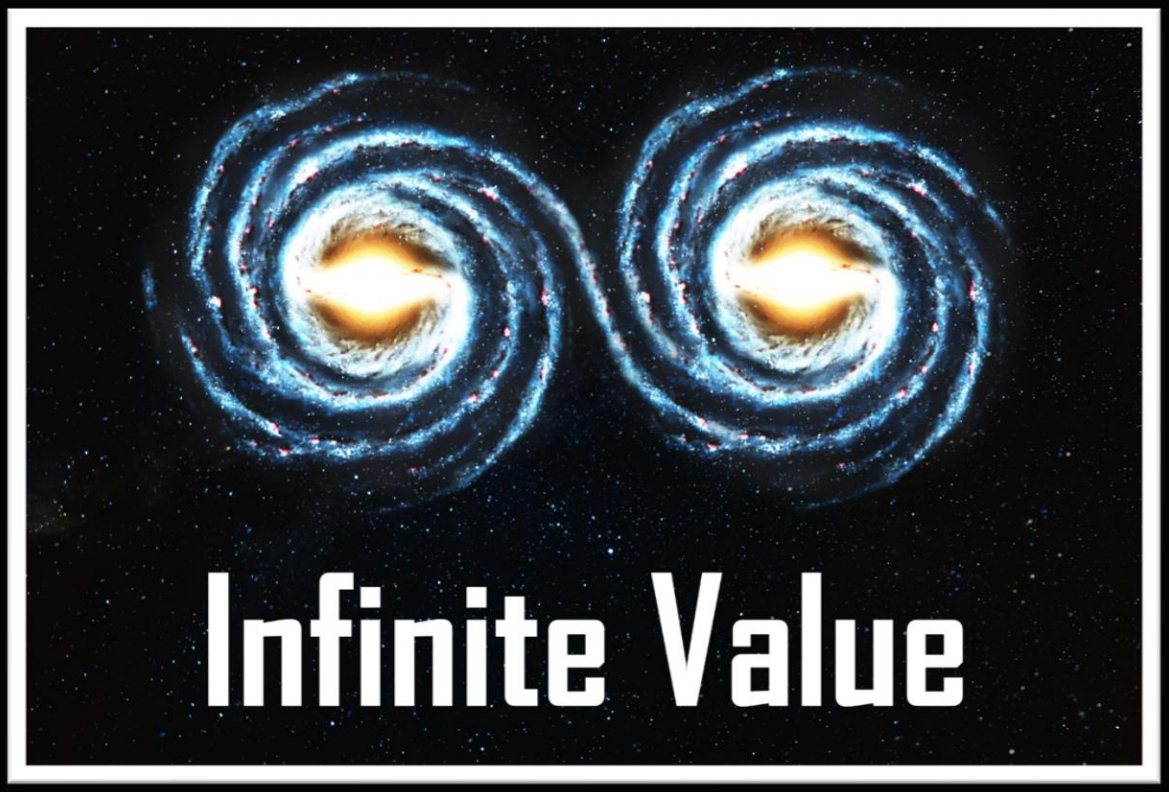


Video tutorials

Play the online demo

Public API

Tables of content

InfVal

Description

InfVal is the core structure of the asset.

It holds 2 private fields, a BigInteger representing digits, and an **int** representing an exponent.
These fields make the **InfVal** able to represent any number, integer or floating point, with a gigantic precision.

$$value = digits \times 10^{exponent}$$

It is similar to a Java BigDecimal with an exponent instead of a scale (exponent = -scale).

