

Unity Composition Variables

PiRho Soft

Overview	1
Creating Variables	2
Defining Variables	5
Accessing Variables	7
The Global Store	10
The Scene Store	10
Accessing Built in Properties	10
Writing Expressions	12
Math	12
Logic	12
Conditionals	13
Type Testing	13
Assignment	13
Commands	13
Constants	15
Multiple Statements	16
List Management	16
Writing Custom Commands	17
Exposing Variables	18
Variable References	18
Custom Variable Stores	19
IVariableStore	19
Mapped Variables	20
Class Maps	21

Overview

The variables system provides dynamic creation and storage of Variable Values that can be manipulated and accessed at runtime. They are stored in Variable Stores and looked up, set, and modified using Variable References and Expressions. This is the core of the composition framework and enables editor based creation of logic using [Instruction Graphs](#), and automatic user interface updates using [Variable Bindings](#). This document will cover how to create objects that contain variables, how to access those variables, and how to extend the variables system with custom objects and commands.

The sections in this topic are:

Creating Variables

Creating objects that hold Variable Values

Defining Variables

Creating Variable Schemas to define a reusable set of Variable Definitions

Accessing Variables

Using Variable References to access variables

Writing Expressions

Writing Expressions to perform computations and assign values

Writing Custom Commands

Writing Commands that can be used by Expressions

Exposing Variables

Declaring and using Variable References and Expressions in user created scripts

Custom Variable Stores

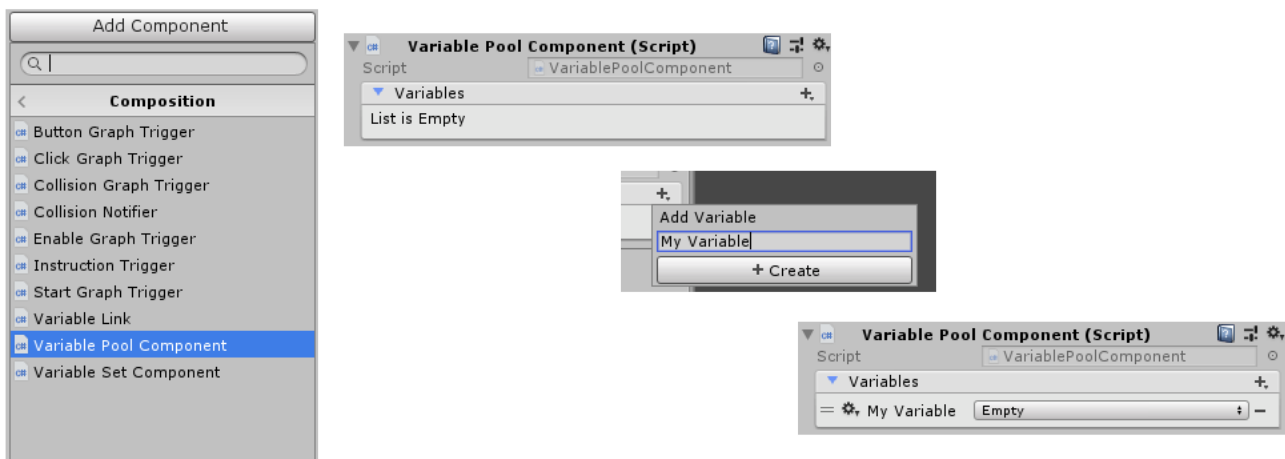
Writing Objects in code that can be used as Variable Stores


Creating Variables

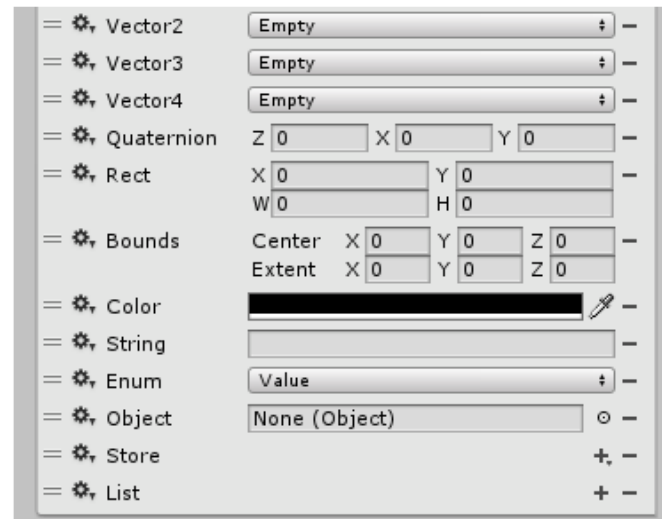
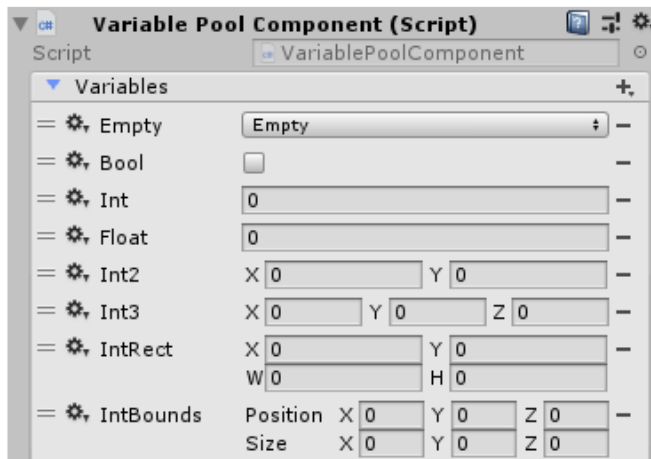
All variables exist in a Variable Store, often referred to as just *store*. Four Unity objects are included that can be used as stores:

