updated 2021-05-18 published 2017-09-08

Game Objects and Scripts Creating a Clock

Build a clock with simple objects.
Write a C# script. current：adj.当前的
Rotate the clock's arms to show the time.
Animate the arms. component：n.组成部分；成分；部件

tutorial: n.教程; adj.辅导的;导师的;私人教师的

This is the first tutorial in a series about learning the basics of working with Unity. In it we will create a simple clock and program a component to have it display the current time. You don't need to have any experience with the Unity editor yet, but you're assumed to have some experience with multi-window editor applications in general. assumed:adj.假设的 experience:n.经验

At the bottom of all my recent tutorials you'll find links to the tutorial license, a repository containing the finished tutorial project, and a PDF version of the tutorial page. repository：n.仓库

This tutorial is made with Unity 2020.3.6f1.

It is time to create a clock.

1 Creating a Project

Before we can start working with the Unity editor we must first create a project.

1.1 New Project

When you open Unity you will be presented with the Unity Hub. This is a launcher and installer app from where you can create or open projects, install Unity versions, and do some other things. If you don't have Unity 2020.3 or higher installed add it now.

appropriate:adj.合适的

Which Unity versions are appropriate?

release:v.释放;放出

Unity releases multiple new versions per year. There are two parallel release schedules. The most stable and safe are the LTS releases. LTS stands for long term support, which is two years in Unity's case. I stick to LTS versions for my tutorials. This tutorial uses 2020.3.6 specifically. The third portion of the version number indicates the patch release. Patch releases contain bug fixes and only rarely new functionality. A further f1 suffix indicates an official final release. Any 2020.3 version will do for this tutorial.

The highest Unity version is of the development branch, which introduces new features and possibly removes old functionality. These versions aren't as reliable as LTS versions and only remain supported for a few months each.

adv.偶尔地 vt.含有

Occasionally my tutorials contain little questions and their answers, always in a gray box, like the one above. On a web page the answer is hidden by default. This can be toggled by clicking or tapping the question.

n.样板

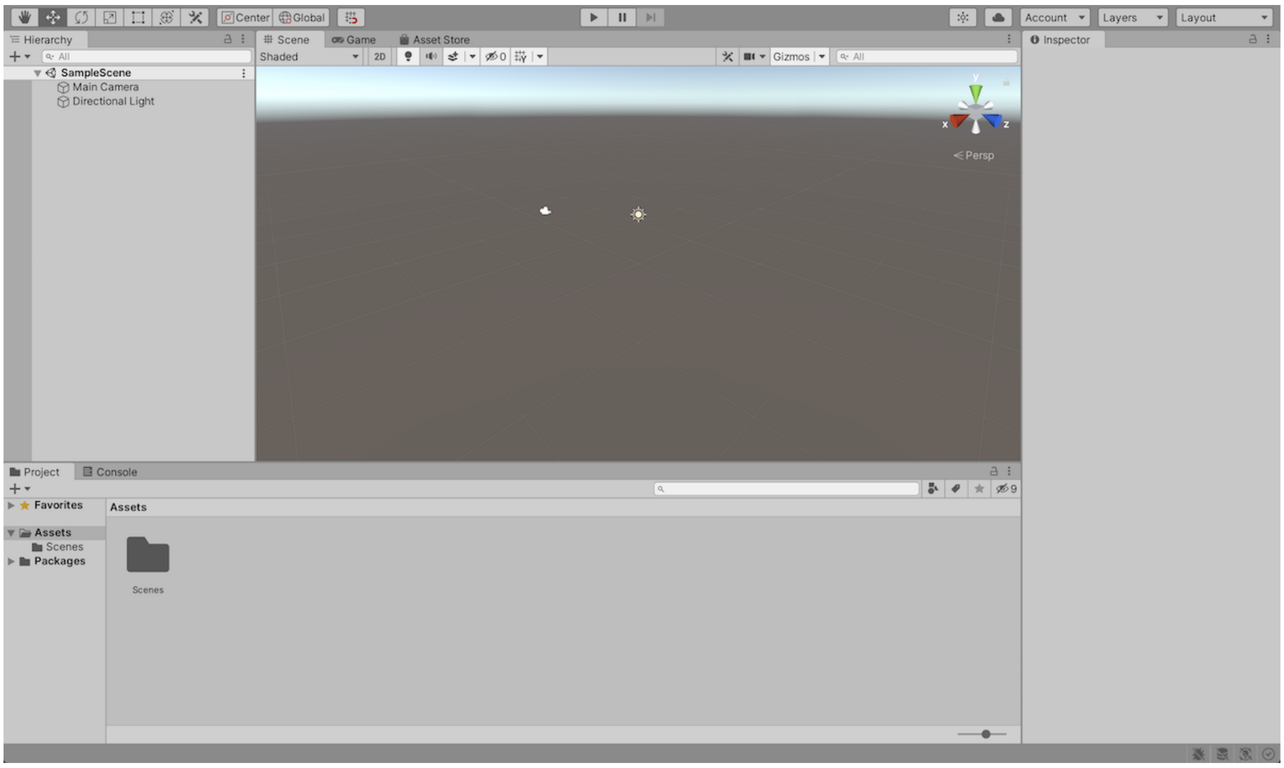
When you create a new project you get to pick its Unity version and a template. We'll use the standard 3D template. Once it's created it gets added to the list of projects and gets opened in the appropriate version of the Unity editor.

Can I create a project with a different render pipeline?

Yes, the only difference is that the project will have more things in its default scene and your materials will look different. Your project will also contain the appropriate packages.

1.2 Editor Layout

If you haven't customized the editor yet, you will end up with its default window layout.



Default editor layout.

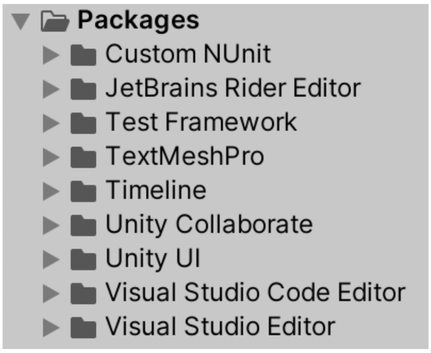
recordering:n.重排序

The default layout contains all the windows that we need, but you can customize it as you like, by reordering and grouping windows. You can also open and close windows, like the one of the asset store. Each window also has its own configuration options, accessible via the triple-dot button in their top right corner. Besides that most also have a toolbar with more options. If your window doesn't look the same as in the tutorials—for example the scene window has a uniform background instead of a skybox—then one of its options is different.

You can switch to a preconfigured layout via the dropdown menu at the top right of the Unity editor. You can also save your current layout there so you can revert to it later.

1.3 Packages

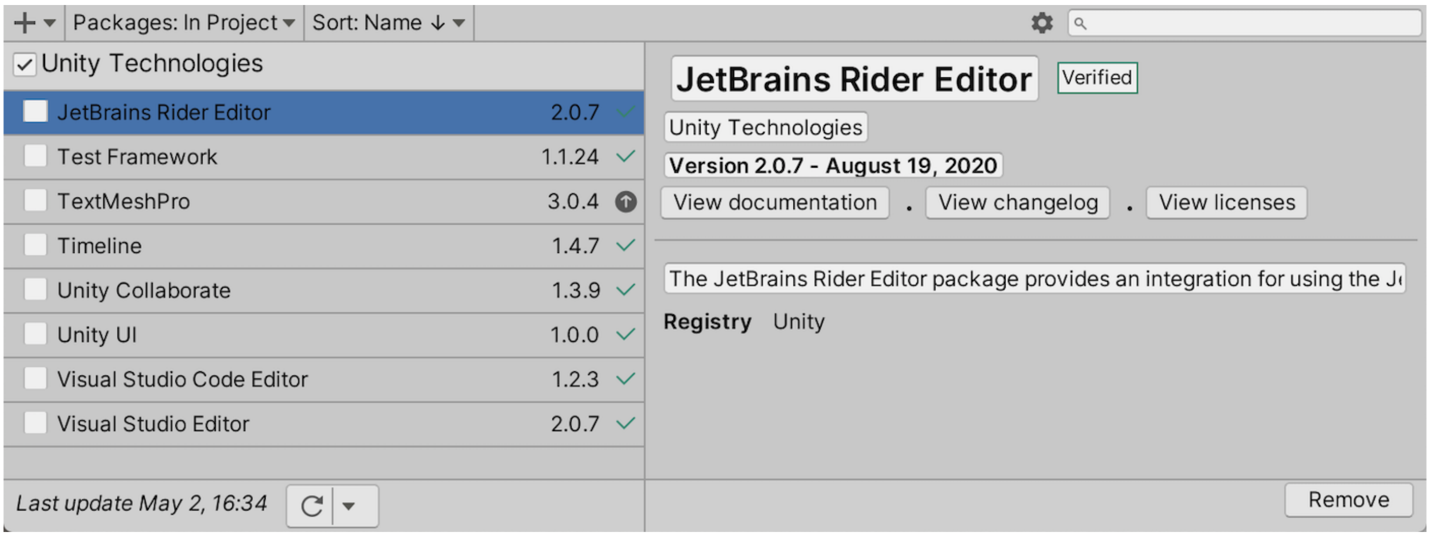
Unity's functionality is modular. Besides the core functionality there are extra packages that can be downloaded and included in your project. The default 3D project currently includes a few packages by default, which you can see in the project window under *Packages*.



Default packages.

These packages can be hidden, by toggling the button at the top right of the project window that looks like an eye with a dash through it. This is purely to reduce visual clutter in the editor, the packages are still part of the project. The button also displays how many such packages there are.

You can control which packages are included in your project via the package manager, which can be opened via the *Window / Package Manager* menu item.



Package manager, only showing packages in project.

The packages add extra functionality to Unity. For example, *Visual Studio Editor* adds integration for the Visual Studio editor, used to write code. This tutorial doesn't use the functionality of the included packages, so I removed them all. The only exception is *Visual Studio Editor* because that's the editor that I use for writing code. If you use a different editor you'd want to include its integration package, if it exists.

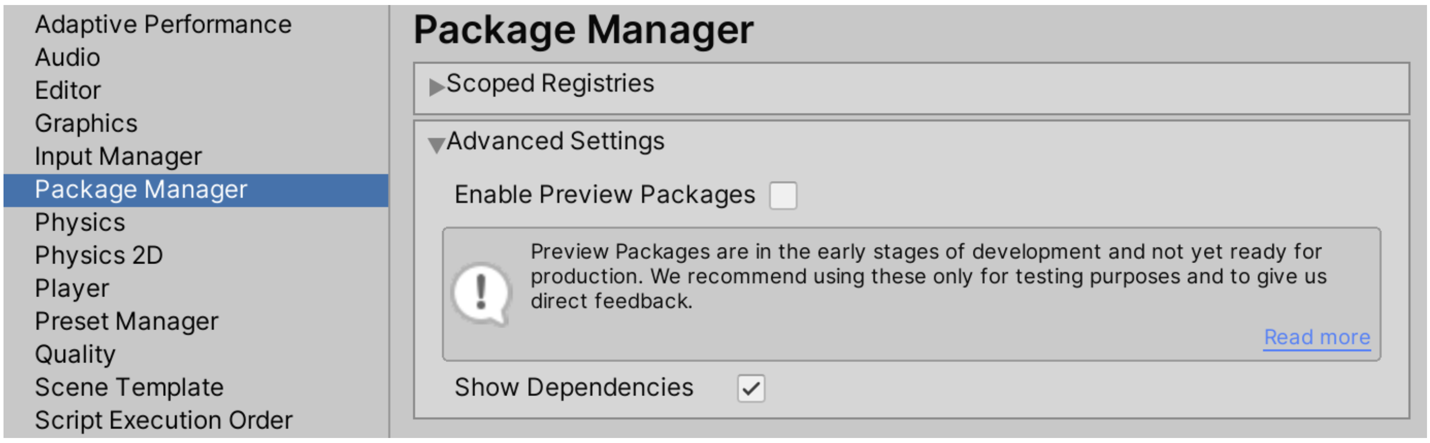
Don't you also need the *Visual Studio Code Editor* package?

