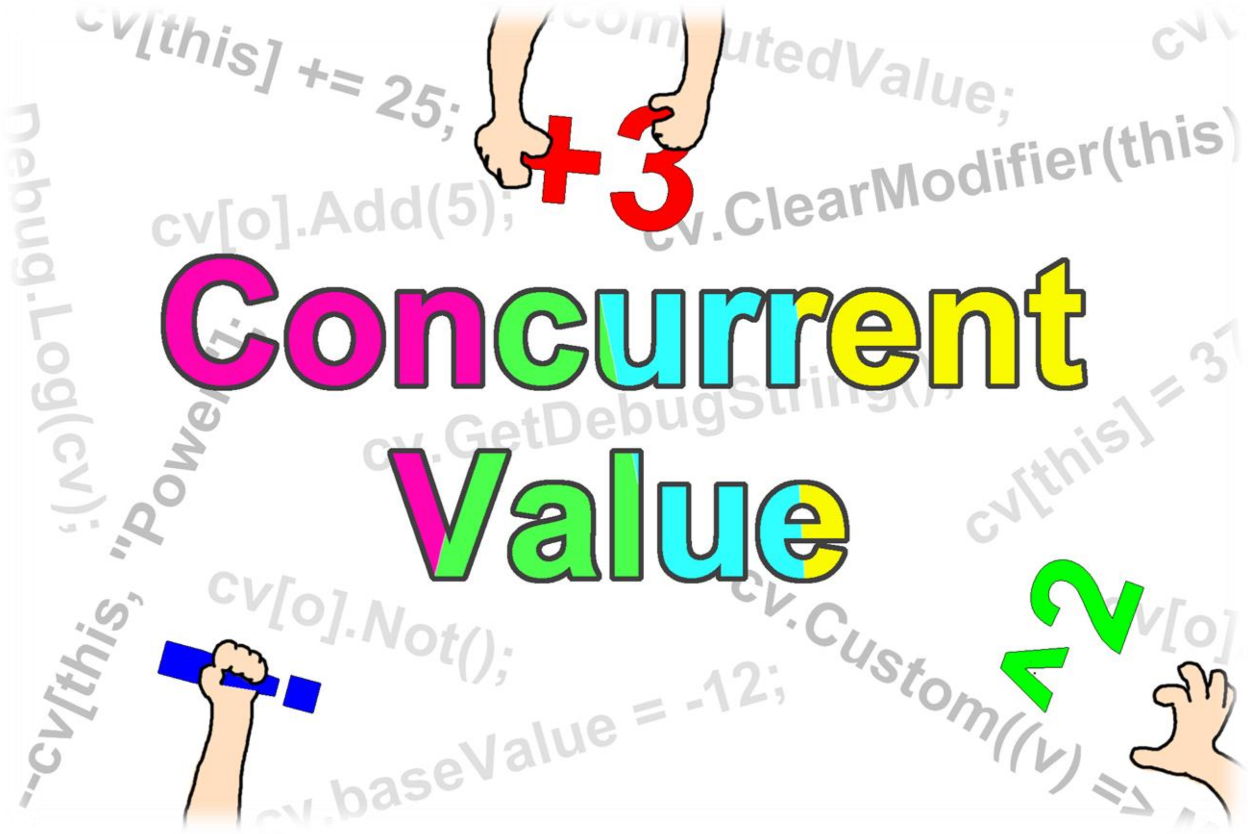
Concurrent Value
+3
^2
cv[this] += 25;
cv.computedValue;
cv.ClearModifier(this)
cv[o].Add(5);
Debug.Log(cv);
"Power"
cv.GetDebugString();
cv[this] = 37
cv[o].Not();
cv.Custom((v) =>
cv.baseValue = -12;
--cv[this,

Video tutorials

https://www.youtube.com/playlist?list=PLGeFTP0u31pE4o3QmJHHWP2y3l3OqCj9N

Summary

Why use a Concurrent Value ?

Let's say for example that you want to have two MonoBehaviour A and B affecting the value of a Rigidbody gravityScale which start at 1.
A will increase the scale by 0.5, and B will multiply the scale by 2. The intended resulting gravityScale would be:

- If neither A or B is active, 1
- If A only is active, 1.5
- If B only is active, 2
- If both A and B are active, 3

Problem is depending on when you activate and deactivate A and B you may end up with different gravityScale, for example if you were to activate B before A you would end up with a gravityScale of 2.5 (1 * 2 + 0.5) which isn't intended.
If you then deactivate B you will end up with a gravityScale of 1.25 (2.5 / 2), etc.