	Type	Purpose
Variable Pool Component	Mono Behaviour	Variables are defined and edited directly in the inspector
Variable Pool Asset	Scriptable Object	Variables are defined and edited directly in the inspector
Variable Set Component	Mono Behaviour	Variables are edited in the inspector and defined by a Variable Schema
Variable Set Asset	Scriptable Object	Variables are edited in the inspector and defined by a Variable Schema





The simplest of these is the Variable Pool Component.



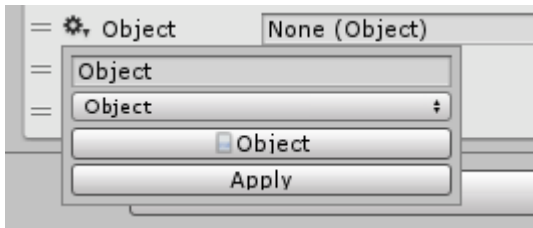
This is created like any other [Component](#) and can be found in the *Add Component* menu under **PiRho Soft › Composition › Variable Pool Component**. Variables are added to the component with the  button in the upper right of the *Variables* list. Clicking the button will open a popup where the name of the variable is entered. After clicking *Create*, the variable will be added to the store where it can be edited. Before setting a value for the variable, the type must be specified by selecting a type from the drop down list. The available types are:



	Description
Empty	No value has been set
Bool	The value can be either true or false
Int	The value is an integer number
Float	The value is a decimal number
Int2	The value is a Vector2Int
Int3	The value is a Vector3Int
IntRect	The value is a RectInt
IntBounds	The value is a BoundsInt
Vector2	The value is a Vector2
Vector3	The value is a Vector3
Vector4	The value is a Vector4
Quaternion	The value is a Quaternion
Rect	The value is a Rect
Bounds	The value is a Bounds
Color	The value is a Color
String	The value is a text string
Enum	The value is an enum of a type defined in code (Enum values must be constrained)
Object	The value is an Object
Store	The value is a Variable Store (this cannot be set in the inspector however Variable Stores can be assigned as Objects)
List	The value is a list of other values

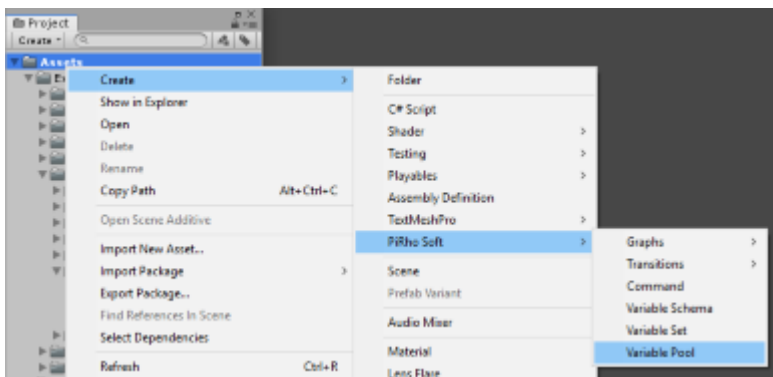
Variables can be reordered by clicking and dragging the  to the left of each, removed by clicking the  to the right, or have their type changed using the  menu. Also in the  menu, some types can have a constraint specified to restrict the value the variable can hold. The types that can be

constrained are:



	Description
Int	The number must be between a minimum and maximum value
Float	The number must be between a minimum and maximum value
String	The value is selected from a list
Enum	The value is from a specific enum type defined in code (Enum values must be constrained)
Object	The object must be a specific object type
Store	The store holds values defined in a Variable Schema (described in the next section)
List	The list holds values of a specific type

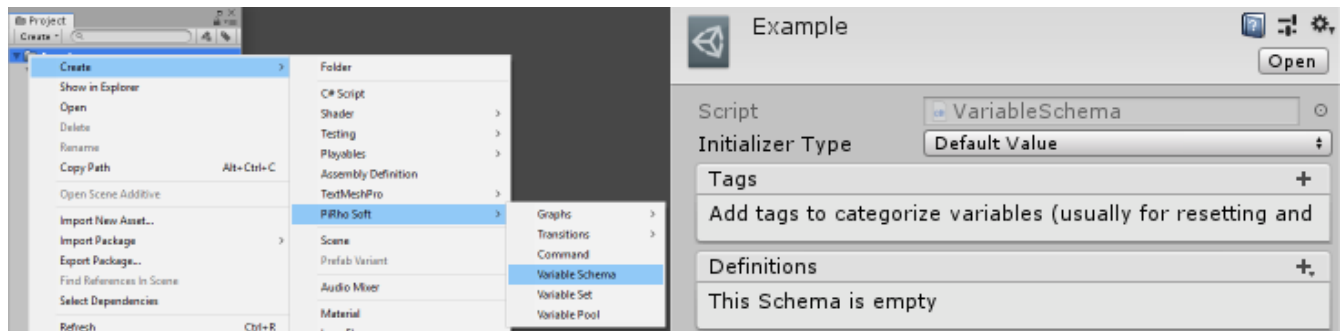
A Variable Pool Asset is edited and used the same as a Variable Pool Component, however it is created as an asset using the [project window](#) rather than being added to a Game Object as a Component. It is located in the *Create* menu under **PiRho Soft > Variable Pool**



