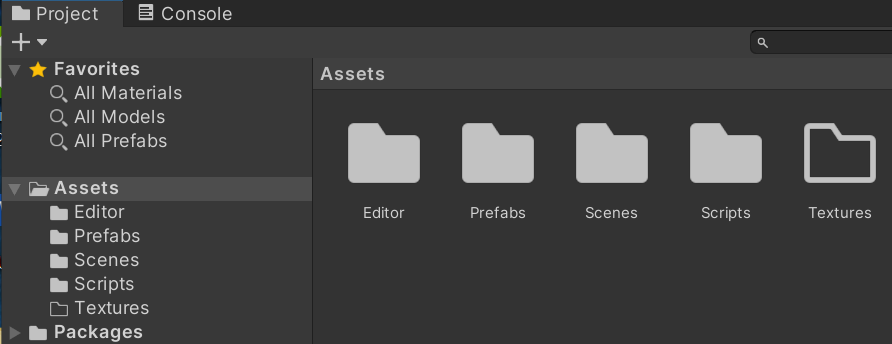
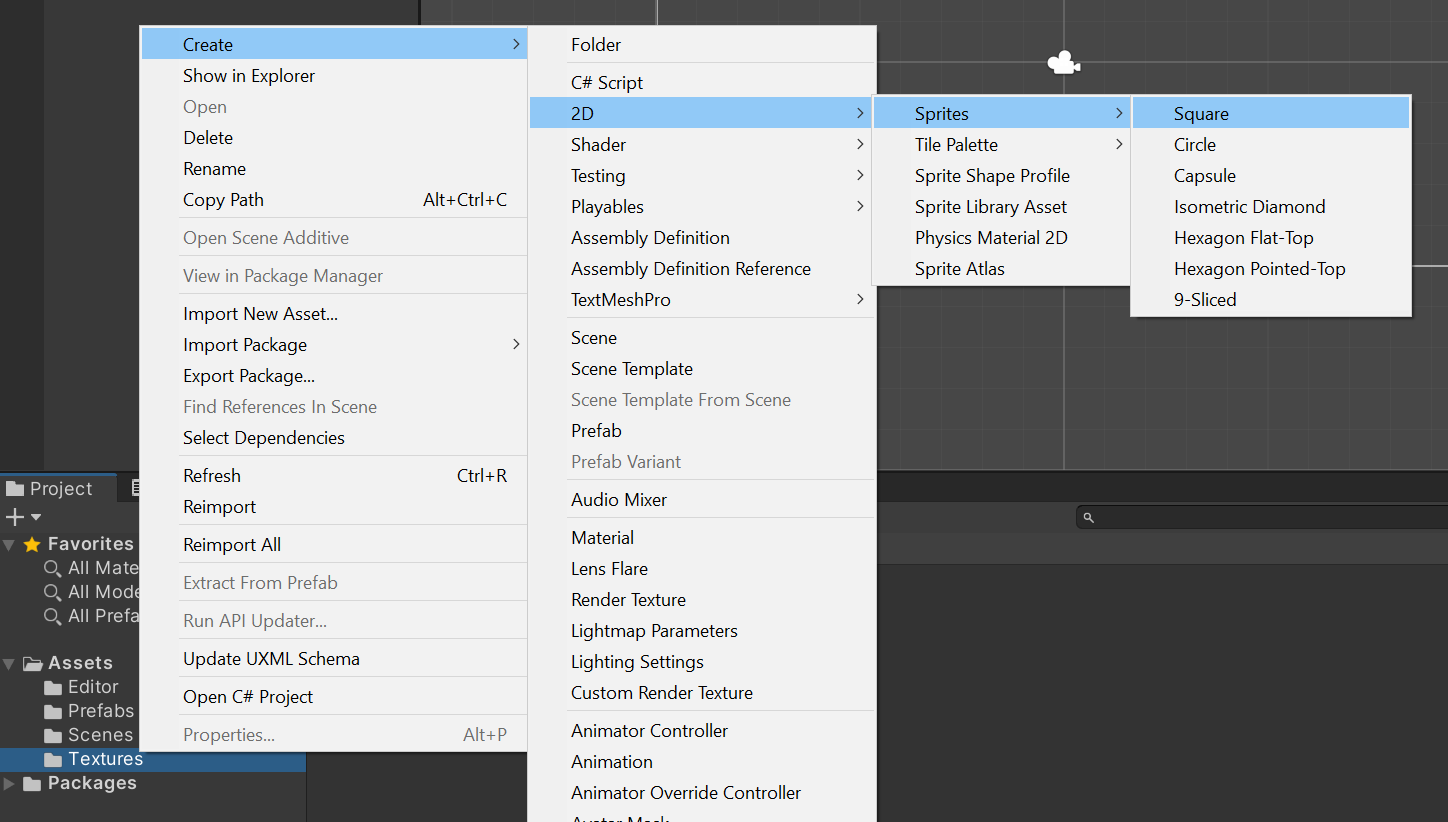
# Milestone 1 – Deliverable 2 Detailed Tutorial

## Textures

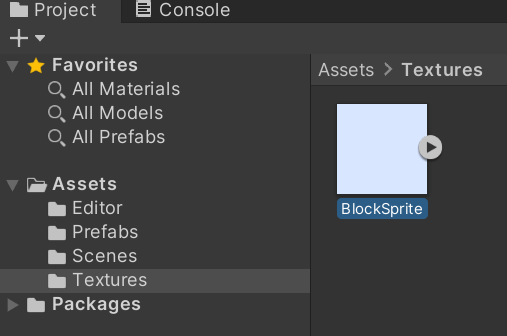
In the Assets folder, create a new folder and name it Textures.



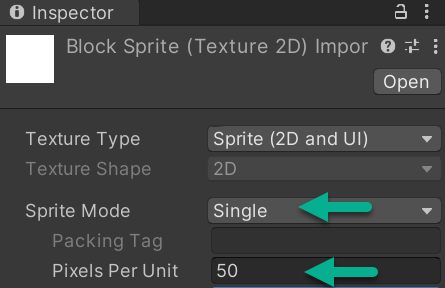
Double-click to open the Textures folder. Right click to Create > 2D > Sprites > Square



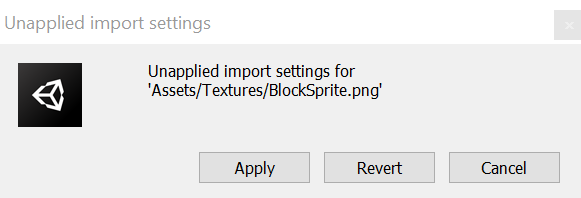
Name the Sprite BlockSprite



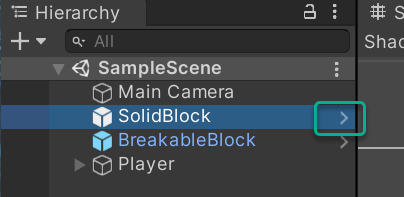
With BlockSprite selected, change the following settings in Inspector > Block Sprite. Set Sprite Mode to Single. Set Pixels per Unit to 50.



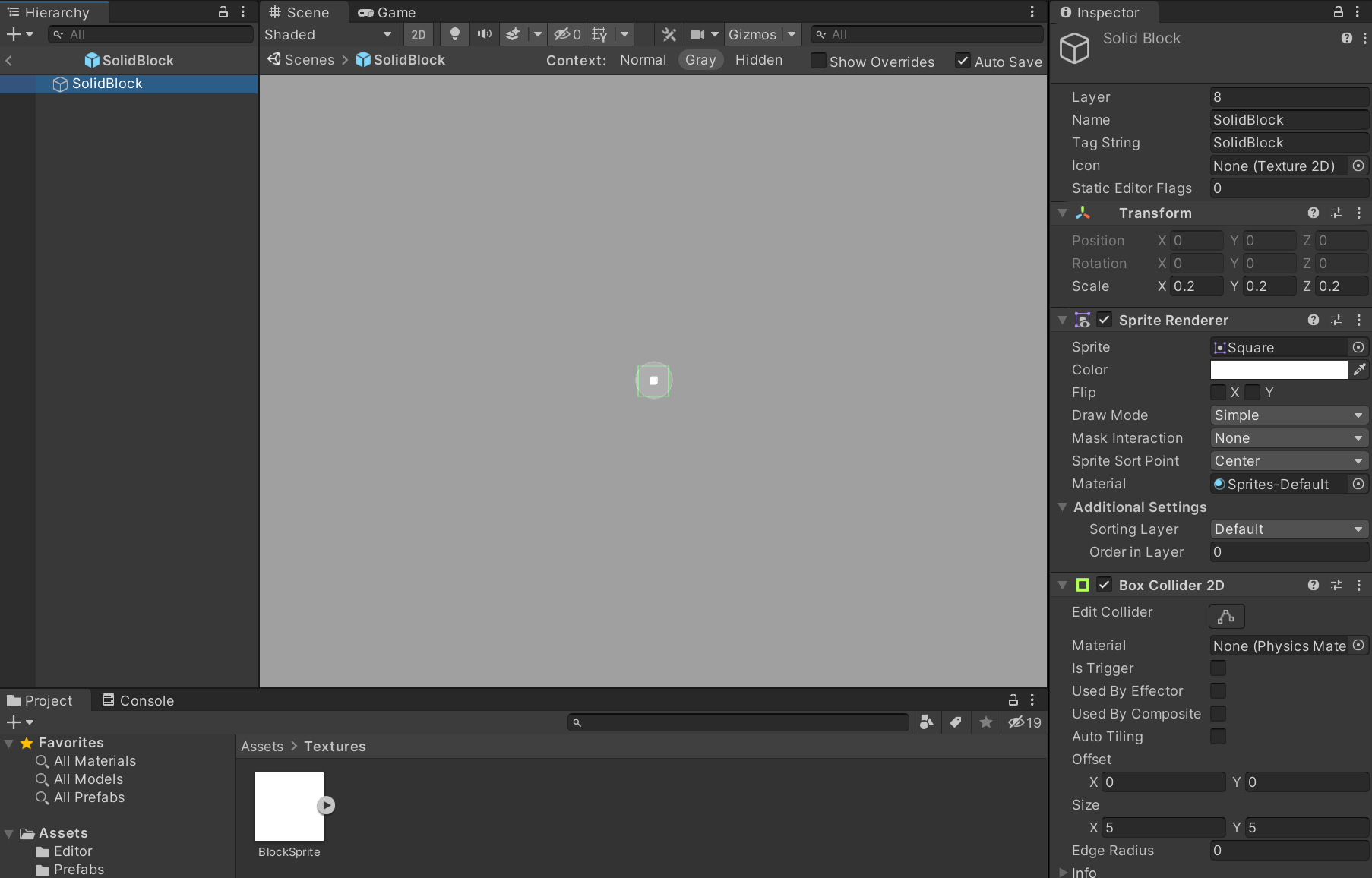
If you see a popup box asking you to apply or cancel to BlockSprite, select Apply. Save your Project.



