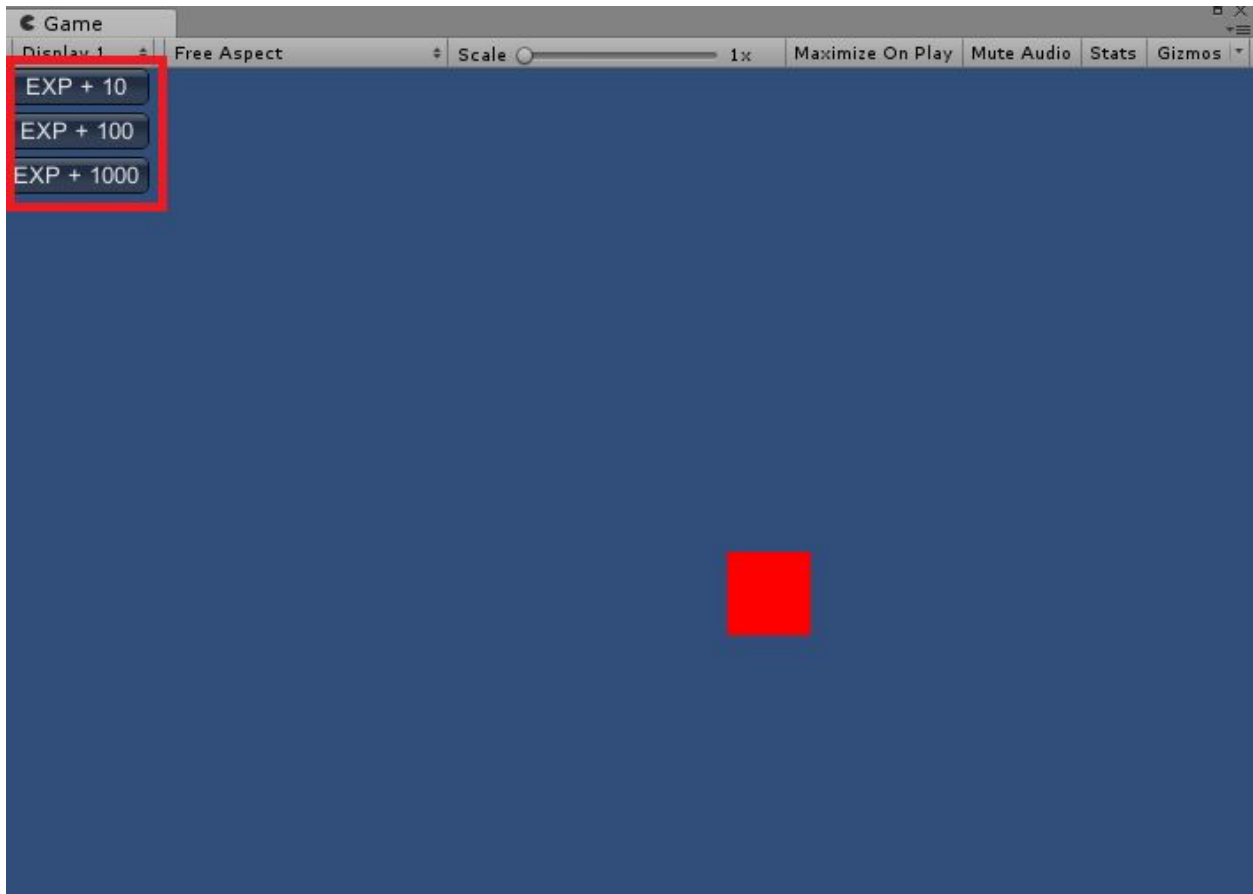


Companion Document for Game

| | |
|---|-----------|
| 1. The Game World: | 3 |
| 1.1. PlayerMovement.cs: | 4 |
| 1.2. TestLevelUp.cs: | 5 |
| 1.3. EnterBattle.cs: | 6 |
| 2. Game Information: | 7 |
| 2.1. GameInformation.cs: | 7 |
| 2.2. SaveInformation.cs: | 8 |
| 2.3. LoadInformation.cs: | 9 |
| 2.4. PPSerialization.cs (Player Prefs Serialization): | 9 |
| 2.5. NewGame.cs: | 10 |
| 3. Player Character: | 12 |
| 3.1. BasePlayer.cs: | 12 |
| 3.2. BaseCharacterClass.cs: | 13 |
| 3.3. BaseHeroClass.cs: | 14 |
| 4. Items: | 15 |
| 4.1. BaseItem.cs: | 15 |
| 4.2. BaseWeapon.cs: | 16 |
| 4.3. BaseBaseballBat.cs: | 17 |
| 5. Stats: | 18 |
| IncreaseExperience.cs: | 18 |
| LevelUp.cs: | 19 |
| Money.cs: | 20 |
| 6. Battle: | 21 |
| 6.1. TurnBaseBattle.cs: | 22 |
| 6.2. BaseNote.cs: | 23 |

1. The Game World:



This scene is what the user first sees when they run the project in unity. The player (Currently represented by a red square) can be controlled with the arrow keys (Or W,A,S,D). The highlighted region in the top-left corner shows a temporary GUI used to test the levelling script. Prezzing the 'Z' key will transition the user to the battle scene using a battle load function.

