

.NET Framework & C# 5.0

Lesson 11: Parallel and Async programming with C#

Lesson Objectives

- Dynamic types
- Parallel Programming
- Async Keyword



CLR and DLR enhancements

- Optional Parameters and Named Arguments
- Dynamic type
 - Dynamic Programming
- Co-Variance and Contra-Variance
 - Support for Generic interfaces and delegates
 - Limitations



Copyright © Capgemini 2015. All Rights Reserved 3

Add the notes here.

Async Programming

- Async Feature
- Caller Information



Copyright © Capgemini 2015. All Rights Reserved 4

Add the notes here.

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



Copyright © Capgemini 2015. All Rights Reserved 5

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.

Optional Parameters

- Optional parameters allows you to omit arguments in method invocation
- They become very handy in cases where parameter list is too long . E.g. : In case of COM method calls
- A parameter is declared optional simply by providing a default value for it
 - E.g. : public void DoSomething(int x, int y = 5, int z = 10)

Here y and z are optional parameters and can be omitted in calls:

- DoSomething(1, 2, 3); //ordinary call to DoSomething
- DoSomething(1, 2); //omitting z, equivalent to DoSomething (1, 2, 10);
- DoSomething(1); // omitting y and z, equivalent to DoSomething (1, 5,10)



Copyright © Capgemini 2015. All Rights Reserved 6

Add the notes here.

Optional Parameters

- C# does not permit you to omit arguments between commas as in DoSomething (1, , 3). This could lead to highly unreadable code
- Instead any argument can be passed by name.



Copyright © Capgemini 2015. All Rights Reserved 7

Add the notes here.

Named Arguments

- Named arguments is a way to provide an argument using its parameter name, instead of relying on its position in the argument list
- Named arguments act as a in-code documentation to help you remember which parameter is which.
- A Named argument is declared simply by providing the name before argument value
 - E.g.: DoSomething (1, y:100, z:200) //valid named arguments
 - DoSomething (1, z:200, y:100) //valid named arguments
 - DoSomething (1, y:100, 25) //invalid way, named arguments must appear after all positional parameters
- An argument with argument-name is a named argument, An argument without an argument-name is a positional argument.



Copyright © Capgemini 2015. All Rights Reserved 8

Add the notes here.

Dynamic keyword (dynamic type)

- C# 4.0 supports late-binding using a new keyword called 'dynamic'.
- The type 'dynamic' can be thought of like a special version of type 'object', which signals that the object can be used dynamically
- C# provides access to new DLR (Dynamic language runtime) through this new dynamic keyword
- When dynamic keyword is encountered in the code, compiler will understand this is a dynamic invocation & not the typical static invocation
 - E.g. : `dynamic d = GetDynamicObject();
d.Add(5);`
 - Here d is declared as dynamic, so C# allows you to call method with any name & any parameters on d. Thus in short the compiler defers its job of resolving type/method names to the runtime



Copyright © Capgemini 2015. All Rights Reserved 9

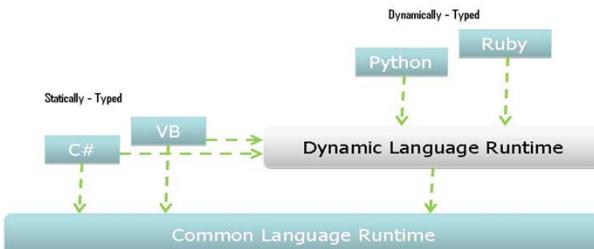
C# now supports late-binding. C# has always been strongly typed & continues to be in version 4.0 as well. But Microsoft thought providing the concept of dynamic binding in C#. It would make life easier for C# developers while they communicate with systems using COM objects or other non .NET based objects.

C# 4.0 introduces a new static type named dynamic. This resolves issues with type binding especially if the receiving object is a non .net based object. Here whatever you do with a dynamic type will take effect only at runtime i.e. it will be resolved at runtime.

E.g. : `dynamic d = GetDynamicObject();
d.Add(5);`

C# allows you to call a method with any name and any arguments on d because it is of type dynamic. At runtime the actual object that d refers to will be examined to determine what it means to "call Add with an int" on it. This approach using dynamic type is much similar to using object type, but there are subtle differences between the two.

C# and DLR (dynamic language runtime)



- The infrastructure that supports ‘dynamic’ operations is called as Dynamic Language Runtime
- This new library introduced with .NET framework 4.0, for each dynamic operation acts as a broker between the language which initiated it and the object it occurs on



Copyright © Capgemini 2015. All Rights Reserved 10

Dynamic binding provides a unified approach to selecting operations dynamically. With dynamic binding developer does not need to worry about whether a given object comes from e.g. COM, IronPython, the HTML DOM or reflection; operations can uniformly be applied to it and the runtime will determine what those operations mean for that particular object.

