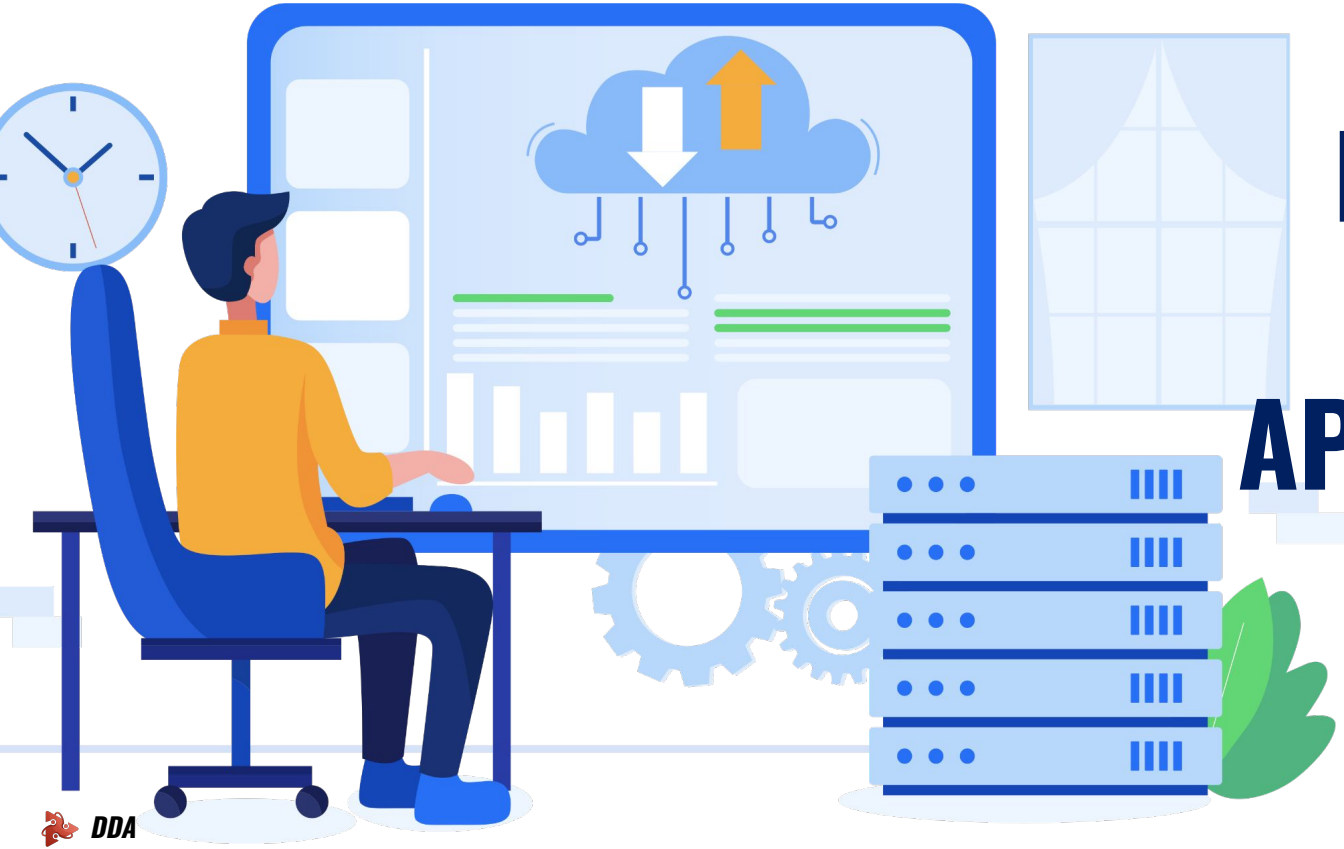
WEEK 3



(DDA) 2.2

DEVELOPING DYNAMIC APPLICATIONS

2021

DDA

Lesson 3: Getting it Right with Firebase

DDA

RECAP

Week 2

Learning Objectives

1. Working with CRUD
2. Working with Auth

DDA

COVERAGE

Week 3

Learning Objectives

1. Common Questions
2. CRUD Revisited
3. Working with Authentication
4. Planning Your Game
5. Leaderboard
6. CA: Assg Checkpoint + Design your Flow

Revisiting JSON - What you should **NOT** do…

Common student issues

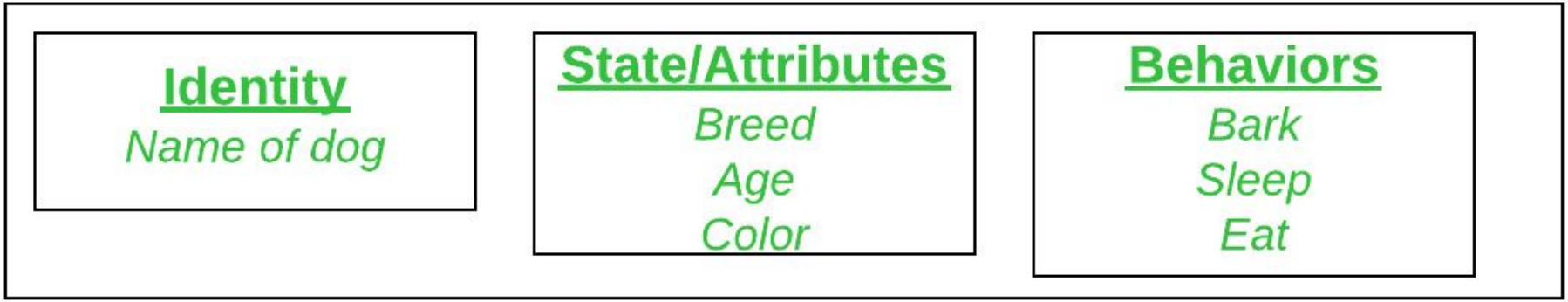
Thinking JSON conversion is writing to database
Not attaching scripts properly to the gameobjects

Revisiting - Student Queries/Doubts

- What exactly does continuewithmainthread
- What is Task?
- Handling firebase auth error
- What is data snapshot ?
- Why we need to loop
- Why editor not showing squiggy..
- How to CRUD
- How to Auth

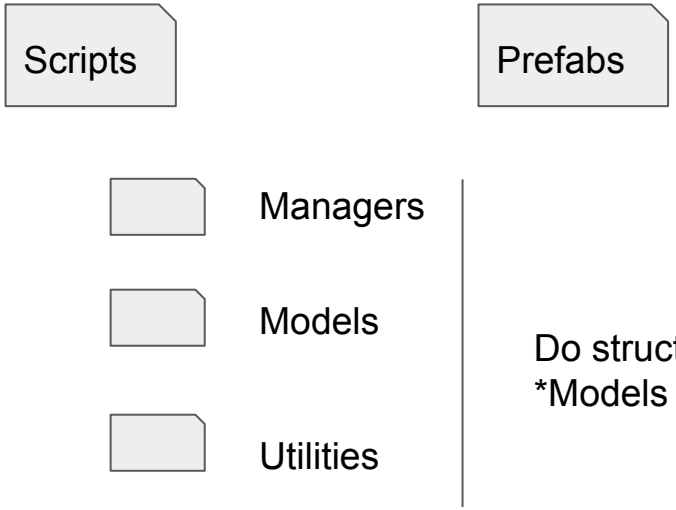
Revisiting Classes & Objects

Treat classes like a skeleton.



