Dice Game

Here we will work together to create the basic template for a dice game.

Basic Layout

Before we jump right into coding, we need to plan out our work. Though it can be really fun to just start coding, but without a plan it can be very difficult to break apart a big problem like "make a dice game".

First, let's go over what our dice game is :

> Two players take turns rolling dice. The value of their rolls will move them closer to winning, perhaps damaging an enemy. With each roll a story of sorts is also printed out. Once a player wins, display their victory for all to see.

Think of it similar to a game of Dungeons and Dragons.
With this in mind let us dissect the problem into smaller chunks. We'll start it off here than hopefully you can take over and do the rest yourself.

1. Have two "players" with a value of sorts to keep track of (EX : health).
2. Alternate between the players turns generating a random value for each.

Hopefully you can see the pattern. Take the big problem and try to break it into a smaller task.
Try to write down the rest of what we might need. Once you are done or if you get stuck take a peek below.

▶ The pieces of the puzzle

With this plan in mind we can start building a template. Follow along with this to get the idea of how we are building this game. Afterwards you will have this as a template to work with and expand for the actual dice game.

The Players

To start let's first make a new C# Script or even a new project if you'd like a fresh start.
Give this script the name "DiceGameTemplate" or similar. Open it up and let's start coding!

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

public class DiceGameTemplate : MonoBehaviour {

        // Start is called before the first frame update
        void Start() {

        }
```

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

        }

}
```

For the template we are going to use very generic names and story. Make sure to be more creative for your own.
The first thing we said we would need is to keep track of values for two players. Pretty simply, we will make a variable for each player.
Now it is important where we put those variables. We want to keep these vars throughout this script and not "re-instantiate" (think of that as resetting). Remembering what we talked about during the scope section of the last lesson we said that if a scope ends, such as the end of a function or `if` statement, any variables within it are deleted.
To prevent this let's declare these variables at the top of the class like so :

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

public class DiceGameTemplate : MonoBehaviour {

        int playerOneHealth = 50;
        int playerTwoHealth = 50;

        // Start is called before the first frame update
        void Start() {

        }

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

        }
```

```
}
```

Declaring your variables here in what we will call the "class scope" means that the var can be accessed anywhere within this script. It also means that it will only be declared once when our game object with this script is loaded into the game. Remembering that we need to attach this script to an object in our game, the class `DiceGameTemplate` will only be created this once. In turn those two variables are only declared once.
Here we've used `int` as the type because it makes sense for a "health" value and 50 is just an arbitrary starting point. That's a fine starting point and does what we needed to do in the first part of the breakdown of our problem.

Taking Turns

The next part of our problem involves alternating between our players turns. Let's start with figuring out how we should go about this.
This may not be inherently obvious, but we are going to use another `int` and check if it is even or odd. This is a very common method.
Start by creating another variable called `turnCounter` or similar. It will be an `int` and have an initial value of `0`.

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

public class DiceGameTemplate : MonoBehaviour {

        int playerOneHealth = 50;
        int playerTwoHealth = 50;
        int turnCounter = 0;

        // Start is called before the first frame update
        void Start() {

        }
```

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

        }

}
```

Then as we said, we need to check it `turnCounter` is even or odd. We can do this with the "modulus operator".
The modulus or remainder operator gives us the remainder of division between two numbers. It's used like `5 % 2` will return `1` because `2` can go into `5` twice with one left over.
Perhaps you can see just from that how we can check for evenness. All we need to do is take the remainder of two. If it returns one, the number is odd. If it returns zero, the number is even. We can test this like so :

```
3 % 2; // Equal to 1
6 % 2; // Equal to 0
0 % 2; // Equal to 0
13 % 2; // Equal to 1
```

With this info we can test if a number is even using `if(num % 2 == 0)` where `num` is the number we are checking. Using this in our code will look like this : (full script omitted for brevity)

```
// Start is called before the first frame update
void Start() {

        if(turnCounter % 2 == 0) {
                // Player One's Turn
        } else {
                // Player Two's Turn
        }

}
```

Just put this code into the `Start` function for now. Now we do need to update the `turnCounter` value otherwise it will stay zero forever. We want to always increment `turnCounter` regardless of whose turn it is, so we should increment the value **After** the `if` and `else` statement.