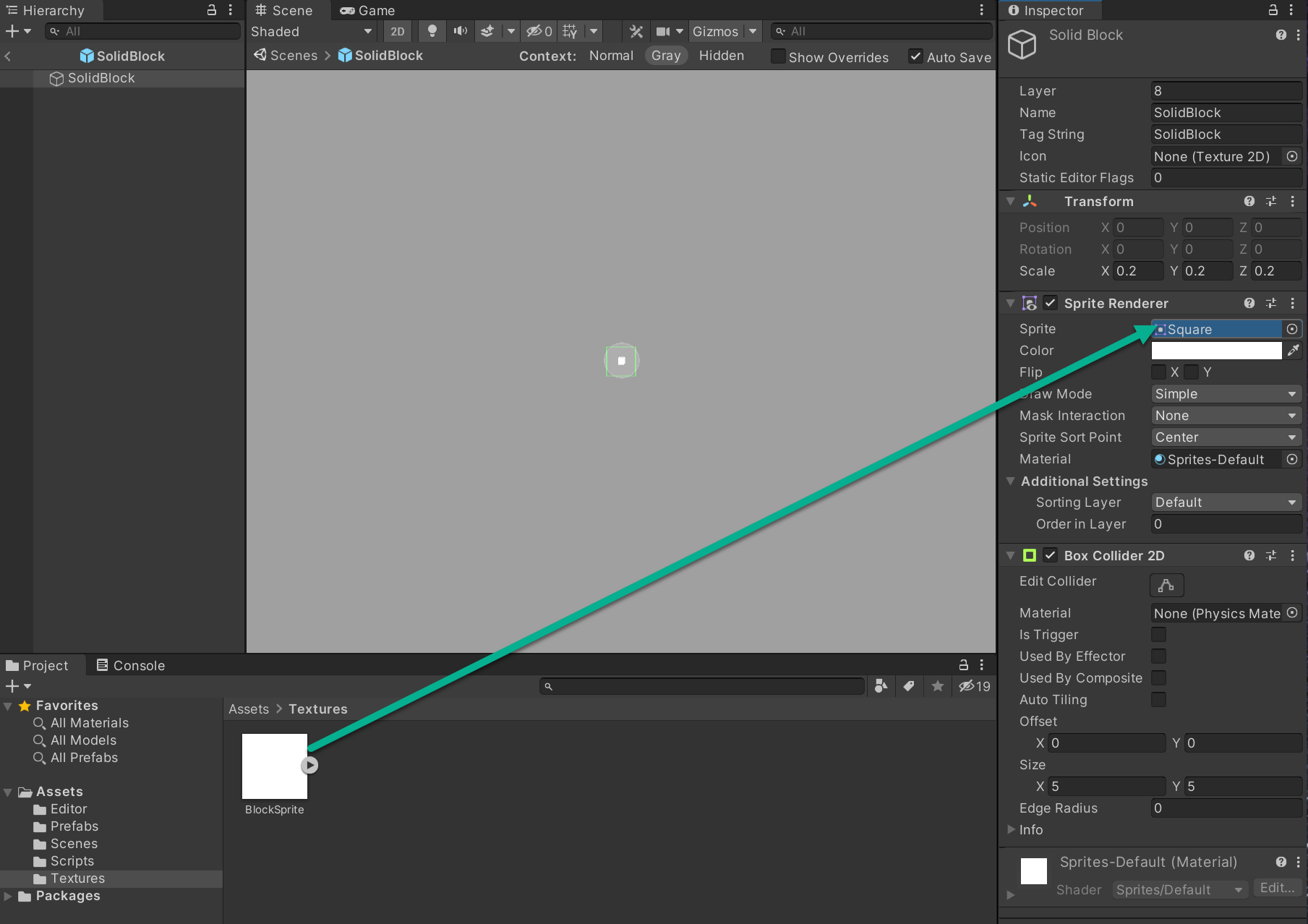
In Hierarchy, select SolidBlock. Select > to the right of SolidBlock. This selection opens SolidBlock in Prefab mode. When you change a prefab directly (in this isolation mode), all instances of a prefab on the scene will get that change.



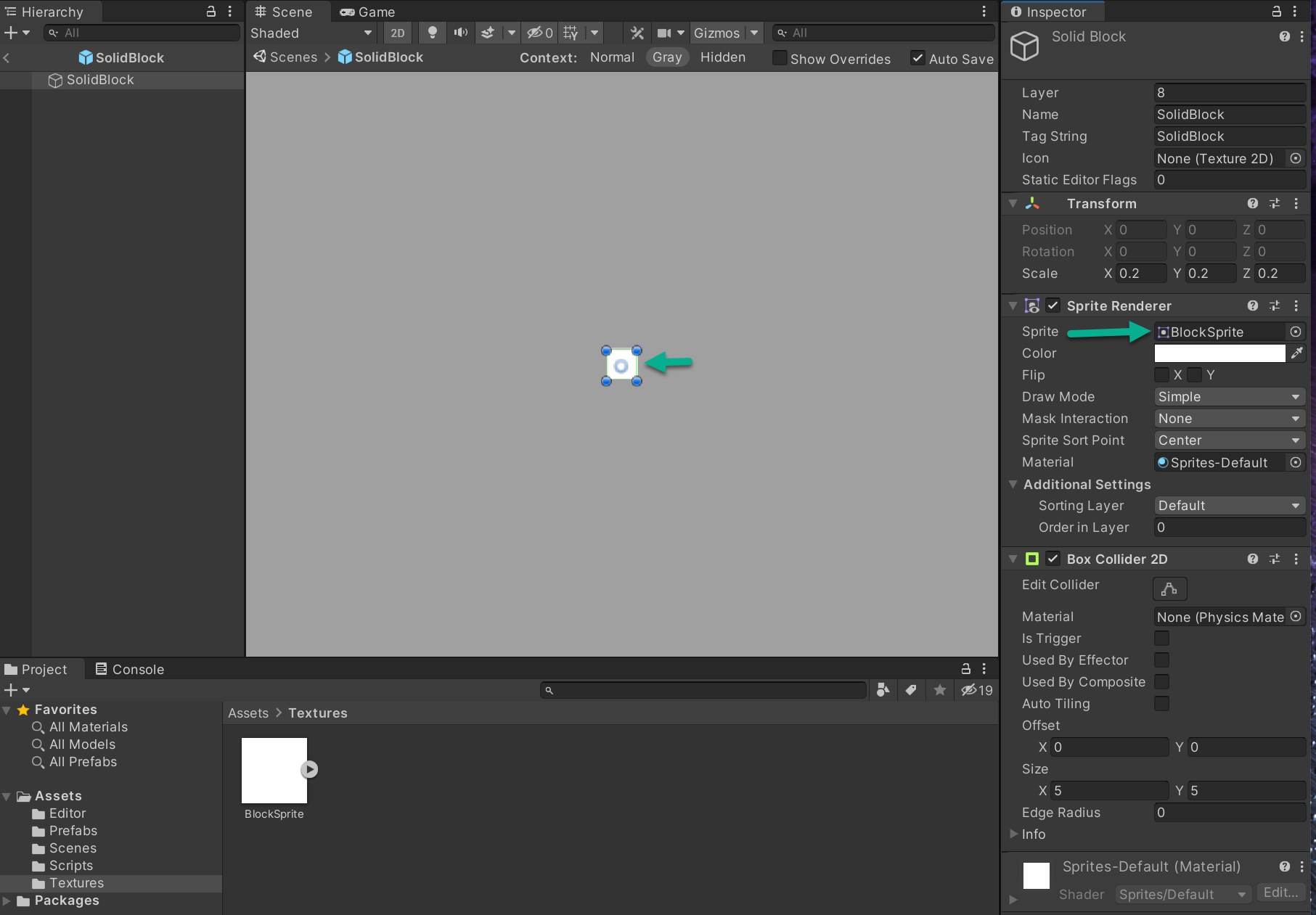
Prefab mode changes your Scene view.



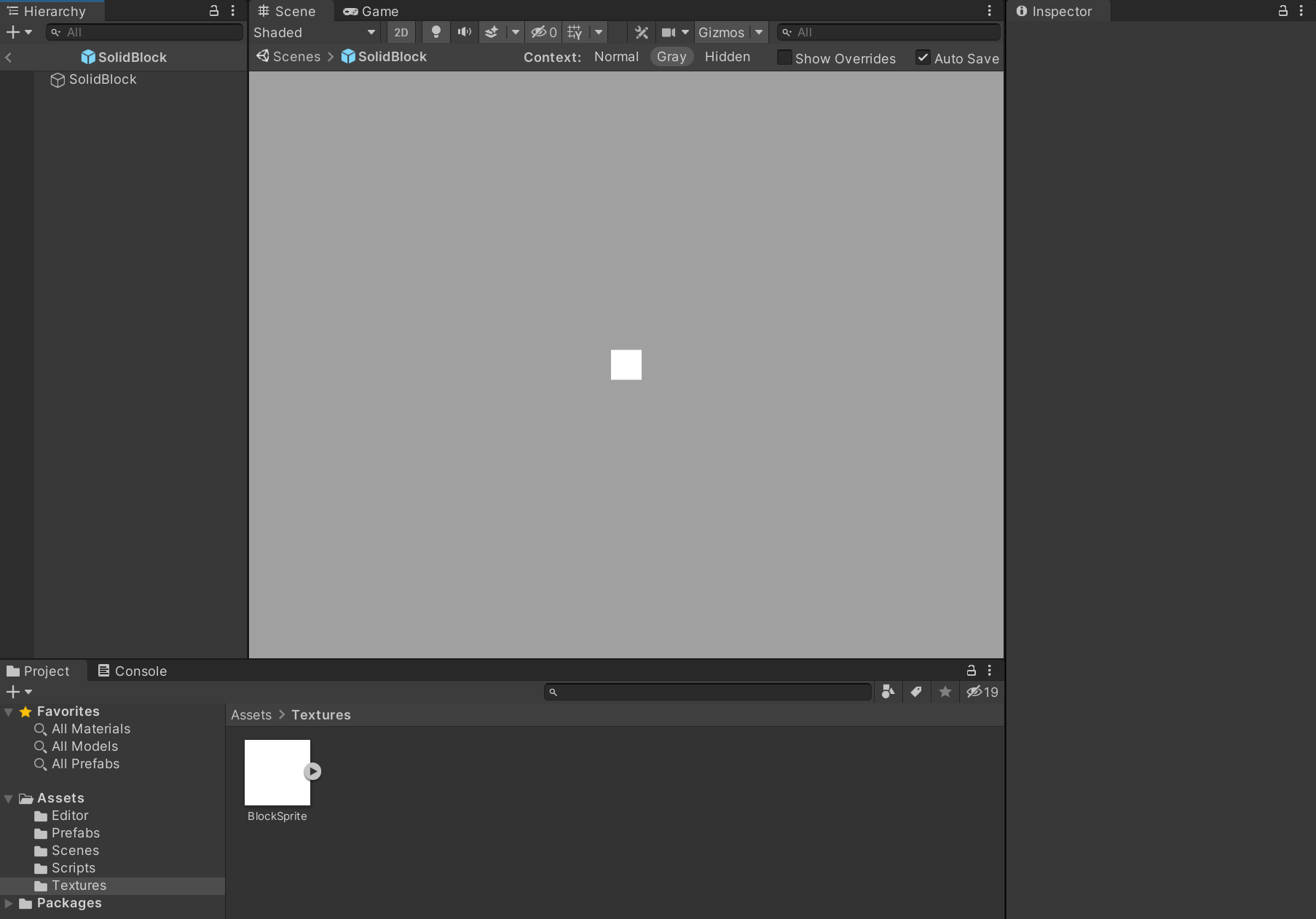
In Hierarchy, select SolidBlock. In Inspector, click on the down arrow on Sprite Renderer to open and view its settings. In Textures, drag and drop BlockSprite to the Inspector > Sprite Renderer field (directly on the word Square. You will see a +) in Sprite Renderer.



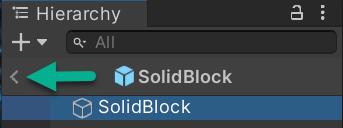
Your Scene changes with BlockSprite selected to:

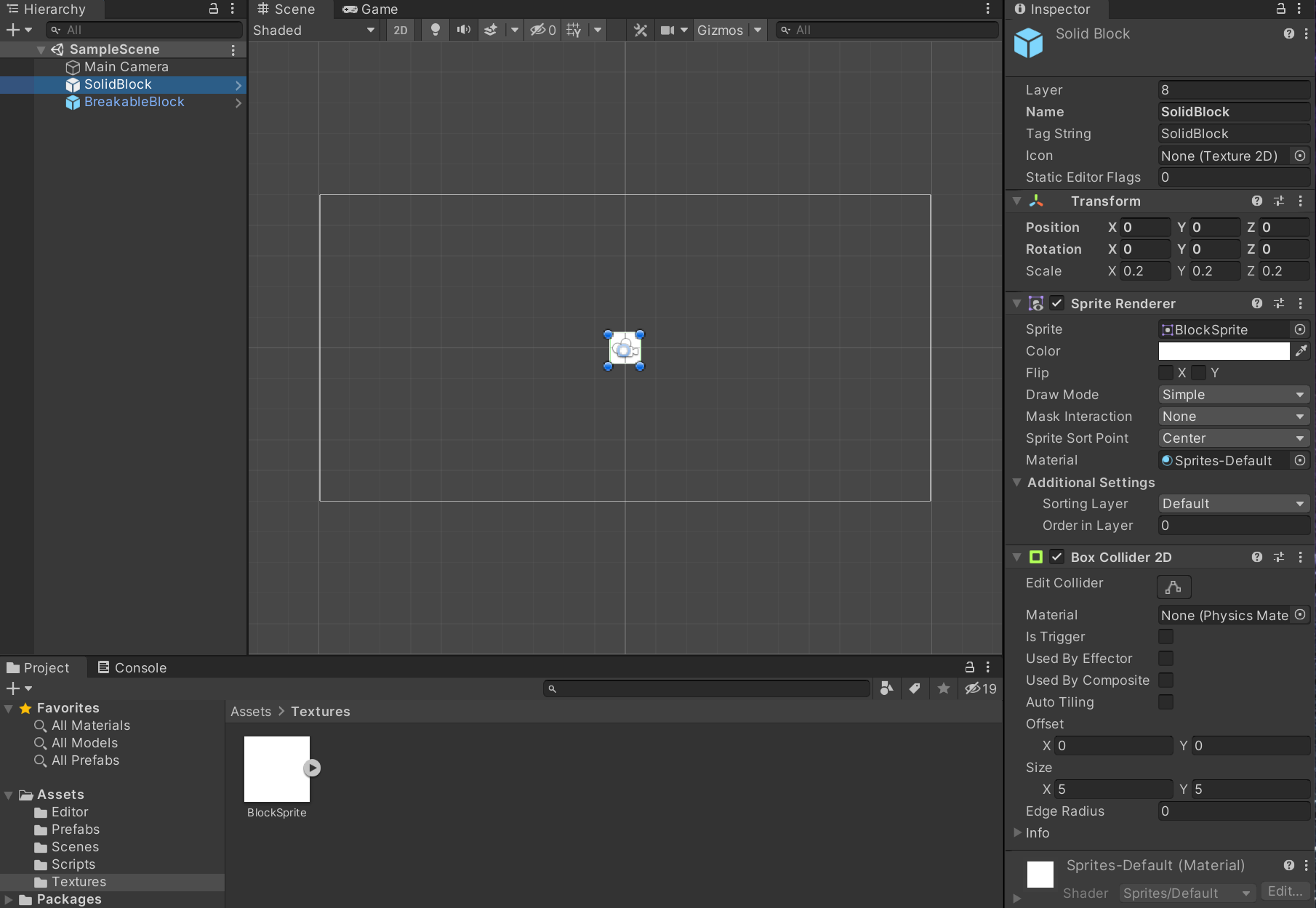


And without BlockSprite selected you see:

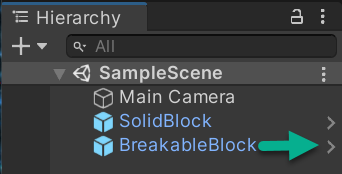


Select the < symbol on the left of SolidBlock in your Hierarchy tab to exit SolidBlock Prefab mode. Save your Project.

