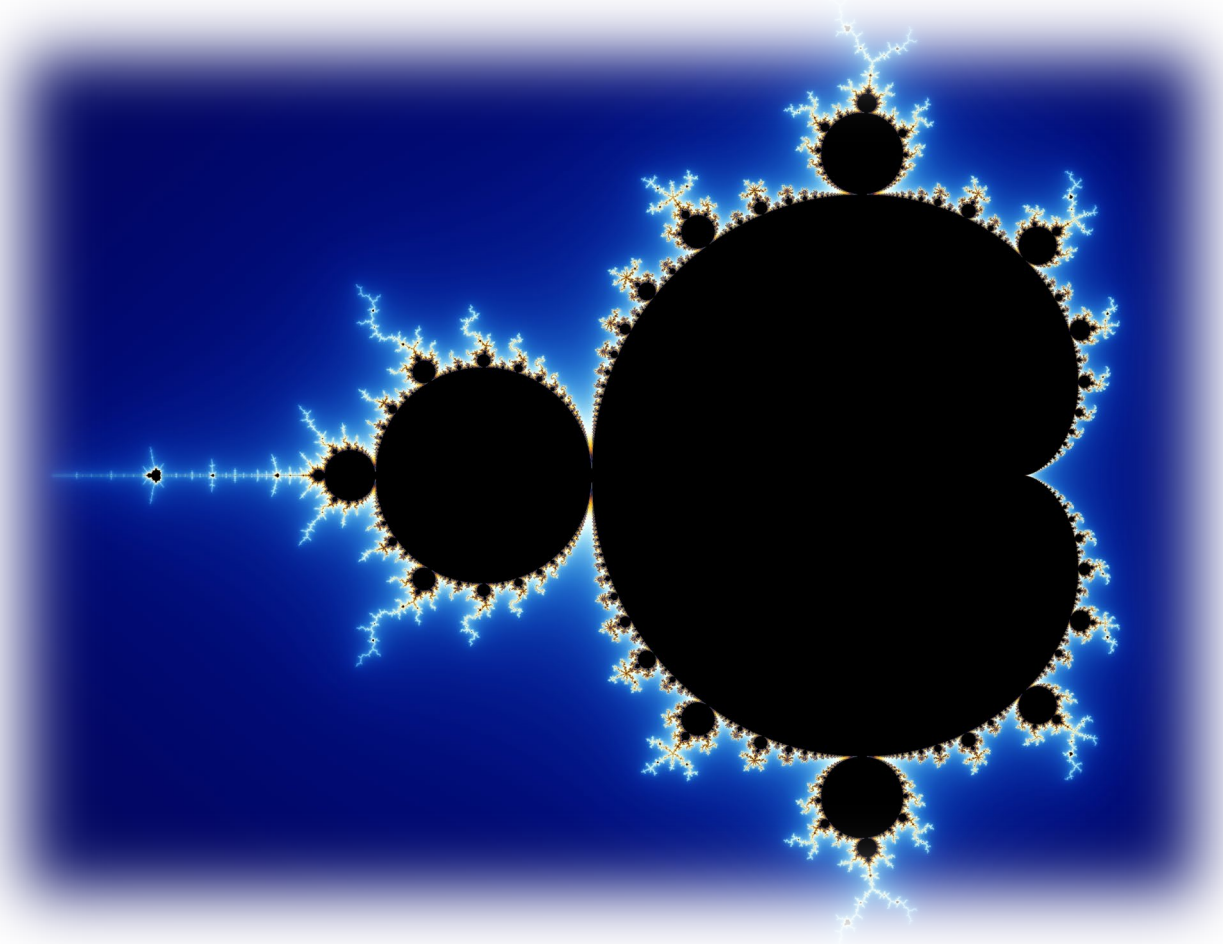


Object Oriented Programming

OOP's

Disclaimer

Object Oriented Programming (OOP) is hard even for seasoned programmers

It may take a while to appreciate why its worth the effort

The hierarchy may change as you understand more about the problem, so expect to be changing your code a lot

Once mastered it will make your life much easier

Player, Asteroid(s), later Bullet & UFO

In game they all have different behaviour

- Ship: Moves under player control, wraps screen, can be hit

- Big Rock: Auto rotates, moves with random velocity, wraps screen, when hit splits into 2 medium rocks, gives player 50 points