Usage

The **InfVal** type is part of the **InfiniteValue** namespace, you should specify it with the using keyword.

```
using InfiniteValue;
```

An **InfVal** is a structure and can therefore be used like any other non-reference value type (int, float, Vector3, etc).

```
InfVal a = default;
InfVal b = a;

// This won't modify a.
b = b + b;
```

On the public side, it is immutable (read-only).
This means you can safely use the in keyword when using an **InfVal** as a method argument. This will result in a performance gain.

```
void Method(in InfVal arg) { ... }
```

Any primitive value type can be implicitly cast to an **InfVal**.

```
InfVal iv = 1.23f;
```

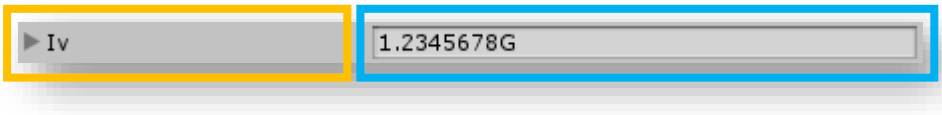
And an **InfVal** can be explicitly cast to any primitive value type.

```
float f = (float)iv;
```

It has multiples constructors, operators, methods and properties that you can use, check the Public API for a description of them.

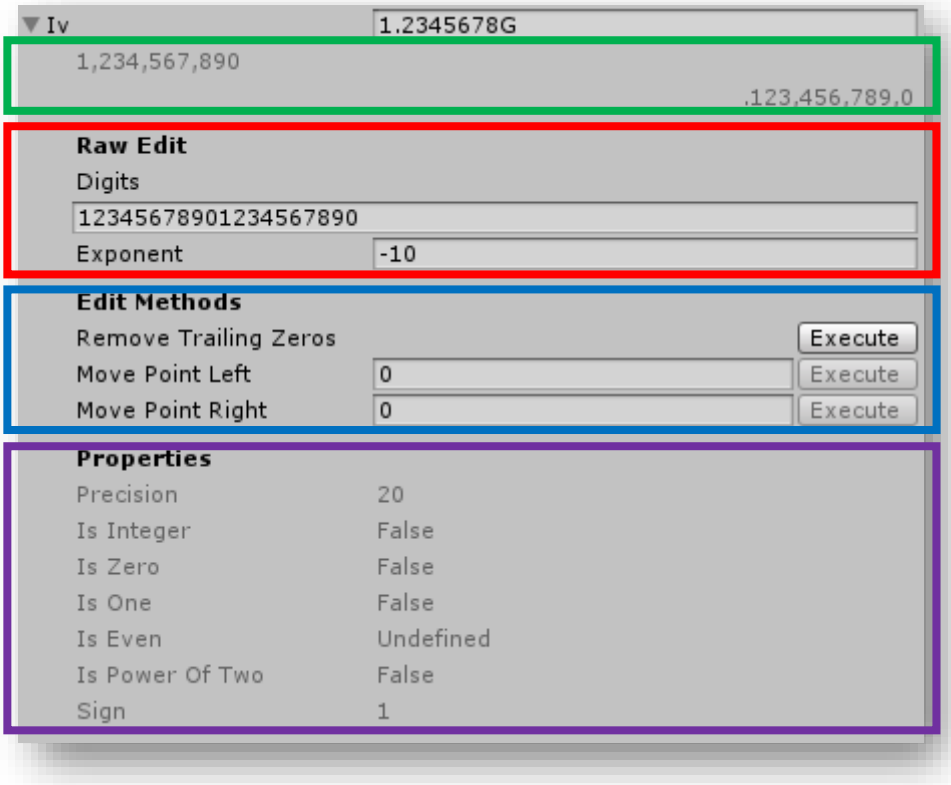
Inspector

The **InfVal** structure is serializable, therefore it will appear in the inspector if it is used as a public field or with the SerializeField attribute.



Click the arrow or double-click the label to unfold/fold the drawer.

Use this field to see or edit the value. The text will be interpreted using the ParseOrDefault method.



This is the full value; the integer and decimal parts are separated.

In here you can edit the digits and exponent directly.

In here you can call methods that will edit the InfVal from the inspector, just type in the argument if need be and click on the *Execute* button.

In here you can see all the InfVal properties also accessible from scripts. "Undefined" means the property threw an Exception.

The inspector drawer can be customized (see Customization).