This affords enormous flexibility, and can greatly simplify the source code, but it does come with a significant drawback: Static typing is not maintained for these operations. A dynamic object is assumed at compile time to support any operation, and only at runtime will an error occur if it was not so.

Dynamic Programming

E.g. - : dynamic d = new MyDynamicObject();
d.Add(5);

By declaring d to be of type dynamic, the code that consumes the MyDynamicObject instance effectively opts out of compile-time checking for the operations d participates in. Use of dynamic means “I don’t know what type this is going to be, so I don’t know what methods or properties there are right now. Compiler, please let them all through and then figure it out when you really have an object at run time.” So the call to Add compiles even though the compiler doesn’t know what it means. Then at run time, the object itself is asked what to do with this call to Add.

C# and DLR (dynamic language runtime)

- If the dynamic operation is not handled by the object it occurs on, a runtime component of the C# compiler handles the bind



Copyright © Capgemini 2015. All Rights Reserved 11

Dynamic binding provides a unified approach to selecting operations dynamically. With dynamic binding developer does not need to worry about whether a given object comes from e.g. COM, IronPython, the HTML DOM or reflection; operations can uniformly be applied to it and the runtime will determine what those operations mean for that particular object.

This affords enormous flexibility, and can greatly simplify the source code, but it does come with a significant drawback: Static typing is not maintained for these operations. A dynamic object is assumed at compile time to support any operation, and only at runtime will an error occur if it was not so.

Dynamic Programming

```
E.g. : dynamic d = new MyDynamicObject();
        d.Add(5);
```

By declaring d to be of type dynamic, the code that consumes the MyDynamicObject instance effectively opts out of compile-time checking for the operations d participates in. Use of dynamic means “I don’t know what type this is going to be, so I don’t know what methods or properties there are right now. Compiler, please let them all through and then figure it out when you really have an object at run time.” So the call to Add compiles even though the compiler doesn’t know what it means. Then at run time, the object itself is asked what to do with this call to Add.

Demo

- Optional Parameters and Named arguments
- Dynamic Programming in C# 5.0



Copyright © Capgemini 2015. All Rights Reserved 12

Add the notes here.

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.



Copyright © Capgemini 2015. All Rights Reserved 13

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` 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
{
    I 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`, if A has an implicit reference conversion to B,



Copyright © Capgemini 2015. All Rights Reserved 14

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.



Copyright © Capgemini 2015. All Rights Reserved 15

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;
```



Copyright © Capgemini 2015. All Rights Reserved 16

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>



Copyright © Capgemini 2015. All Rights Reserved 17

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



Copyright © Capgemini 2015. All Rights Reserved 18

Demo

- Covariance and Contravariance



Copyright © Capgemini 2015. All Rights Reserved 19

Add the notes here.

Simplified COM calls using Optional parameters

- Interoperability with COM on Microsoft .NET Platform from C# has been a daunting task.
- But with the invent of Optional and Named arguments, developers job has become bit easy and greatly improve the experience of interoperating with COM APIs.



Copyright © Capgemini 2015. All Rights Reserved 20

Simplified COM Calls with Optional Parameters and Named Arguments

Interoperability with COM on Microsoft .Net Platform from C# has been a daunting task. But with the invent of Optional and Named arguments, developers job has become bit easy and greatly improve the experience of interoperating with COM APIs.

Omit REF keyword at COM call sites

- COM uses a different programming model where it uses a lot of reference parameters to attain performance benefit.
- Actually, a COM method call will not modify its parameters even when they are passed by reference.
- So declaring temporary variables and passing them as COM arguments seems to be unnecessary.
- In COM, compiler allows to declare method call passing the arguments by value
- And it automatically generates the necessary temporary variables to hold the values in order to pass them by reference.
- It will also discard their values after the call returns



Copyright © Capgemini 2015. All Rights Reserved 21

Omitting ref

Since COM uses a different programming model, it usually uses a lot of reference parameters because of a perceived performance benefit. Actually in common case, a COM method call will not modify its parameters even when passed by reference. In this case, it seems unnecessary for a caller to declare temporary variables for all these arguments and pass them by reference. Specifically for COM methods, the compiler allows to declare the method call passing the arguments by value and will automatically generate the necessary temporary variables to hold the values in order to pass them by reference and will discard their values after the call returns.

For e.g.:

```
object filename = "Test.docx";
object missing = Missing.Value;
document.SaveAs(ref.filename,
    ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing)
```

missing, ref missing,
ref missing, ref missing, ref missing);
Can now be written like this :

```
document.SaveAs("Test.docx",
```

```
    Missing.Value, Missing.Value, Missing.Value,
```

```
    Missing.Value, Missing.Value, Missing.Value);
```

Omitting REF: Example

- Example :

Specify Syntax Here