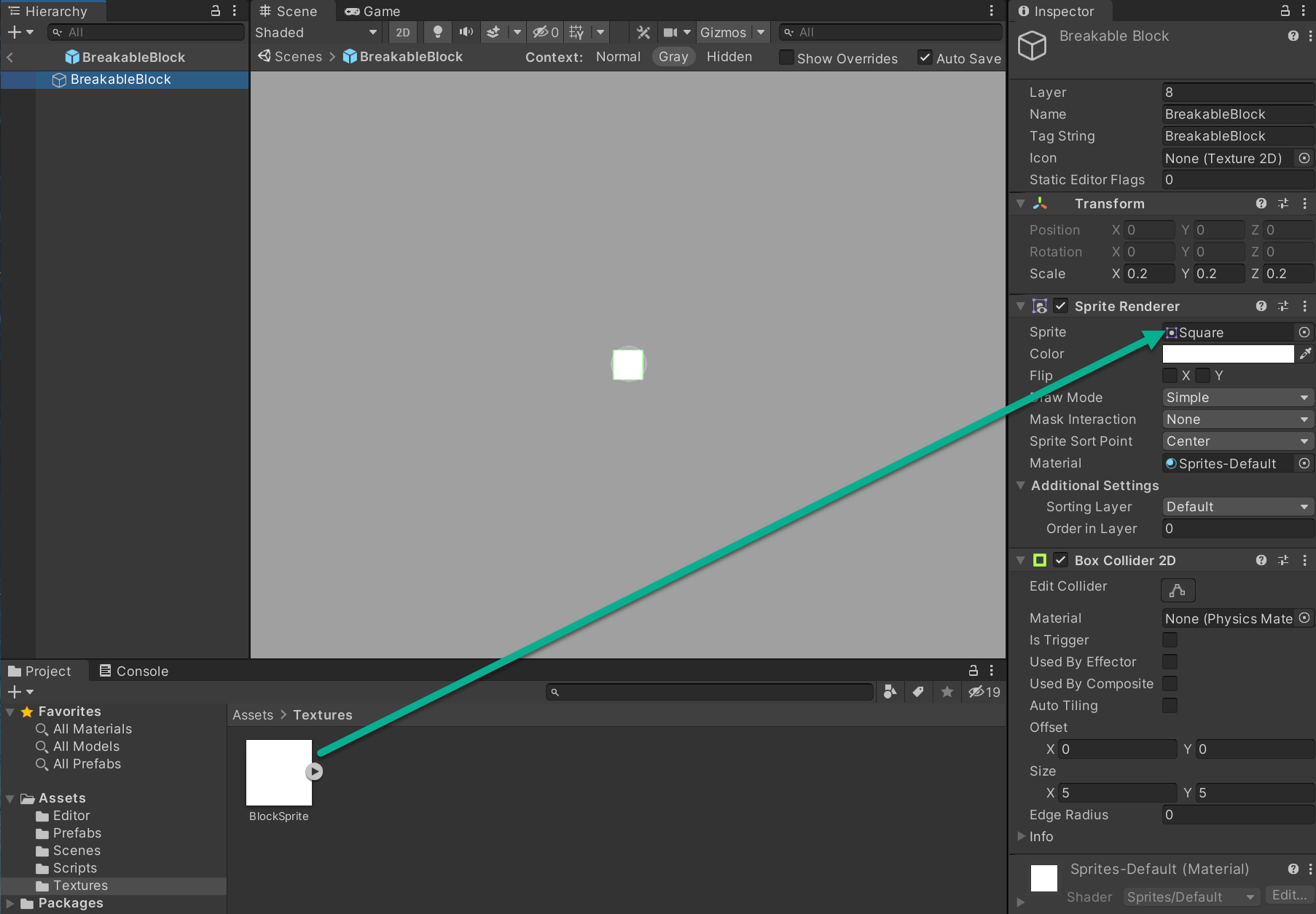




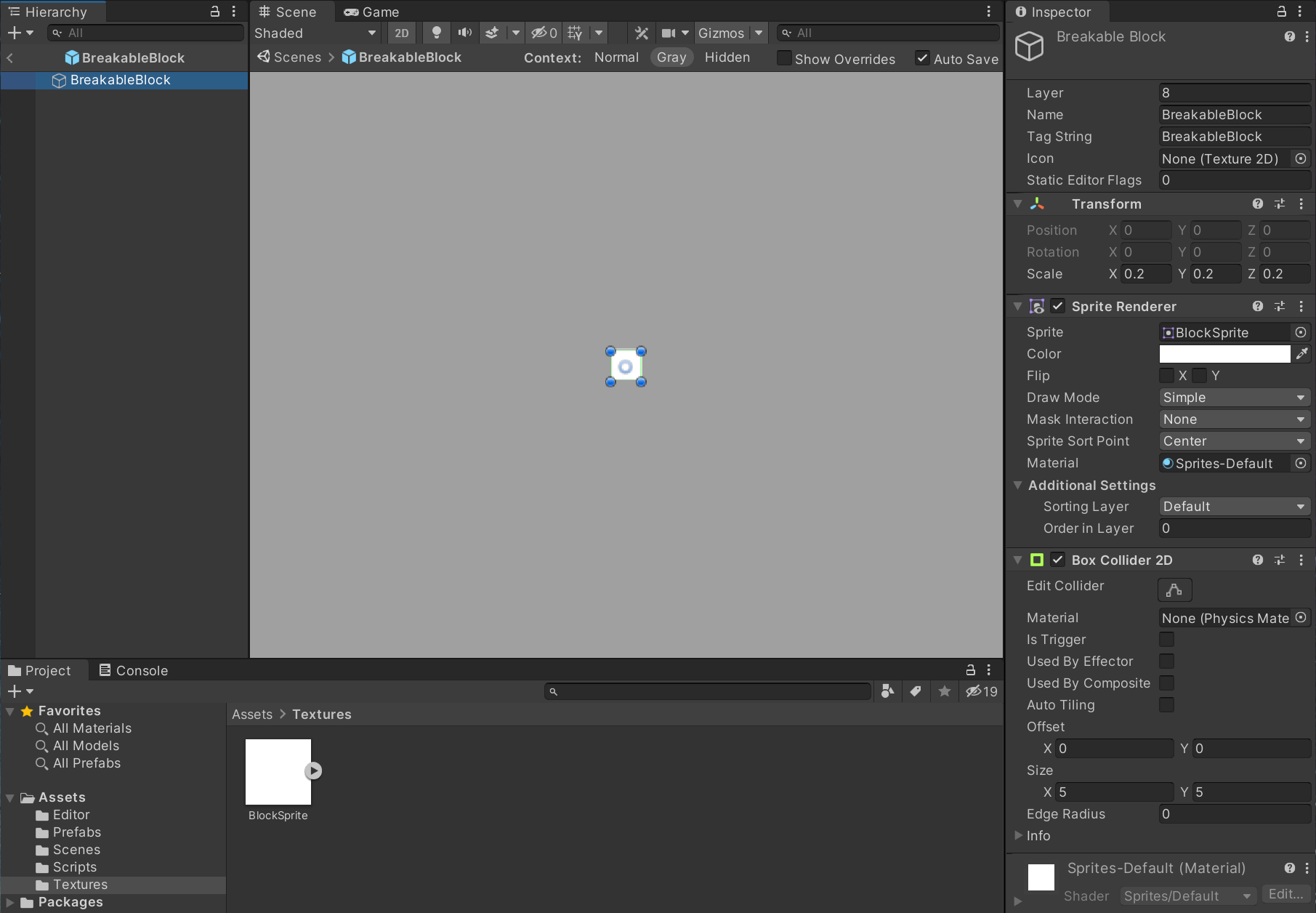
Enter Prefab mode for BreakableBlock with >



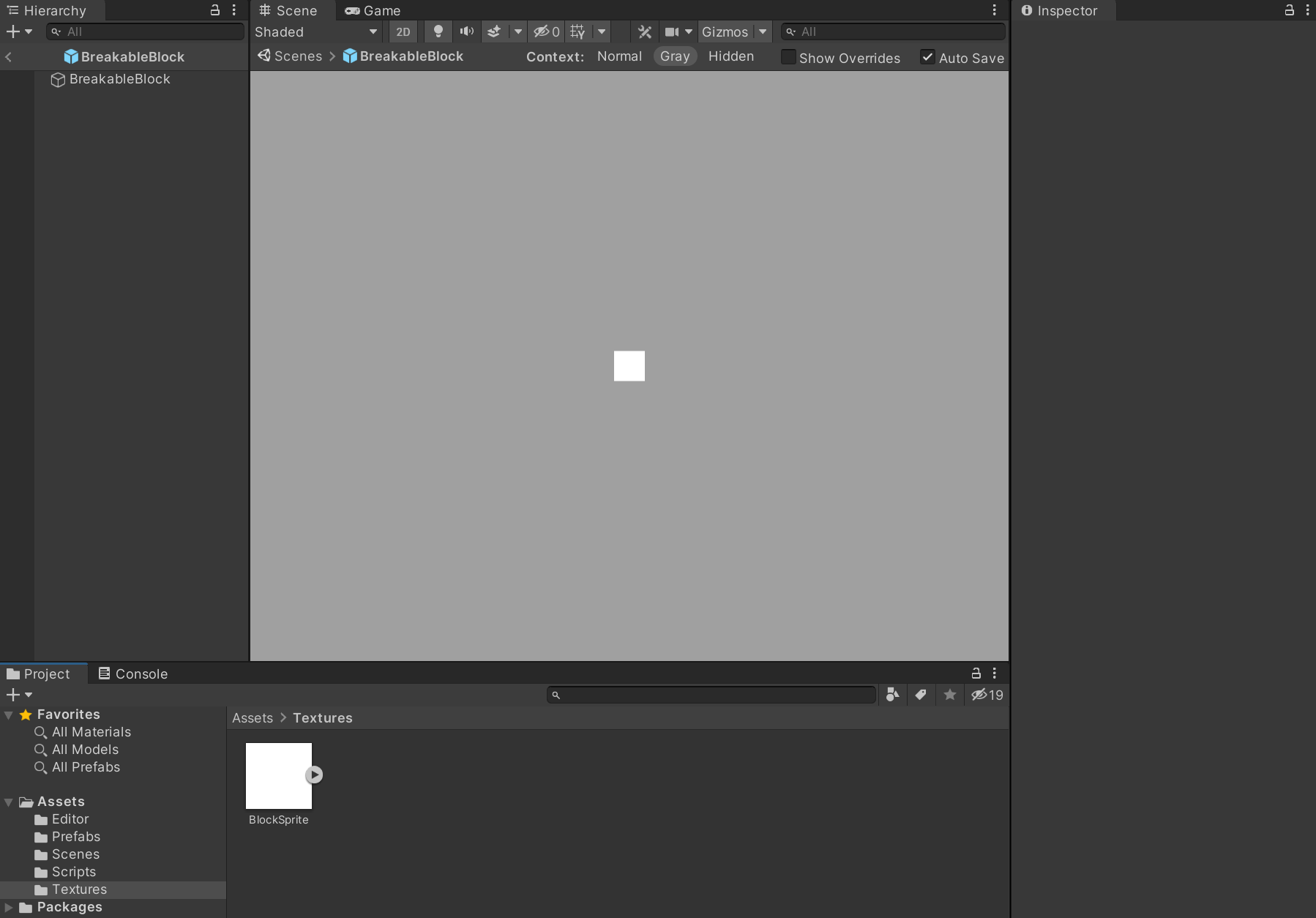
In Hierarchy, select BreakableBlock. In Inspector, click on the down arrow on Sprite Renderer to open and view its settings. In Textures, drag and drop BlockSprite to the Inspector > Sprite Renderer field (directly on the word Square. You will see a +) in Sprite Renderer. Save your Project.