Despite the similar names, *Visual Studio* and *Visual Studio Code* are two different editors. You only need one of the packages, depending on which editor you use.

The easiest way to remove packages is by first using the toolbar to limit the package list to *In Project* only. Then select the packages one at a time and use the *Remove* button at the bottom right of the window. Unity will recompile after each removal, so it takes a few seconds before the process is finished.

After removing everything except *Visual Studio Editor* I am left with three packages visible in the project window: *Custom NUnit*, *Test Framework*, and *Visual Studio Editor*. The other two are still there because *Visual Studio Editor* depends on them.

You can make dependencies and implicitly imported packages visible in the package manager via the project settings window, opened via *Edit / Project Settings...* Select its *Package Manager* category and then enable *Show Dependencies* under *Advanced Settings*.



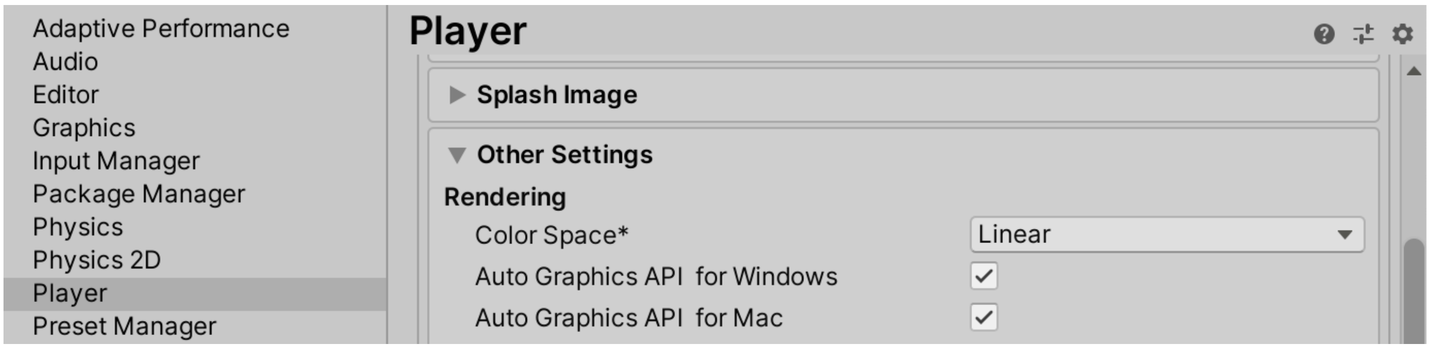
Preview Packages are in the early stages of development and not yet ready for production. We recommend using these only for testing purposes and to give us direct feedback.

Show Dependencies ✓

Package manager project settings; Show Dependencies enabled.

1.4 Color Space

Nowadays rendering is usually done in linear color space, but Unity still configured to use gamma color space by default. For best visual results select the *Player* category of the project settings window, open the *Other Settings* panel, and scroll down to its *Rendering* section. Make sure that *Color Space* is set to *Linear*. Unity will show warning that this might take a long time, but this won't be the case for a nearly-empty project. Confirm to switch.



Color Space* Linear
Auto Graphics API for Windows ✓
Auto Graphics API for Mac ✓

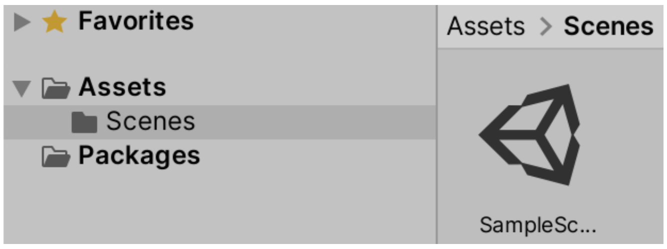
Color space set to linear.

Is there a reason to ever use gamma color space?

Only when you're targeting old hardware or old graphics APIs. OpenGL ES 2.0 and WebGL 1.0 don't support linear space, besides that gamma can be faster than linear on old mobile devices.

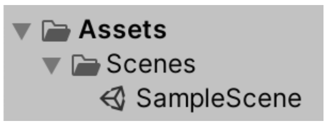
1.5 Sample Scene

The new project contains a sample scene named *SampleScene*, which is opened by default. You can find its asset under *Assets / Scenes* in the project window.



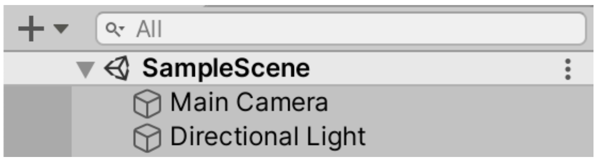
Sample scene in project window.

By default the project window uses a two-column layout. You can switch to a one-column layout via its triple-dot configuration menu option.



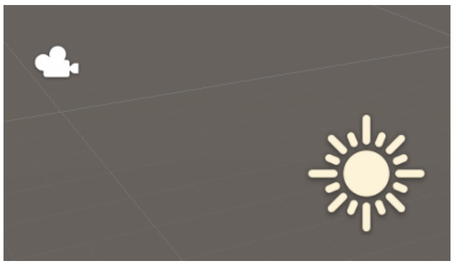
One-column layout.

The sample scene contains a main camera and a directional light. These are game objects. They are listed in the hierarchy window, under the scene.



Object hierarchy within scene.

You can select a game object either via the hierarchy window or the scene window. The camera has a scene icon that looks like an oldfashioned film camera while the directional light's icon looks like a sun.

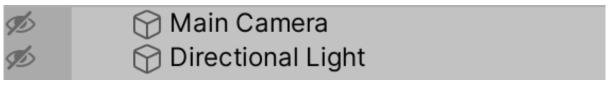


Icons in scene window.

How do I navigate the scene window?

You can use the alt or option key in combination with the cursor to rotate the view. You can also use the arrow keys to move the point of view, and zoom by scrolling. Also, pressing the F key focuses the view on the game object that is currently selected. There are more possibilities, but these are enough to find your way around the scene.

When an object is selected details about it will be shown in the inspector window, but we'll cover those when we need them. We won't need to modify the camera nor the light, so we can hide them in the scene by clicking the eye icon to the left of them in the hierarchy window. This icon is invisible by default but will appear when we hover the cursor there. This is purely to reduce visual clutter in the scene window.



Hidden objects.

What does the hand-like icon next to the eye do?

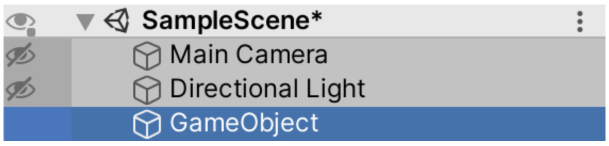
Next to the column that contains the eye icons is another column that contains hand-like icons. These icons are also invisible by default. When a game object's hand icon is active it is impossible to select the object via the scene window. This way you can control which objects respond to selection via the scene window.

2 Building a Simple Clock

Now that our project is set up correctly we can start creating our clock.

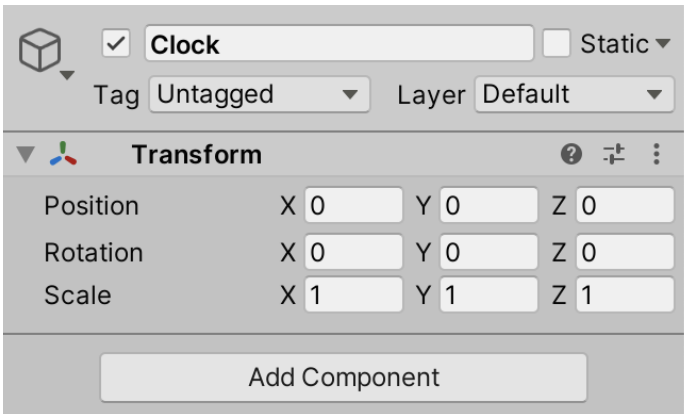
2.1 Creating a Game Object

We need a game object to represent the clock. We'll start with the simplest possible game object, which is an empty one. It can be created via the *GameObject / Create Empty* menu option. Alternatively, you can use the *Create Empty* option in the context menu of the hierarchy window, which you can open with an alternative click, usually a right-click or a two-finger tap. This will add the game object to the scene. It's visible and immediately selected in the hierarchy window under *SampleScene*, which is now marked with an asterisk to indicate that it has unsaved changes. You can also immediately change its name or leave that for later.



Hierarchy with new game object selected.

The inspector window shows the details of the game object as long as it is selected. At its top is a header with the object's name plus a few configuration options. By default, the object is enabled, is not static, is untagged, and sits on the default layer. These settings are fine, except its name. Rename it to *Clock*.



Clock
Tag Untagged
Layer Default
Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

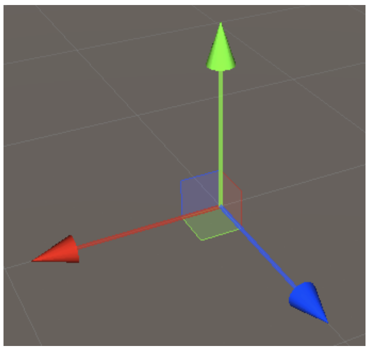
Inspector window with clock selected.

Below the header is a list of all the components of the game object. The list always has a `Transform` component at the top, which is all our clock currently has. It controls the position, rotation, and scale of the game object. Make sure that all the clock's position and rotation values are set to 0. Its scale should be uniformly 1.

What about 2D objects?

When working in 2D instead of 3D, you can ignore one of the three dimensions. Objects specifically meant for 2D—like UI elements—typically have a `RectTransform` instead, which is a specialized `Transform` component.

Because the game object is empty it isn't visible in the scene window itself. However, a manipulation tool is visible at the game object's location, which is at the center of the world.

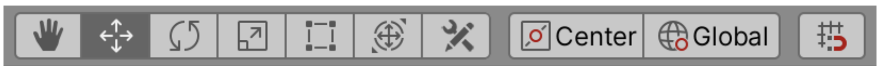


Selected with move tool.

Why don't I see a manipulation tool after selecting the clock?

The manipulation tool exists in the scene window. Make sure that you're looking at the scene window, not the game window.

Which manipulation tool is active can be controlled via the buttons at the top left of the editor toolbar. The modes can also be activated via the Q, W, E, R, T, and Y keys. The rightmost button in the group is for enabling custom editor tools, which we don't have. The move tool is active by default.