Scripts:

1.1. PlayerMovement.cs:

```
public class PlayerMovement : MonoBehaviour {
    public float moveSpeed = 10f;
    // Use this for initialization
    void Start () {
    }

    // Update is called once per frame
    void Update ()
    {
        if (Input.GetKey("a"))
        {
            transform.Translate((Vector2.left)*moveSpeed*Time.deltaTime);
        }
        if (Input.GetKey("d"))
        {
            transform.Translate((Vector2.right) * moveSpeed * Time.deltaTime);
        }
        if (Input.GetKey("w"))
        {
            transform.Translate((Vector2.up) * moveSpeed * Time.deltaTime);
        }
        if (Input.GetKey("s"))
        {
            transform.Translate((Vector2.down) * moveSpeed * Time.deltaTime);
        }
        if (Input.GetKeyDown("z"))
        {
            EnterBattle.BattleStart(new BaseSkeleton(), new BaseSkeleton(), null);
        }
        if (Input.GetKeyDown("x"))
        {
            //Cancel Key
        }
    }
}
```

This script handles all of the player input while navigating the world. This script inherits from `MonoBehaviour`, which to my understanding means it needs to be linked to an instance of a gameobject in unity to run, in this case it is assigned to the camera in the 'Game World' scene. All `MonoBehaviour` scripts have a `start()` function, which executes exactly once when the object is created and an `update()` function, which is called once at the start of each frame.

This script generates a vector dependent on the key presses in 2 dimensional space and translates the position of the player accordingly. The red region shows that pressing the 'Z' key will load the 'BattleStart()' function in the `EnterBattle.cs` script. This current example will create 2 enemy objects with the enemy class 'Skeleton' for the player to fight (See 6.1.). The blue region shows a public variable which controls the movement speed of the player.

1.2. TestLevelUp.cs:

```
public class TestLevelUp : MonoBehaviour {

    // Use this for initialization
    void Start () {
        NewGame.NewGameInformation();
        Debug.Log(GameInformation.Attack1);
        Debug.Log("Lv: " + GameInformation.Player1Level + " " + " +
    }

    // Update is called once per frame
    void Update () {

    }

    private void OnGUI()
    {
        if(GUILayout.Button("EXP + 10"))
        {
            IncreaseExperience.AddExperience(10);
            Debug.Log("Lv: " + GameInformation.Player1Level + "
        }
        if (GUILayout.Button("EXP + 100"))
        {
            IncreaseExperience.AddExperience(100);
            Debug.Log("Lv: " + GameInformation.Player1Level + "
        }
        if (GUILayout.Button("EXP + 1000"))
        {
            IncreaseExperience.AddExperience(1000);
            Debug.Log("Lv: " + GameInformation.Player1Level + "
        }
    }
}
```

This is a temporary script which exists to test the experience and levelling scripts (See 5.1.). When one of the GUI buttons is clicked, the corresponding amount of experience will be added to the player character.

1.3. EnterBattle.cs:

```
public class EnterBattle {  
  
    //Transitions the game the the battle scene and loads the relevent enemy data  
    public static void BattleStart(BaseEnemyClass EnemyClass1, BaseEnemyClass EnemyClass2, BaseEnemyClass EnemyClass3)  
    {  
        Application.LoadLevel("Battle");  
  
        //Creates an instance of the enemy object and loads stats of EnemyClass1  
        BaseEnemy Enemy1 = new BaseEnemy();  
        Enemy1.EnemyClass = EnemyClass1;  
        Debug.Log(Enemy1.EnemyClass.EnemyName);  
        //If there are multiple enemies, creates enemy objects from EnemyClass2 and/or EnemyClass3  
        if (EnemyClass2 != null)  
        {  
            BaseEnemy Enemy2 = new BaseEnemy();  
            Enemy2.EnemyClass = EnemyClass2;  
        }  
  
        if (EnemyClass3 != null)  
        {  
            BaseEnemy Enemy3 = new BaseEnemy();  
            Enemy3.EnemyClass = EnemyClass3;  
        }  
    }  
}
```

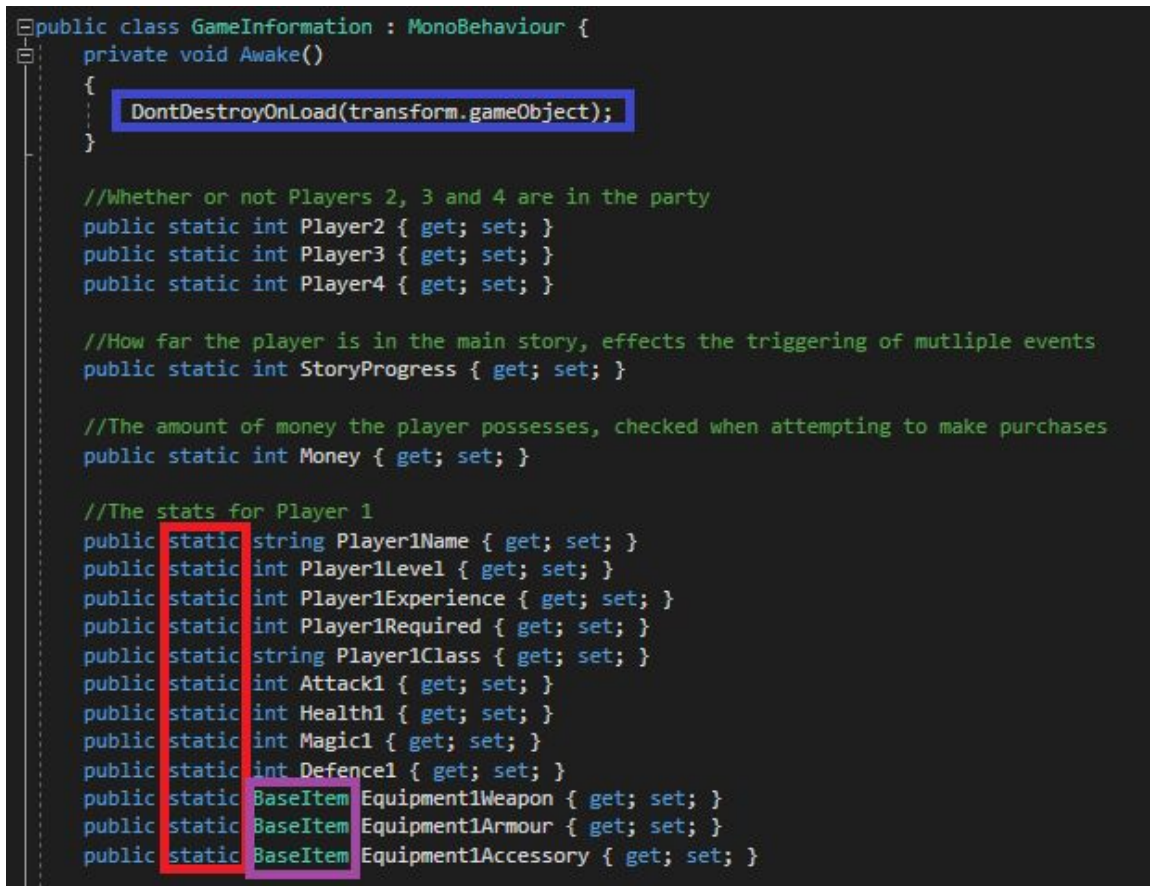
The BattleStart() function in this script is called whenever the player enters a battle. It takes three BaseEnemyClass() objects as inputs, transitions to the battle scene and creates enemy objects corresponding to given objects (See 6.1.)