If a set of variables is reused across many objects Variable Set Component should be used instead of Variable Pool Component and Variable Set Asset should be used instead of Variable Pool Asset. These objects use a Variable Schema to define the variables they contain thus eliminating the need to define them individually on each object or update every object to add or remove a variable. These are explained in the [next section](#).

Defining Variables

Variables are defined using a Variable Schema, often referred to as just *schema*. A schema is created in the [project window](#) using the *Create* menu and found under **PiRho Soft > Variable Schema**.

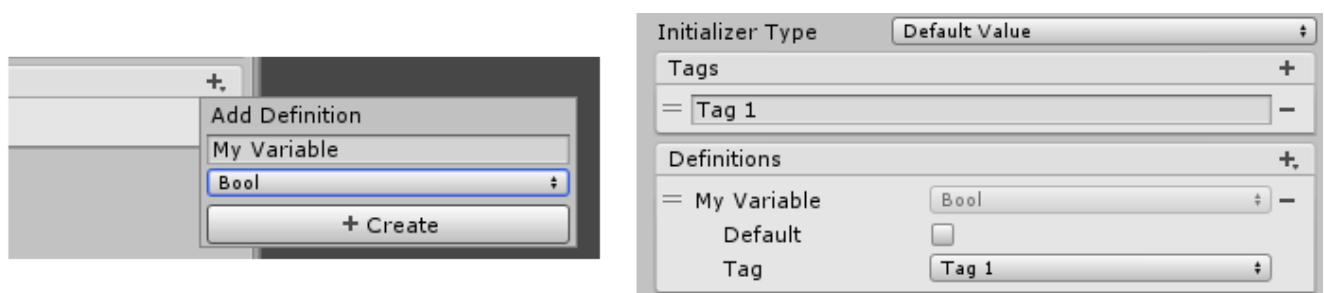


A Variable Schema holds a set of Variable Definitions which describe the type and constraint for values in stores that use the schema. In addition, an *Initializer Type* can be specified for each definition that will set the value of a variable when it is first created or reset. The possible settings for *Initializer Type* are:

	Description
None	The variable is set to the default value for its type
Default Value	The variable is set to an exact value
Expression	The variable is set to the result of an Expression (described in Writing Expressions)

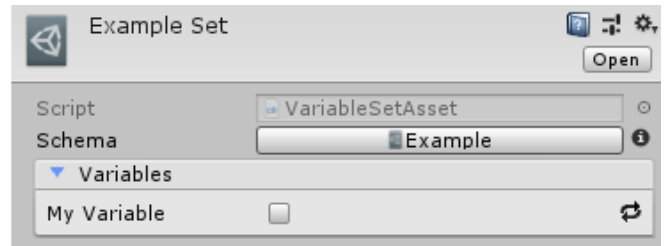
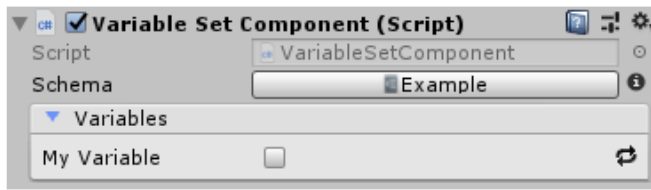
The schema can also define a list of *Tags*. Each definition in the schema can then select one of these tags as a grouping mechanism to be used for [resetting the values](#).

The definitions themselves are added to the *Definitions* list.



The + button will open a popup where the name, type, and constraint of the definition is specified. Once a definition is added, the *Initializer* or *Default* (depending on the schema's *Initializer Type* setting) can be specified as well as the *Tag* (if the schema has defined any in *Tags*). The definitions can be reordered using the ≡ handle or removed using the - button.

Once a schema has been created, it can be used as a constraint for variables defined with type Store, or with Variable Set Component and Variable Set Asset. These are similar to the corresponding [Variable Pool Component](#) and [Variable Pool Asset](#) except instead of adding the variables directly to the object, they are automatically added by selecting a schema.



Editing a Variable Set is very straight forward. Select the *Schema* to use, which will then populate the *Variables* list. The value of each variable can then be edited depending on the type and constraint defined by the schema. If the schema is edited, the *Variables* list will be updated to reflect any changes, preserving any values that have already been set. If the selected *Schema* changes, the *Variables* list will be repopulated.