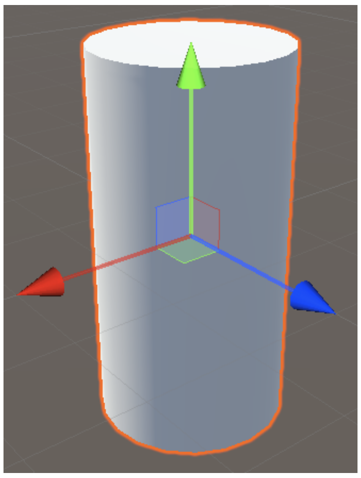


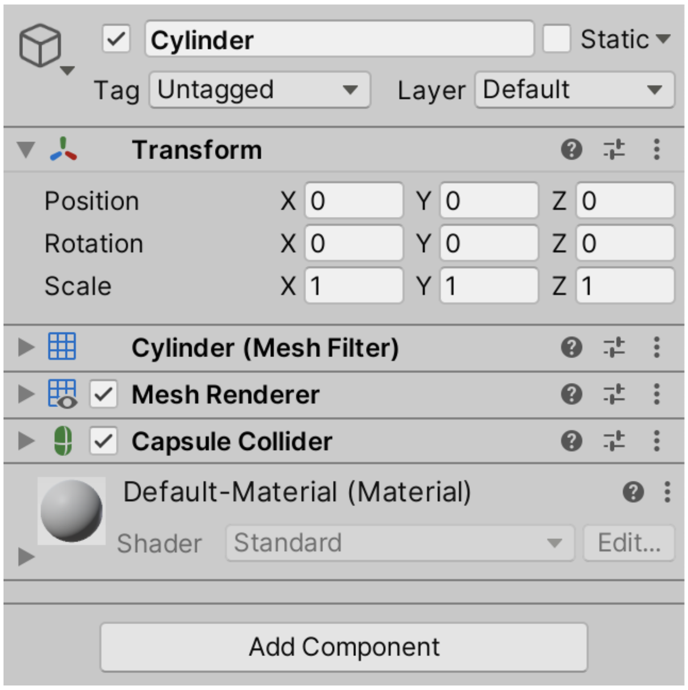
Manipulation mode toolbar.

Next to the mode buttons are three more buttons to control the placement, orientation, and snapping of manipulation tools.

2.2 Creating the Face of the Clock

Although we have a clock object, we don't see anything yet. We'll have to add 3D models to it so something gets rendered. Unity contains a few primitive objects that we can use to build a simple clock. Let's begin by adding a cylinder to the scene via *GameObject / 3D Object / Cylinder*. Make sure that it has the same `Transform` values as our clock.





Cylinder ☑ Static
Tag Untagged Layer Default

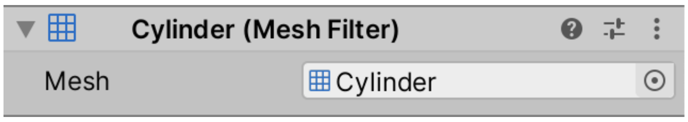
Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Cylinder (Mesh Filter)
☑ Mesh Renderer
☑ Capsule Collider
Default-Material (Material)
Shader Standard

Game object representing a cylinder.

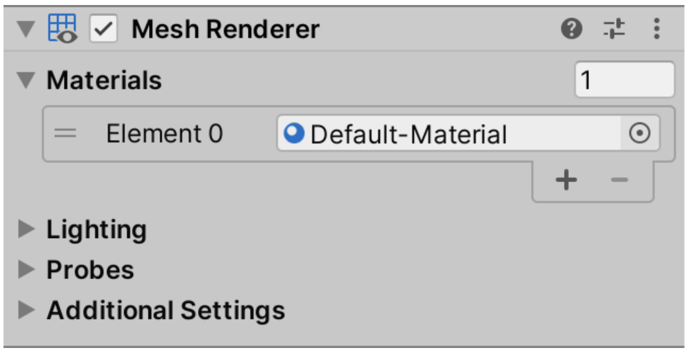
The new object has three more components than an empty game object. First, it has a `MeshFilter`, which contains a reference to the built-in cylinder mesh.



Cylinder (Mesh Filter)
Mesh Cylinder

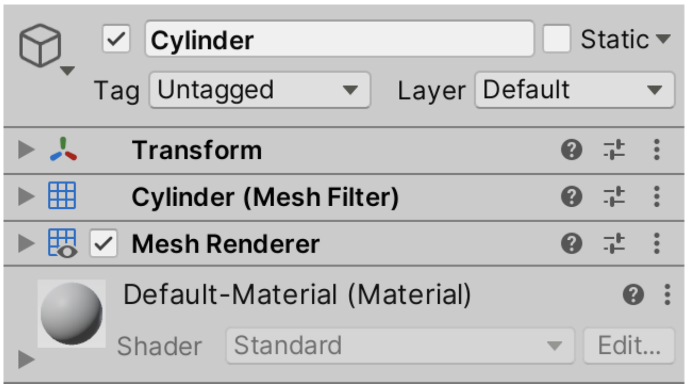
`MeshFilter` *component, set to cylinder.*

Second is a `MeshRenderer`. This component's purpose is to ensure that the object's mesh gets rendered. It also determines what material is used for rendering, which is the default material. This material is also shown in the inspector, below the component list.


Mesh Renderer
Materials 1
Element 0 Default-Material
Lighting
Probes
Additional Settings

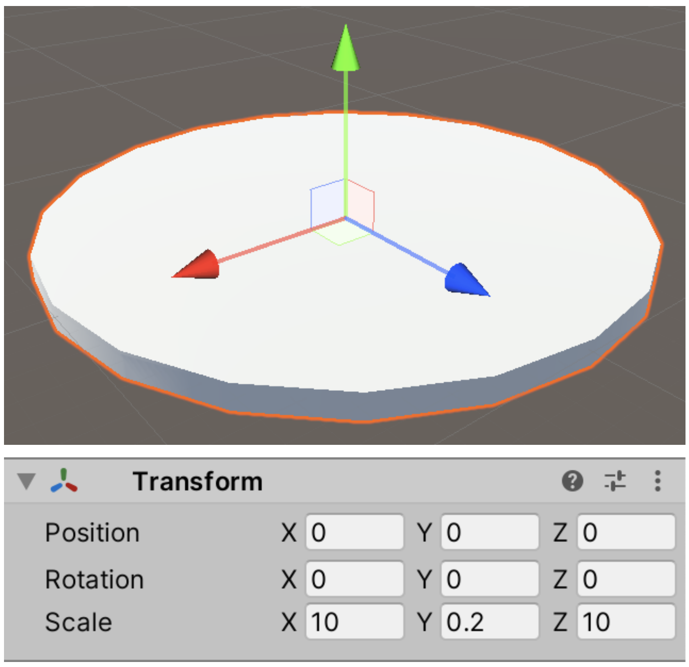
`MeshRenderer` *component, set to default material.*

Third is a `CapsuleCollider`, which is for 3D physics. The object represents a cylinder, but it has a capsule collider because Unity doesn't have a primitive cylinder collider. We don't need it, so we can remove this component. If you'd like to use physics with your clock, you're better off using a `MeshCollider` component. Components can be removed via the triple-dot dropdown menu in their top right corner.


Cylinder Static
Tag Untagged Layer Default
Transform
Cylinder (Mesh Filter)
Mesh Renderer
Default-Material (Material)
Shader Standard Edit...

Cylinder without collider.

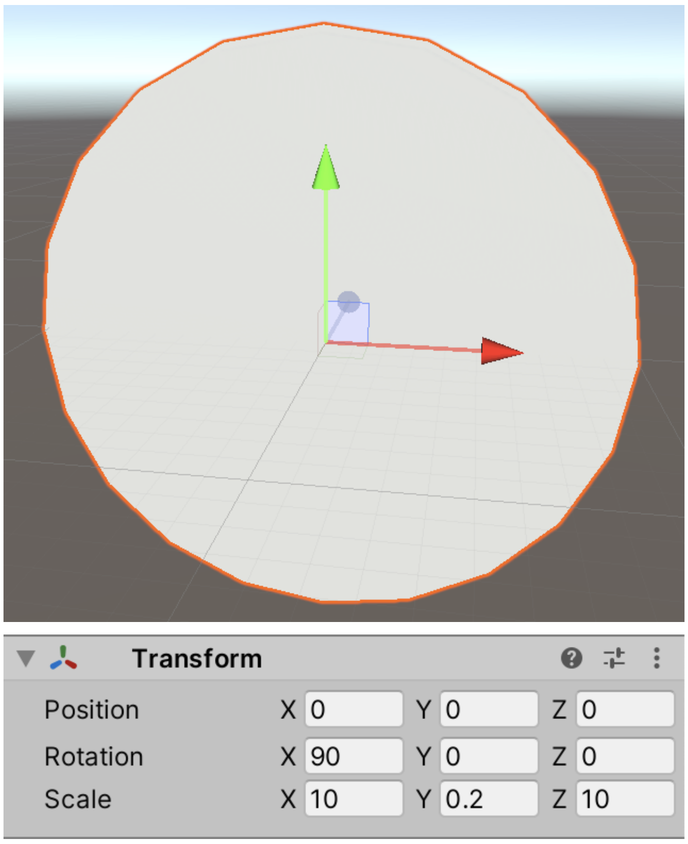
We'll turn the cylinder into the clock's face, by flattening it. This is done by decreasing the Y component of its scale. Reduce it to 0.2. As the cylinder mesh is two units high, its effective height becomes 0.4 units. Let's also make a big clock, so increase the X and Z components of its scale to 10.



	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	10	0.2	10

Scaled cylinder.

Our clock is supposed to stand or hang on a wall, but its face is currently laying flat. We can fix this by rotating the cylinder a quarter turn. In Unity the X axis points right, the Y axis points up, and the Z axis points forward. So let's design our clock with the same orientation in mind, meaning that we see its front while we're looking at it along the Z axis. Set the cylinder's X rotation to 90 and adjust the scene view so the clock's front is visible, so the blue Z arrow of the move tool points away from you, into the screen.

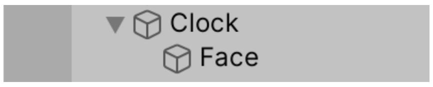


Transform

Position	X 0	Y 0	Z 0
Rotation	X 90	Y 0	Z 0
Scale	X 10	Y 0.2	Z 10

Rotated cylinder.

Change the name of the cylinder object to *Face*, as it represents the face of the clock. It is only one part of the clock, so we make it a child of the *Clock* object. We do this by dragging the face onto the clock in the hierarchy window.



Clock
Face

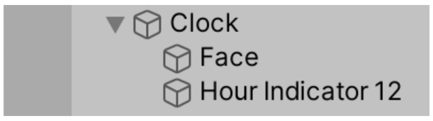
Face child object.

Child objects are subject to the transformation of their parent object. This means that when *Clock* changes position, *Face* does as well. It's as if they are a single entity. The same goes for rotation and scale. You can use this to make complex object hierarchies.

2.3 Creating the Clock Periphery

The outer ring of a clock's face usually has markings that help indicate what time it is displaying. This is known as the clock periphery. Let's use blocks to indicate the hours of a 12-hour clock.

