LECTURE 6

Advanced Blueprint Concepts 1

LECTURE GOALS AND OUTCOMES

Goals

The goals of this lecture are to

- Show how to use structures
- Present the array, set, and map containers
- Show how to use enumerations
- Introduce data tables
- Present latent functions
- Explain the functions used to save and load data

Outcomes

By the end of this lecture you will be able to

- Use structures and enumerations
- Use containers to organize data
- Use latent functions
- Implement a basic save/load system

ARRAYS

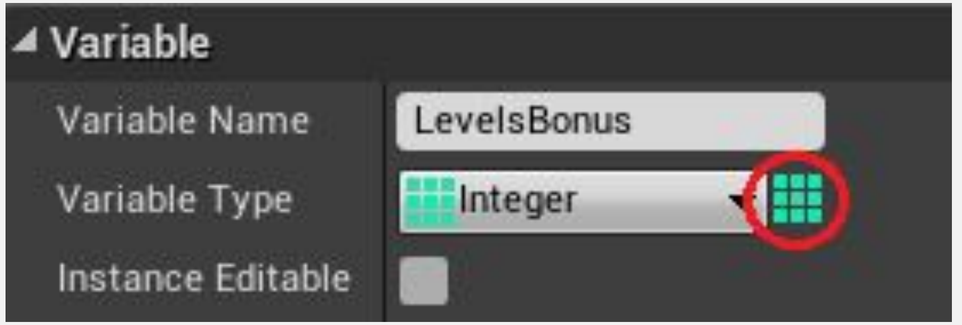
An **array** is an ordered list, and you set and get its elements via the use of an integer-based index.

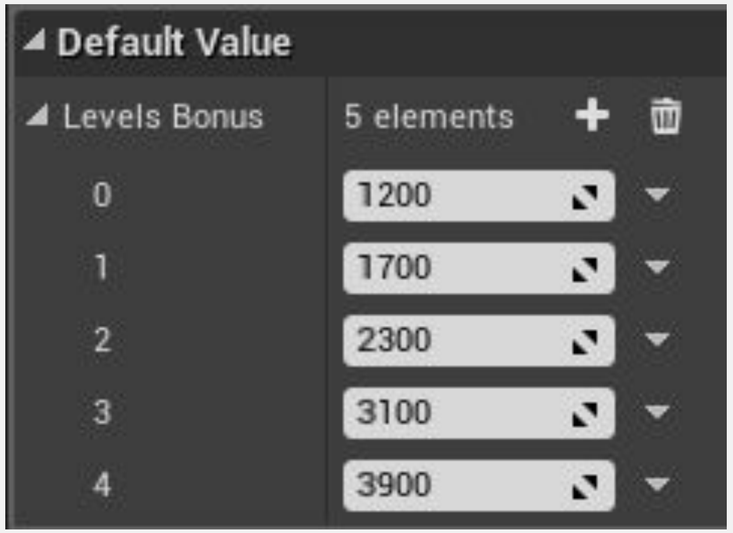
The use of arrays allows grouping of variables of the same type. Creating an array in Blueprints is very simple.

First, create a new variable and select the desired type.

Click the icon next to the **Variable Type** drop-down and choose “**Array**” (see top image on right).

After compiling the Blueprint, it is possible to fill in the array elements with default values, as shown in the bottom image.

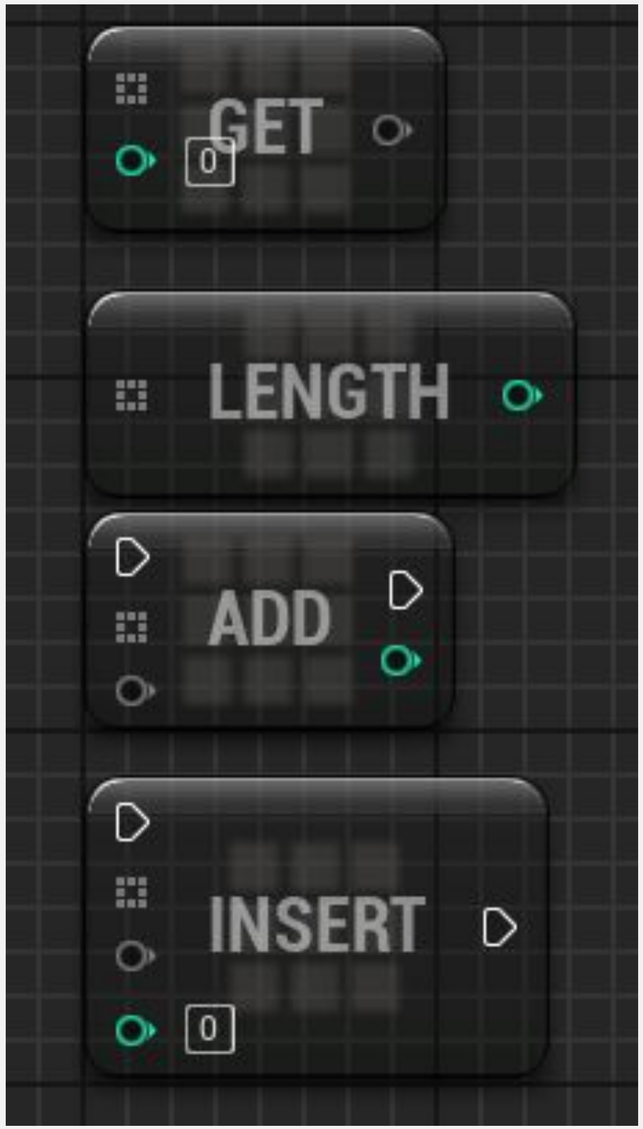




ARRAYS:
MAIN NODES

The main nodes related to arrays are as follows:

- **Get**: Returns the element that is in the position specified by the index used.
- **Length**: Returns the number of array elements.
- **Add**: Adds a new element at the end of the array.
- **Insert**: Inserts a new element at the position specified by the index parameter.



SETS

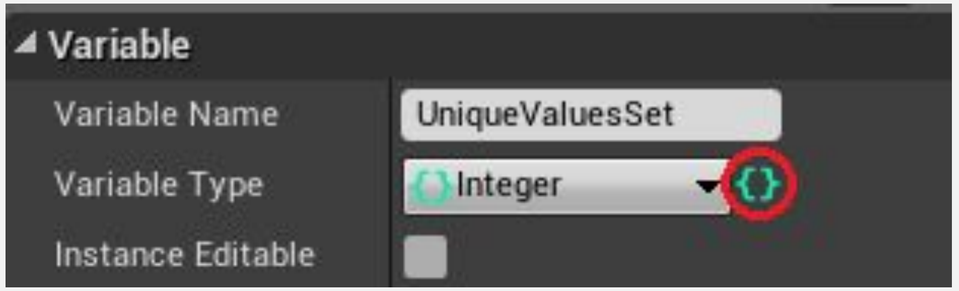
A **set** is a type of container similar to an array, but unlike an array it is an unordered list of elements that are searched for based on their value. There is no index. It can be used to group variables of the same type. An important difference is that sets do not allow duplicate elements.