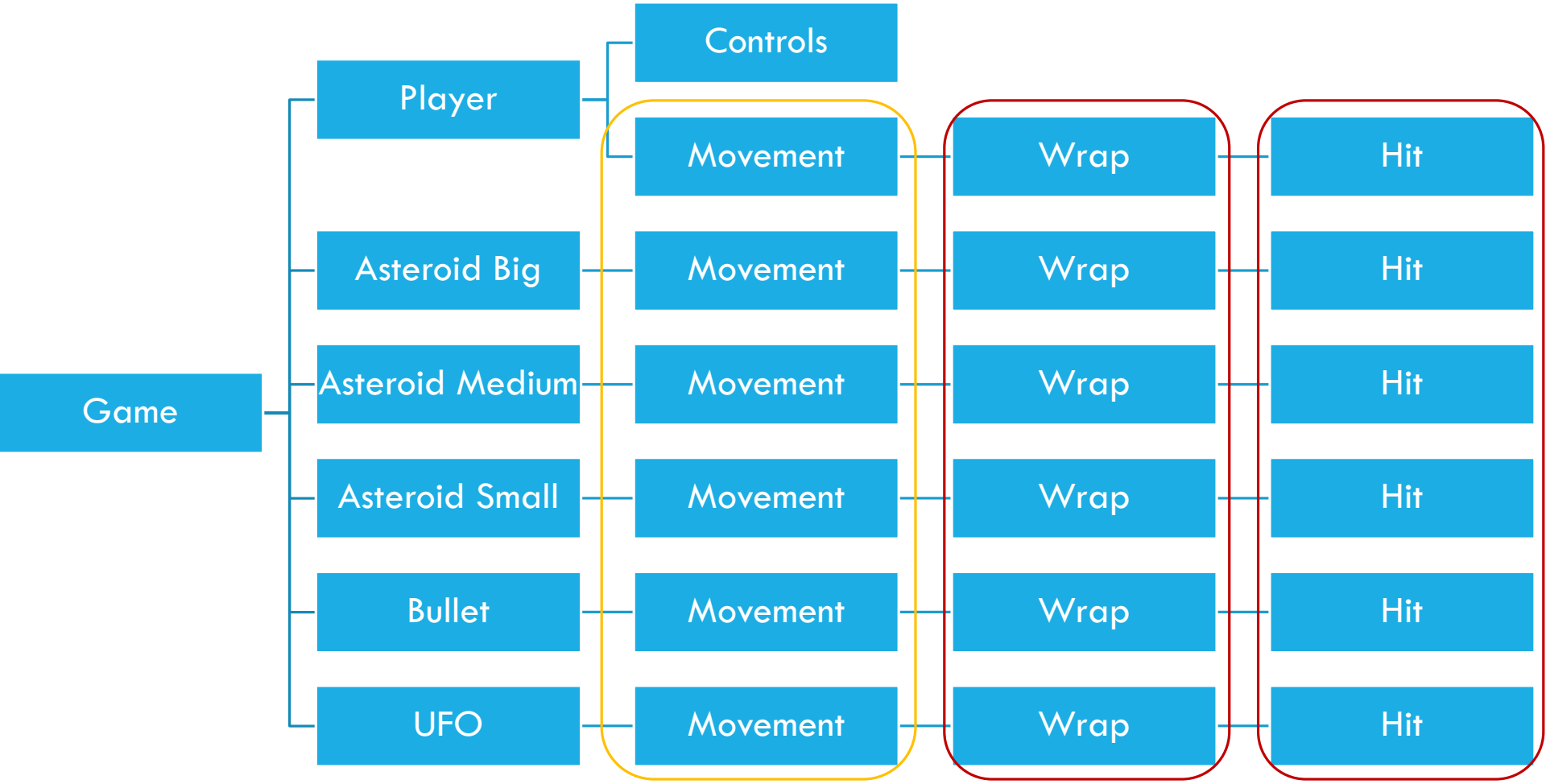
- Medium Rock: Auto rotates, moves with random velocity, wraps screen, when hit splits into 2 small rocks, gives player 100 points

- Small Rock: Auto rotates, moves with random velocity, wraps screen, when hit disappears, gives player 150 points

- Bullet: Moves in direction of ship when it was fired, wraps screen, dies after 5 seconds

- UFO: Moves with random velocity but changes direction randomly, wraps screen, when hit disappears, gives player 250 points

Our game world



We could have scripts for each Object

Each would contain all the functionality the Object needs

Warp()

Collision()

Start()

Update()

Velocity

Etc.