2. Game Information:

Currently, all game information is linked to an invisible object called 'Game Information' which persists through scene changes. It keeps track of; Which characters are currently in your party, your character's levels and stats, what equipment your characters currently have, your progress in the main story and the amount of money your party currently holds.

Scripts:

2.1. GameInformation.cs:



```
public class GameInformation : MonoBehaviour {
    private void Awake()
    {
        DontDestroyOnLoad(transform.gameObject);
    }

    //Whether or not Players 2, 3 and 4 are in the party
    public static int Player2 { get; set; }
    public static int Player3 { get; set; }
    public static int Player4 { get; set; }

    //How far the player is in the main story, effects the triggering of mutliple events
    public static int StoryProgress { get; set; }

    //The amount of money the player possesses, checked when attempting to make purchases
    public static int Money { get; set; }

    //The stats for Player 1
    public static string Player1Name { get; set; }
    public static int Player1Level { get; set; }
    public static int Player1Experience { get; set; }
    public static int Player1Required { get; set; }
    public static string Player1Class { get; set; }
    public static int Attack1 { get; set; }
    public static int Health1 { get; set; }
    public static int Magic1 { get; set; }
    public static int Defence1 { get; set; }
    public static BaseItem Equipment1Weapon { get; set; }
    public static BaseItem Equipment1Armour { get; set; }
    public static BaseItem Equipment1Accessory { get; set; }
```

The scripts which keeps track of all the current game information. This is currently accessed when the player enters battle to generate his characters with the appropriate states as well as when saving, loading and creating a new game. This will eventually be accessed to check whether to trigger story events and whether the player has enough funds to purchase items.

The red region shows that all the variables in this script are public static variables. This means they can be accessed from other scripts and that they persist after being called by a function. Ideally, every variable which is permanently changed at some stage should be stored in this script.

The blue region shows a call to a function which prevents the object this script is linked to the being destroyed when scenes are changed.

The purple region shows that some of the stored information is neither an integer nor a string, but a custom class detailed in another script (See 4.1.).

2.2. SaveInformation.cs:

```
public class SaveInformation {  
  
    public static void SaveAllInformation() //This function is called when the  
    {  
        PlayerPrefs.SetInt("PLAYER2", GameInformation.Player2);  
        PlayerPrefs.SetInt("PLAYER3", GameInformation.Player3);  
        PlayerPrefs.SetInt("PLAYER4", GameInformation.Player4);  
  
        PlayerPrefs.SetInt("PROGRESS", GameInformation.StoryProgress);  
        PlayerPrefs.SetInt("MONEY", GameInformation.Money);  
  
        PlayerPrefs.SetInt("PLAYER1LEVEL", GameInformation.Player1Level);  
        PlayerPrefs.SetInt("PLAYER1EXP", GameInformation.Player1Experience);  
        PlayerPrefs.SetString("PLAYER1CLASS", GameInformation.Player1Class);  
        PlayerPrefs.SetInt("ATTACK1", GameInformation.Attack1);  
        PlayerPrefs.SetInt("DEFENCE1", GameInformation.Defence1);  
        PlayerPrefs.SetInt("HEALTH1", GameInformation.Health1);  
        PlayerPrefs.SetInt("MAGIC1", GameInformation.Magic1);  
    }  
}
```

This script saves the current values in the GameInformation.cs script to a text file whenever the SaveAllInformation() function is called. This causes the values to be stored under specified tags such as "PLAYER1LEVEL", "ATTACK1", etc.

```
if (GameInformation.Equipment1Weapon != null)  
{  
    PPSerialization.Save("EQUIPMENT1WEAPON", GameInformation.Equipment1Weapon);  
}
```

Only strings and integers can be stored in this manner, so a serialisation script is required to save information stored in custom classes i.e. equipment.

