Dynamic Programming in C# 10

Introducing Dynamic C#

Jason Roberts

.NET Developer

@robertsjason dontcodetired.com

Version Check

This version was created by using:

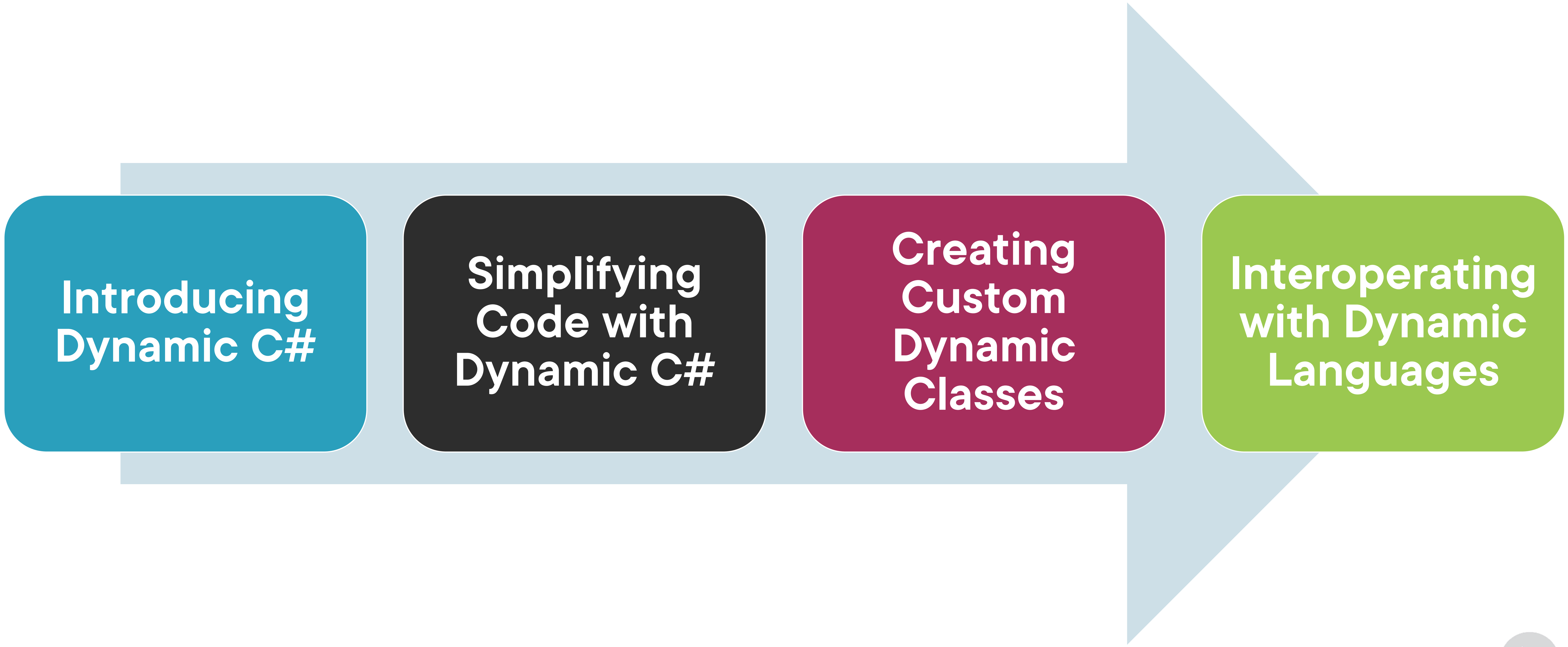
- C# 10
- .NET 6
- Visual Studio 2022

Version Check

This course is 100% applicable to:

- C# 11
- .NET 7
- Visual Studio 2022

Course Outline


Introducing Dynamic C#
Simplifying Code with Dynamic C#
Creating Custom Dynamic Classes
Interoperating with Dynamic Languages

Overview

Why dynamic C#?

