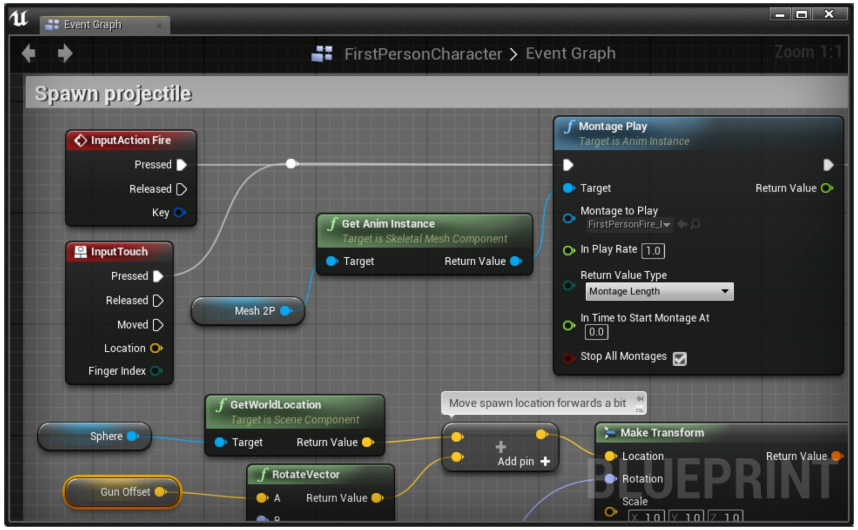

Unreal Engine 5
Blueprint Basic

WHAT IS BLUEPRINT?

Blueprint is a **visual scripting language** created by Epic Games for Unreal Engine. It is used within the Unreal Editor **to create new classes and gameplay functionality.**

A script in Blueprint is **represented by graphs of nodes connected by wires** that define the flow of execution.

The word “Blueprint” is also used to refer to a game object created using Blueprint.



MAJOR BLUEPRINT TYPES

Level Blueprint

A **Level Blueprint** is a special type of Blueprint that **belongs to a Level**. It is used to **define specific events and actions in a Level.**

A Level Blueprint can be used to interact with Blueprint Actor classes and to manage some systems, such as cinematics and Level streaming.

Blueprint Class

A **class** is the definition of data and behavior that will be used by a particular type of object. **A Blueprint class can be based on a C++ class or on another Blueprint class.**

A Blueprint class is used to create interactive objects for the game and can be **reused in any Level.**

OBJECT, ACTOR, AND ACTOR COMPONENT CLASSES

When creating a new Blueprint class, you must define the **parent class**. All variables and actions of the parent class will be part of the new class, which is known as a **child class** or **subclass**. This concept is called **inheritance**.

Below are some parent classes:

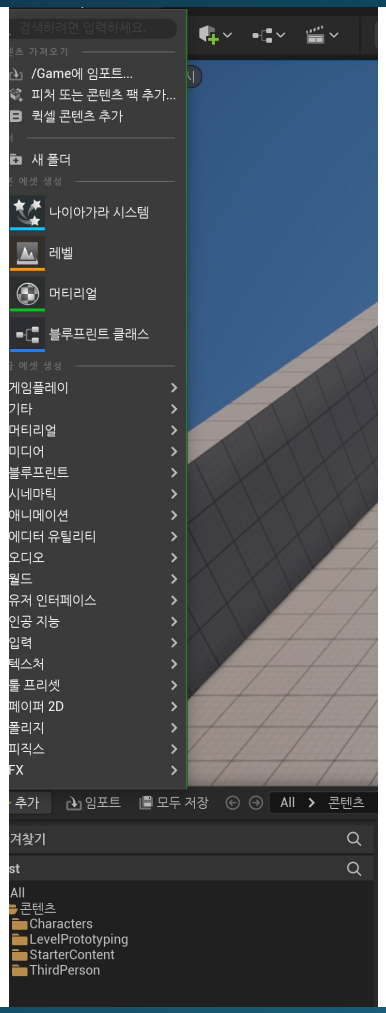
- **Object**: The base class for objects in Unreal Engine. All other classes are subclasses of the Object class.
- **Actor**: The base class used for objects that can be placed or spawned into a Level.
- **Actor Component**: The base class for components that define **reusable behavior that can be added to Actors.**

Therefore, **every Actor is an Object, but not all Objects are Actors**. For example, an Actor Component is an Object, but it is not an Actor.

INSTANCES OF A CLASS

"**Instance**" is a term used to reference an object of a class.

An example can be seen in the image on the right. Assuming there is a class called "**Blueprint_Chair**" that represents a chair, the image shows that four **instances** of the **Blueprint_Chair** class have been added to the Level.



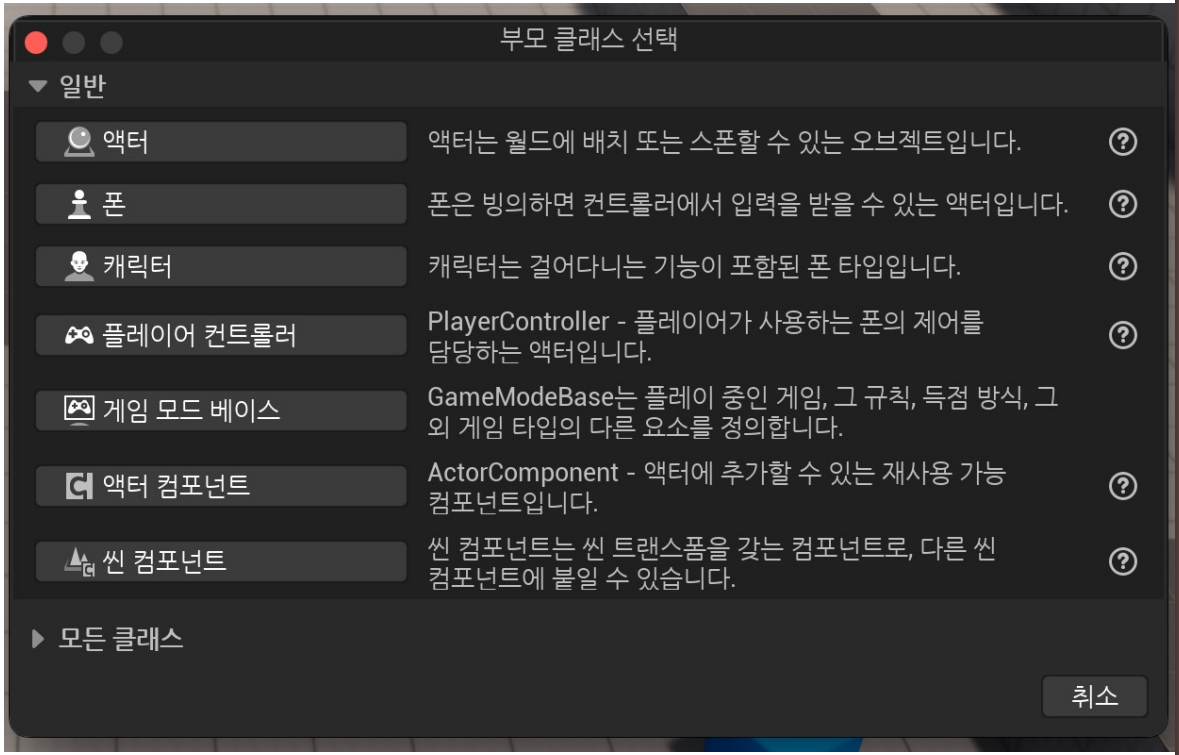
CREATING BLUEPRINT CLASSES: THE CONTENT BROWSER

To create a new Blueprint class in the **Content Browser**, click the green **Add New** button and select “**Blueprint Class**”.

CREATING BLUEPRINT CLASSES: PICK PARENT CLASS WINDOW

The next step is to choose the parent class of the new Blueprint.

The image on the right shows the most commonly used classes listed in the **Pick Parent Class** window. **Other parent classes can be found by expanding the All Classes option.**



CREATING BLUEPRINT CLASSES: ACTORS IN THE LEVEL

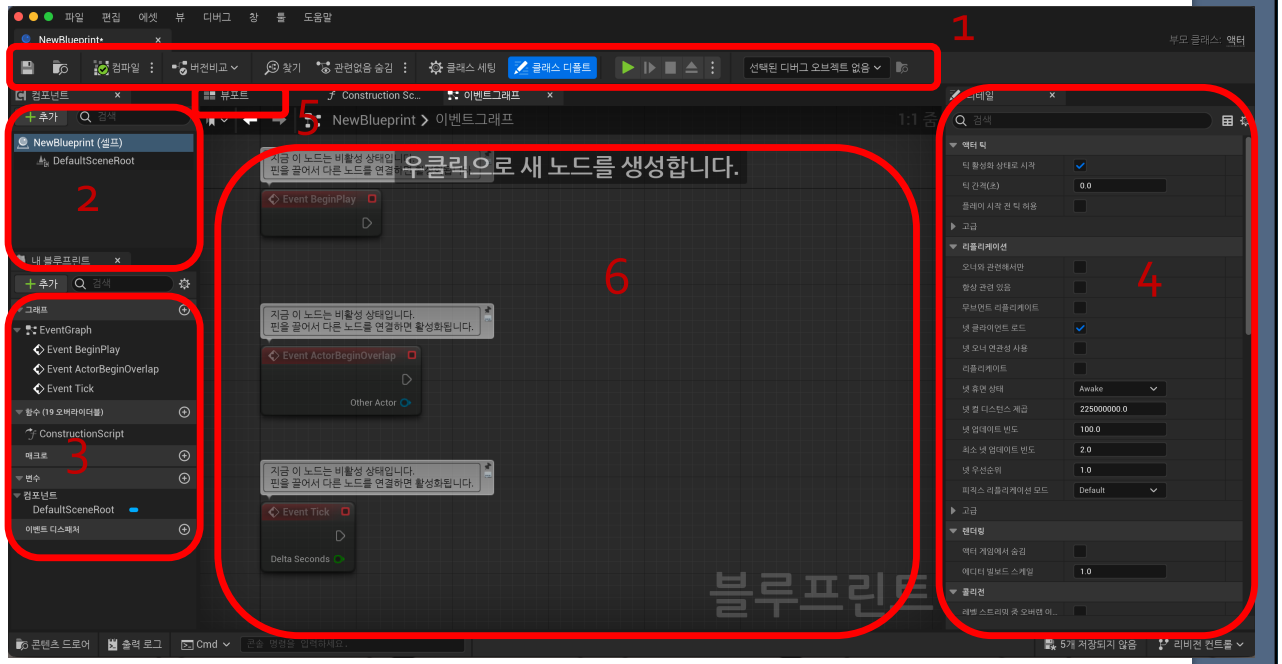
Blueprint editor interface

BLUEPRINT EDITOR: OVERVIEW

To open the **Blueprint Editor**, double-click on a Blueprint or right-click and choose “**Edit...**”.

The main parts of the Blueprint Editor, highlighted in the image to the right, are as follows:

1. Toolbar
2. Components panel
3. My Blueprint panel
4. Details panel
5. Viewport
6. Event Graph



BLUEPRINT EDITOR: TOOLBAR

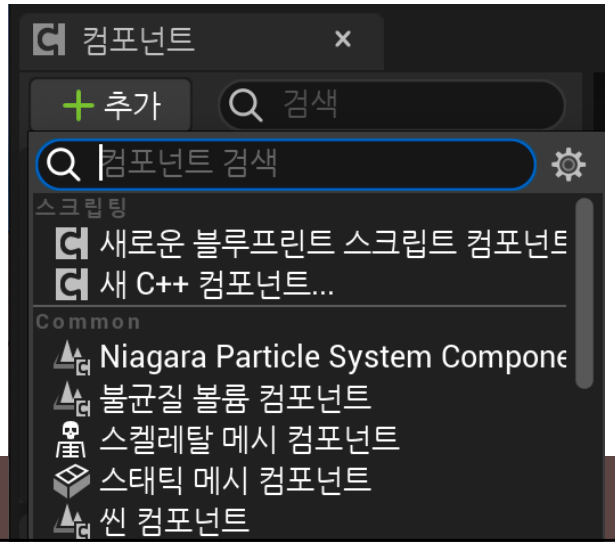
The **Toolbar**, located at the top of the Editor, has a few essential buttons for editing Blueprints:

- **Compile**: “Compiles” the Blueprint, necessary for validating the code and applying modifications.
- **Save**: Saves all changes made to the current Blueprint to disk.
- **Browse**: Shows the current Blueprint class in the Content Browser.
- **Find**: Searches nodes within a Blueprint.
- **Class Settings**: Opens Blueprint properties.
- **Class Defaults**: Allows for modification of the initial values of the Blueprint variables.

BLUEPRINT EDITOR: COMPONENTS PANEL

In the **Components** panel, various types of components can be added to the current Blueprint.