Accessing Variables

Once a set of variables has been defined and created in stores, they can now be accessed and used by other objects. Objects that use variables to control their functionality will have Variable References and Expressions exposed as a way to specify the variable they should access. Variable References, which refer to a variable directly, are covered here. Expressions, which can use multiple variables to compute a value, are covered in the [next section](#).

Most commonly Variable References are encountered as properties on [Graph Nodes](#). The documentation for the specific node will indicate the type of value it expects. [Graph Nodes](#) resolve Variable References using an Instruction Store. The Instruction Store has the following Variable Stores exposed and available to Variable References:

Name	Description
input	predefined variables that are sent to the graph when it is run (detailed in Context Inputs and Outputs)
output	predefined variables that are returned from the graph (detailed in Context Inputs and Outputs)
local	variables available only to the current graph that are not predefined or constrained in any way
global	variables available and shared everywhere that are not predefined or constrained in any way (detailed below)
scene	a special store that provides access to scene objects by name (detailed below)



Variable References used outside of [Graph Nodes](#) may have a different set of stores exposed. The corresponding documentation will indicate what those are, but the global and scene stores will always be available.

Variable References are specified as a string and have a simple syntax. Basic references that refer to a variable in a store or specified with the store name, followed by a '.', followed by the variable name. For example to reference a variable named attack on the local store the reference would be local.attack. Because variables can also be stores themselves, this syntax can be chained. For example, to access the attack variable on a player in the global store, the reference would be global.player.attack. To access the variables inside a variable with type List, brackets are used along with the index of the variable to access. For example if the global player has a list of items. The second item would be referenced with global.player.items[1] (note the indexes are 0 based so the second item is at index 1). A variable in a List could itself be a variable store, in which case the same chaining can be used. So if an item has a name, that name would be referenced with global.player.items[1].name. This chaining works with any combination of List and Store variables to any depth.

The final capability of Variable References is casting. If a Variable Reference is supposed to refer to a certain component type, but the available stores only have access to a sibling of that component,

the component can be accessed with `as`. For example if the items of the player in the previous example are stored on a different component of type `Inventory` that is a sibling of the player component, the items would be referenced with `global.player as Inventory.items`. Note that `Inventory` in this case is the exact name of the Component type that is being looked up.



Often Variable References to objects do not need to be cast as the code using the reference will perform the cast automatically. This is explained in more detail in the [Graph Inputs and Outputs](#) section.

Several of the built in variable types can have properties of their value accessed using the same syntax as accessing a value on a store. Specifically, the following types have the listed values available:

Type	Property	
Int2	x	The value of x on the Vector2Int
	y	The value of y on the Vector2Int
Int3	x	The value of x on the Vector3Int
	y	The value of y on the Vector3Int
	z	The value of z on the Vector3Int
IntRect	'x'	The value of x on the RectInt .
	'y'	The value of y on the RectInt .
	'w'	The value of width on the RectInt .
	'h'	The value of height on the RectInt
IntBounds	'x'	The value of x on the BoundsInt .
	'y'	The value of y on the BoundsInt .
	'z'	The value of z on the BoundsInt .
	'w'	The value of size.x on the BoundsInt .
	'h'	The value of size.y on the BoundsInt .
	'd'	The value of size.z on the BoundsInt
Vector2	x	The value of x on the Vector2
	y	The value of y on the Vector2
Vector3	x	The value of x on the Vector3
	y	The value of y on the Vector3
	z	The value of z on the Vector3

Type	Property	
Vector4	x	The value of x on the Vector4
	y	The value of y on the Vector4
	z	The value of z on the Vector4
	w	The value of w on the Vector4
Quaternion	x	The value of x on the Quaternion
	y	The value of y on the Quaternion
	z	The value of z on the Quaternion
	w	The value of w on the Quaternion
Rect	'x'	The value of x on the Rect .
	'y'	The value of y on the Rect .
	'w'	The value of width on the Rect .
	'h'	The value of height on the Rect
Bounds	'x'	The value of center.x on the Bounds .
	'y'	The value of center.y on the Bounds .
	'z'	The value of center.z on the Bounds .
	'w'	The value of size.x on the Bounds .
	'h'	The value of size.y on the Bounds .
	'd'	The value of size.z on the Bounds
Color	r	The value of r on the Color
	g	The value of g on the Color
	b	The value of b on the Color
	a	The value of a on the Color

The following table contains a complete breakdown of the Variable Reference syntax:

	Symbol	Description	Example
Store Access	.	Looks up a variable in a store	local.attack
Property Access	.	Looks up a property on a value	local.position.x

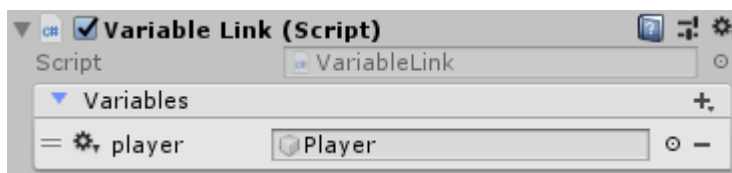
	Symbol	Description	Example
List Access	[and]	Looks up a variable in a list	<code>global.player.items[1]</code>
Casting	as	Looks up a sibling object	<code>global.player as Inventory</code>



If a variable reference is entered with incorrect syntax, the text box will be colored red indicating there is an error. This will not check if the referenced variable actually exists or is the correct type as that can only be known at runtime. These runtime errors will be indicated by printing an error to the [console window](#) and can be tracked down using the built in [debugging features](#).