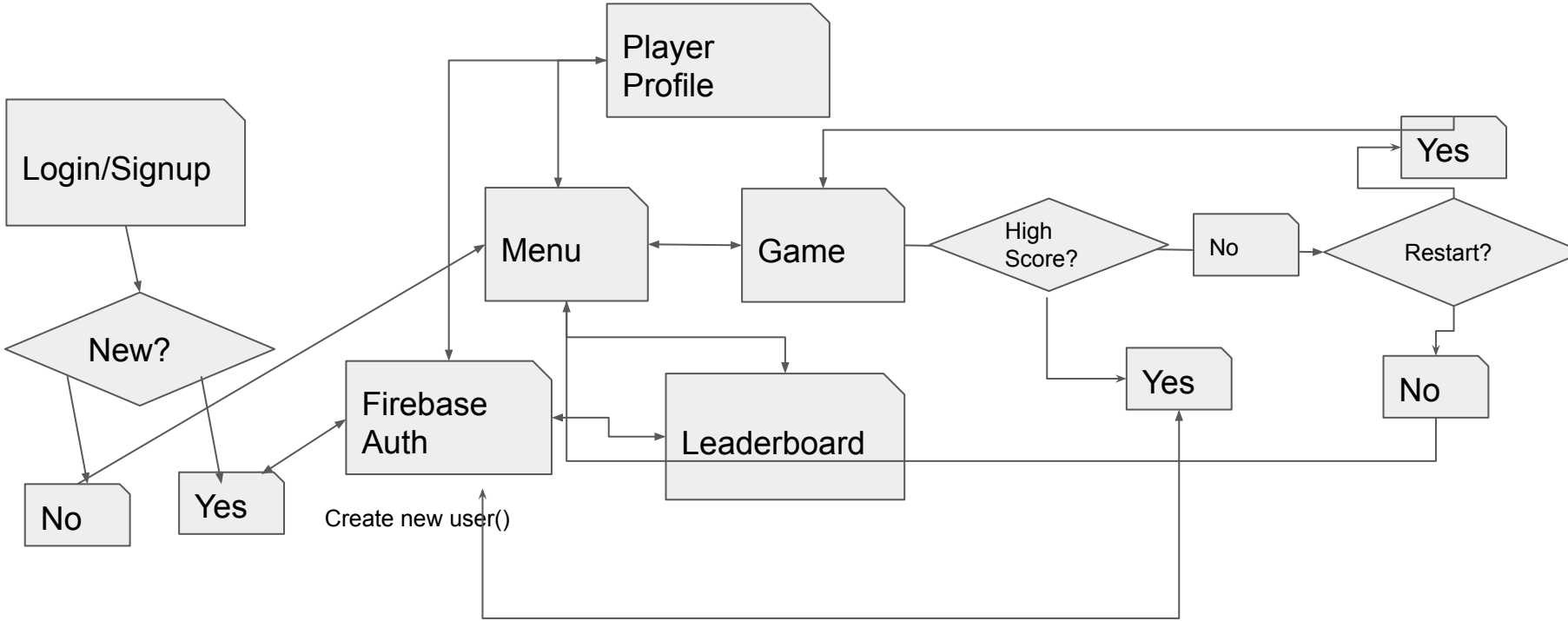
Reading Reference : C# Classes & Objects
https://www.geeksforgeeks.org/c-sharp-class-and-object/

Structuring Projects



Do structure your code into smaller folders
*Models can contain classes of objects

Simple Flow

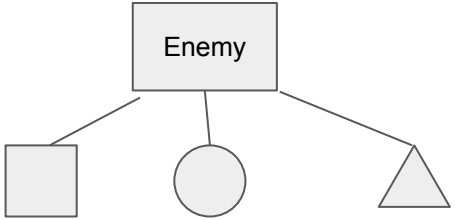

Player Profile
Login/Signup
Yes
Menu
Game
High Score?
No
Restart?
New?
Yes
No
Firebase Auth
Leaderboard
No
Yes
Create new user()

REVISITING #CRUD

Working with CRUD: Create with a Plan

First, we got to understand the concepts of Classes, Objects and JSON

```
//take note of the class name
//it's in PascalCase (each word, first letter
capital)
public class GameEnemy
{
    //class properties
    //only PUBLIC access properties can be converted
to JSON
    public string enemyName;
    public int xp;
    public int level;

    public GameEnemy()
    {

    }

    //constructor with params
    //we are mapping the class properties to the
parameters given
    //c# is smart enought to they are different
    public GameEnemy(string enemyName, int xp, int
level)
    {
        this.enemyName = enemyName;
        this.xp = xp;
        this.level = level;
    }
}
```



We pass values to our class
The keyword **new** creates our object

ONLY public properties/methods can be accessed using the dot notation when using in other classes

Example

In order to make many enemies, we need to create many different objects.

But we are “lazy”, hence we write a blueprint/skeleton of the enemy that allows us to use a template.
Each enemy has a name, xp, level

How do we create enemies?

```
//creating an enemy using constructor with params
GameEnemy square = new GameEnemy("Square Red Man", 100, 5);

//creating an enemy with empty constructor
GameEnemy circle = new GameEnemy();
//then we give it values
circle.enemyName = "Circle Red Man";
circle.xp = 5;
circle.level = 1;
```

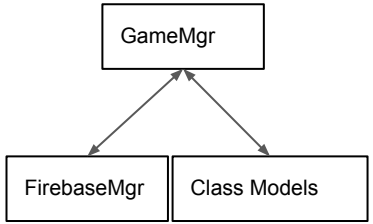
enemyName
Equals
GetHashCode
GetType
level
ToString
xp
(field) int GameEnemy.xp

Working with CRUD: Dealing with Firebase

Now, we have our basic classes for our templates (class models) we can start working with Firebase. The classes allows easy data manipulation and maneuvering. Provides easy way to modify data in the database.

Flow

1. Create an Firebase Manager
2. Create a Game Manager
3. Provide functionalities to both to interact with auth and models easily



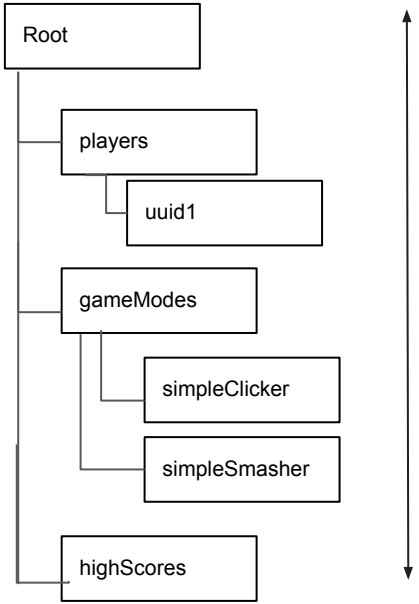
FirebaseMgr
-Awake()
-CreateNewPlayer()
-CheckPlayerNameExist()
-GetLatestPlayer()
-UpdatePlayerData()
-FindPlayer()
+EventHandlers

Example of some methods related to FirebaseMgr

Gamemanager - handle all game logic
Firebasemanager - handles our firebase specific calls, db references
Models - Contains our classes that represent our data

Working with CRUD: Planning some data modeling

We also need spend some time to plan our data.
In our game, we can have players, highscores, game modes



All these are nodes

Of course, uuid here references the unique id of the user.

Players Child Node

```
{
  "players": {
    "uuid1": {
      "playerName": “Charlene”,
      "xp": 10,
      “.......
    },
    "uuid2": {
      "playerName": “Charlene”,
      "xp": 10,
      “.......
    }
  }
}
```

What kind of player info do you want to hold?