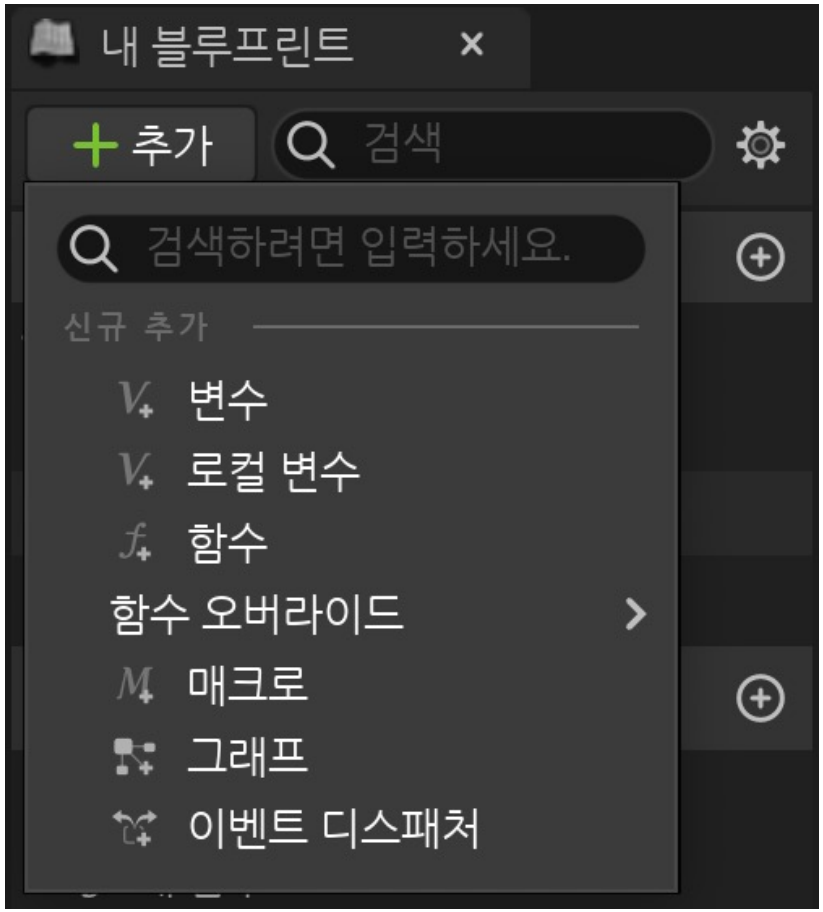
Examples of components are Static Meshes, lights, sounds, and geometric volumes used in collision tests.



BLUEPRINT EDITOR: MY BLUEPRINT PANEL

The **My Blueprint** panel is used to manage the variables, macros, functions, and graphs of the Blueprint.

It is separated into categories, and each category has a "**+**" button for adding new elements.

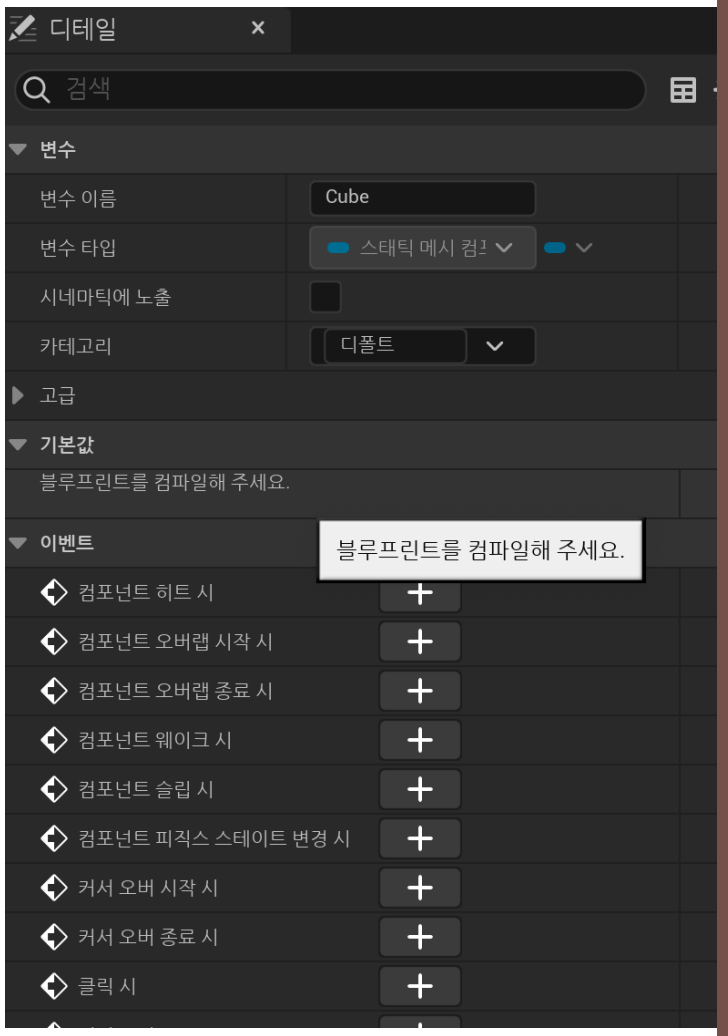


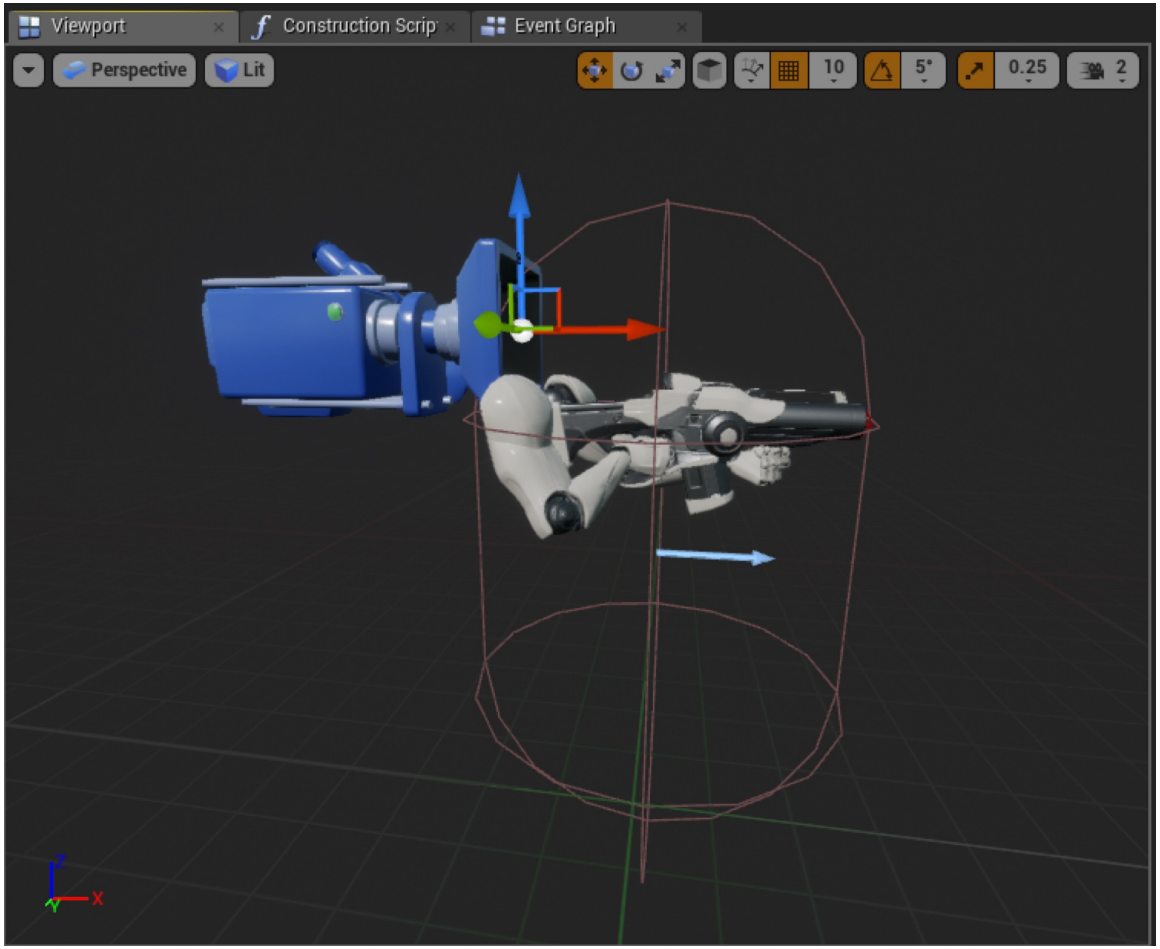
BLUEPRINT EDITOR: DETAILS PANEL

The **Details** panel shows the properties of the currently selected element of the Blueprint class, which can be a variable, function, or component.

These properties are organized into categories, and their values can be modified.

At the top of the panel, there is a search box that can be used to filter the properties.





BLUEPRINT EDITOR: VIEWPORT

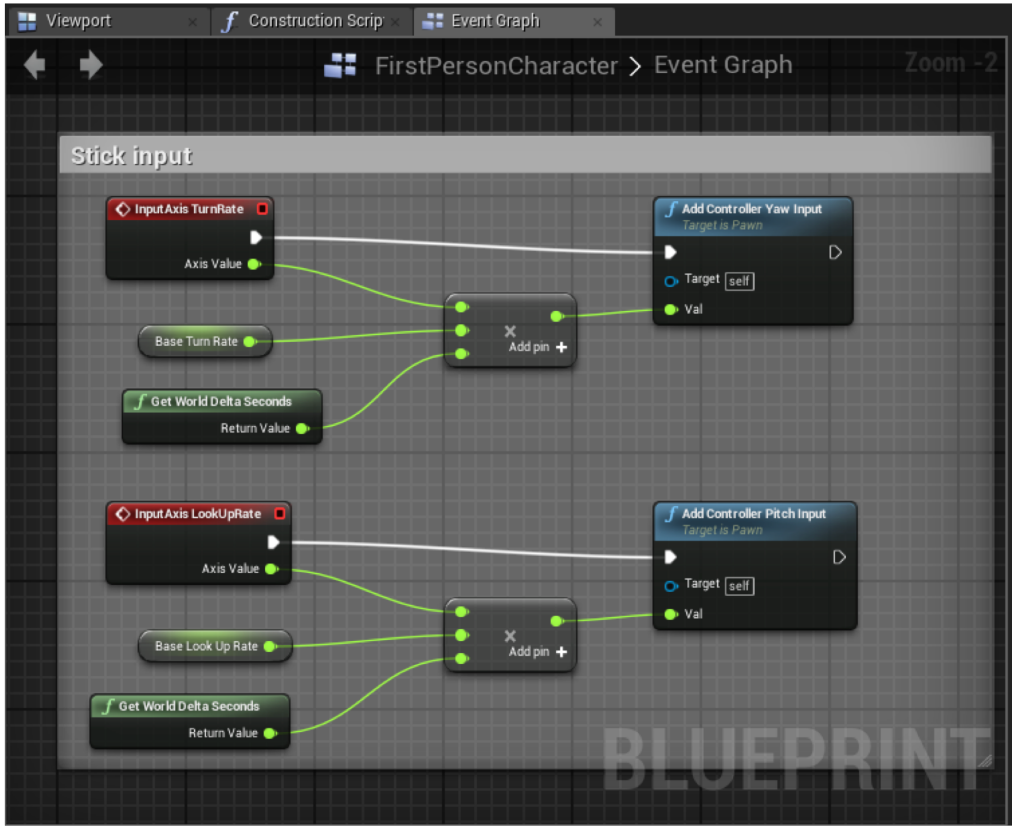
The **Viewport** contains the **visual representation of the components that are part of the Blueprint.**

The components can be manipulated in the Viewport using the transformation widgets in the same way as in the Level Editor.

BLUEPRINT EDITOR: EVENT GRAPH

The **Event Graph** is where the gameplay logic that determines a Blueprint class's behavior during gameplay appears.

The Event Graph contains events and actions represented by a node graph.

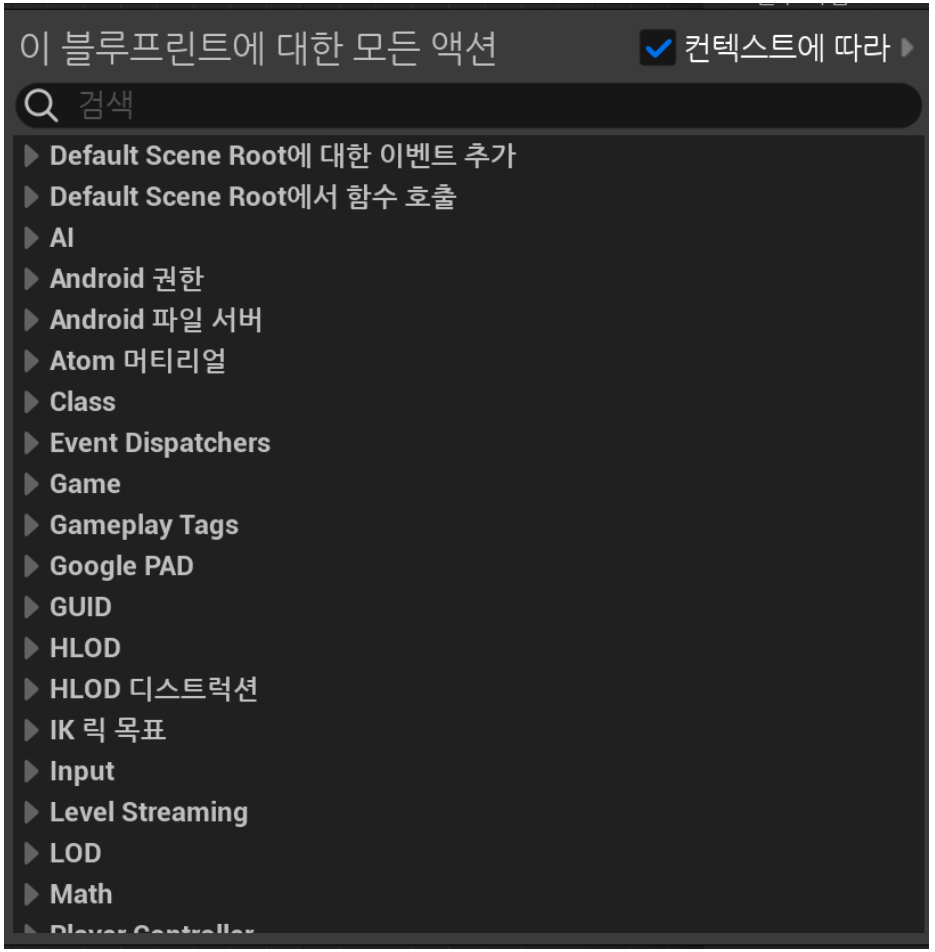


PLACING NODES

PLACING NODES: CONTEXT MENU

The **context menu** is used to add nodes to the graph. The nodes represent variables, operators, functions, and events.