```
// Start is called before the first frame update
void Start() {

        if(turnCounter % 2 == 0) {
                // Player One's Turn
        } else {
                // Player Two's Turn
        }

        turnCounter++;

}
```

That should be all for determining who's turn it is. Onto the next bit of the turns, generating the random numbers.

Random Numbers

After getting our turns setup, we need to actually do something in our turns. Otherwise, this would be one boring game.
We now need to "roll some dice". We can do this using a pre-made Unity class called "Random". The Random class has a method perfect for us called "Range".
If you open up that page you can read a bit about how it works. You can also see from the examples how we need to access this function. First we need to reference the class, `Random`, then access the `Range` function with a `.` like so `Random.Range();`.
Now this will error because we have not given the function any parameters. If you look at the Scripting API it will tell you that `Range` needs two numbers. These can be either `float` or `int` types, but seeing as we are simulating dice we will only want whole numbers.
Let's start typing it out.

```
// Start is called before the first frame update
void Start() {

        int dieOne = Random.Range(1, 7);

        if(turnCounter % 2 == 0) {
                // Player One's Turn
        } else {
                // Player Two's Turn
        }

        turnCounter++;

}
```

Alright that might seem a little weird so let's pick it apart.
First thing is that we want to store this random number into a variable (`int dieOne` , We've used the suffix `One` here as there will be two dice later) because will need to use this number throughout the turn. If we did not store the number to a variable we'd only be able to access it once. Every time we call the `Range` function it creates a **new** number, so we need to use a variable to access it more than once.
Next is the placement. We've put it above the `if` statements because it doesn't matter who's turn it is we always need a random number. This also makes it accessible throughout the rest of the turn and in any `if` s we have.
The last bit is the values we've passed in, `1` and `7` . Now you could pass whatever you would like into this, here we are simply emulating a six sided die. The reason `1` is the lower limit and `7` is the upper limit is because the Scripting API tells us the integer version of `Range` , which we are using, has an inclusive minimum and exclusive maximum.
What this means is the first number passed in, here being `1` , is going to be inclusive. `Range` **can** return `1` .
The second number, here being `7` , is going to be exclusive. `Range` **cannot** return `7` but can return up to it.
Because of these inclusive and exclusive rules the possible numbers we can get back are `1, 2, 3, 4, 5, 6` when we input `(1, 7)` into range. As a more flexible definition think of it like this.

> When we pass in the parameters `min` and `max` to `Range` we can receive a value from and including `min` up to and including `max - 1` .

All of this boils down to having `dieOne` set to a random number between one and six at the start of a players turn.

Turn Logic

Now that we have a random number and have figured out whose turn it is, we ought to do something.
In this template we will use the number rolled to determine the "power" of an attack and take that away from the players' health. If that makes sense it's encouraged that you try doing this on your own first and see how it goes. You can always come back after and use the methods listed here. There isn't really a wrong way to do this so go ahead and try it!
First let's list out what we'd like each dice roll to do. For this template how about :

- `1` & `2` deal `5` damage.
- `3` & `4` deal `10` damage.
- `5` deals `15` damage.
- `6` will deal `30` damage!

Chose to do whatever you'd like for these.

Structure

Now that we've got that plan lets start implementing it. We will simply use `if else` statements to decide on the damage. The main thing we need to consider with this bit of logic is the order to do it in. If we lay out our if statements right we can make them simpler and faster.
Let's get the structure down.