Your Scene changes with BlockSprite selected to:



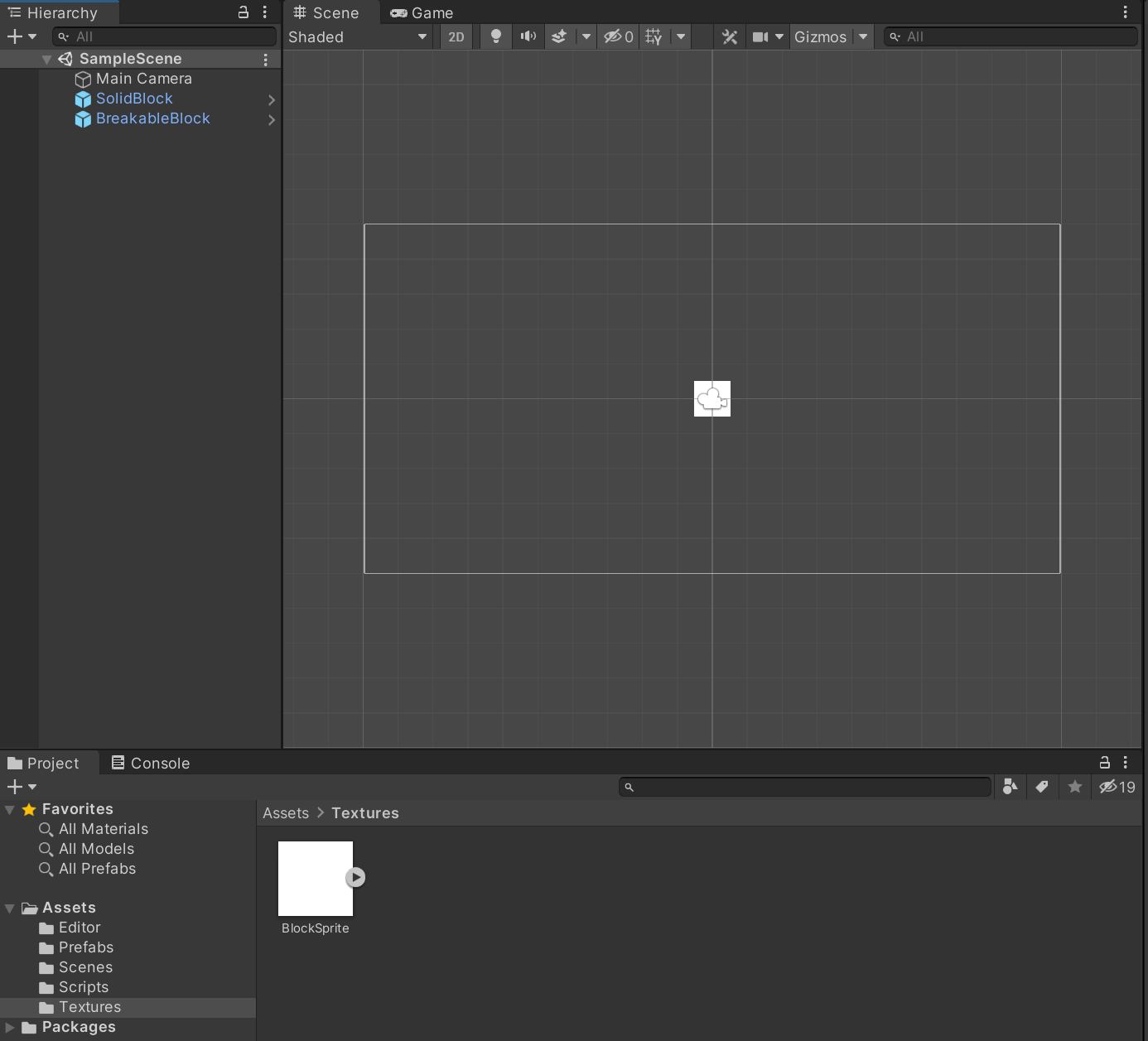
And without BlockSprite selected you see:



Select the < symbol on the left of BreakableBlock in your Hierarchy tab to exit BreakableBlock Prefab mode. Save your Project.

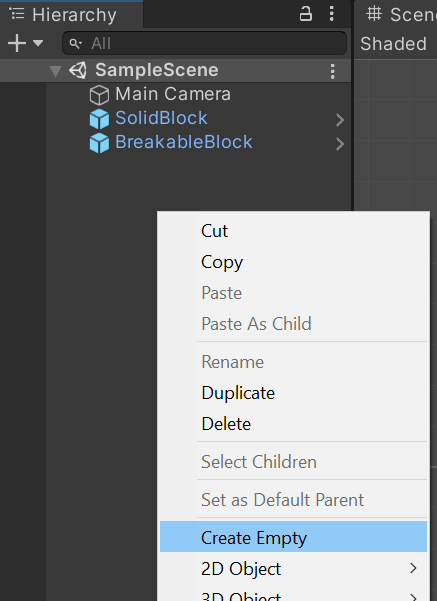


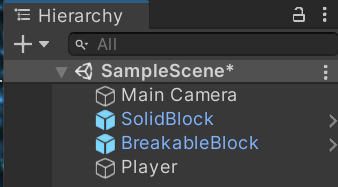
Your Scene, at this point:



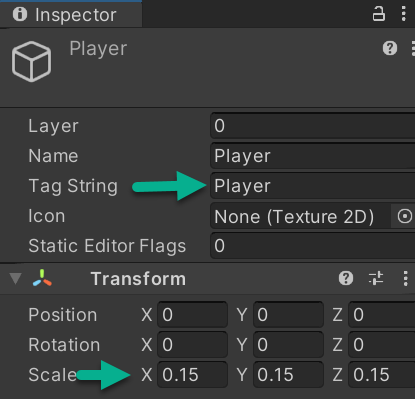
## Player prefab from Create Empty

Right click inside the Hierarchy window and select Create Empty to create an empty GameObject. Name the GameObject Player.

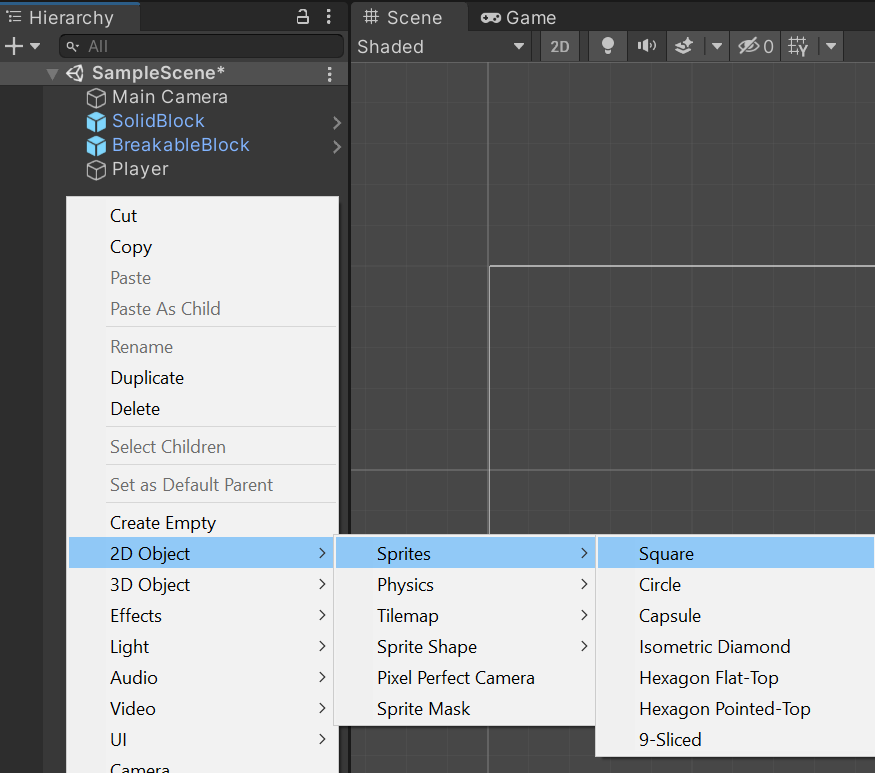


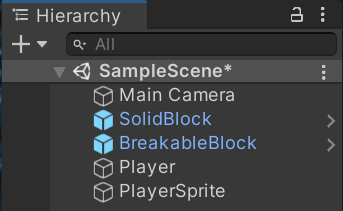


To tag Player: Select Player, go to Inspector > Transform, Scale and change Scale to X = 0.15, Y = 0.15, and Z = 0.15 and change Tag String from Untagged to Player. Save your Project.

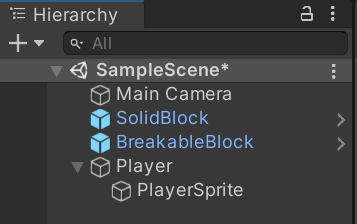


In Hierarchy, create a new 2D Object > Sprite > Square object. Name it PlayerSprite





Add PlayerSprite as a child to the Player object by dragging it from below Player to into Player (no tag needed).

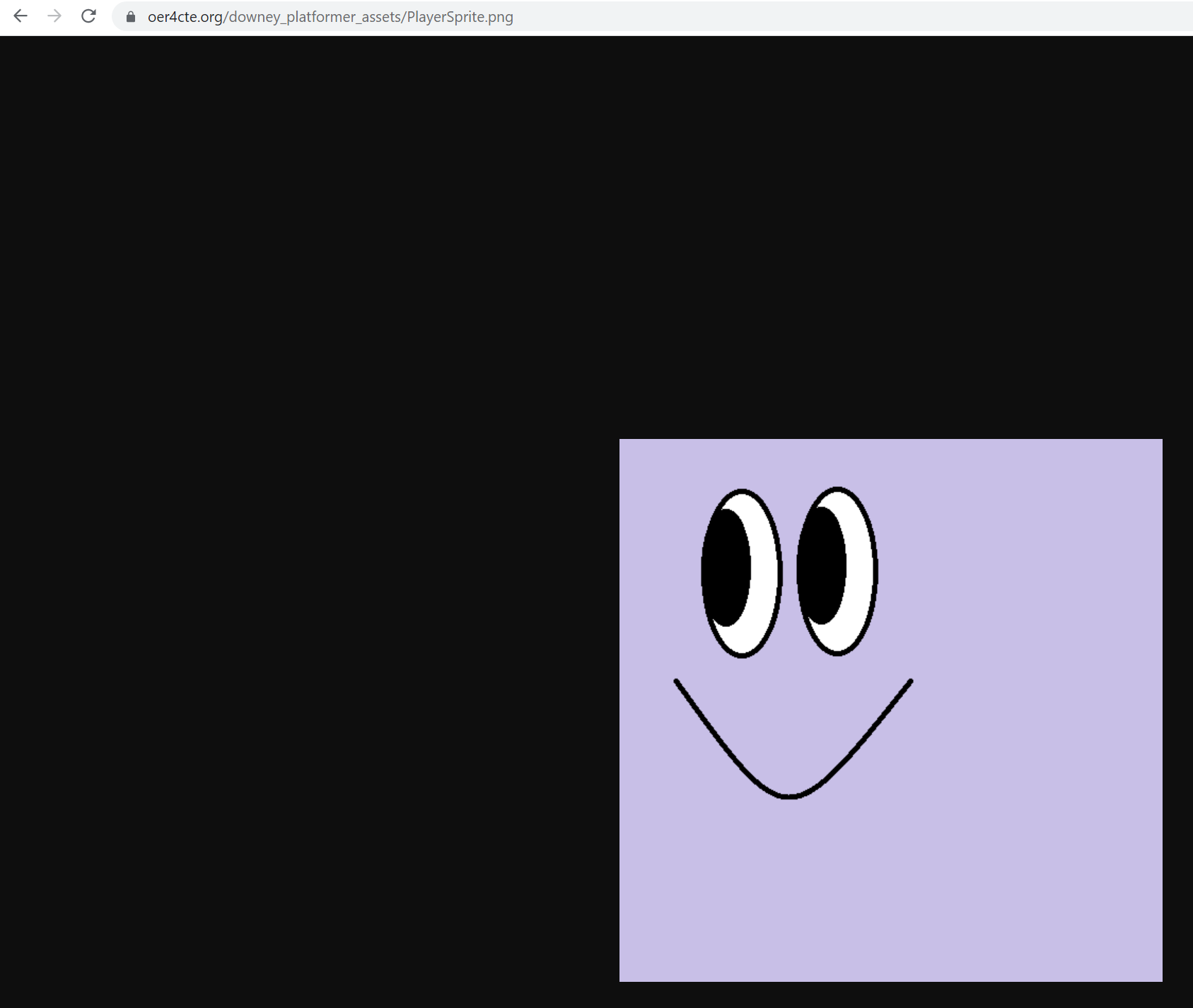


Ideally, keep the player (in this case PlayerSprite) to about 75% the size of the blocks. Change the Scale of PlayerSprite in Transform to X 0.75, Y 0.75, and Z 0.75 then Save your Project.



