LEARN INTERMEDIATE OOP BY BUILDING A PLATFORMER GAME

Introduction:

Coding a game is a great way to grasp fundamental programming principles, while also creating an interactive gaming experience.

In this platformer game project, you'll continue to learn about classes, objects, inheritance, and encapsulation. You'll also learn how to design and organize game elements efficiently and gain insights into problem-solving and code reusability.

Step 1:

In this project, you are going to learn intermediate Object Oriented Programming principles by building a platformer game. All of the HTML and CSS have been provided for you.

Start by using `document.getElementById()` to get the `#start-btn` and `#canvas` elements.

Store them in `const` variables named `startBtn` and `canvas` respectively.

Step 2:

Next, you will need to use `document.querySelector` to get the `.start-screen` and `.checkpoint-screen` elements.

Store them in `const` variables called `startScreen` and `checkpointScreen` respectively.

Step 3:

The next step is to target the paragraph element inside the `.checkpoint-screen` element.

Use `document.querySelector` and the child combinator `>` to target the paragraph element.

Assign that value to a `const` variable called `checkpointMessage`.

Step 4:

Before you can begin building out the functionality for the game, you will need to set up the ability to add 2D graphics.

The Canvas API can be used to create graphics in games using JavaScript and the HTML `canvas` element.

You will need to use the `getContext` method which will provide the context for where the graphics will be rendered.

Example Code:

```
canvas.getContext("2d");
```

Assign that `getContext` method to a `const` variable called `ctx`.

Step 5:

The `canvas` element has a `width` property which is a positive number that represents the width of the canvas.

Example Code:

```
canvas.width
```

Below your `const` declarations, append the `width` property to the `canvas` variable.

Step 6:

The `innerWidth` property is a number that represents the interior width of the browser window.

Assign `innerWidth` to `canvas.width`.

Step 7:

The `innerHeight` property is a number that represents the interior height of the browser window.

Below your `canvas.width`, append the `height` property to the `canvas` variable and assign it `innerHeight`.

Step 8:

In your platformer game, the main player will need to jump between the different platforms. When the player jumps, you will need to apply gravity to bring them back down.

Create a new `const` variable called `gravity` and assign it the number `0.5`.

Step 9:

In the game, the player will have the opportunity to cross different checkpoints. You will need to keep track of the status for the checkpoint collision detection.

Use `let` to create a new variable called `isCheckpointCollisionDetectionActive` and assign it the value of `true`.

Step 10:

As you are designing the game, you will need to make sure that the size of the elements in the game are responsive and adapt to different screen sizes.

Start by creating an arrow function called `proportionalSize` that takes in a `size` parameter.

Step 11:

The `width` and the `height` of the main player, platforms and checkpoints will be proportional sized relative to the `innerHeight` of the browser screen. The goal is to make the game responsive and visually consistent across different screen sizes.

Inside your `proportionalSize` function, you will need to return a ternary that checks if `innerHeight` is less than `500`. If so, return `Math.ceil((size / 500) * innerHeight)`, otherwise return `size`.

Step 12:

The next step is to define some characteristics for the main player of the game.

Start by creating a new `class` called `Player`.

Step 13:

Inside your Player class, you will need to define the player's position, velocity, width, and height values. All of these values will be defined inside the constructor method.

Create an empty constructor inside your Player class.

Step 14:

Inside your constructor, use the this keyword to set the position property to an empty object.

Step 15:

Inside your position object, add a new key called x with a value of proportionalSize(10). After that, add another key called y with a value of proportionalSize(400).

You need to use the proportionalSize function here to make sure that the player's position is always proportional to the screen size. This is important because you want the player to be able to move around the screen regardless of the screen size.

Step 16:

Below your position object, use the this keyword to set the velocity property to an object.

Inside that new velocity object, create a key called x with a value of 0 and a new key called y with a value of 0.

The velocity property will be used to store the player's speed in the x and y directions.

Step 17:

Below your `velocity` object, use the `this` keyword to set the `width` property to `proportionalSize(40)`.

Below your `width` property, use the `this` keyword to set the `height` property to `proportionalSize(40)`.

You are using the `proportionalSize()` function here to set the `width` and `height` properties of your class to be proportional to the height of the screen.