Usage scenarios and benefits

Dynamic Language Runtime (DLR)

Static and dynamic binding

dynamic keyword & implicit conversions

Difference between ‘var’ and ‘dynamic’

Run time method resolution

Object and dynamic types

Limitations when calling methods

ExpandoObject

Why Dynamic C#?

Compliments “normal” statically typed C#

Don’t know object/data structure at compile time…

…or where you do but compiler doesn’t

Improved source code

- Simplifying code / less clutter
- Improving overall readability / intent
- Reducing amount of code (productivity)

Weakly typed data: JSON, XML, plain text

COM interop code

Interop with dynamic languages

It’s like saying to the compiler:

“I know you don’t know if you can do this or not **now**, but just trust me, I know at **runtime** that everything will be fine.”

Example Usage

Replacing reflection code

Simpler COM interop

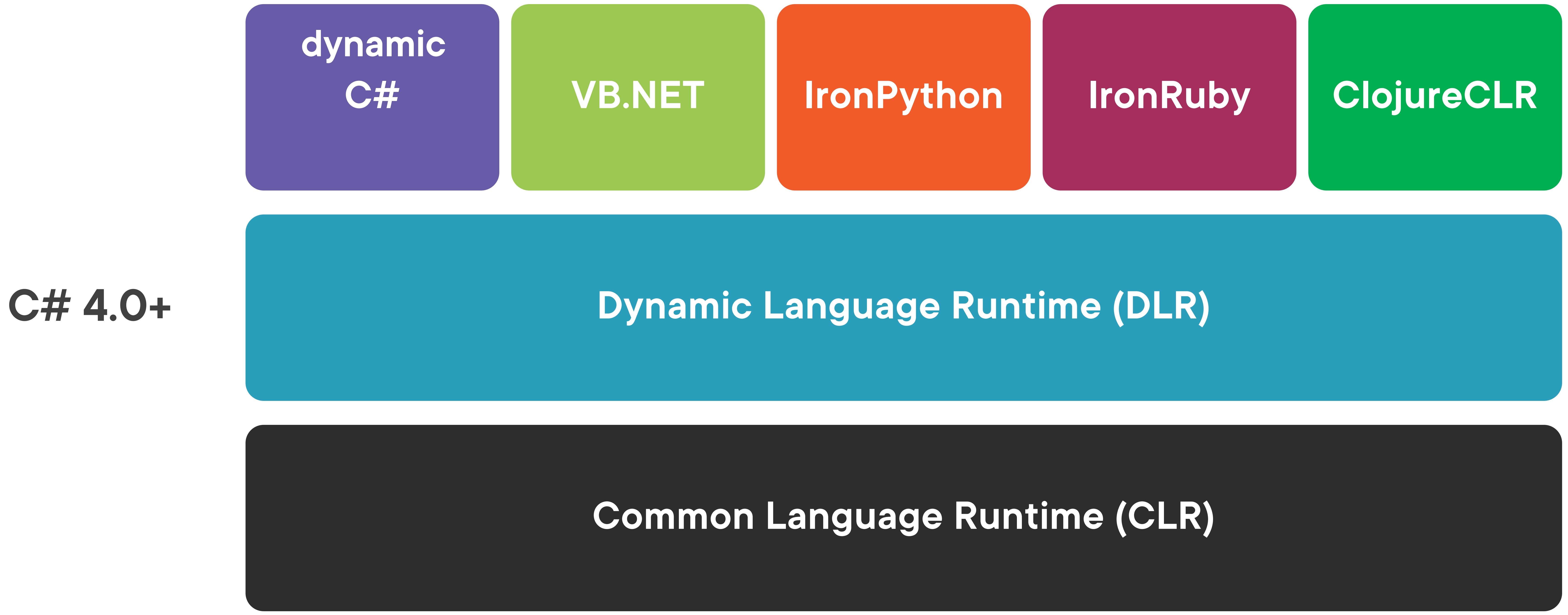
Dynamic JSON processing

XAML databinding

Data access code

Automated test code

Introducing the DLR



“The dynamic language runtime (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR). The DLR makes it easier to develop dynamic languages to run on the .NET Framework and to add dynamic features to statically typed languages.”

