# Intro

## What are Expressions?

ExpressionAnimations (or Expressions, for short) are a new type of animation introduced to Windows App developers in Windows 10 to provide a more expressive animation model than what is provided from traditional KeyFrameAnimations and XAML Storyboards.

Expressions are mathematical equations and relationships that are defined by the developer and used by the system to calculate the value of an animation property each frame. These equations can be used to define relationships between objects such as relative size to more complex UI experiences such as Parallax, Sticky Headers, and other input-driven experiences.

The documentation below assumes you are familiar with the Composition and CompositionAnimation APIs, including Expressions. For more information on these, check out the following resources:

* [Composition Overview](https://msdn.microsoft.com/windows/uwp/graphics/visual-layer)
* [Composition Animation Overview](https://msdn.microsoft.com/windows/uwp/graphics/composition-animation)
* [Windows UI Dev Labs Github](https://github.com/Microsoft/WindowsUIDevLabs)
* [ExpressionAnimation MSDN Documentation](https://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.composition.expressionanimation.aspx)

## Why ExpressionBuilder?

To use ExpressionAnimations today, developers are required to write their mathematical equation/relationship in a string (example shown below).



This experience presents a series of challenges for developers:

* No Intellisense or auto complete support.
* No type safety when building equations.
* All errors are runtime errors, many of which are desirable to be detected at compile time.
* Working with strings for complicated equations not intuitive or ideal.

To improve the Expression authoring experience, the team put together the ExpressionBuilder classes, which act as a series of “helper classes” to bring type safety, Intellisense, and compile-time errors to the Expression-building experience. The classes can be used as an alternative experience to building Expressions than what is available today.

# How to: Build Core Components of Expressions

The following sections will cover how the classes work to create the core components of an Expression.

*Note:* Each section will provide short code lines that implement the Expression Builder syntax to demonstrate the core concept. These are not meant to be E2E working samples - E2E walkthroughs using the Expression Builder Classes are provided in [Section 4](#_E2E_Building_Examples).

## General Construction

*[What is the general model for building Expressions with the classes? How do the classes plug into existing entry points for CompositionAnimations? (StartAnimation, ExpressionKeyFrames)*

Before talking about the core components of Expressions and how the ExpressionBuilder classes achieve them, let’s first discuss how to think about these classes, their general architecture and how to integrate into your existing app code.

### Interacting with ExpressionNodes via Static Methods

To start, let’s talk about how Expressions are represented in the Helper Class. When using the Expression Builder Classes, developers will be generating ExpressionNodes – a single node, or a combination of nodes, can be used to define an Expression. It is important to note that you do not need to “new-up” ExpressionNodes; instead, there are a series of static methods to create typed Expression nodes such as::

* ScalarNode
* Vector2Node
* Vector3Node
* Vector4Node
* ColorNode
* QuaternionNode
* Matrix3x2Node
* Matrix4x4Node

For example, to generate a Vector3ExpressionNode, use a static method:



Don’t worry about understanding the syntax right now, we’ll cover that later 😊. For now, understand that you will create most ExpressionNodes using static methods off two main classes:

* ExpressionValues
* ExpressionFunction

As stated earlier, a combination of ExpressionNodes can be used to define an Expression. Like any math equation, ExpressionNodes can be combined using mathematical or logical operators and mathematical functions (more on this later).

For example, the code snippet below shows adding together two Vector3Nodes that will result in a Vector3Node (invalid combinations will be caught as compile-time errors).



### Implicit Conversion to ExpressionNodes

In addition to using the static methods to generate ExpressionNodes, the classes will also handle implicit conversion of numerical values (e.g. System.Numerics) to the appropriate ExpressionNode type. This is done so you do not need to explicitly create new nodes for existing Numerics objects you have already defined.

