Scripting Walkthrough - Part 3

With 2D Game Kit

For the purpose of this tutorial, it is presumed that you have the 2D Game Kit set up and you have some basic knowledge of working with Unity from the previous tutorials.

Unit 3: Power-up

What if we wanted to place an item into the game world, which would make a superhero out of our player? How could we do that?

Object representing a power-up item

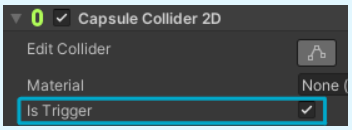
⭐**Task**: Create an object representing a power-up item at a suitable place in the scene.

Proceed similarly to creating a bullet in Part 1.
For now, you can use sprite from Assets/2DGamekit/Art/Sprites/AnimatedSprites/Key/.
Hint:

Component for picking the item up

The player should pick the item up if they move over it.
First we would like to detect when the player overlaps with the item.

Trigger collider (collider with `IsTrigger` checked) functions as a trigger.



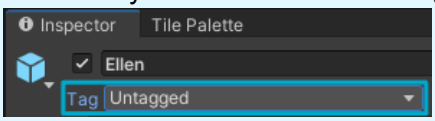
Player can go through the object and it will not block the movement.
`OnTriggerEnter2D(Collider2D collision)` - Called on the start of overlap.
`OnTriggerExit2D(Collider2D collision)` - Called on the end of overlap.

Unity - Scripting API: Collider2D.isTrigger (unity3d.com)
Unity - Scripting API: Collider2D.OnTriggerEnter2D(Collider2D) (unity3d.com)

⭐**Task**: Setup the object as a trigger so that it is able to detect overlaps.

Hint:
Then we need to make sure it is the player what overlaps.

We can use *Tags* to label objects and then find them according to it.
Creating a tag: Select an object in the Hierarchy → drop down menu Tag in the Inspector (on top) → Add Tag… → click on the symbol + → name the tag.



Assigning a tag: Select an object → drop down menu Tag in the Inspector → select.
GameObject.CompareTag(string tag) - Method to find out if the object has the given tag.

Unity - Manual: Tags (unity3d.com)
Unity - Manual: Tags and Layers (unity3d.com)

Unity - Scripting API: GameObject.CompareTag (unity3d.com)

⭐**Task**: Assign the player a Player tag and then recognize him in the code based on that.

Hint:
Solution:

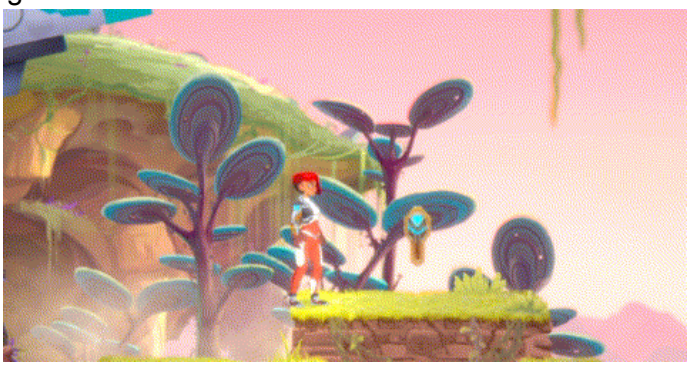
Now that we know what tags are, we can summarize different ways to *get reference to an object*:

- through *public fields*
- by *type/component* - `FindObjectOfType<T>()` - first active object with the given component, very slow (performs search)
- by *name* - `GameObject.Find(string name)` - first active object of the given name, very slow (performs search)
- by *tag* - `GameObject.FindGameObjectWithTag(string tag)`

Finally we would like to apply some effect on the player (it may be very simple, e.g. scaling the player up) and make the item disappear.

⭐**Task**: Affect the player somehow and destroy the item.

Hint:
Solution:
Make sure everything works as intended.



Effect as a UnityEvent

We can make everything more general by taking the specific effect of the power-up item out of the code and assigning it in the editor.

