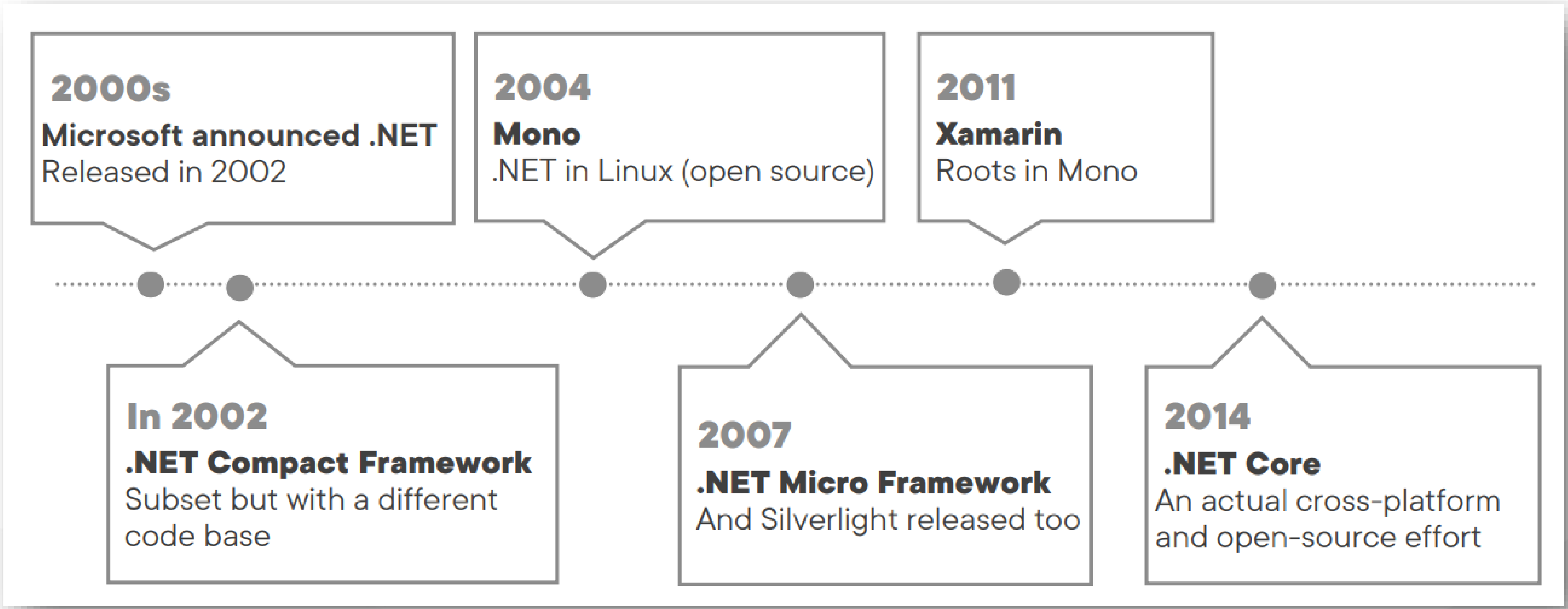
Core C# Programming

What is .NET?

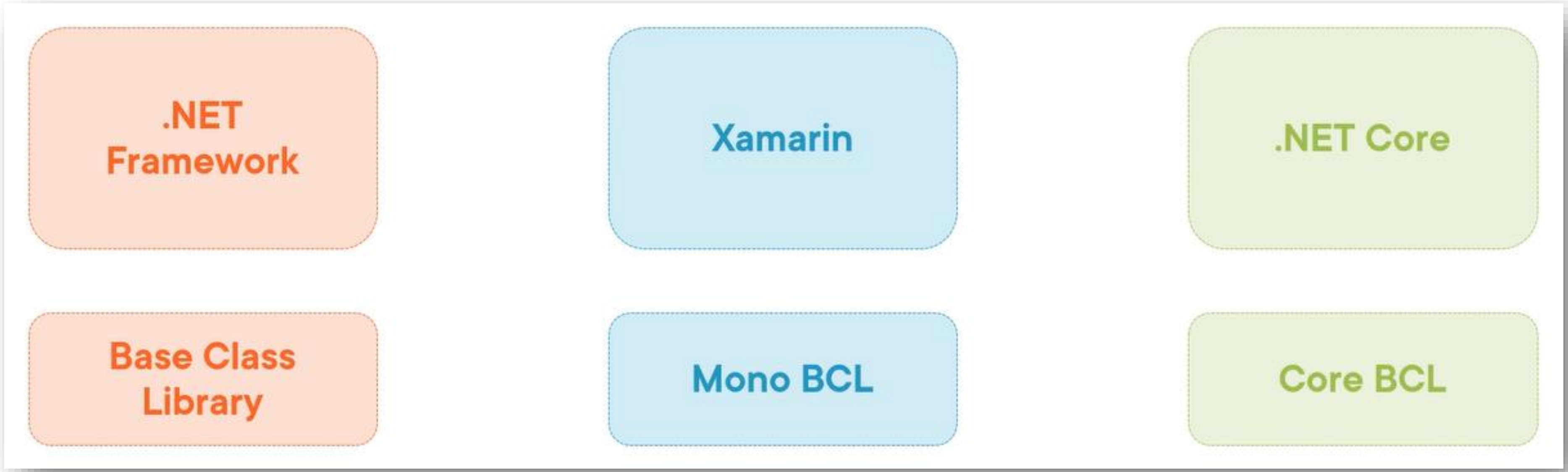
- A free, cross-platform, open source developer platform & framework that supports many languages, made up of a number of tools & libraries for developing many different types of applications
- Used to build web, mobile, desktop, gaming, cloud-centric, ML, IoT based applications and more
- Formerly known as **NGWS**
- Originally developed by Microsoft
- Implementation is **language agnostic**