For example, if you have already defined a System.Numerics object, you can use it directly when combining with other ExpressionNodes:



When building your Expression with ExpressionNodes, you can use a numerics type anywhere an ExpressionNode would normally be used and it will get implicitly converted to one. For example, the math function Length(…) takes in a QuaternionNode, if a System.Numerics.Quaternion object was provided, it would get implicitly converted:



### Using ExpressionNodes with StartAnimation

Once you’ve created an ExpressionNode using the ExpressionBuilder classes, you need to connect the ExpressionNode to a target (CompositionObject). The ExpressionBuilder classes include an extension method that mimics the publicly available StartAnimation(…) API, but instead of taking in a CompositionAnimation, it takes in an ExpressionNode. This extension method is defined in CompositionExtension.StartAnimation(…), and is accessible via CompositionObject.StartAnimation(…).

The following uses the ExpressionNode defined earlier and attaches it via StartAnimation to a Composition Visual:



### Using ExpressionNodes with ExpressionKeyFrames

In the existing CompositionAnimation API, you can also use Expressions with KeyFrameAnimations. This is done by using an ExpressionKeyFrame where you define the progression point and a string representing the equation (the Expression provided is used by the system to evaluate the keyframe value each frame).

The ExpressionBuilder classes also provides an extension method for the InsertExpressionKeyFrame API that takes in an ExpressionNode instead of a string.

The following defines a KeyFrameAnimation that uses an ExpressionKeyFrame with an ExpressionNode:



Note: In the above example, the Expression only consists of constant parameters for example purposes. Your Expression should always contain at least one reference; an Expression made up of only constant parameters is wasteful as it can be equivalently accomplished with a direct property set via the API .

### Using ExpressionNodes in other places

Using ExpressionNodes with StartAnimation and ExpressionKeyFrames will be the most common places you will utilize Expressions. However, there are other places that Expressions are used today – for each of the cases below, extension methods are provided that will take in an ExpressionNode instead of a string:

* InteractionTrackerInertiaMotion Extension Methods
  + SetCondition
  + SetMotion
* InteractionTrackerInertiaRestingValue ExtensionMethods
  + SetCondition
  + SetMotion

## Parameters

The big value prop to use ExpressionAnimations is that you can define equations and mathematical relationships that utilize constants and reference values from other objects. These objects are often other CompositionObjects or variables that make the mathematical relationship more meaningful. For example, you can use a parameter to create an equation that references another’s object’s x Offset.

There are two types of Parameters: Constants and References. Both Constants and References can be described as either a dynamic or static Parameter - this defines whether you can change what they refer to. These topics will be discussed in more detail in the next sections.

### Definitions: Constants vs. References

There are two types of Parameters that can be used in an Expression. Moving forward, we’ll use the following definitions to distinguish Constants vs References:

* Constant: A typed value (Scalar, Vector2/3/4, etc.), which will be used directly as a literal in the Expression.



* Reference: A CompositionObject (Visual, Clip, InteractionTracker, etc.), whose properties will be evaluated each frame the Expression is processed in the Compositor.
  + The usefulness of including a Reference Parameter in an Expression is to reference properties off it (e.g. referencing the Offset property of a Visual).



### Definitions: Dynamic vs. Static Parameters

Templating, which is discussed in more detail in [section 2.2.6](#_Templating) refers to the reuse of an Expression while changing the values of dynamic parameters. Unless you are in a templating scenario, you will only need static parameters.

* Static: A parameter in which the value or CompositionObject it references will never change
* Dynamic: A parameter in which the value or CompositionObject can be changed without modifying the Expression. It is changed by associating a string parameter name with a new value or CompositionObject. Dynamic parameters are required for templating.

How to create static and dynamic parameters for both Constants and References will be discussed in the next two sections.

### Creating Constant Parameters

How to create *static* Constant Parameters:

* Simply place the value straight into the equation

In the example below, we want to utilize a float constant variable that gets defined earlier in the equation.