GameModes Child Node

```
{
  "gameModes": {
    "simpleClicker": {
      "Uuid1": true,
      "Uuid2": true
    },
    "simpleSmasher": {
      "Uuid1": true,
      "Uuid2": false
    }
  }
}
```

What kind of game details

Firebase: Asynchronous & ContinueWithMainThread

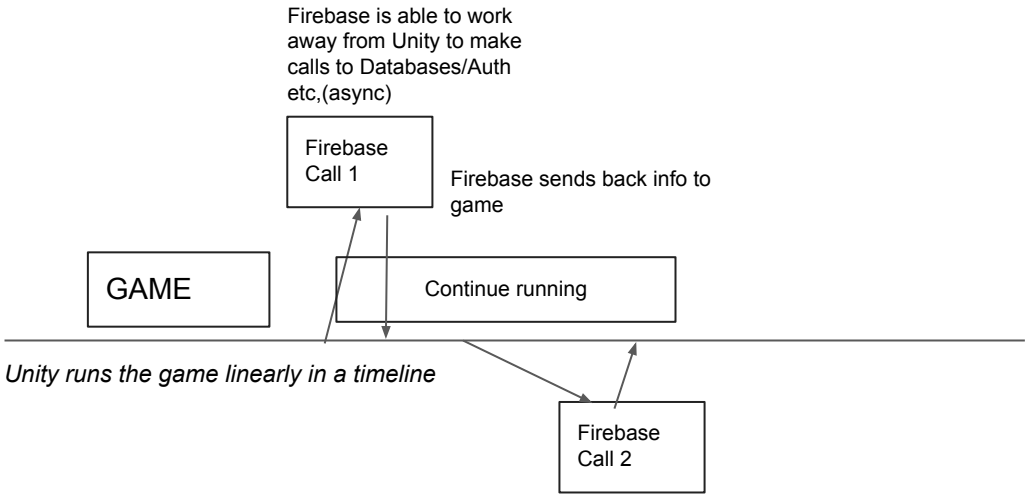
Flow

1. Check for Firebase Dependencies
2. Create threads of tasks to execute. This is where async comes into action

Firebase makes use of asynchronous and does not affect the game play.

However, we need a few techniques to overcome some of the issues you might encounter.

Coroutines
Async/Awake methods
ContinueWithMainThread



ContinueWithMainThread

https://www.youtube.com/watch?v=5GzFDXvZxKM

What is DataSnapshot

A DataSnapshot instance contains data from a FirebaseDatabase location.

Summary

A DataSnapshot instance contains data from a FirebaseDatabase location. Any time you read FirebaseDatabase data, you receive the data as a DataSnapshot.

They are efficiently-generated immutable copies of the data at a FirebaseDatabase location. They can't be modified and will never change. To modify data at a location, use a DatabaseReference reference (e.g. with DatabaseReference.SetValueAsync(object)

Reading Reference
https://firebase.google.com/docs/reference/unity/class/firebase/database/data-snapshot

CRUD: Updating/Saving (Single Value 1)

Flow

1. Use a database reference
2. Identify the path reference or child node to save to
3. Convert data to object then referencing the value to change

```
dbPlayerReference.Child(playerName).GetValueAsync().ContinueWithOnMainThread(task =>
```

This snippet refers to Firebase creating a reference path and retrieving the values related to the path ON the same thread as the game

```
SimplePlayer player = JsonUtility.FromJson<SimplePlayer>(snapshot.GetRawJsonValue());
```

Here we convert the snapshot value to a json string. Then we convert to a SimplePlayer object type which is named “player”

`SetValueAsync` allows us to define the new click counter. Of course, we take the old and add the new numbers in.

```
//Set a single value
//we know the path /$playerName/clicks/<newClickValue>
public void UpdateClickCounter(string playerName, int addClicks)
{
dbPlayerReference.Child(playerName).GetValueAsync().ContinueWithOnMainThread(task
=>
            {
                if (task.IsFaulted)
                {
                    // Handle the error...
                    throw task.Exception;
                }
                if (!task.IsCompleted)
                {
                    return;
                }

                else if (task.IsCompleted)
                {
                    DataSnapshot snapshot = task.Result;
                    if (snapshot.Exists)
                    {
                        SimplePlayer player =
JsonUtility.FromJson<SimplePlayer>(snapshot.GetRawJsonValue());
                        Debug.Log("player details" + player.clicks);
                        int clicks = player.clicks + addClicks;

dbPlayerReference.Child(playerName).Child("clicks").SetValueAsync(clicks);
                        Debug.Log("player details" + player.clicks);
                    }
                    else
                    {
                        Debug.Log("Snapshot not found");
                    }
                    //Debug.Log("" + snapshot.["clicks"]);
                }
            });
    }
```

Path: players/$playerName/

Path: players/$playerName/clicks

CRUD: Updating/Saving (Single Value 2)

Flow

1. Use a database reference
2. Identify the **full path reference**
3. Referencing the hard data based on the earlier path queried.

```
dbPlayerReference.Child(playerName).Child("clicks").GetValueAsync().Con
tinueWithOnMainThread(task =>
```

*Note: notice the new path used using `.Child("clicks")`
This snippet refers to Firebase creating a reference path to clicks and retrieving the values related to the path ON the same thread as the game

```
int clicks = Int32.Parse(snapshot.Value.ToString()) + addClicks;
```

Here we convert the snapshot value to a string and then convert it to an integer so we can add them properly

`SetValueAsync` allows us to define the new click counter. Of course, we take the old and add the new numbers in.

```
Path: players/$playerName/clicks
```

```
    public void UpdateClickCounter(string playerName, int addClicks)
    {

dbPlayerReference.Child(playerName).Child("clicks").GetValueAsync().ContinueWithOnM
ainThread(task =>
            {
                if (task.IsFaulted)
                {
                    // Handle the error...
                    throw task.Exception;
                }
                if (!task.IsCompleted)
                {
                    return;
                }

                else if (task.IsCompleted)
                {
                    DataSnapshot snapshot = task.Result;

                    if (snapshot.Exists)
                    {
                        int clicks = Int32.Parse(snapshot.Value.ToString()) +
addClicks;

dbPlayerReference.Child(playerName).Child("clicks").SetValueAsync(clicks);
                        //Debug.Log("player details" + player.clicks);
                    }
                    else
                    {
                        Debug.Log("Snapshot not found");
                    }
                    //Debug.Log("" + snapshot.["clicks"]);
                }
            });
    }
```

CRUD: Updating/Saving (Multiple Values)

Flow

1. Use a dictionary
2. Identify path of Parent node and then update with the dictionary data

* The dictionary works like a whole suite of child nodes you want it to path to and the value to update

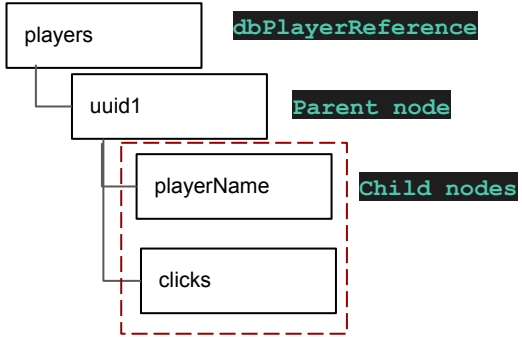
```
Dictionary<string, System.Object> childUpdates = new Dictionary<string, System.Object>();
```

This snippet uses the Dictionary data structure to create key:value pairs. The keys are the path to map to, with the Object as the values.

```
dbPlayerReference.Child(playerName).UpdateChildrenAsync(childUpdates)
```

Here we reference the Parent path we want and send the child information over

```
public void UpdatePlayerDataWithUserNameAsKey(string playerName)
    {
        Dictionary<string, System.Object> childUpdates = new Dictionary<string, System.Object>();
        childUpdates["/xp"] = 100;
        childUpdates["/clicks"] = 100;

        //path: /players/$playerName/

dbPlayerReference.Child(playerName).UpdateChildrenAsync(childUpdates);
    }
```



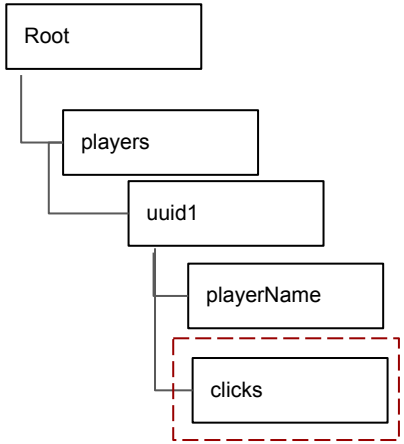
Path: players/$playerName/

CRUD: Deleting

Flow

1. Deleting works by referencing a path and delete the specific path node
2. Use RemoveValueAsync() to remove specified node

