# Datatypes

.NET

*In computer science and computer programming, a data **type** or simple **type** is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data.*

HTTPS://EN.WIKIPEDIA.ORG/WIKI/DATA_TYPE

# Primitive Types in C# vs Java

https://medium.com/omarelgabrys-blog/primitive-data-types-in-c-vs-java-5b8a597eef05#:~:text=
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types

---

The most familiar primitive data types are:
int, *object*, *short*, *char*, *float*, *double*, *char*, *bool*. They are called ***primitive*** because they are the main built-in types and could be used to build other data types.

In C#, what are called primitive data ***types*** in other languages are actually ***objects***. This means when you write:

- int foo = 10;
- string myString = “This is a string”;

The variables *foo and myString* are Objects. They have helper functions built into C# to manipulate the data. This is one reason C# is ***Strongly Typed***. The compiler must know the ***type*** to be able to  supply the helper functions.

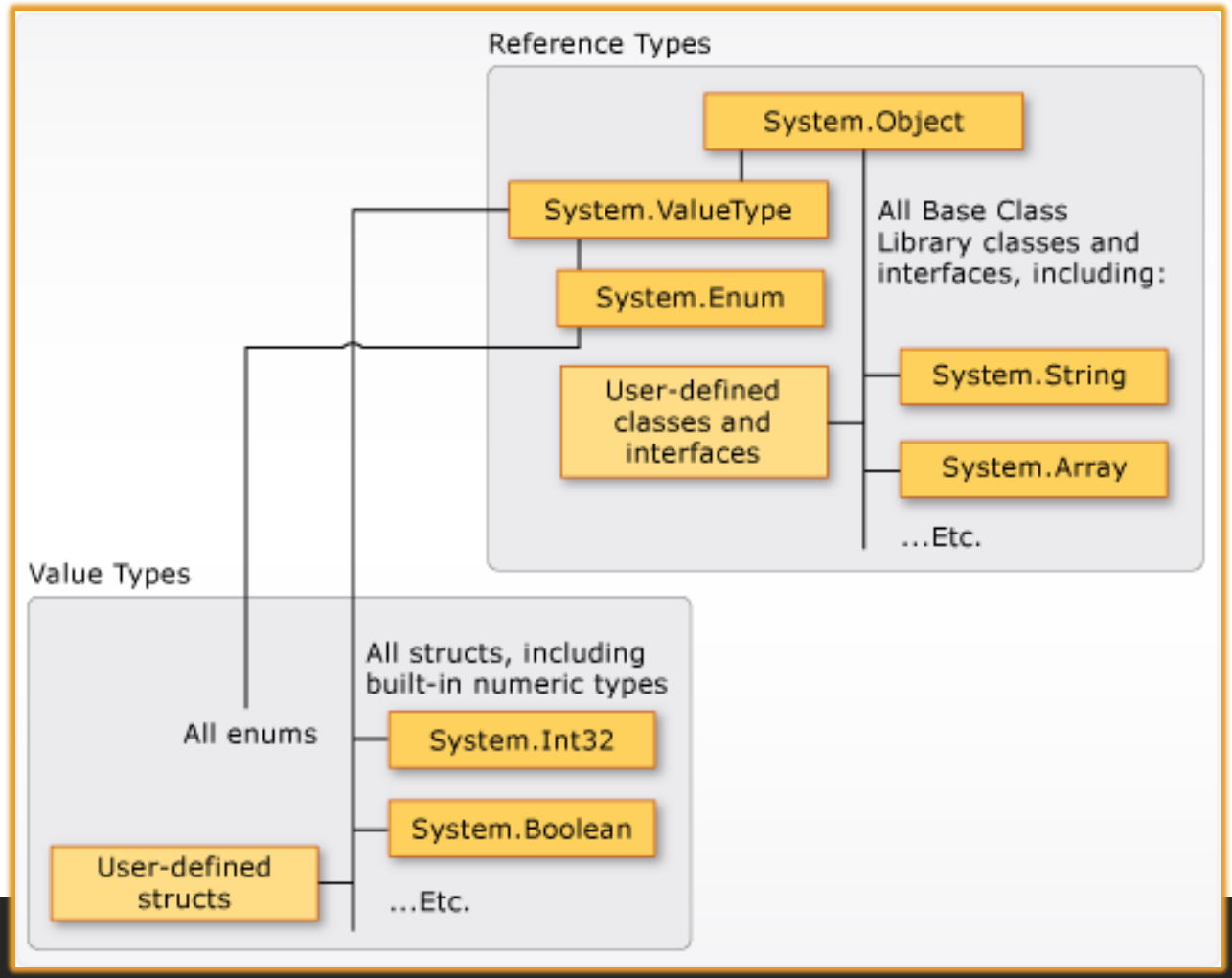
# C# Datatypes Structure

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/

All data types inherit from the base Class ***Object***.