Note: Plugging in CompositionObject values directly into the equation will have the same effect, as they are just variable values. For example, plugging in \_visual.Offset will get evaluated to its Vector3 value and treated as a Constant. If you want the equation to use the frame-accurate value of a CompositionObject property, make it either a static or dynamic Reference Parameter.

How to create *dynamic* Constant Parameters:

* Create a named parameter using the static methods off ExpressionValues.Constants class

You can create a constant parameter via static Create\*Parameter() methods (e.g. ExpressionValues.Constants.CreateScalarParameter(“foo”, 7)). Note: setting the intial value as part of the creation is optional; you can always set the value of the parameter using ExpressionNode.Set\*Parameter(). Let’s expand the above example. In this case, let’s say we want to create a generic equation that can be reused for similar scenarios, but tailored by changing the value of constant(s). In the example below, we create the Expression that contains a Constant Parameter, using ExpressionValues.Constant.CreateConstantVector3(…). Before connecting it to a target, the Expression is tailored by setting the parameter using ExpressionNode.SetVector3Parameter(…).



### Creating Reference Parameters

How to create *static* Reference Parameters:

* Call the GetReference() extension method off of the CompositionObject you would like to create a reference for

In the example below, we further expand on the above code such that instead of using a constant Vector3 for the first part of the equation, we will reference the frame-accurate Offset of a Visual:



Thus, as the Offset value of \_redBall changes (via direct property set or animation), so will the output of this equation.

How to create dynamic Reference Parameters:

* Create a Parameter using the static methods off the ExpressionValues.Reference class
  + The value of the parameter can be set via the SetReferenceParameter function

In the example below, we further expand on the above code such that instead of using a static reference to the redBall visual, we use a named one, so that we can change the parameter to refer to a blueBall Visual later.



To refer to a property in a CompositionPropertySet, you will need to get or create a reference to the PropertySet, then call GetParameter() function and pass in the name of the property in the form of a string.



### Subchanneling (Swizzling)

*[Also known as “dotting into things”]*

When using a vector or matrix node type in your equations, you also can access a subchannel of (or “dot into”) the parameter to use an individual component property. For example, when using a Vector3 Constant, developers can dot into its X, Y, or Z component:



In addition, when using Reference Parameters, all the animatable properties on the different CompositionObjects can be subchanneled into as well. For example, modifying the above example to use the offset of a Visual instead of a Vector3 constant:



The classes only provide the common subchannels off the different types. However, in Expressions, you can also reference more complicated subchannels such as XX, XXY, etc. In this case, you can use the Subchannels(…) function to define a particular combination:

In the example below, the developer wants to grab a subchannel reference to an XY component of a Visual’s Offset. The output of this is a Vector2Node:



### Templating

There are times where you want to use a generic Expression across different parts of your app to animate multiple CompositionObjects. However, depending on which target the Expression is connected to, different values for parameters in the equation may be desired.

This is where using named Dynamic Parameters with Constant and Reference Parameters comes into play. When creating parameters using the ExpressionValues.Constant or ExpressionValues.Reference classes, you define a string name that you will later use to set the value of the parameter. For constants, this will be a different variable/value that you want to use. For References, this will be a different CompositionObject that you want to reference.

Note: Whenever a templated Expression is connected to a target (via StartAnimation(…)), an instance of that Expression is created and associated with that target. For this reason, any changes to parameters (via Set\*Parameter(…)) only affect the template and future instances created from that template. For example, take an Expression with a parameter “P” that is connected to three targets: “T1”, “T2”, and “T3” (in order). If the value of “P” is changed after “T1” and “T2” have been connected, this new value will only be used in “T3”.

In the example below, we create a generic Expression and attach to two different Visuals. Each time, we change the value of the parameter before starting the animation.



A real-world example that demonstrates the need for changing the value of a Parameter using diffferent constants would be using the index number of an itemized List as a Constant Parameter.