The Evolution



CitiusTech

.NET Flavors



- The BCLs are similar, but not identical
- Code sharing between flavours of .NET was not possible **(write code once & run on different platforms)**
- Solution is:

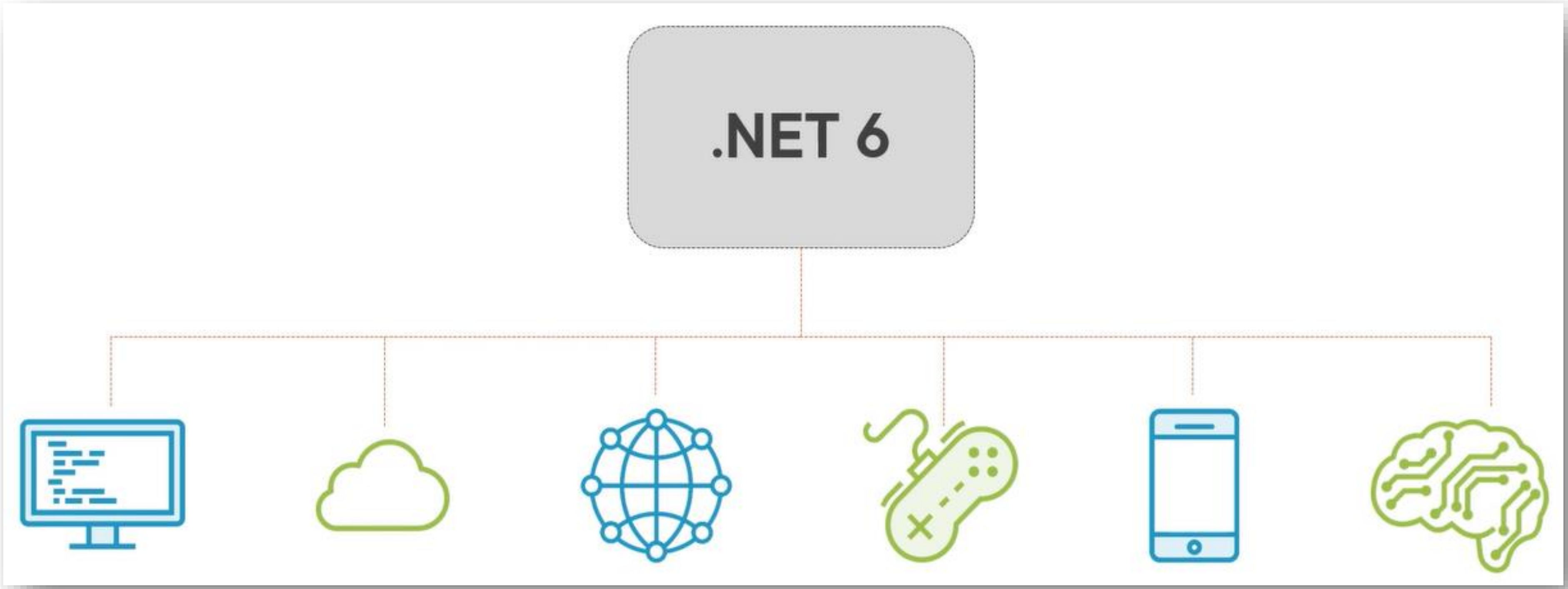
.NET Standard

What & Why .NET Standard?

- A specification that defined a set of APIs which allows writing of code that can be referenced by .NET Framework, Xamarin & .NET Core

