Week 6: Objects and C# Classes (Part 2)

Welcome to Week 6!

Presentation Slides

Worksheet (Document)

Worksheet (PDF)

Table of Contents

6.1 – Prefabs and How to Use Them

6.1.1 – GameObject Fields

```
public class FieldExample : MonoBehaviour
{
    public GameObject objectInScene;
    public GameObject prefabObject;
}
```

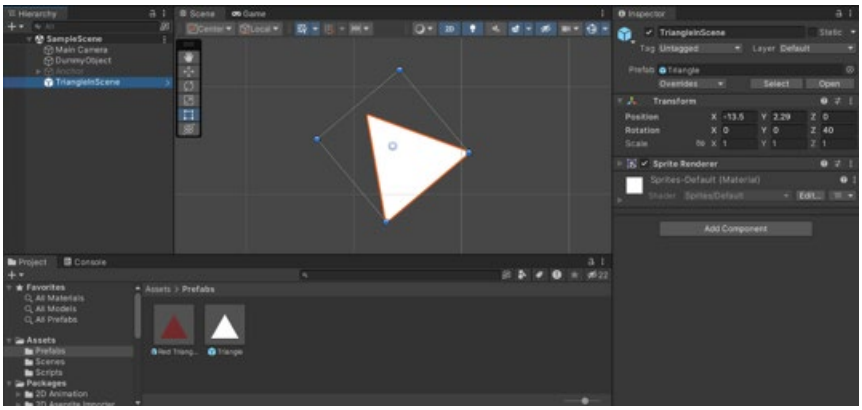
Consider this component to the right.

Its fields have **GameObject** as their data type. When viewed in the *Inspector* window of the Unity editor, such fields have prompts that you can click-and-drop objects in the scene onto.

6.1.2 – Prefabs

Using **GameObject** fields to reference objects in the scene is useful to reference data on other objects. However, this can become cumbersome in scenarios like creating projectiles for a weapon or spawning monsters; both would require having an object in the scene to reference as a "template" of sorts, and that's usually not ideal.

To resolve this problem, you can use a *prefab*.
A *prefab* is an asset which represents an object's hierarchy. To create a prefab, click-and-drop an object in the scene onto the Assets folder in the *Project* window.



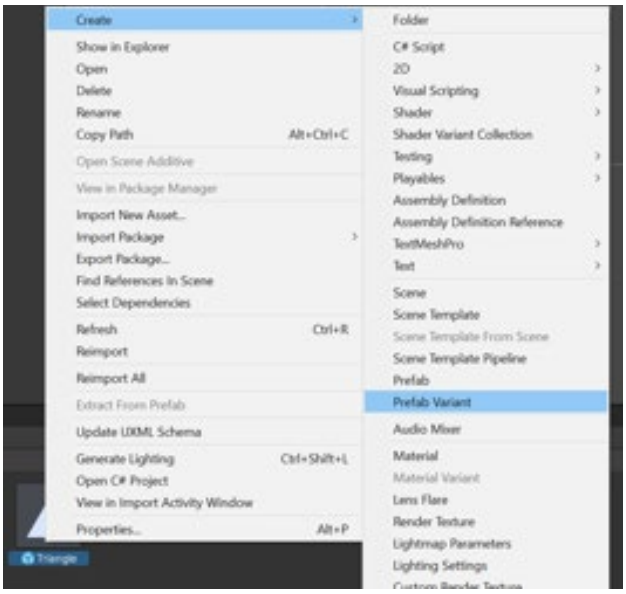
After creating the prefab, you can assign an instance of it placed in the scene (**TriangleInScene**) or the asset itself (**Triangle**) to a **GameObject** field.

6.1.3 – Prefab Variants

Prefab instances placed in the scene can be modified without affecting the original asset.

This is fine if only a few instances of the prefab are being used, but consider the following: the project has a prefab with a Square object in it, and you want to place several instances which are red, green and blue. Later down the line, you decide to make blue Squares fall faster than red and green Squares.