However its more effective to put shared behaviour in a base class

Should contain **common** functionality

Should allow default behaviour to be overridden

Should not contain anything too specific, which may only be used by some derived classes

Base class

Just a normal class template, which also inherit from another base class typically MonoBehaviour

Default behaviour - Can be instantiated

Can be **abstract**, which means only a derived version (unless also abstract) can be instantiated

Instantiate means use the class template to allocate one in memory

Derived class

Inherits behaviour from base class

Can selectively override base class behaviour

- Can choose to add own behaviour or totally replace it

If it overrides behaviour, it gets called rather than base class

Can only override functions (methods), getters & setters, **however not variables**

We have already met a good use of a base class

Collider2D

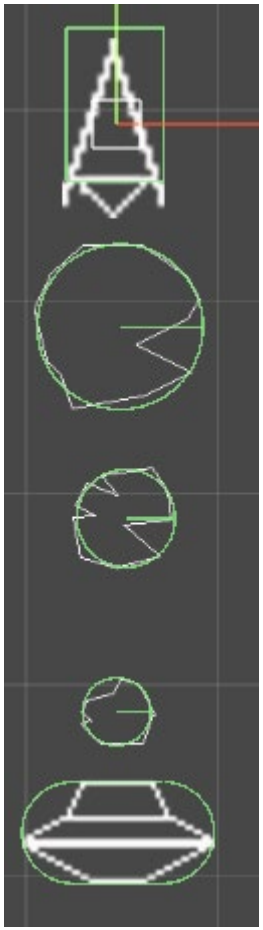
- BoxCollider2D
- CircleCollider2D
- CapsuleCollider2D

They are all children of Collider2D

- So they are all also all Colliders2D's

As each game object has a different shape, assigned in the IDE we do not want to hardcode the type of collider.

However as all Colliders2D have a common base class, and as long as we don't need to access any special properties found on the child only, we can use the base class to communicate with it.



As all Collider2D's can be triggers this property lives on the base class

So we can write generic code which does not care which type of collider we have attached

```
private Collider2D mCollider = null;        //Added in IDE but found with code
private Rigidbody2D mRB2 = null;    //We will set this up in code
private float mRotation = 0.0f;

void Start()
{
    mCollider = GetComponent<Collider2D>(); //Get BoxCollider2 added via IDE
    Debug.Assert(mCollider != null, "Please assign BoxCollider2D in IDE");

    mCollider.isTrigger = true;      //Set this in code in case we forget in IDE
```

Generic code is great because it can be easily reused

This will work for any Collider2D, so if you change your mind about the optimal shape, you code will stay the same

Objects

In C# an Object is a class

Classes combine;

1. Data (private, protected & public)
2. Functions (a.k.a) Methods

Classes can be children of other classes

1. A child class will usually inherit data & functions from its parent
   - In C# the overall parent of all classes is “Object”
   - Every bit of code we have written so far has been inherited from MonoBehaviour

MonoBehaviour

A base class, which contains code for

- Transforms
- GameObject
- Component system
- Memory management

We have already used this for every script so far

```
public class MovePlayer1 : MonoBehaviour
```

Designing a base class for our objects

1. Default Setup
2. Handle basic physics
3. Screen wrap
4. Deal with collisions
5. Allow for movement input

- MovePlayer1 already does a lot of this, so its an ideal candidate for a the starting point of a base class

FakePhysics.cs

Our base class for all moving objects

- It will have default behaviours for StartUp, Move, Wrap
- These can be overridden in any children to add new behaviour

1. The base class creates **virtual functions**
2. The child class inherits these and **overrides** them

- New keywords virtual & override
- New access modifier **protected**; it allows children to see the function /variable in the parent however hides it from any class which is not a child

Functional outline

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

public class FakePhysics : MonoBehaviour
{

    [SerializeField] //Show in IDE
    [Range(0.5f, 5f)]
    protected float MaxSpeed = 5.0f;

    [SerializeField] //Show in IDE
    protected float MaxRotation = 360.0f; //Now protected

    protected Vector2 mVelocity = Vector2.zero; //Now protected

    private Collider2D mCollider = null;        //Added in IDE but found with code
    private Rigidbody2D mRB2 = null;    //We will set this up in code

    protected virtual void Start() ...