https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview

DLR Benefits

Simplify creation/porting of dynamic languages to .NET

Enable dynamic behaviour in statically typed languages

“dynamic” keyword in C#

Enables library sharing between languages

Enables object interoperability

- IDynamicMetaObjectProvider
- DynamicObject class
- ExpandoObject class

Call site caching

Static and Dynamic Binding

Binding

Binding is the association of a syntactic element (such as the name of a method) with a logical program element.

```
Calculator c = new Calculator();

c.Add(100);
```

Binding

Syntactic element “Add”

Binds to Add() method of variable c representing a Calculator object

```
Calculator c = new Calculator();

c.Add(100);

c.Xyz(100);
```

Static Binding

Binding occurs at compile time

Compiler knows Xyz() doesn’t exist in Calculator

Compile error, cannot build & execute program

```
dynamic c = CreateCalculator();

c.Add(100);

c.Xyz(100);
```

Dynamic Binding

Binding occurs at run time

Compiler doesn’t know if Add() or Xyz() exists

Program compiles and can be executed (run time error)

Even with dynamic C#, type safety is still enforced, only this time it's at run time.

Var and Dynamic

```
dynamic d = "Hi there";
```

Static (compile time) type of d is dynamic. Run time type will be string.

```
string s = "Hi there";
```

Static (compile time) type of s is string. Run time type will be string.

```
var s2 = "Hi there";
```

Static (compile time) type of s is string. Run time type will be string.

var = Compiler working out the type

dynamic = Runtime working out the type

Limitations of Callable Methods

Extension Methods

```
static class StringExtensions
{
    public static string PrependHello(this string s)
    {
        return $"Hello {s}";
    }
}
```

```
dynamic gentry = "Gentry";

// RuntimeBinderException
WriteLine(gentry.PrependHello());
```

Extension Methods

```
static class StringExtensions
{
    public static string PrependHello(this string s)
    {
        return $"Hello {s}";
    }
}


// Can still call “extension” method via static class

string s = StringExtensions.PrependHello(gentry);

WriteLine(s);
```

Explicitly Implemented Interface Members

```
interface IHelloable { string PrependHello(); }


class Person : IHelloable
{
    public string FirstName { get; set; }

    string IHelloable.PrependHello()
    {
        return $"Hello {FirstName}";
    }
}
```

Explicitly Implemented Interfaces Members

```
IHelloable p = new Person { FirstName = "Gentry" };

dynamic pd = p;

WriteLine(pd.PrependHello()); // RuntimeBinderException
// Person does not contain a definition for 'PrependHello'
```

Explicitly Implemented Interfaces Members

```
interface IHelloable { string PrependHello(); }


class Person : IHelloable
{
    public string FirstName { get; set; }

    string IHelloable.PrependHello()
    {
        return $"Hello {FirstName}";
    }
}
```

Consuming Void Methods

```
class Person
{
    public void DoStuff()
    {
        WriteLine("DoStuff() was called");
    }
}

dynamic p = new Person();

var x = p.DoStuff(); // RuntimeBinderException
// Cannot implicitly convert type 'void' to 'object'
```

Introducing ExpandoObject

General purpose class

System.Dynamic

Similar to dictionary with string based keys

Store/retrieve key/values

Keys added dynamically (not by string)

Improve readability

Reduce magic strings

IDynamicMetaObjectProvider custom dynamic behaviour

Implement custom dynamic objects with IDynamicMetaObjectProvider

Summary

Why dynamic C#?

Productivity, readability, simplicity

Dynamic Language Runtime (DLR)

Static and dynamic binding

dynamic keyword & implicit conversions

Difference between ‘var’ and ‘dynamic’

Run time method resolution

Object and dynamic types

Limitations when calling methods e.g. extension methods

dynamic d = new ExpandoObject();

Up Next:

Simplifying Code with Dynamic C#