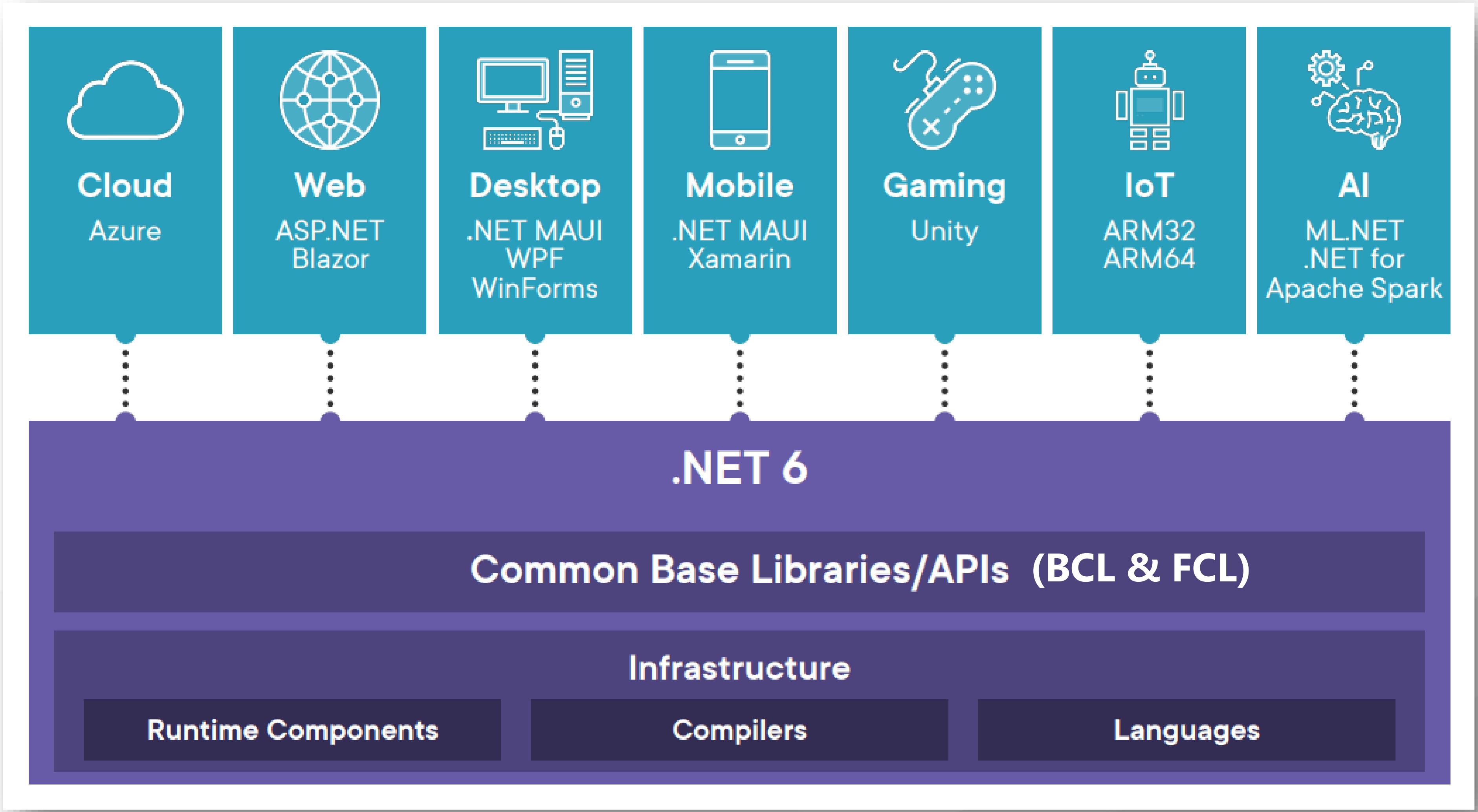
- Allows sharing of non-UI code across different .NET flavors