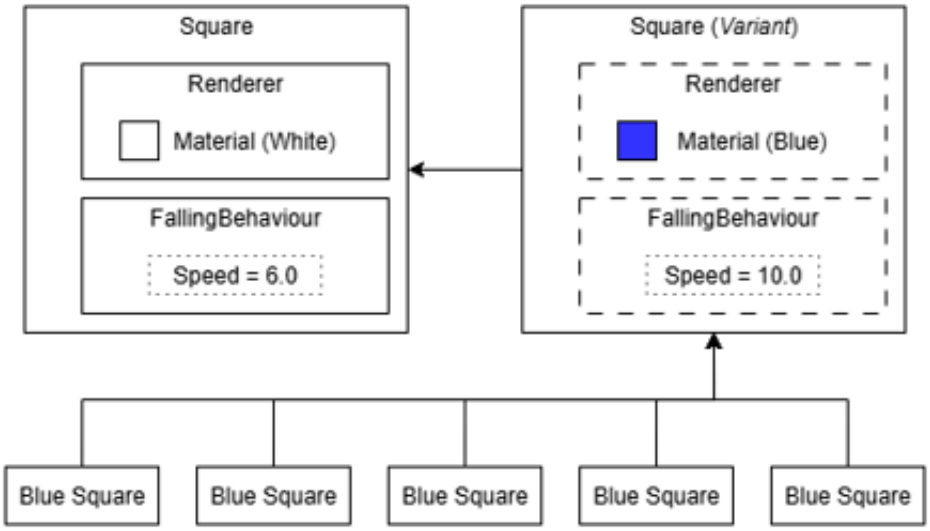
Instead of manually updating each blue Square in the scene, you could have instead made blue Squares a *variant* of the original prefab.



A *prefab variant* is essentially a copy of an existing prefab. Modifying the variant won't affect the original prefab, but now you only have to modify the prefab variant asset for blue Squares instead of every blue Square object in the scene!

To create a prefab variant, right click the prefab in the Assets folder, hover over the **Create** option and then click the **Prefab Variant** option.

An example of how objects in a scene reference a prefab variant and how the prefab variant references the original prefab is shown below.



6.2 – Object, GameObject and Their Methods

6.2.1 – Object Inheritance

In your C# scripts, you may have noticed that the definition for the **class** is like the following:

public class DerivingClass : BaseClass

This syntax indicates that **DerivingClass** *inherits* its fields and methods from **BaseClass** .
For this course, all you need to know regarding inheritance is that variables whose data type is **BaseClass** can have **DerivingClass** objects assigned to them.

Inheritance in Unity

All components in Unity inherit from **MonoBehaviour** .
For **MonoBehaviour** , this allows components to implement definitions for **Start** , **Update** , etc.

Keep in mind that trying to do the same in a **class** that does not inherit from **MonoBehaviour** will instead "hide" the method from the base **class** .

This course will not cover the standard rules of inheritance for brevity and a lack of relevance in Unity.

6.2.2 – Handling Objects in Code

Object and GameObject have static[1] methods for controlling the lifespan of objects in the scene.

- **Object.Instantiate(Object obj)** – spawns a copy of **obj** and its components
- **Object.Destroy(Object obj)** – Permanently removes **obj** from the scene
- **Object.Destroy(Object obj, float t)** – Permanently removes **obj** from the scene after **t** seconds
- **GameObject.Find(string name)** – Searches the scene for an object whose name is the same as **name**

GameObject inherits from **Object** , so a **GameObject** instance can be provided to the **Object** methods. This is how you would, for example, use a **GameObject** field set to a prefab asset to spawn projectiles from a weapon or spawn monsters.

Object vs Object

A data type with the name **Object** is found in both the **UnityEngine** and **System** namespaces. In both this course and most Unity-related information, **Object** refers to the data type in the **UnityEngine** namespace.

---

[1] ***Static*** **methods are called on the data type rather than an object instance.**

6.3 – Booleans and Code Flow

Refresh: Booleans
Booleans are value-type objects representing a value with two states.
Consider the value to be things like the answer to a "Yes or No" question or a light switch that can be turned on and off.

The **bool** data type is used to represent booleans, and its values are **true** and **false** .

6.3.1 – Boolean Expressions

C# contains several symbols (*operators*) for comparing objects, and the result of the comparisons is a **bool** object.

- **objA == objB** – Object equality[2]
- **objA != objB** – Object inequality (**true** when **==** would be **false** , and vice versa)
- **objA < objB** – **true** if **objA** is less than **objB**
- **objA <= objB** – **true** if **objA** is less than or equal to **objB**
- **objA > objB** – **true** if **objA** is greater than **objB**
- **objA >= objB** – **true** if **objA** is greater than or equal to **objB**

C# also contains operators for combining and manipulating boolean objects.

