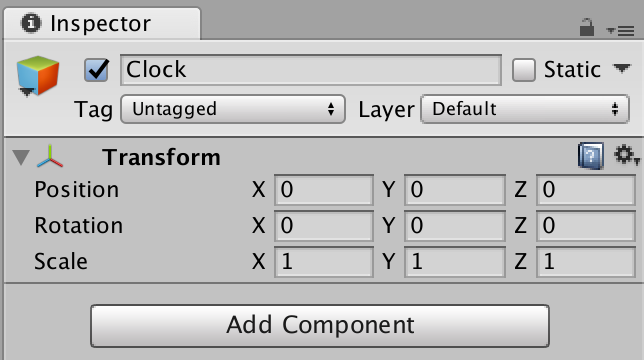
# Basic Unity

## Transform

There is always a Transform component, which is all our clock currently has.

Inspector window with clock selected.

The *Transform* component contains the position, rotation, and scale of the object in 3D space. Make sure that the clock's position and rotation are both 0. Its scale should be 1.

# Unity Coding

## Escape Characters

1. \n New line
2. \t Tab
3. \v Vertical Tab
4. \b Backspace
5. \r Carriage return
6. \f Formfeed
7. \\ Backslash
8. \' Single quotation mark
9. \" Double quotation mark
10. \d Octal
11. \xd Hexadecimal
12. \ud Unicode character

## What's a class, technically?

You can think of a class as a blueprint that can be used to create objects that reside in a computer's memory. The blueprint defines what data these objects can contain and what functionality they have.

Classes can also define data and functionality that don't belong to objects, but to the class itself. This is often used to provide globally-available functionality.

To turn Clock into a subtype of MonoBehaviour, we have to change our type declaration so that it extends that type, which is done with a colon. This makes Clock inherit all the functionality of MonoBehaviour.

public class Clock : MonoBehaviour {}

However, this will result in an error after compilation. The compiler complains that it cannot find the MonoBehaviour type. This happens because the type is contained in a namespace, which is UnityEngine. To access it, we have to use its fully-qualified name, UnityEngine.MonoBehaviour.

## What is a structure?

A structure is a blueprint, just like a class. The difference is that whatever it creates is treated as a simple value, like an integer or color, instead of an object. It has no sense of identity. Defining your own structure works the same as defining a class, except you use struct instead of class.

## Class vs Structure

**Classes** are reference types, and thus multiple variables can all reference and affect the same object

**Structs** are value types, and thus each variable contains it’s own copy of the values

The ‘ref” and ‘out’ keywords can be used to force any modification made to the variable inside a method, to also affect the original variable outside of that method. This is applicable to both reference types and value types.

‘ref’ requires the variable to already be initialized before entering the method while ‘out’ doesn’t

The built-in datatypes (int, float, char, bool, etc.) are all value types, but string is a reference type, which nevertheless behaves like a value type

## What's a property?

A property is a method that pretends to be a field. It might be read-only or write-only. The convention is to capitalize properties, but Unity often doesn't do this.

## What's a namespace?

A namespace is like a website domain, but for code. Just like domains can have subdomain, namespaces can have subnamespaces. The big difference is that it's written the other way around. So instead of forum.unity3d.com it would be com.unity3d.forum. The code comes with Unity, you don't have to go online to fetch it separately. Namespaces are used to organize code and prevent name clashes.

## What's a quaternion?

Quaternions are based on complex numbers and are used to represent 3D rotations. While harder to understand than simple 3D vectors, they have some useful characteristics. For example, they don't suffer from gimbal lock.

UnityEngine.Quaternion is used as a simple value. It is a structure, not a class.

## What's local about the rotation?

localRotation refers to the actual rotation of a transform component, independent of the rotation of its parents. In other words, it is the rotation in the object's local space. It's what gets displayed in its transform component in the inspector. So if we were to rotate the clock's root object, its arms would rotate along with it, as we would expect.

There is also a rotation property. It refers to the final rotation of a transform component in world space, taking the transformations of its parents into account. Had we used that, the arms would not adjust when we rotate the clock, as its rotation will be compensated for.

## Instantiate

The Instantiate method gives us a reference to whatever it created. Because we gave it a reference to a Transform component, that's what we get in return. Let's keep track of it with a variable.

void Awake () {

Transform point = Instantiate(pointPrefab);

}

Now we can adjust the point's position, by assigning a 3D vector to it.

## Vector3

3D vectors are created with the Vector3 struct. As it's a struct, it acts like a value, similar to a number, not an object. For example, let's set the X coordinate of our point to 1, leaving its Y and Z coordinates at zero. Vector3 has a right property for this.

Transform point = Instantiate(pointPrefab);

point.localPosition = Vector3.right;

## Can you multiply structs and numbers? (Syntactic Sugar)

Normally you cannot, but it is possible to define such functionality. This is done by creating a method with a special syntax, so it can be invoked as if it were a multiplication. In this case, what appears to be a simple multiplication is actually a method invocation, something like Vector3.Multiply(Vector3.right, 2f).

Being able to use methods as if they were simple operations makes writing code faster and easier to read. It is not essential, but nice to have, just like being able to implicitly use namespaces. Such convenient syntax is known as syntactic sugar.

Having said that, methods should only be used as operators if they strictly match the original meaning of that operator. In the case of vectors, some mathematical operators are well-defined, so it's fine for those.