    //Wrap Go position
    protected Vector2 WrappedPosition(Vector2 tCurrentPosition) ...

    protected virtual void DoMove() ...

    //Whilst we are using RB's we will do all the movement , as we don't use Physics we dont need FixedUpdate
    void Update() {
        DoMove(); //Move Object
    }
}
```

FakePhysics.cs works with PlayerFP.cs

PlayerFP.csinherits from FakePhysics now, rather than MonoBehaviour

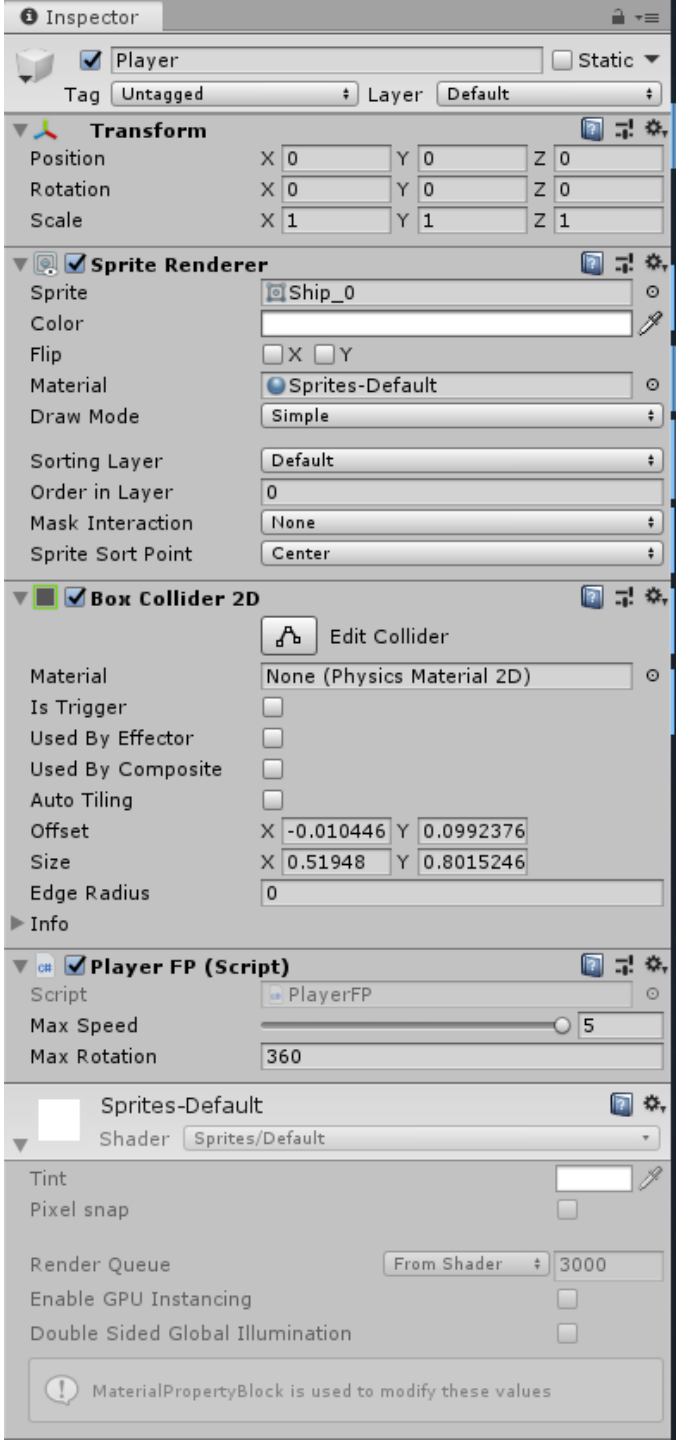
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerFP : FakePhysics
{
    protected override void DoMove() {
        Vector2 tMoveInput = new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")); /

        transform.Rotate(0, 0, -tMoveInput.x * MaxRotation * Time.deltaTime);

        //Rotate ship on player command, note axis reversed
        Vector2 tForce = transform.up;  //Up vector is now direction we are pointing in
        tForce *= MaxSpeed * tMoveInput.y; //Apply User input & MaxSpeed
        mVelocity += tForce;    //Calculate new velocity, scale for time
        base.DoMove(); //Call base class to update position
    }
}
```

