Datatypes

.NET

*In computer science and computer programming, a data **type** is an attribute of data which tells the compiler 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

---

Typically, the most familiar primitive data types are:
int, object, short, float, double, char, bool. These are called primitive because they are the types used to build other, more complex, data types.

What are considered ***primitive data types*** in other languages are ***objects*** in C#.

When you write:

- int foo = 7;
- string myString = “Caravan”;

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

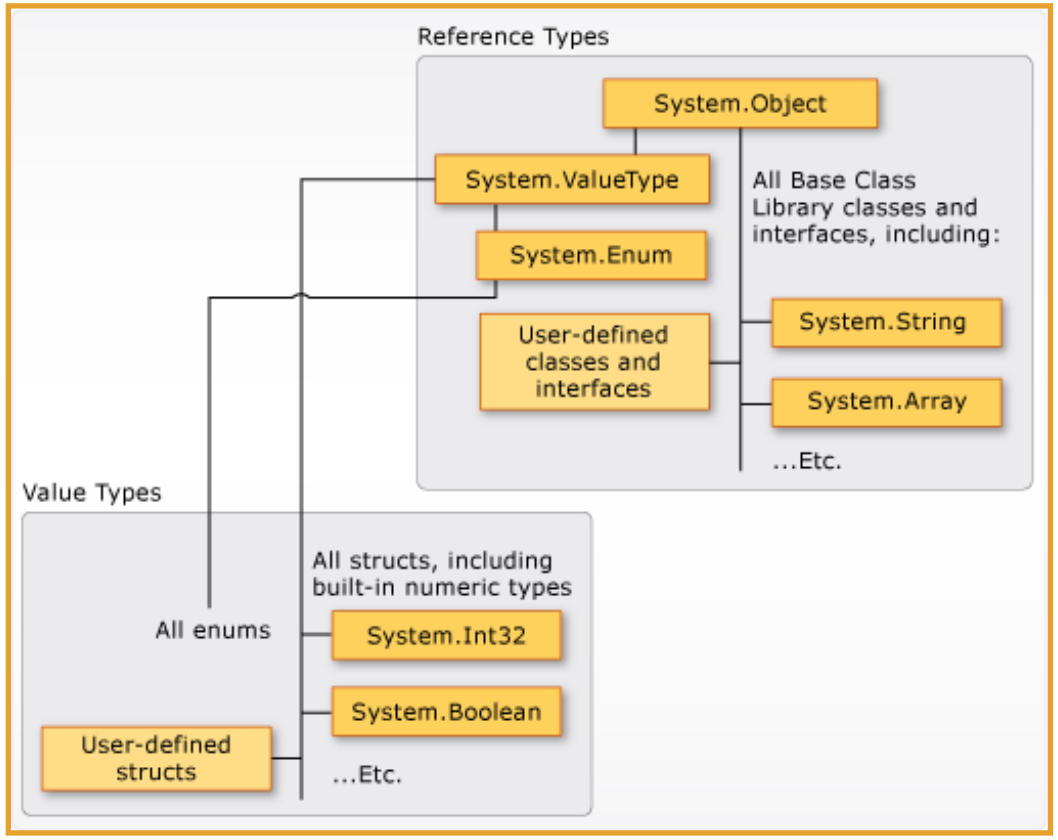
C# Datatypes Structure

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) of type 'int',
- which inherits from ***System.ValueType***.
- ***System.ValueType*** inherits from ***System.Object***