## Range

Range is an attribute type defined by Unity. An attribute is a way to attach metadata to code structures, in this case a field. Unity's inspector checks whether a field has a Range attribute attached to it. If so, it will use a slider instead of the default input field for numbers. However, to do this it needs to know the allowed range. So Range has two parameters, for the minimum and maximum value.

## SetParent

We can set up this relationship after instantiating a cube, by invoking the **SetParent** *method* of the **cube**(prefab)'s **Transform** *component*. We have to supply it the **Transform** *component* of its new *parent*. We can directly access the object's **Transform** *component* via its **transform** *property*, which **Graph(**class name**)** inherited.

prefab.SetParent(transform);

## Couldn't we directly assign to point.localPosition.y?

**for** (**int** i = 0; i < points.Length; i++) {

[**Transform**](http://docs.unity3d.com/Documentation/ScriptReference/Transform.html) point = points[i];

[**Vector3**](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html) position = point.localPosition;

position.y = position.x \* position.x \* position.x;

point.localPosition = position;

}

If localPosition were a field, then this would be possible. We could directly set the Y coordinate of the point's position. However, localPosition is a property. It passes a vector to us, or accepts one from us. So we'd end up adjusting a local vector value, which doesn't affect the point's position at all. As we haven't explicitly stored it in a variable first, the operation is meaningless and will produce a compiler error.

## What's Mathf?

It is a struct that contains a collection of mathematical functions and constants for working with numbers and vectors. As it works with floating-point numbers, it was given the f suffix.

## VAR

it is redundant to explicitly declare the variable's type as well. Instead, we can use the var keyword. This implicitly declares the variable's type to match whatever is immediately assigned to it, which is something that the compiler can figure out in this case.

**void** Save () {

**var** writer = **new** [**BinaryWriter**](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create));

}

We now have a writer variable that references a new binary writer. Its type is obvious.

## When should var be used?

The **var** keyword is syntactic sugar that you don't need to use at all. While you could use it everywhere that the compiler can infer which type is meant, it's better to only do this when readability is improved and types are explicit. I only use **var** in these tutorials when a variable is declared and immediately assigned to, using the **new** keyword. So only in expressions of the form **var** t = **new** Type.

The **var** keyword is very useful when working with Language Integrated Query (LINQ) and anonymous types, but that's outside the scope of these tutorials.

## Static methods

By default, methods and fields are associated with specific object or value instances of a class or struct type. But this need not be the case. We can indicate that this association doesn't exist. That's done by putting the static keyword in front of the method or field definition.

## What's the point of making our methods static?

Because static methods aren't associated with object instances, the compiled code doesn't have to keep track of which object you're invoking the method on. This means that static method invocations are a bit faster, but it's usually not significant enough to worry about.

A static variable’s value is same for all instances of the script (eg for all objects in a scene with the script attached)

## Delegates

A simple if-else block works for two functions, but it gets unwieldy fast when trying to support more functions. It would be much more convenient if we could use a variable to store a reference to the method that we want to invoke. This is possible, by using a delegate type. A delegate is a special type that defines what kind of method something can reference.

Do we have to use a new script?

It's actually possible to define the delegate type inside Graph, but putting each type in its own script makes it explicit that they are separate things. In bigger projects, small types that are only used in the context of another type are often nested inside those types.

**public** **delegate** **float** **GraphFunction** (**float** x, **float** t);

Calling:

You can call the functions stored in a delegate by writing: myDelegate.Invoke(parameters); which has a shortcut as myDelegate(parameters)

Example of Use:

You can use delegates as parameters in functions so you can call them and different functions are called in the parent functions, depending upon the value of the delegate parameter. Eg.

**delegate int myD(PlayerStats stats);**

**void onStart()**

**{**

**int x;**

**x = Foo(Bar);**

**x = Foo(Sauce);**

**}  
int Foo(myD foo)**

**{**

**return foo(stats);**

**}**

**int Bar(PlayerStats stats)**

**{**

**return stats.kills;**

**}**

**int Sauce(PlayerStats stats)**

**{**

**return stats.sauce;**

**}**

U can add or remove functions from delegates, called ‘Subscribing’ and ‘Unsubscribing’; by using symbols ‘+=’ and ‘-=’.

## Events

Lies under namespace “System”.

The “event” keyword makes a delegate to become only subscribable and unscribable from classes other than where it is declared.

So u can “call it” or “give it a completely new value” only within the class where it is declared.

## Actions and Funcs

They are like shorthand delegates for creating events.

Used as:

**public event Action someEvent;**

Action: Delegate with void return type. No parameters

Action<T>, Action<T1, T2, …>: Delegate with void return type, custom parameters

Func<T>: Delegate with custom return type, no parameters

Func<T1, T2>, Func<T1, T2, T3, …>: Delegate with custom return type, custom parameters; here the last type specified is used as the return type.

Eg. Func<int, string, bool> myD;

Is equivalent to: delegate bool myD(int a, string b);

myD myDelegate;

## Lambda Expressions

In the above example, we can replace the function

**myD x=Sauce(stats);**

**int Sauce(PlayerStats stats)**

**{**

**return stats.sauce;**

**}**

With:

myD x= stats => stats.sauce;

where ‘=>’ is known as the Lambda Expression, and has syntax:

(input of function) => (output of function)

## How do arrays work?

Arrays are objects of fixed length that contain a linear sequence of variables. When declaring a variable, putting square brackets behind its type indicates that you want an array of that type. So int myVariable; gets you an integer, while int[] myVariable; get you an array of integers.