`UnityEvent` is a very similar concept to C# delegate but it allows for better manipulation through the Inspector.
To use it in the code we need to add `using UnityEngine.Events;`, then we can declare a public data field of type `UnityEvent` and invoke the assigned event/s with `UnityEvent.Invoke()`.
Assigning the event: Click on the + symbol in the Inspector, drag-and-drop some object or component to the field on the left and select which method of the object/component will be called in the field on the right (if necessary, set a parameter below it).

Unity - Manual: UnityEvents (unity3d.com)
Unity - Scripting API: UnityEvent (unity3d.com)
We will start by declaring a UnityEvent which will be called to apply the effect of the item.

⭐**Task**: Generalize the effect of the power-up item by making it a UnityEvent.

Hint:
Solution:

Now we need to prepare the actual effects the power-up may have.

⭐**Task**: Create a component for various effects, attach it to the Ellen object and implement one specific effect.

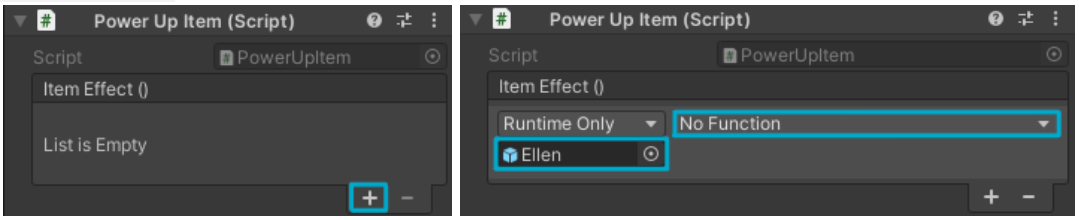
Hint:

Solution:

Finally we can connect it together and assign a specific effect as the event.

⭐**Task**: Assign the event in the Inspector.

Solution:



Temporary effect

Now we would like to apply the effect only for some predetermined amount of time and after that remove the effect.

`Coroutine` - A function which is executed in intervals. It works with special *yield statements* which *suspend the code execution* for a specific time. Then when the function continues, execution begins again right from where it left off.
Return type must be `IEnumerator` and the function must return anything implementing it.
`yield return null` - The code will resume in the next frame
`yield return new WaitForSeconds(3f)` - The code will resume after 3 seconds.
`StartCoroutine` - A function for starting the coroutine, takes a parameter of coroutine call or string of its name.

Unity - Manual: Coroutines (unity3d.com)
Unity - Scripting API: Coroutine (unity3d.com)
Unity - Scripting API: MonoBehaviour.StartCoroutine (unity3d.com)
Coroutines - Unity Learn
Unity - Scripting API: WaitForSeconds (unity3d.com)

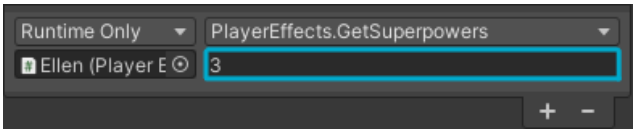
⭐**Task**: Use a coroutine to revert the effect after a fixed amount of time.

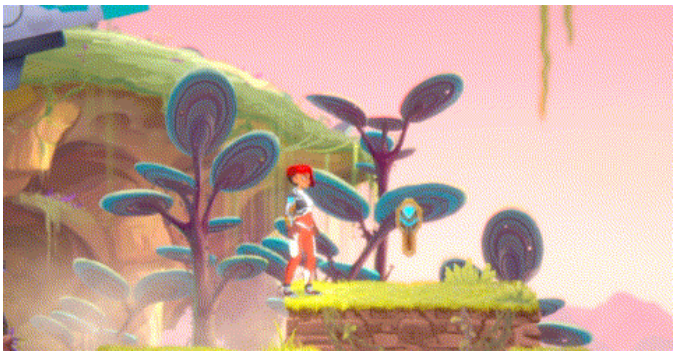
Hint:

Solution:

⭐**Task**: Add a parameter which will determine duration of the effect.