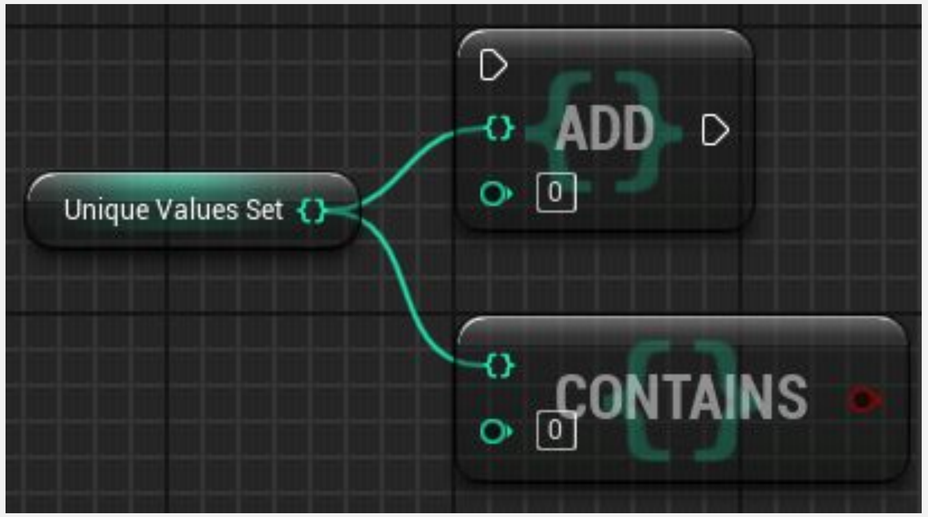
The key value used to find an item is the item itself.

To define a variable as a set, click the icon next to the **Variable Type** drop-down and choose “**Set**” (see top image on right).

Following are some common nodes related to sets:

- **Add**: Adds an item to the set.
- **Contains**: Checks to see if the set contains the given item.
- **Remove**: Removes an item from the set.
- **Length**: Returns the number of items in the set.





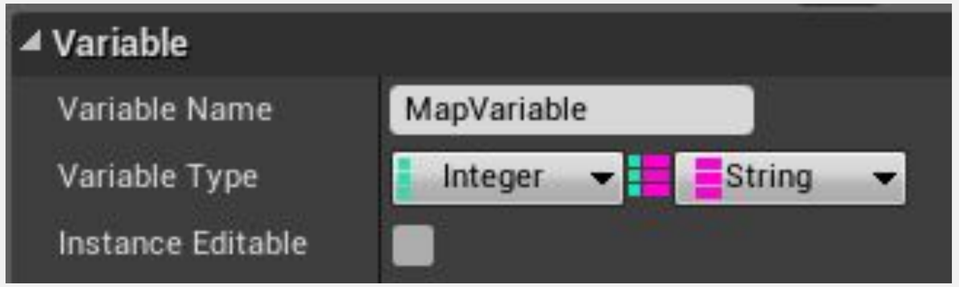
MAPS

There is another type of container called a **map**. To define a variable as a map, click the icon next to the **Variable Type** drop-down and choose “**Map**”.

Each element of a map has a key associated with a value. A map is unordered and is searched using the key value. The top image on the right shows a map whose key type is “**Integer**” and whose value type is “**String**”.

The key values of a map must be unique.

In the bottom image is an example of a map where a number is associated with the name of a game item.



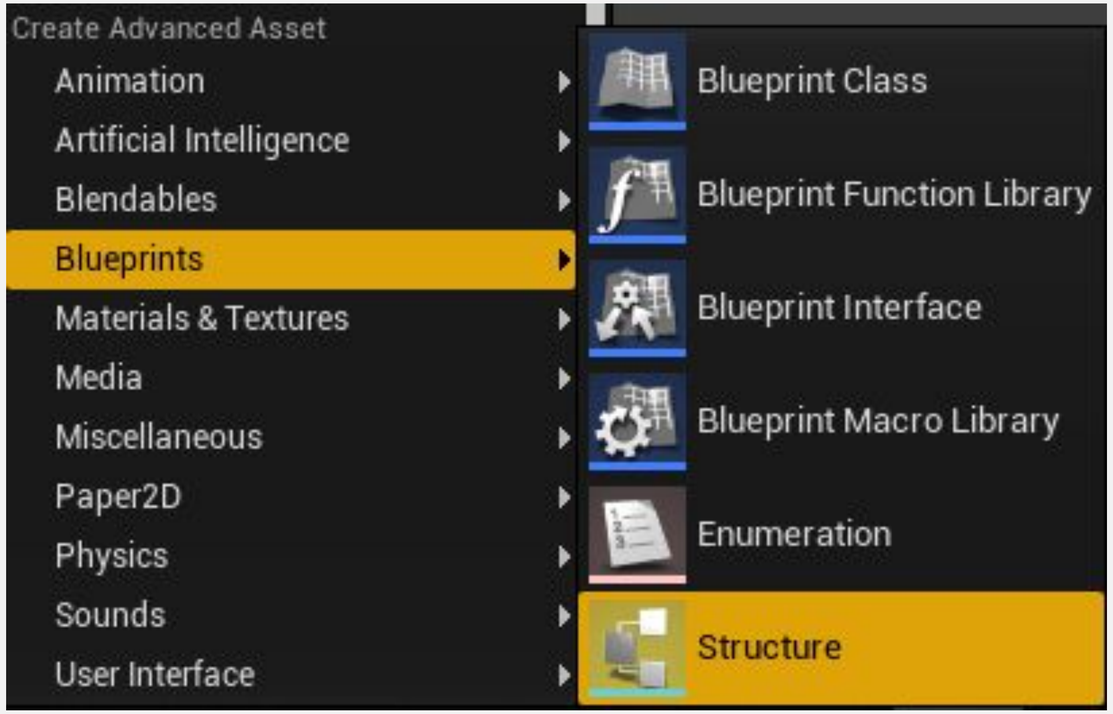
STRUCTURES

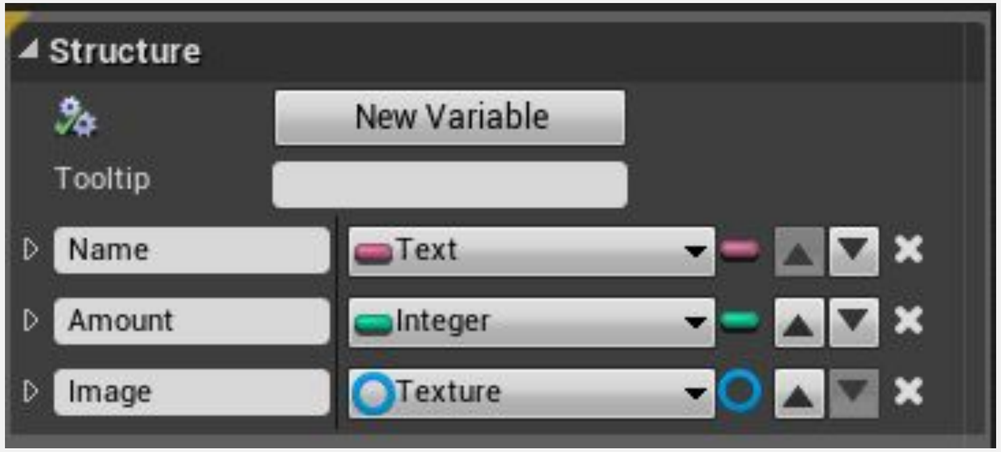
A **structure** (**struct**) can be used to gather in one place several related variables. The variables that belong to a structure can be of different types. You can also have structures that contain other structures and arrays.