IMPORTANT: ALWAYS INITIALIZE ARRAYS (var array= new var[<size>]); OR ASSIGN VALUES (if public, then from Unity; else privately) Else, u may get NullReference errors

Accessing one of the entries inside an array is done by putting its array index – not its position – between square brackets behind the variable. So myVariable[0] gets you the first entry in the array, myVariable[1] gets you the second, and so on.

Actually creating an array and assigning it to the variable is done with myVariable = new int[10]; which in this case creates a new array with room for ten entries. Alternatively, you can create one implicitly by listing its initial values between curly brackets, like myVariable = {1, 2, 3}; does.

Array.Length for 1d arrays; array.GetLength(*n*) for *n*-dimension arrays

## How do two-dimensional arrays work?

You can add a second dimension to an array by inserting a comma inside its brackets. You then also need to provide two indexes whenever you want to access one of the array's elements. This approach extends to higher dimensions as well.

## An Array of Delegates

Although we've moved the if-else block out of the loop, we still haven't eliminated it. We can get rid of it completely by replacing it with indexing an array. Now that we have a GraphFunction type, we can add a functions array field of this type to Graph.

**GraphFunction**[] functions;

We're always going to put the same elements in this array, so we can explicitly define its contents as part of its declaration. This is done by assigning an array element sequence, between curly brackets. The simplest is an empty sequence.

**GraphFunction**[] functions = {};

This means that we immediately get an array instance, but it is empty. Change this so it will contain reference both function methods, first SineFunction, followed by MultiSineFunction.

**GraphFunction**[] functions = {

SineFunction, MultiSineFunction

};

Because this array is always the same, there's no point to create one per graph instance. Instead, let's define it once for the Graph type itself, making it static like our function methods.

**static** **GraphFunction**[] functions = {

SineFunction, MultiSineFunction

};

## *List*

We could add an array field to <Game> and fill it with references, but we don't know ahead of time how many cubes will be created. Fortunately, the System.Collections.Generic namespace contains a [List](http://social.msdn.microsoft.com/search/en-us?query=List) class that we can use. It works like an array, except that its size isn't fixed.

How can the list's size be dynamic?

Internally, [List](http://social.msdn.microsoft.com/search/en-us?query=List) uses an array to store its contents, which it initializes at some size. Items added to the list get put in this array, until it is full. If more items are added, the list will copy the contents of the full array to a new larger array and uses that one from now on. We could do this array management manually, but [List](http://social.msdn.microsoft.com/search/en-us?query=List) takes care of it for us. Also, Unity supports [List](http://social.msdn.microsoft.com/search/en-us?query=List) fields just like it supports array fields. They're editable via the inspector, their contents are saved by the editor, and they survive recompilation while in play mode.

[List](http://social.msdn.microsoft.com/search/en-us?query=List) insists that we specify the type of its contents. [List](http://social.msdn.microsoft.com/search/en-us?query=List) is a generic type, which means that it acts like a template for specific list classes, each for a concrete content type. The syntax is [List](http://social.msdn.microsoft.com/search/en-us?query=List)<T>, where the template type T is appended to the generic type, between angle brackets. In our case the correct type is [List](http://social.msdn.microsoft.com/search/en-us?query=List)<[Transform](http://docs.unity3d.com/Documentation/ScriptReference/Transform.html)>.

[**List**](http://social.msdn.microsoft.com/search/en-us?query=List)<**[Transform](http://docs.unity3d.com/Documentation/ScriptReference/Transform.html)**> objects;

Like an array, we have to ensure that we have a list object instance before we use it. We'll do that by creating the new instance in the [Awake](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Awake.html) method. In the case of an array, we'd have to use new [Transform](http://docs.unity3d.com/Documentation/ScriptReference/Transform.html)[]. But because we're using a list, we have to use new [List](http://social.msdn.microsoft.com/search/en-us?query=List)<[Transform](http://docs.unity3d.com/Documentation/ScriptReference/Transform.html)>() instead. This invokes the special constructor method of the list class, which can have parameters, which is why we have to append round brackets after the type name.

IMPORTANT: ALWAYS INITIALIZE LISTS ( var listName= new List<var>() ); OR ASSIGN VALUES (if public, then from Unity; else privately) Else, u may get NullReference errors

Rob’s Tip:

“Huh, you could have avoided making so many lists by making lists of lists

Three dimensions: Region, habitability, index

Or even four, Region, ishabitable, object-type, index

So, list<list<list<list<GameObject>>>> PlanetEntity = new list<list<list<list<GameObject>>>>(); //[inner, medium, outer][habitable, uninhabitable][Bodies, liquids, props][index] So, PlanetEntity[0][1][1][n] returns the nth inner uninhabitable liquid

“

## Enumerations

It would be clearer if we had a dropdown list containing meaningful names. We can use an enumeration to achieve this.

Enumerations can be created by defining an **enum** type.

The minimal definition of an enumeration works the same as a **class**, except that **enum** is used instead of **class**.

**public** **enum** **GraphFunctionName** {}

The block after the enumeration's name contains a comma-separated list of labels. These are strings that follow the same rules and conventions as type names. As names for our functions, use Sine and MultiSine.

**public** **enum** **GraphFunctionName** {

Sine,

MultiSine

}

Enumerations can be considered syntactical sugar. By default, each label of the enumeration represents an integer. The first label corresponds to 0, the second label to 1, and so on. So we can keep using the enumeration field to index our array. However, the compiler will complain that an enumeration cannot be implicitly cast to an integer. We have to explicitly perform this cast when using it as an index in [Update](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html).