It contains all the code which only the player needs, FakePhysics.cs deals with velocity & Wrapping

```
protected virtual void DoMove() {
    transform.position += (Vector3)mVelocity * Time.deltaTime;
    transform.position = WrappedPosition(transform.position); ,
}
```

Inside the IDE

PlayerFP.csalso has access to public/protected functions and variables found in FakePhysics.cs



Adding collision

The base class deals with the Unity collision callback

```
private void OnTriggerEnter2D(Collider2D collision) {
    FakePhysics tFF = collision.GetComponent<FakePhysics>(); //Get FakePhysics component
    Debug.AssertFormat(tFF != null, "Other object {0} is not FakePhysics",collision.name); //Ensure its valid
    CollidedWith(tFF);  //Pass to collision handler
}

//Default Collision handler
protected virtual void CollidedWith(FakePhysics vOtherFF) {
    Debug.LogFormat("Collision between {0} and {1}", name, vOtherFF.name); //Print Message
}
```

After verification it turns it into a FakePhysics friendly versions, with a default handler

In PlayerFP.cs

We can override this for the Player ONLY to do specific stuff

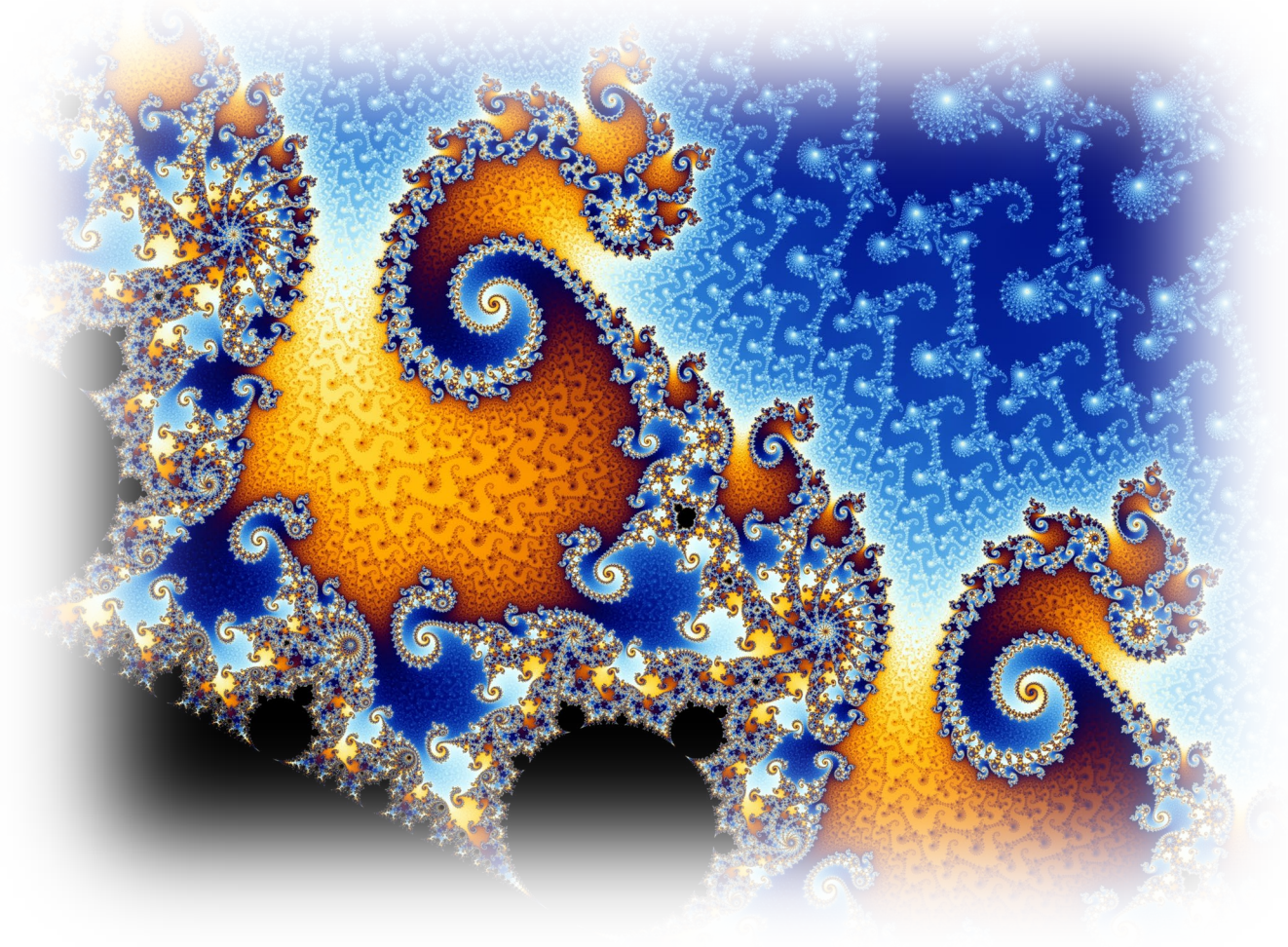
```
protected override void CollidedWith(FakePhysics vOtherFF) {
    Debug.LogFormat("Player hit by {0}", vOtherFF.name); //Player specifc code
    //We do not call parent as we will handle
}
```