Step 18:

The next step is to create a `draw()` method, which will be responsible for creating the player's `width`, `height`, `position`, and fill color.

Below your constructor, create an empty `draw()` method.

Step 19:

Now, you need to set the color for your player.

Inside the `draw()` method, assign the string `"#99c9ff"` to `ctx.fillStyle`.

Step 20:

Below your `ctx.fillStyle`, you need to create the player's shape by calling the `fillRect()` method on the `ctx` object which you instantiated earlier.

Example Code:

```
fillRect(x, y, width, height)
```

Inside the fillRect() method add the this.position.x, this.position.y, this.width and this.height values.

Step 21:

The next step is to create an update() method which will be responsible for updating the player's position and velocity as it moves throughout the game.

Below your draw() method, create an empty update() method.

Step 22:

Inside the update() method, call the draw() method to ensure that the player is continually drawn on the screen as the game updates.

Don't forget to include the this keyword.

Step 23:

When the player moves to the right, you will need to adjust its velocity.

Use the addition assignment operator to add the velocity's x coordinate to the player's x position.

Don't forget to include the this keyword for the velocity and position.

Step 24:

When the player jumps up, you will need to add the logic for adjusting its velocity.

Use the addition assignment operator to add the velocity's `y` coordinate to the player's `y` position.

Don't forget to include the `this` keyword for the velocity and position.

Step 25:

Right now, when the player jumps up, it is possible for it to move past the height of the canvas.

To fix that, you will need to add a condition to stop the player from falling past the height of the canvas.

Create an empty `if` statement that checks if the sum of the player's `y` position, height, and `y` velocity is less than or equal to the height of the canvas.

Step 26:

In the `if` statement, add another `if` statement to check if the player's `y` position is less than `0`.

Step 27:

Inside the inner `if` statement, assign 0 to the player's `y` position.

Step 28:

Below the `this.position.y = 0`, assign `gravity` to the velocity's `y` position.

Step 29:

Below your inner `if` statement, use the addition assignment operator to add `gravity` to the `y` velocity.

Step 30:

Add an `else` clause that assigns `0` to `this.velocity.y`.

Step 31:

The final condition you need to add inside the `Player` class is to ensure that the player stays within the boundaries of the canvas screen and doesn't move too far off to the left.

Create an `if` statement, to check if the player's `x` position is less than the width.

Step 32:

Inside the `if` statement, assign the width to the player's `x` position.

Step 33:

For the last condition, you will need to check if the player's `x` position has exceeded the right edge of the canvas. If it has, you will need to set the player's `x` position to the maximum value so the player does not accidentally go off screen to the right.

Inside your `update` method, create an `if` statement that checks if `this.position.x >= canvas.width - this.width * 2`.

Step 34:

Inside your `if` statement, assign `canvas.width - this.width * 2` to `this.position.x`.

This will ensure that the player's `x` position will never exceed the right edge of the canvas.

Step 35:

The next step is to use the `new` keyword to create a new instance of the `Player` object and assign it to a new `const` variable called `player`.

Step 36:

Now it is time to see your new player drawn on the screen.

Start by creating an empty arrow function called `startGame`.

Step 37:

Inside your `startGame` function, you will need to display the `canvas` element and hide the `startScreen` container.

Use `canvas.style.display` to change the display value to `"block"`.

Below that, use `startScreen.style.display` to change the display value to `"none"`.

Step 38:

To visualize the player on the screen, you need to draw it on the canvas.

Inside the `startGame` function, call the `.draw()` method of your `player` object.

Step 39:

Now it's time to add the functionality for the start game button.

Add an `addEventListener` to the `startBtn` and pass in a `click` event and a reference to the `startGame` function.

Click on the start game button, and you should see a light blue square on the screen which represents the main player.

Step 40:

Now that you can see the player on the screen, it is time to start adding the functionality for moving the player across the screen.

Create a new empty arrow function called `animate`.

Step 41:

The requestAnimationFrame() web API, takes in a callback and is used to update the animation on the screen. The `animate` function will be responsible for updating the player's position and continually drawing it on the canvas.

Inside the `animate` function, call the `requestAnimationFrame()` API and pass `animate` as the argument.

Step 42:

As the player moves through the game, you will need to clear the canvas before rendering the next frame of the animation.