How about deleting many nodes?
Just delete the parent node



Bunch of parent and child nodes

Path: /players/$playerName/clicks

```
dbPlayerReference.Child(playerName).Child("clicks").RemoveValueAsync()
```

```
Path: players/$playerName/clicks
```

Reading your Data

```
using System.Threading.Tasks;
using Firebase.Extensions;// for ContinueWithOnMainThread
```

Include **Firebase Extensions**.
This is to handle Threading

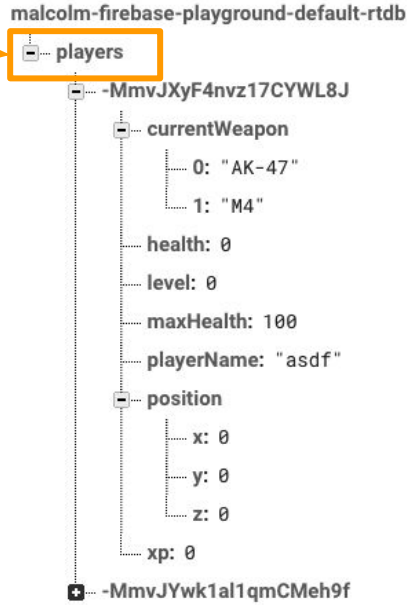
Depending on how you structure, get the reference node of your JSON tree. This is the json path (“path/to/query…. .”) eg. ("players/…..”)

```
FirebaseDatabase.DefaultInstance
    .GetReference("players")
    .GetValueAsync().ContinueWithOnMainThread(task =>
{
        if (task.IsFaulted) {
          // Handle the error...
        }
        else if (task.IsCompleted) {
          DataSnapshot snapshot = task.Result;
          // Do something with snapshot...
        }
      });
```

We use **ContinueWithOnMainThread** to use the main thread that Unity is using.

Other alternatives is to use Coroutines

** You might see examples using ContinueWith(..), this uses a new thread on your computer*



The task result will contain a snapshot containing all data (entries) at that location, including child data. If there is no data, the snapshot returned is null.

https://firebase.google.com/docs/reference/unity/clas
https://firebase.google.com/docs/database/unity/retrieve-data

READING
#CRUD
#DEMO

Working with Handlers

Listen for events

You can add event listeners to subscribe on changes to data:

Event	Typical usage
`ValueChanged`	Read and listen for changes to the entire contents of a path.
`ChildAdded`	Retrieve lists of items or listen for additions to a list of items. Suggested use with `ChildChanged` and `ChildRemoved` to monitor changes to lists.
`ChildChanged`	Listen for changes to the items in a list. Use with `ChildAdded` and `ChildRemoved` to monitor changes to lists.
`ChildRemoved`	Listen for items being removed from a list. Use with `ChildAdded` and `ChildChanged` to monitor changes to lists.
`ChildMoved`	Listen for changes to the order of items in an ordered list. `ChildMoved` events always follow the `ChildChanged` event that caused the item's order to change (based on your current order-by method).

Reading Reference
https://firebase.google.com/docs/reference/unity/class/firebase/extensions/task-extension
https://firebase.google.com/docs/database/unity/retrieve-data

Working with Handlers

```
FirebaseDatabase.DefaultInstance
        .GetReference("Leaders")
        .ValueChanged += HandleValueChanged;
    }

    void HandleValueChanged(object sender,
ValueChangedEventArgs args) {
      if (args.DatabaseError != null) {

Debug.LogError(args.DatabaseError.Message);
        return;
      }
      // Do something with the data in
args.Snapshot
    }
```

ValueChangedEventArgs contains a **DataSnapshot** that contains the data at the specified location in the database at the time of the event. Calling **Value** on a **snapshot returns a Dictionary<string, object>** representing the data. If no data exists at the location, calling Value returns null.

In this example, **args.DatabaseError** is also examined to see if the read is canceled. For example, a read can be canceled if the client doesn't have permission to read from a Firebase database location. The DatabaseError will indicate why the failure occurred.

Reading Reference

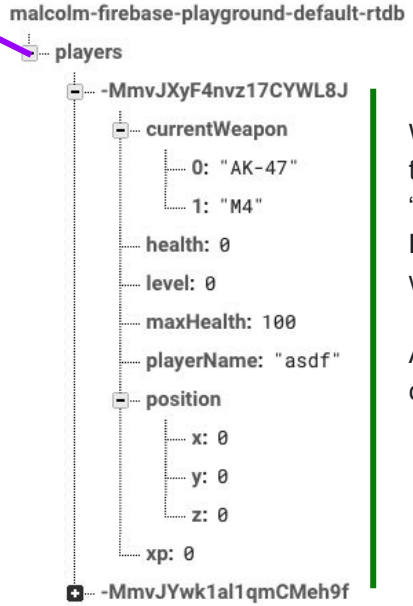
https://firebase.google.com/docs/reference/unity/class/firebase/extensions/task-extension

https://firebase.google.com/docs/database/unity/retrieve-data

Working with Handlers

```
DatabaseReference playerRef =
FirebaseDatabase.DefaultInstance.GetReference("players");

playerRef.ValueChanged += HandlePlayerValueChanged;//parent node
has changes
void HandlePlayerValueChanged(object send, ValueChangedEventArgs
args)
    {
        if (args.DatabaseError != null)
        {
            Debug.LogError(args.DatabaseError.Message);
            return;
        }

        numPlayers = (int)args.Snapshot.ChildrenCount;
        txtPlayerCount.text = "Total Players in Game: " +
numPlayers;

        Debug.Log("HandlePlayerChange numChild:" + numPlayers);
    }
```

In this example, we are having the listener function to update the total number of players in the game



Whenever there's movement in the nodes under parent node "players", the listener **HandlePlayerValueChanged()** will be triggered.

A snapshot of the node and its children will be send back.

Reading Reference

https://firebase.google.com/docs/reference/unity/class/firebase/extensions/task-extension

https://firebase.google.com/docs/database/unity/retrieve-data

HANDLERS
#CRUD
#DEMO

CRUD - Update/Write

Basic write operations

For basic write operations, you can use SetValueAsync() to save data to a specified reference, replacing any existing data at that path. You can use this method to pass types that correspond to the available JSON types as follows:

- string
- long
- double
- bool
- Dictionary<string, Object>
- List<Object>

If you use a typed C# object, you can use the built in JsonUtility.ToJson() to convert the object to raw Json and call **SetRawJsonValueAsync().** For example, you may have a User class that looked as follows:

Reading Reference
https://firebase.google.com/docs/reference/unity/class/firebase/extensions/task-extension
https://firebase.google.com/docs/database/unity/retrieve-data

CRUD - Update

Map the given path and key, then set the desired value

```
DatabaseReference dbReference = FirebaseDatabase.DefaultInstance.RootReference;
dbReference.Child("/path/to/save").SetValueAsync(playerName);
//dbPlayerReference.Child(playerKey).Child("playerName").SetValueAsync(playerName);


DatabaseReference reference = FirebaseDatabase.Instance.GetReference("path/to/save");
reference.UpdateValueAsync(new Dictionary<string, object>(){
    {"child1","value1"},{"child2","value2"}
}, 10, (res) =>
{
    if (res.success)
    {
        Debug.Log("Write success");
    }
    else
    {
        Debug.Log("Write failed : " + res.message);
    }
});
```