```
Word.Document document = new Word.Document();
object filename = "Test.docx";
object missing = Missing.Value;
document.SaveAs(ref filename,
                ref missing, ref missing, ref missing,
                ref missing, ref missing);
```

Can now be written as :

```
document.SaveAs(fileName : "Test.docx");
```



Copyright © Capgemini 2015. All Rights Reserved 22

Add the notes here.

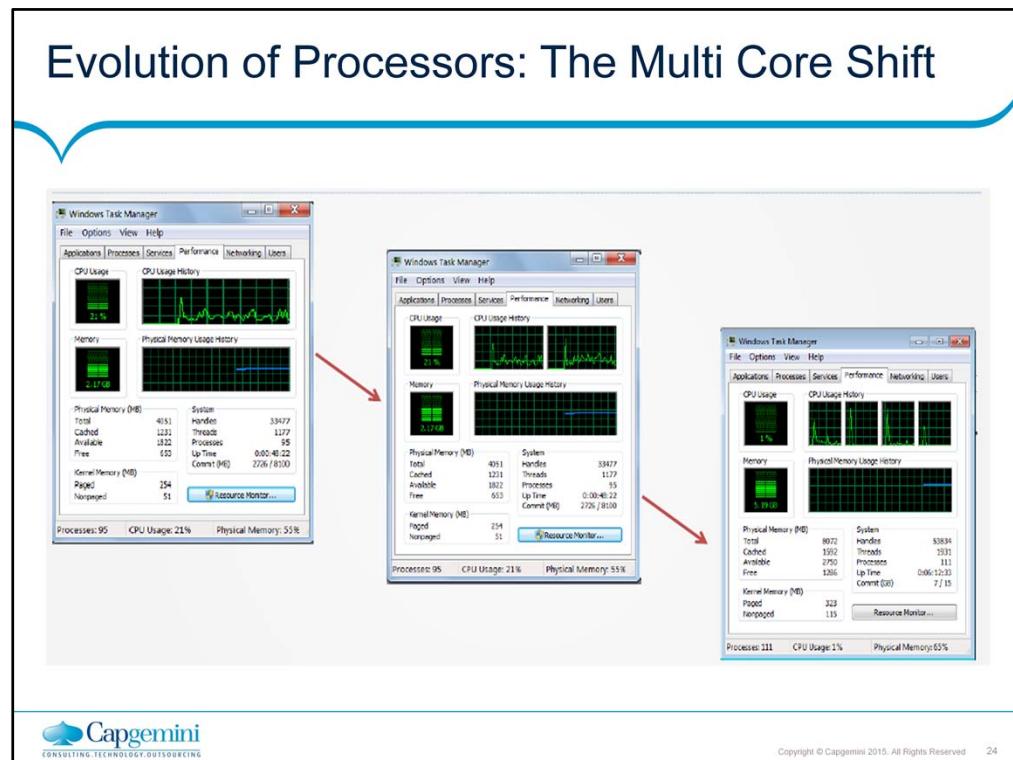
Demo

- Simplified COM Calls using Optional Parameters
- Omitting REF with COM calls



Copyright © Capgemini 2015. All Rights Reserved 23

Add the notes here.



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



Copyright © Capgemini 2015. All Rights Reserved 25

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



Copyright © Capgemini 2015. All Rights Reserved 26

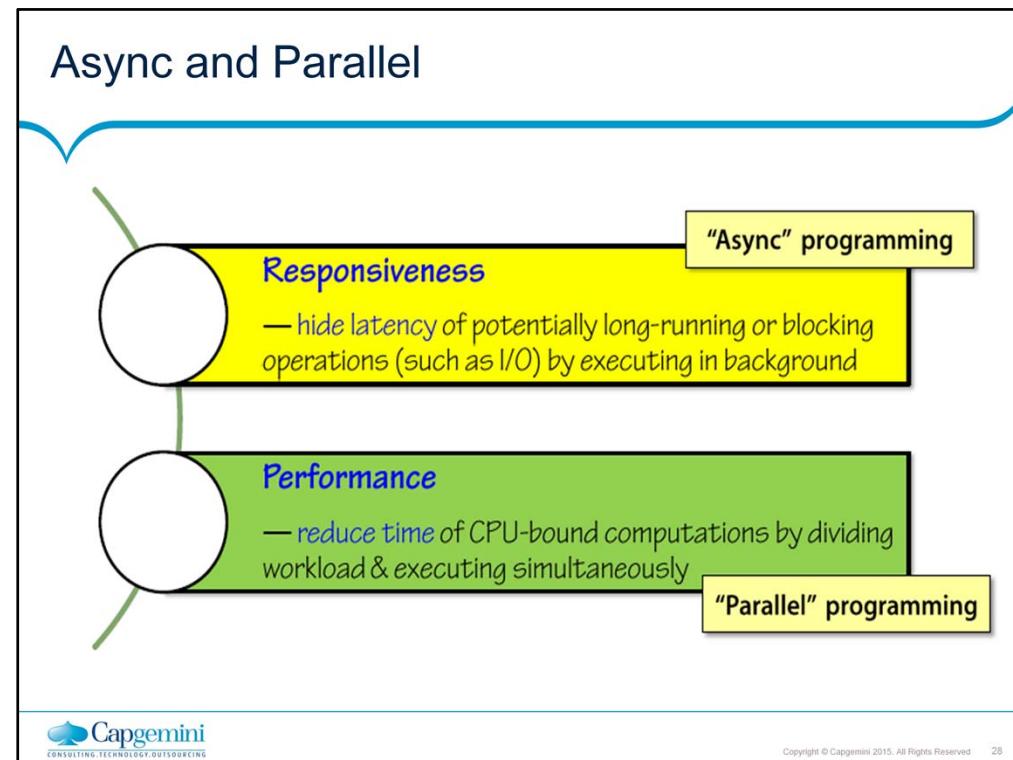
In the last 20-30 years we have seen a tremendous growth in processing power where the average speed of the processor has increased year by year. But over the last couple of years we are observing a shift in this trend. Instead of increase in processor speed the number of processors on a single chip 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)



Copyright © Capgemini 2015. All Rights Reserved 27



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**



Copyright © Capgemini 2015. All Rights Reserved 29

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 Thread Pool, 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



Copyright © Capgemini 2015. All Rights Reserved 30

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**



Copyright © Capgemini 2015. All Rights Reserved 31

Threads Vs. Tasks

Threads

High on Memory
Run on Single Core
Context Switching
Fire and Forget

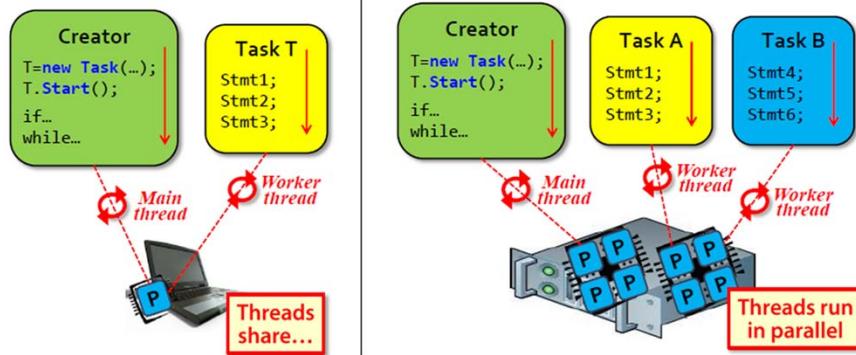
Tasks

Light Weight
Multi Core Aware
Asynchronous
Operations
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



Copyright © Capgemini 2015. All Rights Reserved 34

Data Parallelism