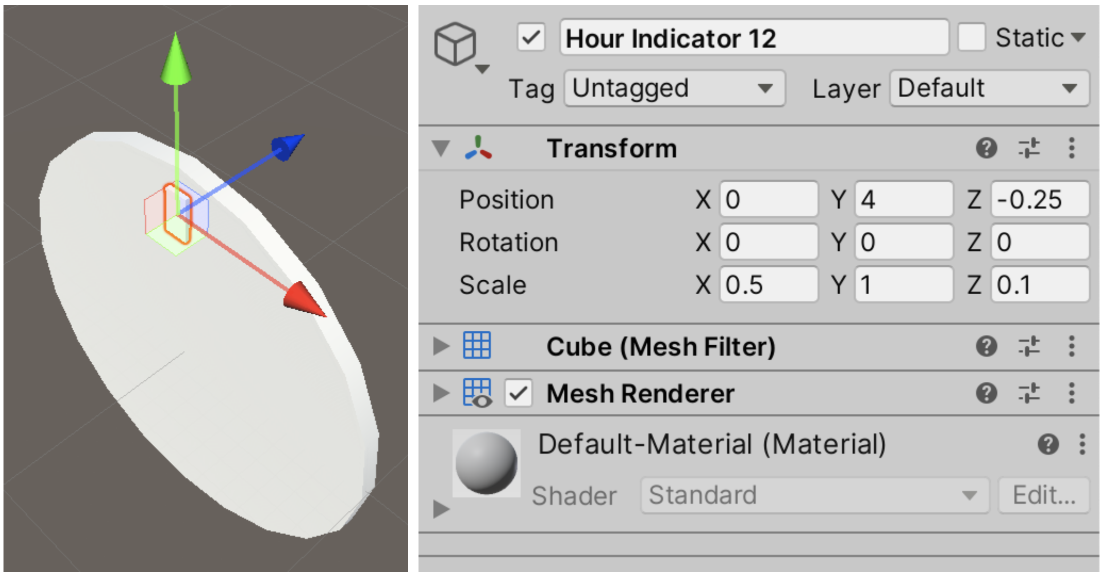
Add a cube object to the scene via *GameObject / 3D Object / Cube*, name it *Hour Indicator 12*, and also make it a child of *Clock*. The order of the child objects in the hierarchy doesn't matter, you could place it either above or below the face.



```
▼ Clock
    Face
    Hour Indicator 12
```

Hour indicator child object.

Set its X scale to 0.5, Y scale to 1, and Z scale to 0.1 so it becomes a narrow flat long block. Then set its X position to 0, Y position to 4, and Z position to −0.25. That places it on top of the face to indicate hour 12. Also remove its `BoxCollider` component.



Hour Indicator 12 — Static
Tag: Untagged — Layer: Default

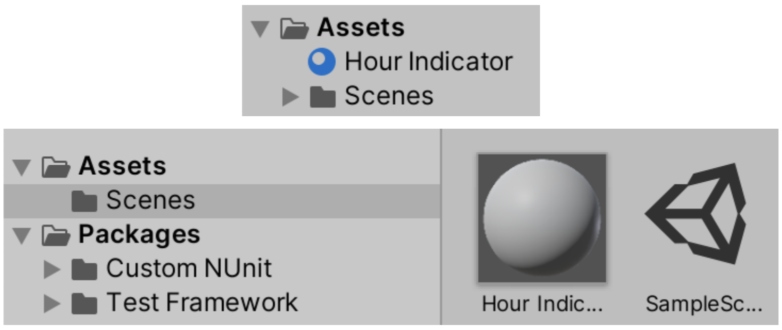
Transform

	X	Y	Z
Position	0	4	-0.25
Rotation	0	0	0
Scale	0.5	1	0.1

Cube (Mesh Filter)
Mesh Renderer
Default-Material (Material)
Shader: Standard — Edit...

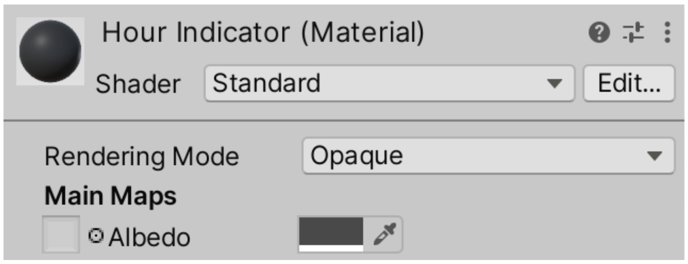
Indicator for hour 12.

The indicator is hard to see, because it has the same color as the face. Let's create a separate material for it, via *Assets / Create / Material*, or via the plus button or context menu of the project window. This gives us a material asset that is a duplicate of the default material. Change its name to *Hour Indicator*.



Hour indicator in project window, one and two column layout.

Select the material and change its *Albedo* to something else, by clicking its color field. That opens a color popup window which offers various ways to pick a color. I chose dark gray, corresponding to hexadecimal 494949, which is the same as uniform 73 for RGB 0–255 mode. We don't use the alpha channel so its value is irrelevant. We can also leave all other material properties as they are.

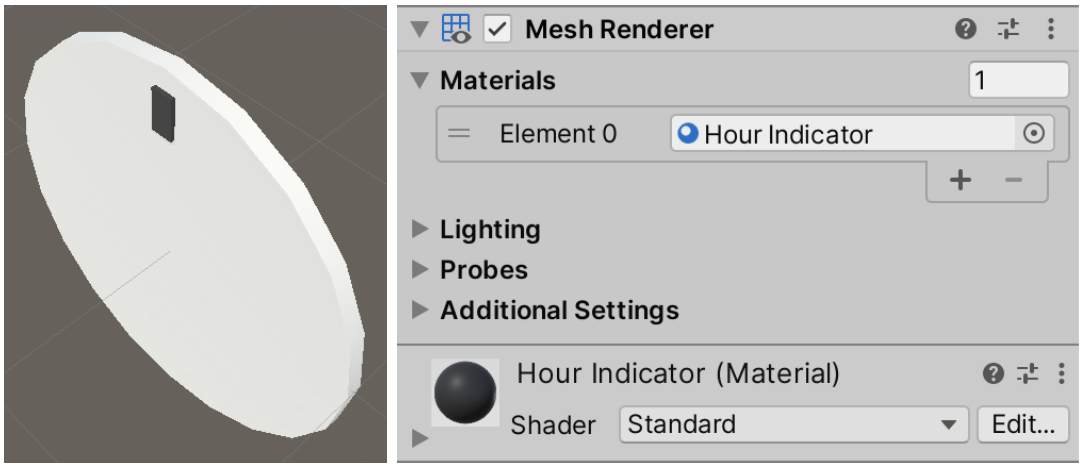

Hour Indicator (Material)
Shader Standard
Rendering Mode Opaque
Main Maps
Albedo

Dark gray albedo.

What is albedo?

Albedo is a Latin word which means whiteness. It's the color of something when illuminated by white light.

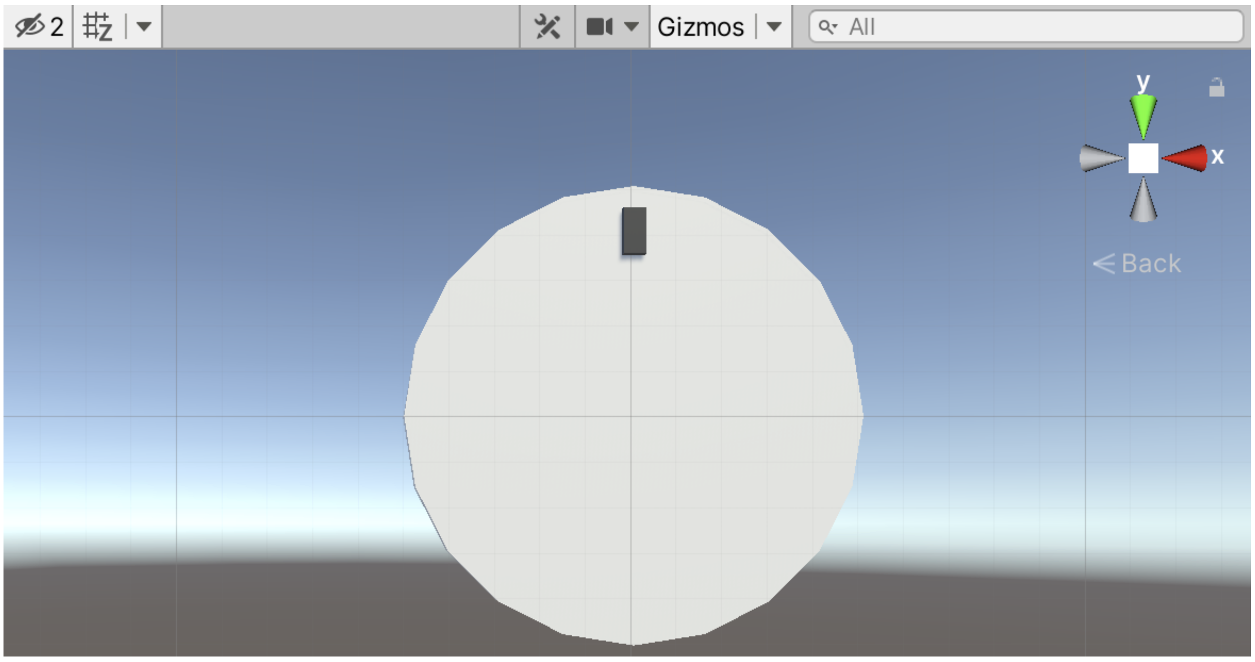
Make the hour indicator use this material. You can do this by dragging the material onto the object in either the scene or hierarchy window. You can also drag it to the bottom of the inspector window when the indicator game object is selected, or change *Element 0* of the *Materials* array of its `MeshRenderer`.



Dark hour indicator.

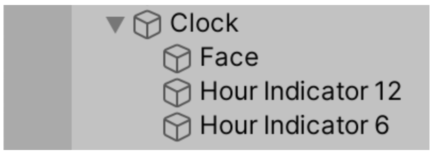
2.4 Twelve Hour Indicators

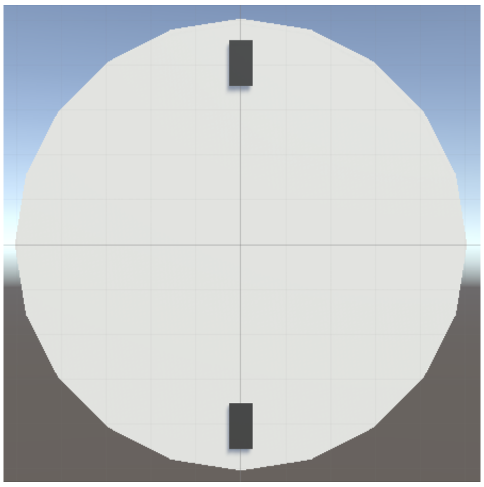
We could make do with a single indicator for hour 12, but let's include one for every hour. Begin by orienting the scene view camera so we look straight down the Z axis. You can do this by clicking on the axis cones of the view camera gizmo at the top right of the scene view. You can also change the axis of the scene grid to Z via the grid toolbar button.



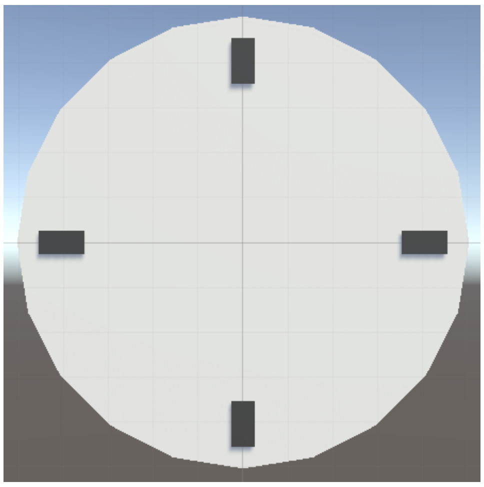
Looking straight at clock, along Z axis.