Reading Reference

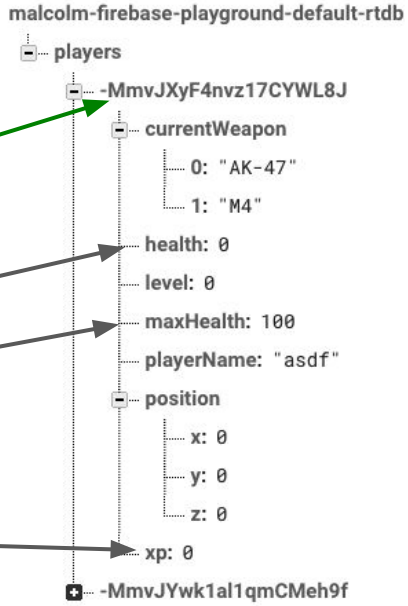
https://firebase.google.com/docs/database/unity/retrieve-data

https://firebase.google.com/docs/reference/unity/class/firebase/database/database-reference

CRUD - Update

Map the given path and key, then set the desired value

```
DatabaseReference dbReference = FirebaseDatabase.DefaultInstance.RootReference;
dbReference.Child("/path/to/save").SetValueAsync(playerName);

//dbPlayerReference.Child(playerKey).Child("playerName").SetValueAsync(playerName);

DatabaseReference reference =
FirebaseDatabase.DefaultInstance.GetReference("players/");
// Handle the error...
string id = "-MmvJXyF4nvz17CYWL8J";
Dictionary<string, object> childUpdates = new Dictionary<string, object>();
childUpdates[id + "/health"] = 456;
childUpdates[id + "/maxHealth"] = 456;
childUpdates[id + "/xp"] = 456;

reference.UpdateChildrenAsync(childUpdates);
```



Reading Reference

https://firebase.google.com/docs/database/unity/retrieve-data

https://firebase.google.com/docs/reference/unity/class/firebase/database/database-reference

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0

Updating
#CRUD
#DEMO

CRUD - Delete

The simplest way to delete data is to call **RemoveValueAsync**() on a reference to the location of that data.

You can also delete by specifying null as the value for another write operation such as SetValueAsync() or UpdateChildrenAsync(). You can use this technique with UpdateChildrenAsync() to delete multiple children in a single API call.

```
dbRef.OrderByChild("playerName").EqualTo(playerToSearch).Reference.RemoveValueAsync();
```

What the code means
dbRef - The FirebaseDatabase reference you are using
OrderByChild - this is an ordering / sorting feature of Firebase. It sorts by the desired child property
EqualTo - Does a filter based on the child given, in this case "playerToSearch"
Reference - Retrieves a snapshot of the found nodes
RemoveValuesAsync - Makes a call to **permanently remove** the nodes affected on that location

Reading Reference
https://firebase.google.com/docs/database/unity/retrieve-data

Formatting Strings

```
Debug.LogFormat("Using current user {0}", auth.CurrentUser.Email);

clicksPerSecondDisplay.text = string.Format("Clicks per sec: {0:0.00} CPS", score / currentSec);
```

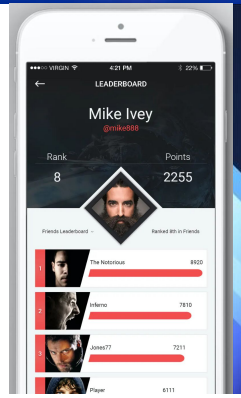

Reading Reference : Formatting Strings
https://docs.microsoft.com/en-us/dotnet/api/system.string.format?redirectedfrom=MSDN&view=net-5.0#overloads

Leaderboard

A typical leaderboard has some of the following information.
Displaying player username/identifier together with their **ranking position**
Game based attributes that determine leader
Player info to give more depth to the players

Player Name + Position	Attributes to compare (score, time, hits)	Player info/Statistics (xp, clan, level country, avatar, profile pic)
….	….	….

Fortnite Leaderboard Example

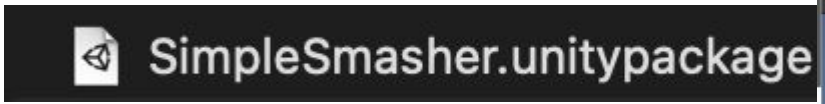


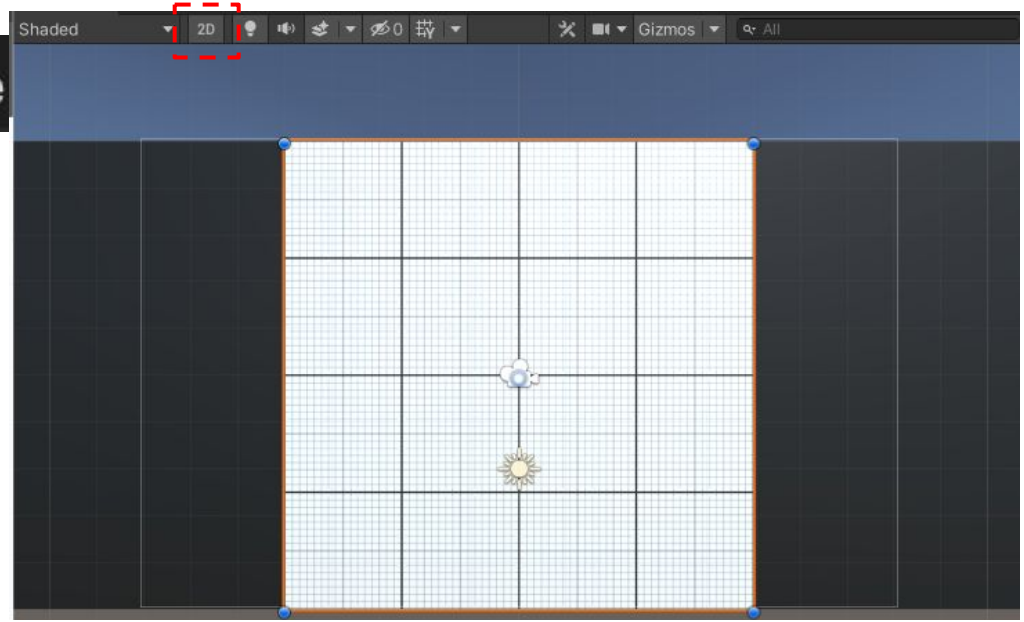
Working with Time

Make #SIMPLEGAME

Step 1: Create Project and Switch to 2D View

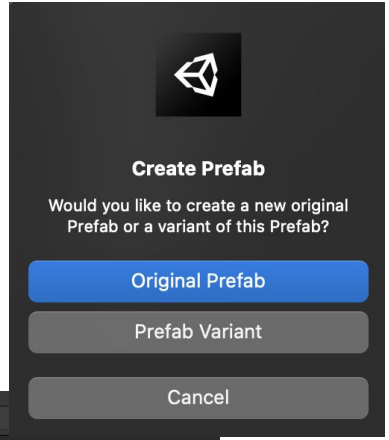
1. Download **SimpleSmasher** from Github
2. Import the .**unitypackage** into your project
3. Open the **SimpleSmash** scene, then delete the **sample scene** without saving
4. Click on the **2D icon** in Scene view to put Scene view in **2D**.