```
// Sequential version  
foreach (var item in sourceCollection)  
{  
    Process(item);  
}  
  
// Parallel equivalent  
Parallel.ForEach(sourceCollection, item => Process(item));
```

- 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.



Copyright © Capgemini 2015. All Rights Reserved 35

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.  
 */
```



Copyright © Capgemini 2015. All Rights Reserved 36

Creating code Tasks – preferred approach

- More efficient way..

```
using System.Threading.Tasks;
```

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

```
.
```

```
.
```

original
code

Creates & starts task
in one call...

task
"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;  
});
```



Copyright © Capgemini 2015. All Rights Reserved 38

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 I Enumerable 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



Copyright © Capgemini 2015. All Rights Reserved 39

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 introduces a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and the Windows Runtime



Copyright © Capgemini 2015. All Rights Reserved 40

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



Copyright © Capgemini 2015. All Rights Reserved 41

Async

- **async:** This modifier indicates the method is now asynchronous.
- The benefit of using this keyword is that 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.
- This 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



Copyright © Capgemini 2015. All Rights Reserved 42

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



Copyright © Capgemini 2015. All Rights Reserved 43

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.



Copyright © Capgemini 2015. All Rights Reserved 44

.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



Copyright © Capgemini 2015. All Rights Reserved 46

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)
    ...
}
```



Copyright © Capgemini 2015. All Rights Reserved 47

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



Copyright © Capgemini 2015. All Rights Reserved 49

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)



Copyright © Capgemini 2015. All Rights Reserved 50

Add the notes here.

Review Question

- Question 1: Which feature of C# automatically generates temporary variables to pass them as reference to COM methods?
- Linking of PIA
- Omit REF
- Indexed Properties
- Question 2: C# 4.0 Compiler offers to embed definitions inside PIA in your application assembly
- True/False
- Question 3: _____ offers No-PIA option so that entire PIA need not be deployed along with your application setup.



Copyright © Capgemini 2015. All Rights Reserved 51

Add the notes here.