We can extend this concept by imagining a real-world scenario in which a common equation is needed across many targets: list items. A list is typically comprised of homogeneous items, each with a unique ID indicating its position in the list. Each item needs to be behave very similarly, with slight differences based on its position. For this example, a single Expression could be designed that gives a consistent behavior across all items, but is customized by using the list item ID as a Constant Parameter. When connecting this Expression template to each list item, the ID Parameter is set using SetScalarParameter(…) with the ID of the current list item.

### Keywords

In Expressions, there are several certain keywords that can be used as shortcuts when defining the equation. These keywords are available off of the ExpressionValues object:

* ExpressionValues.Target – This keyword defines a reference to whichever CompositionObject this Expression is connected to.
* ExpressionValues.StartingValue – This keyword defines a reference to the property the Expression targets, sampled at the first frame of execution. Note: if the Expression is connected to a subchannel of a property (e.g. “Offset.X”), then StartingValue will be of the same data type as the subchannel (e.g. ScalarStartingValue for “Offset.X”).
* ExpressionValues.CurrentValue – This keyword defines a frame-accurate reference to the property the Expression targets. Note: if the Expression is connected to a subchannel of a property (e.g. “Offset.X”), then CurrentValue will be of the same data type as the subchannel (e.g. ScalarCurrentValue for “Offset.X”).

In the example below, we create an Expression using the Target keyword:



The usage of the Target keyword here is a shortcut to a Reference Parameter that references the CompositionObject being targeted by the Expression. In this example, that would be \_visual, but will always refer to the object StartAnimation(…) is called on.

## Math shortcuts & basic operators

### Basic Operators

As mentioned earlier, you define an Expression by a single ExpressionNode or multiple – when defined by multiple, they are combined using operators. The basic supported operators are:

* Plus (+)
* Minus (-)
* Multiply (\*)
* Divide (/)
* Mod (%)

The classes are designed such that you will be able to use Intellisense to identify compile-time errors for invalid math operations. For example, the following ExpressionNode attempts to add a Vector3 reference from a Visual to Vector2 constant – note that this will throw a compile time error in Visual Studio:



### Math Shortcuts (Functions)

To build more complex equations, more advanced mathematical operations are needed. Some operations are tedious to perform manually, so helper functions (a subset of System.Numerics functions, e.g. Min, Max, etc.) are available in the ExpressionFunctions class. The following example creates an Expression that calculates the length of a Quaternion:



## Advanced Operations

### Comparison Operators

In addition to the basic mathematical operations (+, -, /, etc.), you can also create Expressions that use comparison operators:

* Greater than (>)
* Less than (<)
* Greater than or equal to (>=)
* Less than or equal to (<=)
* Equal to (==)
* Not Equal to (!=)

The following example demonstrates creating an Expression that outputs a Boolean Node showing whether the length of one Quaternion is equal to another:



### Conditional Operation

Finally, you can make some of the most powerful Expressions using a Conditional operation. This enables developers to define different behaviors for an Expression depending on a condition. This operation is defined by the ExpressionFunctions.Conditional method and contains three parts to mimic the standard ***condition ? true: false*** ternary operator:

* The Boolean Expression condition that is checked
* The Expression to be run if the condition is true
* The Expression to be run if the condition is false

The following example builds upon the above example. It compares the length of two quaternions, and based on the result uses one of two rotations in the form of a quaternion:



## Tips and Tricks for using Classes

Below are some tips and tricks that can be used for interacting with the classes

### Shortening Class Names

One of the challenges with this class model is an ExpressionNode can get very verbose and lengthy because the needing to “dot into” a class object to access the static method. If you run into this yourself, you can shorten the naming of the classes by defining a shortened version via the “using” syntax at the top of your file:



# Translating Old World to New

If you’re familiar with building Expressions in the old world by writing the equation as a string, the following sections outlines how the creation of an Expression compares between the old and new way.