To create a new structure, click the green **Add New** button in the **Content Browser**, and in the **Blueprints** submenu select “**Structure**”. Rename it “**Item Struct**”.

Double-click **Item Struct** to edit it.

In the **My Blueprint** panel, click the “**+**” button in the **Variables** category to add variables to the structure.



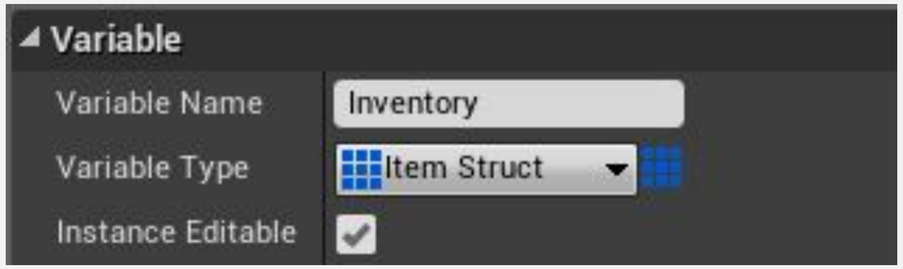


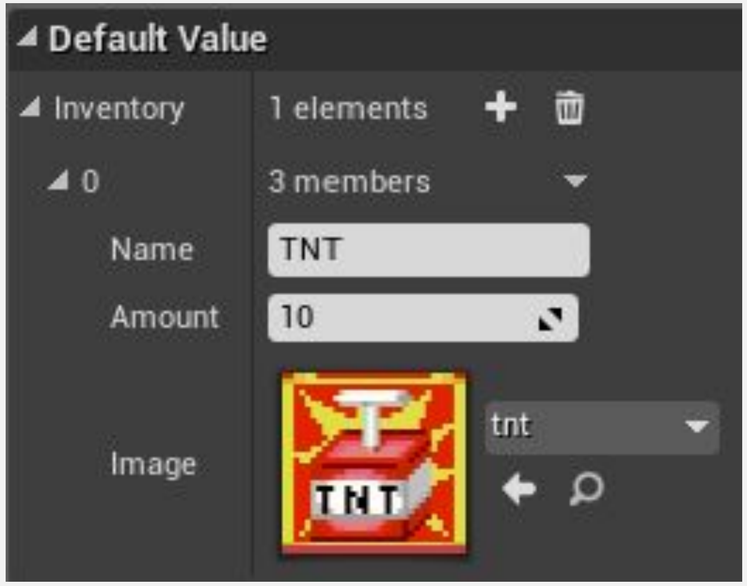
STRUCTURES: EXAMPLE

As an example, create a new variable in a Blueprint that represents the inventory of the player. Name it “**Inventory**” and use the variable type “**Item Struct**”, which was created in the previous slide. Click the icon next to the **Variable Type** drop-down and choose “**Array**”.

Compile the Blueprint.

New elements can be added to the array in the **Default Value** section of the **Details** panel for the **Inventory** variable. Each element will contain the variables defined in the **Item Struct** structure.





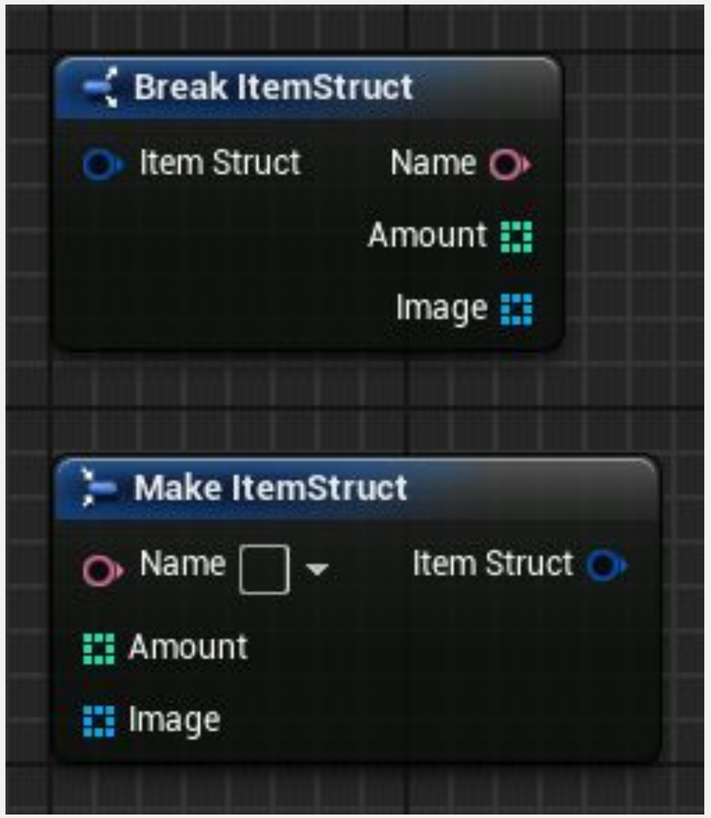
STRUCTURES:
BREAK AND MAKE NODES

When a structure is referenced, the **Break** and **Make** nodes become available for use in the Blueprint.

The **Break** node receives a structure as input and separates its elements.

The **Make** node receives the separate elements as input and creates a new structure.

The image on the right shows the **Break** and **Make** nodes of the **Item Struct** structure.