- **boolA && boolB** – **true** when both **boolA** and **boolB** are **true**
- **boolA || boolB** – **true** when either **boolA** or **boolB** is **true**
- **!boolA** – **true** when **boolA** is **false** , and vice versa

Unity Booleans

In Unity, **GameObject** and **MonoBehaviour** objects can be treated as booleans and will be equal to **true** if they exist in the scene.

---

[2] By default, value-type objects are equal when their values are equal whereas reference-type objects are equal when their *references* point to the same object.

6.3.2 – Controlling Code Flow

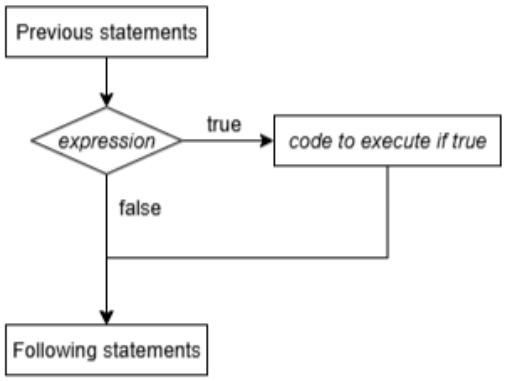
```
control word ( expressions )
{
    contents
}
```

A *code flow statement* is a series of boolean expressions and symbols that affects what code is executed. The syntax for code flow statements generally follows the snippet to the right.

- **control word** – defines how the execution of the code is handled
- **expressions** – the requirements for the code flow
- **contents** – the code to execute when the requirements are met

The **if** code flow statement will execute its contents when the boolean expression is **true** .

```
if ( expression )
{
    code to execute if true
}
```



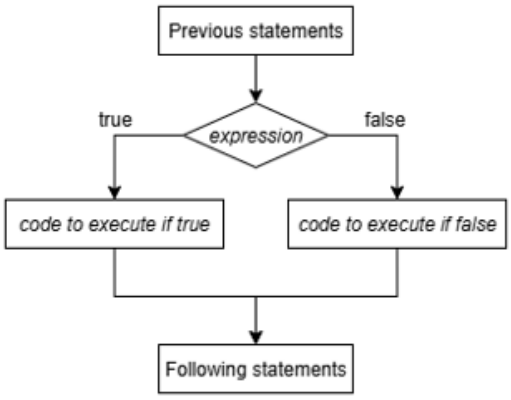
Checking If Objects and Components Exist

Since **GameObject** and **MonoBehaviour** can be treated as booleans, they can also be used as the expression for code flow statements.
A common usage of this behavior is to check if components exist on a **GameObject** .

```
OtherComponent component = GetComponent<OtherComponent>( );

if (component)
{
    // This GameObject had the other component
    this.otherComponent = component;
}
```

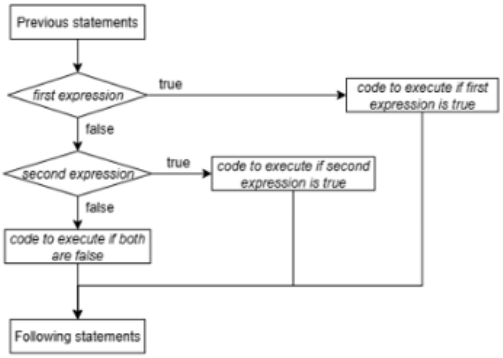
The **if/else** code flow statement has two sets of contents. The first set is executed when the boolean expression is **true** , and the second set is executed when the boolean expression is **false** .

```
if ( expression )
{
    code to execute if true
}
else
{
    code to execute if false
}
```



The **if/else if/else** code flow statement has two or more sets of contents. If the first boolean expression is **true** , the first set is executed. Otherwise, if the second boolean expression is **true** , the second set is executed. This repeats for every boolean expression and set. If none of the boolean expressions were **true** , the final set is executed[3].

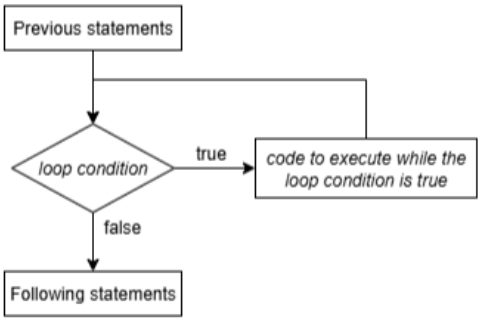
```
if ( first expression )
{
    code to execute if first expression is true
}
else if ( second expression )
{
    code to execute if second expression is true
}
else
{
    code to execute if both are false
}
```



When code flow ends up at a statement it had previously executed, it is considered to have been *looped*. Hence, a *code flow looping statement* is a code flow statement which causes the code flow to loop.