To open the **context menu**, right-click in an empty area of the **Event Graph** or drag a wire from a pin of a node and release the mouse button.

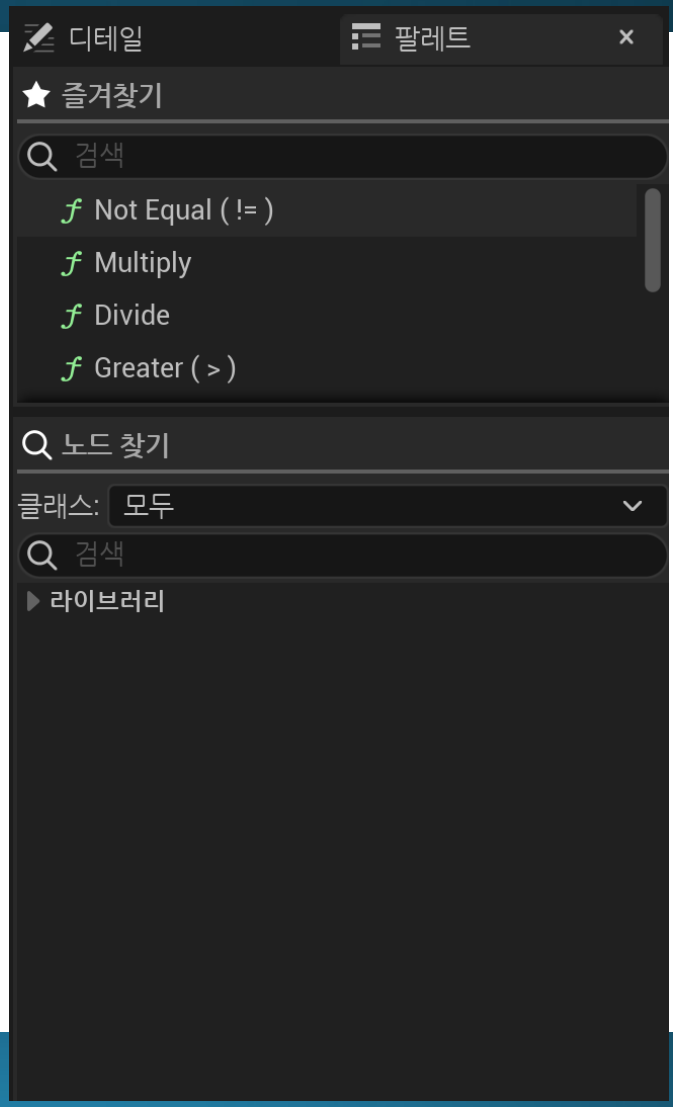


PLACING NODES: PALETTE PANEL

There is a **Palette** panel that can be opened by going to **Window > Palette** in the **Blueprint Editor**.

The Palette panel contains a list of all nodes that can be used in Blueprints.

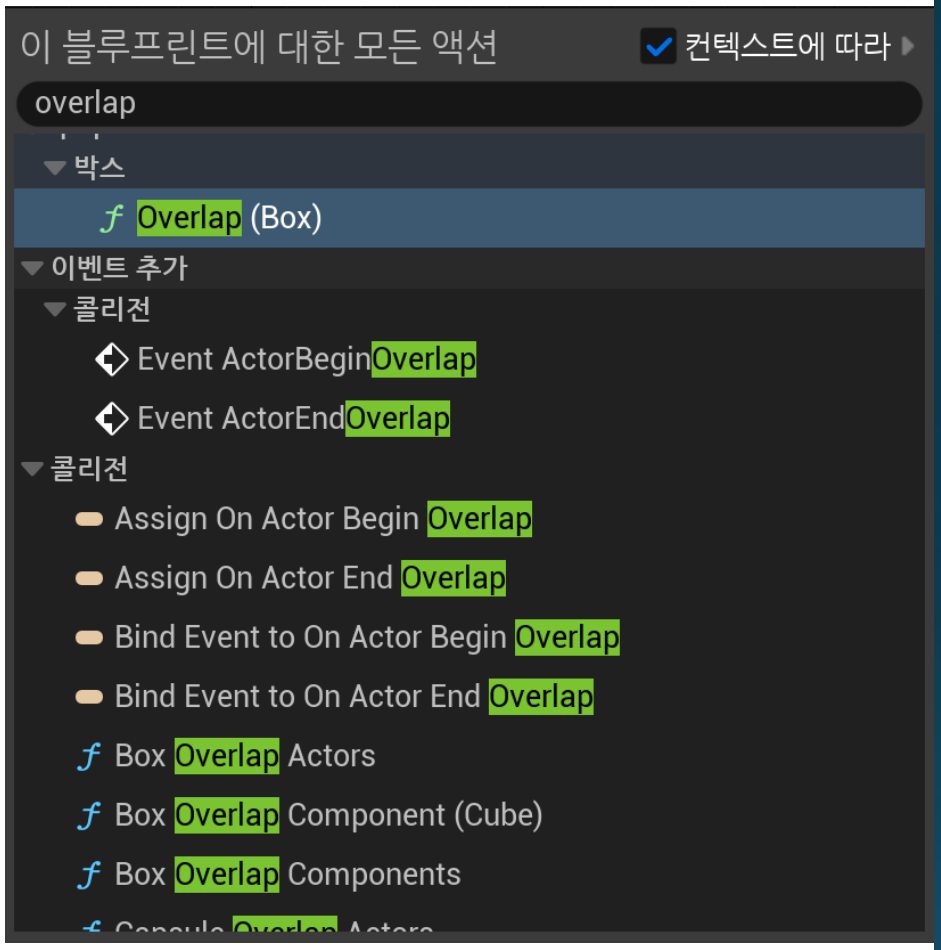
At the top of the panel there is a **Favorites** section that shows the favorite and most frequently used nodes. To set a node as a favorite, right-click on it in the **Palette** panel and choose “**Add to Favorites**”.



PLACING NODES: SEARCH BARS

The **context menu** has **a search bar** that filters the nodes list when the user is typing.

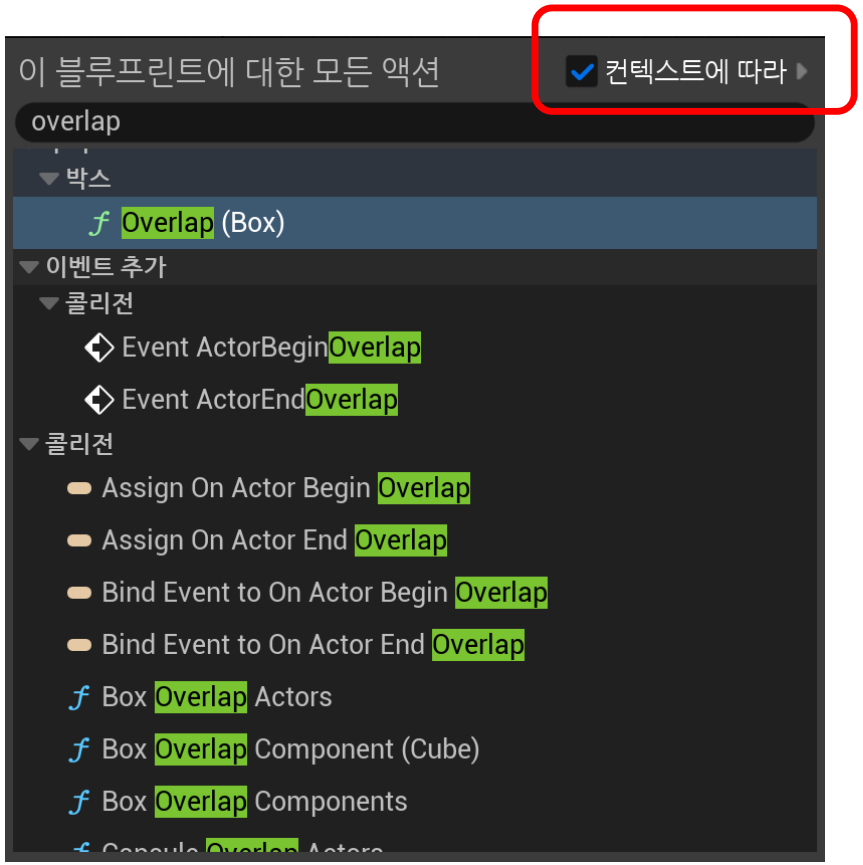
The **Palette** panel has **two search bars**, one for the **Favorites** section and one for the **Palette** list.

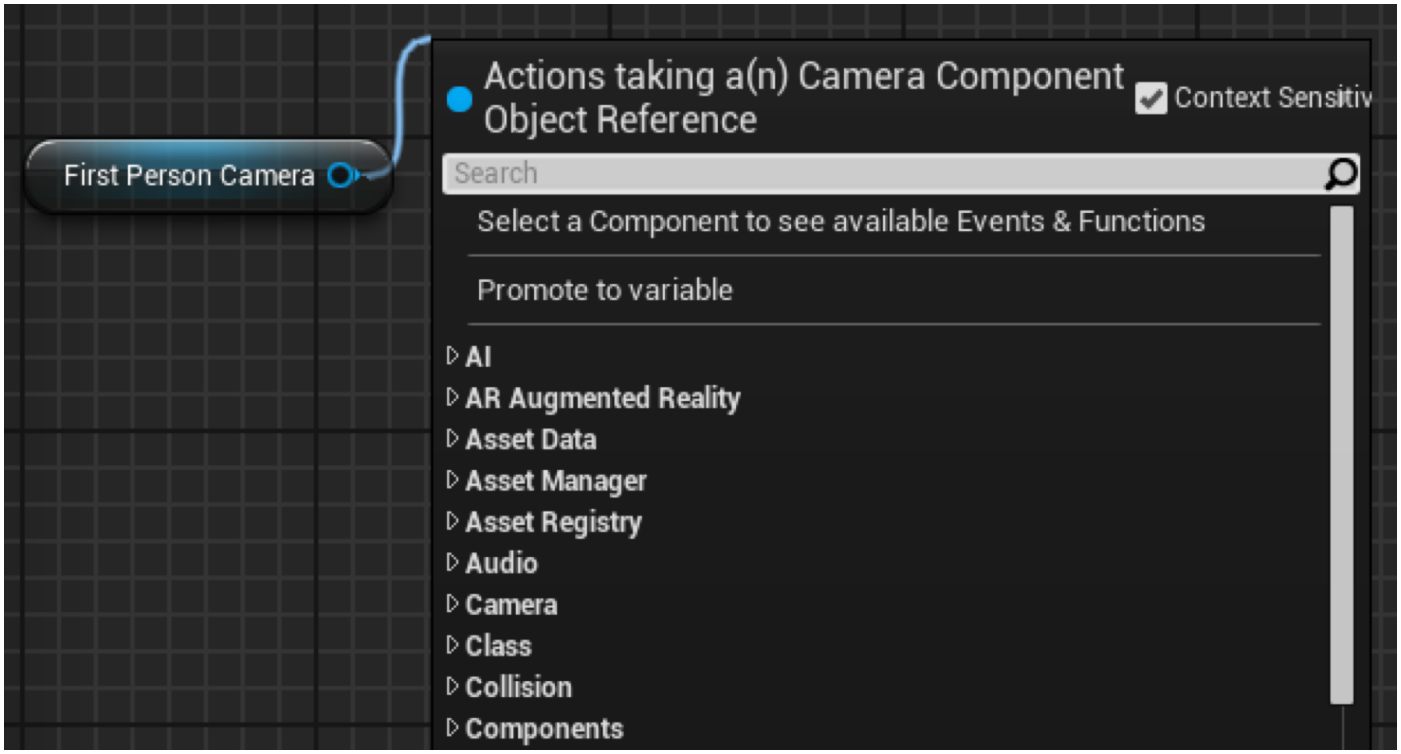


PLACING NODES: CONTEXT SENSITIVE

At the top of the context menu, there is a check box named “**Context Sensitive**”. If it is checked, the list of nodes will be filtered to actions that can be used in the current context.

If the context menu was opened by right-clicking in the Event Graph, the context will be the current Blueprint class. If it was opened by dragging a wire from a pin of a node, the context will be the pin type.





ex> First Person Camera 노드와
연결 가능한 것만 필터링

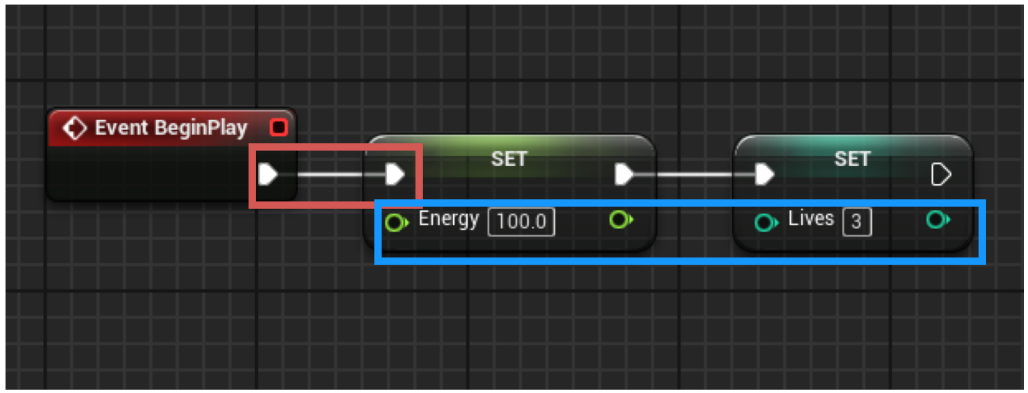
Nodes, Pins, and Wires

BLUEPRINT GRAPH EXECUTION

The **execution** of the nodes of a Blueprint starts with a red **Event** node and follows the **white wire** from left to right until it reaches the last node. After that, it moves on to another Blueprint event that has been triggered.