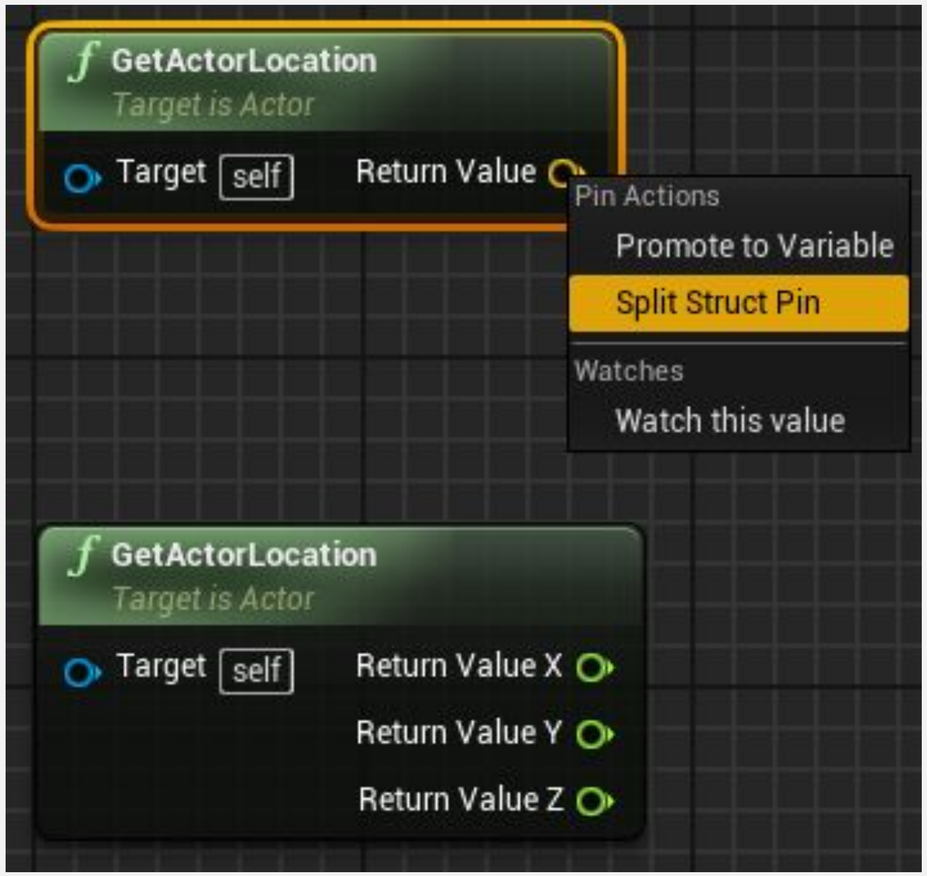
STRUCTURES: SPLIT STRUCT PIN

When a structure is an input or output parameter of a function, its pin can be split to create an output pin for each element of the structure.

To do this, right-click on the struct pin and choose “**Split Struct Pin**”.

For example, a **vector** in Blueprint is a structure containing three float variables with the names “**X**”, “**Y**”, and “**Z**”.

The image on the right shows the **GetActorLocation** function with the struct pin and with pins for each element of the structure.



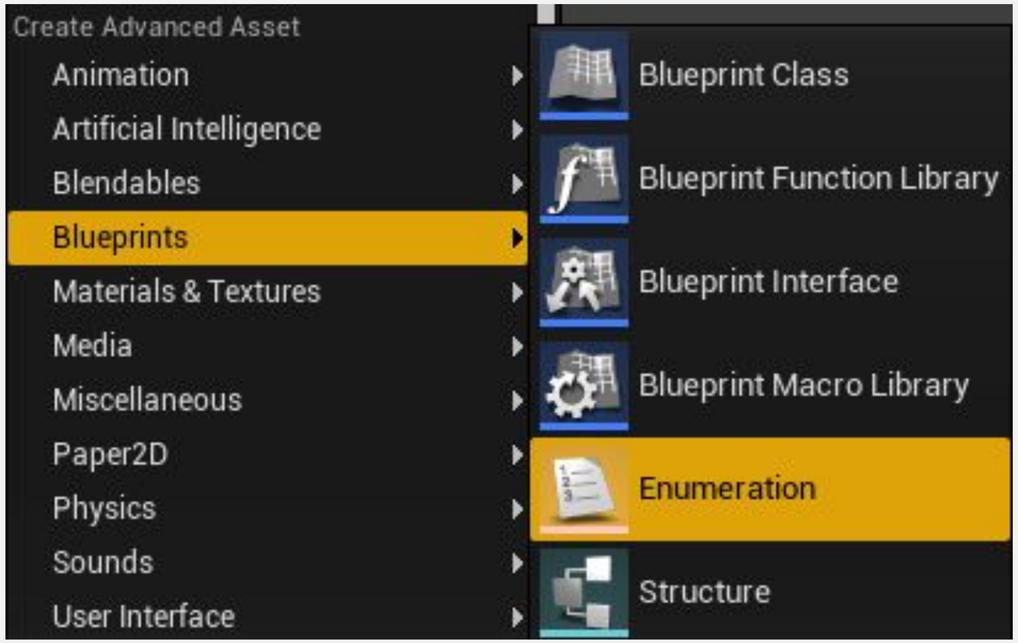
ENUMERATIONS

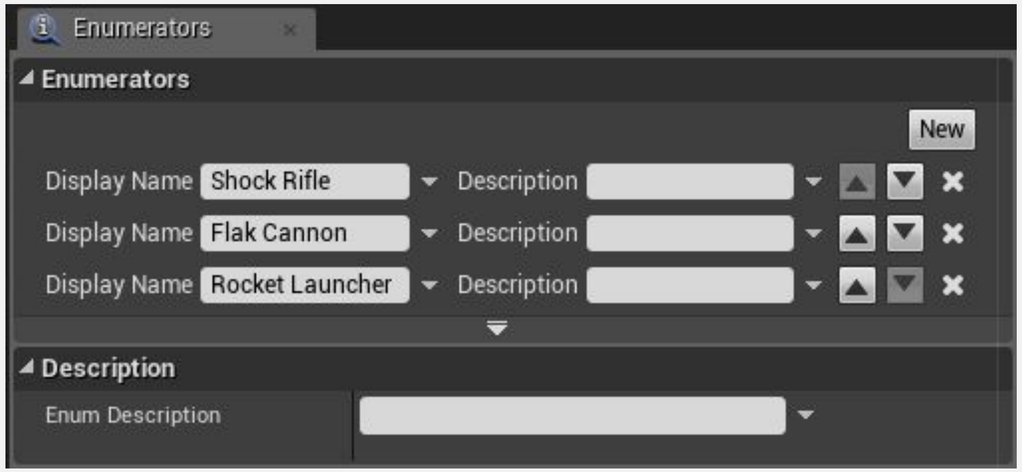
An **enumeration** (**enum**) is a set of constants with meaningful names that specifies all possible values a variable can have.

To create a new enumeration, click the green **Add New** button in the **Content Browser**, and in the **Blueprints** submenu select “**Enumeration**”.

Double-click the enumeration that was created to edit it.

Click the **New** button to add the names that are part of the enumeration.