Unlike DoMode() we do not call the parent as we want to fully override the behaviour

Adding a Rock

We simply define the rock behaviour by its differenced to FakePhysics

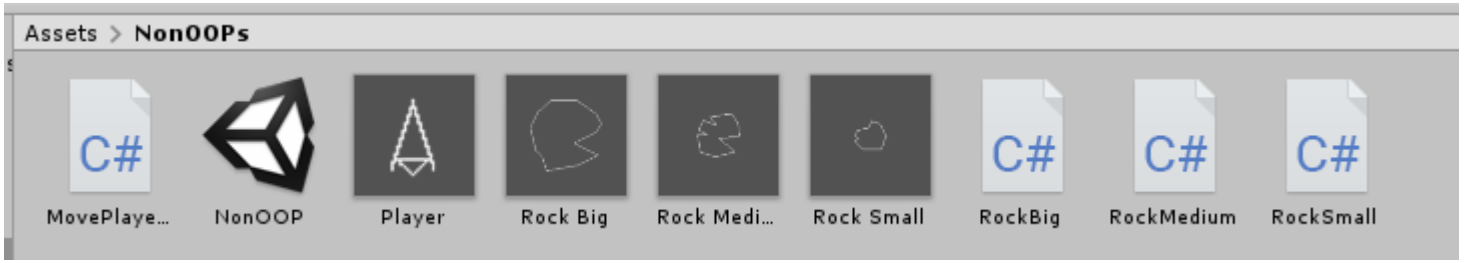
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RockBigFP : FakePhysics {
    // Start is called before the first frame update

    float mRotation;

    protected override void Start() {
        mVelocity = new Vector2(Random.Range(-1.0f, 1.0f), Random.Range(-1.0f, 1.0f)).normalized*MaxSpeed;
        mRotation = Random.Range(-4.0f, 4.0f);
        base.Start(); //Now call parent
    }

    //Rock's own movement
    protected override void DoMove() {
        transform.Rotate(0, 0, -mRotation * MaxRotation * Time.deltaTime);
        base.DoMove();
    }

    protected override void CollidedWith(FakePhysics vOtherFF) {
        Debug.LogFormat("RockBigFP hit by {0}", vOtherFF.name); //RockBigFP specifc code
        //We do not call parent as we will handle
    }

}
```



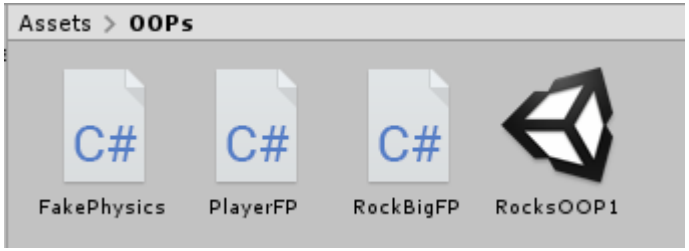
Workshop

Review the RocksOOP1 package

It contains non OOP code



And the OOP version



Review the scripts and note how the OOP version reduces cut & paste code, consider why this is a benefit?

Review the BigRockFP.cs script

1. Add a MediumRockFP.cs script which makes a rock which moves twice as fast and rotate 1.5 times faster then the ones using BigRockFP, create a medium rock object and ensure it has an appropriate collider and test to make sure there are no errors
2. Add a SmallRockFP.cs script which makes rock which moves 1.5x as fast and rotates 1.5 times faster then the ones using MediumRockFP, create a small rock object and ensure it has an appropriate collider and test to make sure there are no errors
3. Harder: Add a UFO script which changes direction every 1-5 seconds, add collider & test