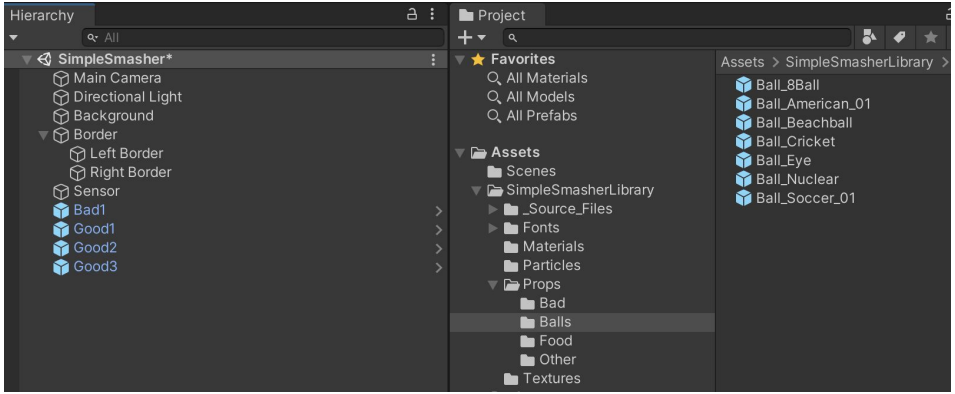


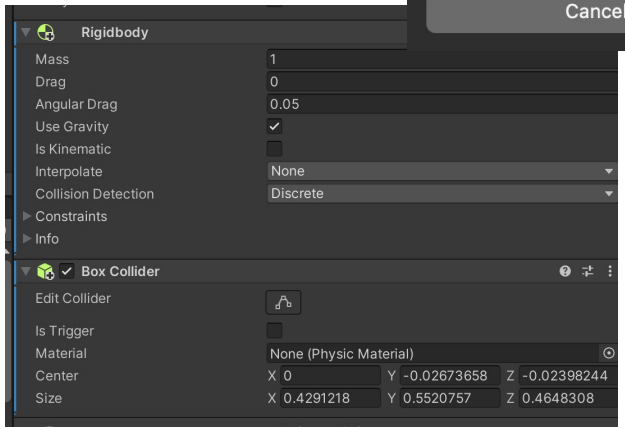


Step 2: Create good and bad objects

1. From the **Library**, drag 3 “good” objects and 1 “bad” object into the Scene, rename them “Good 1”, “Good 2”, “Good 3”, and “Bad 1”
2. Add **Rigid Body** and **Box Collider** components, then make sure that Colliders surround objects properly
3. Create a new Scripts folder, a new “Target.cs” script inside it, attach it to the **Target objects**
4. Drag all 4 targets into the **Prefabs folder** to create “original prefabs”, then **delete** them from the scene







Step 3: Toss objects randomly in the air

Now that we have 4 target prefabs with the same script, we need to toss them into the air with a random force, torque, and position.

1. In **Target.cs**, declare a new ***private Rigidbody targetRb;*** and initialize it in **Start()**
2. In **Start()**, add an **upward force** multiplied by a **randomized speed**
3. Add a **torque** with randomized **xyz values**
4. Set the **position** with a randomized **X value**

```
void Start()
    {
        targetRb = GetComponent<Rigidbody>();
        targetRb.AddForce(Vector3.up *
Random.Range(12, 16), ForceMode.Impulse);
        targetRb.AddTorque(Random.Range(-10, 10),
Random.Range(-10, 10),
            Random.Range(-10,10), ForceMode.Impulse);
        transform.position = new
Vector3(Random.Range(-4, 4), -6);
    }
```

Test it out by throwing prefabs into your secne

Step 4: Replace messy code with new methods

Now that we have 4 target prefabs with the same script, we need to toss them into the air with a random force, torque, and position.

1. In **Target.cs**, declare a new ***private Rigidbody targetRb;*** and initialize it in **Start()**
2. In **Start()**, add an **upward force** multiplied by a **randomized speed**
3. Add a **torque** with randomized **xyz values**
4. Set the **position** with a randomized **X value**

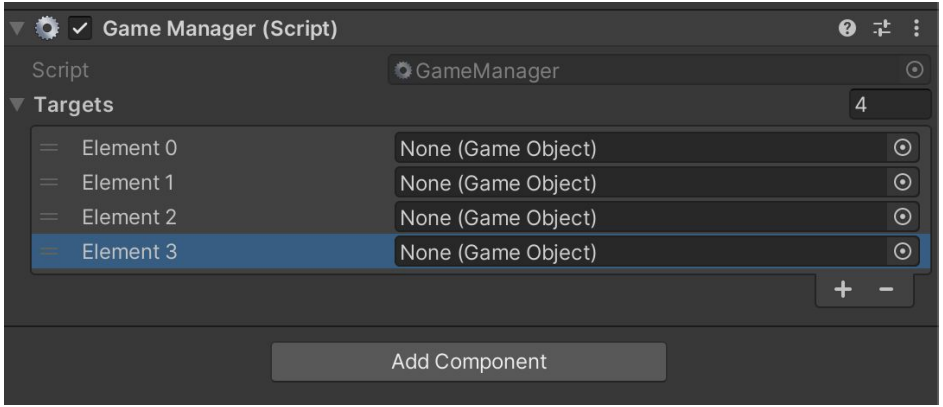
```
void Start()
    {
        targetRb = GetComponent<Rigidbody>();
        targetRb.AddForce(Vector3.up *
Random.Range(12, 16), ForceMode.Impulse);
        targetRb.AddTorque(Random.Range(-10, 10),
Random.Range(-10, 10),
            Random.Range(-10,10), ForceMode.Impulse);
        transform.position = new
Vector3(Random.Range(-4, 4), -6);
    }
```

Test it out by throwing prefabs into your secne

Step 5: Create object list in Game Manager

Now that we have a list of object prefabs, we should instantiate them in the game using coroutines and a new type of loop.

1. Declare and initialize a new ***private float spawnRate*** variable
2. Create a new ***IEnumerator SpawnTarget ()*** method
3. Inside the new method, ***while(true),*** wait **1 second**, generate a **random index**, and spawn a random **target**
4. In **Start()**, use the ***StartCoroutine*** method to begin spawning objects



Step 6: Create a coroutine to spawn objects

The next thing we should do is create a list for these objects to spawn from. Instead of making a Spawn Manager for these spawn functions, we're going to make a Game Manager that will also control game states later on

1. Create a new "Game Manager" **Empty object**, attach a new **GameManager.cs** script, then open it
2. Declare a new ***public List<GameObject> targets;***, then in the Game Manager inspector, change the list **Size** to 4 and assign your **prefabs**

```
public List<GameObject> targets;
```

```
public List<GameObject> targets;
private float spawnRate = 1.0f;

IEnumerator SpawnTarget()
{
    while (true)
    {
        yield return new
WaitForSeconds(spawnRate);
        int index = Random.Range(0,
targets.Count);
        Instantiate(targets[index]);
    }
}
// Start is called before the first frame update
void Start()
{
    StartCoroutine(SpawnTarget());
}
```

Step 7: Destroy target with click and sensor

Now that our targets are spawning and getting tossed into the air, we need a way for the player to destroy them with a click. We also need to destroy any targets that fall below the screen.

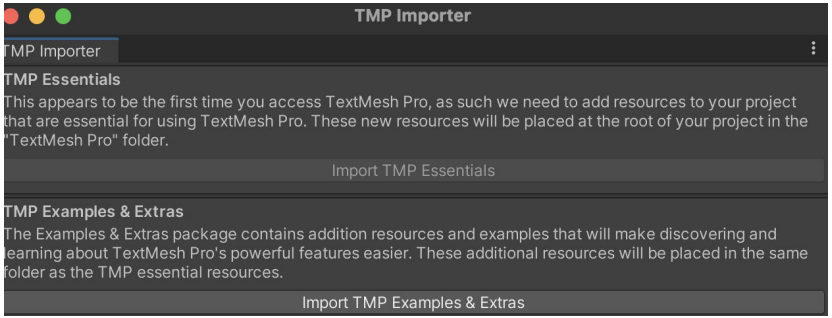
1. In **Target.cs**, add a new method for ***private void OnMouseDown() { }*** , and inside that method, destroy the gameObject
2. Add a new method for ***private void OnTriggerEnter(Collider other)*** and inside that function, destroy the gameObject