One .NET Platform for Unification

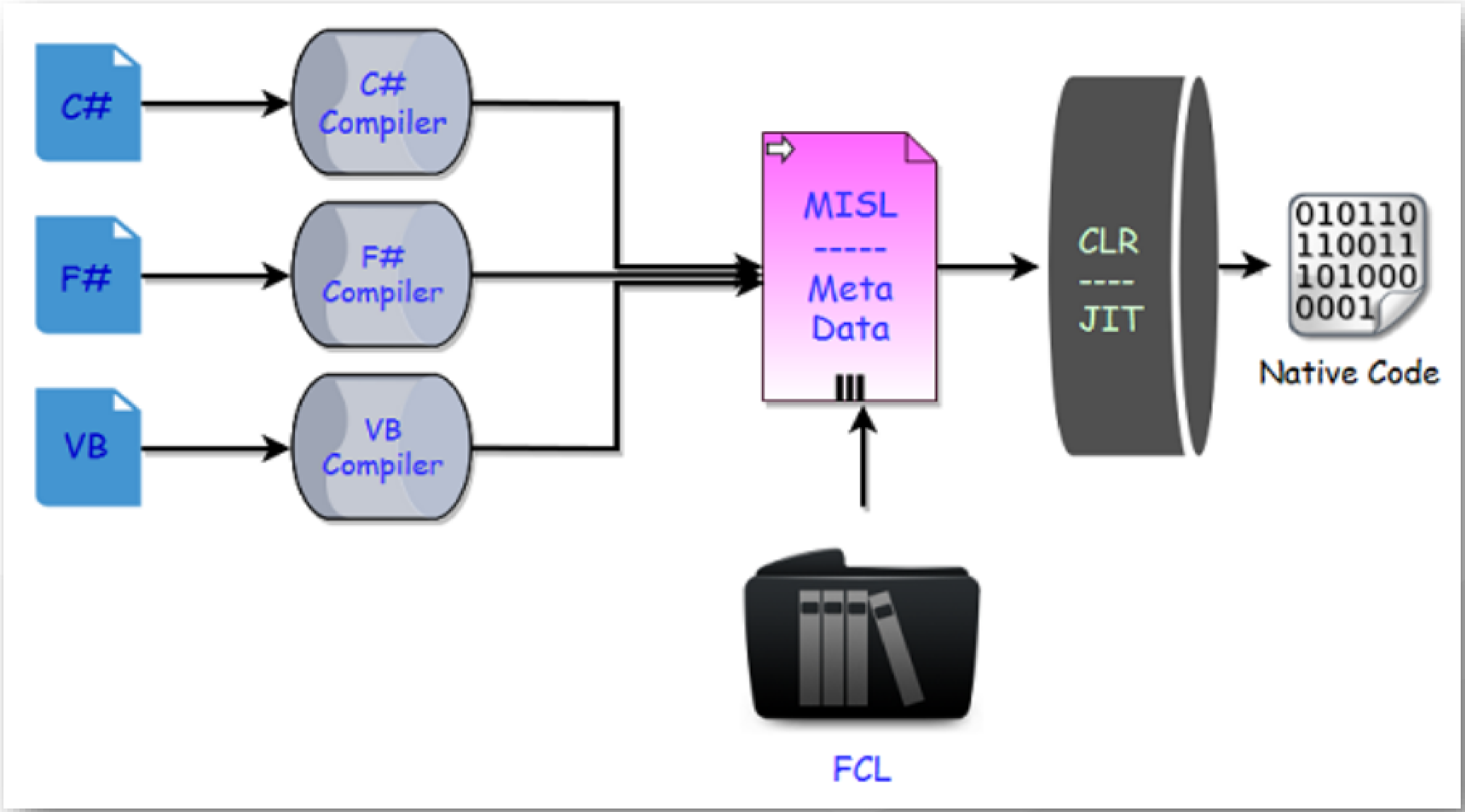


- Released in 2021 *(.NET 5 was released in 2020)*
- Single SDK & BCL
- Cross-platform native UI (MAUI)
- Cross-platform Web UI (Blazor)

The .NET Architecture



.NET Code Execution – A High Level Overview



The .NET CLR

- Provides a **managed environment** for code execution with runtime services

- Runs **managed code**

- CLR Services:
  - Cross language integration *(interoperability)*
  - COM Interoperability
  - Exception Handling
  - Garbage Collection
  - Multithreading
  - CAS
  - *And more…..*

Assemblies in .NET

- An assembly is the smallest unit of deployment, version control & reuse in the .NET world
- Is a collection of types, logically grouped into **namespaces**
- Can be an **exe** ***(.exe)*** or a **dll** ***(.dll)***
- An assembly is built from one or more source code files compiled by a .NET compiler
- Can be shared between applications by putting them into a **GAC**

CTS and CLS in .NET

- CTS is a common way to describe all supported **types** in .NET

- Why CTS?
  - Cross-language integration
  - Define a set of rules that all languages must follow when it comes to working with types
  - A common set of primitive types like Int32, Boolean, Byte, etc. that can be used for application development
  - A common object-oriented model for all .NET languages

- CLS is a subset of CTS

- Defines a common set of rules which all .NET languages must follow

- Why CLS?
  - To enable full language interoperability

Introducing C# Language

- An object-oriented, type-safe, strongly-typed & statically typed programming language developed for the .NET Framework

- A case-sensitive langage

- Designed by **Anders Hejlsberg** from Microsoft in 2000

- Originally known as **C with Classes**

- Has undergone many changes *(versions)* since its first release
  - https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history
  - Current stable version as of 2022 is 11.0