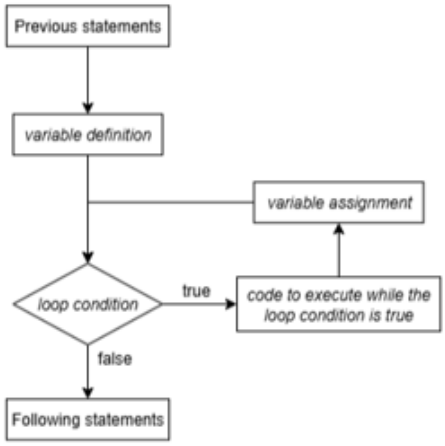
The **while** code flow looping statement will execute its contents as long as its boolean expression, called a *loop condition*, is **true** .

```
while ( loop condition )
{
    code to execute while the loop condition is true
}
```



The **for** code flow looping statement is a more complex version of the **while** statement. It contains both an expression for defining a variable to use within its contents and an expression for modifying the variable.

```
for ( variable definition ; loop condition ; variable assignment )
{
    code to execute while the loop condition is true
}
```




```
int[ ] array = new int[ ]
{
    1, 2, 3, 5, 7, 11, 13, 17, 19, 23
};

for (int i = 0; i < array.Length; i = i + 1)
{
    Debug.Log(array[i]);
}
```

The **for** statement is usually used to iterate over the objects in a collection object. For example: sending each object in an array to the Unity Console.

Variable Scopes

It was explained in Week 5 that local variables (*locals*) only exist within the body of the method which creates them. This is technically correct, but it fails to cover the whole story.

For this course, just know that a *scope* refers to the region of code where a local variable exists. Both methods and code flow statements define scopes for locals.

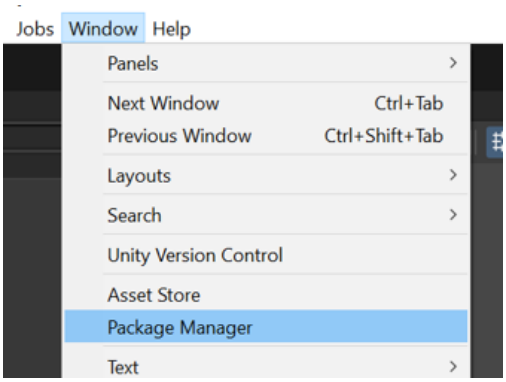
The **if/else** and **if/else if/else** statements define a scope for each **if** , **else if** and **else** section.

The **for** statement's variable definition expression is in the same scope as its contents.

[3] **The else section can be omitted. In that case, no code is executed if none of the boolean expressions were true .**

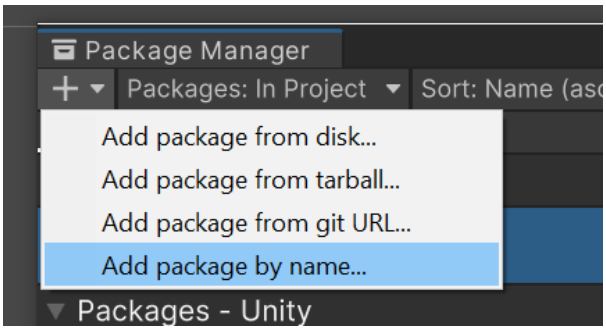
6.4 – Responding to user Input

6.4.1 – Installation and Initial Setup

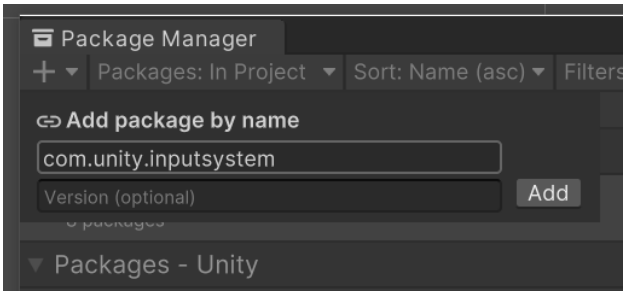


The *Input System* package from the Unity Package Manager will be used to listen for user inputs. The documentation can be found in [this Unity site](#)[4].