Open the Textures folder. Please go to:

<https://www.oer4cte.org/downey_platformer_assets/PlayerSprite.png>



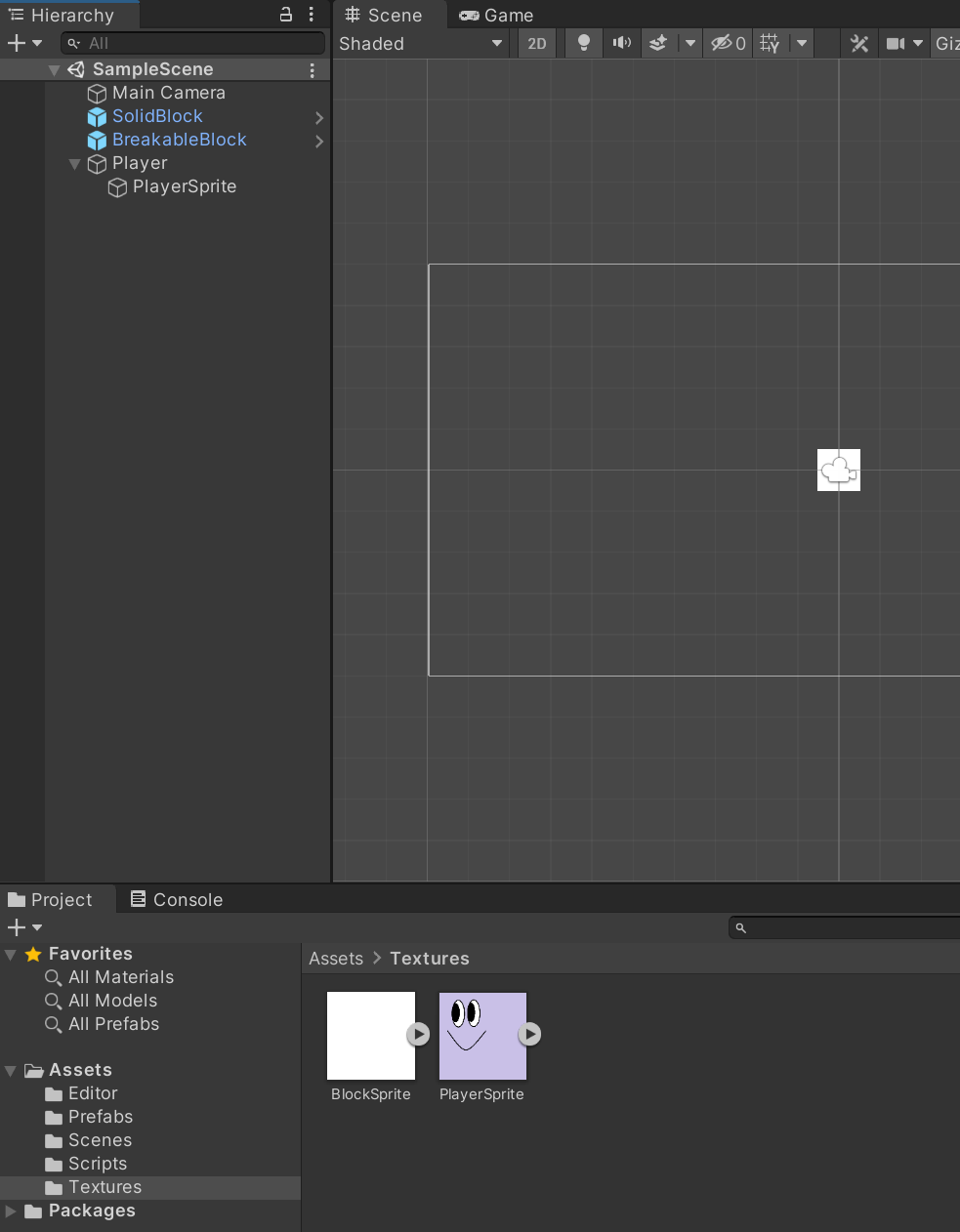
You will download this file, PlayerSprite.png.

Right-click to Save as…

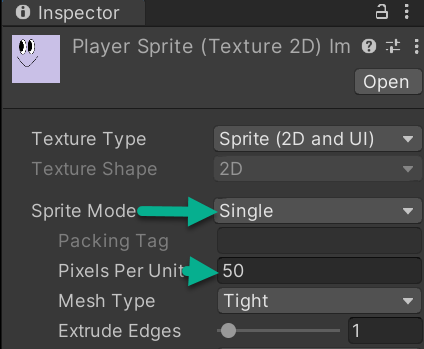
PlayerSprite.png

to your Unity Project in your Downey\_Platformer/Assets/Textures folder.

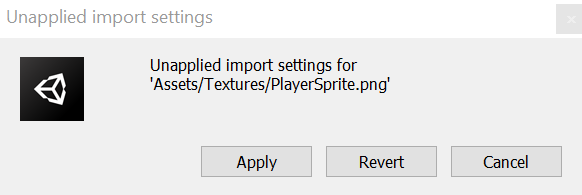
PlayerSprite will automatically populate in Project > Assets > Textures



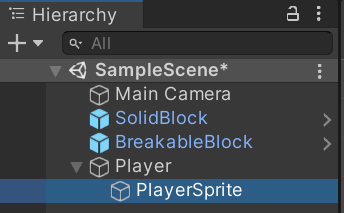
With PlayerSprite selected, view and, if needed, change the following settings in Inspector > Player Sprite. Set Sprite Mode to Single. Set Pixels Per Unit to 50.



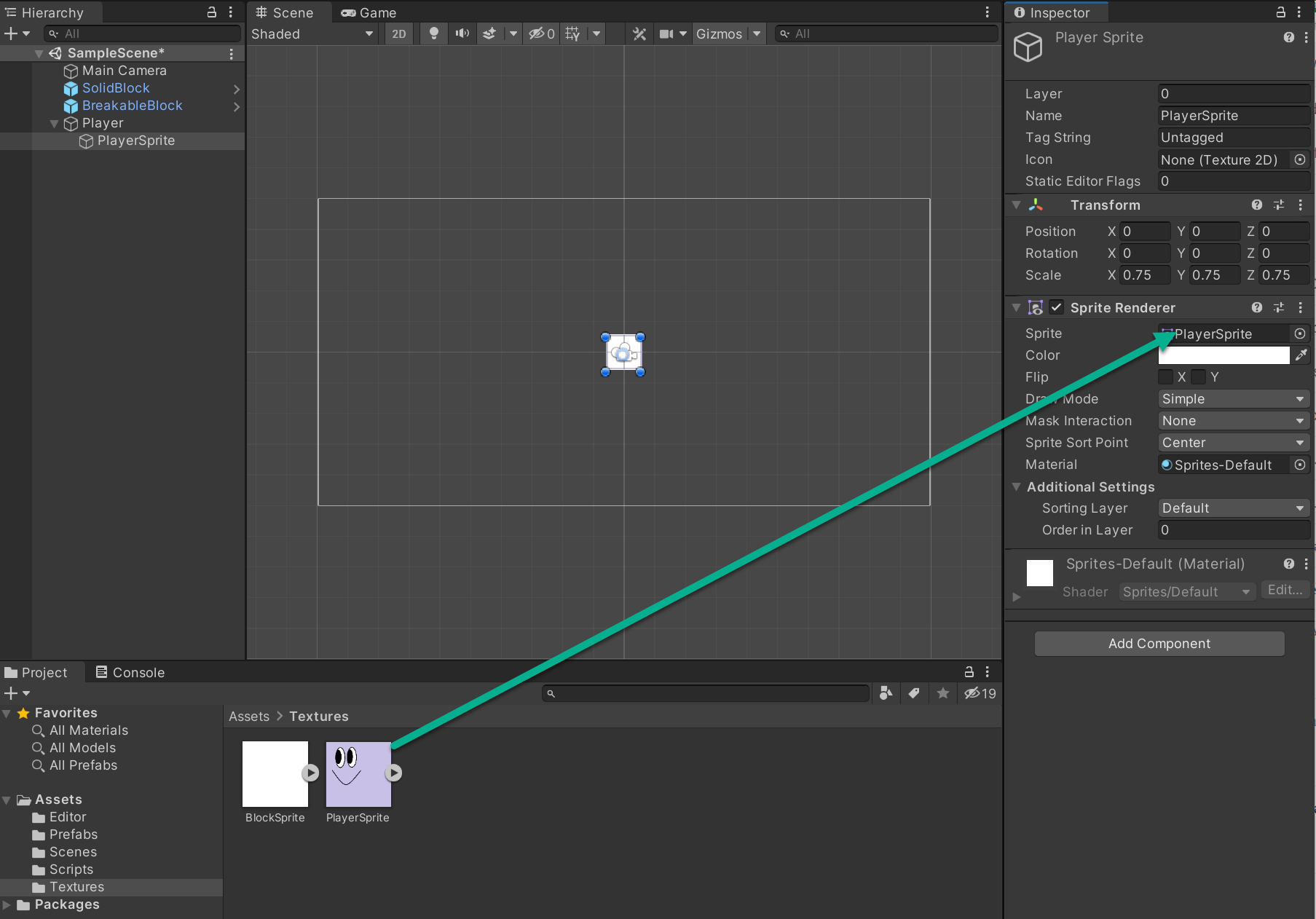
If asked, select Apply:



Select PlayerSprite in Hierarchy.



With PlayerSprite selected in Hierarchy; drag, and drop PlayerSprite from Assets > Textures to the Sprite Renderer component of the PlayerSprite object. Save your Project.

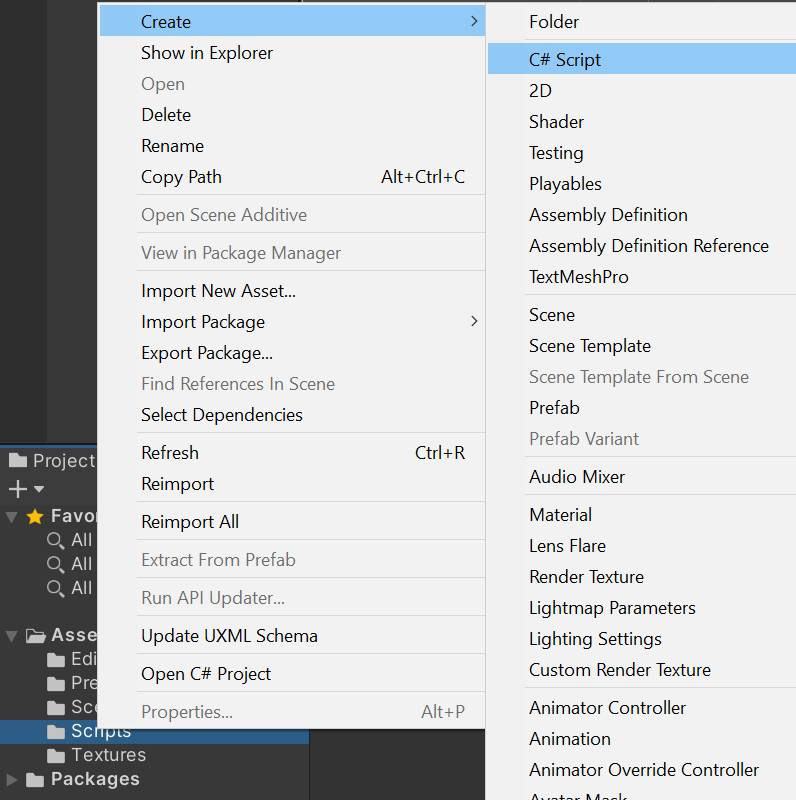


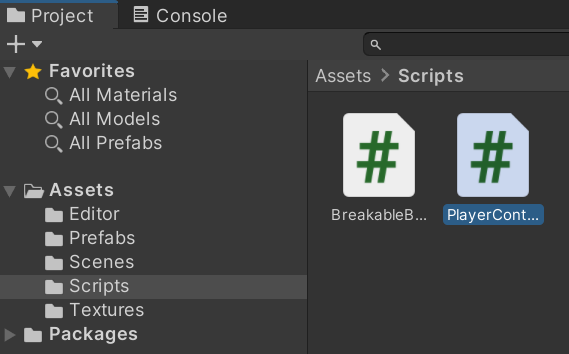
## Player Script

### Term Definitions:

* **Declarations** are keywords that define what the accessibility level of a variable or method is for a class.
* **Public** level means that other scripts and classes can see and access that data.
* **Private** means that only the class it is a part of can access it.