2.3. LoadInformation.cs:

```
public class LoadInformation {  
  
    public static void LoadAllInformation() //This function is called when the  
    {  
        GameInformation.Player2 = PlayerPrefs.GetInt("PLAYER2");  
        GameInformation.Player3 = PlayerPrefs.GetInt("PLAYER3");  
        GameInformation.Player4 = PlayerPrefs.GetInt("PLAYER4");  
  
        GameInformation.StoryProgress = PlayerPrefs.GetInt("PROGRESS");  
        GameInformation.Money = PlayerPrefs.GetInt("MONEY");  
  
        GameInformation.Player1Level = PlayerPrefs.GetInt("PLAYER1LEVEL");  
        GameInformation.Player1Experience = PlayerPrefs.GetInt("PLAYER1EXP");  
        GameInformation.Player1Class = PlayerPrefs.GetString("PLAYER1CLASS");  
        GameInformation.Attack1 = PlayerPrefs.GetInt("ATTACK1");  
        GameInformation.Defence1 = PlayerPrefs.GetInt("DEFENCE1");  
        GameInformation.Health1 = PlayerPrefs.GetInt("HEALTH1");  
        GameInformation.Magic1 = PlayerPrefs.GetInt("MAGIC1");  
    }  
}
```

This script overwrites the current GameInformation data with the values stored in the playerprefs file whenever the LoadAllInformation() script is called. To my understanding, these values are encrypted to prevent players from opening the file and changing the tag values.

2.4. PPSerialization.cs (Player Prefs Serialization):

```
public class PPSerialization  
{  
    public static BinaryFormatter binaryFormatter = new BinaryFormatter();  
  
    public static void Save(string saveTag, object saveObj)  
    {  
        MemoryStream memoryStream = new MemoryStream();  
        binaryFormatter.Serialize(memoryStream, saveObj);  
        string temp = System.Convert.ToBase64String(memoryStream.ToArray());  
        PlayerPrefs.SetString(saveTag, temp);  
    }  
  
    public static object Load(string saveTag)  
    {  
        string temp = PlayerPrefs.GetString(saveTag);  
        if (temp == string.Empty)  
        {  
            return (null);  
        }  
        MemoryStream memoryStream = new MemoryStream(System.Convert.FromBase64String(temp));  
        return binaryFormatter.Deserialize(memoryStream);  
    }  
}
```

This script converts objects to a binary string which allows them to be saved under a tag in playerprefs. I use this to enable saving and loading of the character equipment as it's a custom data class. I pretty much lifted this from a tutorial series I was watching, so I don't understand how to handle MemoryStreams in other contexts.

2.5. NewGame.cs:

```
public class NewGame
{
    public static void NewGameInformation()
    {
        //Creates instances of player objects in order to set up initial
        BasePlayer Player1 = new BasePlayer();
        BasePlayer Player2 = new BasePlayer();
        BasePlayer Player3 = new BasePlayer();
        BasePlayer Player4 = new BasePlayer();

        Player1.PlayerClass = new BaseHeroClass();
        Player2.PlayerClass = new BaseMusicianClass();
        Player3.PlayerClass = new BaseAcademicClass();
        Player4.PlayerClass = new BaseAthleteClass();

        Player1.CurrentExp = 0;
        Player2.CurrentExp = 0;
        Player3.CurrentExp = 0;
        Player4.CurrentExp = 0;

        //Sets initial story progress and money
        GameInformation.StoryProgress = 0;
        GameInformation.Money = 0;

        //Sets base stats and starting equipment for player 1
        GameInformation.Player1Name = "";
        GameInformation.Player1Level = 1;
        GameInformation.Player1Experience = 0;
        GameInformation.Player1Required = 20;
        GameInformation.Attack1 = Player1.PlayerClass.Attack;
        GameInformation.Defence1 = Player1.PlayerClass.Defence;
        GameInformation.Health1 = Player1.PlayerClass.Health;
        GameInformation.Magic1 = Player1.PlayerClass.Magic;
        GameInformation.Equipment1Weapon = new BaseFist();
    }
}
```

This script sets the default values for all Game Information when the player begins a new game. It temporarily creates an instance of the BasePlayer() objects, assigns them a class and then saves the stats corresponding to that class directly to the GameInformation object. When the NewGameInformation() script finishes running these instances are destroyed.