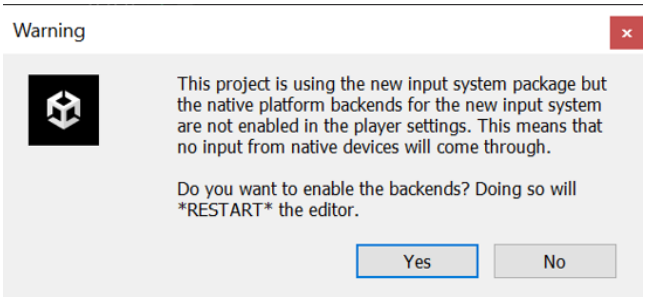
To install the package, you first need to open the *Package Manager* window from the *Window* tab in the Unity editor.



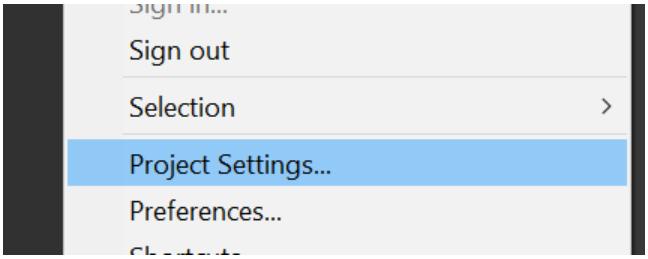
Click the  button, then select the **Add package by name…** option.



Enter ***com.unity.inputsystem*** as the name of the package and click the Add button.

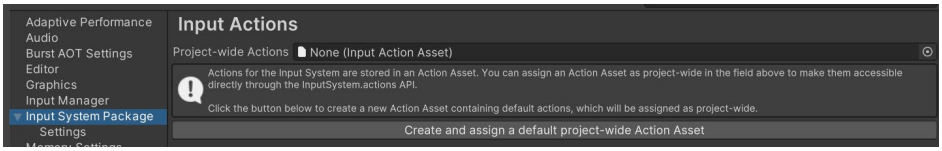


Unity will now download the *Input System* package. After that has finished, you will be asked to restart the Unity editor. Click **Yes** to finish the installation of the package.

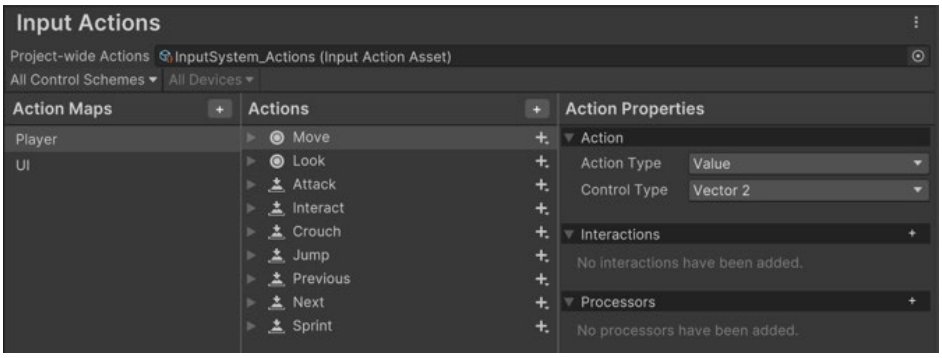


Once your Unity project has reopened, you will need to set up the default *actions* in the *Project Settings* window. You can find an option for it in the *Edit* tab.

In the *Project Settings* window, select the *Input System Package* option.
It should look like the following image.



Click the large **Create and assign a default project-wide Action Asset** button to generate the default actions for the project.
The *Project Settings* window should then update to the following image.



Feel free to open the dropdown menus for each action to see what controls are assigned to them. For example, the **Move** action has the WASD keys as one of its controls.

---

[4] https://docs.unity3d.com/Packages/com.unity.inputsystem@1.14/manual/index.html

---

6.4.2 – InputAction

The *Input System* package exposes its actions through the **InputAction** class in the **UnityEngine.InputSystem** namespace.

An **InputAction** object will listen for various input types, such as keyboard keys, controller buttons, controller joysticks and more.

For this course, you only need to know of the "single key" input type: 

A Brief Rundown on Namespaces

You may have noticed that every C# script file starts with the following lines.

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

For example, **MonoBehaviour** is located in the **UnityEngine** namespace.