**GraphFunction** f = functions[(**int**)function];

## Why use \*= 0.5f instead of /= 2f?

Both approaches are mathematically equivalent, but multiplication instructions are quicker than division instructions. If you're performing a lot of calculations inside loops, it's a simple optimization to make. It's not necessary for this tutorial, but it's a fine habit. You can go ahead and replace the division in MultiSineFunction too.

## What's an enumerator?(CoRoutines)

Enumeration is the concept of going through some collection one item at a time, like looping over all elements in an array. An enumerator – or iterator – is an object that provides an interface for this functionality. **System.Collections.IEnumerator** describes such an interface.

Why do we need this? Because coroutines use them. This is also why Unity includes System.Collections in their default script template, and why I included it as well.

## Is Initialize invoked before Start?

Yes, it is. First the new game object is created. Then a new <Object> component is created and added to it. At this point its **Awake** and **OnEnable** methods would be invoked, if they had existed. Then the **AddComponent** method finishes. Directly after that we invoke **Initialize**. The call to **Start** won't happen until the next frame.

## What does return do?

You use the return keyword to indicate that a method is finished and what its result is. What you return must match the type of the method. If it's a void method then you simply return nothing.

It's not needed to have a return statement at the end of a void or a special constructor method, for all other methods it's required.

It is possible to have multiple return statements inside a method. In that case there are multiple possible exit points. You'd typically use if statements to determine which return gets used.

## What does yield do?

The yield statement is used by iterators to make life easy for them. To make enumeration possible, you'd need to keep track of your progress. This involves some boilerplate code that is essentially always the same. What you'd really want is to just write something like return firstItem; return secondItem; until you are done. The yield statement allows you to do exactly that.

So whenever you're using yield, an enumerator object is created behind the scenes to take care of the tedious bits.

By the way, you can also yield another iterator. In that case this other iterator will be processed completely, so you can stitch them together in creative ways.

Uses: yield return new WaitForSeconds(*n*); =>pause coroutine for n secs

Yield return null => pause coroutine till next frame

yield return StartCoroutine(DoSomething()); => pause Coroutine till “DoSomething” has finished running

IMP: To stop a coroutine; ONLY NAME WONT WORK;

U’ll need to a reference to it to pass into “StopCoroutine” method:

IEnumerator currentCoroutine = DoSomething();

StartCoroutine(currentCoroutine);

StopCoroutine(currentCoroutine);

## How do coroutines work?

When you're creating a coroutine in Unity, what you're really doing is creating an iterator. When you pass it to the StartCoroutine method, it will get stored and gets asked for its next item every frame, until it is finished.

The yield statements produce the items. The statements in between – the stuff that you want to happen – are side-effects of the iterator doing its job.

You can yield special things like WaitForSeconds to have more control over when your own code continues, but the overall approach is simply that of an iterator.

## How does the random range work?

**Random** is a utility class that contains some stuff to create random values. Its Range method can be used to generate a random value within some range.

There are two versions of the *Range* method. You can call it with two floats, in which case it returns a float between the minimum and maximum value, both inclusive.

Alternatively, you can call Range with two integers, in which case it returns an integer between the minimum, inclusive, and maximum, exclusive. The typical use case for this version is selecting an index at random, like someArray[Random.Range(0, someArray.Length)].

## What does Lerp do?

**private** **void** [**Start**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Start.html) () {

gameObject.AddComponent<**[MeshFilter](http://docs.unity3d.com/Documentation/ScriptReference/MeshFilter.html)**>().mesh = mesh;

gameObject.AddComponent<**[MeshRenderer](http://docs.unity3d.com/Documentation/ScriptReference/MeshRenderer.html)**>().material = material;

[**GetComponent**](http://docs.unity3d.com/Documentation/ScriptReference/Component.GetComponent.html)<**[MeshRenderer](http://docs.unity3d.com/Documentation/ScriptReference/MeshRenderer.html)**>().material.color =

[**Color**](http://docs.unity3d.com/Documentation/ScriptReference/Color.html).Lerp(**[Color](http://docs.unity3d.com/Documentation/ScriptReference/Color.html)**.white, [**Color**](http://docs.unity3d.com/Documentation/ScriptReference/Color.html).yellow, (**float**)depth / maxDepth);

**if** (depth < maxDepth) {

StartCoroutine(CreateChildren());

}

}

Lerp is shorthand for linear interpolation. Its typical signature is Lerp(a, b, t) and it computes a + (b - a) \* t, with t clamped to the 0–1 range. Multiple versions exist for various types, including floats, vectors, and colors.

## *get* and *set* shorthands

### How does that property work?

Remember that properties are methods that pretend to be a field. We provide the FPS as public information, but only the component itself needs to update the value. The syntax used is shorthand notation for an automatically generated property, which would look something like this.

**int** fps;

**public** **int** FPS {

**get** { **return** fps; }

**private** **set** { fps = value; }

}

This shorthand doesn't work with Unity's serialization,

## Serializable (with eg)

As a final touch to the FPS labels, we can colorize them. This can be done by associating colors with FPS values. Such an association can be represented with a custom struct.

[System.[**Serializable**](http://social.msdn.microsoft.com/search/en-us?query=Serializable)]

**private** **struct** **FPSColor** {

**public** [**Color**](http://docs.unity3d.com/Documentation/ScriptReference/Color.html) color;

**public** **int** minimumFPS;

}

As FPSDisplay is the only thing that will use this structure, we put the struct definition directly inside that class and make it private so that it won't show up in the global namespace. Make it serializable so that it can be exposed by the Unity editor.