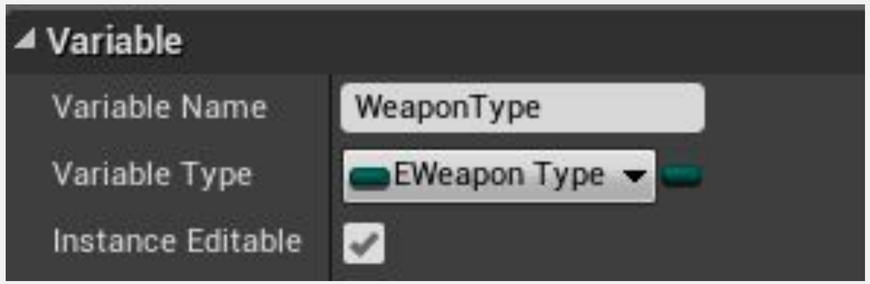


ENUMERATIONS: CREATING A VARIABLE

To create a new variable using an enumeration, just select the name of the enumeration in the **Variable Type** drop-down. The top image on the right shows an enumeration named “**EWeaponType**” being used as the variable type.

Set the **Instance Editable** property to “**true**” so that the value of the enum can be chosen in the Level Editor.

The bottom image shows the values that can be chosen for the enum variable.

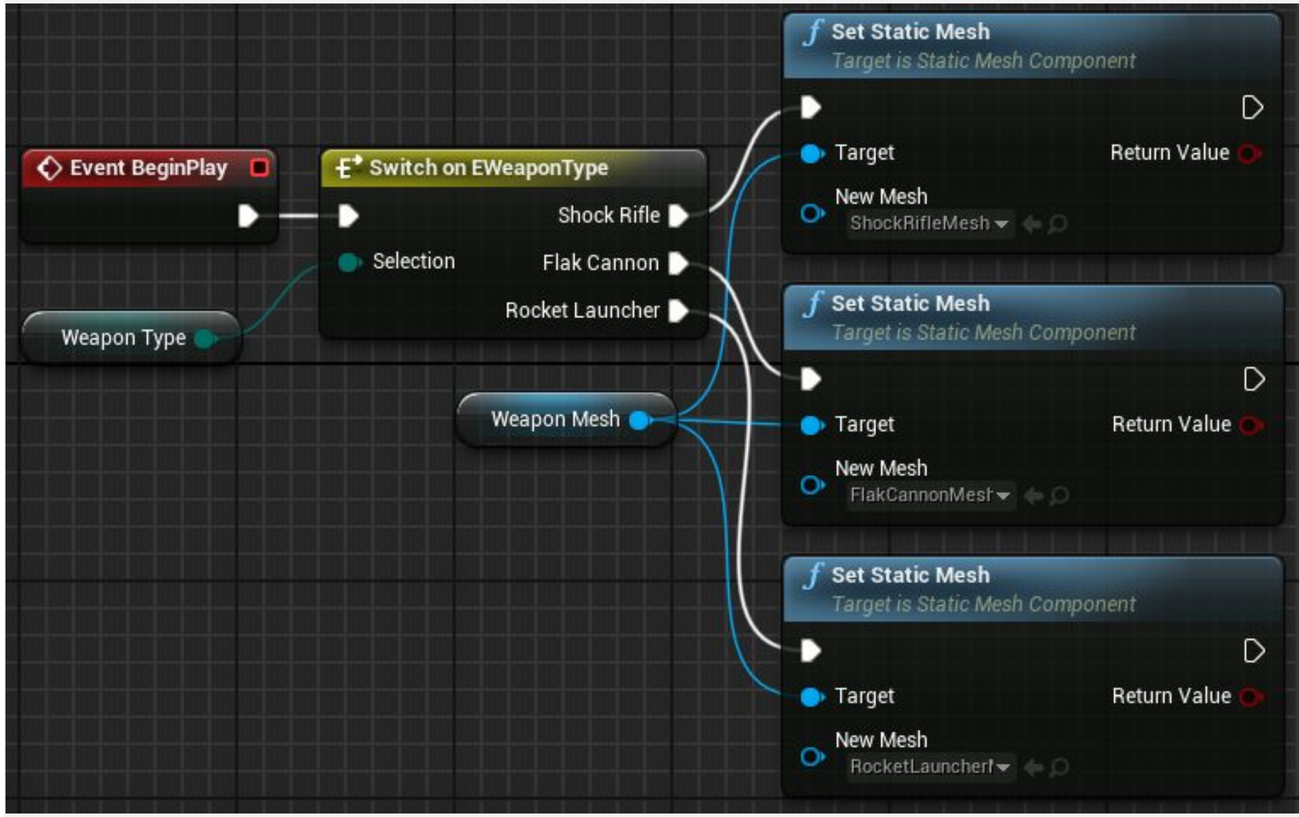


ENUMERATIONS: SWITCH ON

There is a type of **switch** node that determines the flow of execution in accordance with the value of the enumeration.

In the image on the right, “**Weapon Type**” is an enumeration variable and “**Weapon Mesh**” is a Static Mesh component.

The Static Mesh will be set according to the type of weapon.



DATA TABLES

A **data table** can be used to represent a spreadsheet document. This is useful for data-driven gameplay.

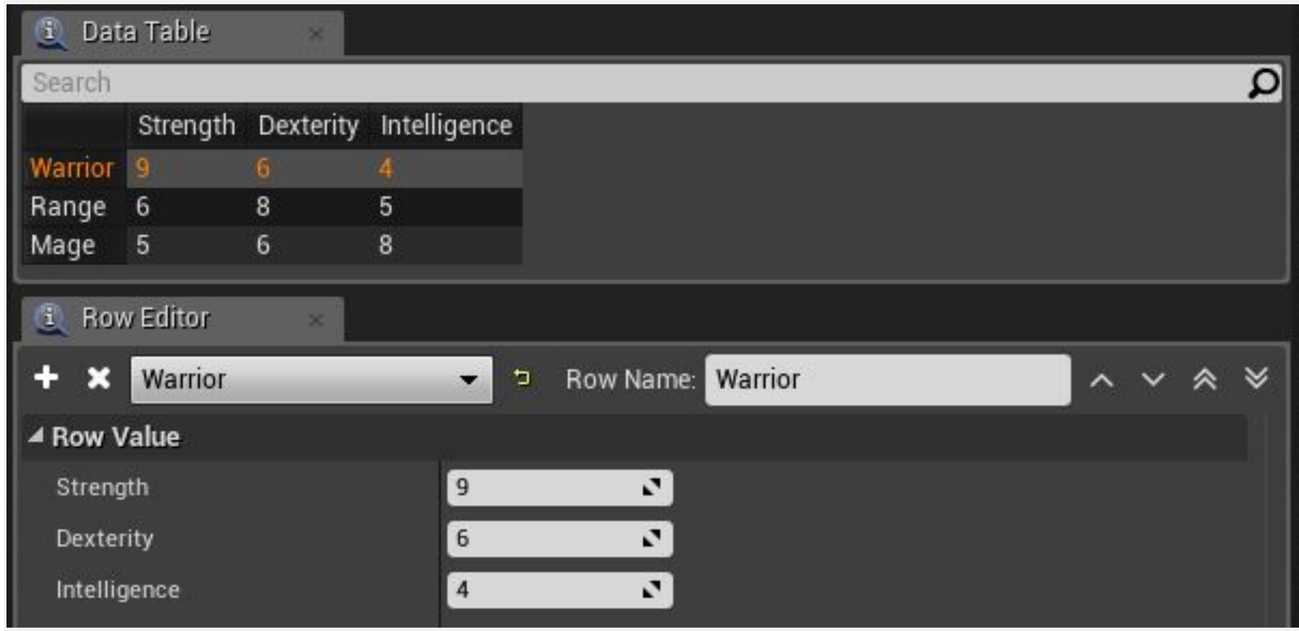
To create a new data table, click the green **Add New** button in the **Content Browser**, and in the **Miscellaneous** submenu select “**Data Table**”.