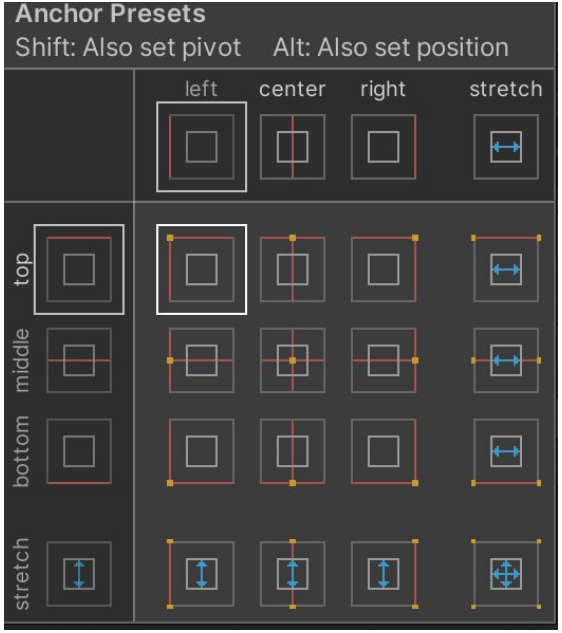
- Random objects are tossed into the air on intervals
- Objects are given random speed, position, and torque
- If you click on an object, it is destroyed

Step 8: Add Score text, position it on screen

Now that the basic text is in the scene and positioned properly, we should edit its properties so that it looks nice and has the correct text

1. In the Hierarchy, *Create* > *UI* > *Text* - ***TextMeshPro text***, then if prompted click the button to **Import TMP Essentials**
2. Rename the new object "Score Text", then **zoom out** to see the **canvas** in Scene view
3. Change the **Anchor Point** so that it is anchored from the **top-left corner**
4. In the inspector, change its **Pos X** and **Pos Y** so that it is in the top-left corner

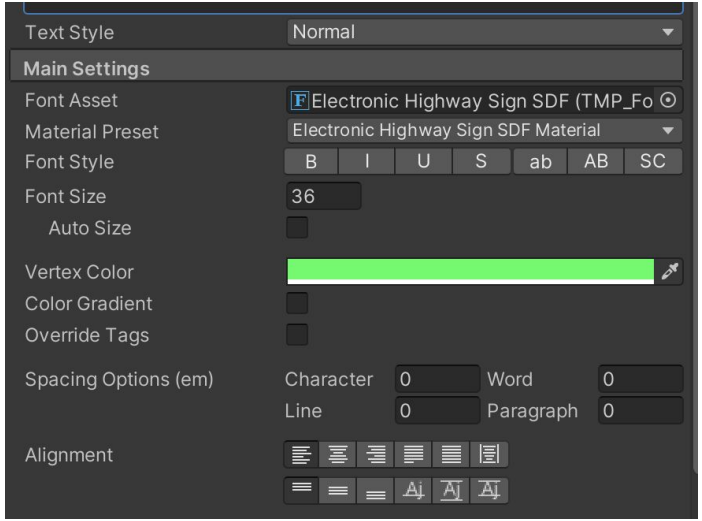




Step 9: Edit the Score Text's properties

Now that the basic text is in the scene and positioned properly, we should edit its properties so that it looks nice and has the correct text.

1. Change its text to "Score:"
2. Choose a **Font Asset**, **Style**, **Size**, and **Vertex color** to look good with your background



Step 10: Initialize score text and variable

We have a great place to display score in the UI, but nothing is displaying there! We need the UI to display a score variable, so the player can keep track of their points.

1. At the top of **GameManager.cs**, add "using TMPro;"
2. Declare a new ***public TextMeshProUGUI scoreText***, then assign that variable in the inspector
3. Create a new ***private int score*** variable and initialize it in ***Start()*** as ***score = 0;***
4. Also in **Start()**, set ***scoreText.text = "Score: " + score;***

```
using TMPro;
public class GameManager : MonoBehaviour
{
    public List<GameObject> targets;
    private float spawnRate = 1.0f;

    public TextMeshProUGUI scoreText;
    private int score;

    IEnumerator SpawnTarget()
    {
        while (true)
        {
            yield return new WaitForSeconds(spawnRate);
            int index = Random.Range(0, targets.Count);
            Instantiate(targets[index]);
        }
    }
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(SpawnTarget());
        score = 0;
        scoreText.text = "Score: " + score;
    }
```

Step 11: Create a new UpdateScore method

The score text displays the score variable perfectly, but it never gets updated. We need to write a new function that racks up points to display in the UI

1. Create a new ***private void UpdateScore*** method that requires one ***int scoreToAdd*** parameter
2. Cut and paste ***scoreText.text = "Score: " + score;*** into the new method, then call ***UpdateScore(0)*** in ***Start()***
3. In ***UpdateScore()***, increment the score by adding
4. ***score += scoreToAdd;***
5. Call ***UpdateScore(5)*** in the **spawnTarget()** function

```
using TMPro;
public class GameManager : MonoBehaviour
{
    public List<GameObject> targets;
    private float spawnRate = 1.0f;

    public TextMeshProUGUI scoreText;
    private int score;

    IEnumerator SpawnTarget()
    {
        while (true)
        {
            yield return new WaitForSeconds(spawnRate);
            int index = Random.Range(0, targets.Count);
            Instantiate(targets[index]);
        }
    }
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(SpawnTarget());
        score = 0;
        scoreText.text = "Score: " + score;
    }
```

Step 12: Add score when targets are destroyed

Now that we have a method to update the score, we should call it in the target script whenever a target is destroyed.

1. In GameManager.cs, make the ***UpdateScore*** method ***public***
2. In Target.cs, create a reference to ***private GameManager gameMana private void Start()***
3. ***{***
4. ***gameManager = GameObject.Find("GameManager").Get Component<GameManager>();***
5.
6. ***ger;***
7. Initialize GameManager in ***Start()*** using the ***Find()*** method
8. When a target is **destroyed**, call ***UpdateScore(5);***, then **delete** the method call from SpawnTarget()

```
private void Start()
{
    gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();

    //overwrite unity function
    private void OnMouseDown()
    {
        gameManager.UpdateScore(5);
        Destroy(gameObject);
    }
```

Target.cs

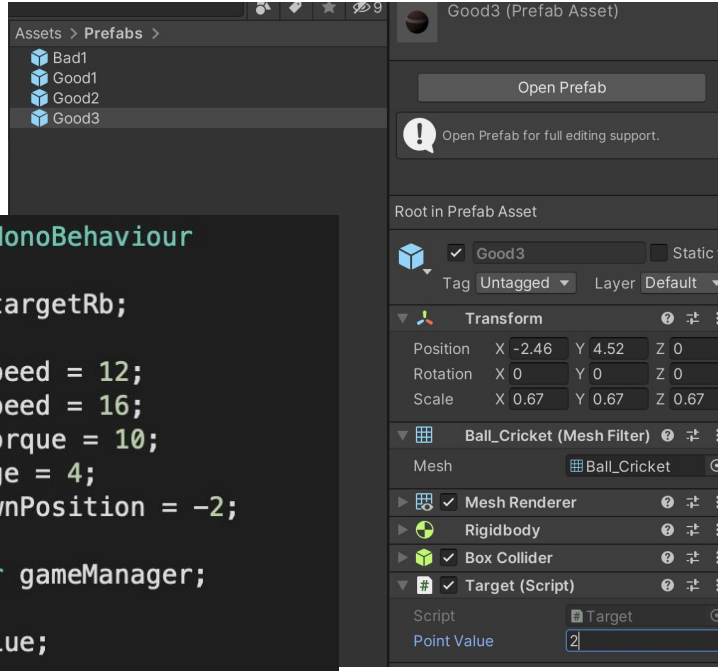
```
public void UpdateScore(int scoreToAdd)
{
    score += scoreToAdd;
    scoreText.text = "Score: " + score;

}

IEnumerator SpawnTarget()
{
    while (true)
    {
        //UpdateScore(5);
        yield return new WaitForSeconds(spawnRate);
        int index = Random.Range(0, targets.Count);
        Instantiate(targets[index]);
    }
}
```

GameManager.cs

Step 13: Assign a point value to each target

The score gets updated when targets are clicked, but we want to give each of the targets a different value. The good objects should vary in point value, and the bad object should subtract points

1. In Target.cs, create a new ***public int pointValue*** variable
2. In each of the **Target prefab's** inspectors, set the **Point Value** to whatever they're worth, including the bad target's **negative value**
3. Add the new variable to ***UpdateScore(pointValue);***




```
public class Target : MonoBehaviour
{
    private Rigidbody targetRb;

    private float minSpeed = 12;
    private float maxSpeed = 16;
    private float maxTorque = 10;
    private float xRange = 4;
    private float ySpawnPosition = -2;

    private GameManager gameManager;

    public int pointValue;
```

Step 14: Add particle system

If we want some "Game Over" text to appear when the game ends, the first thing we'll do is create and customize a new UI text element that says "Game Over".