Now add an array of these struct so we can configure the coloring of the FPS labels. We'd typically add a public field for that, but we can't do that because the struct itself is private. So make the array private as well and give it the [SerializeField](http://docs.unity3d.com/Documentation/ScriptReference/SerializeField.html) attribute so Unity exposes it in the editor and saves it.

Loops:

For, while, do.. while, for each.

**For** loops are used when a specific number of iterations is required

**While** loops are used when the number of iterations is unknown

**Do..while** loops are like while loops but will always run atleast once

Foreach loops can be used to retireve elements 1 by1 from a collection(array, list, etc).

Syntax => **foreach (** **<**typeof**>** ***identifier\_variable****<*like “I” in for> **in** **<**collection\_identifier**>)**{}

**Continue** skips to the next iterationof the loop (skips remaining block of code)

**Break**  terminates the loop

# Maths:

## CreatingShapes Methods

### Cylinder

**static** [**Vector3**](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html) Cylinder (**float** u, **float** v, **float** t) {

[**Vector3**](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html) p;

**float** r = 1f;

p.x = r \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Sin(pi \* u);

p.y = v;

p.z = r \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Cos(pi \* u);

}

### Sphere

**static** [**Vector3**](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html) Cylinder (**float** u, **float** v, **float** t) {

**[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** p;

**float** r = **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Cos(pi \* 0.5f \* v);

p.x = r \* **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Sin(pi \* u);

p.y = **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Sin(pi \* 0.5f \* v);

p.z = r \* **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Cos(pi \* u);

**return** p;

}

### Torus

**static** **[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** Torus (**float** u, **float** v, **float** t) {

**[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** p;

**float** s = **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Cos(pi \* 0.5f \* v);

p.x = s \* **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Sin(pi \* u);

p.y = **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Sin(pi \* 0.5f \* v);

p.z = s \* **[Mathf](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html)**.Cos(pi \* u);

**return** p;

}

### Ring Torus

**static** **[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** Torus (**float** u, **float** v, **float** t) {

**[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** p;

**float** r1 = 1f;

**float** r2 = 0.5f;

**float** s = r2 \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Cos(pi \* v) + r1;

p.x = s \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Sin(pi \* u);

p.y = r2 \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Sin(pi \* v);

p.z = s \* [**Mathf**](http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html).Cos(pi \* u);

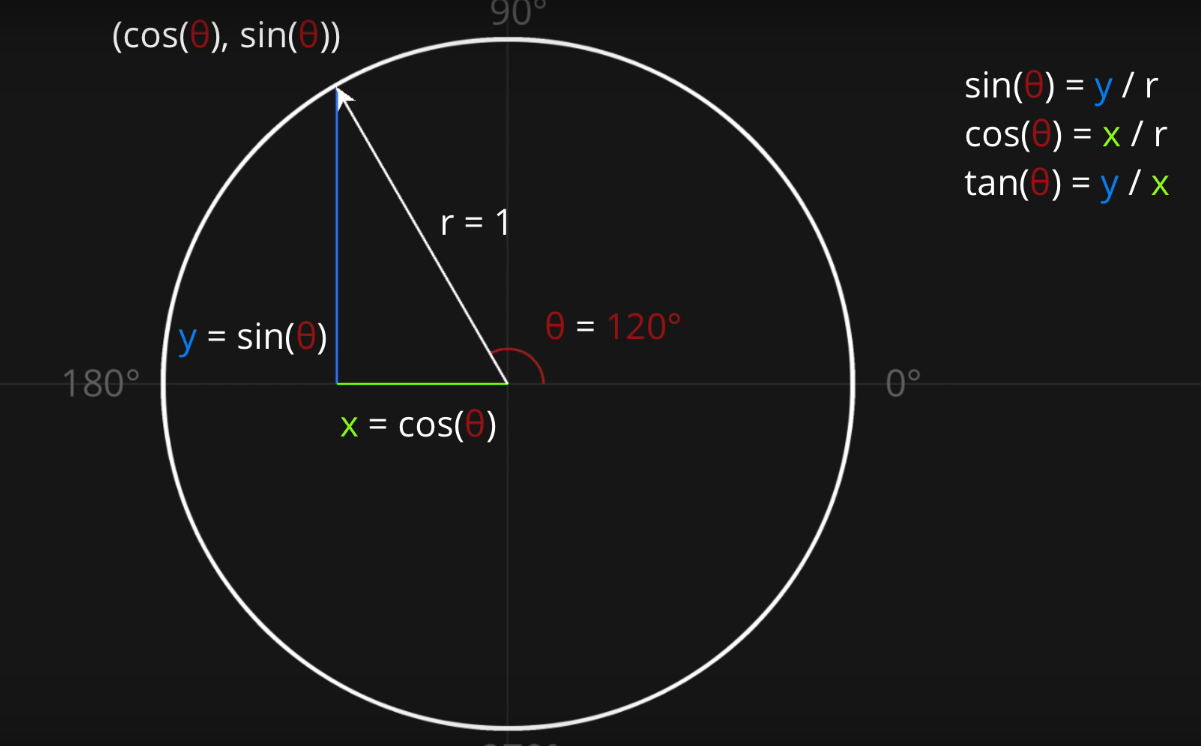
**return** p;

}

## Trigonometry

(Y: Sebastian Lague; search “trigonometry”

### Directions



Tip: In unity; to make directions in math same as “Rotations” in transform;

* do (90-theta) for theta being the angle; u get a clock-wise rotating circular path as value of theta increases
* or, just swap sin and cos: → x=sin(θ) and y=cos(θ)

# Shaders

## How do surface shaders work?

Unity provides a framework to quickly generate shaders that perform default lighting calculations, which you can influence by adjusting certain values. Such shaders are known as surface shaders.