You can use the clearRect() Web API to accomplish this. It takes in an `x`, `y`, `width`, and `height` arguments.

Below your `requestAnimationFrame`, call the `clearRect()` method on the `ctx` variable and pass in `0, 0, canvas.width, canvas.height` as the arguments.

Step 43:

The next step is to update the player's position as it moves throughout the game.

Below your `ctx.clearRect()`, call the `update()` method on the player.

Step 44:

To manage the player's movement in the game, you will need to monitor when the left and right arrow keys are pressed.

Create a new `const` variable called `keys` and assign it an empty object.

Step 45:

Inside the `keys` object, add a new key called `rightKey` and assign it an object with the key-value pair of `pressed: false`.

Below the `rightKey` object, create a `leftKey` object and assign it an object with the key-value pair of `pressed: false`.

Step 46:

The next step is to add the logic for increasing or decreasing a player's velocity based on if they move to the left or right of the screen.

Inside the `animate` function, create an `if` statement where the condition checks if the right key was pressed and the player's `x` position is less than `proportionalSize(400)`.

You need to use the `proportionalSize` function here to make sure the player's `x` position is always proportional to the screen size.

Step 47:

Inside the `if` statement, assign the number `5` to the player's `x` velocity.

Step 48:

Add an `else if` statement where the condition checks if the left key was pressed and the player's `x` position is greater than `proportionalSize(100)`. You need to use the `proportionalSize` function here to make sure the player's `x` position is always proportional to the screen size.

Inside the `else if` statement, assign the number `-5` to the player's `x` velocity.

Step 49:

Add an `else` clause that assigns the number `0` to the player's `x` velocity.

Step 50:

The next step is to add the functionality that will be responsible for moving the player across the screen.

Create a new arrow function called movePlayer that has three parameters called key, xVelocity, isPressed.

Step 51:

In the game, the player will interact with different checkpoints. If the isCheckpointCollisionDetectionActive is false, then you will need to stop the player's movements on the x and y axes.

Start by creating an if statement where the condition checks if the isCheckpointCollisionDetectionActive is false.

Remember that you can use the ! operator to check if the variable is false.

Step 52:

Inside the if statement, set the player's x velocity to 0 and the player's y velocity to 0.

Below that, add a return statement.

Step 53:

Below the if statement, create a switch statement with a value of key.

Step 54:

The first case you will want to add is when the left arrow key is pressed.

Inside the switch statement, add a new case called "ArrowLeft".

Step 55:

Inside the `case` clause, assign `isPressed` to `keys.leftKey.pressed`.

Below that, add an `if` statement that checks if `xVelocity` is equal to 0. If so, assign the `xVelocity` to `player.velocity.x`.

Step 56:

Below your `if` statement, use the subtraction assignment operator to subtract the `xVelocity` from `player.velocity.x`.

To close out this case, make sure to add a `break` statement.

Step 57:

The player can jump up by using the up arrow key or the spacebar.

Add three new cases for `"ArrowUp"`, `" "`, and `"Spacebar"`. Remember that you can group cases together when they share the same operation.

Inside those cases, use the subtraction assignment operator to subtract `8` from `player.velocity.y`.

To close out these cases, make sure to add a `break` statement.

Step 58:

The last case you will need to add will be for `"ArrowRight"`.

Inside that case, assign `isPressed` to `keys.rightKey.pressed`.

Add an if statement that checks if `xVelocity` is equal to 0. If so, assign the `xVelocity` to `player.velocity.x`.

Below that `if` statement, use the addition assignment operator to assign the `xVelocity` to `player.velocity.x`.

Step 59:

Now it is time to add the event listeners that will be responsible for calling the movePlayer function.

Start by adding an addEventListener to the global window object.

For the arguments, pass in the keydown event and an arrow function that uses the destructuring assignment to get the key property from the event object in the event listener parameter.

Here is the syntax for using the destructuring assignment in the parameter list of the arrow function:

Example Code:

```
btn.addEventListener('click', ({ target }) => {

  console.log(target);

});
```

Step 60:

Inside the arrow function, call the movePlayer function and pass in key, 8, and true as arguments.

Step 61:

Add another addEventListener to the global window object and pass in the keyup event and use destructuring to pass in the key property from the event.