```
// Start is called before the first frame update
void Start() {

        int dieOne = Random.Range(1, 7);

        if(turnCounter % 2 == 0) {

                // Player One's Turn
                if(dieOne == 6) {
```

```
                } else if(dieOne == 5) {

                } else if(dieOne >= 3) {

                } else {

                }

        } else {
                // Player Two's Turn
        }

        turnCounter++;

}
```

We've made a few arbitrary choices here and a few important ones.

To start with the necessary choices. This is mainly the fact that we are using `if else` statements rather than simply `if` statements.
The reason for this is that if the die has one value, that is the only value we should use. Meaning if `dieOne` is `5` we only want to use the logic in `5` 's section of the if statement and not the `3` and `4` section obviously. By using `else` statements we can make sure only one thing will happen per die value.
If we were to re-write this without the `else` 's it would look like this :

```
if(dieOne == 6) {

}
if(dieOne == 5) {

}
if(dieOne >= 3) {
```

```
  }
  if(dieOne >= 1) {

  }
```

Now if `dieOne` was set to `4` the first two statements would fail and not execute their code. Then it would hit the third statement and it is true and execute that code. Then because there is no `else` statements, `dieOne` would be checked against the fourth `if` statement. Since `4` is greater or equal to `1` the code within that `if` would execute.
As you can see, this logic is flawed for our purpose as multiple `if` 's can be true, and we do not want that, thus, we should use `else` statements. Of course, you can use just `if` 's but the logic needs to be laid out differently.

Now for the arbitrary choice.
This would be the logic we have put inside our `if` statements. Let's start by saying you could write this any way you'd like and if it works, it pretty well should be good.
Here we've started with checking `6` directly with a `==` . This is simply because we want to check if the number rolled was a `6` . Then the same goes for `5` . In our little list above we had `6` and `5` separate so it makes sense to test for them individually.
After that we have `dieOne >= 3` . This checks to see if `dieOne` is `3` or above. As we showed above, if the number is `5` or `6` you may think that this will still trigger here. But remember, because we've used `else` statements only one of them may execute. Because we only have to check this third statement if both of the first two fail we can be certain the number is not `5` or `6` .
The same goes for the last `else` statement. We do not strictly need an `if` here because if all the above checks have failed the only options left are `1` and `2` so it must be true. Of course, you can just as well put `if(dieOne >= 1)` and nothing will change.
There are really countless ways to do this. You can check every number with a `==` if you'd like. As long as it works when you test it. What we have here would be considered the "right" way of doing this.

Now that we've got the structure laid out let's start doing some damage, literally!

Dealing Damage

Now that our turn is set up properly we can start actually doing something with our dice rolls. This bit is nice and easy as we will simply subtract the values we stated above from the players' health.
This is all we will do in the example but if you can think of a different set of rules to go with try it out. Perhaps dealing damage times the die roll or use two dice and add in bonus damage if you roll snake eyes, damage based on current health, etc.
We simply need to add `playerVar -= damageValue` to each of our if statements while subbing in the correct values.

Note : This is player ONE's turn, so we should damage player TWO.

```
if(dieOne == 6) {
        playerTwoHealth -= 30;
} else if(dieOne == 5) {
        playerTwoHealth -= 15;
} else if(dieOne >= 3) {
        playerTwoHealth -= 10;
} else {
        playerTwoHealth -= 5;
}
```

Note we are just working within player one's turn for now.

Now when we roll a die we should see our player's health decrease by that much. If you were to run the game right now you won't see anything though. Let's remedy that by actually printing out what happened.

Printing A Story

Again this bit should not be too hard. We will simply use `Debug.Log` to write out some messages.

```
if(dieOne == 6) {

        playerTwoHealth -= 30;
        Debug.Log("Player two has taken 30 damage!");

}
```

This is fairly simple and works well for our purposes, but we will leave an extra note here.
If your damage is **not** constant you will need to do something along the lines of :

```
if(dieOne == 6) {

        int damageToDeal = (Put some math here);
        playerTwoHealth -= damageToDeal;
        Debug.Log("Player two has taken " + damageToDeal + " damage!");

}
```

Refer back to the variables lesson for more info on why we would do this. You could also use this to say "they have X health left".

Now we can just expand our code to print every time we deal damage.