Duplicate the *Hour Indicator 12* game object. You can do this via *Edit / Duplicate*, via the indicated keyboard shortcut, or via the its context menu in the hierarchy window. The duplicate will appear underneath the original in the hierarchy window, also a child of *Clock*. Its name is set to *Hour Indicator 12 (1)*. Rename it to *Hour Indicator 6* and negate the Y component of its position so it indicates hour 6.

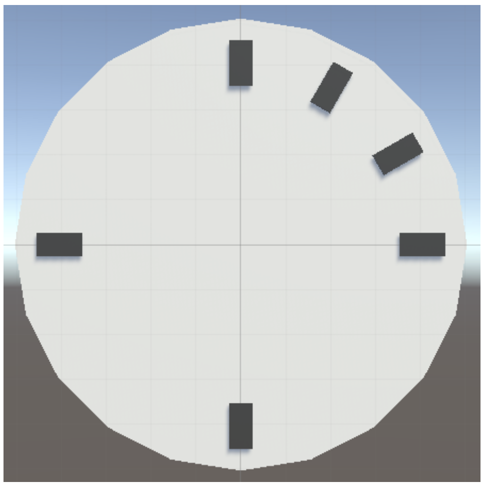

Clock
Face
Hour Indicator 12
Hour Indicator 6


Indicators for hours 6 and 12.

Create indicators for hours 3 and 9 in the same way. In this case their X positions should be 4 and −4 while their Y positions should be zero. Also, set their Z rotation to 90 so they're turned a quarter circle.


Four hour indicators.

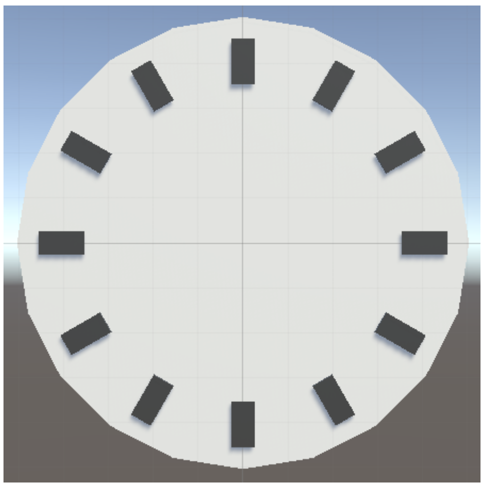
Then create another duplicate of *Hour Indicator 12*, this time for hour 1. Set its X position to 2, its Y position to 3.464, and its Z rotation to −30. Then duplicate that one for hour 2, swap its X and Y positions, and double its Z rotation to −60.


Indicators for hours 1 and 2.

Where do those numbers come from?

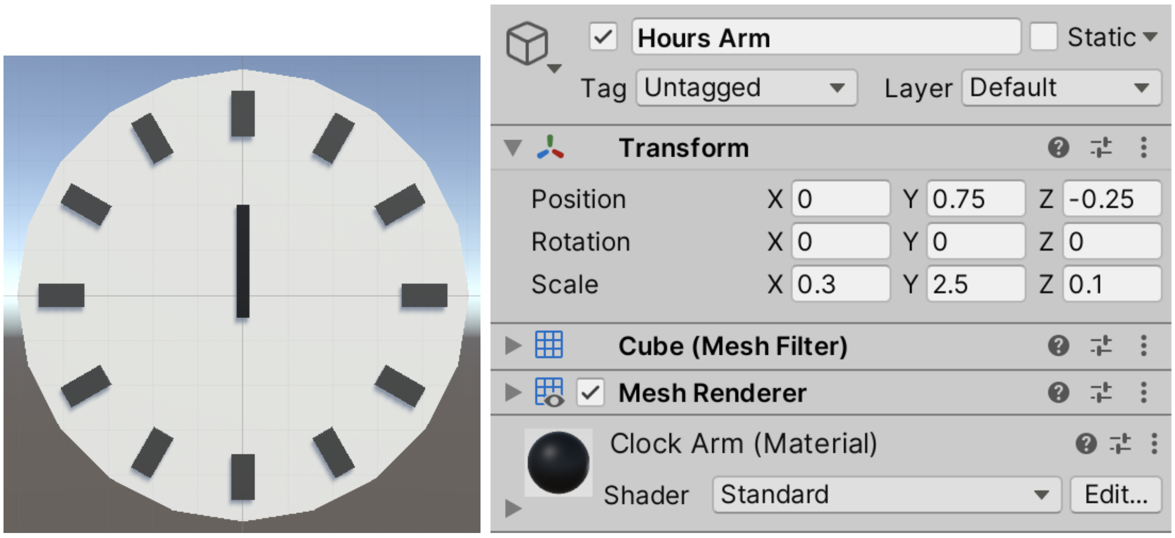
Each hour covers a 30° clockwise rotation along the Z axis. In this case we use negative rotations because Unity's rotation is counterclockwise. We can find the position for hour 1 via trigonometry. The sine of 30° is ½ and its cosine is $\frac{\sqrt{3}}{2}$. We scale those by the distance at which we position the hours indicator from the center, which is 4. So we end up with X 2 and Y $2\sqrt{3} \approx 3.464$. For hour 2 the rotation is 60°, for which we can simply swap the sine and cosine.

Duplicate these two indicators and negate their Y positions and their rotations to create the indicators for hours 4 and 5. Then use the same trick on hours 1, 2, 4, and 5 to create the remaining indicators, this time negating their X positions and again their rotations.


All hour indicators.

2.5 Creating the Arms

The next step is to create the arms of the clock. We start with the hour arm. Again duplicate *Hour Indicator 12* and name it *Hours Arm*. Then create a *Clock Arm* material and make the arm use it. In this case I made it solid black, hexadecimal 000000. Decrease the arm's X scale to 0.3 and increase its Y scale to 2.5. Then change its Y position to 0.75 so it points towards hour 12, but also a bit in the opposite direction. That makes it look as if the arm has a little counterweight when it rotates.


Hours Arm ✓ Static ☐
Tag Untagged Layer Default

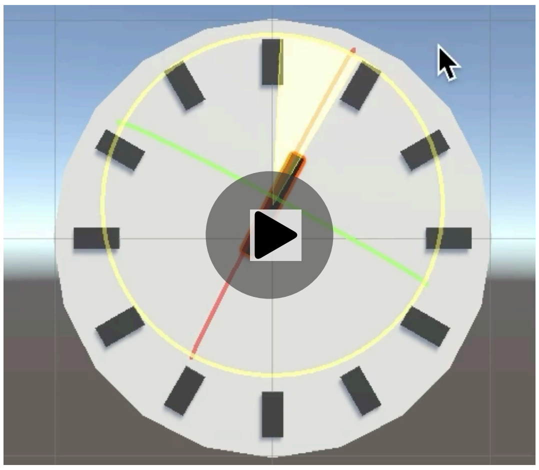
Transform

Position	X 0	Y 0.75	Z -0.25
Rotation	X 0	Y 0	Z 0
Scale	X 0.3	Y 2.5	Z 0.1

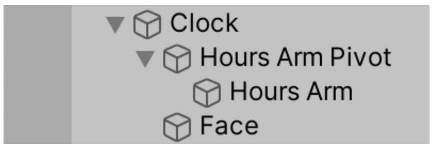
Cube (Mesh Filter)
Mesh Renderer ✓
Clock Arm (Material)
Shader Standard Edit...

Hours arm.

The arm must rotate around the center of the clock, but changing its Z rotation makes it rotate around its own center.

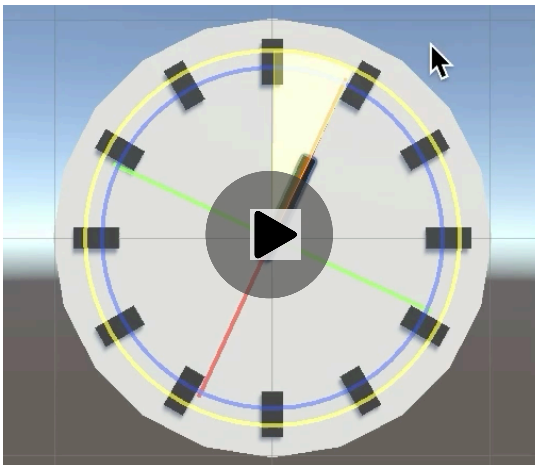

Clock arm rotates around its center.

This happens because rotation is relative to the local position of the game object. To create the appropriate rotation we have to introduce a pivot object and rotate that one instead. So create a new empty game object and make it a child of *Clock*. You can do this directly by creating the object via the context menu of *Clock* in the hierarchy window. Name it *Hours Arm Pivot* and make sure that its position and rotation are zero and its scale is uniformly 1. Then make *Hours Arm* a child of the pivot.



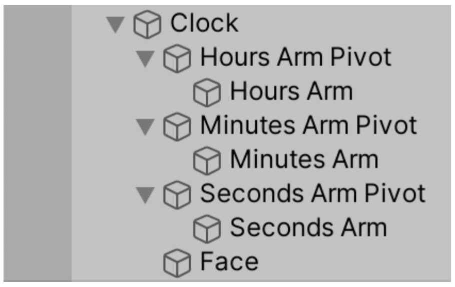
Hours arm with pivot.

Now try rotating the pivot. If you do this via the scene view make sure that the tool handle position mode is set to *Pivot* instead of *Center*.



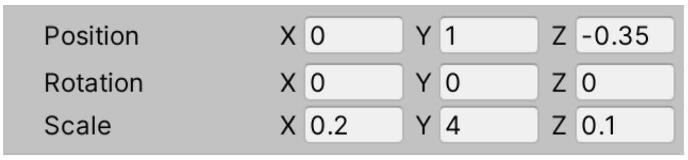
Clock arm rotates around the pivot.

Duplicate *Hours Arm Pivot* twice to create a *Minutes Arm Pivot* and a *Seconds Arm Pivot*. Rename them accordingly, including the duplicated arm child objects.



All arm hierarchies.

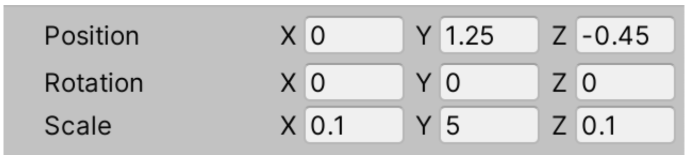
Minutes Arm should be narrower and longer than *Hours Arm*, so set its X scale to 0.2 and Y scale to 4, then increase its Y position to 1. Also change its Z position to −0.35 so it sits on top of the hours arm. Note that this applies to the arm, not its pivot.



Position	X 0	Y 1	Z -0.35
Rotation	X 0	Y 0	Z 0
Scale	X 0.2	Y 4	Z 0.1

`Transform` *of Minutes Arm.*

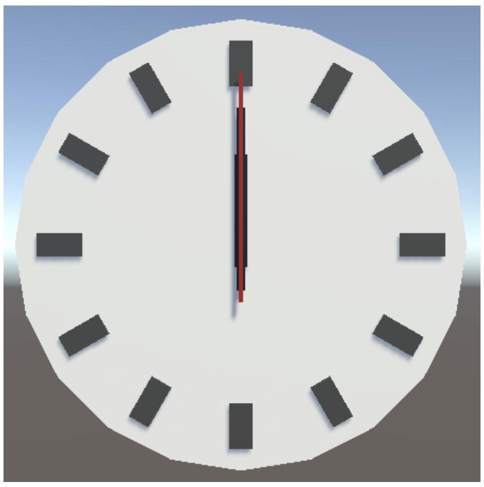
Adjust *Seconds Arm* as well. This time use 0.1 and 5 for the XY scale and 1.25 and −0.45 for the YZ position.