ToString

Debug

If you want a clear representation of an **InfVal** without any formatting, you can use the **ToDebugString** method.
It will return a **string** with the digits and exponent.

```
InfVal iv = new InfVal(1234.5f, 6);

Debug.Log(iv.ToDebugString());
// output: (123450, -2)
```

Manual format

Like any other type, you can convert an **InfVal** to its **string** representation using the ToString method.
If needed, the **string** representation will be in scientific E notation.

When using **ToString**, you can optionally specify multiples arguments that will change how the value is represented:

- **maxDisplayedDigits**: an **int** that will define the maximum number of digits the **string** will contain. Extra digits will be truncated.
  A negative or zero value means we will display as much digits as needed.

- **unitsList**: a **string[]** that will define the units used to display this value. Units will replace the E and number notation when possible.
  A null or empty array means we will not use any units.

- **displayOptions**: a **bit mask (Flag enum)** that will provide additional constraints on how we should represent the value**.**
  Use the | operator to select the options you want:
  - AddSeparatorsBeforeDecimalPoint
  - AddSeparatorsAfterDecimalPoint
  - ForceScientificNotationOrUnit
  - KeepZerosAfterDecimalPoint

```
InfVal iv = new InfVal(1234.5f, 6);

Debug.Log(iv.ToString(4, null, DisplayOption.None));
// output: 1234

Debug.Log(iv.ToString(0, null, DisplayOption.KeepZerosAfterDecimalPoint |
    DisplayOption.AddSeparatorsBeforeDecimalPoint));
// output: 1,234.50

Debug.Log(iv.ToString(2, new string[] { "k" },
    DisplayOption.ForceScientificNotationOrUnit));
// output: 1.2k
```

If you do not specify any of these arguments, it will use a default value (see Customization).

String format

Instead of providing the 3 arguments detailed above, you can specify a **string** that will act as a format specifier.

This is especially useful when using the string.Format method.

The **string** will be parsed (white spaces are ignored) in order to get all the possible arguments:

- To specify a **maxDisplayedDigits**, simply write a number.
  If you write multiple numbers, only the first one will be used.

- To specify a **unitsList**, write each unit in order inside **'** characters.
  Invalid units will be ignored. A unit is invalid if it is empty, a duplicate, contains a digit, or contains a white space.

- To specify **displayOptions** you need to add specific characters:
  - **_** or **/**: None
  - **<**: AddSeparatorsBeforeDecimalPoint
  - **>**: AddSeparatorsAfterDecimalPoint
  - **E** or **e**: ForceScientificNotationOrUnit
  - **Z** or **z**: KeepZerosAfterDecimalPoint

Any argument not set will use the default value (see Customization).
This means that if you want to use an empty units list, you should add a single **'** character (otherwise it will use the default **unitsList**).

```
InfVal iv = new InfVal(1234.5f, 6);
string s1, s2;

// The next 3 lines are equivalent:
s1 = iv.ToString(0, null, DisplayOption.None);
s1 = iv.ToString("0 ' _");
s1 = string.Format("{0:0 ' _}", iv);

// The next 3 lines are equivalent:
s2 = iv.ToString(3, new string[] { "k", "M" }, DisplayOption.Everything);
s2 = iv.ToString("3 'k' 'M' < > E Z");
s2 = string.Format("{0:3 'k' 'M' < > E Z}", iv);
```

Culture specific

Not every language uses the same characters for the decimal point or separators.
C# provides an easy solution to this problem, the CultureInfo class.

You can provide a **CultureInfo** as the last argument of the **ToString** method to make your value readable by people from any culture.

```
InfVal iv = new InfVal(1234.5f, 6);
string s;

s = iv.ToString(System.Globalization.CultureInfo.CurrentCulture);
```

As always, if you do not specify a culture, the **ToString** method will use a default value (see Customization).

Parsing

If you want to convert a **string** to an **InfVal**, you can do it easily using 3 possible static methods:

- **InfVal.Parse**: This method will return an **InfVal** constructed from the **string** given as an argument. It will throw an Exception if the string is invalid.