A concurrent value fixes this issue by considering which component made modifications and in which order they should be applied !

How to use

Instantiating a Concurrent Value

To create a Concurrent Value you can use the generic (template) class with the ValueType (primitive types like int or a struct) you want to use:

```
public Concurrent<int> concInt = new Concurrent<int>(5);
```

Because Unity use to not be able to serialize generic class, if you are on a Unity version older than 2020.1 and want to see the Concurrent in the inspector you should use a non generic class inheriting from a Concurrent of the type you want to use.

```
public ConcurrentInt concInt = new ConcurrentInt(5);
```

The asset comes with 6 of them for some commonly used types: bool, int, float, Vector2, Vector3 and Vector4.
You can create more of them manually or automatically by using the auto extend window (recommended).

Note that a Concurrent is a reference type, if you want to copy a Concurrent you should do:

```
ConcurrentInt copy = new ConcurrentInt(concInt);
```

Usage

A Concurrent Value will generate a computed value by starting with a base value and applying to it some modifiers.
You can get or set the base value by accessing the *baseValue* property and you can get the compute value by accessing the *computedValue* property.

A Concurrent will also implicitly cast to its computed value:

```
int variableToAssign = concInt;
```

Edit modifiers

A modifier is an operation that will be applied to the Concurrent value tied to a UnityEngine Object (usually the calling object, « this »). To create or edit one you can use the *EditModifier* method:

```
concInt.EditModifier(this, UnaryOperator.Invert);
```

You can specify a unique identifier string, useful if you want the same UnityEngine Object to have multiples modifiers:

```
concInt.EditModifier(this, "Unary Op", UnaryOperator.Invert);
concInt.EditModifier(this, "Binary Op", BinaryOperator.Assign, 5);
concInt.EditModifier(this, "Custom Op", (int i) => Mathf.Max(i, 25));
```

You can also specify an order (between -128 and 127) as the last method argument. Modifiers will be applied in ascending order.

```
concInt.EditModifier(this, UnaryOperator.Invert, 10);
```

If you want to disable modifiers, you can do it by accessing the *ignoreModifications* property.

```
concInt.IgnoreModifications = true;
```

Delete Modifiers

If you want to delete a modifier, you can use the *ClearModifier* method (safely usable if the modifier doesn't exist):

```
concInt.ClearModifier(this, "Unique Identifier");
```

You can also delete all modifiers coming from a UnityEngine Object using the *ClearAllModifiers* method. This should be done when the object goes out of scope, in a MonoBehaviour you could do:

```
void OnDisable()
{
    concInt.ClearAllModifiers(this);
}
```

Additionally, you can filter the modifiers list using the *FilterModifiers* method, leaving only those that match a Predicate you provide as an argument.
If you don’t provide a predicate, the method will delete every modifiers.

```
// Delete every modifier that doesn't come from this UnityEngine Object.
concInt.FilterModifiers((m) => m.from == this);

// Delete every modifier that have a unique key identifier.
concInt.FilterModifiers((m) => m.uniqueKey == null);

// Delete every modifier that have a negative order.
concInt.FilterModifiers((m) => m.order >= 0);

// Delete every unary operator modifier.
concInt.FilterModifiers((m) => !(m.type == OperatorType.Unary));

// Do all of the above at once
concInt.FilterModifiers((m) => {
    return m.from == this && m.uniqueKey == null &&
           m.order >= 0 && !(m.type == OperatorType.Unary);
});

// Delete every modifiers.
concInt.FilterModifiers();
```

Quick syntax

You can also edit a modifier using a quick syntax, it will have a cost on performances but you may find it increase your script clarity:

```
concInt[this, "Unary Op"].Not();
concInt[this, "Binary Op"].Set(5);
concInt[this, "Custom Op"].Custom((int i) => Mathf.Max(i, 25));
```

If you are editing the modifier to be a binary operation, you can also use syntax like this:

```
concInt[this, "Assign"] = 5;
concInt[this, "Add"] += 35;
```

Note that this assign syntax will only work with a ValueType (a struct or a primitive value type).