The Global Store

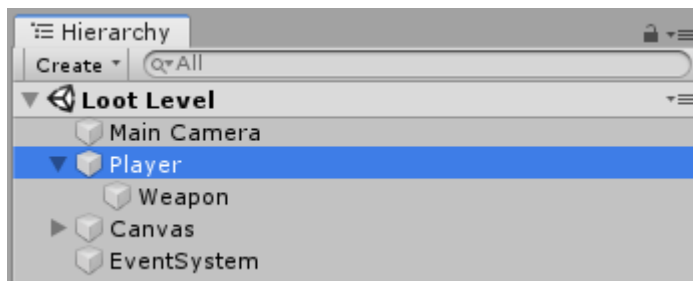
Variables in the global store are available to every Variable Reference and Expression and can be added and changed at any time. To add variables during editing, use the Variable Link Component. This component can be added to any object and any number of them can be used. When a Variable Link Component is loaded and enabled, the variables it defines will be added to the global store. When it is unloaded or disabled, the variables will be removed.



Variables are defined and added in the same way as for [Variable Pool Components](#).

The Scene Store

The scene store provides access to all loaded objects by name. The contained objects are always a reflection of the currently loaded scenes and do not need to be added or removed manually. To access an object in a scene that has been assigned the name Player, it can be referenced using `scene.Player`.



Accessing Built in Properties

The properties of [Objects](#) that are not variable stores can be accessed if the [Object's](#) class has a ClassMap defined. ClassMaps for [Transform](#) and [Camera](#) are built in and custom class maps can be defined as described [here](#).

The exposed properties can be accessed just like any variable in a store. For example, the position of a [GameObject](#) named `Player` could be retrieved with the [VariableReference](#) `scene.Player` as `Transform.position`.

Writing Expressions

Math

Expressions have a wide application of uses but at their simplest are an extension of Variable References allowing math operations to be performed on variables. For example if a [Graph Node](#) is expecting an attack value and a player exists with an attack stat, the damage from a critical hit could be given in the form of the expression `global.player.attack * 2`. The full suite of math operations is available:

Operator	Description
+	Adds two values
-	Subtracts a value from another
*	Multiplies two values
/	Divides a value by another
%	Computes the remainder of a division
^	Raises a value to an exponent
-	Negates a value

These math operations are computed in the expected order based on the standard order of operations (exponents first, then multiplication and division, and last addition and subtraction). Additionally, operations can be grouped with parentheses to force a certain execution order.

Logic

Some expressions, for example *Condition* on a [Conditional Node](#), expect the result to have type Bool. For this, a set of comparison and logic operators is available:

Operator	Description
==	Determines if two values are equal
!=	Determines if two values are not equal
<	Determines if a value is less than another value
>	Determines if a value is greater than another value
≤	Determines if a value is less than or equal to another value
≥	Determines if a value is greater than or equal to another value
&&	Determines if two values are both true
,	
,	Determines if either or both values true
!	Inverts a value (i.e true becomes false and false becomes true)



Logic operations are executed before comparison operations, which are executed before math operations.



`&&` and `||` perform short circuiting, meaning if the result can be determined from the left hand side, the right hand side will not be executed. This can be useful in conjunction with the Type Testing feature listed below.

Conditionals

To perform simple conditional checks inside an expression, use the ternary syntax of `condition ? statement if true : statement if false`. For example, to compute a damage value based on whether or not the attack is a critical hit, the expression could be `local.isCritical ? global.player.attack * 2 : global.player.attack` or alternatively and equally as valid `global.player.attack * (local.isCritical ? 2 : 1)`.

Type Testing

The type of a variable can be tested using the `is` statement. For example, to test if the variable `local.isCritical` is a `Bool` use `local.isCritical is Bool`. A variable can also be checked for existence using the `Empty` type. The full list of types can be seen [here](#).

Assignment

In addition to accessing variables, expressions can set variables. This is done with the `=` operator. For example to store a computed damage value instead of returning it like the above example, write the expression `local.damage = global.player.attack * 2`. The damage value can be then later accessed by other Expressions and Variable References as `local.damage`. Each of the math and logic operations also exist as an assignment to more simply modify a value. These are: `+=`, `-=`, `*=`, `/=`, `^=`, `&=`, and `|=`. For example, the player's `attack`, can be permanently buffed with the statement `global.player.attack *= 2`. Shortcuts for adding and subtracting one from a variable are available with the ``` and ``--`` operators. Place the operator before a value to include the increment or decrement in the result. For example, if `local.number`` has the value `4``, the expression `local.result = local.number`` will assign 5 to both `local.result` and `local.number` whereas the expression `local.result = local.number++` will assign 4 to `local.result` and 5 to `local.number`.