- **InfVal.TryParse**: This method will not throw any **Exception** but instead return a **bool**; true if the parsing succeeded and false if the **string** is invalid.
  The constructed **InfVal** is send to the caller as an extra out argument.

- **InfVal.ParseOrDefault**: This method will return an **InfVal** constructed from the **string** given as an argument like the **Parse** method, but it will never throw an **Exception**.
  Instead it will stop the parsing when the **string** becomes invalid.
  For example, parsing "QWE" will return 0, parsing "123eQWE" will return 123, etc.

All 3 methods will ignore white spaces.
Also, adding zeros in the end after the decimal point is useful to add to the **InfVal** precision.

```
Debug.Log(InfVal.Parse("12 34 56").ToDebugString());
// output: (123456, 0)

Debug.Log($"{InfVal.TryParse("123456.00", out InfVal b)} {b.ToDebugString()}");
// output: True (12345600, -2)

Debug.Log(InfVal.ParseOrDefault("12-34").ToDebugString());
// output: (12, 0)
```

You can optionally specify a **unitsList** and a **CultureInfo** to use when parsing the string.
If you do not specify any, the default will be used (see Customization).

```
Debug.Log(InfVal.Parse("12 345,6789k", new string[] { "k" },
    CultureInfo.GetCultureInfo("fr-fr")).ToDebugString());
// output: (123456789, -1)
```

Troubleshooting

Strange results

You may be wondering why this will output 0:

```
Debug.Log(new InfVal(1) / new InfVal(3)); // output: 0
```

This is because an **InfVal** doesn't have a predefined precision, if you want to have more decimal places, then you should specify it like this:

```
Debug.Log(new InfVal(1, 3) / new InfVal(3)); // output: 0.33
// or
Debug.Log(new InfVal(1).ToPrecision(3) / new InfVal(3)); // output: 0.33
```

Arithmetic operations will return an **InfVal** with a precision set to the highest of the two operands.

Float have a default precision of 9, **double** of 17 and **decimal** of 29.

Calling methods do nothing

The **InfVal** structure is readonly (on the public side), this mean that doing this:

```
iv.ToPrecision(3);
```

Will not do anything. If you want to edit an **InfVal** you should assign it like this:

```
iv = iv.ToPrecision(3);
```

Limits

An **InfVal** isn't truly infinite because of technical reasons and does have maximum values:
It can theoretically contain any value between $10^{4294967294}$ and $-10^{4294967295}$.
With the smallest possible difference between two values (epsilon) being $10^{-2147483648}$.

The maximum precision (number of digits) is 2,147,483,647. Note that you probably won't be able to make an **InfVal** this precise because it would take multiples Giga of RAM.

Performances

Best thing to do if you are running into issues regarding performances is to have a set precision for your **InfVal**.

To do this, all you have to do is define a precision every time you set/create your **InfVal**:

```
InfVal iv;

// The next 3 lines are equivalent:
iv = new InfVal(0).ToPrecision(256);
iv = new InfVal(0, 256);
iv = ((int)0, 256);
```

This should be enough to ensure that your **InfVal** will have a constant precision because arithmetic operations and most **MathInfVal** methods will return a structure with the highest precision of the operands.
If you do anything that changes the precision, you can simply set it back to what you want using **ToPrecision**.

Note that the precision doesn't have to be the same as the number of digits you choose to display, in most cases it should be greater.

If the above solution cannot be used or isn't enough, you can follow these general tips:

- Cache results as much as possible.
  For example, instead of using the **ToString** method every frame, use it only when the **InfVal** changes and store the result in a field.

- Make sure to use the in keyword when using an **InfVal** as a method argument.

- Use the **RemoveTrailingZeros** method to reduce your **InfVal** digits count when possible (this will not change the value but can change the exponent and precision).