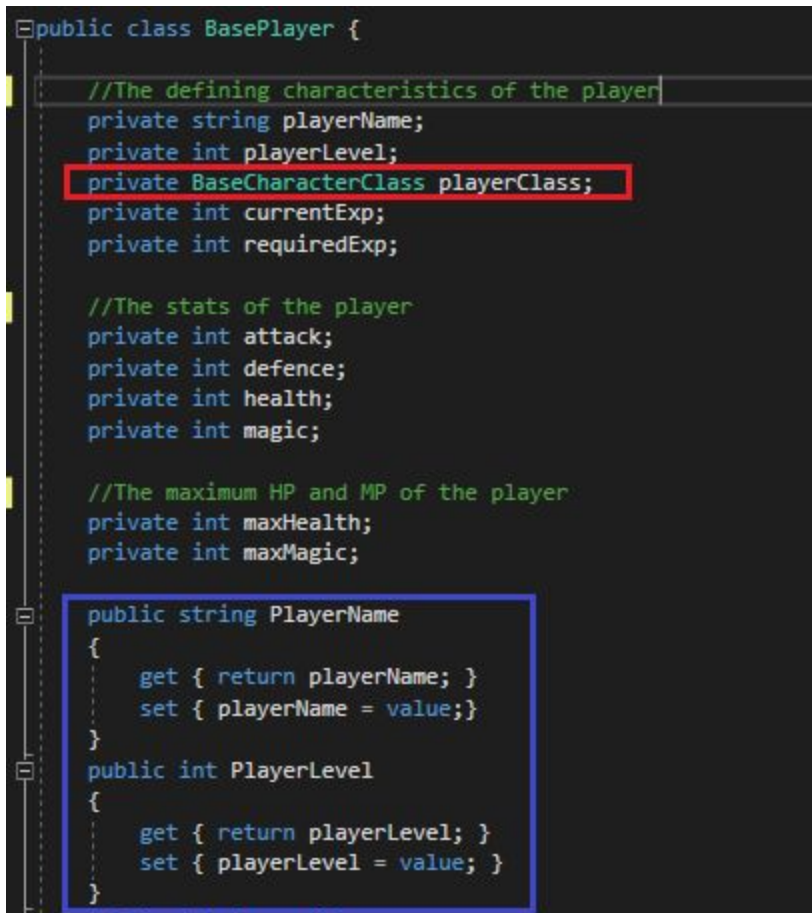
The red region shows the newly created BasePlayers() having their class assigned. Classes govern stats as well as what type of equipment can be used. The blue region shows the different type of classes, these are each contained in their own scripts which inherits from the BaseCharacterClass.cs script (See 3.3.).

The purple region shows an instance of an item object being assigned to the character. BaseFist() is a class representing a piece of equipment and inherits from the BaseItem.cs script (See 4.3).

3. Player Character:

Scripts:

3.1. BasePlayer.cs:



```
public class BasePlayer {  
    //The defining characteristics of the player  
    private string playerName;  
    private int playerLevel;  
    private BaseCharacterClass playerClass;  
    private int currentExp;  
    private int requiredExp;  
  
    //The stats of the player  
    private int attack;  
    private int defence;  
    private int health;  
    private int magic;  
  
    //The maximum HP and MP of the player  
    private int maxHealth;  
    private int maxMagic;  
  
    public string PlayerName  
    {  
        get { return playerName; }  
        set { playerName = value; }  
    }  
    public int PlayerLevel  
    {  
        get { return playerLevel; }  
        set { playerLevel = value; }  
    }  
}
```

This script defines the characteristics of a player object. So each player has a name, as well as various integer stats such as exp, attack and health. The red region shows that each player also has a class, this class determines the stats of the player.

The blue region shows that for each private variable belonging to this class, it creates a public variable of the same name and type (Note the capital first letters in the public variables). This definition comes with getter and setter functions, which allows these values to be read and written to from other scripts. I'm not 100% sure why this is necessary instead of just having all the variables public to begin with but that's something I'm looking into.

3.2. BaseCharacterClass.cs:

```
public class BaseCharacterClass
{
    private string characterClassName; //The name of the characters class

    //Base stats for the class
    private int attack;
    private int defence;
    private int health;
    private int magic;

    //Stats for the class at max level
    private int maxAttack;
    private int maxDefence;
    private int maxHealth;
    private int maxMagic;

    public string CharacterClassName
    {
        get { return characterClassName; }
        set { characterClassName = value; }
    }

    public int Attack
    {
        get { return attack; }
        set { attack = value; }
    }

    public int MaxAttack
    {
        get { return maxAttack; }
        set { maxAttack = value; }
    }
}
```

This script defines the characteristics of a base character class (Remembering that from our previous script, that all players have a character class). Due to poor variable naming convention on my part, the max stats here DO NOT mean the same thing as they for the the playerclass. Here, maxStat refers to the stat that a player of this class would have at lv 100. The base stats refer to the stat that a player of this class would have at lv 1. These values are interpolated between based on the players level to determine the actual stats for the player (See 5.2.).

3.3. BaseHeroClass.cs:

```
public class BaseHeroClass : BaseCharacterClass
{
    public BaseHeroClass()
    {
        CharacterClassName = "Hero";
        Attack = 10;
        MaxAttack = 200;
        Defence = 10;
        MaxDefence = 200;
        Health = 20;
        MaxHealth = 400;
        Magic = 5;
        MaxMagic = 100;
    }
}
```

This script defines the stats for the class 'Hero'. The red region shows that this class inherits from the BaseCharacterClass.cs script. So if we were to create a new instance of a player object and say:

"Player.playerclass = BaseHeroClass()"