```
if(dieOne == 6) {

        playerTwoHealth -= 30;
        Debug.Log("Player two has taken a big hit of 30 damage!");

} else if(dieOne == 5) {

        playerTwoHealth -= 15;
```

```
            Debug.Log("Player two has taken 15 damage.");

    } else if(dieOne >= 3) {

            playerTwoHealth -= 10;
            Debug.Log("Player two has taken 10 damage.");

    } else {

            playerTwoHealth -= 5;
            Debug.Log("Player two has taken a measly 5 damage.");

    }
```

You will have to make your messages more unique for your own dice game. We've done a little of that here.
You can also add messages to the start of the game or anywhere else in order to flesh out the story. Perhaps `Welcome to the game` at the beginning of the start function.
You can now go and take this logic for damage and printing and copy and paste it into player two's turn. Remember, you will have to change the variable names.

Winning The Game

Of course, we can't really have a game if you can't win. We should check after dealing damage if either player's health is zero or below. Again this should be just a simple `if` .

```
// Start is called before the first frame update
void Start() {

        int dieOne = Random.Range(1, 7);

        if(turnCounter % 2 == 0) {
                // Code Omitted for brevity
        } else {
                // Code Omitted for brevity
```

```
        }

        if(playerOneHealth <= 0 || playerTwoHealth <= 0) {
                Debug.Log("The game ended.");
        }

        turnCounter++;

}
```

Now obviously for this template we're being rather basic. For yours make sure the message is more interesting than "Game Over".
We've put this after dealing the damage because checking for a win is not specific to whose turn it is.
Hopefully using `<= 0` is fairly obvious. We are seeing if the player's health is zero or less. We do need to check both players as the game ends whenever one of them dies. This can be done with one `if` using an "or" (`||`) to combine the two checks. If you don't remember what the or does check back to here.

Looping Through Turns

If you play your game right now it will likely be very boring. There will only be one turn printed. This is because our code is in the start function which is only run once.
Since we want to loop through the turns until one player dies we will use a while loop.
We need to wrap our entire turn in this loop like so :

```
// Start is called before the first frame update
void Start() {

        while(/*Boolean Statement*/) {

                int dieOne = Random.Range(1, 7);

                if(turnCounter % 2 == 0) {
                        // Code Omitted for brevity
```

```
                } else {
                        // Code Omitted for brevity
                }

                if(playerOneHealth <= 0 || playerTwoHealth <= 0) {
                        Debug.Log("The game ended.");
                }

                turnCounter++;

        }

}
```

We aren't using a `for` loop because we don't know how many turns we will have. Because of this `while` loops will be much nicer.
Now we just need to figure out what to put in our `while` loop for a boolean statement. Well, we want to keep taking turns until one player dies and we already did that.
Using the same boolean statement we used when checking if the game ended we can set up our loop to go until someone wins like this :

```
// Start is called before the first frame update
void Start() {

        while(playerOneHealth > 0 && playerTwoHealth > 0) {

                int dieOne = Random.Range(1, 7);

                if(turnCounter % 2 == 0) {
                        // Code Omitted for brevity
                } else {
                        // Code Omitted for brevity
                }

                if(playerOneHealth <= 0 || playerTwoHealth <= 0) {
```

```
                        Debug.Log("The game ended.");
                }

                turnCounter++;

        }

}
```

You might notice that now our check for a win is a little redundant seeing as the loop will stop if someone wins. So we can just move our print to after the loop and it will be printed when someone wins.

```
// Start is called before the first frame update
void Start() {

        while(playerOneHealth > 0 && playerTwoHealth > 0) {

                int dieOne = Random.Range(1, 7);

                if(turnCounter % 2 == 0) {
                        // Code Omitted for brevity
                } else {
                        // Code Omitted for brevity
                }

                turnCounter++;

        }

        Debug.Log("The game ended.");

}
```

Woo hoo! We should have a working dice game now. In the next lesson we'll talk about the criteria for your dice game and you'll set off to make a wonderful game.

Advanced Layout

Only do this section if you feel confident with coding.

There are a few things we can do to our code to make it run a bit better. None of these are required as you can see that your game works just fine without them. Here we will talk about these things in order to make our code better or more practical. The main thing to go over here are functions.
If you take a look at the code we have here you may notice something.