The white pins of nodes are called **execution pins**. The other colored pins are the **data pins**.

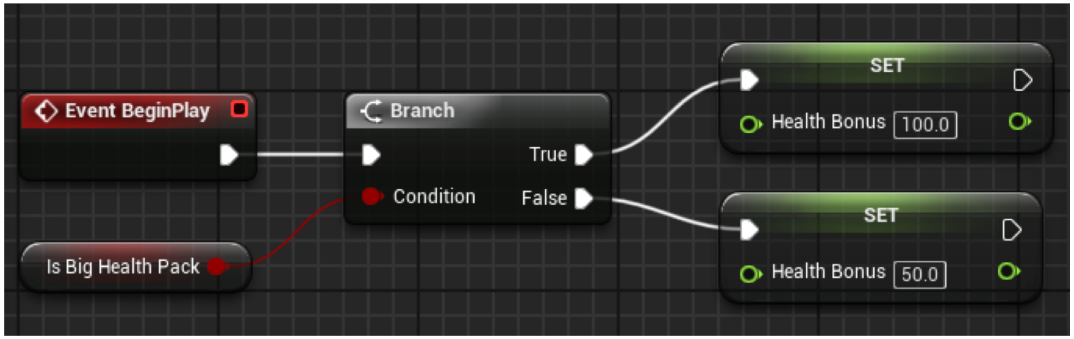
In the image on the right, values are assigned to the **Energy** and **Lives** variables when the **BeginPlay** event is triggered.



EXECUTION PATHS

There are some nodes that control the flow of execution of the Blueprint. These nodes determine the **execution path** based on conditions.

For example, the image on the right shows a **Branch** node that is using as a condition the value of the Boolean variable **Is Big Health Pack**. If the value is "**true**", then the execution will continue on the **True** pin. If the value is "**false**", the execution will continue on the **False** pin.

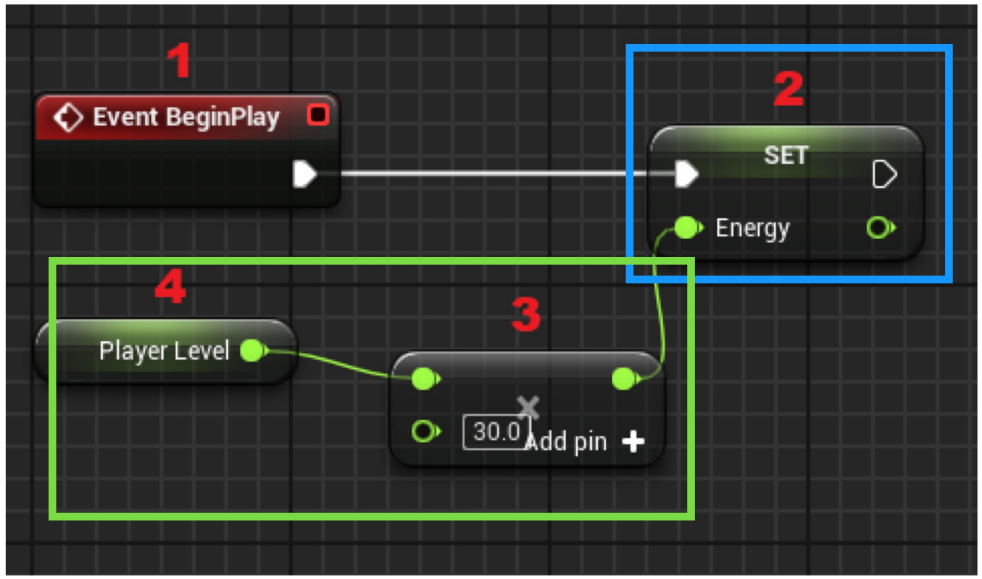


If (is Big Health Pack) Health Bonus = 100.0
Else Health Bonus = 50

DATA WIRES

When a node with data pins runs, it fetches the required data using the **data wires** before completing its execution.

In the image on the right, the execution starts with the **BeginPlay** event. The **Set** node assigns a new value to the **Energy** variable, but this value must be obtained using the data wire that is connected to a multiplication node that will need to get the value of the **Player Level** variable using another data wire.



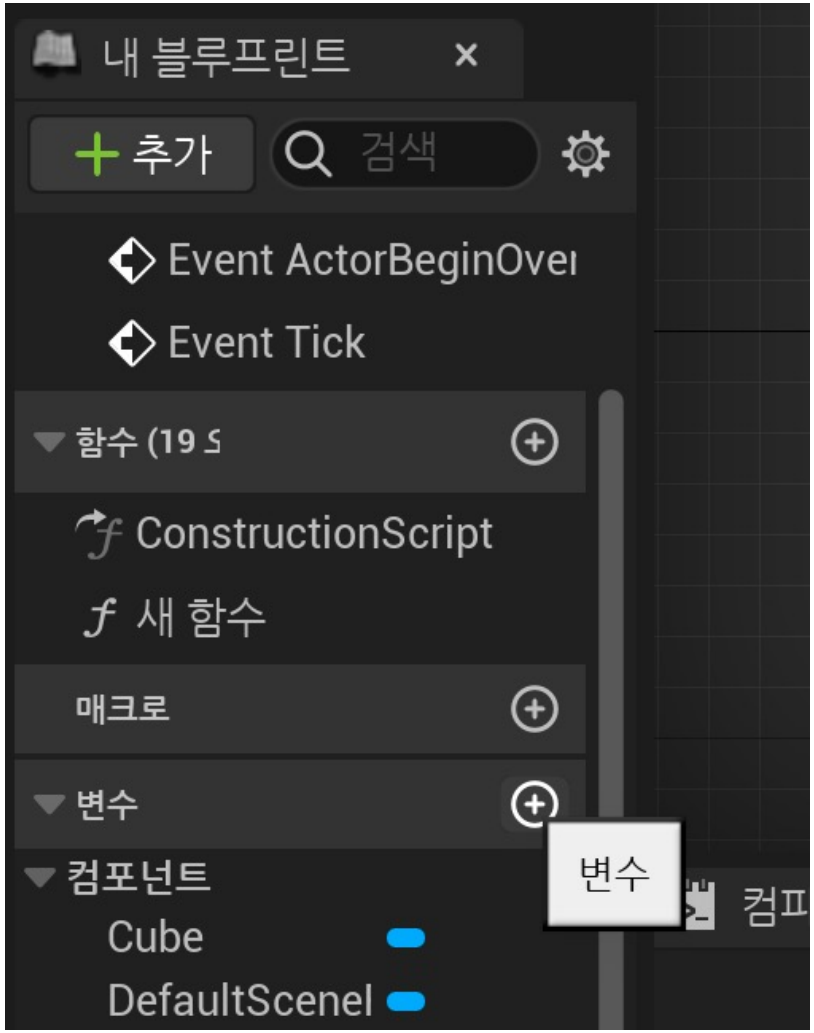
Energy = Player Level * 30

Variable

CREATING VARIABLES

Variables are used to store values and attributes in Blueprints that can be modified during the execution of the game. The variables can be of different types.

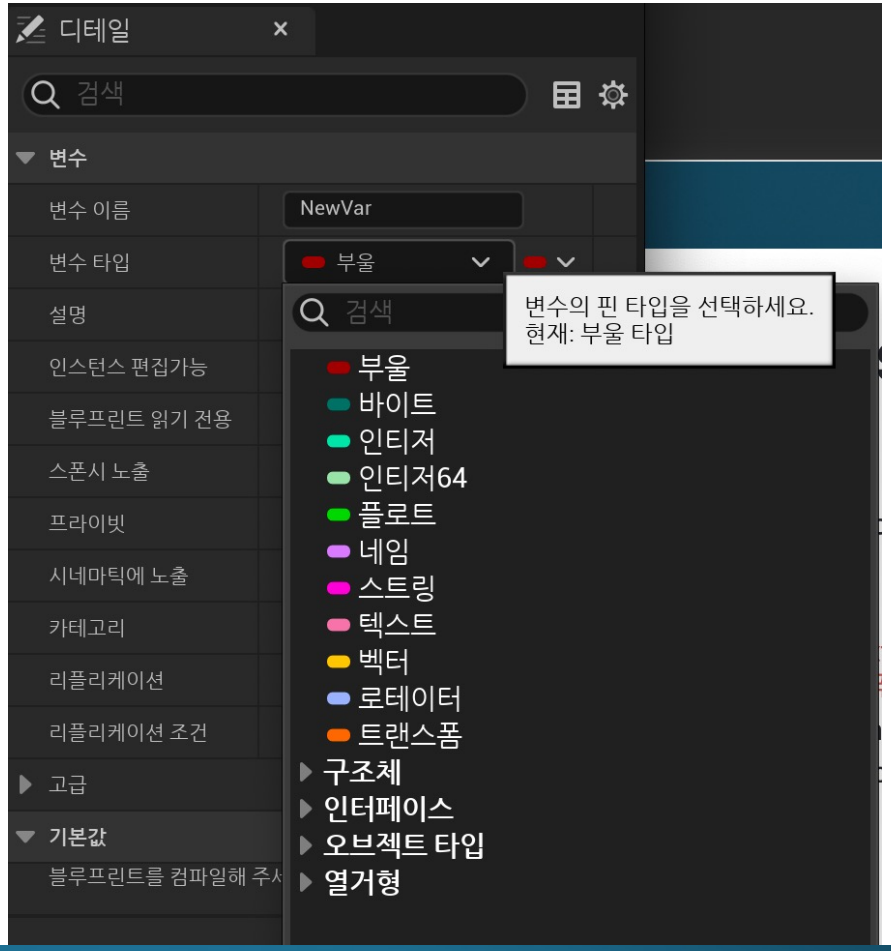
To create a variable, go to the **My Blueprint** panel in the **Blueprint Editor** and click the “**+**” button in the **Variables** category.



VARIABLE DATA TYPES

Following are common types of variables:

- **Boolean**: Can only hold the value “true” or “false”.
- **Integer**: Used to store integer values.
- **Float**: Used to store decimal values.
- **String / Text**: Used to store text. The Text variable is preferable since it supports localization(국제화/현지화).
- **Vector**: Contains the float values X, Y, and Z.
- **Transform**: Used to store location, rotation, and scale.

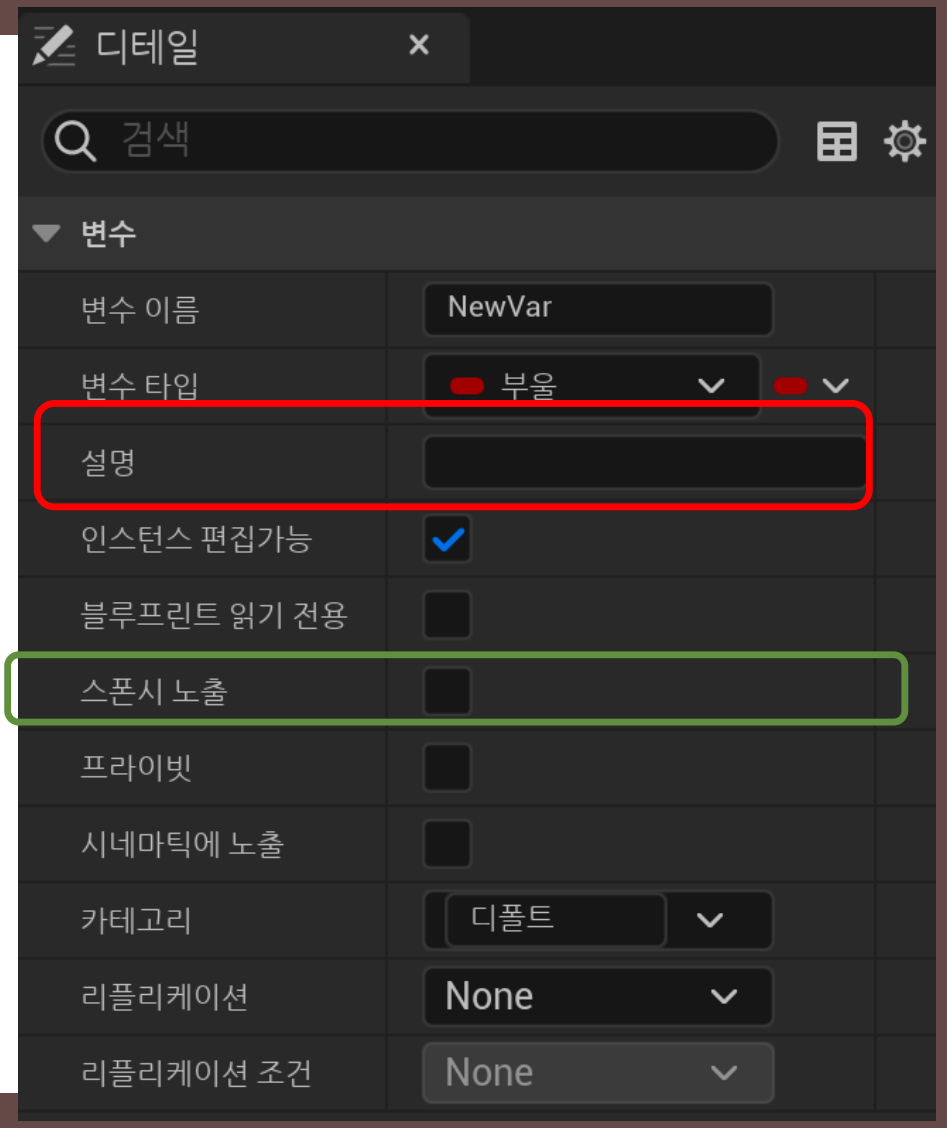


DETAILS PANEL

When a variable is selected, its properties are displayed in the **Details** panel. This is where changes can be made to the variable's name and type.