Position	X 0	Y 1.25	Z -0.45
Rotation	X 0	Y 0	Z 0
Scale	X 0.1	Y 5	Z 0.1

`Transform` *of Seconds Arm.*

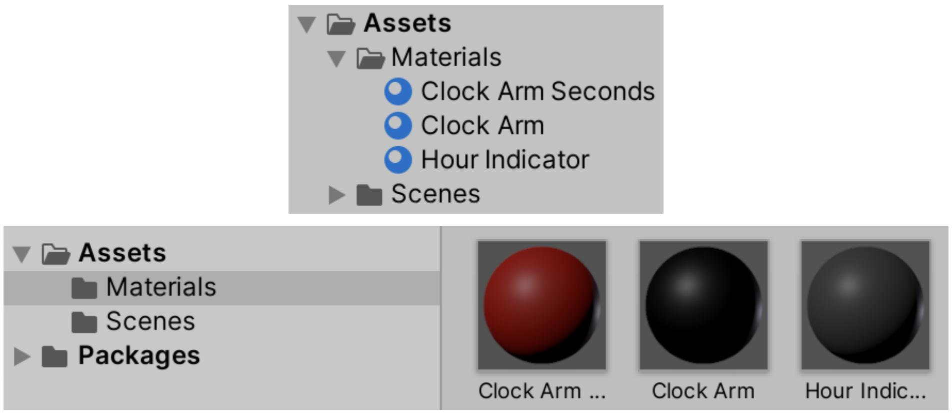
Let's make the seconds arm stand out by creating a separate material for it. I gave it a dark red color, hexadecimal B30000. Also, I turned off the grid in the scene window as we finished building our clock.



Clock with three arms.

If you haven't done so already, this is a good moment to save the scene, via *File / Save* or the indicated keyboard shortcut.

It is also a good idea to keep the project's assets organized. As we have three materials let's put them in a *Materials* folder that we create via *Assets / Create / Folder* or via the project window. You can then drag the materials there.



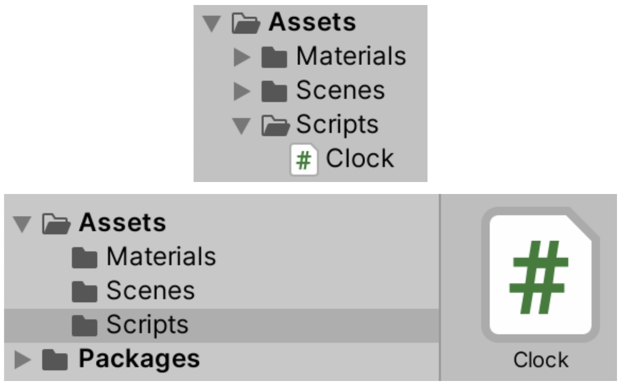
Material folder in project window, one and two column layout.

3 Animating the Clock

Our clock currently does not tell the time, it's always stuck at twelve o'clock. To animate it we have to add a custom behavior to it. We do this by creating a custom component type, which is defined via a script.

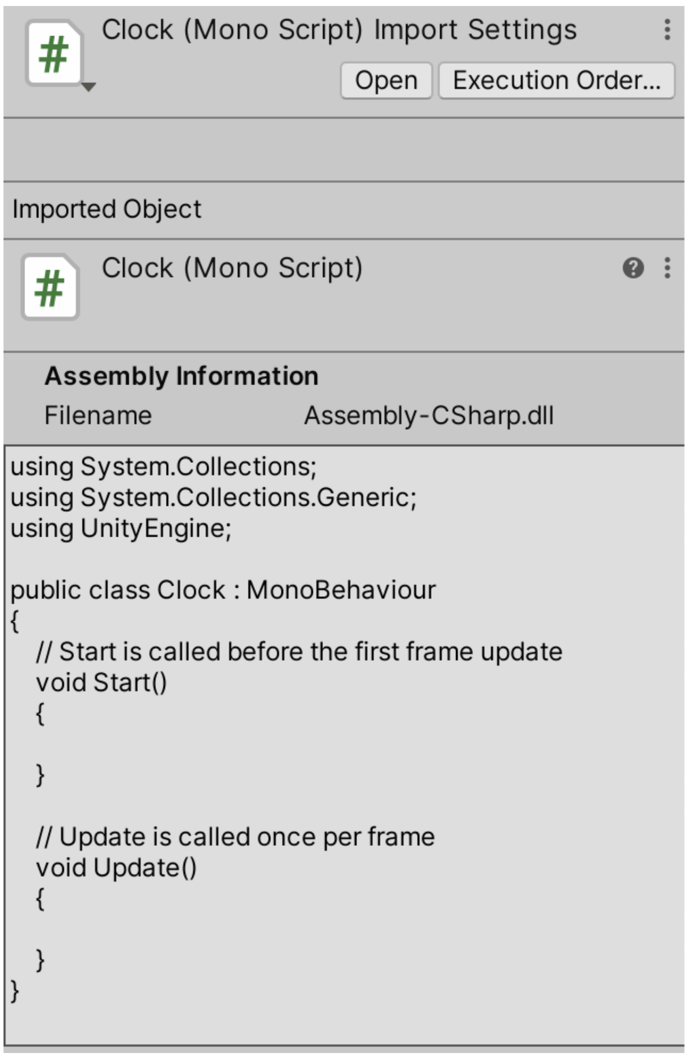
3.1 C# Script Asset

Add a new script asset to the project via *Assets / Create / C# Script* and name it *Clock*. C# is the programming language used for Unity scripts and is pronounced as *C-sharp*. Let's also immediately put it in a new *Scripts* folder to keep the project tidy.



Scripts folder with `Clock` *script, one and two column layout.*

When the script is selected the inspector will show its contents. But to edit the code we'll have to use a code editor. You can open the script for editing by pressing the *Open...* button in its inspector or by double-clicking it in the hierarchy window. Which program gets opened can be configured via Unity's preferences.



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

public class Clock : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Inspector of C# `Clock` *asset.*

3.2 Defining a Component Type

Once the script is loaded in your code editor begin by deleting the standard template code, as we'll create the component type from scratch.

An empty file defines nothing. It must contain the definition of our clock component. What we're going to define isn't a single instance of a component. Instead, we define the general class or type known as `Clock`. Once that's established, we could create multiple such components in Unity, even though we'll limit ourselves to a single clock in this tutorial.

In C#, we define the `Clock` type by first stating that we're defining a class, followed by its name. In the code fragments below, changed code has a yellow background, or dark red if you're using the dark web page theme to view this tutorial. As we start with an empty file, the contents of it should literally become `class Clock` and nothing else, though you could add spaces and newlines between words as you like.

```
class Clock
```

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 contain and what functionality they have.

Classes can also define data and functionality that don't belong to object instances, but to the class itself. This is often used to provide globally-available functionality. We'll use some of that, but `Clock` won't have it.

Because we don't want to restrict which code has access to our `Clock` type, it is good form to prefix it with the `public` access modifier.

```
public class Clock
```

What is the default access modifier for classes?

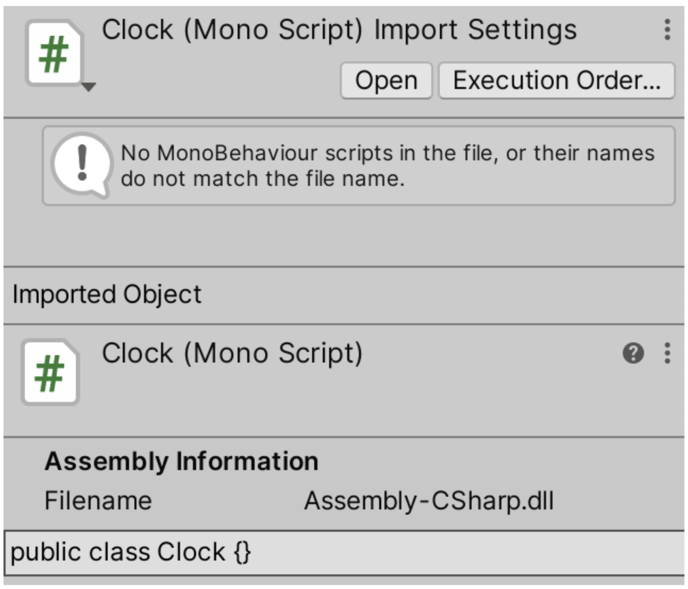
Without the access modifier, it would be as if we had written `internal class Clock`. That would restrict access to code from the same assembly, which becomes relevant when you use code packaged in separate assemblies. To make sure it always works, make classes public by default.

At this point we don't have valid C# syntax yet. If you were to save the file and go back to the Unity editor then compilation errors will get logged in its console window.

We indicated that we're defining a type, so we must actually define what it is like. That's done by a block of code that follows the declaration. The boundaries of a code block are indicated with curly brackets. We're leaving it empty for now, so just write `{}`.

```
public class Clock {}
```

Our code is now valid. Save the file and switch back to Unity. The Unity editor will detect that the script asset has changed and triggers a recompilation. After that is done, select our script. The inspector will inform us that the asset does not contain a `MonoBehaviour` script.


Clock (Mono Script) Import Settings

No MonoBehaviour scripts in the file, or their names do not match the file name.

Imported Object

Clock (Mono Script)

Assembly Information

Filename	Assembly-CSharp.dll

```
public class Clock {}
```

Non-component script.

What this means is that we cannot use this script to create components in Unity. At this point, our `Clock` defines a basic C# object type. Our custom component type must extend Unity's `MonoBehaviour` type, inheriting its data and functionality.

What does mono-behavior mean?

The idea is that we can program our own components to add custom behavior to game objects. That's what the behavior part refers to. It just happens to use the British spelling, which is an oddity. The mono part refers to the way in which support for custom code was added to Unity. It used the Mono project, which is a multi-platform implementation of the .NET framework. Hence, `MonoBehaviour`. It's an old name that we're stuck with due to backwards-compatibility.

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 after our type name, followed by what it extends. This makes `Clock` inherit everything of the `MonoBehaviour` class type.

```
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`.

```
public class Clock : UnityEngine.MonoBehaviour {}
```

What's a namespace?

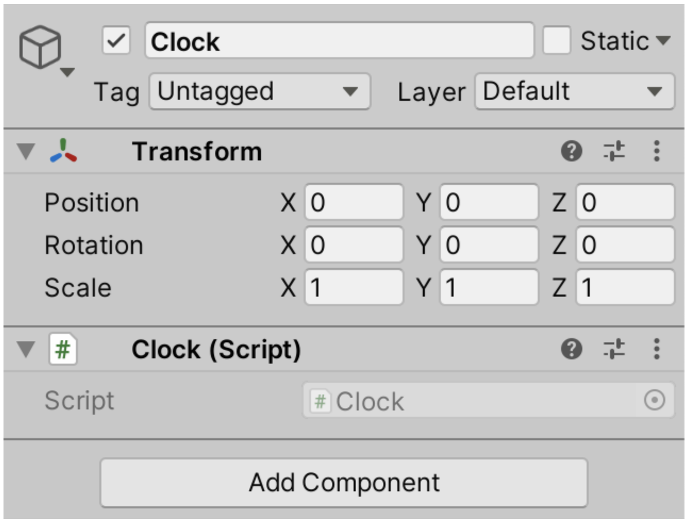
A namespace is like a website domain, but for code. Just like domains can have subdomains, namespaces can have sub-namespaces. The big difference is that it's written the other way around. So instead of forum.unity.com it would be com.unity.forum. Namespaces are used to organize code and prevent name clashes.

The assembly containing the `UnityEngine` code comes with Unity, you don't have to go online to fetch it separately. The project file used by the code editor should be set up automatically to recognize it, if you imported the appropriate editor integration package.

It is inconvenient to always have to include the `UnityEngine` prefix when accessing Unity types. Fortunately we can declare that the namespace should be searched automatically to complete type names in the C# file. This is done by adding `using UnityEngine;` at the top of the file. The semicolon is required to mark the end of the statement.