In Projects, select Assets > Scripts. With Scripts selected, right click to Create > C# Script. Add a new C# script and name it PlayerController





Double click on PlayerController to open it inside Visual Studio.

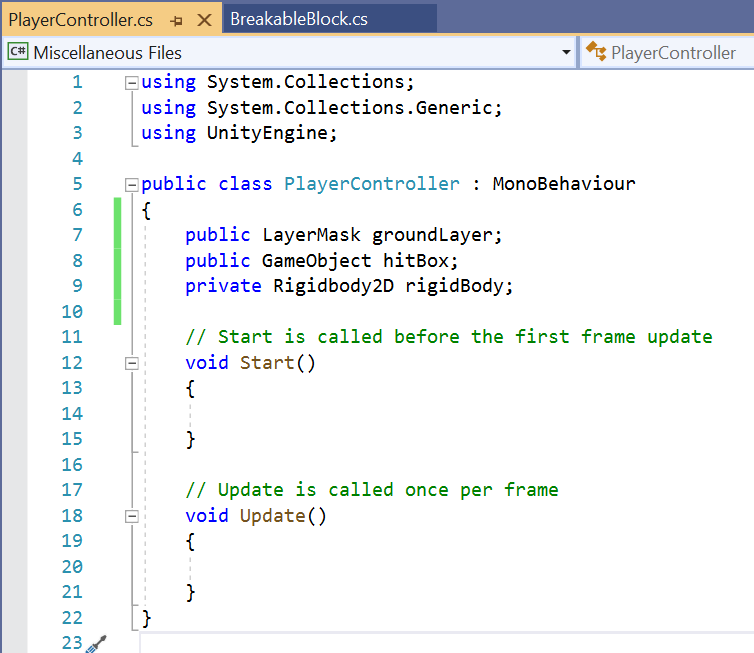
Note\* Be hyper-aware of where you place code over the next several pages. See the accompanying images and check the bottom of each, the remaining closing brackets should match your screen.

Above the Start()method, add the following variables, and save:

public LayerMask groundLayer;

public GameObject hitBox;

private Rigidbody2D rigidBody;



groundLayer holds the layer value for ground which will be used when you check whether you are on the ground or not. hitBox holds the reference to your player weapon so that you can destroy breakable blocks. Both groundLayer and hitBox are public, you can assign their values in Unity’s Inspector window. rigidBody is private because only this script needs to see and access it.

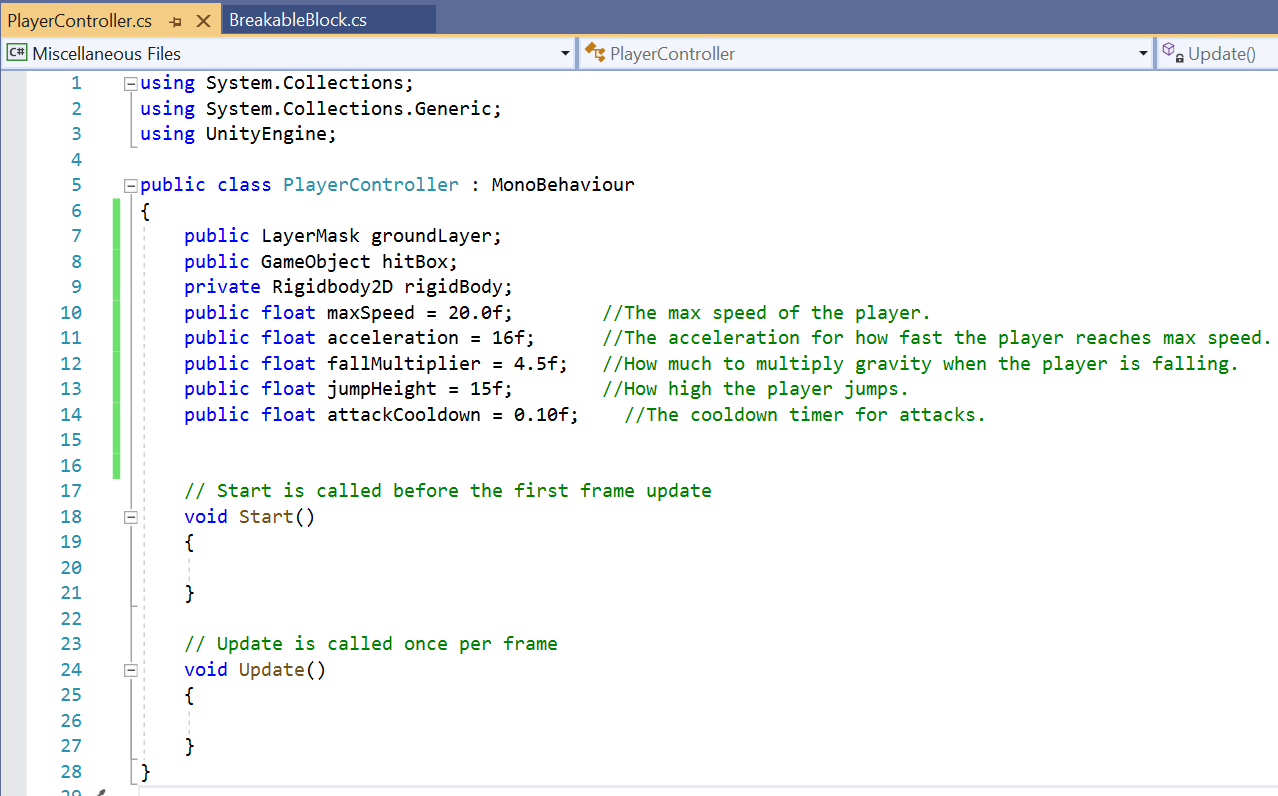
Add the following public variables. These are the stats of the player that can be adjusted. Note: In Visual Studio these will be single line comments, they appear as multi-line in this handout due to the .pdf layout. Save your file.

public float maxSpeed = 20.0f; //The max speed of the player.

public float acceleration = 16f; //The acceleration for how fast the player reaches max speed.

public float fallMultiplier = 4.5f; //How much to multiply gravity when the player is falling.

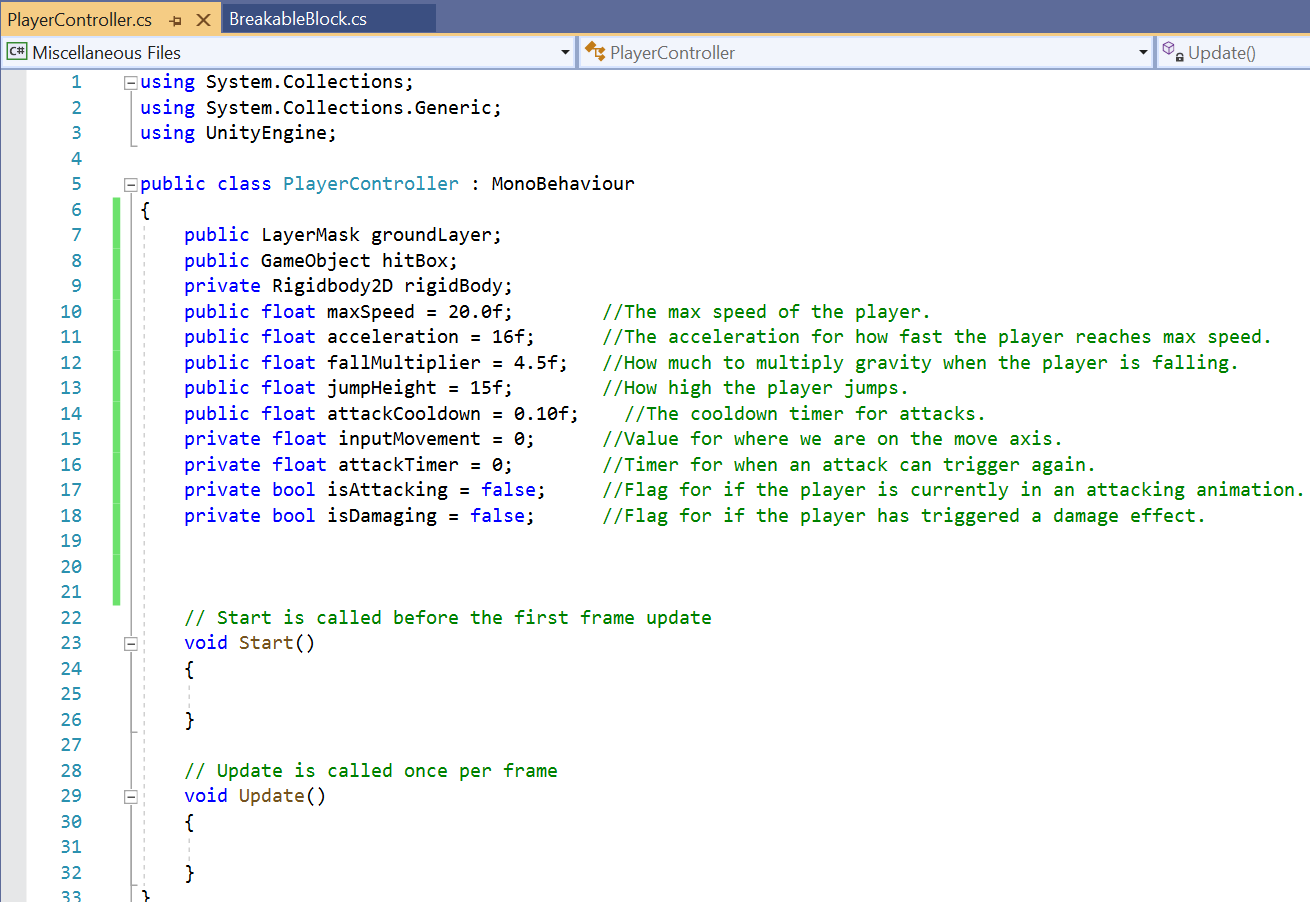
public float jumpHeight = 15f; //How high the player jumps.  
public float attackCooldown = 0.10f; //The cooldown timer for attacks.



Add the following private variables. These are variables that you need to keep track of data and states inside the player controller. Save your file.

private float inputMovement = 0; //Value for where we are on the move axis.  
private float attackTimer = 0; //Timer for when an attack can trigger again.

private bool isAttacking = false; //Flag for if the player is currently in an attacking animation.  
private bool isDamaging = false; //Flag for if the player has triggered a damage effect.