Other properties found in the Details panel include the following:

- **Instance Editable**: If checked, the variable can be changed in the instances that are in the Level. (인스턴스 상 변화 허용 여부)
- **Blueprint Read Only**: If checked, the variable cannot be changed by Blueprint nodes.
- **Tooltip**: Contains information shown when the cursor hovers over the variable.
- **Expose on Spawn**: If checked, the variable can be set when spawning the Blueprint.

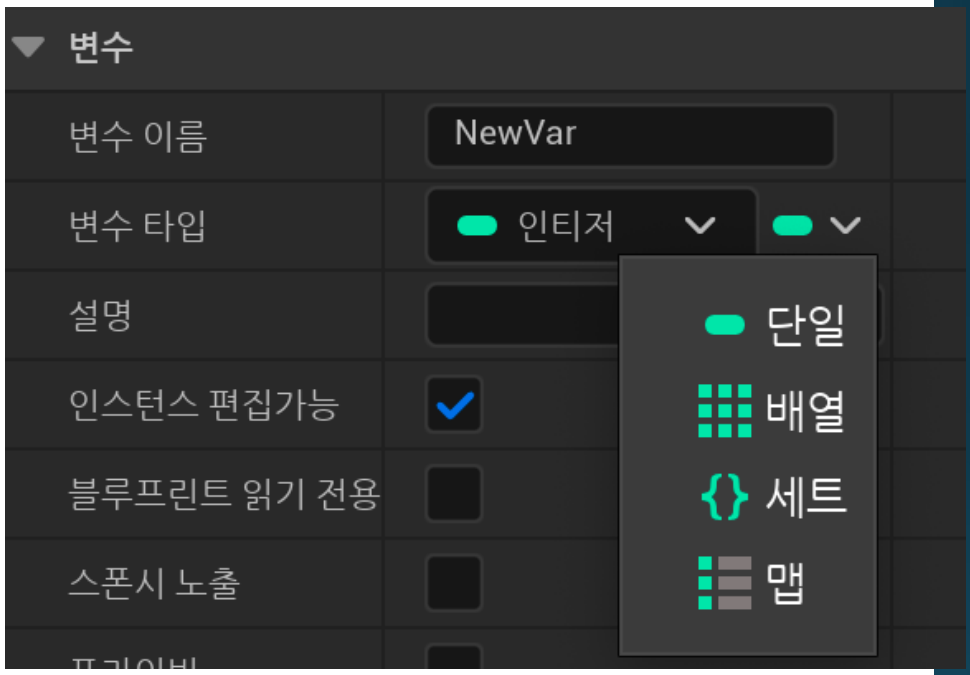


ARRAY, SET, AND MAP

The **Variable Type** property includes a button that is used to convert the variable into a **container**.

A container can store several elements of the same type. Listed below are the types of containers available.

- **Array**: An ordered list of values that are accessed using an index value.
- **Set**: An unordered collection of values. Duplicate values are not allowed.
- **Map**: A list that uses a key-value pair to define each entry. Duplicate key values are not allowed.



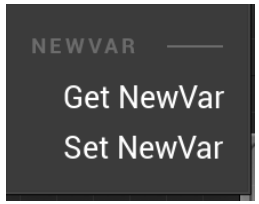
GETTERS AND SETTERS

When a variable is dragged and dropped into the Event Graph, a context menu appears with the options **Get** and **Set**.

Get nodes are used to read the value of the variable.

Set nodes are used to store a new value in the variable.

There are useful shortcuts to create **Get** and **Set** nodes. To create a **Get** node, press the **Ctrl** key when dragging and dropping a variable. The **Set** node is created using the **Alt** key.



Operator

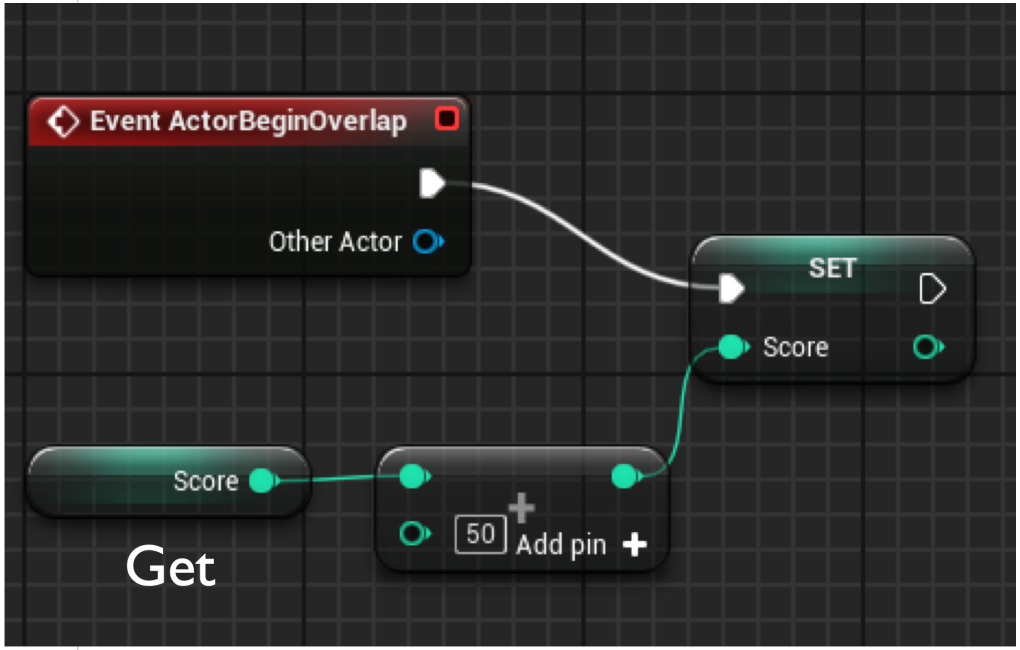
ARITHMETIC OPERATORS

The **arithmetic operators** (**+**, **-**, *****, **/**) can be used to create mathematical expressions in Blueprints.

The image on the right shows a simple expression that adds a value of "**50**" to the current **Score** variable, then sets the newly calculated value in the **Score** variable.

The "**+**" operator receives two input values on the left and gives the operation result on the right. To use more than two input values, just click on the **Add pin** option.

The input values can be entered directly into the nodes or can be obtained from variables.

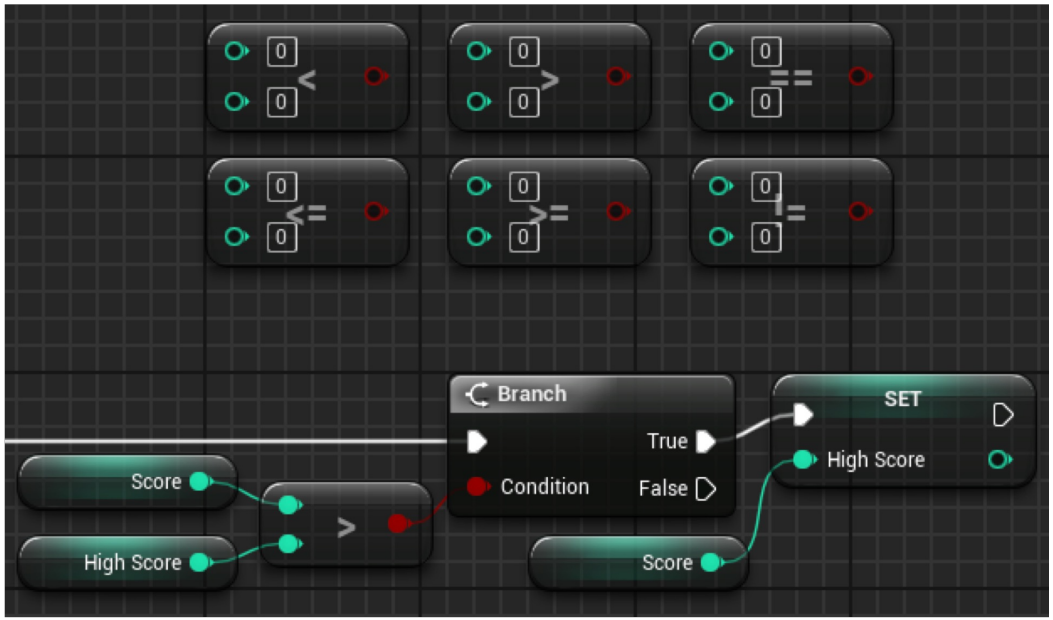


Score+=50

RELATIONAL OPERATORS

Relational operators perform a comparison between two values and return a Boolean value (“true” or “false”) as a result of the comparison.

The image on the right shows the relational operators and an example using a **Branch** node. At the end of a game, the current player’s score (**Score** variable) is compared with the highest recorded game score (**High Score** variable). If the player’s score is higher, the value of the **Score** variable will be stored in the **High Score** variable.



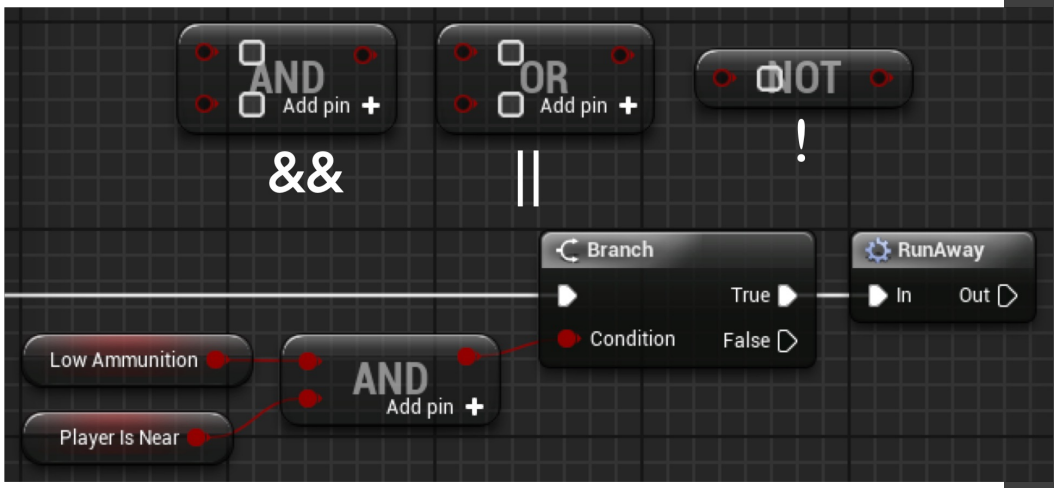
If (score > high score) high score=score

LOGICAL OPERATORS

Logical operators perform an operation between Boolean values and return a Boolean value (“true” or “false”) as a result of the operation. The main logical operators are as follows:

- **OR**: Returns a value of “true” if any of the input values are “true”.
- **AND**: Returns a value of “true” only if all input values are “true”.
- **NOT**: Receives only one input value, and the result will be the reverse value.

The example on the right simulates a simple decision of an enemy in a game. If the enemy is low on ammo (**Low Ammunition** variable) and the player is nearby (**Player Is Near** variable), then the enemy decides to run away.



A.I. for Enemy
- 적이 탄약이 적은 상태에서 플레이어가 다가오면, 도망가라

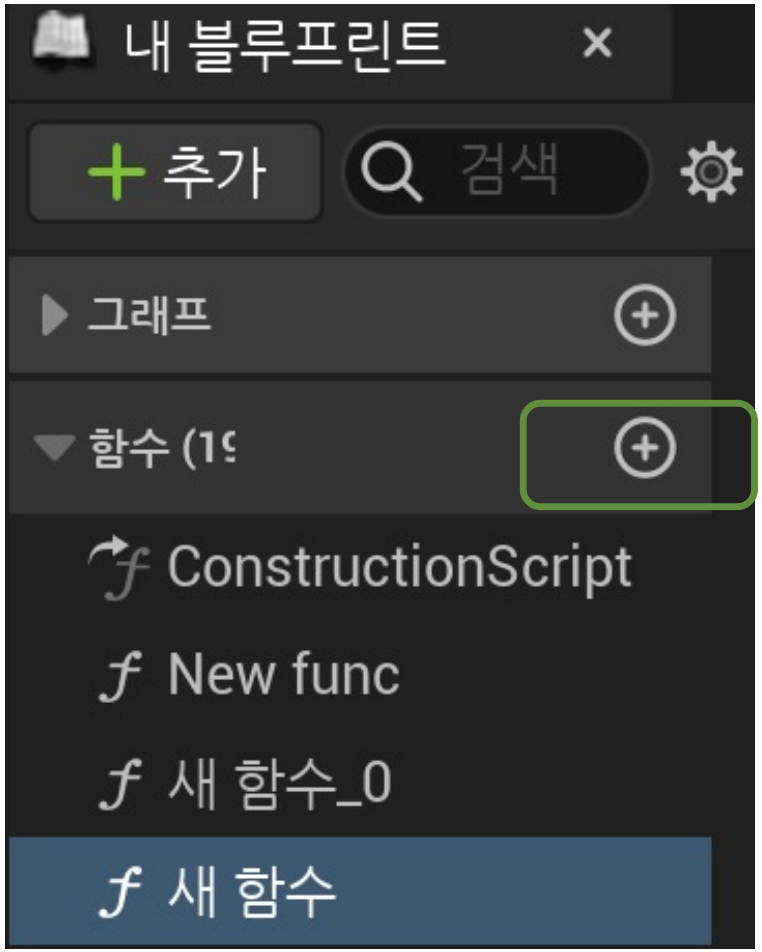
Function, Event and Macro

FUNCTIONS

Functions allow a set of actions that are executed in various parts of the Blueprint to be gathered in one place for easy organization and maintenance of the script.