## Creating an Expression

In the old world, you use the CreateExpressionAnimation() method off the Compositor. In the new one, you simply assign the variable an ExpressionNode (output from static methods of ExpressionValue, ExpressionFunctions or extension methods)



## Defining Constant Parameters

In the old world for both Constants and References, whether you intended them to be static or dynamic, you were required to define a string name and set the parameter value separately. In the new world, you only need to set the parameter if you want to template the Expression and later change what the parameter points to. Otherwise, you simply include the value directly in the equation.

In the example below, we plan to template this expression, varying the value of “extraOffset”. Shown is how to achieve this in the new and old way:



## Building Constants

In the old world, you could construct constant types within the string equation. In the new world, you use the static methods off the ExpressionFunction class.



## Defining Reference Parameters

In the old world, if you wanted to create a reference to a CompositionObject, it needed to have the parameter set for the string name in the equation. In the new world, you can either *get* the reference using the extension method off the CompositionObject, or create one using static methods off ExpressionValue.



## Using Math Functions & Math Operators

In the old world, you would include the function name inside the string equation. This presented problems with misspelling, knowing what parameters to provide and type safety on the output. In the new world, you get this through Intellisense as all the available Math functions are available off the ExpressionFunction class.

For operators, they were simply included in the string in the old world. In the new world, you can use them similar to the System.Numerics experience.

****

## Using Ternary and Conditional Operators

In the old world you would use the ***Condition ? ifTrue : ifFalse*** format for the ternary operation, using the appropriate conditional operators in the condition portion of the string. In the new world, the Ternary operator behavior is found off the Conditional function under the ExpressionFunctions class. All the same comparison operators are supported and can be used in the same format like the basic math operators.



## Keywords

In the old world, there were reserved string keywords that can be used to achieve specific behavior:

* This.StartingValue
* This.CurrentValue
* This.Target
* Pi
* True/False

The challenge with this model in the old world was they were not discoverable.

In the new world, the StartingValue/CurrentValue/Target keywords are made available off the ExpressionValue class. For the use of Pi and True/False, the values defined in C# are sufficient.



## Starting an Expression on a CompositionObject

In the old world, developers utilized the StartAnimation() method off of CompositionObject that passed in two values: the string name of the property animate and the ExpressionAnimation defined by a string. In the new world, there is an extension method that takes in an ExpressionNode instead an ExpressionAnimation.



# E2E Building Examples

This section is dedicated to walking through building a few different Expressions using the Expression Builder Classes. Each of the examples will start with an Expression (and any needed supporting code) and break down how these can be re-written using the new classes. All the examples will be pulled from samples on the [Windows UI Dev Labs Github Project](https://github.com/Microsoft/WindowsUIDevLabs).

There is an assumption that the reader has a general understanding of what Expressions are and how the ExpressionBuilder classes work. If not, it is recommended to read the [Intro](#_Intro) and [How to: Build Core Components of Expressions](#_How_to:_Build) first.

## Parallaxing Listing Items

([Github Link](https://github.com/Microsoft/WindowsUIDevLabs/tree/master/SampleGallery/Samples/SDK%2010586/ParallaxingListItems))

The first example we will walk through is the Parallaxing List Item sample found on the Windows UI Dev Labs Github Sample Gallery project. In this sample, we want to create a UI experience such that the background image for each list item parallax as the user scrolls through the list.

### Old Expression

Let’s first look at the relevant code for how the Expression is built today using strings:



### Summary of Expression definition

* The core of this Expression is uses a ScrollManipulationPropertySet, a CompositionPropertySet that contains information about the XAML ScrollViewer that manages the item in the XAML ListView.
  + Specifically, we are looking at the Translation.Y property. When building our Expression, we will need to grab a reference to this property.
* There are three other scalar parameters that comprise the remainder of this equation (StartOffset, ParallaxValue and ItemHeight). Note, that in this sample, the intent was to make this Expression a template, meaning that these values may need to be changed later.
  + If the intent was not to template, the Expression would have been created differently, with the values being written directly into the string.
* Finally, the equation itself has a common component (we’ll denote it “A”) that gives it the form A\*Parallax – A.
  + In this case “A” is:   
    "(ScrollManipulation.Translation.Y + StartOffset - (0.5 \* ItemHeight))”