1. Right-click on the **Canvas**, create a new *UI* > ***TextMeshPro - Text*** object, and rename it "Game Over Text"
2. In the inspector, edit its **Text**, **Pos X**, **Pos Y**, **Font Asset**, **Size**, **Style**, **Color**, and **Alignment**
3. Set the "Wrapping" setting to "Disabled"

Step 15: Create a Game Over text object

The score is totally functional, but clicking targets is sort of… unsatisfying. To spice things up, let's add some explosive particles whenever a target gets clicked

1. In Target.cs, add a new ***public ParticleSystem explosionParticle*** variable
2. For each of your target prefabs, assign a **particle prefab** from *Course Library > Particles* to the **Explosion Particle** variable
3. In the ***OnMouseDown()*** function, **instantiate** a new explosion prefab

```
public ParticleSystem explosionParticle;



//overwrite unity function
   private void OnMouseDown()
   {
       Destroy(gameObject);
       gameManager.UpdateScore(pointValue);
       Instantiate(explosionParticle,
transform.position,
explosionParticle.transform.rotation);

   }
```

Step 16: Make GameOver text appear

We've got some beautiful Game Over text on the screen, but it's just sitting and blocking our view right now. We should deactivate it, so it can reappear when the game ends.

1. In GameManager.cs, create a new ***public TextMeshProUGUI gameOverText;*** and assign the **Game Over** object to it in the inspector
2. **Uncheck** the Active checkbox to **deactivate** the Game Over text by default
3. In ***Start()***, activate the Game Over text

Step 17: Create GameOver function

The "Game Over" message appears exactly when we want it to, but the game itself continues to play. In order to truly halt the game and call this a "Game Over', we need to stop spawning targets and stop generating score for the player.

1. Create a new ***public bool isGameActive;***
2. As the **first line** In ***Start()***, set ***isGameActive = true;*** and in ***GameOver()***, set ***isGameActive = false;***
3. To prevent spawning, in the ***SpawnTarget()*** coroutine, change ***while (true)*** to ***while (isGameActive)***
4. To prevent scoring, in Target.cs, in the ***OnMouseDown()*** function, add the condition ***if (gameManager.isGameActive) {***

Step 18: Add a Restart button

Our Game Over mechanics are working like a charm, but there's no way to replay the game. In order to let the player restart the game, we will create our first UI button

1. Right-click on the **Canvas** and *Create* > *UI* > *Button*
   Note: You could also use **Button - TextMeshPro** for more control over the button's text.
2. Rename the button "Restart Button"
3. Temporarily **reactivate** the Game Over text in order to reposition the Restart Button nicely with the text, then **deactivate** it again
4. Select the Text child object, then edit its **Text** to say "Restart", its **Font, Style,** and **Size**

Step 19: Make the restart button work

We've added the Restart button to the scene and it LOOKS good, but now we need to make it actually work and restart the game.

1. In GameManager.cs, add ***using UnityEngine.SceneManagement;***
2. Create a new ***public void RestartGame()*** function that reloads the current scene
3. In the **Button's** inspector, click **+** to add a new **On Click event**, drag it in the **Game Manager** object and select the ***GameManager.RestartGame*** function

Step 20: Show restart button on game over

The Restart Button looks great, but we don't want it in our faces throughout the entire game. Similar to the "Game Over" message, we will turn off the Restart Button while the game is active.

1. At the top of GameManager.cs add ***using UnityEngine.UI;***
2. Declare a new ***public Button restartButton;*** and assign the **Restart Button** to it in the inspector
3. **Uncheck** the "Active" checkbox for the **Restart Button** in the inspector
4. In the ***GameOver*** function, activate the **Restart Button**

Step 18: Stop spawning and score on GameOver

We've temporarily made the "Game Over" text appear at the start of the game, but we actually want to trigger it when one of the "Good" objects is missed and falls.

1. Create a new ***public void GameOver()*** function, and **move** the code that activates the game over text inside it
2. In Target.cs, call ***gameManager.GameOver()*** if a target collides with the **sensor**
3. Add a new "Bad" tag to the **Bad object**, add a condition that will only trigger game over if it's *not* a bad object

C# Classes w/ JSON - Things to Note

```
/* Example of a User class */
public class User
{
    //public - access modifiers (seen outside the class)
    //required when we are using JsonUtility
    public string username;
    public string email;

    //private is default for all classes in c#
    //private only seen inside the class
    private int age = 10;

    //empty constructor
    public User()
    {

    }

    //constructor with 2 values
    public User(string username, string email)
    {
        //this refers to the properties in the class
        this.username = username;
        this.email = email;
    }
}
```

Got to set class properties to **public** access modifier
Required when we want to use JsonUtility.ToJson()

This property age won't be ported over when using JsonUtility.ToJSON()

https://docs.unity3d.com/ScriptReference/JsonUtility.ToJson.html

```
User squid = new User();
squid.username = "Oh Ii-Nam";
squid.email = "iinam@squid.com";
Debug.Log(JsonUtility.ToJson(squid));
//Output Json
//age variable value is missing here
//{"username": "Oh Ii-Nam", "email":
"iinam@squid.com"}
```

Creating Authentication - Code Basics

```
//place other needed directives
using Firebase.Auth;

public class AuthManager : MonoBehaviour
{
   Firebase.Auth.FirebaseAuth auth;

   //initialize our auth instance
   private void Awake()
   {
       auth = FirebaseAuth.DefaultInstance;

   }
//methods to handle authentication
```

```
//automatically pass user info to the firebase project
//attempt to create new user or check with there's already one
auth.CreateUserWithEmailAndPasswordAsync(email, password).ContinueWith(task =>
{
    //perform task handling
    if(task.IsFaulted || task.IsCanceled)
    {
        Debug.LogError("Sorry, there was an error creating your new account,
ERROR: " + task.Exception);
        return;//exit from the attempt
     }else if (task.IsCompleted)
     {
         Firebase.Auth.FirebaseUser newPlayer = task.Result;
         Debug.LogFormat("Welcome to Sotong Games {0}", newPlayer.Email);
         //do anything you want after player creation eg. create new player

     }
});
```

Creating Users

Perform validation and checks on your form, then pass the details to the `CreateUserWithEmailAndPasswordAsync` This will create the user IF the email is a proper email and the password meets Auth requirements

Note that the function executes separately

Reading Reference

https://firebase.google.com/docs/auth/unity/start

Creating Authentication - What's Next?

Start Making Your Players
We can adopt the generated unique User UID when a new user is created. That can be our $key for new players into the system.

Users can further update their profile details using a data structure in your Realtime Database

Start Modifying Your Reset Emails

Start Creating User Related Features
Some features to think about are:

- Logout
- Login errors
- Modifying email verification email
- handling Authentication issues
- storing the User UID in the game,
- Anything else that is user related :)
- Checking for unique usernames
- Using Multi-Sign in providers options
- Handling Lost password

Creating Authentication - Signin

Signing in Players

Signin your players and check whether they are valid

Can also say when was the last sign in

Unity Issues: Editor Messing Up

https://blog.terresquall.com/2020/11/fixing-visual-studios-intellisense-autocomplete-in-unity/

WEEK 4

NEXT WK HOME BASED!

Lesson Mode

TBA

MS TEAMS