To use the data types from the *Input System* package, you will need to add a statement for its **UnityEngine.InputSystem** namespace.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

InputAction jumpAction;

void Start( )
{
    jumpAction = InputSystem.actions.FindAction("Jump");
}

void Update( )
{
    if (jumpAction.WasPressedThisFrame( ))
    {
        Jump( );
    }
}
```

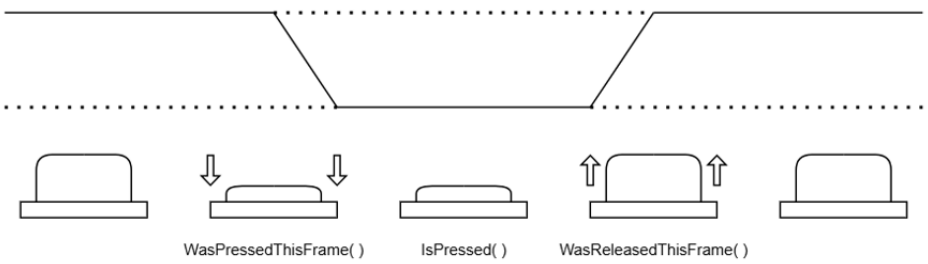
6.4.3 – Reading Input

InputAction objects can be obtained from the **actions** field in the **InputSystem class** .

The snippet to the right showcases how you'd use that field to find an **InputAction** , then using that **InputAction** to listen for inputs.

For this course, you only need to use the following three methods when listening for input:

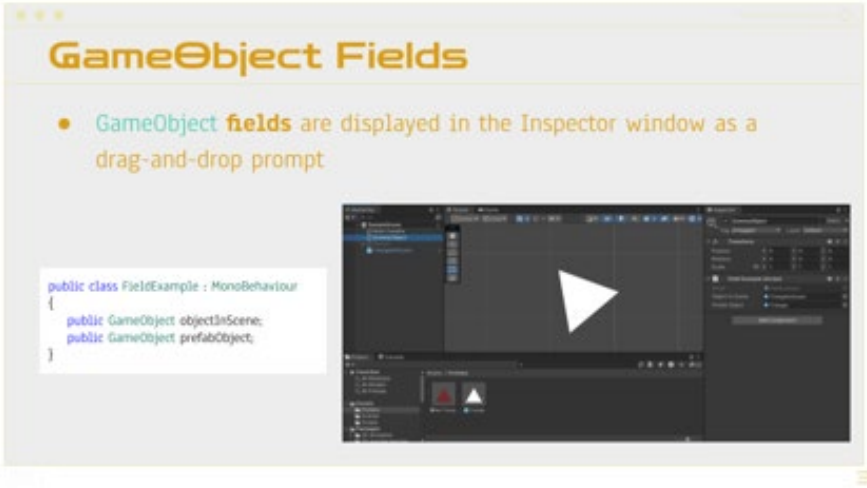
- **WasPressedThisFrame( )** – Returns **true** on the first frame when the control is being pressed
- **IsPressed( )** – Returns **true** on every frame while the control is being held down
- **WasReleasedThisFrame( )** – Returns **true** on the first frame after the control is no longer being pressed



The visual to the right indicates when each method would return true during a button press.

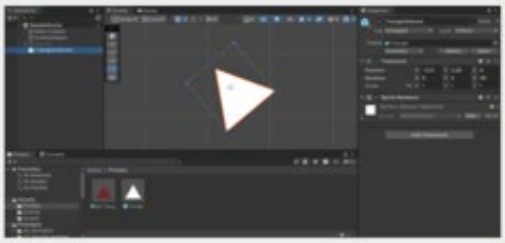
6.5 – Slides

Note: The colors for text and certain elements in the following images of the slides have been modified to produce lighter colors when printing with the *Black and white* Color setting and do not reflect the colors used in the actual slides.



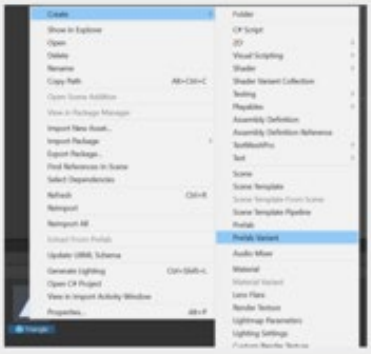
Prefabs

- GameObjects as **assets**
- Commonly used for "template" or "base" instances for GameObjects
- Modifying placed prefabs in the **scene** will not affect the **asset**



Prefab Variants

- An **asset** representing a copy of a prefab
- Modifying the original prefab will affect all variants



Detour: Data Type Inheritance

- `class DerivingClass : BaseClass` declares that `DerivingClass` inherits from `BaseClass`
- All components in Unity inherit from `MonoBehaviour`
- For this course, know that:
  - `BaseClass` variables can be assigned a `DerivingClass` object
  - `DerivingClass` contains the same methods as `BaseClass`

Questions?

Refresh: Booleans

- **Value-type** objects representing a value with two states
  - "True / False"
  - "Yes / No"
  - "On / Off"
- `bool`
  - Keywords: `true` and `false`

Object/GameObject Methods

The UnityEngine.Object and GameObject types contain various methods for controlling object lifespans

- Object.Instantiate(Object obj) – Creates a copy of obj and its **components**
- Object.Destroy(Object obj) – Permanently removes obj from the **scene**
- Object.Destroy(Object obj, float t) – Permanently removes obj from the **scene** after t seconds
- GameObject.Find(string name) – Searches the **scene** for an object whose name is the same as name

GameObject inherits from UnityEngine.Object

Boolean Logic

C# contains many symbols (**operators**) for comparing objects. The result of the comparisons are **bool** objects.

- `objA == objB` – Object equality
  - For **value-type** objects, **true** when their values are equal.
  - For **reference-type** objects, **true** when their *references* point to the same object.
- `objA != objB` – Object inequality
  - **false** when `==` would be **true**, and vice versa

Boolean Logic

C# contains many symbols (**operators**) for comparing objects. The result of the comparisons are **bool** objects.

- `objA < objB` – **true** if objA is less than objB
- `objA <= objB` – **true** if objA is less than or equal to objB
- `objA > objB` – **true** if objA is greater than objB
- `objA >= objB` – **true** if objA is greater than or equal to objB

Boolean Logic

Operators for combining and modifying boolean objects also exist.

- `boolA && boolB` – **true** when both boolA and boolB are **true**
- `boolA || boolB` – **true** when either boolA or boolB is **true**
- `!boolA` – **true** when boolA is **false**

In Unity, GameObject and MonoBehaviour objects can be treated as a boolean and will be **true** if they exist in the **scene**.

Code Flow

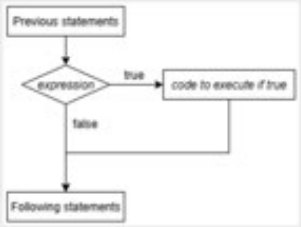
Code flow statements generally follow this syntax:

46

Code Flow