After creating a data table, it is necessary to choose a structure that will represent the contents of the table (see top image on right).

Once a new data table is created, double-click on it to edit it.

The example on the right is using a structure with the variables **Strength**, **Dexterity**, and **Intelligence**.

This data table has three rows, each one representing an RPG class.

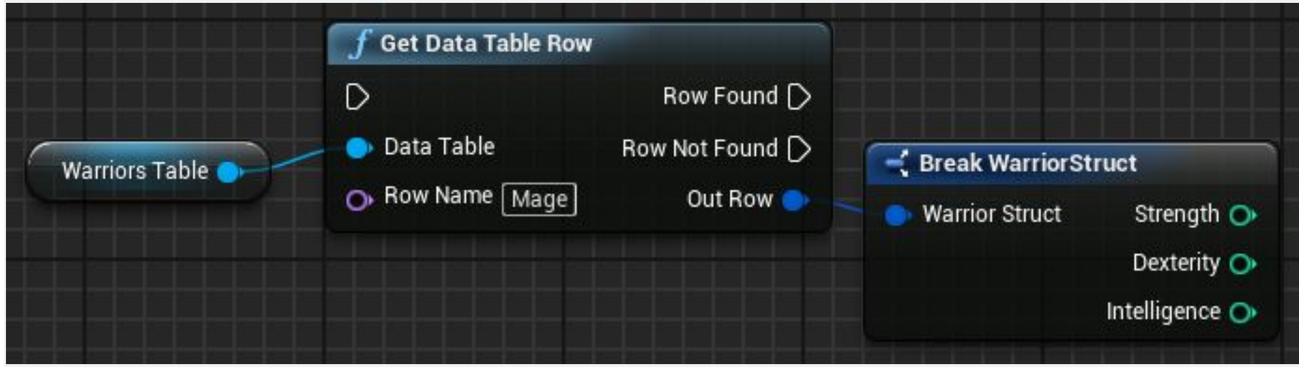


DATA TABLES: ACTIONS

A variable can be created to represent a data table in Blueprint.

The function **Get Data Table Row** uses the **Row Name** parameter to return a line of the table as a structure.

A **Break Struct** node can be used to access the attributes of the struct.



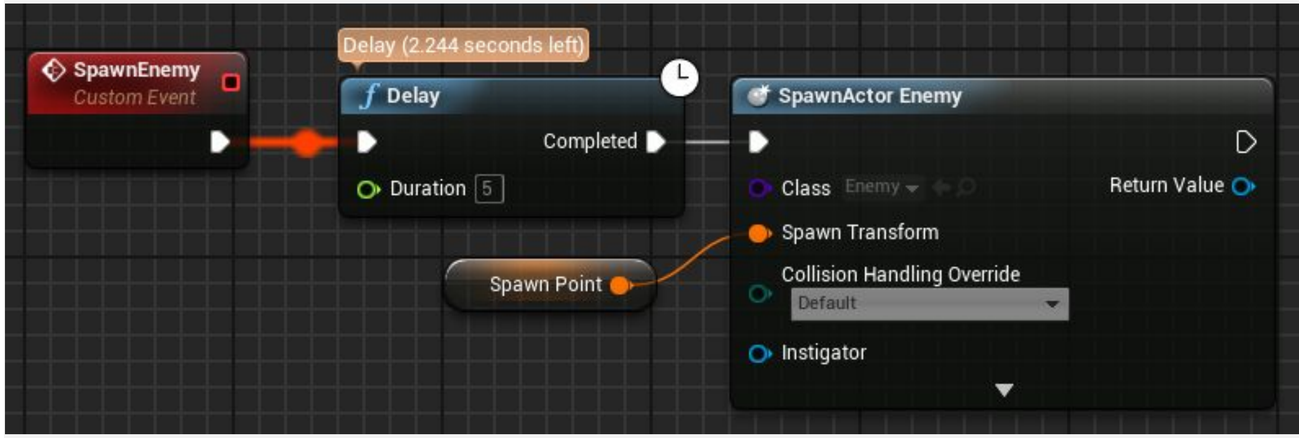
LATENT FUNCTIONS: DELAY

The **Delay** function is a **latent function** that performs the actions connected to the **Completed** pin only after the time specified in the **Duration** parameter has elapsed.

Latent functions do not follow the normal flow of execution of the Blueprints. They run in parallel and can take several ticks until they complete.

The example on the right shows a custom event called “**SpawnEnemy**” that has a **Delay** function to ensure that at least five seconds have elapsed before it spawns a new enemy Actor. Even if the **SpawnEnemy** event is called again in less than five seconds, the **Delay** function does not allow the creation of a new enemy.

Play in the Editor to see the time remaining in a **Delay** function (see example).



LATENT FUNCTIONS: RETRIGGERABLE DELAY

The **Retriggerable Delay** function is a latent function that performs the actions connected to the **Completed** pin only after the time specified in the **Duration** parameter has elapsed.

The difference between the **Retriggerable Delay** and **Delay** functions is that the countdown value for the **Duration** parameter will be reset if the **Retriggerable Delay** function is called again.

LATENT FUNCTIONS: TIMER

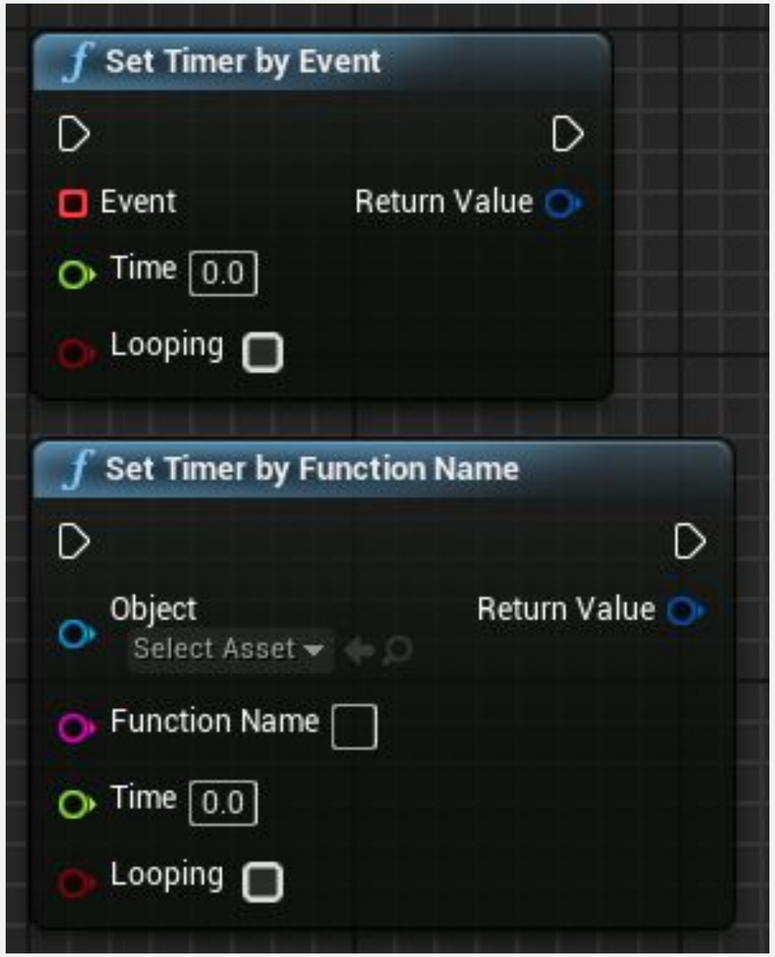
A **Timer** is programmed to perform a given function (or custom event) after a specified time has elapsed.

