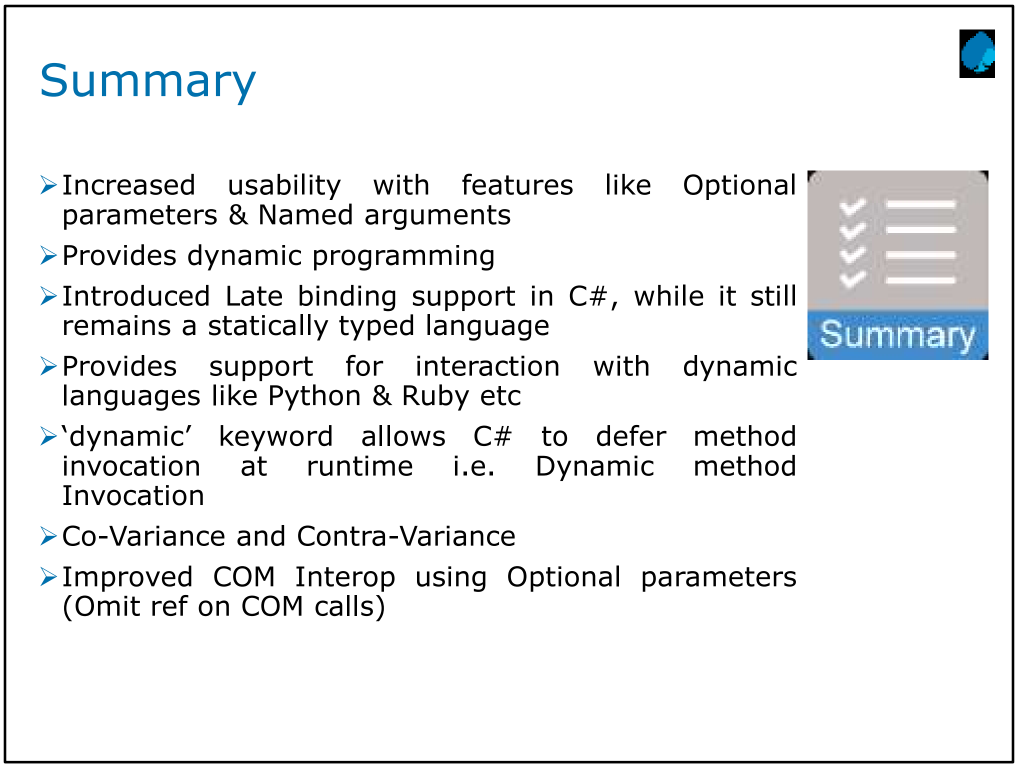
```
public class Stream
{
  ...
  public Task WriteAsync(byte[] buffer, int offset, int count)
  public void Write(byte[] buffer, int offset, int count)
  ...
  public int Read(byte[] buffer, int offset, int count)
  public Task<int> ReadAsync(byte[] buffer, int offset, int count)
  ...

}
```

Task Based Asynchronous Model

```
async Task<string> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();
    Task<string> getStringTask =
    client.GetStringAsync("http://msdn.microsoft.com");
    DoAnyOtherIndependentWork();
    string urlContents = await getStringTask;
    return urlContents;
}
```

Demo



Summary

- Increased usability with features like Optional parameters & Named arguments
- Provides dynamic programming
- Introduced Late binding support in C#, while it still remains a statically typed language
- Provides support for interaction with dynamic languages like Python & Ruby etc
- ‘dynamic’ keyword allows C# to defer method invocation at runtime i.e. Dynamic method Invocation
- Co-Variance and Contra-Variance
- Improved COM Interop using Optional parameters (Omit ref on COM calls)

Add the notes here.

Answers for the
Review Questions:

Q1) Omit REF
Q2) True
Q 3) Type Equivalenc
(Embed Interop Type

Review Question

- Which feature of C# automatically generates temporary variables to pass them as reference to COM methods?
  - Linking of PIA
  - Omit REF
  - Indexed Properties
- C# 4.0 Compiler offers to embed definitions inside PIA in your application assembly
  - True/False
- ________ offers No-PIA option so that entire PIA need not be deployed along with your application setup.

Add the notes here.