```
if ( expression )
{
    code to execute if true
}
```

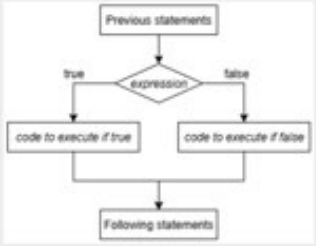
```
OtherComponent component = GetComponent<OtherComponent>();

if (component)
{
    // This GameObject had the other component
    this.otherComponent = component;
}
```



47

Code Flow

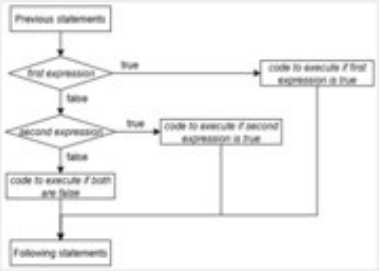
```
if ( expression )
{
    code to execute if true
}
else
{
    code to execute if false
}
```



48

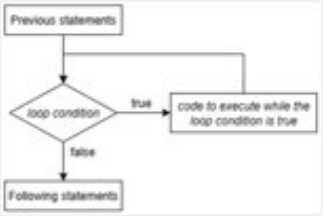
Code Flow

```
if ( first expression )
{
    code to execute if first expression is true
}
else if ( second expression )
{
    code to execute if second expression is true
}
else
{
    code to execute if both are false
}
```



Code Flow

```
while ( loop condition )
{
    code to execute while the loop condition is true
}
```



Code Flow

```
for ( variable definition ; loop condition ; variable assignment )
{
    code to execute while the loop condition is true
}
```