Two functions define a Timer:

- **Set Timer by Event**: Has as an input parameter a reference to a custom event.
- **Set Timer by Function Name**: Has as input parameters the function name and the Object that contains the function.

Both functions have the following parameters:

- **Time**: Represents the length of the Timer in seconds.
- **Looping**: Indicates whether the Timer will continue its execution or execute only once.



LOADING LEVELS

The **Open Level** function can be used to load Levels in a game.

The value of the **Level Name** parameter can be the folder name plus the name of the Level. If the **Open Level** function does not have the full path, it will try and open the first Level (.umap) it finds with the specified name. If there are multiple .umaps with that same name, it will choose the first one it finds.

This function can be used in a Blueprint that when overlapped by the player will transport them to another Level.

It can also be used in a game's Start menu, where different Levels can be loaded based on the chosen option.

QUITTING THE GAME

The **Quit Game** function can be used to exit the game or move the application to the background.

This function contains the following input parameters:

- **Specific Player**: Represents the player that will exit the game. “**Player 0**” is the default value.
- **Quit Preference**: Represents how the player wants to exit the game. The value can be “**Quit**” or “**Background**”.

SAVING AND LOADING DATA

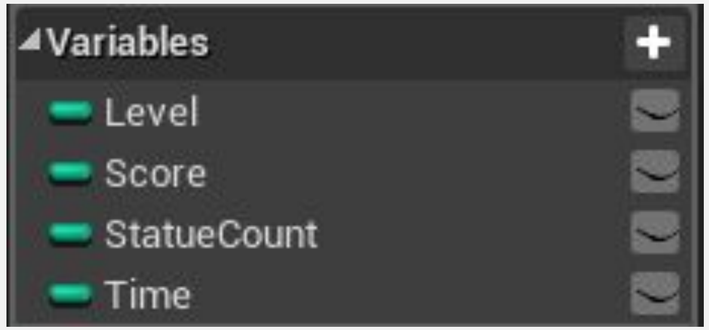
SAVE GAME BLUEPRINT

To save a game, the information that will be stored needs to be gathered in variables of a Blueprint of type “**SaveGame**”.

The first step to saving and loading games with Blueprints is to create a new Blueprint using “**SaveGame**” as its parent class.

The next step is to create variables in the Save Game Blueprint to keep the information that will be saved.

The bottom image on the right shows an example based on the game created in Lecture 5.

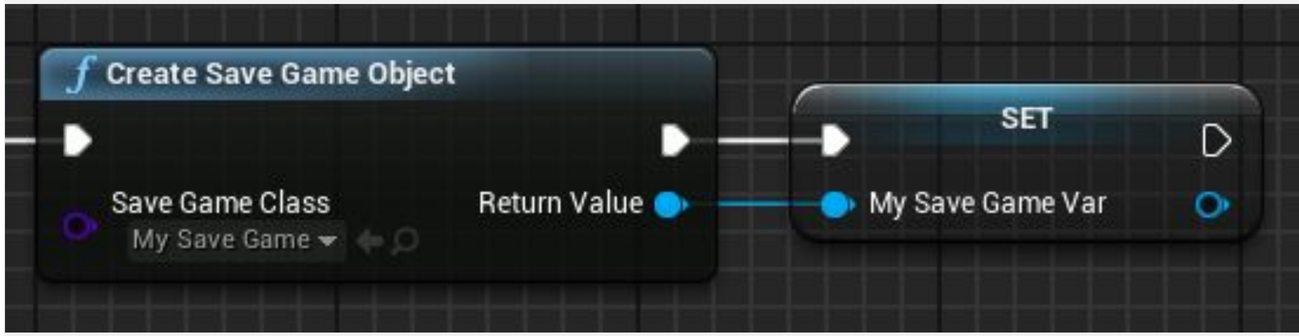


CREATE SAVE GAME OBJECT

The **Create Save Game Object** function creates a Save Game Object based on a Save Game class.

The information to be saved will be set in the variables of this Object.

MySaveGame is a Blueprint of the Save Game class.