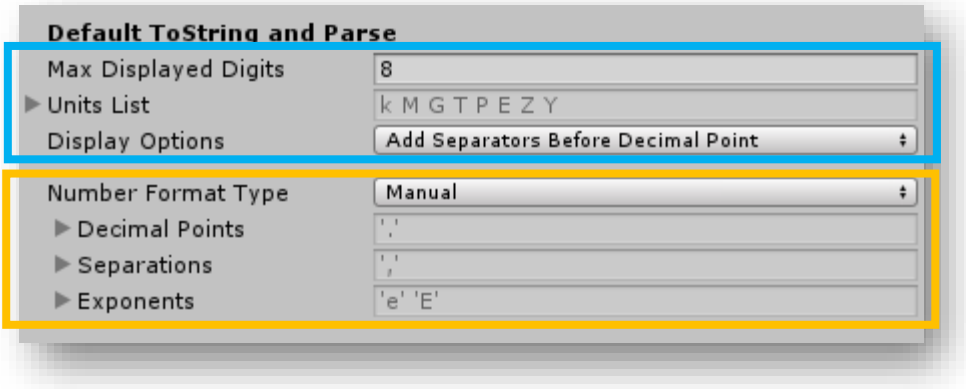
Customization

Configuration file

You can set default values used by the **InfVal** structure ToString and Parsing methods or customize the inspector drawer by editing the **Configuration** file.

You can open it by going into the **top bar > Tools > Infinite Value > Configuration**, or by clicking on it at **InfiniteValue/Resources/Configuration**.

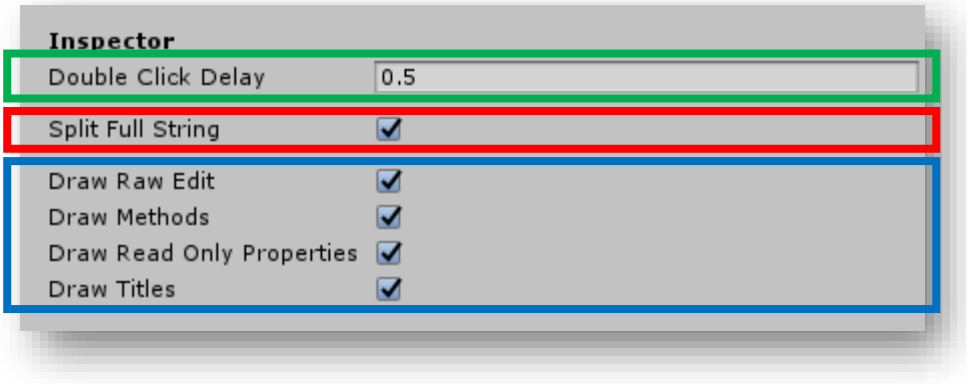
Default string representation



Choose here the default argument to use when formatting the return of the ToString method (see Manual format).

Choose here the default special characters/strings used to display and parse a string from an InfVal. You can set them manually or automatically from a C# culture (see Culture specific).

Inspector drawer personalization



Maximum delay for 2 clicks to be considered a double click (you can unfold/fold an InfVal drawer by double clicking on the label).

Should we separate the InfVal integer and decimal parts when drawing its full string representation.

Define which category should we display or not, and whether they should have a title.

define

The **InfVal** is also customizable through the usage of preprocessor directives.

If you are unsure whether you should change this or not, do not do it. The default behaviour should be good for most usages.

Implicit cast from string

By default, you can cast a **string** into an **InfVal**, but only explicitly (this will use the ParseOrDefault method).

```
InfVal iv = (InfVal)"1.23";
```

You can make it implicit if you want, this will let you write:

```
InfVal iv = "1.23";
```

But it can produce unexpected results, for example this:

```
Debug.Log(new InfVal(1) + " " + new Infval(2));
```

Will be interpreted as 1 + 0 + 2 and therefore output “3” in the console.

To change this behaviour, uncomment or comment this line at the top of the **Infinite Value/Core/InfVal (Constructors, Casts).cs** file:

```
#define CastFromStringIsImplicit
```

IConvertible implementation

If you intend on using the Convert class with an **InfVal**, then you would need the structure to implement the IConvertible interface.