## What's albedo and alpha?

The color of the diffuse reflectivity of a material is known as its albedo. Albedo is Latin for whiteness. It describes how much of the red, green, and blue color channels are diffusely reflected. The rest is absorbed.

Alpha is used as a measure of opacity. At alpha 0 a surface is fully transparent, while at alpha 1 it is fully opaque.

## Default color

A default color has all its four channels set to zero. This includes the alpha channel, which controls opacity. If you haven't changed the alpha channels, you'll get fully transparent onjects.

## Meshes

### Vertices and Triangles

A mesh in Unity is the shape of an object. It consists of a MeshFilter component, to contain the actual mesh, and a MeshRenderer component to Draw the mesh.

Any shape in the mesh is composed of triangles.

MeshRenderer.vertices is the array of positions of the vertices of the mesh

MeshRenderer.triangles is the array of triangle positions of a mesh. It is stored as index: 0,1,2 are the 3 vertices of a triangle, then 3, 4 and 5 the next vertices of a triangle and so on.

### How do normals work?

A normal is vector that is perpendicular to a surface. We always use normals of unit length and they point to the outside of their surface, not to the inside.

Normals can be used to determine the angle at which a light ray hits a surface, if at all. The specifics of how it is used depends on the shader.

As a triangle is always flat, there shouldn't be a need to provide separate information about normals. However, by doing so we can cheat. In reality vertices don't have normals, triangles do. By attaching custom normals to vertices and interpolating between them across triangles, we can pretend that we have a smoothly curving surface instead of a bunch of flat triangles. This illusion is convincing, as long as you don't pay attention to the sharp silhouette of the mesh.

### How do tangents work?

Normal maps are defined in tangent space. This is a 3D space that flows around the surface of an object. This approach allows us to apply the same normal map in different places and orientations.

The surface normal represents upward in this space, but which way is right? That's defined by the tangent. Ideally, the angle between these two vectors is 90°. The cross product of them yields the third direction needed to define 3D space. In reality the angle is often not 90° but the results are still good enough.

So a tangent is a 3D vector, but Unity actually uses a 4D vector. Its fourth component is always either −1 or 1, which is used to control the direction of the third tangent space dimension – either forward or backward. This facilitates mirroring of normal maps, which is often used in 3D models of things with bilateral symmetry, like people. The way Unity's shaders perform this calculation requires us to use −1.

## What is dynamic batching?

Dynamic batching is a form of draw call batching performed by Unity. In short, it combines meshes that share the same material into larger meshes. Doing so reduces the amount of communication between the CPU and the GPU. You can enable or disable it via Edit / Projects Settings / Player, in the Other Settings group.

It only works for small meshes. For example, you'll find that it works with Unity's default cube, but not with the default sphere.

# Physics

## RigidBody

**using** UnityEngine;

[**[RequireComponent](http://docs.unity3d.com/Documentation/ScriptReference/RequireComponent.html)**(**typeof**(**[Rigidbody](http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody.html)**))]

**public** **class** **Nucleon** : [**MonoBehaviour**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {

**public** **float** attractionForce;

[**Rigidbody**](http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody.html) body;

**void** [**Awake**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Awake.html) () {

body = [**GetComponent**](http://docs.unity3d.com/Documentation/ScriptReference/Component.GetComponent.html)<**[Rigidbody](http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody.html)**>();

}

**void** [**FixedUpdate**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html) () {

body.AddForce(transform.localPosition \* -attractionForce);

}

}

>RequireComponent= adds the components u denote

>AddForce: Is used as a property of *Rigidbody* Gameobject. adds force along ‘*a’.* : Syntax: AddForce(Vector3 a [,*Forcemode*])

>Transform.localPosition: returns position of object relative to world center.   
So this function pushes object toward center.

ALWAYS use with FixedUpdate🡪 better.

## ForceModes

*Acceleration*, *Force*, 🡪 Constant Force

*Impulse*, *Velocity*🡪 Instant Force (ball getting kicked)

## Why use FixedUpdate and not Update?

Using FixedUpdate keeps the spawning independent of the frame rate. If the configured time between spawns is shorter than the frame time, using Update would cause spawn delays. And as the point of this scene is to tank our frame rate, that will happen.

You could use a while loop instead of an if check to catch up on missed spawns, but this would cause infinite spawn loops when timeSinceLastSpawn is accidentally set to zero. Limiting spawning to once per fixed time step is a sane limitation.

# Player Input

## *KeyCode*

We're going to spawn cubes in response to player input, so our game must be able to detect this. We'll use Unity's input system to detect key presses. Which key should be used to spawn a cube? The C key seems appropriate, but we can make this configurable via the inspector, by adding a public KeyCode enumeration field to Game. Use C as the default option when defining the field, via an assignment.

public KeyCode createKey = KeyCode.C;

## When exactly does [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html).GetKeyDown return true?

We can detect whether the key is pressed by querying the static [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html) class in an [Update](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html) method. The [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html).GetKeyDown method returns a boolean that tells us whether a specific key was pressed in the current frame. If so, we have to instantiate our prefab.