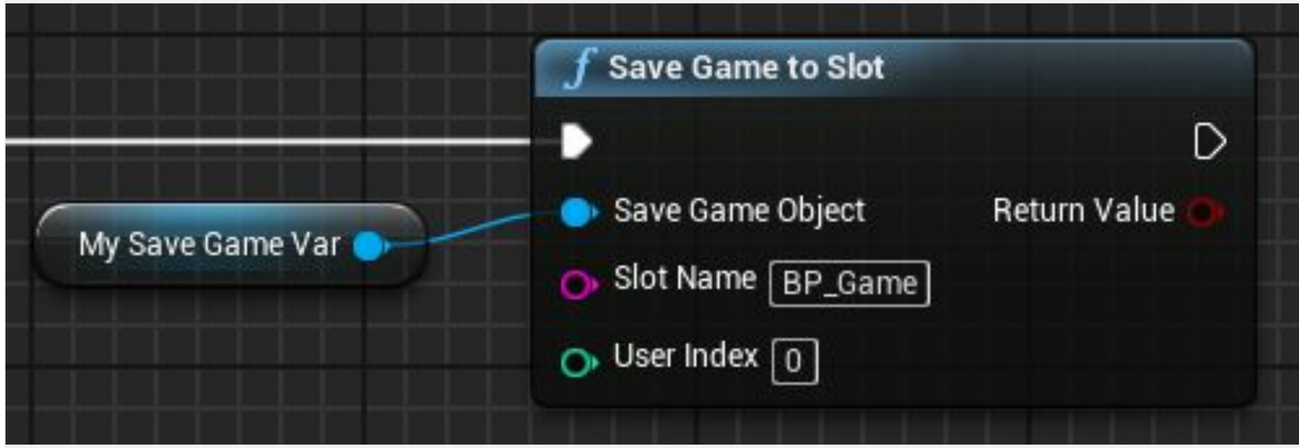


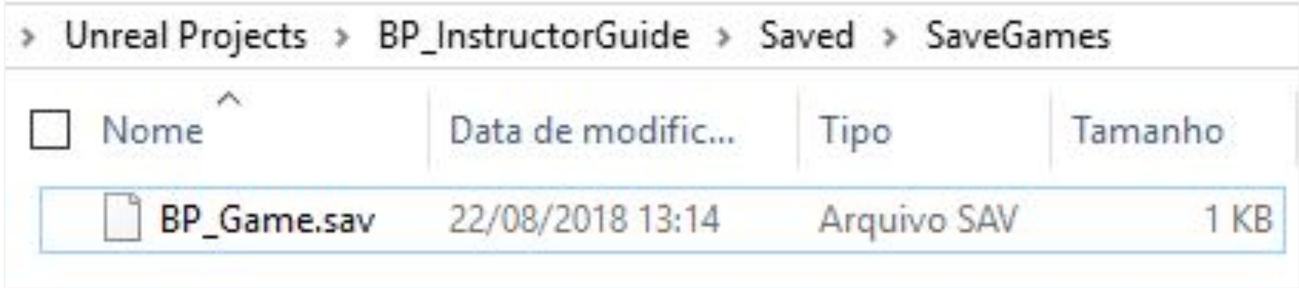
SAVE GAME TO SLOT

The **Save Game to Slot** function saves the contents of a Save Game Object to a file on the disk.

The image on the right shows the **Slot Name** parameter with the value “**BP_Game**”. This means that a file with the name “**BP_Game.sav**” will be created with the information that needs to be saved.

The “**BP_Game.sav**” file is stored in the folder “**ProjectName > Saved > SaveGames**”.

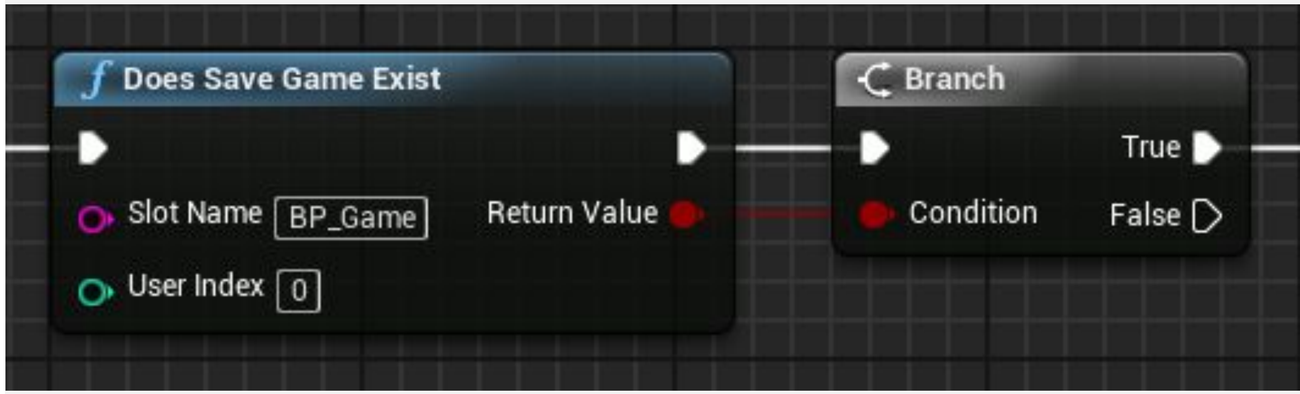




DOES SAVE GAME EXIST

The **Does Save Game Exist** function checks to see if a Save Game exists in the given **Slot Name** parameter.

This is a useful test to do before attempting to load a Save Game.

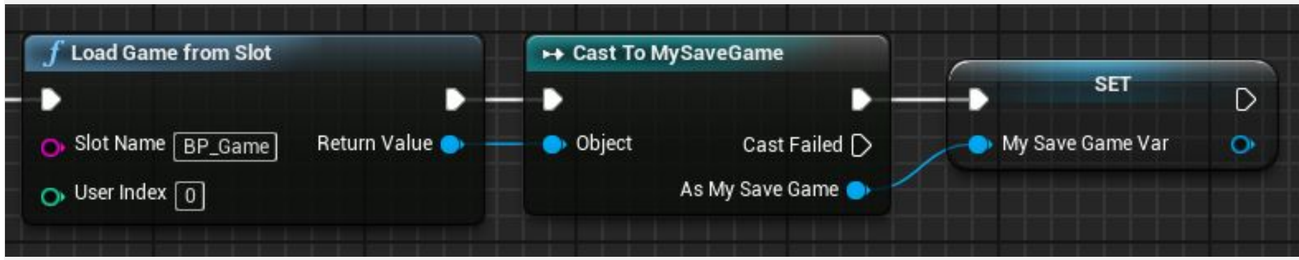


LOAD GAME FROM SLOT

The **Load Game from Slot** function loads the contents of a slot and creates a Save Game Object with the contents.

The output parameter **Return Value** is a reference to a generic Save Game Object, so it is necessary to cast the reference to the correct Save Game Blueprint in order to access the variables.

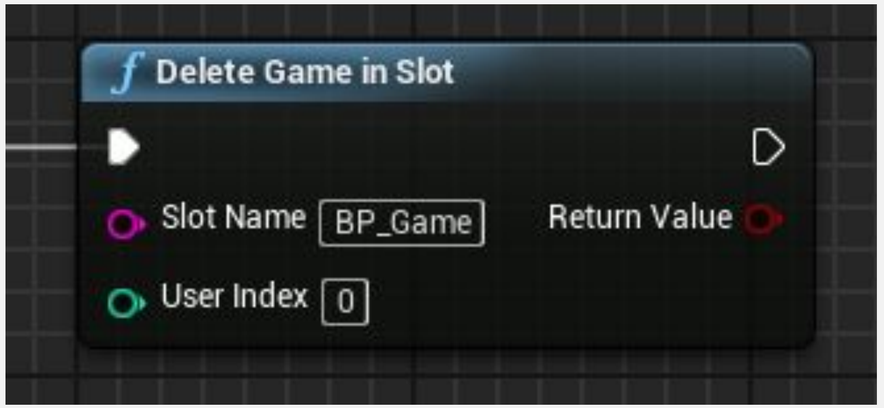
In the example on the right, the **Load Game from Slot** function casts to **MySaveGame**. After that, the reference to the **MySaveGame** Object is set in a variable.



DELETE GAME IN SLOT

The **Delete Game in Slot** function is used to delete Save Game data in a slot.

The **Return Value** parameter is “**true**” if the file was found and deleted.



SUMMARY

This lecture introduced many advanced Blueprint concepts and presented structures, arrays, sets, maps, enumerations, latent functions, and data tables.

The functions necessary to implement a basic save/load system were explained.