Commands

Expressions can also use commands for getting the results of a more complex operation. The syntax for calling a command consists of the command name, followed by `'('` followed by any number of values (called parameters) to send to the command, followed by `)'`. A parameter can be a literal value, a variable reference, or an entire statement. The built in commands are:

Name	Description	Example
Abs	Returns the absolute value of a number	<code>local.speed = Abs(local.velocity)</code>

Name	Description	Example
Acos	Computes the angle whose cosine is the input	local.direction = Acos(local.x)
Asin	Computes the angle whose sine is the input	local.direction = Asin(local.y)
Atan	Computes the angle whose tangent is the input	local.direction = Atan(local.y / local.x)
	Computes the angle with correct sign whose tangent is the first parameter divided by the second	local.direction = Atan(local.y, local.x)
Ceiling	Rounds the input value up	local.rounded = Ceiling(local.number)
Clamp	Clamps the input between two values	local.clamped = Clamp(local.number, 5, 10)
Cos	Computes the cosine of an angle (in radians)	local.x = Cos(local.direction)
Floor	Rounds the input value down	local.rounded = Floor(local.number)
Lerp	Mathf.Lerp	local.interpolated = Lerp(5, 10, 0.3)
Log	Computes the base 10 logarithm of the input	local.log = Log(5)
	Computes the logarithm of the input using the given base	local.base2log = Log(8, 2)
Max	Returns the biggest of a list of values	local.biggest = Max(5, 8, -12, 2.5)
Min	Returns the Smallest of a list of values	local.biggest = Min(5, 8, -12, 2.5)
Pow	Computes the exponent of a value	local.cubed = Pow(5, 3)
Random	Returns a random number between 0 and 1	local.random = Random()
	Returns a random number between 0 and the input	local.random = Random(100)
	Returns a random number between the two inputs	local.random = Random(-100, 100)
	If the input is a list, returns a random item in the list	local.item = Random(local.list)
Rounds	Rounds the input to the closest integer	local.rounded = Round(local.number)
Sign	Returns -1 when the input is negative, 0 when the input is 0, and 1 when the input is positive	local.sign = Sign(local.number)

Name	Description	Example
Sin	Computes the sine of an angle (in radians)	<code>local.y = Sin(local.direction)</code>
Sqrt	Computes the square root an input	<code>local.root = Sqrt(local.number)</code>
Tan	Computes the tangent of an angle (in radians)	<code>local.angle = Tan(local.number)</code>
Truncate	Rounds the input toward 0	<code>local.truncated = Truncate(local.number)</code>
Time		Realtime



Custom commands can be defined as described in the [Creating Custom Commands](#) section.

Commands exist for creating values of each of the types that cannot be specified with a literal. These are:

Name	Parameters
Vector2	(x, y)
Vector2Int	(x, y)
Vector3	(x, y, z)
Vector3Int	(x, y, z)
Vector4	(x, y, z, w)
Quaternion	(rotation about x, rotation about y, rotation about z)
Rect	(x, y, width, height) or (position, size)
RectInt	(x, y, width, height) or (position, size)
Bounds	(position, size)
BoundsInt	(x, y, z, width, height, depth) or (position, size)
Color	(r, g, b) or (r, g, b, a)
List	() or (count)
Store	() or (schema name)



When using a schema for creating a Store, the schema must be in a Resources folder in the project.

Constants

Several constant values are available as well and can be accessed directly by name:

Name	Description
PI	A variable of type Float containing the value of pi (3.14...)
Deg2Rad	A variable of type Float containing the value for converting an angle in degrees to radians ($\text{PI} / 180$)
Rad2Deg	A variable of type Float containing the value for converting an angle in radians to degrees ($180 / \text{PI}$)

Multiple Statements

Expressions can consist of multiple statements with the final statement computing the expression's result. For example, a more complex damage calculation might look like this:

```
local.isCritical = Random() > 0.5
local.strength = global.player.strength / global.target.defense
local.damage = global.player.weapon.attack * strength * (local.isCritical ? 2 : 1)
```



If an expression is entered with incorrect syntax, the text box will be colored red indicating there is an error. This will not check if the expression actually executes correctly or returns a variable with the correct type. These runtime errors will be indicated by printing an error to the [console window](#) and can be tracked down using the built in [debugging features](#).

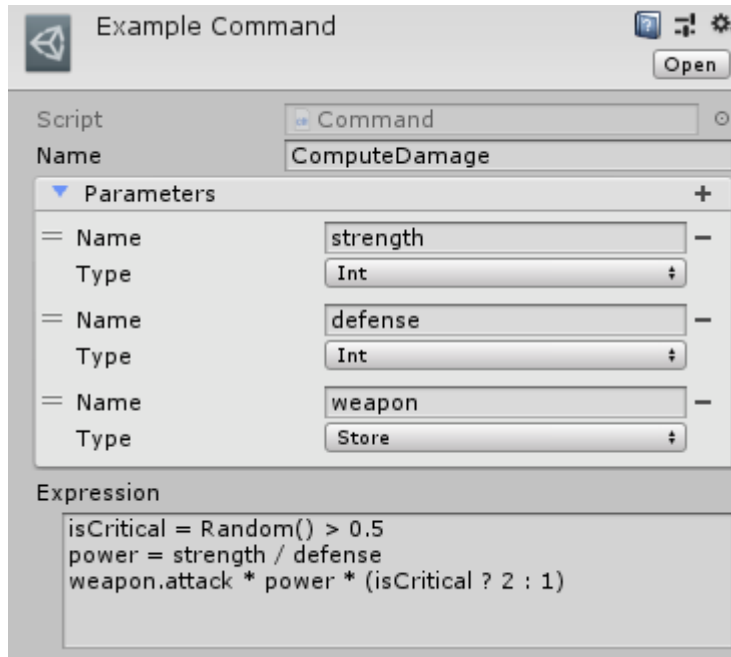
List Management

Several operators have special handling for List variables. These are:

Operator	Description
<code>+=</code>	Adds the item to the right if the operator to the end of the list.
<code>-=</code>	Removes the item to the right of the operator from the list.
<code>++</code>	Adds an empty item to the end of the list
<code>--</code>	Removes the item at the end of the list

Writing Custom Commands

Calculations can be defined in a Command as a way to share expressions among different objects or keep complex calculations isolated from the places they are used. This can improve organization and make it easier to manage different pieces of game logic. Commands are created in the [project window](#) using the *Create* menu and found under **PiRho Soft > Command**. A Command should always be placed in a *Resources* folder so it can be [loaded at runtime](#) by the [CompositionManager](#) (this will happen automatically).



A command consists of a *Name*, a list of input *Parameters*, and an *Expression*. *Name* is the text to use in an expression to call the command. *Parameters* contains the definitions for the input values the expression will use to perform the computation. *Expression* defines the command with the final statement computing the result. Each of the parameters are available to the expression directly and variables can be created without reference any store. For example, the damage calculation from the [previous section](#) could be written as a command named `ComputeDamage` with parameters `strength`, `defense`, and `weapon`:

```
isCritical = Random() > 0.5
power = strength / defense
weapon.attack * power * (isCritical ? 2 : 1)
```

This command would then be called with `local.damage = ComputeDamage(global.player.strength, global.target.defense, global.player.weapon)`

Exposing Variables

Variable References and Expressions can be used when writing [graph nodes](#), [components](#), or [assets](#) to interface custom code with the variables system.

Variable References

The [VariableReference](#) class provides the functionality for both looking up variables and assigning values to variables. To use a [VariableReference](#), simply add one as a public field to your object. Use the *GetValue* method to look up a [VariableValue](#) and the *SetValue* method to assign. Both of these methods take an [IVariableStore](#) as a parameter. While this can be any object that implements the [IVariableStore](#) interface, in most cases the [CompositionManager.DefaultStore](#) should be used which gives access to the [global and scene stores](#). When implementing an [InstructionGraphNode](#) or [VariableBinding](#) the calling code will be given an [IVariableStore](#) which can then be passed to *SetValue* or *GetValue*.

Custom Variable Stores

When writing [Components](#) or [ScriptableObject](#)s it is often useful to expose the object and its properties to the variables system. This can be done by implementing the [IVariableStore](#) interface. This interface consists of three easy to implement methods that provide all the necessary functionality for fully exposing the object and its properties.

IVariableStore

As an example, consider the LootWeapon [Variable Schema](#) and various weapons that use it in the Loot example. The same setup could be accomplished in code by defining a LootWeapon [ScriptableObject](#) that implements [IVariableStore](#).

```
using System.Collections.Generic;
using PiRhoSoft.CompositionEngine;
using UnityEngine;

namespace PiRhoSoft.CompositionExample
{
    [AddComponentMenu("PiRho Soft/Examples/Loot Weapon")]
    public class LootWeapon : ScriptableObject, IVariableStore
    {
        public string Name;
        public float MinimumSpeed;
        public float MaximumSpeed;
        public int MinimumStrength;
        public int MaximumStrength;

        private static List<string> _names = new List<string>
        {
            nameof(Name),
            nameof(MinimumSpeed),
            nameof(MaximumSpeed),
            nameof(MinimumStrength),
            nameof(MaximumStrength)
        };

        public IList<string> GetVariableNames()
        {
            return _names;
        }

        public VariableValue GetVariable(string name)
        {
            switch (name)
            {
                case nameof(Name): return VariableValue.Create(Name);
                case nameof(MinimumSpeed): return VariableValue.Create(MinimumSpeed);
                case nameof(MaximumSpeed): return VariableValue.Create(MaximumSpeed);
            }
        }
    }
}
```

```

        case nameof(MinimumStrength): return
VariableValue.Create(MinimumStrength);
        case nameof(MaximumStrength): return
VariableValue.Create(MaximumStrength);
        default: return VariableValue.Empty;
    }
}

public SetVariableResult SetVariable(string name, VariableValue value)
{
    switch (name)
    {
        case nameof(Name): return value.TryGetString(out Name) ?
SetVariableResult.Success : SetVariableResult.TypeMismatch;
        case nameof(MinimumSpeed): return value.TryGetFloat(out MinimumSpeed)
? SetVariableResult.Success : SetVariableResult.TypeMismatch;
        case nameof(MaximumSpeed): return value.TryGetFloat(out MaximumSpeed)
? SetVariableResult.Success : SetVariableResult.TypeMismatch;
        case nameof(MinimumStrength): return value.TryGetInt(out
MinimumStrength) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
        case nameof(MaximumStrength): return value.TryGetInt(out
MaximumStrength) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
        default: return SetVariableResult.NotFound;
    }
}
}
}

```

Each of the weapon assets would then be created as instances of `LootWeapon` instead of [Variable Set Asset](#).