When an ***int*** is declared, you are declaring:

- an instance of the ***struct*** (an object), 'int',
- which inherits from ***System.ValueType***,
- which inherits from ***System.Object***



# DataTypes

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables

C# supports two categories of variable ***type***:

- Value ***types***
  - These are the built-in data ***types***, such as ***char***, ***int, bool***, ***float***, and user-defined ***types*** declared with ***struct***. Variables of value types directly contain their data (on the ***stack***).
- Reference ***types***
  - ***Class***, ***Interface***, ***array*** and ***delegate types*** contain other ***types***. Variables of reference ***types*** do not contain an instance of the type, but merely a reference to an instance stored on the ***heap***.

# Value Types – Integral

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types

***Integral*** numeric ***types*** represent ***integer*** numbers. All ***integral*** numeric ***types*** are value ***types***. They are also simple ***types*** and can be initialized with ***literals***.

| Signed Integral | values |
|---|---|
| Sbyte | -128 to 127 |
| Short | -32768 to 32767 |
| Int | -2147483648 to 2147483647 |
| Long | -9223372036854775808 to 9223372036854775807 |

| Unsigned Integral | values |
|---|---|
| Byte | 0 to 255 |
| Ushort | 0 to 65535 |
| Uint | 0 to 4294967295 |
| Ulong | 0 to 18446744073709551615 |

# Value Types

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types#built-in-value-types

| Unicode Characters | value |
|---|---|
| char | 0 and 65535 |

| boolean | Value |
|---|---|
| bool | true and false (NOT 0/1) |

| IEEE binary floating-point | values |
|---|---|
| float | Approx. 1.5 * 10^-45 to 3.4 * 10^38 with precision of 7 digits. |
| double | Approx. 5.0 * 10^-324 to 1.7 × 10^308 with precision of 15-16 digits. |

| High-precision decimal floating-point | Values |
|---|---|
| decimal | 1.0 * 10^-28 to approx. 7.9 * 10^28 with 28-29 significant digits |

# Value Types – Enum

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum

To define an enumeration type (***enum***), use the ***enum*** keyword and specify the names of ***enum*** members. An ***enum*** is a value type defined by a set of named constants of the underlying integral numeric type.
There exist explicit conversions between the ***enum*** type and its underlying ***integral*** type. If you cast an ***enum*** value to its underlying type, the result is the associated integral value of an ***enum*** member.
***Enums*** are immutable.

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");
                                // output: Integral value of Autumn is 2
        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

# Value Types – Struct

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct

A **structure type (struct)** is a ***value*** type that can encapsulate data and related functionality. Use the ***struct*** keyword to define a structure ***type***. Typically, you use structure ***types*** to design small data-centric ***types*** that provide little or no behavior.

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

# Reference Type – Class

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class

Classes are declared using the keyword ***class***. A class can declare class fields, constructors, and methods.

```
class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
```