Functions can be called from other Blueprints and allow the use of input and output parameters.

To create functions, go to the **My Blueprint** panel in the **Blueprint Editor** and click the "**+**" symbol in the **Functions** category.



FUNCTIONS:
INPUTS AND OUTPUTS

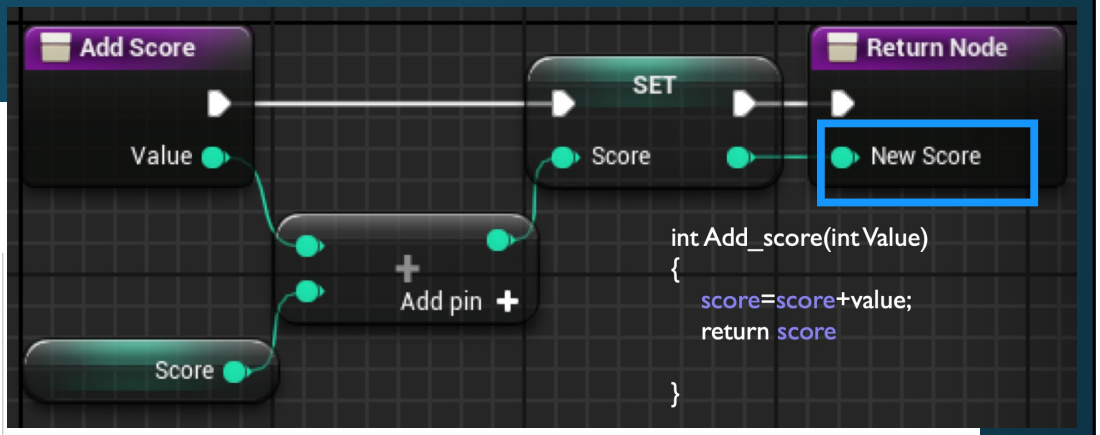
Input parameters are values that can be passed into a function.

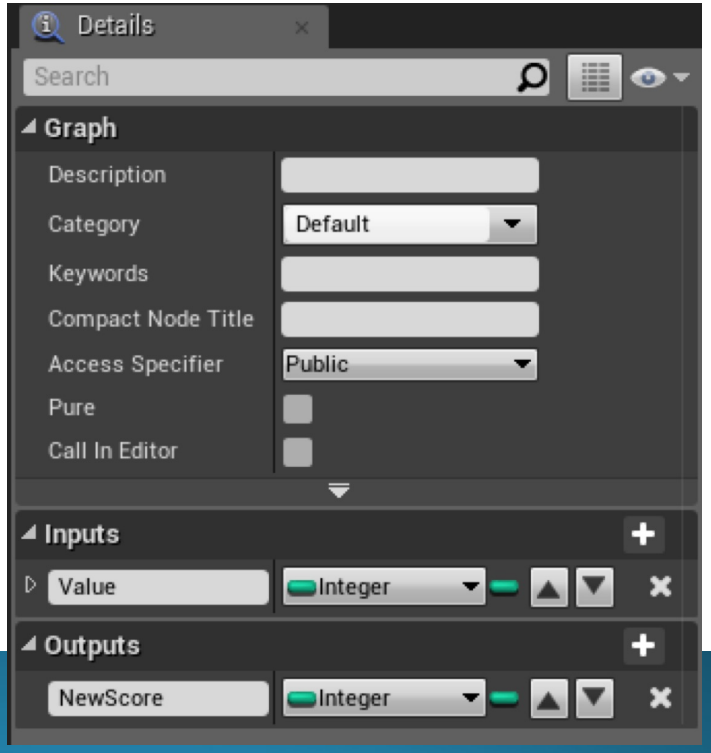
Output parameters are values that can be returned from a function.

To add input or output parameters, select the function in the **My Blueprint** panel and use the **Details** panel.

The images on the right show a function with an input parameter named “**Value**” that is added to the **Score** variable.

The result of the sum is set in the **Score** variable, then returned with the output parameter named “**New Score**”.

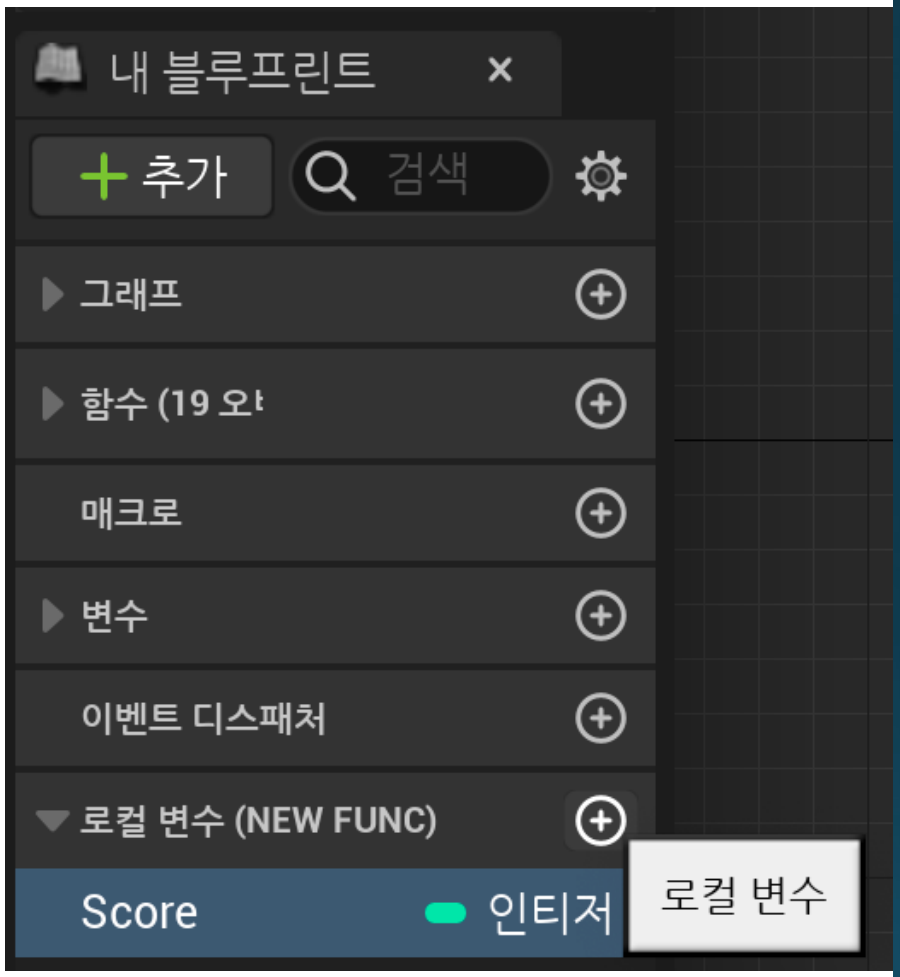




FUNCTIONS: LOCAL VARIABLES

Functions allow the use of **local variables** that are only visible inside the function. They are very effective at assisting in complex functions and do not mix with the other variables of the Blueprint.

To create a local variable, double-click on a function to edit it, then look at the **My Blueprint** panel. At the bottom of the panel, you will find a category named **Local Variables** with the name of the function in parentheses. Click the "**+**" button in the **Local Variables** category.

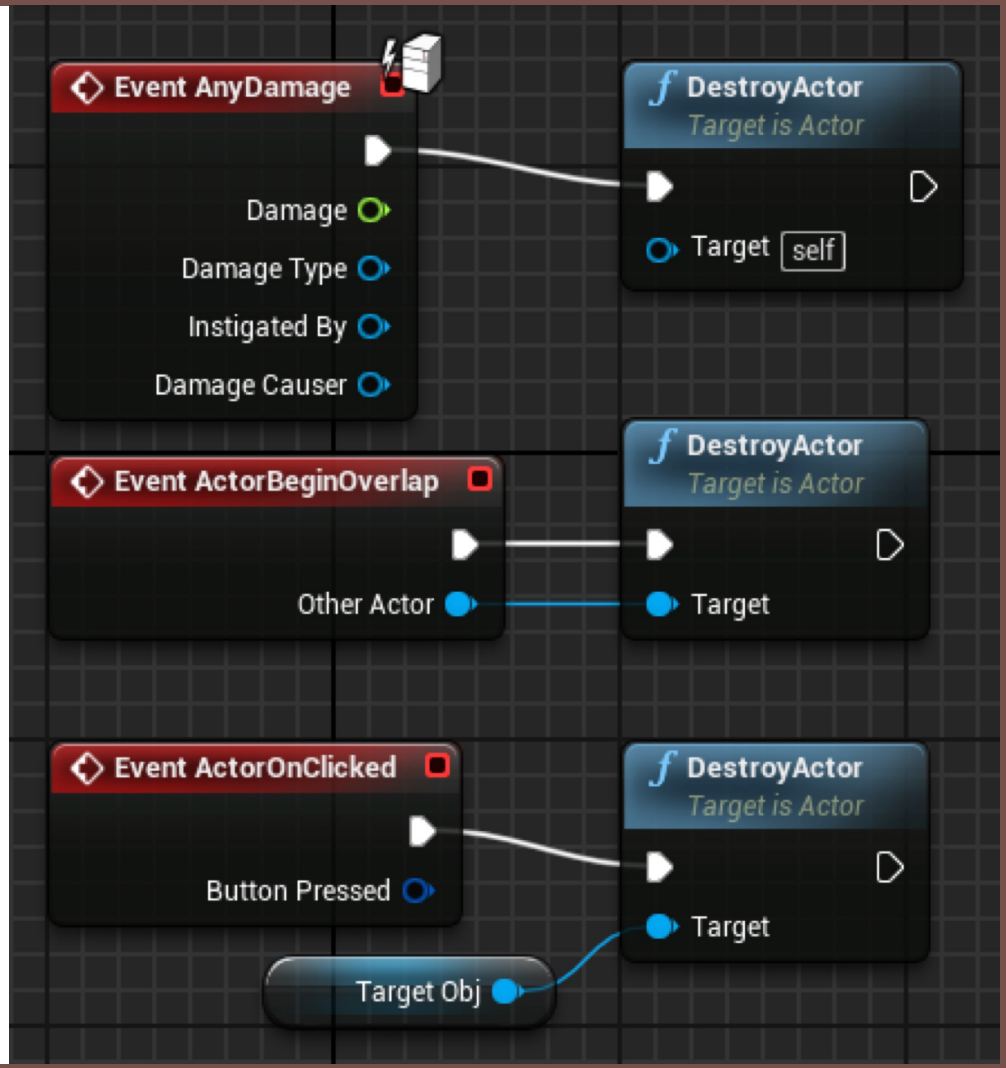


FUNCTIONS:
THE TARGET PARAMETER

The **Target** parameter is common to several functions and indicates the object that will be modified with the function call.

The default value for this parameter is "**self**", which is a special reference to the Actor or Object instance that owns the script being executed.

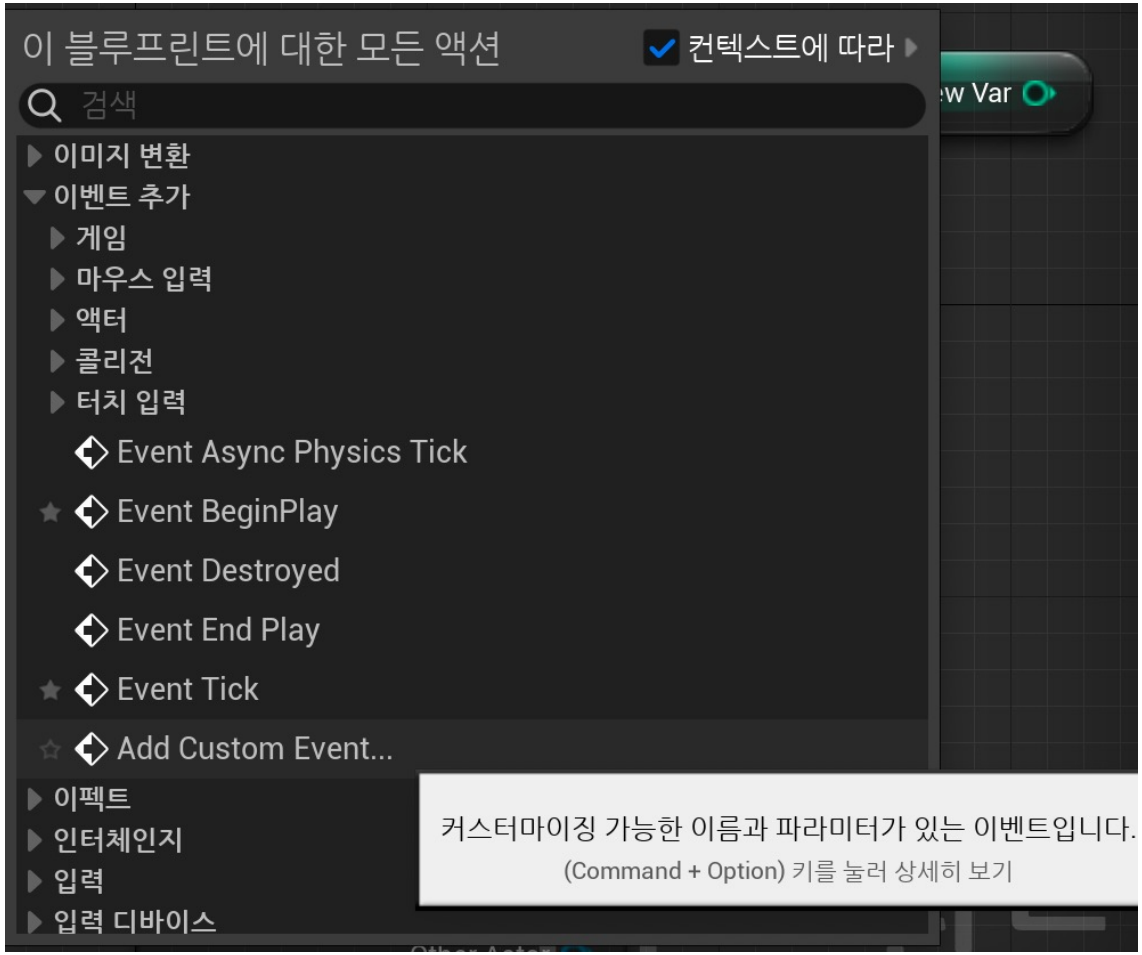
The image on the right shows different ways to use the **Target** parameter of the **DestroyActor** function.