```
int[ ] array = new int[ ]
{
    1, 2, 3, 5, 7, 11, 13, 17, 19, 23
};

for (int i = 0; i < array.Length; i = i + 1)
{
    Debug.Log(array[i]);
}
```

Questions?

Handling User Input

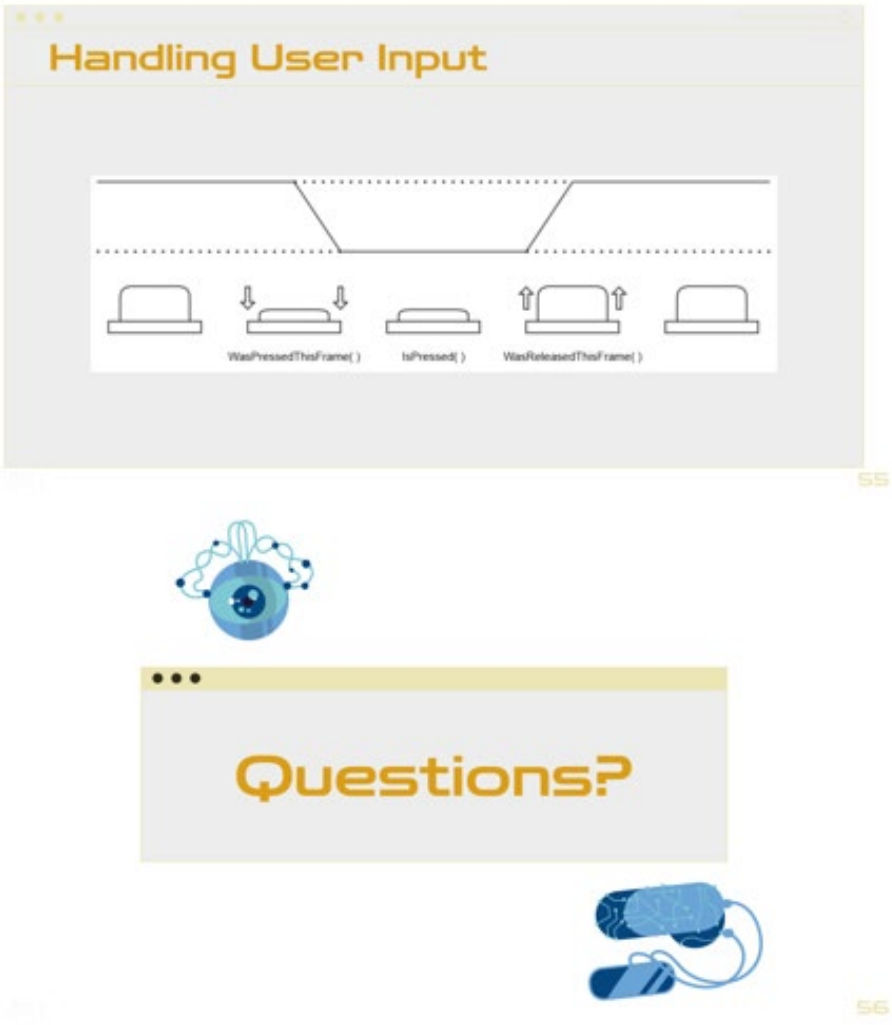
- *Input System* package in the Unity Package Manager
- Supports keyboard keys, controller buttons, controller joysticks and more
- Handled via `InputAction` objects

```
InputAction jumpAction;

void Start()
{
    jumpAction = InputSystem.actions.FindAction("Jump");
}

void Update()
{
    if (jumpAction.wasPressedThisFrame())
    {
        Jump();
    }
}
```


Handling User Input

`InputAction` provides methods used to listen for keyboard key and controller button actions.

- `bool WasPressedThisFrame()` – `true` for one frame when the key/button is pressed
- `bool IsPressed()` – `true` on every frame while the key/button is being held down
- `bool WasReleasedThisFrame()` – `true` for one frame when the key/button is no longer pressed



6.6 – Exercises

Reinforcement

The reinforcement section is used to check the understanding of the information that was presented in the PowerPoint/document. Please answer the questions in your own words.

R — 6.1 What is a prefab?

R — 6.2 What is a prefab variant?

R — 6.3 What does it mean when a `class` inherits from another `class`?

R — 6.4 Which methods are used to handle objects in a scene, and what is their purpose?

R — 6.5 What is a boolean?

R — 6.6 How does object equality differ between reference-type and value-type objects?

R — 6.7 What is a code flow statement?

R — 6.8 What do namespaces control?

R — 6.8 What data type is used to listen for inputs?

Project

The project section's tasks are used in work toward creating the final Flying Bird game project for the end of the course.

P — 6.1 Create a script with a `GameObject` field, assign a prefab asset to it in the Unity editor and then spawn an object using the field.

P — 6.2 Create a script which destroys an object in the scene.

P — 6.3 Create a script that sends a message to the Unity console when the spacebar is pressed.