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}
```

# Reference Type – Interface

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

An ***interface*** contains definitions for a group of related functionalities that a non-abstract ***class*** or a ***struct*** must implement.

An ***interface*** defines a “contract”.

Any ***class*** or ***struct*** that implements that contract agrees to provide an implementation of the members defined in the ***interface***.

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

# Reference Type – Delegate

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/delegates
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types

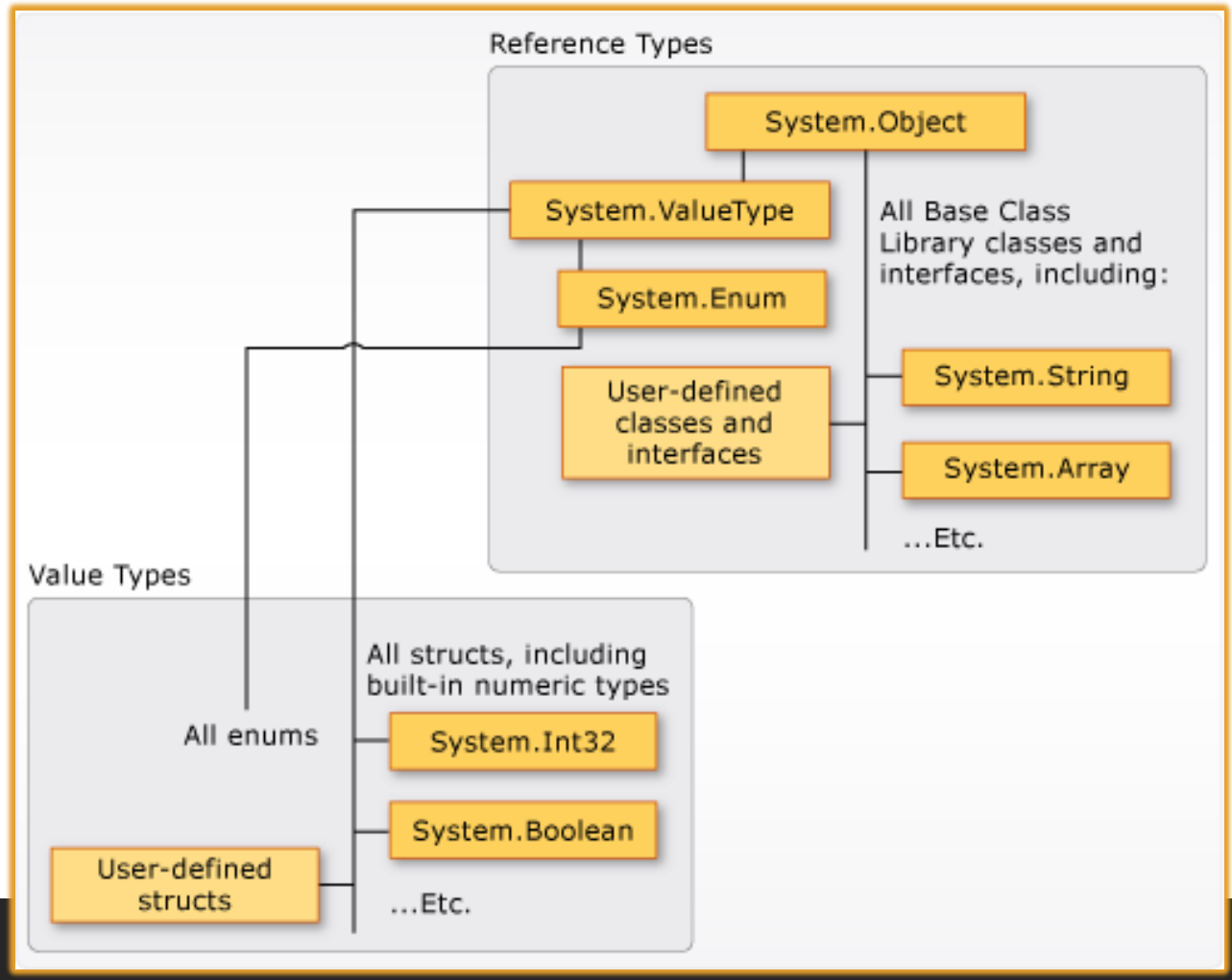
A ***delegate type*** represents references to methods. ***Delegates*** make it possible to treat methods as entities that can be assigned to variables and passed as parameters. ***Delegates*** are similar to the concept of function pointers found in other languages, but unlike function pointers, ***delegates*** are object-oriented and type-safe.

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor)
    {
        this.factor = factor;
    }
    public double Multiply(double x)
    {
        return x * factor;
    }
}
class DelegateExample
{
    static double Square(double x)
    {
        return x * x;
    }
    static double[] Apply(double[] a, Function f)
    {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main()
    {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

# Reference Type – Object

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types

In the ***Unified Type System (UTS)*** of C#, all types inherit directly or indirectly from System.Object.
You can assign values of any ***type*** to variables of ***type object***. Any ***object*** variable can be assigned to its default value using the literal ***null***.
When a variable of a value ***type*** is converted to ***object***, it is said to be boxed. When a variable of type object is converted to a value type, it is said to be unboxed.



# Reference Type – String

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types

The ***string type*** represents a sequence of zero or more Unicode characters. ***string*** is an alias for System.String.

The addition operator '+' and the equality operators '==' and '!=' are defined to concatenate and compare the values of ***string objects*** (not the references).

Strings are ***immutable***--the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can.

This example displays "True" and then "False" because the content of the strings are equivalent, but `a` and `b` do not refer to the same string instance.

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);// True
Console.WriteLine(object.ReferenceEquals(a, b)); // False
```

# Reference Type – String

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types

The [] operator can be used for readonly access to individual characters of a string or iterating over them in a loop. Valid index values start at 0 and must be less than the length of the string.

```
string str = "test";
char x = str[2];  // x = 's';
```

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```