After the comment line // Start is called before the first frame update, replace the code starting with void and through the second-to-last closing brace, with the code below. Be careful to not copy over the file final closing brace. Use multiline comment syntax.

private void Start()

{

/\*Keep the rigidbody private, which means you need access to the Rigidbody component attached to the gameObject that this script is attached to. GetComponent takes in the type and gives you the component attached to the player object.\*/

rigidBody = GetComponent<Rigidbody2D>();

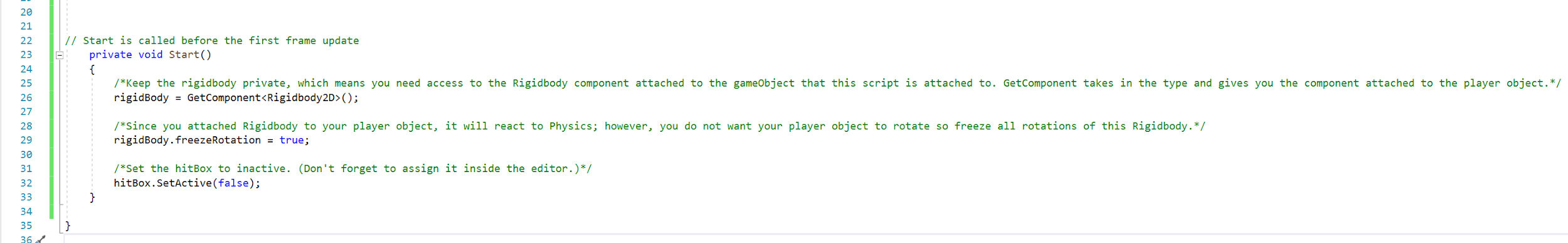
/\*Since you attached Rigidbody to your player object, it will react to Physics; however, you do not want your player object to rotate so freeze all rotations of this Rigidbody.\*/

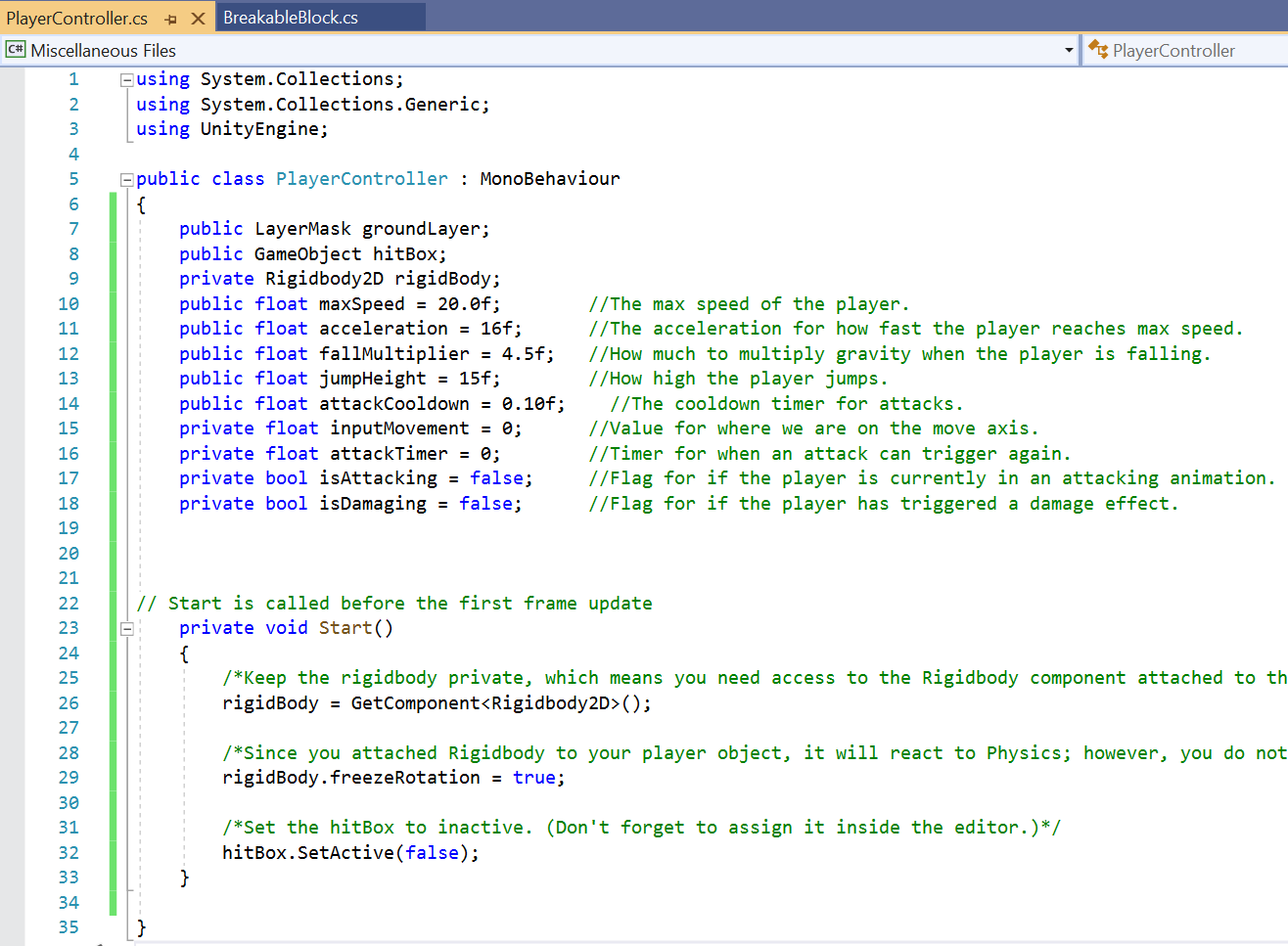
rigidBody.freezeRotation = true;

/\*Set the hitBox to inactive. (Don't forget to assign it inside the editor.)\*/ hitBox.SetActive(false);

}

Your code will appear across the screen. The first image below shows the general shape of what you will see. The second image zooms in to show placement of code.





Add a new method FixedUpdate() as shown below. FixedUpdate() is called on a fixed interval with the Physics system, instead of Update() which is called every frame. It's more stable to implement Physics-related operations inside this function. If you use regular Update() which is called every frame, your Physics calculation will be dependent on the frame rate of the device. If you have one device running the game at 200 FPS and another device running at 20 FPS, you will notice inconsistent Physics behavior because Physics is applied differently on both devices, which is the behavior you do not want. Use FixedUpdate() because it is independent of the number of FPS the game is running.

private void FixedUpdate()

{

/\* Moves the Player object along the horizontal axis. If the player is below the max speed, you can keep adding force to the Rigidbody.\*/

if (rigidBody.velocity.magnitude < maxSpeed)

{

Vector2 currentSpeed = new Vector2(inputMovement, 0);

rigidBody.AddForce(acceleration \* currentSpeed);

}

/\*This code dampens the momentum when the arrow keys are not pressed. You only dampen when not in the air. Velocity is a vector that stores the direction in which the Rigidbody moves along with magnitude/speed. Normalizing the Velocity keeps the direction of the velocity the same and changes the magnitude/speed to 1.\*/

if(inputMovement == 0 && rigidBody.velocity.y == 0)

{

rigidBody.velocity.Normalize();

}

/\*This code adds gravity when the player is falling with the fallMultiplier variable. This makes the jumping feel less floaty and gives you more control.\*/

if (rigidBody.velocity.y < 0)

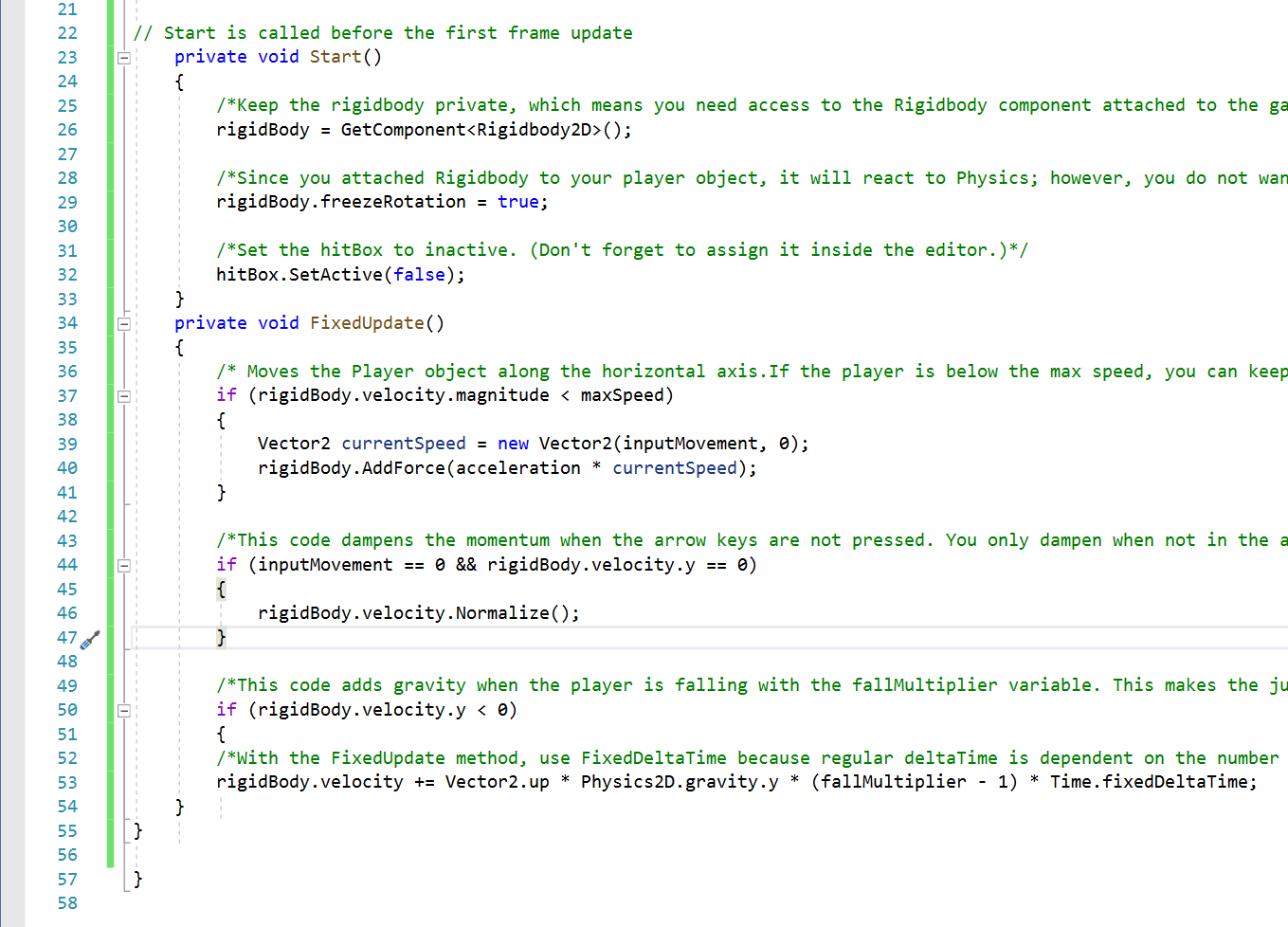
{

/\*With the FixedUpdate method, use FixedDeltaTime because regular deltaTime is dependent on the number of FPS. deltaTime intervals in seconds from the last frame to the current one.\*/

rigidBody.velocity += Vector2.up \* Physics2D.gravity.y \* (fallMultiplier - 1) \* Time.fixedDeltaTime;

}

}



Update() is used when you want to perform an action continuously. All of the inputs you receive from users are implemented inside Update() because it's called every frame so you can't miss input from users. If you use FixedUpdate() for receiving input, you might miss the input. FixedUpdate() is called after a fixed interval. If users press in that interval, you will not receive input; that's why you use Update(). Add the following code to your Update() method. You’ll get an error message when you add IsGrounded(), you’ll add that method next.

private void Update()

{

/\*Input.GetButtonDown returns a bool variable, which has a 'true' value in the frame where the user pressed the button. You do your jump physics here since you're applying a burst force, not a continuous force. You only jump when IsGrounded() is true. TheJump button by default is the Spacebar.\*/

if (Input.GetButtonDown("Jump") && IsGrounded())

{

Vector2 jumpForce = new Vector2(0, jumpHeight);

rigidBody.AddForce(jumpForce, ForceMode2D.Impulse);

}

/\* You get the Horizontal Axis from Unity's Input system, which is returned as a float. When you press the left arrow key or the "A" key, the value is -1. When you press the right arrow key or the "D" key, the value is 1. When no keys are pressed, the value is 0.\*/

inputMovement = Input.GetAxis("Horizontal");

/\*This code flips the Player object so that it always faces the direction you are moving.\*/

if (inputMovement > 0)

{

transform.rotation = Quaternion.Euler(0, 180f, 0);

}

else if (inputMovement < 0)

{

transform.rotation = Quaternion.Euler(Vector3.zero);

}

/\*When you attack, first check if the user has pressed the left mouse button / left ctrl. you also need to check if the player is already attacking or not, then check if attackTimer is <= 0. The Fire1 button by default is Left Mouse or Left Ctrl.\*/

if (Input.GetButtonDown("Fire1") && !isAttacking && attackTimer <= 0)

{

/\*Set the flag to true, set the timer to the cooldown, and activate the hitBox.\*/

isAttacking = true;

attackTimer = attackCooldown;

hitBox.SetActive(true);

}

/\*If the isAttacking flag is true, you go through your attacking logic.\*/

if (isAttacking)

{

/\*While the timer is not at zero, remove time from it. This is to stop players from attacking multiple times at once. You want there to be a delay in between attacks.\*/

if (attackTimer > 0)

{

attackTimer -= Time.deltaTime;

}

//When the timer reaches 0, turn off hitBox and set the flag to false.

else

{

hitBox.SetActive(false);