```
using UnityEngine;

public class Clock : MonoBehaviour {}
```

Now we can add our custom component to the *Clock* game object in Unity. This can be done either by dragging the script asset onto the object, or via the *Add Component* button at the bottom of the object's inspector.



Clock game object with our `Clock` *component.*

Note that most code types in my tutorials are linked to online documentation. For example, `MonoBehaviour` is a link that takes you to Unity's online scripting API page for that type.

3.3 Getting Hold of an Arm

To rotate the arms, `Clock` objects need to know about them. Let's start with the hours arm. Like all game objects, it can be rotated by adjusting its `Transform` component. So we have to add knowledge of the arm pivot's `Transform` component to `Clock`. This can be done by adding a data field inside its code block, defined as a name followed by a semicolon.

The name `hours pivot` would be appropriate for the field. However, names have to be single words. The convention is to make the first word of a field name lowercase and capitalize all other words, then stick them together. So we'll name it `hoursPivot`.

```
public class Clock : MonoBehaviour {

    hoursPivot;
}
```

Where did the `using` statement go?

It's still there, I just didn't show it. The code fragments will contain enough of the existing code so you know the context of the changes.

We also have to declare the type of the field, which in this case is `UnityEngine.Transform`. It has to be written in front of the field's name.

```
    Transform hoursPivot;
```

Our class now defines a field that can hold a reference to another object, whose type has to be `Transform`. We have to make sure that it holds a reference to the `Transform` component of the hours arm pivot.

Fields are private by default, which means that they can only be accessed by the code belonging to `Clock`. But the class doesn't know about our Unity scene, so there's no direct way to associate the field with the correct object. We can change that by declaring the field as serializable. This means that it should be included in the scene's data when Unity saves the scene, which it does by putting all data in a sequence—serializing it—and writing it to a file.

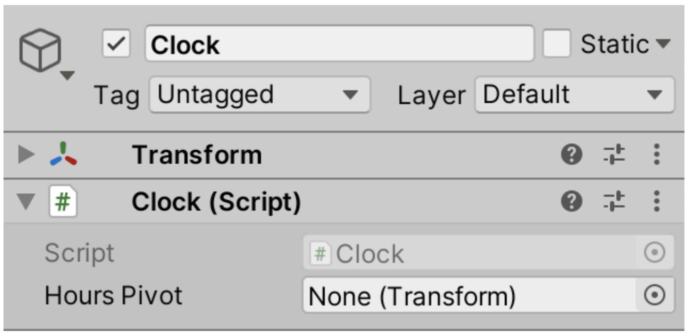
Marking a field as serializable is done by attaching an attribute to it, in this case `SerializeField`. It's written in front of the field declaration between square brackets, typically on the line above it but can also be placed on the same line.

```
	[SerializeField]
	Transform hoursPivot;
```

Can't we just make it `public`?

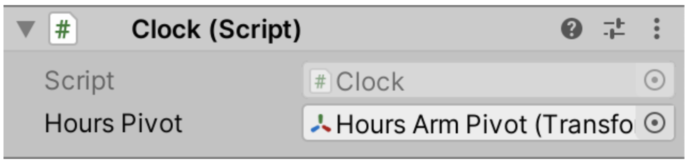
Yes, but it is generally bad form to make class fields publicly accessible. The rule of thumb is to only make class contents public if C# code from other types need access to it, and then prefer methods or properties over fields. The less accessible something is the easier it is to maintain, because there's less code that could directly depend on it. In this tutorial our only C# code is `Clock` so there's no reason to make its contents public.

Once the field is serializable Unity will detect this and display it in the inspector window of the `Clock` component of our *Clock* game object.



Hours pivot field.

To make the proper connection, drag the *Hours Arm Pivot* from the hierarchy onto the *Hours Pivot* field. Alternatively, use the circular button at the right of the field and search for the pivot in the list that pops up. In both cases the Unity editor grabs the `Transform` component of *Hours Arm Pivot* and puts a reference to it in our field.



Hours pivot connected.

3.4 Knowing all Three Arms

We have to do the same for the minutes and seconds arm pivots. So add two more serializable `Transform` fields to `Clock` with appropriate names.

```
	[SerializeField]
	Transform hoursPivot;

	[SerializeField]
	Transform minutesPivot;

	[SerializeField]
	Transform secondsPivot;
```

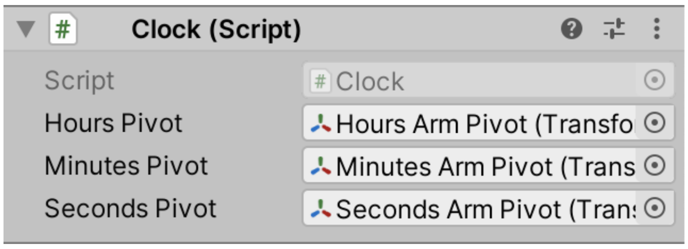
It is possible to make these field declarations more concise, because they share the same attribute, access modifier, and type. They can be consolidated into a comma-separated list of field names following the attribute and type declaration.

```
	[SerializeField]
	Transform hoursPivot, minutesPivot, secondsPivot;

	~~//[SerializeField]~~
	~~//Transform minutesPivot;~~

	~~//[SerializeField]~~
	~~//Transform secondsPivot;~~
```

What does // do?

Double slashes indicate a comment. All text after them until the end of the line is ignored by the compiler. It is used to add text to clarify code, if needed. I also use it to indicate code that has been removed. Besides that, deleted code has a line through it.

Hook up the other two arms in the editor as well.


Clock (Script)

Script	Clock
Hours Pivot	Hours Arm Pivot (Transfo
Minutes Pivot	Minutes Arm Pivot (Trans
Seconds Pivot	Seconds Arm Pivot (Tran:

All three pivots connected.

3.5 Waking Up

Now that we have access to the arm pivots the next step is to rotate them. To do this, we need to tell `Clock` to execute some code. This is done by adding a code block to the class, known as a method. The block has to be prefixed by a name, which is capitalized by convention. We'll name it `Awake`, suggesting that the code should be executed when the component awakens.

```
public class Clock : MonoBehaviour {

    [SerializeField]
    Transform hoursPivot, minutesPivot, secondsPivot;

    Awake {}
}
```

Methods are somewhat like mathematical functions, for example $f(x) = 2x + 3$. That function takes a number—represented by the variable parameter x—doubles it, then adds three. It operates on a single number, and its result is a single number as well. In the case of a method, it's more like $f(p) = c$ where p represents input parameters and c represents whatever code it executes.

Like a mathematical function a method can produce a result, but this isn't required. We have to declare the type of the result—as if it were a field—or write `void` to indicate that there is no result. In our case, we just want to execute some code without providing a resulting value, so we use `void`.

```
    void Awake {}
```

We also don't need any input data. However, we still have to define the method's parameters, as a comma-separated list between round brackets. It's just an empty list in our case.

```
    void Awake () {}
```

We now have a valid method, although it doesn't do anything yet. Just like Unity detected our fields, it also detects this `Awake` method. When a component has an `Awake` method, Unity will invoke that method on the component when it awakens. This happens after it's been created or loaded while in play mode. We're currently in edit mode, so this doesn't happen yet.

Doesn't `Awake` have to be `public`?

`Awake` and a collection of other methods are considered special Unity event methods. The Unity engine will find them and invoke them when appropriate, no matter how we declare them. This happens from outside the managed .NET environment.

Note that `Awake` and other special Unity event methods have bold text in my tutorials and link to their online Unity scripting API page.

3.6 Rotating via Code

To rotate the arms we have to create a new rotation. We can change the rotation of a `Transform` by assigning a new one to its `localRotation` property.

What's a property?

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

Although the rotation of a `Transform` component is defined with Euler angles in degrees per axis in the inspector, in code we have to do it with a quaternion.

What's a quaternion?

Quaternions are based on complex numbers and are used to represent 3D rotations. Although harder to understand than a combination of separate X, Y, and Z rotation angles, they have some useful characteristics. For example, they don't suffer from gimbal lock.

We can create a quaternion based on Euler angles by invoking the `Quaternion.Euler` method. Do this by writing it in `Awake`, followed by a semicolon to end the statement.

```
	void Awake () {
		Quaternion.Euler;
	}
```

The method has parameters used to describe the desired rotation. In this case we'll provide a comma-separated list of containing three arguments, all between round brackets, after the method name. We supply three numbers for the X, Y, and Z rotations. Use zero for the first two and −30 for the Z rotation.

```
		Quaternion.Euler(0, 0, -30);
```

The result of this invocation is a `Quaternion` struct value containing a 30° clockwise rotation around the Z axis, matching hour 1 on our clock.

What's a struct?

A struct—short for 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 write `struct` instead of `class`.

To apply this rotation to the hour arm assign the result of `Quaternion.Euler` to `hoursPivots.localRotation`, using the `=` assignment statement.

```
		hoursPivot.localRotation = Quaternion.Euler(0, 0, -30);
```

What's the difference between `localRotation` and `rotation`?

The `localRotation` property represents the rotation described by the `Transform` component in isolation, thus it is a rotation relative to its parent. It's the rotation that you see in its inspector. In contrast, the `rotation` property represents the final rotation in world space, taking the entire object hierarchy into account. Setting that property would produce weird results if we rotate the clock as a whole, because the arm would ignore that as the property compensates for the rotation of the clock.

Shouldn't there be a warning that `hoursPivot` is never initialized?

The compiler can detect that no code assigns anything to the field and could indeed issue such a warning, because it is unaware that we set it up via Unity's inspector. However, this warning is suppressed by default. The suppression can be controlled via the project settings. There's a *Suppress Common Warnings* toggle under *Player / Other Settings / Script Compilation*. It suppresses warnings about both uninitialized and unused private fields.

Now enter play mode in the editor. You can do this via *Edit / Play*, the indicated keyboard shortcut, or by pressing the play button at top center of the editor window. Unity will switch focus to the game window, which renders what the *Main Camera* in the scene sees. The clock component will awaken, and the clock will be set to one o'clock.

Always one o'clock in play mode.

If the camera isn't focused on the clock you can move it so the clock becomes visible, but keep in mind that the scene is reset when exiting play mode, so any changes that you make to the scene while in play mode will not persist. This isn't true for assets though, changes to them always persist. You can also have the scene window—or even multiple scene and game windows—open while in play mode. Exit play mode before continuing.

3.7 Getting the Current Time

The next step is to figure out the current time when we awaken. We can use the `DateTime` struct to access the system time of the device we're running on. `DateTime` isn't a Unity type, it is found in the `System` namespace. It is part of the core functionality of the .NET framework, which is what Unity uses to support scripting.

`DateTime` has a `Now` property that produces a `DateTime` value containing the current system date and time. To check whether it's correct we'll log it to the console at the start of `Awake`. We can do that by passing it to the `Debug.Log` method.