This is not the case by default because it would add a lot of unnecessary entries that would clutter the methods list when using intellisense in Visual Studio.

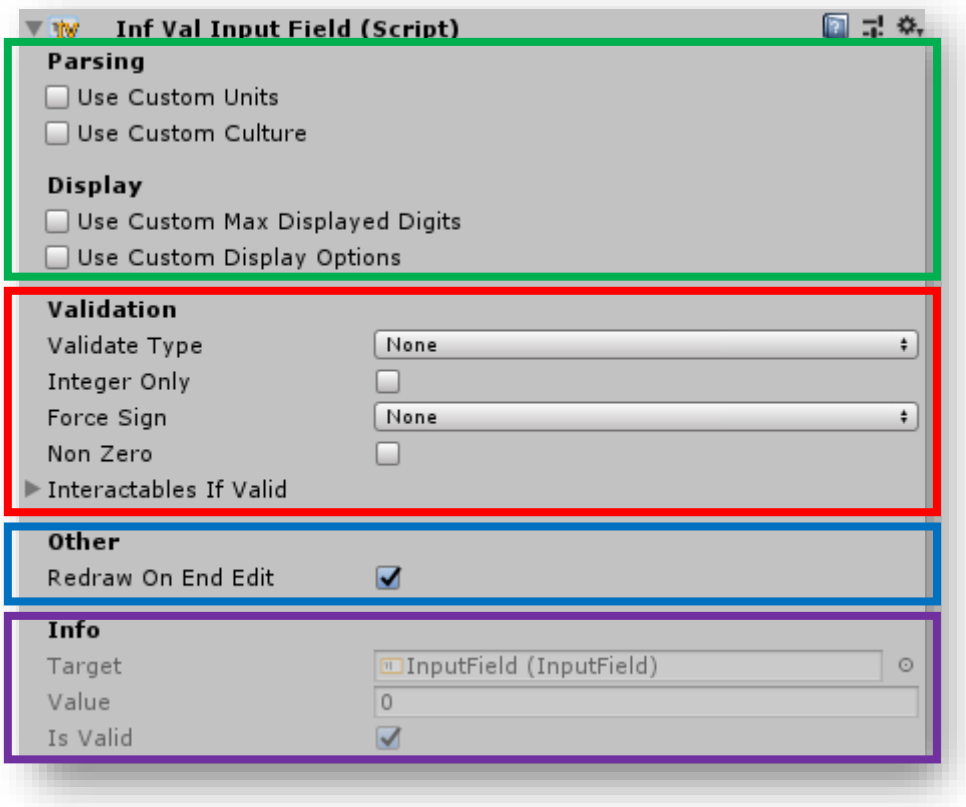
To change this behaviour, uncomment or comment this line at the top of the **Infinite Value/Core/ InfVal (Convert).cs** file:

```
#define DoConvertible
```

Input Field

If you want the player to input an **InfVal**, you can use the **InfValInputField** component.
It works by using an existing Unity InputField or a TMP_InputField and provides extra features relative to an **InfVal**.

To add one to your scene, simply create a new input field GameObject (**right click > UI > InputField** or **TextMeshPro - InputField**) and add the **InfValInputField** component to it (**Add Component > UI > InfVal Input Field Converter**).



Use these fields to override the default string representation if you want to.

Here you can define what should be considered a valid InfVal representation and what to do depending on if it is valid or not.

Check this box if you want to replace the inputted text with a clean representation when the player is done typing.

Here you can see the current state of the InputField.

The **InfValInputField** component is accessible though scripting.
Check the Public API for more information.

Other classes

MathInfVal

The **MathInfVal** static class is like the Math or Mathf classes but dedicated to **InfVal**.
It includes more than 30 methods allowing you to do mathematics operations on an **InfVal**.
Check the Public API for more information.