```
// Start is called before the first frame update
void Start() {

        while(playerOneHealth > 0 && playerTwoHealth > 0) {

                int dieOne = Random.Range(1, 7);

                if(turnCounter % 2 == 0) {

                                // Player One's turn
                        if(dieOne == 6) {

                                playerTwoHealth -= 30;
                                Debug.Log("Player two has taken a big hit of 30 damage!");

                        } else if(dieOne == 5) {

                                playerTwoHealth -= 15;
                                Debug.Log("Player two has taken 15 damage.");

                        } else if(dieOne >= 3) {
```

```
                playerTwoHealth -= 10;
                Debug.Log("Player two has taken 10 damage.");

            } else {

                playerTwoHealth -= 5;
                Debug.Log("Player two has taken a measly 5 damage.");

            }

        } else {

            // Player Two's turn
            if(dieOne == 6) {

            playerOneHealth -= 30;
            Debug.Log("Player one has taken a big hit of 30 damage!");

            } else if(dieOne == 5) {

                playerOneHealth -= 15;
                Debug.Log("Player one has taken 15 damage.");

            } else if(dieOne >= 3) {

                playerOneHealth -= 10;
                Debug.Log("Player one has taken 10 damage.");

            } else {

                playerOneHealth -= 5;
                Debug.Log("Player one has taken a measly 5 damage.");

            }

        }
```

```
                turnCounter++;

        }

        Debug.Log("The game ended.");

}
```

If you ask me, it looks very repetitive and that signals that we should likely use a function. That's what we will do here. Let's take a closer look at the bit of code we'd like to turn into a function.

```
if(dieOne == 6) {

        playerHealth -= 30;
        Debug.Log("Player has taken a big hit of 30 damage!");

} else if(dieOne == 5) {

        playerHealth -= 15;
        Debug.Log("Player has taken 15 damage.");

} else if(dieOne >= 3) {

        playerHealth -= 10;
        Debug.Log("Player has taken 10 damage.");

} else {

        playerHealth -= 5;
        Debug.Log("Player has taken a measly 5 damage.");

}
```

We want to use a function to try to reuse this multiple times. This means we want to use this for both players.
You might have noticed we got rid of the `One` and `Two` 's in that code because of that. We will need to change this up to use variables and dynamically print out those messages. Let's go back and talk about functions quick.

Function Parameters

Functions can take in variables when you call it to and use their values during their execution. Similar to how we've been "passing" a `string` into the function `Debug.Log` to let it print out said `string` .
What do we want to pass into this function? Well if we look at the snippet above, or even put it into your editor to see the errors, we can see that `dieOne` needs to be defined. So we can flip that problem over and say we need to pass in the value of the die roll. With that let's turn this into a function.

```
void decideDamage(int dieRoll) {

        if(dieRoll == 6) {
                playerHealth -= 30;
                Debug.Log("Player has taken a big hit of 30 damage!");
        } else {
                // Rest of the if statement ommitted for brevity
        }

}
```

What we've done is added a parameter to the declaration of our function. This goes inside the parentheses `()` .
This is basically like declaring a variable as it goes type (`int`) and then name (`dieRoll`). We do not need set the value of this variable as we will be passing in a value when we call our function.
Now within the scope of this function we can use and manipulate the variable `dieRoll` . Make sure to use `dieRoll` in place of `dieOne` as that is the variable we are using now.

The next chunk here is making the printed string dynamic so it can display which player has been hit.
To start let's add-in another parameter to our function. We can make this an integer as well and will name it along the lines of "playerNumber". We can do this by putting a comma after the current parameter. Like so :

```
void decideDamage(int dieRoll, int playerNumber) {

}
```

We can now utilize an `int` called `playerNumber` inside our function. Now lets incorporate it into our printed string. Remember we can do this using the `+` operator.