```
using System;
using UnityEngine;

public class Clock : MonoBehaviour {

	[SerializeField]
	Transform hoursPivot, minutesPivot, secondsPivot;

	void Awake () {
		Debug.Log(DateTime.Now);
		hoursPivot.localRotation = Quaternion.Euler(0, 0, -30);
	}
}
```

Now we get a timestamp logged each time we enter play mode. You can see it both in the console window and in the status bar at the bottom of the editor window.

3.8 Rotating the Arms

We're getting close to a working clock. Let's again start with the hours. `DateTime` has an `Hour` property that gets us the hours portion of a `DateTime` value. Invoking it on the current timestamp will give us the hour of the day.

```
		Debug.Log(DateTime.Now.Hour);
```

So to have the hours arm show the current hour we have to multiply the −30° rotation by the current hour. Multiplication is done with the asterisk `*` character. We also no longer need to log the current time so can get rid of that statement.

```
		~~//Debug.Log(DateTime.Now.Hour);~~
		hoursPivot.localRotation = Quaternion.Euler(0, 0, -30 * DateTime.Now.Hour);
```

Currently four o'clock in play mode.

To make it clear that we're converting from hours to degrees, we can define an `hoursToDegrees` field containing the conversion factor. The angles of `Quaternion.Euler` are defined as floating-point values, so we'll use the `float` type. Because we already know the number, we can immediately assign it as part of the field declaration. Then multiply with the field instead of the literal `-30` in `Awake`.

```
	float hoursToDegrees = -30;

	[SerializeField]
	Transform hoursPivot, minutesPivot, secondsPivot;

	void Awake () {
		hoursPivot.localRotation =
			Quaternion.Euler(0, 0, hoursToDegrees * DateTime.Now.Hour);
	}
```

What's a `float`?

Computers cannot store all numbers, they have to be representable in their binary memory which consists of bits that are either 0 or 1. This makes it impossible to store many numbers exactly within a finite memory size, for example ⅓, just like how we cannot exactly write that number in decimal notation. The best we can do is write 0.3333333 and stop at some point.

Suppose that we decide to write at most three digits after the dot and only one in front of it. Then ⅓ is approximated by 0.333. If we were to divide ⅓ by 100 then we'd be forced to write 0.003, which means that we lost two digits of precision. To improve precision of small values let's add a separate exponent that indicates the order of magnitude of our number. Then 0.333×10^{-2} can represent ⅓ divided by 100, without losing meaningful digits. And we can use 0.333×10^{2} to represent multiplication by 100 as well, while keeping only a single digit in front of the dot. Thus the dot can be considered to float, as it doesn't designate a fixed order of magnitude. This allows us to use only a few digits to represents a large amount of numbers.

Floating-point numbers work the same way for computers, except that they use binary instead of decimal digits and also have to represent special values like infinities and not-a-number. A `float` is such a value stored in four bytes, which means that it has 32 bits.

If we declare a whole number without a suffix then it's assumed to be an integer, which is a different value type. Although the compiler converts them automatically, let's make explicit that all our numbers are of type `float`, by adding the f suffix to them.

```
	float hoursToDegrees = -30f;

	[SerializeField]
	Transform hoursPivot, minutesPivot, secondsPivot;

	void Awake () {
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * DateTime.Now.Hour);
	}
```

The amount of degrees per hour is always the same. We can enforce this by adding the `const` prefix to the declaration of `hoursToDegrees`. This turns it into a constant instead of a field.

```
	const float hoursToDegrees = -30f;
```

What's special about const values?

The const keyword indicates that a value will never change and doesn't need to be a field. Instead, its value will be computed during compilation and is substituted for all usage of the constant. This is only possible for primitive types like numbers.

Let's give the same treatment to the other two arms, using the appropriate properties of DateTime. Both a minute and a second are represented by a rotation of negative six degrees.

```
	const float hoursToDegrees = -30f, minutesToDegrees = -6f, secondsToDegrees = -6f;

	[SerializeField]
	Transform hoursPivot, minutesPivot, secondsPivot;

	void Awake () {
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * DateTime.Now.Hour);
		minutesPivot.localRotation =
			Quaternion.Euler(0f, 0f, minutesToDegrees * DateTime.Now.Minute);
		secondsPivot.localRotation =
			Quaternion.Euler(0f, 0f, secondsToDegrees * DateTime.Now.Second);
	}
```

Currently 5:16:31.

We're using DateTime.Now three times, to retrieve the hour, minute, and second. Each time we go through the property again, which requires some work, which could theoretically result in different time values. To make sure that this doesn't happen, we should retrieve the time only once. We can do this by declaring a variable inside the method and assign the time to it, then use this value afterwards. Let's name it time.

What's a variable?

A variable acts like a field, except that it exists only while a method is being executed. It belongs to the method, not the class.

```
	void Awake () {
		DateTime time = DateTime.Now;
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * time.Hour);
		minutesPivot.localRotation =
			Quaternion.Euler(0f, 0f, minutesToDegrees * time.Minute);
		secondsPivot.localRotation =
			Quaternion.Euler(0f, 0f, secondsToDegrees * time.Second);
	}
```

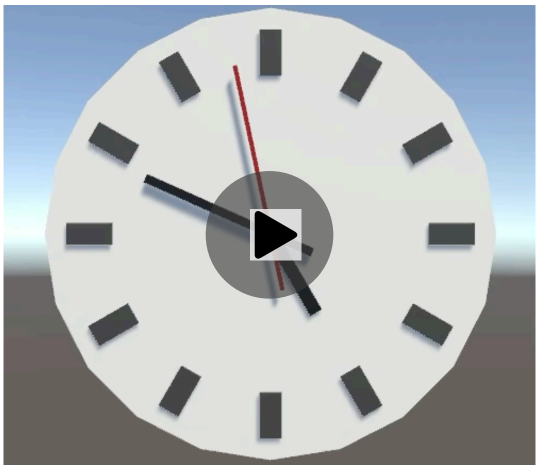
In case of a variable it's possible to omit the type declaration, replacing it with the `var` keyword. This can shorten the code but is only possible when the variable's type can be inferred from what is assigned to it when it is declared. Also, I prefer to only do this when the type is explicitly mentioned in the statement, which is the case here.

```
		var time = DateTime.Now;
```

3.9 Animating the Arms

We get the current time when entering play mode, but after that the clock remains motionless. To keep the clock synchronized with the current time, change the name of our `Awake` method to `Update`. This is another special event method that gets invoked by Unity every frame instead of just once, as long as we stay in play mode.

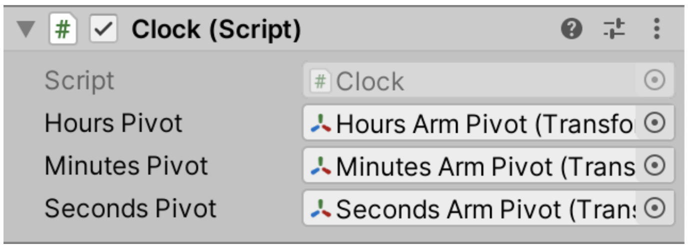
```
	void Update () {
		var time = DateTime.Now;
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * time.Hour);
		minutesPivot.localRotation =
			Quaternion.Euler(0f, 0f, minutesToDegrees * time.Minute);
		secondsPivot.localRotation =
			Quaternion.Euler(0f, 0f, secondsToDegrees * time.Second);
	}
```


Updating clock.

What's a frame?

While in play mode Unity continually renders the scene from the point of view of the main camera. Once rendering is done the result is presented to the display. The display will then show that frame until it gets the next one. Before rendering a new frame everything gets updated. So Unity goes through a sequence of update, render, update, render, and so on. A single update step followed by rendering the scene once is typically considered a single frame, though in reality the timing is more complicated.

Note that our `Clock` component has gained a toggle in front of its name in the inspector. This allows us to disable it, which prevents Unity from invoking its `Update` method.


Clock (Script)

Script	Clock
Hours Pivot	Hours Arm Pivot (Transfo
Minutes Pivot	Minutes Arm Pivot (Trans
Seconds Pivot	Seconds Arm Pivot (Tran:

`Clock` *component that can be disabled.*

3.10 Continuously Rotating

The arms of our clock indicate exactly the current hour, minute, or second. It behaves like a digital clock, discrete but with arms. Typically clocks have slowly-rotating arms that provide an analog representation of time. Let's change our approach so our clock becomes analog.

DateTime doesn't contain fractional data. Fortunately, it does have a TimeOfDay property. This gives us a TimeSpan value that contains the data in the format that we need, via its TotalHours, TotalMinutes, and TotalSeconds properties.

Begin by getting the TimeOfDay struct value from DateTime.Now and store that in the variable instead. As the TimeSpan type isn't mentioned in this statement I'll make the variable's type explicit. Then adjust the properties that we use to rotate the arms.

```
	void Update () {
		TimeSpan time = DateTime.Now.TimeOfDay;
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * time.TotalHours);
		minutesPivot.localRotation =
			Quaternion.Euler(0f, 0f, minutesToDegrees * time.TotalMinutes);
		secondsPivot.localRotation =
			Quaternion.Euler(0f, 0f, secondsToDegrees * time.TotalSeconds);
	}
```

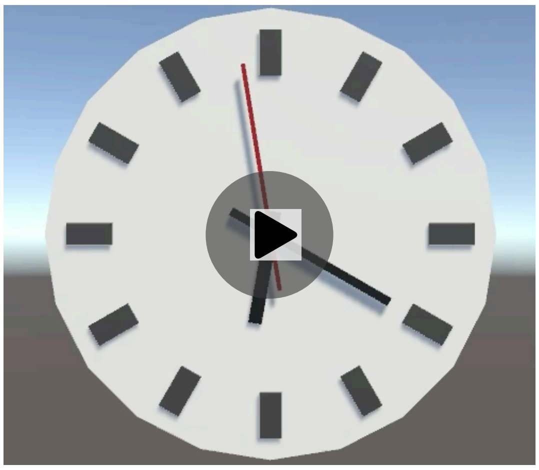
This will result in compiler errors, complaining that we cannot convert from double to float. This happens because the TimeSpan properties produce values with the double-precision floating point type, known as double. These values provide higher precision than float values, but Unity's code only works with single-precision floating point values.

Is single precision enough?

For most games, yes. It becomes a problem when working with very large distances or scale differences. Then you'll have to apply tricks like teleportation or camera-relative rendering to keep the active area near the world origin. While using double precision would solve this problem, it would also double the memory size of the numbers involved, which leads to other performance problems. Game engines typically use single-precision floating-point values, and so do GPUs.

We can solve this problem by explicitly converting from double to float. This process is known as casting and is done by writing the new type within round brackets in front of the value to be converted.

```
		hoursPivot.localRotation =
			Quaternion.Euler(0f, 0f, hoursToDegrees * (float)time.TotalHours);
		minutesPivot.localRotation =
			Quaternion.Euler(0f, 0f, minutesToDegrees * (float)time.TotalMinutes);
		secondsPivot.localRotation =
			Quaternion.Euler(0f, 0f, secondsToDegrees * (float)time.TotalSeconds);
```


Analog clock.

Now you know the fundamentals of object creation and writing code in Unity. The next tutorial is Building a Graph.