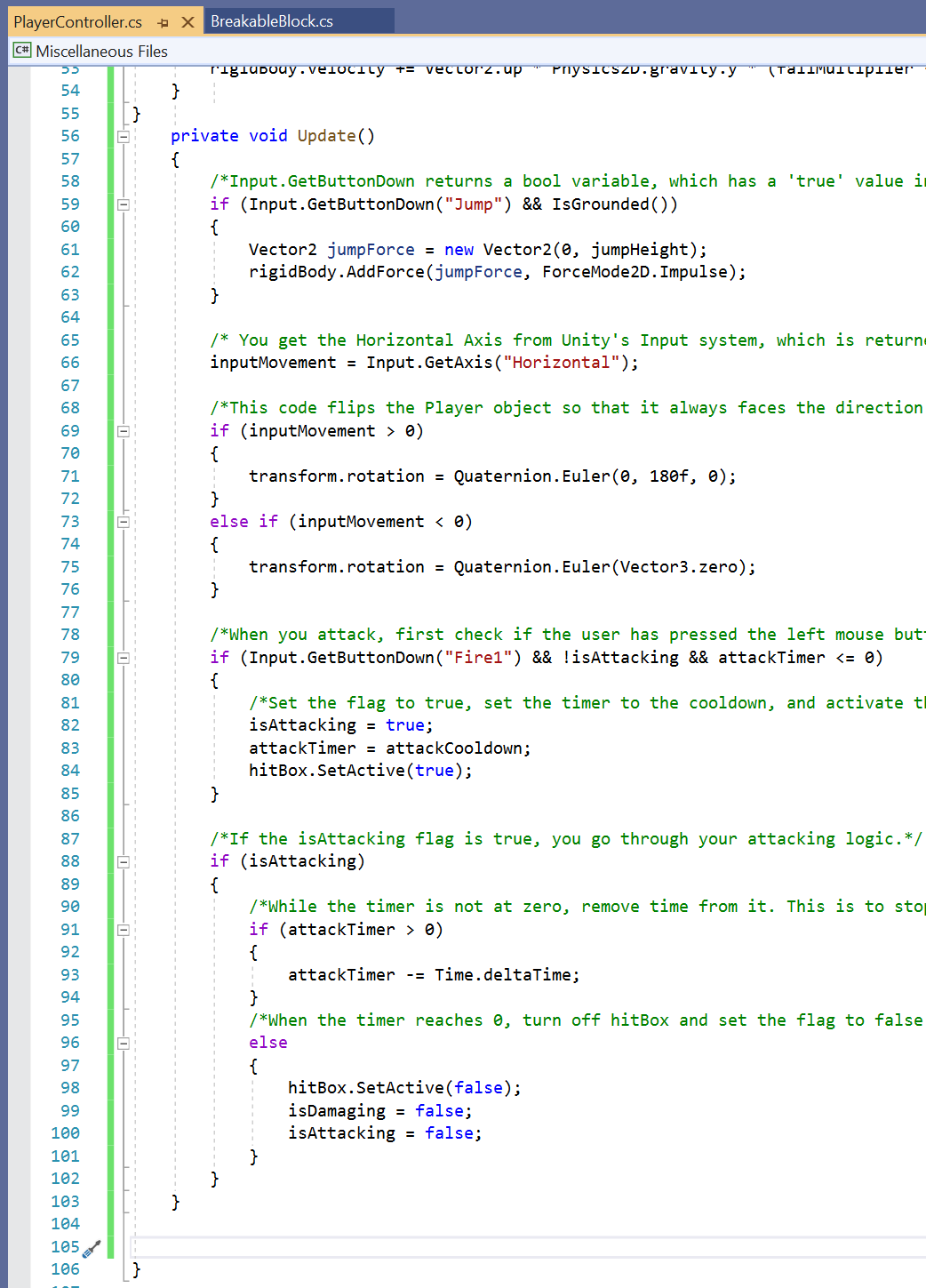
isDamaging = false;

isAttacking = false;

}

}

}



Add a new method IsGrounded(). This method checks to determine if the player is grounded or not:

private bool IsGrounded()

{

/\*Use a CircleCast2D to detect if the player is grounded or not. Using a CircleCast allows you to have a wider range than just a single ray, so it's easier to jump from a ledge. You cast it from an offset of your current position, and it only detects colliders with the groundLayer layerMask.\*/

Vector2 position = new Vector2(transform.position.x, transform.position.y - 0.1f);

RaycastHit2D hit = Physics2D.CircleCast(position, 0.25f, Vector2.down, 0.1f, groundLayer.value);

/\*If the raycast hits an object assigned to the Ground layer, it returns true. Otherwise it is null and returns false.\*/

if (hit.collider != null)

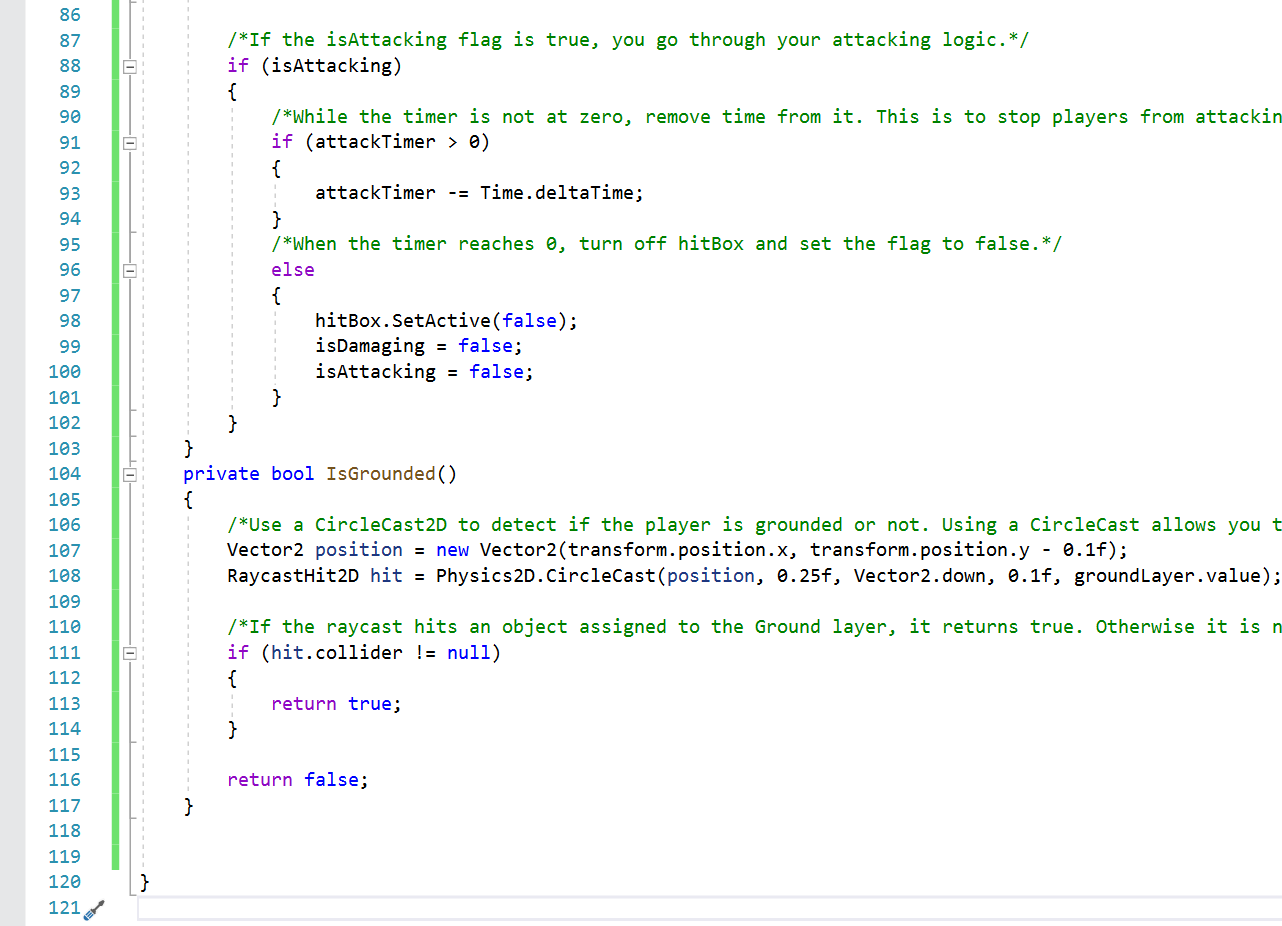
{

return true;

}

return false;

}



Add a new method OnTriggerEnter2D(). If you start typing this into Visual Studio it will autofill the frame of the method, but you’ll need to add the contents. You need this method to detect whether the player has triggered with the Breakable Blocks or not. This is a Physics controlled function which is called when the collider attached to this gameObject gets triggered with another collider.

private void OnTriggerEnter2D(Collider2D collision)

{

/\*Checks if isDamaging is true, which means the player is already doing damage. isDamaging is set to false when attackTimer is less than zero.

if (isDamaging)

{

return;

}

/\*Checks if the collider is active or not. By default, even if the GameObject is inactive, Unity will still call OnTriggerEnter2D, so you need to check for it.\*/

if (hitBox.GetComponent<Collider2D>().isActiveAndEnabled)

{

/\*Checks what tag is on the object that entered the trigger in order to call the methods needed.\*/

if (collision.CompareTag("BreakableBlock"))

{

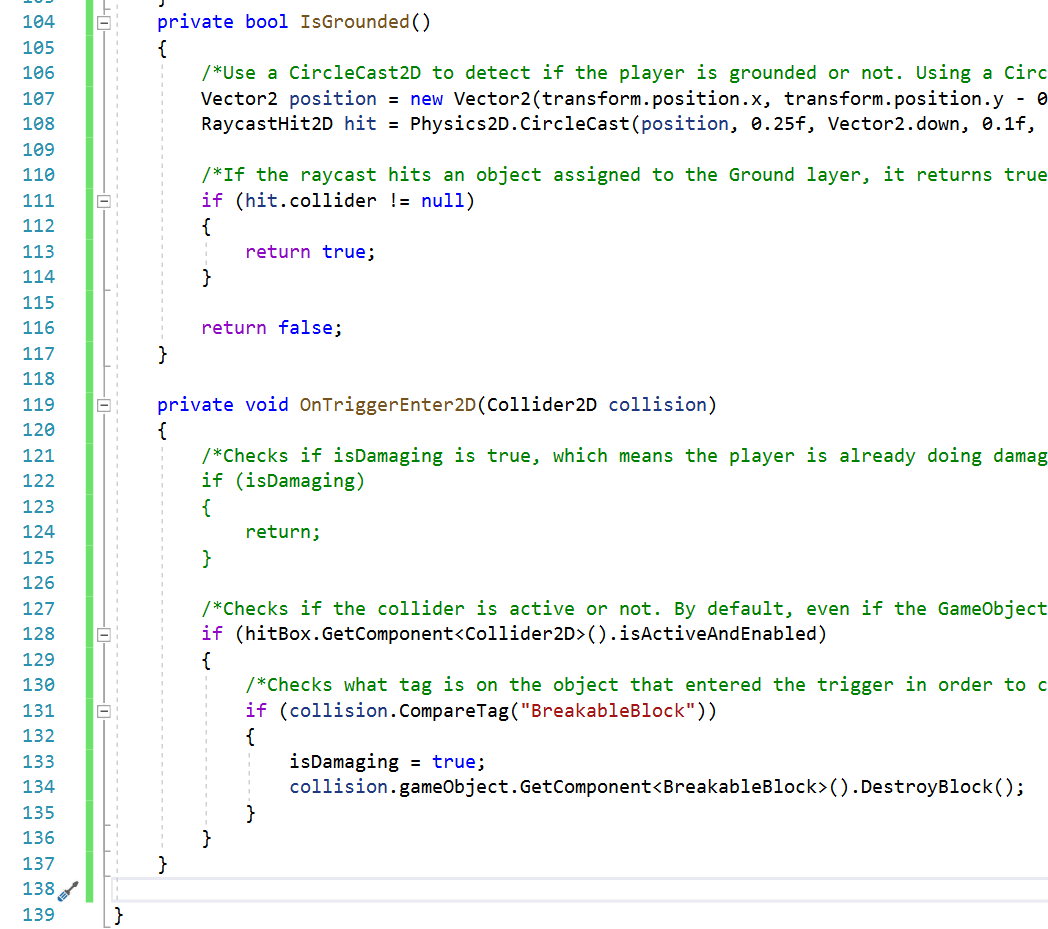
isDamaging = true;

collision.gameObject.GetComponent<BreakableBlock>().DestroyBlock();

}

}

}



Save and exit Visual Studio. Click in Unity for a quick code update.

This ends Milestone 1 Deliverable 2 of Downey Platformer. Proceed to: milestone1\_deliverable3.pdf for the next steps in developing Milestone 1 Deliverable 3 where you begin assembling the player.  
Please link to:

<https://www.oer4cte.org/downey_platformer_tutorials/milestone1_deliverable3.pdf>

and right click to

Save as…

and download the tutorial to your computer.