CitiusTech

Developing .NET Applications with C#

Visual Studio 2022
Windows & Mac

.NET CLI &
Visual Studio Code
All platforms

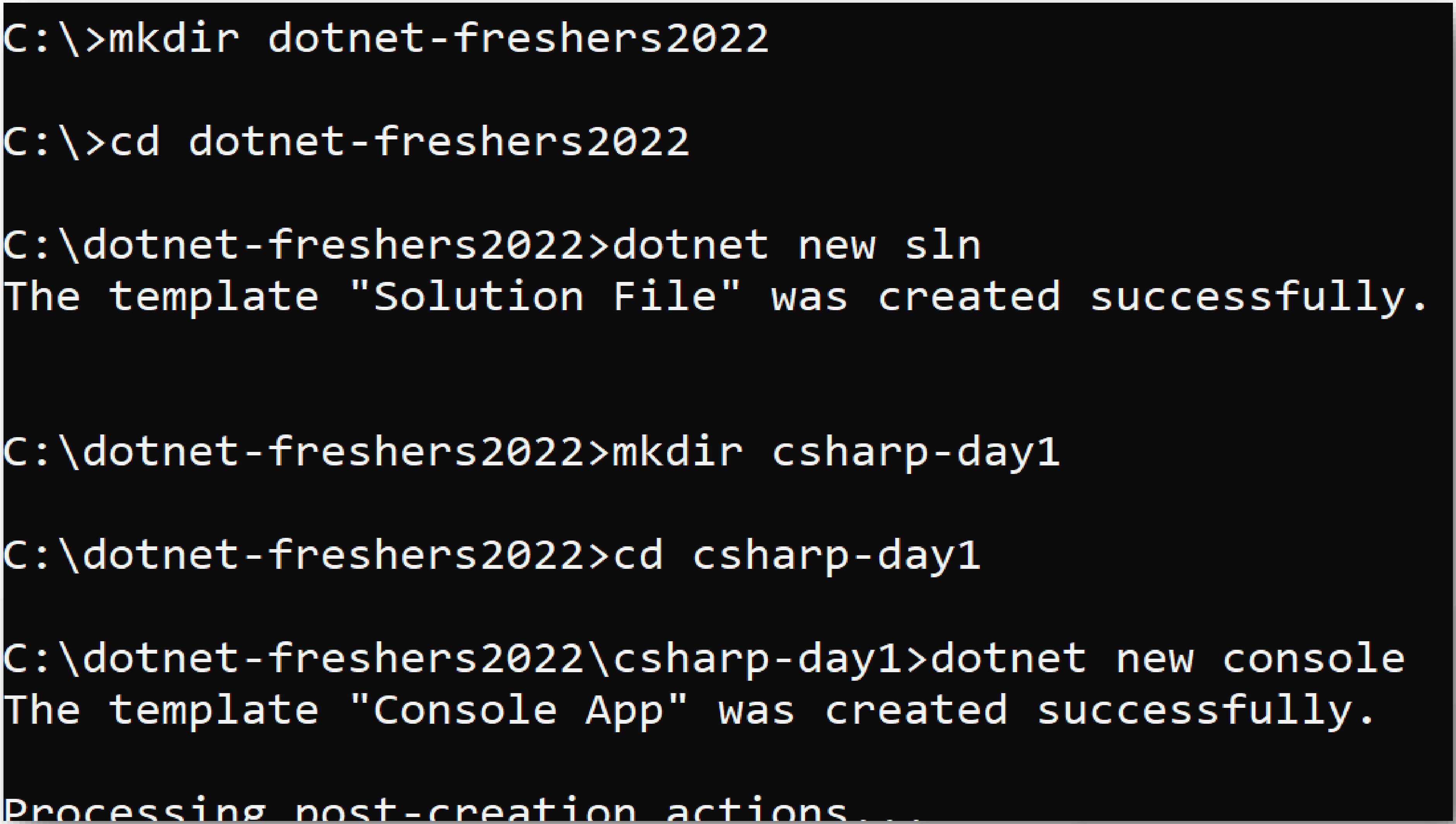
C# Console Applications

- An application which no GUI
- Text-based output on a terminal/console
- Can be created using VS 2022, VS Code or using .NET CLI

Visual Studio Projects & Solutions

- A project contains all files that are compiled into an executable or a library
- Contains source code, images, data files, etc.
- For C#, a project file has **.csproj** extension & for VB.NET, a **.vbproj** extension, etc.
- Each project has a **template**
- A solution a container for one or more related projects
  - In a way, a logical grouping of one or more projects
  - Has a file extension of **.sln**

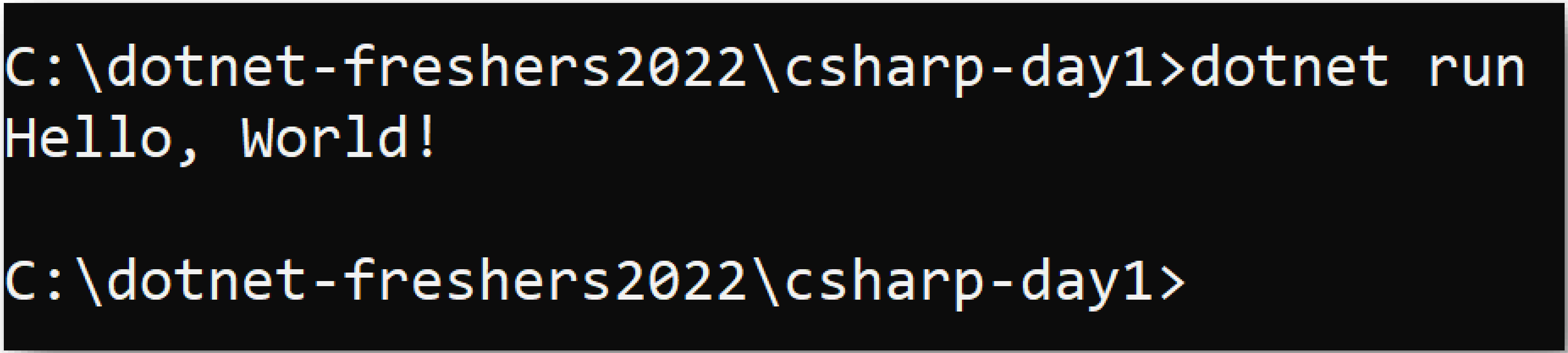
Creating a new C# Console Application - .NET CLI



```
C:\>mkdir dotnet-freshers2022

C:\>cd dotnet-freshers2022

C:\dotnet-freshers2022>dotnet new sln
The template "Solution File" was created successfully.


C:\dotnet-freshers2022>mkdir csharp-day1

C:\dotnet-freshers2022>cd csharp-day1

C:\dotnet-freshers2022\csharp-day1>dotnet new console
The template "Console App" was created successfully.

Processing post-creation actions...
```

CitiusTech

Executing a C# Console Application - .NET CLI



```
C:\dotnet-freshers2022\csharp-day1>dotnet run
Hello, World!

C:\dotnet-freshers2022\csharp-day1>
```