**void** **[Update](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html)** () {

**if** (**[Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html)**.GetKeyDown(createKey)) {

**[Instantiate](http://docs.unity3d.com/Documentation/ScriptReference/Object.Instantiate.html)**(prefab);

}

}

It does so only during the frame that the key's state has changed from not-pressed to pressed, because the player pressed on it. Typically, the key remains in the pressed state for a few frames until the player lets go of the button, but [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html).GetKeyDown returns true only during the first frame. In contrast, [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html).GetKey keeps returning true each frame that the key is held down. There is also [Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html).GetKeyUp, which returns true during the frame that the player let go of the key.

## Why not just use [GameObject](http://docs.unity3d.com/Documentation/ScriptReference/GameObject.html).Find?

This is possible for simple cases, where it's easy to distinguish between objects and there aren't many in the scene. For larger scenes, relying on [GameObject](http://docs.unity3d.com/Documentation/ScriptReference/GameObject.html).Find is a bad idea. [GameObject](http://docs.unity3d.com/Documentation/ScriptReference/GameObject.html).FindWithTag is better, but it's best to keep track of things yourself if you know you'll need them later.

# Saving and Loading

## Intro

(Bascially everything from <https://catlikecoding.com/unity/tutorials/object-management/persisting-objects/>; 2.Saving and Loading)

To support saving a loading during a single play session, it would be sufficient to keep a list of transformation data in memory. Copy the position, rotation, and scale of all cubes on a save, and reset the game and spawn cubes using the remembered data on a load. However, a true save system is able to remember the game state even after the game is terminated. This requires the game state to be persisted somewhere outside the game. The most straightforward way is to store the data in a file.

### What about using [PlayerPrefs](http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html)?

As its name suggests, **[PlayerPrefs](http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html)** is designed with game settings and preferences in mind, not game state. While it's possible to pack game state in strings, this is inefficient, hard to manage, and doesn't scale.

## Save Path

Where game files should be stored depends on the file system. Unity takes care of the differences for us, making the path to the folder that we can use available via the **Application**.**persistentDataPath** *property*. We can grab the text string from this property and store it in a *savePath* *field* in *Awake*, so we need to retrieve it only once.

string savePath;

void Awake () {

savePath = Application.persistentDataPath;

}

This gives us the path to a folder, not a file. We have to append a file name to the path. Let's just use *saveFile*, not bothering with a file extension. Whether we should use a forward or backward slash to separate the file name from the rest of the path again depends on the operating system. We can use the **Path.Combine** *method* to take care of the specifics for us. **Path** is part of the **System.IO** *namespace*.

using System.Collections.Generic;

using System.IO;

using UnityEngine;

public class Game : MonoBehaviour {

…

void Awake () {

savePath = Path.Combine(Application.persistentDataPath, "saveFile");

}

…

}

## Opening a File for Writing

To be able to write data to our save file, we first have to open it. This is done via the [File](http://social.msdn.microsoft.com/search/en-us?query=File).Open method, providing it with a path argument. It also needs to know why we're opening the file. We want to write data to it, creating the file if it didn't already exist, or replacing an already existing file. We specify this by providing [FileMode](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create as a second argument. Do this in a new Save method.

**void** Save () {

**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, **[FileMode](http://social.msdn.microsoft.com/search/en-us?query=FileMode)**.Create);

}

[File](http://social.msdn.microsoft.com/search/en-us?query=File).Open returns a file stream, which isn't useful on its own. We need a data stream that we could write data into. This data has to be of a certain format. We'll use the most compact uncompressed format available, which is raw binary data. The System.IO namespace has the [BinaryWriter](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter) class to make this possible. Create a new instance of this class, using its constructor method, providing the file stream as an argument. We don't need to keep a reference to the file stream, so we can directly use the [File](http://social.msdn.microsoft.com/search/en-us?query=File).Open invocation as the argument. We do need to keep a reference to the write, so assign it to a variable.

**void** Save () {

**[BinaryWriter](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)** writer =

**new** **[BinaryWriter](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)**(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create));

}

We now have a binary writer variable named writer that references a new binary writer.

## Closing the File

If we open a file, we must make sure that we also close it. It's possible to do this via a Close method, but this isn't safe. If something goes wrong between opening and closing the file, an exception could be raised and execution of the method could be terminated before it got to closing the file. We have to carefully handle exceptions to ensure that the file is always closed. There is syntactic sugar to make this easy. Put the declaration and assignment of the writer variable inside round brackets, place the using keyword in front of it, and a code block after it. The variable is available inside that block, just like the iterator variable i of a standard for loop.

**void** Save () {

**using** (

**var** writer = **new** [**BinaryWriter**](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create))

) {}

}

This will ensure that whatever writer references will be properly disposed of, after code execution exist the block, no matter how. This works for special disposable types, which the writer and stream are.

## How does using work, without the sugar?

In our case, it would look like the following code.

**var** writer = **new** [**BinaryWriter**](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create);

**try** { … }

**finally** {

**if** (writer != **null**) {

((**[IDisposable](http://social.msdn.microsoft.com/search/en-us?query=IDisposable)**)writer).Dispose();

}

}

## Writing Data

We can write data to our file by invoking our writer's Write method. It is possible to write simple values, like a boolean, integer, and so on, one at at time. Let's begin by writing only how many objects we have instantiated.