CUSTOM EVENTS

Unreal Engine provides a number of predefined events, but it is possible to create new ones to use in a Blueprint. These events can be called from both the Blueprint they are defined in and other Blueprints.

To create a **custom event**, right-click in the **Event Graph**, expand the **Add Event** category, and select "**Add Custom Event...**".

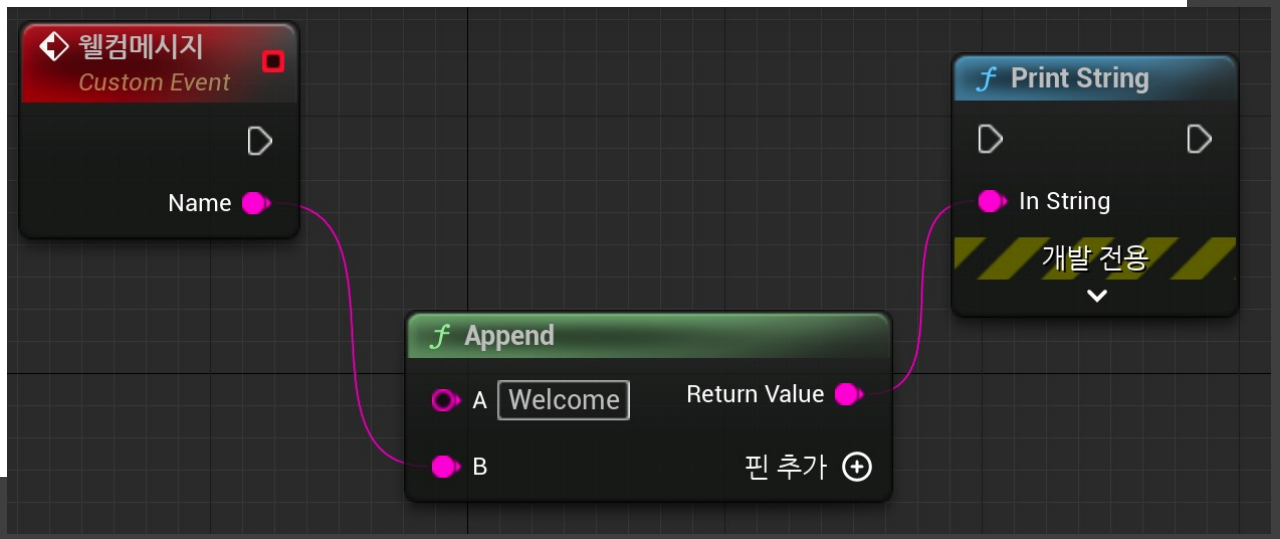


CUSTOM EVENTS: INPUT PARAMETERS

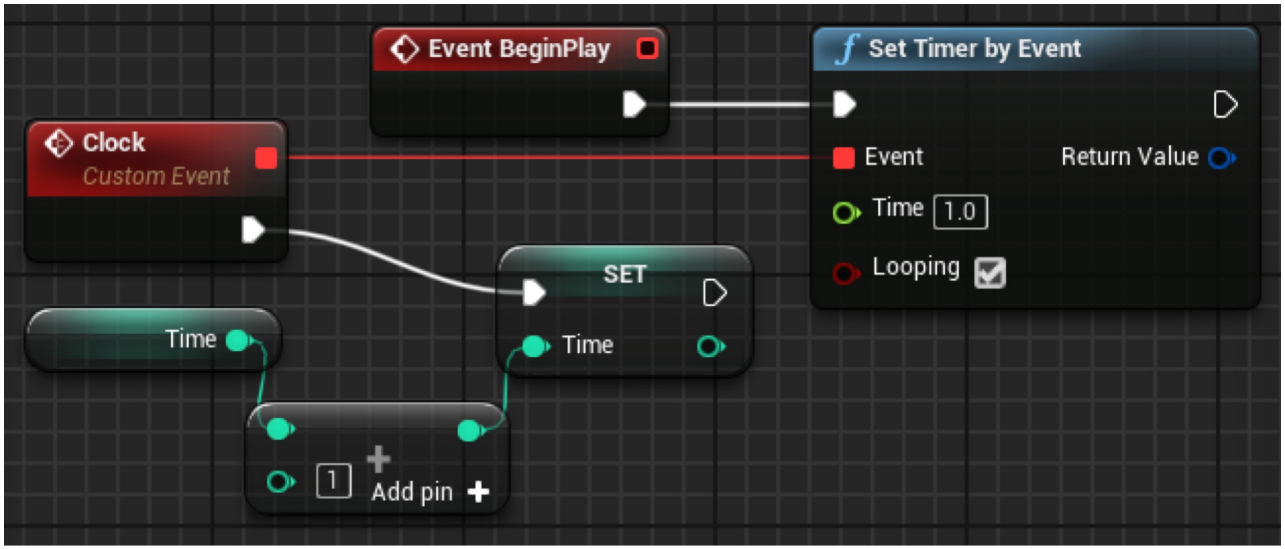
Selecting a Custom Event node allows you to manage the event name and input parameters. Events do not have output parameters.

The images on the right show a custom event named “**WelcomeMessage**” with an input parameter called “**Name**”.

The event will create and print on screen a custom message using the name passed as a parameter.



CUSTOM EVENTS: DELEGATES



An event has a small red square in the right corner that is known as a **delegate**. This is just a reference to the event. Some actions receive an event as a parameter, and using the delegate makes that possible.

In the image on the right, the delegate of the custom event named "**Clock**" is wired to the **Set Timer by Event** node's **Event** input pin, so the **Clock** event will be called every second.

Function 실행시 custom 이벤트 발생

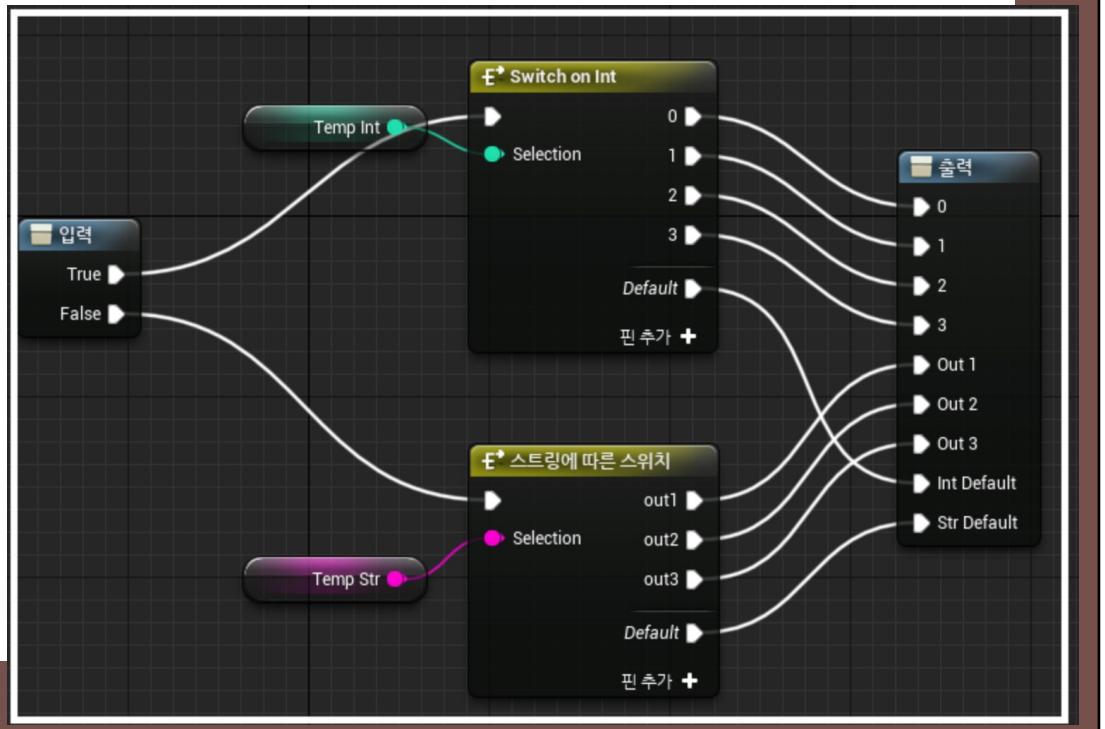
MACROS

매크로 : 여러 노드들로 이루어진 로직의 흐름에 별명을 붙인 것
함수 : 일반적으로 사용할 단일 노드를 정의하는 것

Another way to gather actions in one common place is to use **macros**. A macro is similar to a collapsed graph of nodes. At compile time, the actions of a macro are expanded in the places that the macro is being used.

Macros can have input and output parameters, as well as several input and output execution pins.

함수는 입출력 실행핀을 하나씩만 가질 수 있고,
매크로는 실행핀을 원하는 만큼 추가하여 사용



MACROS VS. EVENTS VS. FUNCTIONS

Macros, custom events, and functions provide different ways to organize script. Each of them has its advantages and limitations. One thing they have in common is that they all have input parameters.

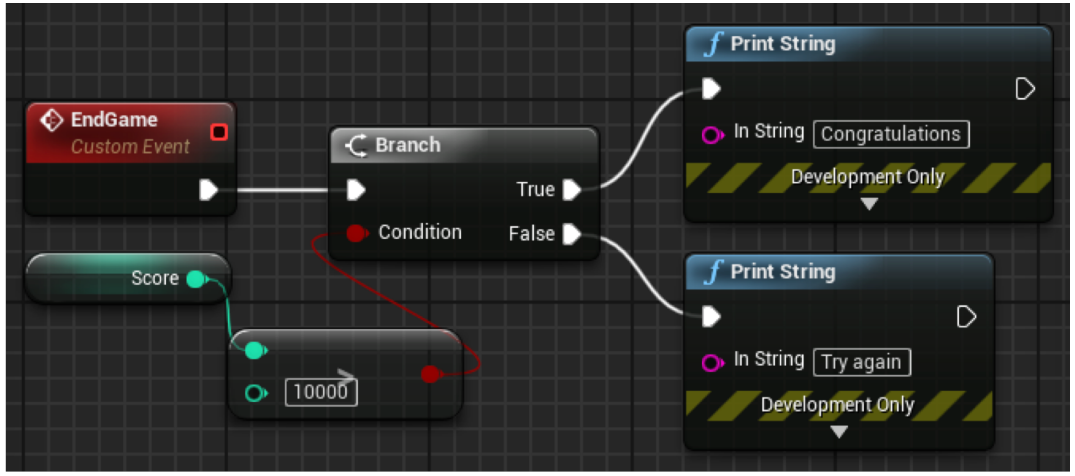
- **Macros** have output parameters and can have many execution paths. They cannot be called from another Blueprint.
- **Events** do not have output parameters. They can be called from other Blueprints and have the “delegate” reference. They support Timelines.
- **Functions** can be called from other Blueprints and have output parameters. Functions do not support latent actions, such as the Delay action.

Program Flow

BRANCH NODE

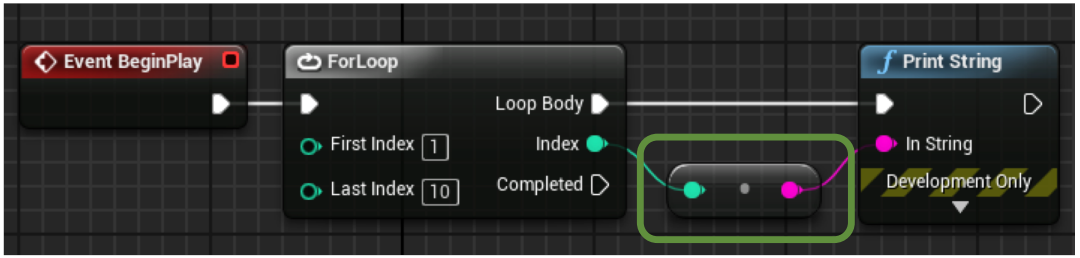
The **Branch** node directs the flow of execution of a Blueprint based on the value of the Boolean input “Condition”, which can be “true” or “false”.

In the image on the right, there is a custom event that is called at the end of the game. The **Branch** node is used to test if the score is greater than “**10000**”. A different message will be shown based on the result.