Basic structure of a C# Program

```
using System;        //root (predefined) namespace
namespace Day1  //custom namespace
{
    class Program        //class
    {
        static void Main()      //Entry point
        {
            //Console is a "class" in the System namespace
            //WriteLine() is a static method in the Console class
            Console.WriteLine("Hello World!");
        }
    }
}
```

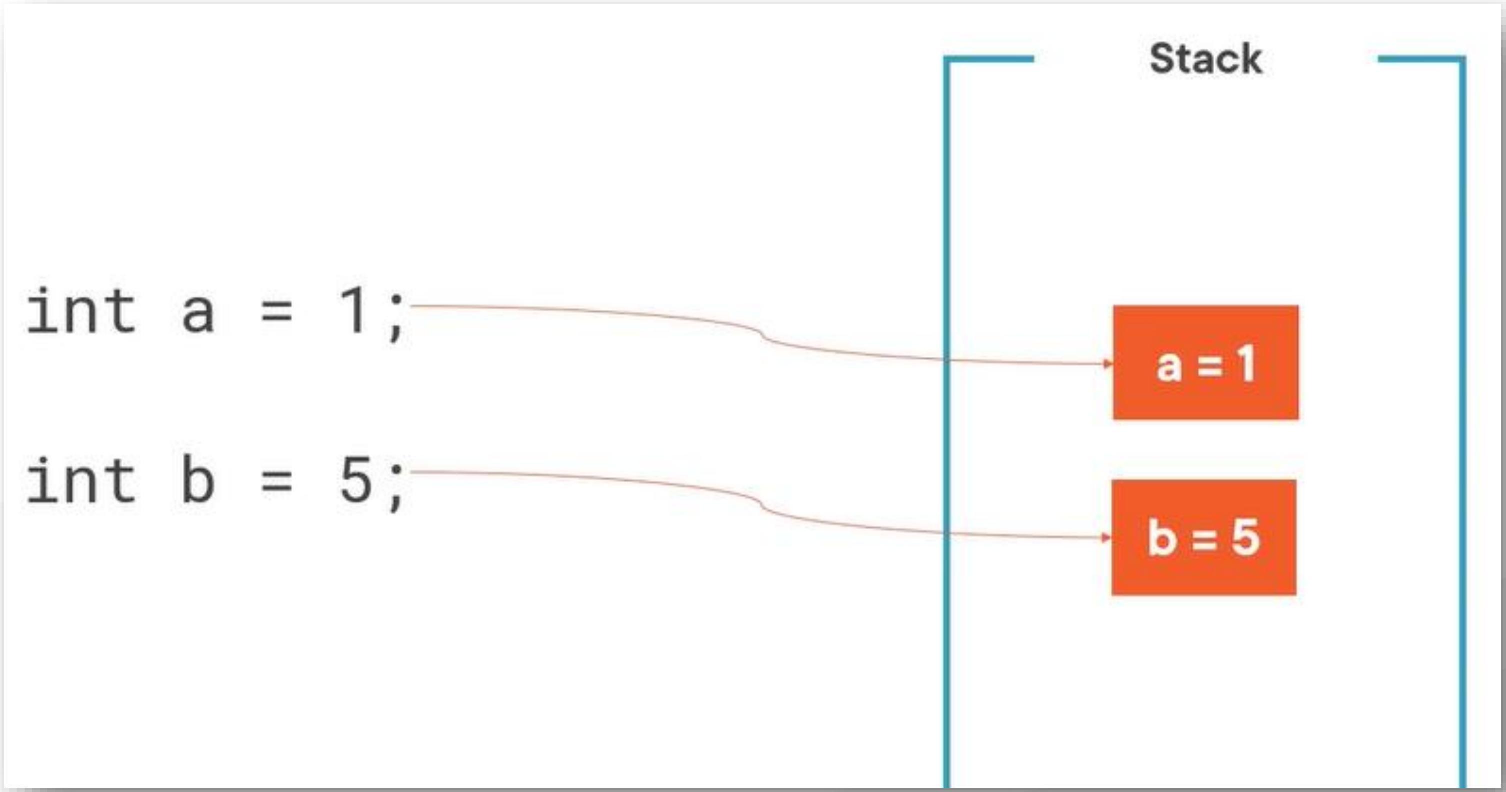
C# variables

- A variable holds a value
- Must have a name & a type
- Must be given a value before it is used in an expression

```
int age = 25;

double amt;
amt = 200.50;

bool b = true;
const byte x = 20;

float f1 = 20.70, f2 = 30.60;
string s = "KARTHIKEYAN";
```