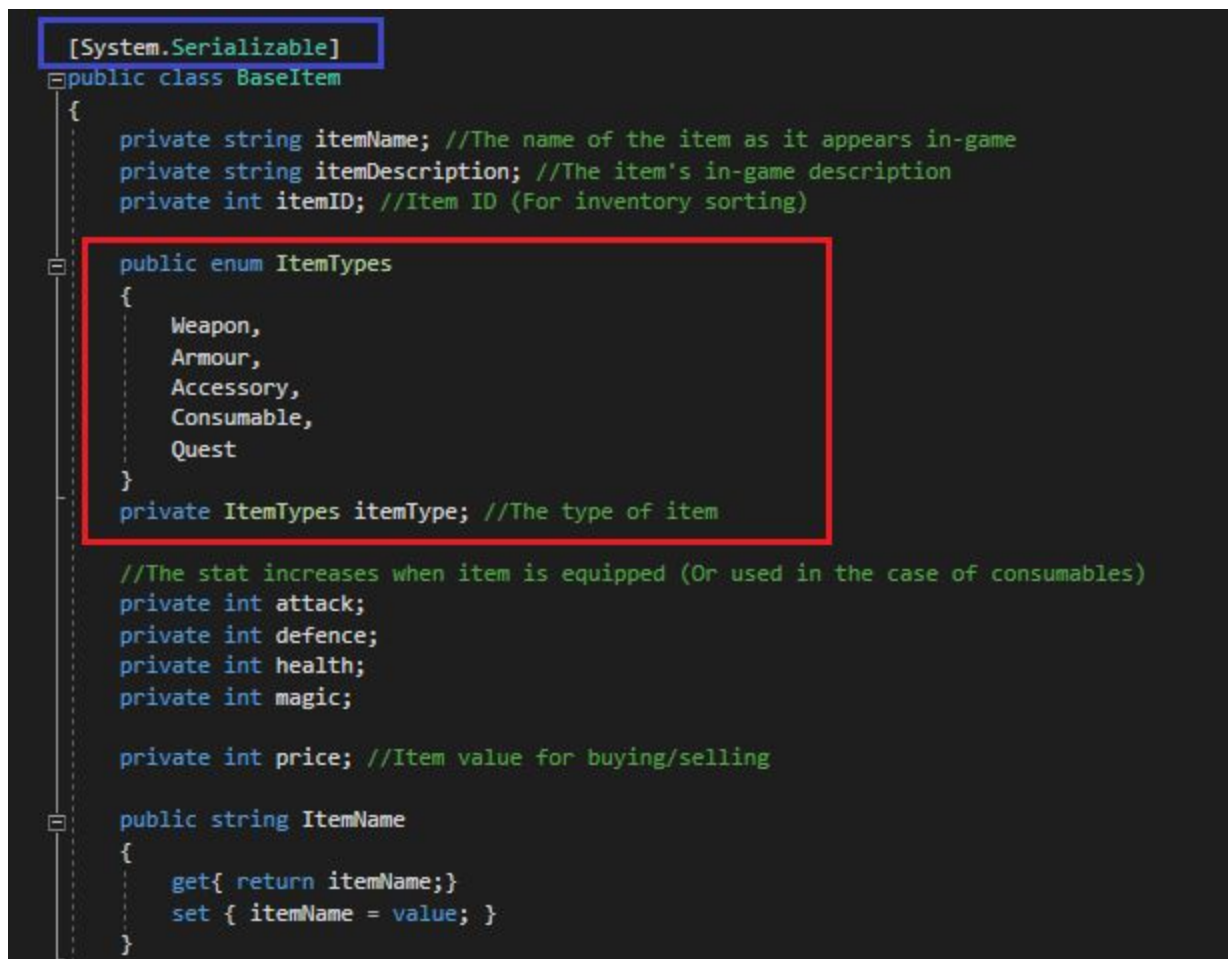
Then it would pre-fill all the variables in the BaseCharacterClass.cs script with the values from the BaseHeroClass() function.

BaseMusicianClass.cs, BaseAcademicClass.cs and BaseAthleteClass.cs are duplicates of this script with different values.

4. Items:

Scripts:

4.1. BaseItem.cs:

A screenshot of a code editor showing the BaseItem.cs script. The code is written in C# and includes several annotations. A blue box highlights the [System.Serializable] attribute at the top. A red box highlights the ItemTypes enum and the itemType property. The code defines a BaseItem class with private fields for itemName, itemDescription, itemID, attack, defence, health, magic, and price. It also includes a public string itemName property with get and set methods. The ItemTypes enum lists Weapon, Armour, Accessory, Consumable, and Quest. The itemType property is of type ItemTypes and is used to determine which slot the item is equipped in.

```
[System.Serializable]
public class BaseItem
{
    private string itemName; //The name of the item as it appears in-game
    private string itemDescription; //The item's in-game description
    private int itemID; //Item ID (For inventory sorting)

    public enum ItemTypes
    {
        Weapon,
        Armour,
        Accessory,
        Consumable,
        Quest
    }
    private ItemTypes itemType; //The type of item

    //The stat increases when item is equipped (Or used in the case of consumables)
    private int attack;
    private int defence;
    private int health;
    private int magic;

    private int price; //Item value for buying/selling

    public string itemName
    {
        get{ return itemName;}
        set { itemName = value; }
    }
}
```

In the same way that BasePlayer.cs defines the different characteristics of player characters, this script does the exact same for items. The red region shows that we've defined an enumeration of the possible kinds of items. The variable 'itemType' defined on the following line, is an instance of enumeration object and is used to determine which slot the item is equipped in (If equippable at all).

The blue region shows that we've enabled serialization for this script. My understanding is that this will help with the inventory system as we can have 20 potions in the same slot but the game recognises them as separate objects or something along those lines.

4.2. BaseWeapon.cs:

```
[System.Serializable]
public class BaseWeapon : BaseItem
{
    public enum WeaponTypes
    {
        Fist,
        Guitar,
        Tech,
        Bat
    }
    private WeaponTypes weaponType;
    private int weaponEffectID;

    public WeaponTypes WeaponType
    {
        get { return weaponType; }
        set { weaponType = value; }
    }

    public int WeaponEffectID
    {
        get { return weaponEffectID; }
        set { weaponEffectID = value; }
    }
}
```

This script inherits from the BaseItem.cs and defines the characteristics of weapon items. We can see that in addition to the variables of all items, a weapon also has a WeaponType and a WeaponEffectID. The weaponEffectID will reference a future list of effects, for example if the value is '1' that might mean the weapon deals fire damage or if the value is '7' it might have an additional life-steal effect etc.

BaseArmour.cs is essentially the same script with an enumerator for different ArmourTypes.

