Session: 17

More Features of C# 7.0 and 7.1

Objectives

- Describe `ref` returns and `ref` locals
- Explain improvements made to `out` variables, tuples, asynchronous `Main()`, and `throw` expressions
- Identify the new expression-bodied members

Additional Features of C# 7.0 and 7.1

Key new features introduced in C# 7.0 and 7.1

- Choosing the preferred language version
- Ref returns, ref locals, and improved out variables
- Improved tuples
- Improved asynchronous Main()
- Improved throw expressions
- More expression-bodied members

`ref` Returns, `ref` Locals, and Improved `out` Variables

In C#, a method can contain a parameter that a developer can pass by reference using the `ref` keyword.

It is compulsory to specify the `ref` keyword in the calling method as well as in the definition of the called method.

`ref` Returns

- Following is a sample code snippet which demonstrates how the `ref` keyword stores values passed by reference in local variables:

Snippet

```
class Program
   {
     static void Main(string[] args)
     {
        string[]  writers = {"Emy George", "Lee Mein", "John Wash",
        "Sicily Wang"};
      ref string writer2 = ref new Program().FindWriter(1, writers);
      Console.WriteLine("Original writer:{0}", writer2);Console.WriteLine();
      writer2 = "Johan Muller";
      Console.WriteLine("Replaced writer:{0}", writers[1]);Console.ReadKey();
      }
      public ref string FindWriter(int num, string[] names)
      {
          if (names.Length > 0)
            return ref names[num];
            throw new IndexOutOfRangeException($"{nameof(num)}unavailable.");
      }
  }
```

`ref` Locals

- Following code snippet demonstrates that it is possible to save a reference in a local variable:

```
class Program {
   public static object ReturnChars { get; private set; }
   static void Main(string[] args)
    {
     Console.WriteLine("Input a string");
     char[] cseq = Console.ReadLine().ToCharArray();
     Console.WriteLine($"Prior to replacing: { new string(cseq)}");
     ref char cref = ref RetRefLocal.SeekCharRef(cseq[0],cseq);
     cref = 'p';
     Console.WriteLine($"Post replacing: {new string (cseq)}");
     Console.ReadLine();
}
class RetRefLocal {
    public static ref char SeekCharRef(char val, char[] cSeq)
    {
        for (int k = 0; k < cSeq.Length; k++)
        {
            if (cSeq[k] == val)
            {
                return ref cSeq[k];
            }
        }
        throw new IndexOutOfRangeException(val + "not there");
    }
   }
}
```

Improved out Variables and Discards

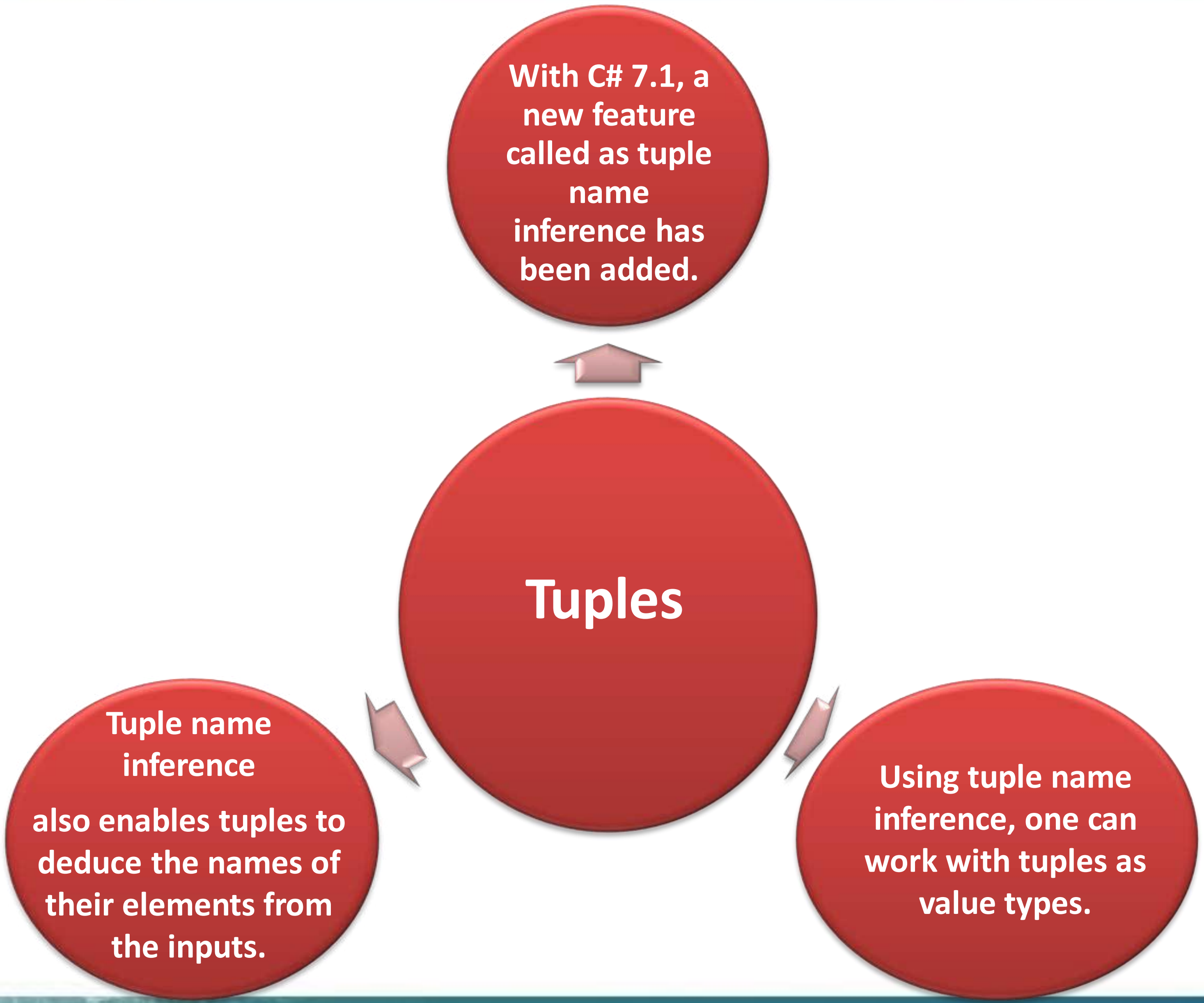
- Following code snippet demonstrates how developers can specify the data type of all the `out` parameters inline:

Snippet

```
class BookApplication
{
 static void Main(string[] args)
 {
   BookByOutArg(out string bName, out string bAuthor);
  Console.WriteLine("Book: {0}, Author: {1}", bName, bAuthor);
  Console.ReadKey();
 }

 static void BookByOutArg(out string name, out string author)
 {
  name = "Harry Potter Part I ";
 author = "J. K. Rowling";
 }
}
```

Improved Tuples (1-3)



Tuples

With C# 7.1, a new feature called as tuple name inference has been added.

Tuple name inference

also enables tuples to deduce the names of their elements from the inputs.

Using tuple name inference, one can work with tuples as value types.

Improved Tuples (2-3)

- Following code snippet shows how two elements of a tuple have been defined without any name inference:

Snippet

```
class Program   {
    static void Main()
    {
     string ename="Emy George";
     int e_age = 30;
     var empTuple = (ename, e_age);
     Console.WriteLine(empTuple.Item1); //Emy George
     Console.WriteLine(empTuple.Item2); //30
     Console.ReadKey();
    }
}
```

Improved Tuples (3-3)

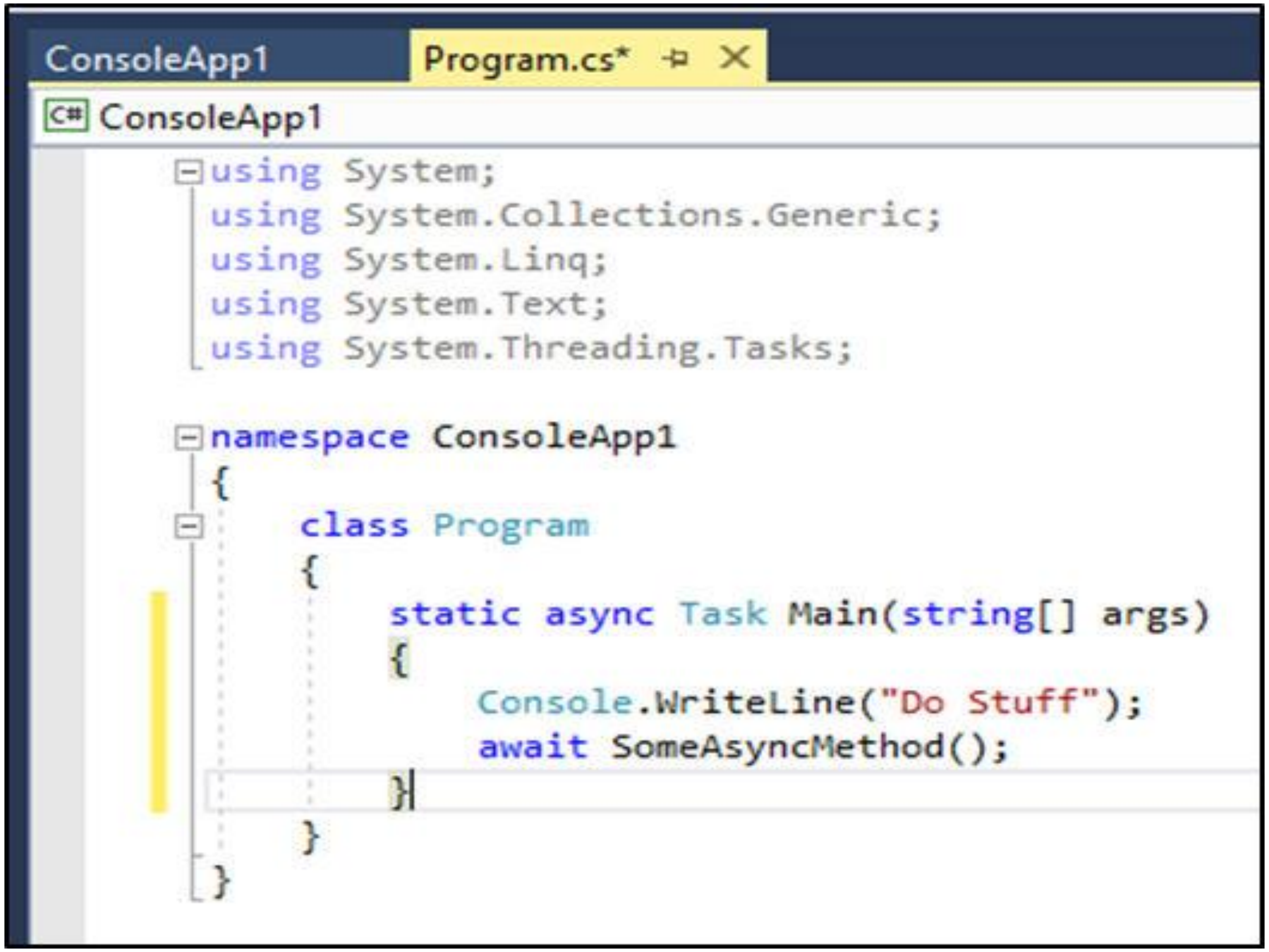
- Following code snippet demonstrates that how to specify element names manually:

Snippet

```
string ename = "Emy George";
int e_age = 30;
var empTuple = (ename: ename, e_age: e_age);
Console.WriteLine(empTuple.ename); //Emy George
Console.WriteLine(empTuple.e_age); //30
```

Improved Asynchronized `Main()`

- Following figure shows how it is easily possible to specify the `Main()` method of a C# 7.1 application as `async`:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Console.WriteLine("Do Stuff");
            await SomeAsyncMethod();
        }
    }
}
```

Improved Throw Expressions