Mapped Variables

For more complex classes with many properties, or classes that want a mix of code defined and editor defined properties, it is much simpler and more flexible to use the [mapped-variable-store, MappedVariableStore](#) class. This can be used directly or by deriving from the [VariableSetComponent](#) (for [MonoBehaviours](#)) or [VariableSetAsset](#) (for [ScriptableObjects](#)). These classes can also be used directly without subclassing as described in [Defining Variables](#) but become more powerful when extended. Deriving automatically adds [VariableSchema](#) support, [IVariableStore](#) access for all schema and code defined properties, variable resetting with [IVariableReset](#), and full editor and [watch window](#) integration.

To expose code defined fields and properties to the variables system, the [MappedVariableAttribute](#) is used. This is as simple as adding the attribute to a field or property on a class derived from [VariableSetComponent](#) or [VariableSetAsset](#). Following is an example of the same `LootWeapon` class defined as a [VariableSetAsset](#) instead of an [IVariableStore](#). This has the same functionality as the above example but can also be extended in the editor with a [VariableSchema](#).

```

using PiRhoSoft.CompositionEngine;
using UnityEngine;

namespace PiRhoSoft.CompositionExample
{
    [AddComponentMenu("PiRho Soft/Examples/Loot Weapon")]
    public class LootWeapon : VariableSetAsset
    {
        [MappedVariable] public string Name;
        [MappedVariable] public float MinimumSpeed;
        [MappedVariable] public float MaximumSpeed;
        [MappedVariable] public int MinimumStrength;
        [MappedVariable] public int MaximumStrength;
    }
}

```

The [MappedVariableAttribute](#) can optionally be passed a parameter indicating whether the variable is allowed to be set by the variables system. The attribute can also be added to properties. A property without a setter will automatically be read only.

All [VariableTypes](#) are supported with [MappedVariableAttribute](#) including [IList<T>](#) implementors when *T* is itself a valid [VariableType](#).

Class Maps

In situations where an [Object](#) class cannot be changed to implement [IVariableStore](#), like for a third party or built in [Component](#), the [ClassMap](#) class is provided. The following is an example of the [LootWeapon](#) class exposed to the variables system using a [ClassMap](#) instead of with [IVariableStore](#) or [VariableSetAsset](#).

```

using PiRhoSoft.CompositionEngine;
using System.Collections.Generic;
using UnityEngine;

namespace PiRhoSoft.CompositionExample
{
    [AddComponentMenu("PiRho Soft/Examples/Loot Weapon")]
    public class LootWeapon : ScriptableObject
    {
        public string Name;
        public float MinimumSpeed;
        public float MaximumSpeed;
        public int MinimumStrength;
        public int MaximumStrength;
    }

    [AddComponentMenu("PiRho Soft/Examples/Loot Weapon")]
    public class LootWeaponMap : ClassMap<LootWeapon>

```

```

{
    private static List<string> _names = new List<string>
    {
        nameof(LootWeapon.Name),
        nameof(LootWeapon.MinimumSpeed),
        nameof(LootWeapon.MaximumSpeed),
        nameof(LootWeapon.MinimumStrength),
        nameof(LootWeapon.MaximumStrength)
    };

    public override IList<string> GetVariableNames()
    {
        return _names;
    }

    public override VariableValue GetVariable(LootWeapon weapon, string name)
    {
        switch (name)
        {
            case nameof(LootWeapon.Name): return
VariableValue.Create(weapon.Name);
            case nameof(LootWeapon.MinimumSpeed): return
VariableValue.Create(weapon.MinimumSpeed);
            case nameof(LootWeapon.MaximumSpeed): return
VariableValue.Create(weapon.MaximumSpeed);
            case nameof(LootWeapon.MinimumStrength): return
VariableValue.Create(weapon.MinimumStrength);
            case nameof(LootWeapon.MaximumStrength): return
VariableValue.Create(weapon.MaximumStrength);
            default: return VariableValue.Empty;
        }
    }

    public override SetVariableResult SetVariable(LootWeapon weapon, string name,
VariableValue value)
    {
        switch (name)
        {
            case nameof(LootWeapon.Name): return value.TryGetString(out
weapon.Name) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
            case nameof(LootWeapon.MinimumSpeed): return value.TryGetFloat(out
weapon.MinimumSpeed) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
            case nameof(LootWeapon.MaximumSpeed): return value.TryGetFloat(out
weapon.MaximumSpeed) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
            case nameof(LootWeapon.MinimumStrength): return value.TryGetInt(out
weapon.MinimumStrength) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
            case nameof(LootWeapon.MaximumStrength): return value.TryGetInt(out
weapon.MaximumStrength) ? SetVariableResult.Success : SetVariableResult.TypeMismatch;
            default: return SetVariableResult.NotFound;
        }
    }
}

```



```
}  
}
```