**void** Save () {

**using** (

**var** writer = **new** [**BinaryWriter**](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter)(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Create))

) {

writer.Write(objects.Count);

}

}

To actually save this data, we have to invoke the Save method. We'll again control this via a key, in this case using S as the default.

**public** [**KeyCode**](http://docs.unity3d.com/Documentation/ScriptReference/KeyCode.html) createKey = [**KeyCode**](http://docs.unity3d.com/Documentation/ScriptReference/KeyCode.html).C;

**public** **[KeyCode](http://docs.unity3d.com/Documentation/ScriptReference/KeyCode.html)** saveKey = **[KeyCode](http://docs.unity3d.com/Documentation/ScriptReference/KeyCode.html)**.S;

…

**void** [**Update**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html) () {

**if** (**[Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html)**.GetKeyDown(createKey)) {

CreateObject();

}

**else** **if** (**[Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html)**.GetKey(newGameKey)) {

BeginNewGame();

}

**else** **if** (**[Input](http://docs.unity3d.com/Documentation/ScriptReference/Input.html)**.GetKeyDown(saveKey)) {

Save();

}

}

## Why not use BinaryFormatter?

While relying on BinaryFormatter can be convenient, it isn't possible to just serialize a game object hierarchy using a BinaryFormatter and deserialize it later. The game object hierarchy has to be recreated manually. Also, writing every bit of data ourselves gives us total control and understanding. Besides that, manually writing data requires less space and memory, is quicker, and makes it easier to support an evolving save file format. Sometimes, games that have already been released drastically change what's stored after an update or expansion. Some of those games can then no longer load a player's old save files. Ideally, a game remains backwards-compatible with all its save file versions.

## Loading Data

To load the data that we just saved, we have to again open the file, this time with [FileMode](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Open as the second argument. Instead of a [BinaryWriter](http://social.msdn.microsoft.com/search/en-us?query=BinaryWriter), we have to use a [BinaryReader](http://social.msdn.microsoft.com/search/en-us?query=BinaryReader). Do this in a new Load method, once again with a using statement.

**void** Load () {

**using** (

**var** reader = **new** **[BinaryReader](http://social.msdn.microsoft.com/search/en-us?query=BinaryReader)**(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, **[FileMode](http://social.msdn.microsoft.com/search/en-us?query=FileMode)**.Open))

) {}

}

The first thing we wrote to the file was the count property of our list, so that is also the first thing to read. We do this with the ReadInt32 method of our reader. We have to be explicit what we read, because there is no parameter that makes this clear. The suffix 32 refers to the size of the integer, which is four bytes, thus 32 bits. There are also larger and smaller integer variants, but we don't use those.

**using** (

**var** reader = **new** [**BinaryReader**](http://social.msdn.microsoft.com/search/en-us?query=BinaryReader)(**[File](http://social.msdn.microsoft.com/search/en-us?query=File)**.Open(savePath, [**FileMode**](http://social.msdn.microsoft.com/search/en-us?query=FileMode).Open))

) {

**int** count = reader.ReadInt32();

}

After reading the count, we know how many objects were saved. We have to read that many positions from the file. Do this with a loop, reading three floats per iteration, for the X, Y, and Z components of a position vector. A single-precision float is read with the ReadSingle method. A double-precision double would be read with the ReadDouble method.

**int** count = reader.ReadInt32();

**for** (**int** i = 0; i < count; i++) {

**[Vector3](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)** p;

p.x = reader.ReadSingle();

p.y = reader.ReadSingle();

p.z = reader.ReadSingle();

}

## What would happen if we loaded before saving anything?

Then you would try to open a file that doesn't exist, which would result in an exception.

# Virtual, Templates and Exceptions

## Example

using System;

using System.Collections;

using System.Collections.Generic;

using System.Drawing;

using System.Runtime.Serialization;

using UnityEngine;

public class Printer : MonoBehaviour

{

void Start()

{

//Virtual Function

Virtual[] obj= { new Derived1(), new Derived2()};

for(int i = 0; i < 2; i++)

{

obj[i].Printer();

}

//Template

var stack = new Template<int, int, int>(1);

//Exception Handling

try

{

stack.Push(1);

print(stack.pop());

print(stack.pop());

}

catch (FULL e)

{

print("Exception:" + e.Message);

}

catch (EMPTY e)

{

print("Exception:" + e.Message);

}

//Optional

/\*finally

\* {

\*

\* }\*/

}

};

public class Virtual

{

public virtual void Printer() { }

};

public class Derived1: Virtual

{

public override void Printer()

{

MonoBehaviour.print("This is Derived1");

}

};

public class Derived2: Virtual

{

public override void Printer()

{

MonoBehaviour.print("This is Derived2");

}

};

public class Template<S1, S2, S3>

{

public int top, size;

public S1 s1;

public S2[] s2;

public S3 s3;

//Exceptions

public Template(int size)

{

this.size = size;

top = -1;

s2 = new S2[this.size];

}

public void Push(S2 n)

{

if ((size - 1) == top)

{

throw new FULL("Stack is full");

}

else

{

s2[++top] = n;

}

}

public S2 pop()

{

if (top == -1)

{

throw new EMPTY("Stack is empty");

}

else

{

return s2[top--];

}

}

};

//Exceptions

[Serializable]

public class FULL : Exception

{

public FULL(string message) : base(message) { }

};

[Serializable]

public class EMPTY : Exception

{

public EMPTY(string message):base(message) { }

};