- Following code snippet demonstrates how the `throw` statement was used previously as a separate statement:

Snippet

```
if (num1 == null) (
   throw new ArgumentNullException(nameof(num1));
}
```

- Following code shows how C# 7.0 uses `throw` expression write code in a more concise manner.

Snippet

```
myNum = num1 ?? throw new ArgumentNullException(nameof(num1));
```

- Following code demonstrates how to use the ?: operator to represent an if/else statement:

Snippet

```
returnval < 10 ? val : throw new
ArgumentOutOfRangeException("Value has to less
than 10");
```

More Expression-bodied Members

With C# 7.0, developers can now extend the expression-bodied members feature to cover more members such as property accessors, destructors, and constructors.

- Following code snippet demonstrates how an expression-bodied method can be used to invoke an asynchronous method in `Main()`:

Snippet

```
static async Task Main(string[] args) =>
WriteLine($"Factorial 6: {await AsFact(6)}");
```

Summary

- The `ref` keyword allows returning and storing values passed by reference.
- The `ref` returns and `ref` locals features are useful for replacing placeholders or reading from large data structures.
- C# 7.1 allows specifying the data type of out parameters inline in a method.
- The compiler for C# 7.1 is capable of inferring the element names of a tuple from local variables, null conditional members, and other members such as properties.
- C# 7.1 allows adding a throw exception in null-coalescing and conditional expressions and expression-bodied members.
- Expression-bodied members can include not only methods but also constructors, destructors, and properties, and property accessors.