You can also specify an order when using the quick syntax:

```
concInt[this, "Unary Op"].Not(10);
concInt[this, "Binary Op"] += (35, 10);
concInt[this, "Custom Op"].Custom((int i) => Mathf.Max(i, 25), 10);
```

Finally, if you want to delete a modifier you can do:

```
concInt[this, "Unique Identifier"].Clear();
```

or

```
--concInt[this, "Unique Identifier"];
```

Other functionality

If you want to get notified every time the computed value changes, you can subscribe to the *OnValueChanged* event.
Note that the event is called whenever a new process is necessary; the computed value may not have changed if you create a modifier that won't affect the value (like adding 0).

```
concInt.OnValueChanged += () => Debug.Log($"Value changed to: {concInt.computedValue}");
```

For debugging purposes, you can generate a string showing how the value was processed by calling the *GetDebugString* method (also displayed in the inspector).
Note that this method shouldn't be used in the final build as it will produce extra computations.

```
Debug.Log(concInt.GetDebugString());
```

A concurrent is also able to serialize its modifiers list and *OnValueChanged* event; you can activate it by accessing the *serializeModifiersAndEvent* property. This is useful when working with scripts executing in edit mode: if you add some modifiers or edit the *OnValueChanged* event in edit mode, the changes will persists when launching the game.

```
concInt.serializeModifiersAndEvent = true;
```

Calculator

Being a generic (template) class, a Concurrent cannot natively handle arithmetic operations.
To deal with this issue, the asset uses a Calculator class that will be used by a Concurrent class to apply operations to the value.

Any types Calculators

The asset comes with 2 calculators that can be used for any type:

The ExpressionCalculator, which rely on LINQ Expressions to dynamically generate functions handling all arithmetic operations.

And the DynamicCalculator which rely on C# dynamic type to do arithmetic operations at runtime on any types. Note that using the dynamic keyword require the API compatibility level to be .NET 4.x (you can set it by going into Edit > Project Settings > Player > Other Settings).

Specific Calculators

Those 2 calculators are very useful for fast prototyping or if getting the best performance isn't a necessity, but if you want the fastest processing possible you will need a Calculator unique to the value type you want to use.

This is done by inheriting from the generic class ASpecificCalculator. It can be done manually or automatically by using the auto extend window (recommended).

The asset come with 6 default calculators for some commonly used types: bool, int, float, Vector2, Vector3 and Vector4.

Inheriting from a Calculator also allow you to change operations sequence for two modifiers with the same order. This is done by overriding the compare methods (*OperatorTypeComparer*, *UnaryOperatorComparer* and *BinaryOperatorComparer*).

If you don't do it, the default operations order will be:

- Binaries: Assign, add, substract, multiply, divide, modulo, and, or, xor, left shift and then right shift.
- Unaries: Plus, negate, invert, ones complement, increment and then decrement
- And finally custom operators.

Choosing a Calculator

You can specify which calculator to use when instantiating a Concurrent by passing the type of the Calculator as an argument.

```
    public Concurrent<int> concInt = new Concurrent<int>(typeof(ExpressionCalculator<int>));
```

If you don't specify or if the type isn't a valid Calculator type for this value, the Concurrent will use the default for its value type (defined by the *typeNameToCalculatorNameDico* dictionnary in the "Extends/DefaultCalculators.cs" file). If the value type doesn't have a default Calculator it will use either an ExpressionCalculator or a DynamicCalculator (defined by the *onNoDefaultUseExpressionCalculator* boolean in the same file).

Note that if you inherit from Concurrent<T>, the constructor will be overloaded and therefore you may not be able to choose the Calculator to use.
For example, the ConcurrentInt will only use the default Calculator for an int value (the IntCalculator).

Windows

You can access all the windows in the top bar by clicking on Tools > Concurrent Value.

Help

In the Help window you will find quick instructions on how to use the asset, like you can see in this documentation but shorter.

There is also a list of all modifiers for a quick reminder of their possible usages.

Auto Extends

The auto extends window allow you generate automatically new script classes related to the asset.
It is recommended you use it over manually extending to limits errors.

In the New Concurrent tab you can generate a new Concurrent<T> inheriting class, it is especially useful if you are on a Unity version older than 2020.1 and want to see the concurrent value in the inspector.

In the New Calculator tab you can create a new specific calculator; this is useful for increasing performances compared to the Expression or Dynamic calculators. You can set the calculator to be the default for its type.

The Other tab let you set or unset a Calculator as the default for its type or delete Concurrents and Calculators.

Benchmarks

In the benchmarks window you can try out different performances test:

The specific vs expression vs dynamic test will try computing the value of the same Concurrent by using different Calculators. It shows that using a specific Calculator is the most performant option.