4.3. BaseBaseballBat.cs:

```
[System.Serializable]
public class BaseBaseballBat : BaseWeapon
{
    public BaseBaseballBat()
    {
        ItemName = "Baseball Bat";
        ItemDescription = "A Baseball Bat";
        ItemID = 5;
        ItemType = ItemTypes.Weapon;
        WeaponType = WeaponTypes.Bat;
        Attack = 20;
        Defence = 5;
        Health = 0;
        Magic = 0;
    }
}
```

This script shows an example of possible weapon stats, when a new BaseBaseballBat() object is created it corresponds to a weapon (Which is also an item) with the following stats. ItemID is a heirarchy for which items appear up the top of the inventory. We can also see that this script inherits from BaseWeapon.cs.

Every other script in the 'Weapons' project folder is essentially the same thing with different values.

5. Stats:

IncreaseExperience.cs:

```
public class IncreaseExperience
{
    //private static LevelUp levelUp = new LevelUp();
    public static void AddExperience(int XP) //Increases player experience for every player in
    {
        GameInformation.Player1Experience += XP;
        if (GameInformation.Player1Experience >= GameInformation.Player1Required)
        {
            BaseCharacterClass playerClass = new BaseHeroClass();
            LevelUp.LevelUpCharacter(playerClass, GameInformation.Player1Level);
        }
        if (GameInformation.Player2 != 0)
        {
            GameInformation.Player2Experience += XP;
            if (GameInformation.Player2Experience >= GameInformation.Player2Required)
            {
                //Level Up Function
            }
        }
    }
}
```

The AddExperience() function is called whenever the player gains XP. It automatically assigns the points to the main character before checking if the other characters are in the party and assigning experience to them (As shown by the blue region).

The red region shows that if the player has enough experience to level up, it creates a new BaseCharacterClass object, assigns it the players' class and calls the LevelUpCharacter() function in the LevelUp.cs script to calculate the new stats and alter the relevant game information data.

LevelUp.cs:

```
public class LevelUp {  
    //Maximum Character Level and the experience required to achieve  
    private static int maxLevel = 100;  
    private static int minLevelExp = 20;  
    private static int maxLevelExp = 40000;  
  
    public static void LevelUpCharacter(BaseCharacterClass Class, int level)  
    {  
        level += 1;  
        GameInformation.Player1Level = level;  
        AssignNewStats(Class, level);  
    }  
  
    private static void AssignNewStats(BaseCharacterClass Class, int level)  
    {  
        //Assigns the player new stats by interpolating between their minimum and maximum values  
        GameInformation.Attack1 = CalculateNewStats(Class.Attack, Class.MaxAttack, level);  
        GameInformation.Defence1 = CalculateNewStats(Class.Defence, Class.MaxDefence, level);  
        GameInformation.Magic1 = CalculateNewStats(Class.Magic, Class.MaxMagic, level);  
        GameInformation.Health1 = CalculateNewStats(Class.Health, Class.MaxHealth, level);  
  
        //Assigns the player a new experience required to reach the next level  
        GameInformation.Player1Required = CalculateNewStats(minLevelExp, maxLevelExp, level);  
    }  
  
    private static int CalculateNewStats(int min, int max, int position)  
    {  
        //Exponential plotting function for new stats  
        float x = Mathf.Log(max / min);  
        float b = x / (maxLevel - 1);  
        float y = (Mathf.Exp(b) - 1);  
        float a = min / y;  
        int newStat = (int)(a * Mathf.Exp(((float)b * position)));  
        newStat = (int)Mathf.Round((float)newStat / 10.0f) * 10;  
        return newStat;  
    }  
}
```

This script is called whenever a player has enough experience to level up via the `LevelUpCharacter()` function. This function first checks the current level of the character that corresponds to this class before increasing it by 1 and calculating the new stats.

The red region defines the maximum player level, the amount of experience required for the first level up and the amount of experience required to achieve max level. This is used for the interpolation in the `CalculateNewStats()` function.

The blue region shows that the characters' stats in the `GameInformation` script will be replaced by values calculated in the `CalculateNewStats()` function.

The purple regions shows that the CalculateNewStats() function calculates the stats the player should have at their current level by interpolating between the base and max stats for their class (See 3.3.). This is done with an exponential function so that level-ups provide an increasing number of stat points. This can be tested with the TestLevelUp() function and kind of works but seems to be a little fucked.