DataTypes

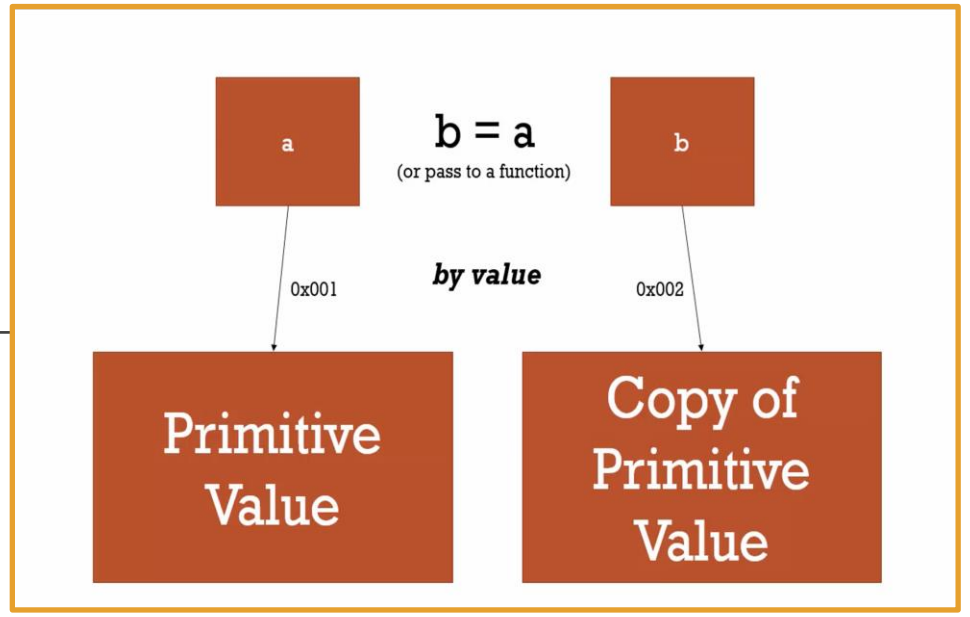
https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables

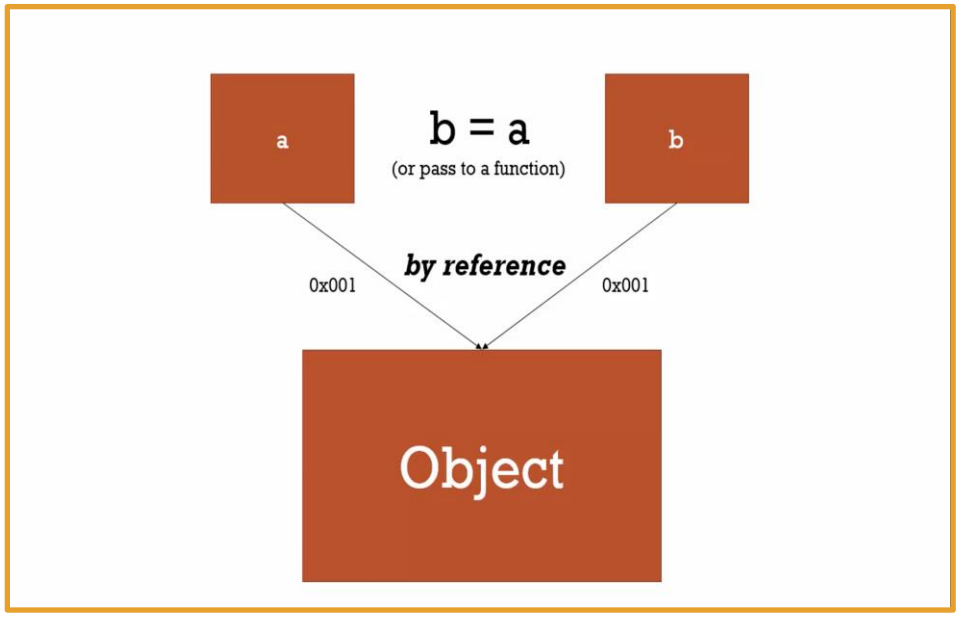
Value *types*:

- These are the built-in data *types:*
  - *char*, *int, bool*, *float*, *double*
  - and user-defined *types* declared with *struct*.
  - Variables of *value* types directly contain their data on the *stack* .

Reference *types*:

- *Class*, *Interface*, *array*, *collection* and *delegate types* contain other *types*.
- Variables of *reference type* do not contain an instance of the type, but merely a reference to an instance stored on the *heap*.