### Building with ExpressionNodes

So let’s get started building this Expression into an ExpressionNode. To start, we’ll make three variables to keep track of the three Scalar Parameters and specifically for templating purposes:



Next, let’s get a reference to that ManipulationPropertySet (specifically, the Translation.Y property). To do that, we need to:

* Get a reference to the PropertySet
* Use the static method to get the Translation.Y property
  + A Scalar property

This can be done all in one line:



(For walkthrough purposes, this is stored as a separate variable and then put into the final equation. This also could have been included directly in the final ExpressionNode parallax below.)

Now let’s build out the “A” component of the A\*Parallax – A format that was described earlier:



Now we are ready to build out the full Expression pass it into the StartAnimation() function call!



### Final code snippet



## PropertySets

([Github Project](https://github.com/Microsoft/WindowsUIDevLabs/tree/master/SampleGallery/Samples/SDK%2010586/PropertySets))

The second example we will walk through is the PropertySets sample on the Windows UI Dev Labs Sample Gallery Github project. In this sample, we want to make a UI experience where we want to have a colored ball orbit another that is moving up and down.

### Old Expression

Let’s first look at the relevant code for how the Expression is built today using strings:



### Summary of Expression definition

* At a high level, this Expression is simply the sum of three components: A Visual reference, a CompositionPropertySet reference and a Vector3 object construction.
* A scalar property in a CompositionPropertySet named “Rotation” that is being animated by a separate KeyFrameAnimation dictates the core behavior of this Expression.
  + This property “Rotation”, and another property “CenterPointOffset, will need to be referenced in the equation.
* The Expression also constructs a Vector3 that takes the Cosine of the Radians-converted property “Rotation” in the CompositionPropertySet

### Building with ExpressionNodes

Note that, as mentioned earlier in the Tips and Tricks section of the doc, this walkthrough uses a shorthand to refer to the ExpressionFunctionClass as EF:



First, we get a reference to the PropertySet and its Rotation and CenterPointOffset properties:



Now we are ready to put together the full Expression:



### Final code snippet



## Curtain

([Github Project](https://github.com/Microsoft/WindowsUIDevLabs/tree/master/SampleGallery/Samples/SDK%2014393/Curtain))

The third example we will walk through is the Curtain sample on the Windows UI Dev Labs Sample Gallery Github project. Although there are a few instances where Expressions are used, we will focus on the Expression that defines the Spring motion of the curtain (the function named ActivateSpringForce()).

### Old Expression

Let’s look at the relevant code for how the Expression is built today with strings:



### Summary Expression Definition

* The equation for this Expression is leveraging the force equation used for damped harmonic oscillators: kx – cv, where k is the Spring Constant, x is the displacement of the spring, c is the damping constant and v is the velocity of the spring.
* The main component of this equation is an InteractionTracker and the associated properties of it to drive the damped harmonic oscillator equation.
  + In particular, the properties Position.Y and PositionVelocityInPixelsPerSecond.Y
* Because this Expression is getting properties from the same InteractionTracker it is animating, the Target keyword will be used here.

### Building with ExpressionNodes

We’ll start with defining out the first Expression, which is the Condition portion of the InertiaMotion Modifier. This is done by using the CompositionExtensions.SetCondition(…) extension method, which is accessed via InterationTrackerInertiaMotion.SetCondition(…).



Next, we’ll use the Target keyword to get a reference to the InteractionTracker object that this Expression will be applied to.



At this point, we are ready to build out the rest of the Expression and set the Motion component of the InertiaModifier, using another extension method CompositionExtensions.SetMotion(…):



### Final code snippet