InterpolateInfVal

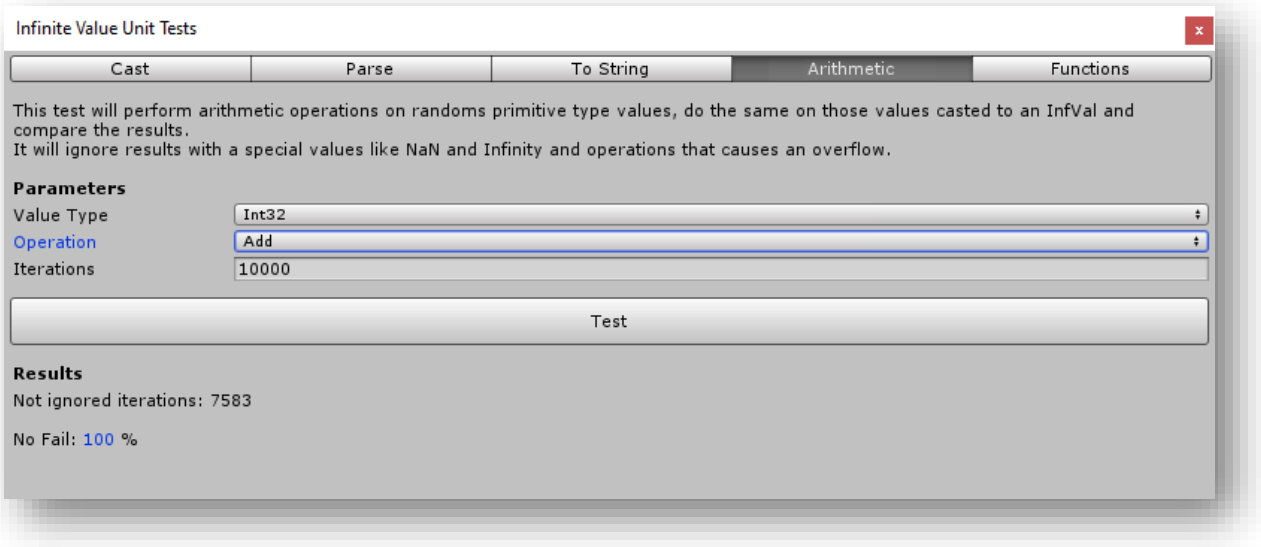
The **InterpolateInfVal** static class is dedicated to interpolating **InfVal** values.
It contains methods comparable to Unity Lerp and SmoothStep but also extra methods allowing for different interpolation.
Check the Public API for more information.

Culture

The **Culture** class let you serialize a **CultureInfo** so that it can be displayed in an inspector and saved with the scene.
It is used by the **Configuration** and the **InfValInputField**.
You can use it too, simply use the **.info** property of the **Culture** instance in places where you would use a **CultureInfo**.

Unit tests window

The asset comes with a unit tests window. You can access it by going into **top bar > Tools > Infinite Value > Unit Tests**.



It is a powerful tool that will test different thing you can do with an **InfVal** automatically using random values.
I used it during development to verify that everything was working well.

You may want to use it if you change some scripts to make sure you did not break anything!

Demo

The asset comes with a complete demo to showcase what you can do with an **InfVal**.
You can try it online here.

It is a clicker type game easy to extend with a bunch of useful features like a save system, a database system that will load scriptable objects and text files, a simple audio system, and many useful UI related scripts! Check it out if want a clear idea on how to use an **InfVal**.
A list of every class from the demo is available in the Public API.

If you want to create a clicker game, you can definitely use it as a starting point for your project. Otherwise, you may want to delete the **Demo** folder entirely in order to remove unused assets from your project.

Author

Created entirely by *Lucas Sarkadi* (**Juste Tools**).

Website: https://justetools.com
Other assets: https://assetstore.unity.com/publishers/52427

This asset is fully compatible with my other asset Concurrent Value.

Contact

If you encounter any bugs, or for any questions regarding the asset, please feel free to contact me directly at:
justetools@gmail.com

or by going to this page:
https://justetools.com/contact