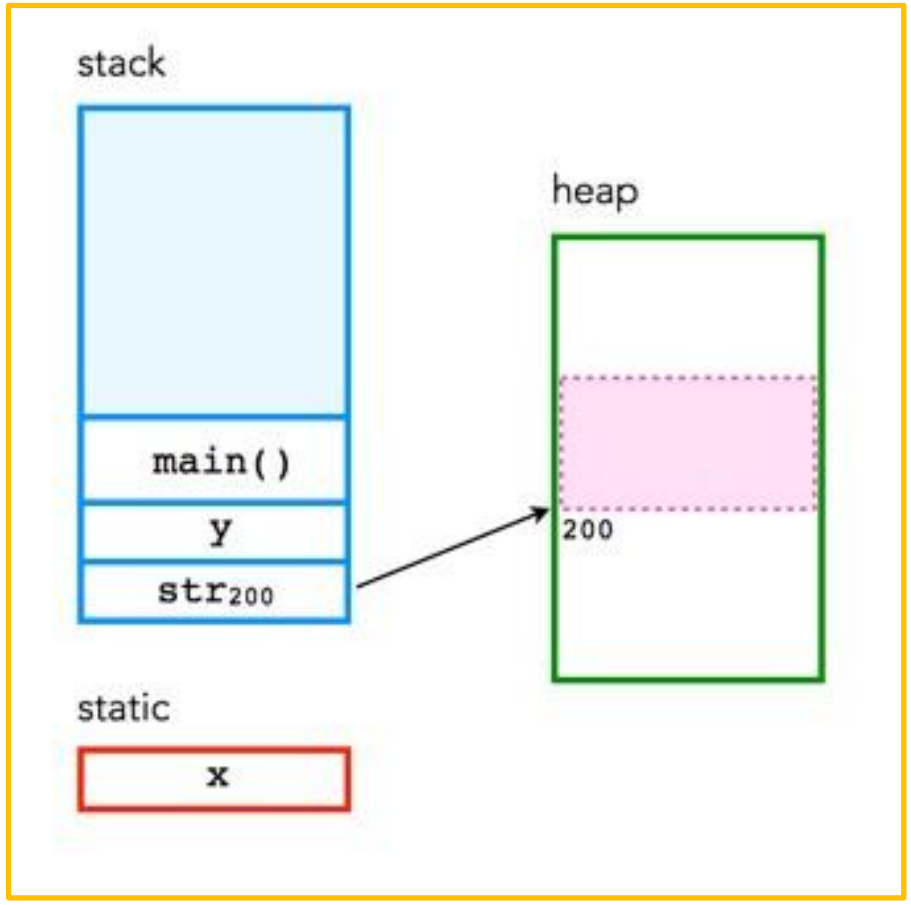


Value Type vs reference type

https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/value-types-and-reference-types

There are two kinds of types in C#: reference types and value types.

- Variables of ***value types***:
  - directly contain their data.
  - Have their own copy of the data
  - It's not possible for operations on one variable to affect the other
- Variables of ***reference types***:
  - store references to their data (objects).
  - Two variables can reference the same object.
  - Operations on one variable can affect the object referenced by the other variable.



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 types*** are ***value types***. They are also simple ***types*** and can be initialized with ***literals***.

Signed Integral	Size	Range
Sbyte	Signed 8-bit	-128 thru 127 (255)
Short	Signed 16-bit integer	-32768 thru 32767
Int	Signed 32-bit integer	-2,147,483,648 thru 2147483647
Long	Signed 64-bit integer	-9223372036854775808 thru 9223372036854775807

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***.

Unsigned Integral	Size	Range
Byte	Unsigned 8-bit integer	0 thru 255
Ushort	Unsigned 16-bit integer	0 thru 65535
Uint	Unsigned 32-bit integer	0 thru 4,294,967,295
Ulong	Unsigned 64-bit integer	0 thru 18446744073709551615

Value Types

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

Unicode Characters	Size	Range
char	16 bit	0 - 65535

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

IEEE binary floating-point	Size	values
float	4 bytes	Approx. 1.5 * 10^-45 - 3.4 * 10^38 with precision of 7 digits.
double	8 bytes	Approx. 5.0 * 10^-324 - 1.7 × 10^308 with precision of 15-16 digits.

High-precision decimal floating-point	Size	Values
decimal	16 bytes	1.0 * 10^-28 - approx. 7.9 * 10^28 with 28-29 significant digits

Signed vs Unsigned values

Value Types – Enum

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

An ***Enumeration*** (***enum***) is a ***value*** type defined by a set of named constants of the underlying ***integral numeric type***.
It is a ***type*** that are custom created. You define the valid examples of the ***enum***.
To define an ***enum***, use the enum keyword and specify the names of the ***enum*** members.
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.
Structs are typically used 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})";
}
```

Nullable Values and Reference Types

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

---

In C#, ***value types*** aren't allowed to be null. To allow a normally non-nullable type to be ***NULL***, add ? to the type declaration.

For example, if you retrieve a database field that may contain no value (***NULL***), you can use bool? or int? in the declaration to allow the variable to be ***NULL***.

Reference types can always be null but there is special syntax that allows the compiler to check reference values and avoid runtime errors. Click the link above for more details.

To check if a ***nullable*** type has a value you can use:

- myVar.HasValue = true/false
- If(myvar == null || myVar.HasValue) { }

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

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***, meaning 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. 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 the zero-indexed individual characters of a string or iterating over them in a loop.