Solution:





Repeated use

Right now the item is usable just once. Instead of destroying it completely after picking it up, we can disable it only for a while so that it may be used repeatedly.

Enabling and disabling individual scripts/components during runtime:
`enabled {get; set;}` - Enable/disable component, true if local state is enabled
`isActiveAndEnabled {get;}` - True if local state is enabled and GameObject is activeInHierarchy.
`OnEnable()` - Called when the script becomes enabled (also during initialization of the object, after Awake, if the object is active).
`OnDisable()` - Called when the script becomes disabled.
Disabling a script means only disabling `Update` and `FixedUpdate` - other methods, e.g. `OnTriggerEnter2D`, will be called and will activate the script.

Unity - Scripting API: Behaviour.enabled (unity3d.com)
Unity - Scripting API: Behaviour.isActiveAndEnabled (unity3d.com)
Unity - Scripting API: MonoBehaviour.OnEnable() (unity3d.com)
Unity - Scripting API: MonoBehaviour.OnDisable() (unity3d.com)
Enabling and Disabling Components - Unity Learn

For our scenario, it is better to disable only components and not deactivate the item as a whole. This way the object will stay in the scene and we can later add some visual indications (e.g. make it partly transparent when disabled).

⭐**Task**: Instead of destroying the object, disable the collider component so that the item cannot be triggered again (overlaps are not detected, OnTriggerEnter2D is not invoked).

Hint:
Solution:

We can now disable the item in such a way that it stays in the scene but does not detect overlaps. Next step would be to enable the item again after some time.

`Invoke(string methodName, float time)` - A function allowing us to schedule method calls to occur at a later time (after some specific time delay).
- or instead of string: `Invoke(nameof(method), …);`

`InvokeRepeating(string methodName, float time, float repeatRate)` - Invokes the method `methodName` in `time` seconds, then repeatedly every `repeatRate` seconds.
`CancelInvoke()` - Cancels all `Invoke` calls on this MonoBehaviour.

Unity - Scripting API: MonoBehaviour.Invoke (unity3d.com)
Unity - Scripting API: MonoBehaviour.InvokeRepeating (unity3d.com)
Unity - Scripting API: MonoBehaviour.CancelInvoke (unity3d.com)
Invoke - Unity Learn

⭐**Task**: Schedule enabling the collider component again after a while.

Hint:
Solution:

The OnTriggerEnter2D method now does three steps:
- applies the item's effect,
- disables the item (collider),
- starts the timer to enable the item again.

Instead of `Invoke` we could use coroutines again (disable the item, wait for a while, enable the item). `Invoke` is used just for the sake of completeness.

Invoke is useful for simple cases - delayed call of method without parameters.

Coroutines allow us to pass parameters, to create more complex sequences very easily and are more efficient.

Conclusion

What you should know after finishing all the tutorials up until now:

- Basic terms: GameObject, Component, Prefab, ScriptableObjects
- Important types and their fields/methods: Transform, UnityEvent, SpriteRenderer
- Common methods of MonoBehaviour: Awake, Start, Update, FixedUpdate, OnCollisionEnter, OnTriggerEnter
- Basic constructs: iterating over children, getting components, finding objects, coroutines, enabling and disabling components, activating and deactivating objects, destroying objects, instantiation of prefabs, layers and tags, creating game objects and adding components
- RigidBody2D and applying forces, Colliders and triggers
- Usage of Time.deltaTime
- Handling input

Sources used

- https://learn.unity.com/tutorial/2d-game-kit-reference-guide#
- https://learn.unity.com/project/beginner-gameplay-scripting?uv=2019.3
- https://learn.unity.com/project/intermediate-gameplay-scripting?uv=2019.3
- Unity - Manual: Unity User Manual 2020.3 (LTS) (unity3d.com)
- slides from Introduction to Computer Game Development
- slides from Practical Course on Managed Game Development

Authors

★ Michaela Štolová
★ Karel Vlachovský