The normal edit vs quick access test will edit modifiers of a Concurrent using the normal method (*EditModifier()*) or using the quick syntax. It shows that using the quick syntax has a slight cost on performances, but it can be a fine tradeoff for the clarity that it provides your code.

Finally the Concurrent vs no Concurrent test will execute the same arithmetic operations using directly a value type or using a Concurrent of the same type. It shows that despite all of its capabilities, using a Concurrent keep will let you process a value in the same order of magnitude of time as not using one.

Manually Extending

Concurrent

To manually create a Concurrent, you can generate a new Concurrent using the auto extends window and edit it afterwards, or simply create a new C# script where you define a class inheriting from Concurrent<T>.

```
[Serializable] public class ConcurrentInt : Concurrent<int>
```

Creating a new Concurrent type is especially useful on Unity version older than 2020.1 so that you can see the Concurrent serialized in the inspector.

You should define some constructors for your custom Concurrent that will call the base Concurrent<T> constructors, for example:

```
public ConcurrentInt() : base() { }
public ConcurrentInt(int baseValue) : base(baseValue) { }
public ConcurrentInt(ConcurrentInt toCopy) : base(toCopy) { }
Public ConcurrentInt(ConcurrentInt toCopy, int baseValue) : base(toCopy, baseValue) { }
```

This is where you can define which calculator to use for this Concurrent Value, either one you specify explicitly (using *typeof*), the default one for this value type (by not passing an argument), or let it be a parameter so that you can choose which calculator to use when instantiating the Concurrent Value.

Note that if you manually create the Concurrent and still want the auto extend window to work with it, you will need to have the file name be the same as the class name, and put the file in the "Extends/Concurrents/" folder.

Calculator

Same thing for manually creating a Calculator, you can generate a new one using the auto extends window and edit it afterwards, or simply create a new C# script where you define a class inheriting from ASpecificCalculator<T>.

```
public class IntCalculator : ASpecificCalculator<int>
```

Creating a new Calculator type let you get the best possible performances, allow you to define each possible operation the way you want it and let you override the order of operations.

You need to override every operator methods:

```
protected override int Decrement(int val) => --val;
protected override int Add(int val, object parameter) => (int)(val + (int)parameter);
// etc...
```

You can optionally override the compare methods (*OperatorTypeComparer*, *UnaryOperatorComparer and BinaryOperatorComparer*) to change the sequence of operations when processing:

```
protected override int BinaryOperatorComparer(BinaryOperator x, BinaryOperator y) => x - y;
```

Additionally, you can override the *twoCancelsOutUnaryOperators* list, this list is used to optimize the process of unary operations:
If an unary operator is in this list, then it will be processed once or none if there is an odd or even number of modifier applying it.
This is useful for any unary operation that cancels itself if used twice.

```
protected override IReadOnlyList<UnaryOperator> twoCancelsOutUnaryOperators =>
    new List<UnaryOperator> { UnaryOperator.Plus, UnaryOperator.Negate, };
```

Note that if you manually create the Calculator and still want the auto extend window to work with it, you will need to have the file name be the same as the class name, and put the file in the "Extends/Calculators/" folder.

Default

To make a calculator a default for its value type, all you have to do is add an entry to the *typeNameToCalculatorNameDico* dictionary in the "Extends/DefaultCalculators.cs" file.

```
public static readonly IReadOnlyDictionary<string, string>
    typeNameToCalculatorNameDico = new Dictionary<string, string>()
    {
        { "System.Int32", "ConcurrentValue.IntCalculator" },
    };
```

The left string should be the full (with namespace) type name of the value that Calculator is for, and the right string the full type name of the Calculator.

If the same type name exists in multiples assemblies, you should specify which one to use by adding the assembly name after a coma, like this:

```
        { "System.Int32, mscorlib", "ConcurrentValue.IntCalculator" },
```

If you enter an invalid name, you will get a warning explaining the issue.

Public API

Concurrent<T>

Properties / Fields

Type	Name	Description
T (generic)	baseValue	The value before any modifiers are applied.
T (generic)	computedValue	The value after all modifiers are applied.
bool	ignoreModifications	Define if modifiers are ignored. If true the computedValue will be the baseValue.
Action (event)	OnValueChanged	Event called whenever the computedValue changes.
bool	serializeModifiersAndEvent	Define if modifiers and the OnValueChanged event are serialized. Useful if you are working with scripts executing in edit mode.