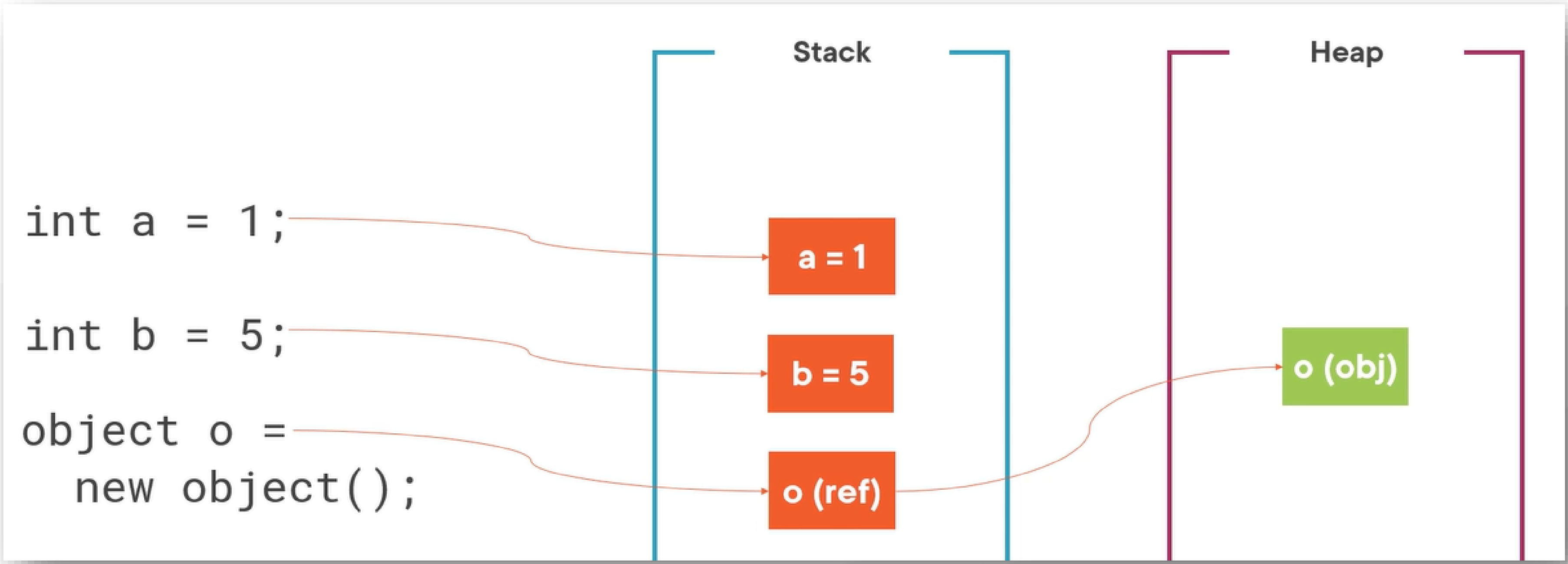
Types in C# and .NET

- **Value Types**
  - Variables of the value types directly contain their data on the stack
  - Variables each have their own copy of the data, and it is not possible for operations on one to affect the other
  - All primitive types are value types. Example: *int, float, char, bool,* etc.



Types in C# and .NET

- **Reference Types**
  - Variables of the reference types store **references *(pointer on the stack)*** to their data which is allocated on the **heap**
  - Possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable
  - Example: classes, delegates, arrays, strings, etc.



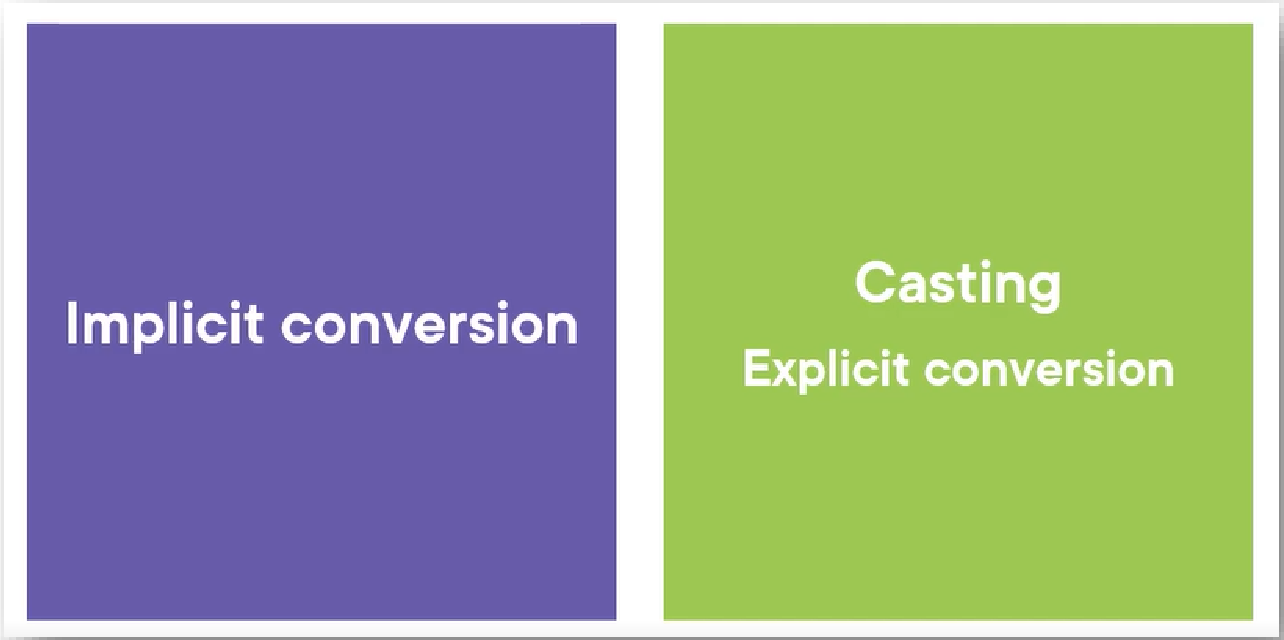
C#’s unified type system

- C#’s type system is unified such that **a value of any type can be treated as an object**