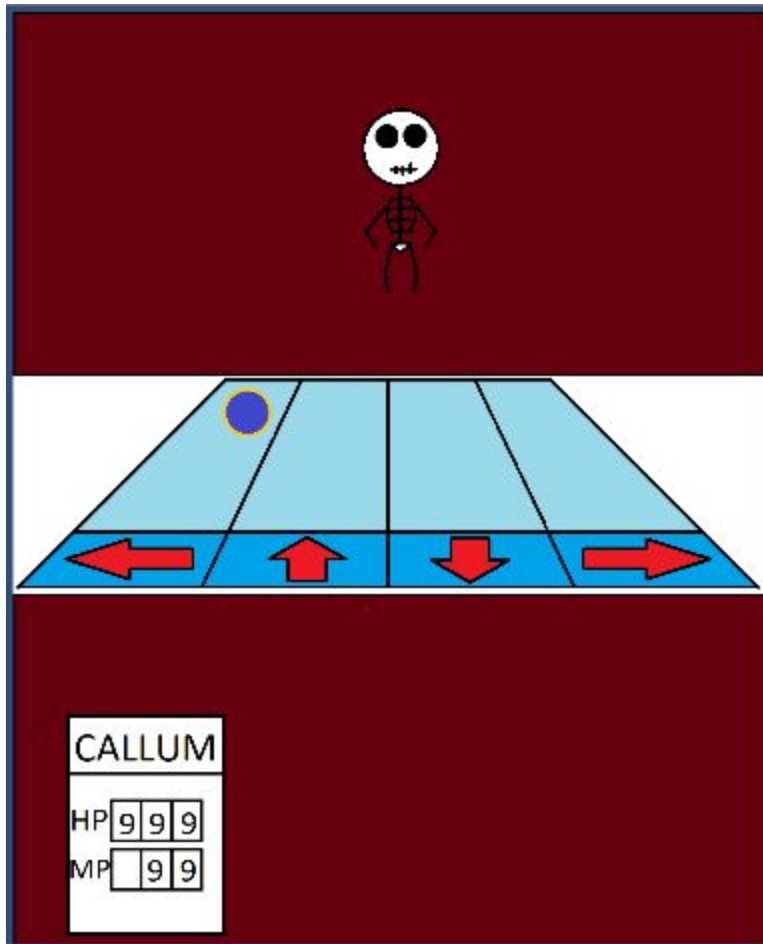
Money.cs:

```
public class Money
{
    public static void IncreaseMoney(int Money)
    {
        GameInformation.Money += Money;
    }
    public static void DecreaseMoney(int Money)
    {
        if (Money >= GameInformation.Money)
        {
            GameInformation.Money -= Money;
        }
    }
}
```

This script contains two functions; IncreaseMoney() which is called whenever the player earns money (Beating an enemy, selling an item, completing a quest etc.) and just adds the specified amount to the Game Information Script.

The DecreaseMoney() function is called whenever the player tries to buy an item, it first checks whether they have enough money before subtracting the specified amount.

6. Battle:



This is currently what the player sees when they enter a battle. These graphics don't do anything other than represent what a battle in-game may look like. We have the enemy sprites up the top, the battle stage in the middle and player stats down the bottom.

During the player turn, they input commands from a list and deal damage to the enemy, while on the enemies' turn the player will play a short, fast-paced rhythm game which represents the enemies attack.

6.1. TurnBasedBattle.cs:

```
public class TurnBasedBattle : MonoBehaviour
{
    public BaseEnemy Enemy1;
    public BaseEnemy Enemy2;
    public BaseEnemy Enemy3;

    public enum Battle //The possible states of the battle
    {
        Start,
        PlayerTurn,
        EnemyTurn,
        Lose,
        Win
    }

    private Battle currentState; //The current state of the battle

    // Use this for initialization
    void Start()
    {
        currentState = Battle.Start;
        Debug.Log(Enemy1.EnemyClass.EnemyName);
    }
}
```

This script governs game behaviour during the battle scene. The red region shows that we have an enumerator that corresponds to the different stages of the battle we could be in. So when the battle start the game sets up the battle and says “Oh shit boy, enemies appear”. Then you alternate between PlayerTurn and EnemyTurn until you either reduce all enemy HP to 0 (Triggers Win state) or have the HP of your party reduced to 0 (Triggers Lose State).

This script is linked to the camera object on the battle scene, and when initialised it will set the value of currentState to the Start state.

The purple object shows that there are three BaseEnemy objects for this script. They have stats corresponding to those defined in the EnterBattle.cs script. This doesn't really work at the moment.

```
private void CreatePlayer() //Creates new BasePlayers representing current party members
{
    BasePlayer Player1 = new BasePlayer();
    Player1.MaxHealth = GameInformation.Health1 + GameInformation.Equipment1Weapon.Health + GameInformation.Equipment1Armour.Health;
    Player1.Health = Player1.MaxHealth;
    Player1.MaxMagic = GameInformation.Magic1 + GameInformation.Equipment1Weapon.Magic + GameInformation.Equipment1Armour.Magic + G
    Player1.Magic = GameInformation.Magic1;
    Player1.Attack = GameInformation.Attack1 + GameInformation.Equipment1Weapon.Attack + GameInformation.Equipment1Armour.Attack +
    Player1.Defence = GameInformation.Defence1 + GameInformation.Equipment1Weapon.Defence + GameInformation.Equipment1Armour.Defence
}
```

During the Start state, a CreatePlayer() function is called which creates player objects and assigns them stats corresponding to their class, level and equipment. Pretty filthy coding atm.

6.2. BaseNote.cs:

```
public class BaseNote : MonoBehaviour {  
    public enum NoteColumns //The possible key  
    {  
        Left,  
        Up,  
        Down,  
        Right  
    }  
    private NoteColumns noteColumn; //Which key  
    private int speed; //The speed of the note  
    private int damage; //The base damage the  
    private int target; //The targeted player  
    private CircleCollider2D noteCollision; //  
  
    public NoteColumns NoteColumn  
    {  
        get { return noteColumn; }  
        set { noteColumn = value; }  
    }  
    public int Speed  
    {  
        get { return speed; }  
        set { speed = value; }  
    }  
}
```

A script which defines the characteristic of notes generated during the battle phase. Currently each note will be in a column, which corresponds to a certain key (Shown in the blue region) as well as a variety of integer stats that determine how fast the note travels and how much damage it does.

The red region shows that each note also has a CircleCollider2D, which is an automatically sized circular collision test around the object which enables it to detect collisions with other objects. Since the notes are going to be round, the collider should fit them almost perfectly.

As with our characters and items, we've used the get/set functions for these variables as well.