Methods

Type	Name	Description
void	EditModifier	Create or edit a modifier.
void	ClearModifier	Delete a modifier.
void	ClearAllModifiers	Clear all modifiers coming from a UnityEngine Object
Void	FilterModifiers	Filter modifiers leaving only those that match a given predicate.
string	GetDebugString	Get a string showing how the value was processed. This will create extra computations and shouldn't be used in the final product.
void	ForceProcess	Force a reprocess next time you get the computedValue, this is only useful if you are dealing with custom operations whose results can differ for the same parameter value.
Void	SetCalculator	Set the Calculator.

Nested Class

The Concurrent<T> class also publicly expose a nested class *QuickAccess* used for the quick syntax.

This class cannot be instantiated, and the [] accessor shouldn't be considered a factory method:
Using a *QuickAccess* instance directly will result in unexpected behavior; you should only use this class the way it is detailed in the next section.

QuickAccess

Binary operations

Operation	Quick Method	Quick Syntax
CV = X	CV[...].Set(X)	CV[...] = X
CV + X	CV[...].Add(X)	CV[...] += X
CV - X	CV[...].Sub(X)	CV[...] -= X
CV * X	CV[...].Mul(X)	CV[...] *= X
CV / X	CV[...].Div(X)	CV[...] /= X
CV % X	CV[...].Mod(X)	CV[...] %= X
CV & X	CV[...].And(X)	CV[...] &= X
CV | X	CV[...].Or(X)	CV[...] |= X
CV ^ X	CV[...].Xor(X)	CV[...] ^= X
CV << X	CV[...].Left(X)	CV[...] <<= X
CV >> X	CV[...].Right(X)	CV[...] >>= X

Unary operations

Operation	Quick Method	Quick Syntax
+CV	CV[...].Pos()	/
-CV	CV[...].Neg()	/
!CV	CV[...].Not()	/
~CV	CV[...].Comp()	/
++CV	CV[...].Inc()	/
--CV	CV[...].Dec()	/

Other

Operation	Quick Method	Quick Syntax
Func(CV)	CV[...].Custom(Func)	/
(Clear the modifier)	CV[...].Clear()	--CV[...]

Optimizations

The Concurrent Value class features some optimization to make it run as fast as possible:

The computed value is processed only when requested and then cached until a modifier is edited.

Trying to edit a modifier to be what it already is won't force a new process of the computed value, this mean that spamming the same exact modifier every frame will have almost no impact on performances. Also, changing only the parameter of a modifier (but keeping the same operator type) is faster than creating a new modifier as it will not edit the collection used for an ordered processing.

A binary operator modifier parameter can either be of the same type as the Concurrent type or be a generic object. It allow you to use a value of a different type than the one of the Concurrent as a parameter (for example multiplying a Vector3 by a float), while still preventing the need for boxing and unboxing when it's possible.
The C# language will automatically choose the correct type if the parameter value can be implicitly cast, but if it can only be explicitly cast to the concurrent value type, doing it yourself will get you a slight performance gain (For example, casting a float to an int when using it as a parameter for an int Concurrent).

When processing the computed value, the calculation will start on the highest order that includes an assign operator. This speeds up calculation as it will ignore any operations that would be overridden by a later assignation.
This mean you should not use the custom operator if the function doesn't depend on the parameter value (if it's just an assignation) and instead use an assign modifier with the parameter being the result of the function.
It also means that processing the value may work even if one of the modifiers throws an error (if it was skipped because of an assign modifier). The debug string (displayed in the inspector) will still show the error.

Finally, some unary operations that cancels out when applied twice (like negate) will only be applied once if needed.
This behavior can be changed by overriding the *twoCancelsOutUnaryOperators* list in a Calculator.

If you have any propositions on how to make the class run faster, please feel free to message me about it. I will implement them as soon as I can.

Possible errors and fix

- **If you get an error « Enable .NET 4.x in the player settings to use the DynamicCalculator. » when using a DynamicCalculator.**

You should set the API compatibility level to be .NET 4.x by going into Edit > Project Settings > Player > Other Settings, or use another type of Calculator like the ExpressionCalculator.

Contact

Created by **Juste Tools**.
justetools.com

If you encounter any errors not covered in this documentation, have any suggestion or for any other questions please feel free to contact me at:
justetools@gmail.com

I hope you will enjoy this asset and create awesome games with it ☺