- Every type in C# directly or indirectly derives from the base type known as **System.Object** *(the ultimate base class of all types)*

- This is to achieve a **polymorphic behavior** across types

C# Type Conversion



CitiusTech

C# Type Conversion - Examples

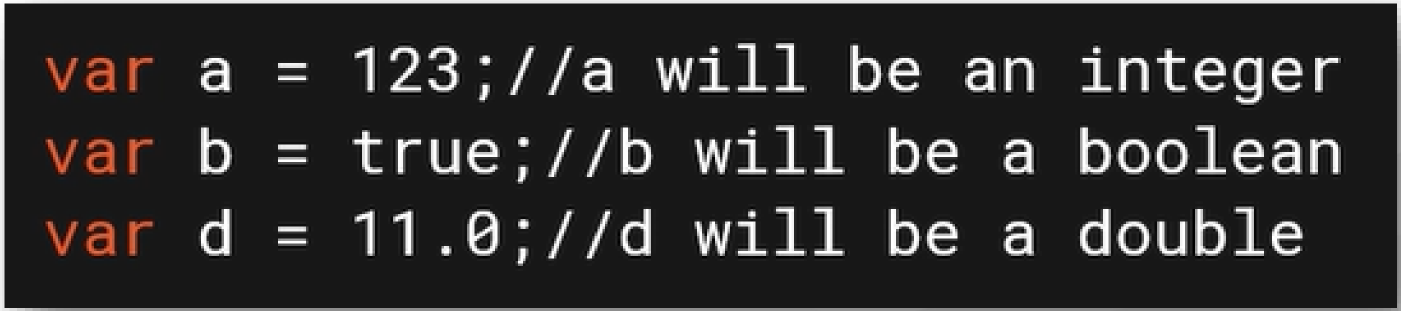
```
int a = 123456;

//implicit since long's range is higher than an int
long l = a;
```

```
double d = 600.34;
int i = (int) d;     //explicit → decimal part is truncated
Console.WriteLine(i);    //displays 600

int x = 100;
byte b = (byte) x;
```

CitiusTech

C#’ implicit typing using “var”

- Introduced from C# 3.0 onwards
- Automatically infers the type of the variable based on the value stored in it
- Variable must be initialized with some value
- Cannot be used as a method parameter and/or as a method’s return type
- Variable is still strongly and statically typed
- Why **var?**
  - When using **anonymous types** in C# 3.0 with **LINQ**



```
var a = 123;//a will be an integer
var b = true;//b will be a boolean
var d = 11.0;//d will be a double
```

Boxing and Unboxing

- Boxing is the process of converting a value type to the type **System.Object**
  - When a value type is **boxed,** its value is wrapped in a **System.Object** instance and stored on the managed heap
- Boxing is an **implicit operation**
- Only value types are boxed
- Unboxing extracts the value type from the object
- Unboxing is an **explicit operation** and requires a cast
- Boxing and unboxing degrade the application performance
- Why boxing and unboxing?
  - Indicates the polymorphic behavior across .NET types
  - Used by non-generic collections

Boxing and Unboxing - Examples

```
int x = 10;

//this is "boxing" (implicit)
object o = x;

//this is "unboxing" (explicit)
int y = (int) o;
```

```
static void Main()
{
        Print(10);      //boxing
        Print(true);    //boxing
        Print(70.88);   //boxing
}
//this is a "polymorphic" method
static void Print(object o)
{
        Console.WriteLine(o);
}
```

Writing methods in C#

- A method is a code block
- Receives parameter(s) and optionally returns a value
- Allows modularity and code reuse
- Must be declared within a class or a struct
- Can be instance or static
- Can be overloaded

Writing methods in C# - example

```
static void Add(int x, int y)
{
        Console.WriteLine("Sum of {0} and {1} is {2}", x, y, (x+y));
}
static int Subtract(int x, int y)
{
        return (x - y);
}
```

```
static void Main()
{
        int x, y;
        Console.Write("Enter operand1 value: ");
        x = int.Parse(Console.ReadLine());

        Console.Write("Enter operand2 value: ");
        y = int.Parse(Console.ReadLine());

        Add(x, y);
        int result = Subtract(x, y);
        Console.WriteLine("Subtraction of {0} and {1} is {2}", x, y, result);
}
```

Default value expressions

- Introduced from **C# 7.1 onwards**
- Produces the default value of a type
- **default()** operator produces the default value of a type and the **default** literal initializes a variable of a type with its default value
- Why default?
  - Assigning or initializing a variable of a type with its default value
  - Declaring an optional method parameter
  - Providing an argument value to a method
  - Returning a value from a method

Default value expressions - example

```
//here x and y will have values as zero
//these variables can now be used in expressions
//this is a "default" literal
int x = default, y = default;
Console.WriteLine("{0} and {1}", x, y);

//print default values of some types using the default() operator
Console.WriteLine(default(bool));
if(default(object) == null)
{
            Console.WriteLine("Default value of System.Object is null");
}
```