```
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
```

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 – 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.

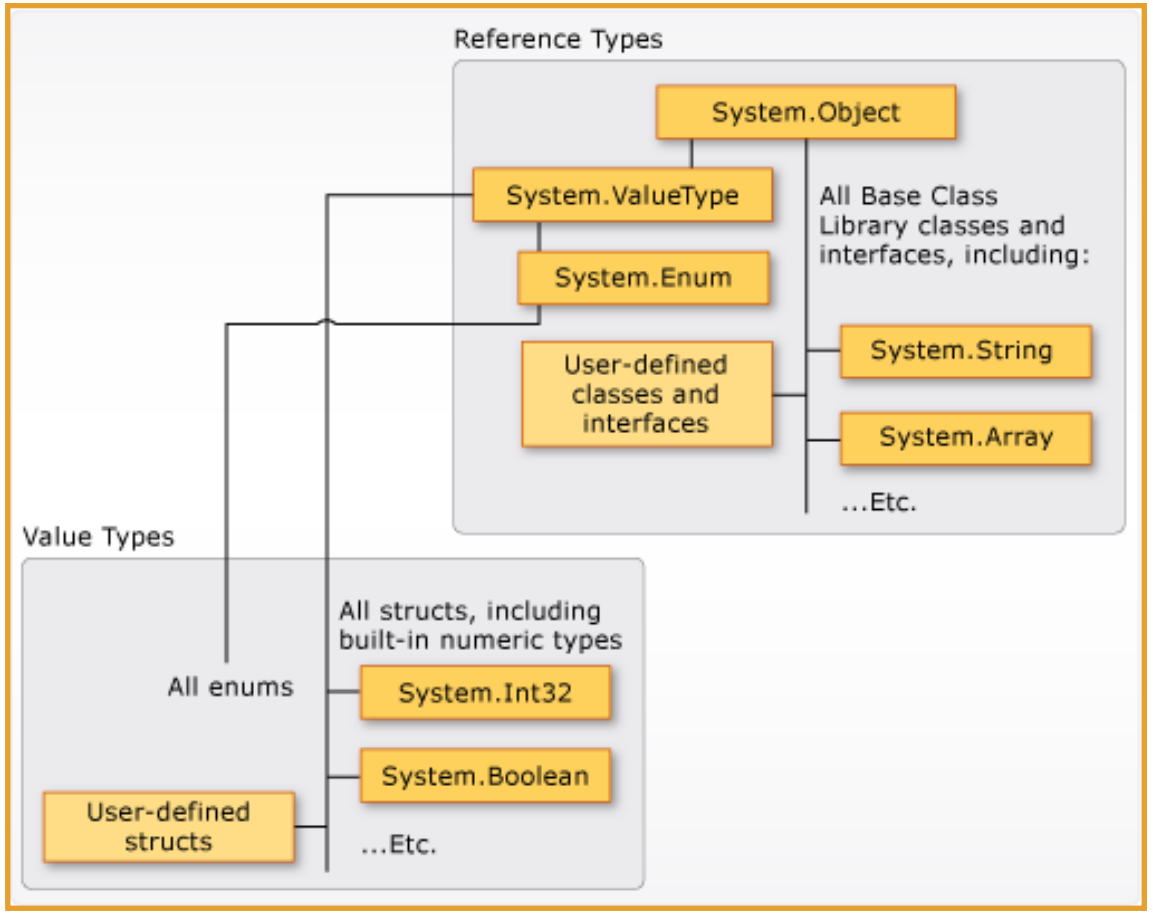
Delegates are similar to function pointers in other programming languages. Unlike function pointers, ***delegates*** are object-oriented and type-safe.

```
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 C#'s ***Unified Type System (UTS)***, 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 ***null***.
When a variable of a value ***type*** is converted to ***Object***, it is boxed*. When a variable of ***type Object*** is converted to a value ***type***, it is unboxed*.



*More on boxing and unboxing later

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();
    }
}
```