게임 종료 이벤트 발생시, 점수가 10000점 이상인지 아닌지 구별

FOR LOOP NODE

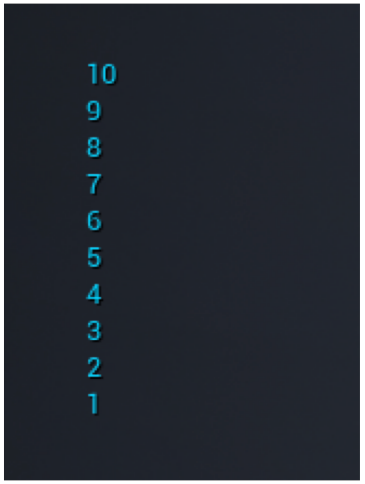


The **ForLoop** node performs the set of actions that are associated with the output pin **Loop Body** for each index.

When the **ForLoop** node completes its execution, the output pin **Complete** is triggered.

On the right, a **ForLoop** node is used to execute the **Print String** node ten times. The value of the **Index** output pin of the **ForLoop** node is used as the input for the **Print String** node.

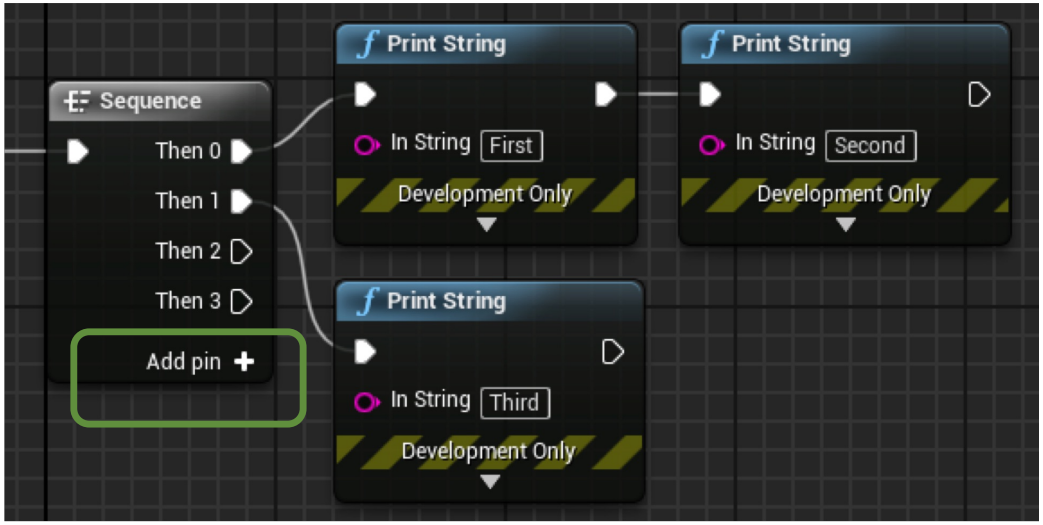
The conversion node is automatically created by the Editor when an integer value is connected to a string input.



SEQUENCE NODE

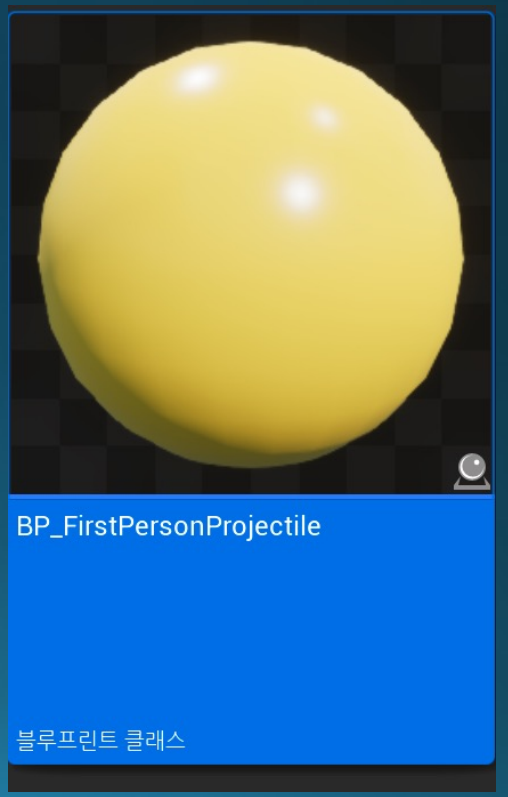
A **Sequence** node can be used to help organize other Blueprint actions. When triggered, it executes all the nodes connected to the output pins in sequential order—that is, it executes all the actions of pin **Then 0**, then all the actions of pin **Then 1**, and so on.

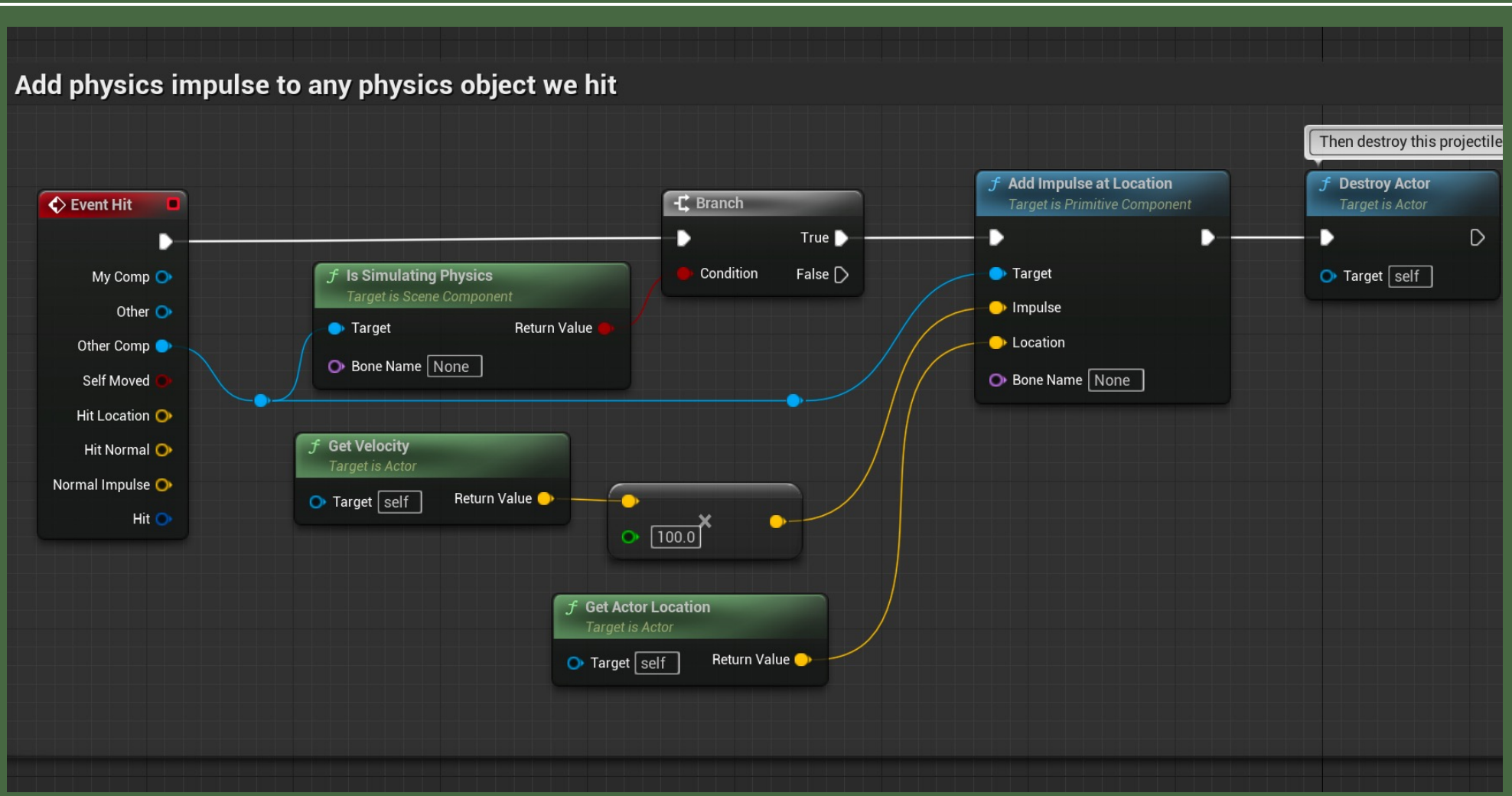
Output pins can be added using the **Add pin +** option. To remove a pin, right-click on the pin and choose the **Remove execution pin** option.



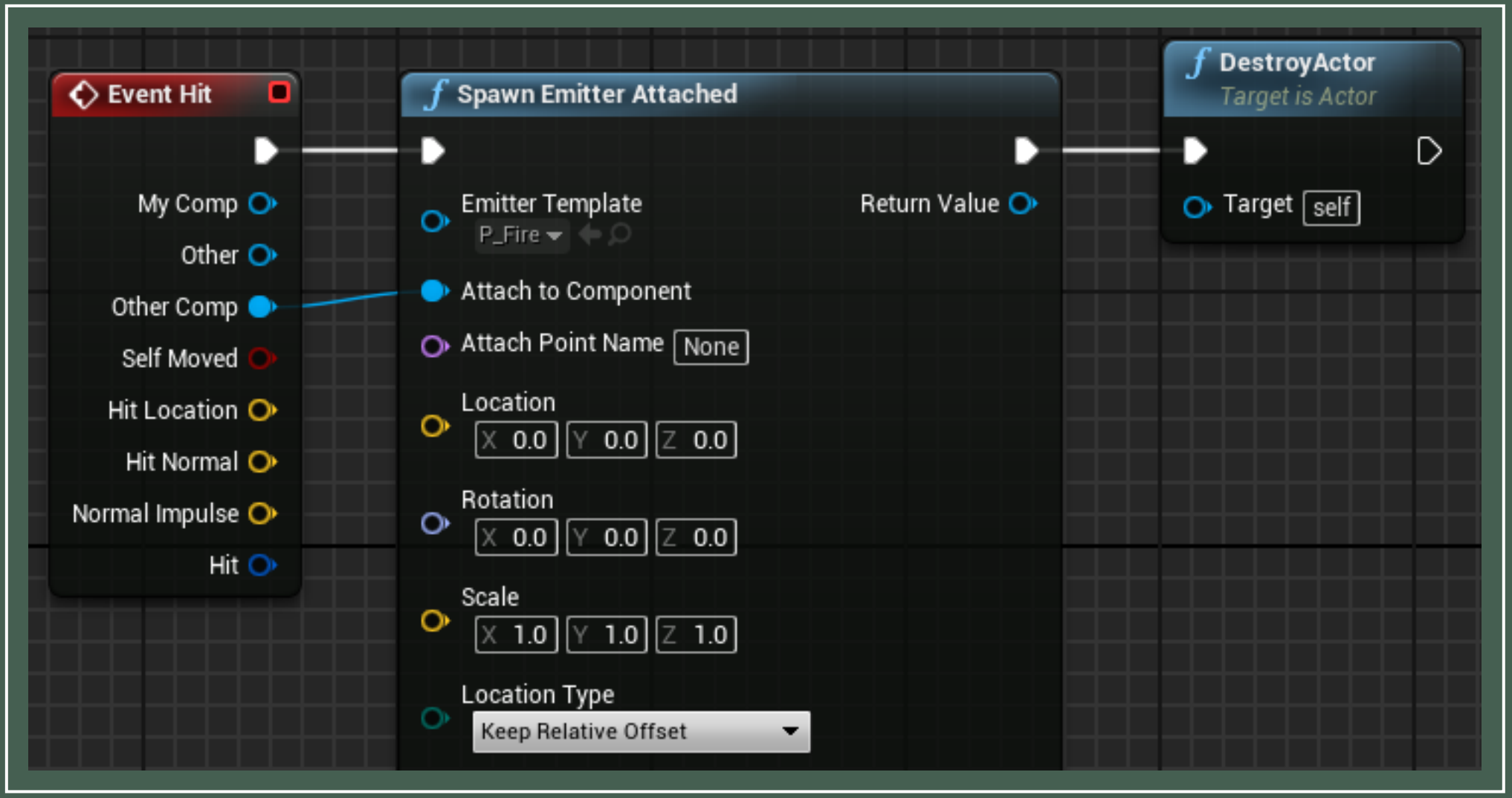
Example

1. Create a new project using the **First Person** template with **starter content**, or use an existing one.
2. Open the **FirstPersonProjectile** Blueprint (path: /Game/FirstPersonBP/Blueprints).




Add physics impulse to any physics object we hit
Event Hit
My Comp
Other
Other Comp
Self Moved
Hit Location
Hit Normal
Normal Impulse
Hit
Is Simulating Physics
Target is Scene Component
Target
Return Value
Bone Name
None
Branch
True
Condition
False
Add Impulse at Location
Target is Primitive Component
Target
Impulse
Location
Bone Name
None
Then destroy this projectile
Destroy Actor
Target is Actor
Target
self
Get Velocity
Target is Actor
Target
self
Return Value
100.0
Get Actor Location
Target is Actor
Target
self
Return Value

이걸로 교체하면?


Event Hit
My Comp
Other
Other Comp
Self Moved
Hit Location
Hit Normal
Normal Impulse
Hit
Spawn Emitter Attached
Emitter Template
P_Fire
Return Value
Attach to Component
Attach Point Name
None
Location
X 0.0
Y 0.0
Z 0.0
Rotation
X 0.0
Y 0.0
Z 0.0
Scale
X 1.0
Y 1.0
Z 1.0
Location Type
Keep Relative Offset
DestroyActor
Target is Actor
Target
self