```
void decideDamage(int dieRoll, int playerNumber) {

        if(dieRoll == 6) {
                playerHealth -= 30;
                Debug.Log("Player " + playerNumber + " has taken a big hit of 30 damage!");
        } else {
                // Rest of the if statement ommitted for brevity
        }

}
```

Now that should be setup pretty well. We can use the function to determine the damage to deal to a player (that's good naming).
BUT!
There is another problem that you may have spotted. Our `playerHealth` variable doesn't exist.
"Add it as a parameter", you say.
Sadly that will not work. This is beyond the scope of our lesson but basically the variable we pass into a function (`func(varToPass);`) and the variable the function uses (`func(int myVar)`) are **not** the same. So, if we edit the var inside the function the one we passed in will not change.

Let's move on and finish up this function.

Parameter Scope

As we briefly stated the variable used in our function is not actually the same as the one we passed in. Let's quickly demonstrate this because it can be a bit confusing.
Stick this code into a new script to test this out :

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

public class testClass : MonoBehaviour {

        void Start() {

                int x = 2;
                Debug.Log(x);
                testFunction(x);
                Debug.Log(x);

        }

        void testFunction(int myInt) {

                myInt = 5;

        }

}
```

If you run this script (by putting it onto an object) you should get this output :

```
2
2
```

As you can see the variable `x` was not changed to `5` inside the function.
Now change it to this :

```
void Start() {

        int x = 2;
        testFunction(x);
        Debug.Log(x);

}

void testFunction(int myInt) {

        Debug.Log(myInt);
        myInt = 5;
        Debug.Log(myInt);

}
```

Now the output should be this :

```
2
5
2
```

As demonstrated in these two examples we can use and change a variable **within** a function but as soon as we exit the scope of our function these changes are discarded.
The variable passed into a function is **not** the same as the variable used in the function.

We will need a different way to find a different way to damage our player. Luckily there is a great method.

Function Return Types

Instead of using our variable inside the function we are going to "return" a value and use that to change our health.
Basically the way this works is at the end of our function we can give back a value.
Let's set up our function to return an `int`. We need to change the keyword `void` before the name of our function to `int`.

```
int decideDamage(int dieRoll, int playerNumber)
```

Nice and easy. Now we just need to add a `return` statement or else our function will error. If the return type (the type name before our function name) is not `void` you **must** return a value of the return type.
How do return a value? Easy just say `return` followed by a value. We want to return the amount of damage to deal so let's change our code to this :

```
int decideDamage(int dieRoll, int playerNumber) {

        if(dieRoll == 6) {
                Debug.Log("Player " + playerNumber + " has taken a big hit of 30 damage!");
                return 30;
        } else {
                // Rest of the if statement ommitted for brevity
        }

}
```

We've changed `playerHealth -= 30;` to `return 30;`.
It is key to note that the `return` statement has to be after the `Log` otherwise the `Log` will not be run. When you use `return` the function will exit and no code after the `return` will be run.
You can expand the return values all the way down your `if else` tree on your own.
There is one more thing we need to do in order for this to work. Let's take a look at the whole class.

```
// Start is called before the first frame update
void Start() {
```

```
        while(playerOneHealth > 0 && playerTwoHealth > 0) {

                int dieOne = Random.Range(1, 7);

                if(turnCounter % 2 == 0) {

                        // Player One's turn
                        int dmg = decideDamage(dieOne, 2);
                        playerTwoHealth -= dmg;

                } else {

                        // Player Two's turn
                        int dmg = decideDamage(dieOne, 1);
                        playerOneHealth -= dmg;

                }

                turnCounter++;

        }

}

int decideDamage(int dieRoll, int playerNumber) {

        if(dieRoll == 6) {
                Debug.Log("Player " + playerNumber + " has taken a big hit of 30 damage!");
                return 30;
        } else {
                // Rest of the if statement ommitted for brevity
        }

}
```

As you can see, because our function returns an `int` we can use that value to set a variable. We then use this value to subtract our player's health.

This may have not changed the output of our function in any way but even by just looking at the code you should be able to tell that this is much cleaner and nicer.
If you'd like to bring your game to a higher level look for opportunities such as this to utilize the tools C# provides.

Onto the project!