Step 62:

Inside the callback function, call the `movePlayer` function and pass in `key`, `0`, and `false` as arguments.

Step 63:

Before you can start moving your player across the screen, you will need to use the `animate` function.

Inside the `startGame` function, delete `player.draw()` and call the `animate` function.

Click the Start Game button and use the left and right arrow keys to move the player across the screen. You can also use the spacebar or the up arrow key to jump up.

Step 64:

The next step is to create the platforms and platform logic.

Start by creating a new `Platform` class.

Step 65:

Inside the `Platform` class, create a constructor that takes in the `x` and `y` coordinates.

Step 66:

When working with objects where the property name and value are the same, you can use the shorthand property name syntax. This syntax allows you to omit the property value if it is the same as the property name.

Example Code:

```
// using shorthand property name syntax
obj = {
  a, b, c
}
```

The following code is the same as:

Example Code:

```
obj = {
  a: a,
  b: b,
  c: c
}
```

Inside the constructor, add `this.position` and assign it an object with the `x` and `y` coordinates. Make sure to use the shorthand property syntax .

Step 67:

Next, add a `width` property to the constructor and assign it the number 200.

Don't forget to use the `this` keyword to access the properties.

Step 68:

Below that, add a `height` property and assign it the number `proportionalSize(40)`. You need to use the `proportionalSize()` function to make sure the `height` is proportional to the screen size.

Remember to use the `this` keyword to access the properties.

Step 69:

Next, add a `draw` method to the `Platform` class.

Step 70:

Inside the `draw` method, assign `"#acd157"` to the `ctx.fillStyle`.

Below that, call the `ctx.fillRect` method and pass in the `x` and `y` coordinates, along with the `width` and `height` properties. Remember to include `this` before each property.

Step 71:

The next step will be to create a list of positions for the platforms.

Start by creating a new `const` variable called `platformPositions` and assign it an empty array.

Step 72:

Inside the `platformPositions`, you will need to add the list of positions for the platforms.

Add a new object that has an `x` property with a value of `500` and a `y` property with a value of `proportionalSize(450)`.

Step 73:

Below that, add another object with an `x` property with a value of `700` and a `y` property with a value of `proportionalSize(400)`.

Step 74:

Add the rest of the platform positions to the `platformPositions` array with the following values:

Example Code:

```
x=850  y=proportionalSize(350)

x=900  y=proportionalSize(350)

x=1050 y=proportionalSize(150)

x=2500 y=proportionalSize(450)

x=2900 y=proportionalSize(400)

x=3150 y=proportionalSize(350)

x=3900 y=proportionalSize(450)

x=4200 y=proportionalSize(400)

x=4400 y=proportionalSize(200)

x=4700 y=proportionalSize(150)
```

Step 75:

The next step is to create a list of new platform instances using the `Platform` class. You will later reference this list to draw the platforms on the canvas.

Start by creating a new `const` variable called `platforms` and assign it `platformPositions.map()`.

Step 76:

In the map callback function, pass in `platform` for the parameter and implicitly return the creation of a new `Platform` instance with the `platform.x` and `platform.y` values passed in as arguments.

Step 77:

Inside the `animate` function, you will need to draw each of the platforms onto the canvas.

Add a `forEach` loop that iterates through the `platforms` array.

Inside the callback function, add a `platform` parameter and for the body of the function call the `draw` method on each `platform`.

Step 78:

If you try to start the game, you will notice that the platforms are rendered on the screen. But as the player moves to the right, the platform does not move with it.

To fix this issue, you will need to update the platform's `x` position as the player moves across the screen.

Inside the `animate` function, add a condition to check if the right key was pressed and if the `isCheckpointCollisionDetectionActive` is true.

Step 79:

Inside your condition, add a `forEach` loop to iterate through the `platforms` array.

Inside the loop, use the subtraction assignment operator to subtract 5 from the platform's `x` position.

Step 80:

Next, add an `else if` statement to check if the left key was pressed and if `isCheckpointCollisionDetectionActive` is true.

Inside that condition, add a `forEach` loop to iterate through the `platforms` array.

Inside the loop, use the addition assignment operator to add 5 to the platform's `x` position.

Step 81:

When you start the game, you will notice that the position of the platforms is animating alongside the player. But if you try to jump below one of the platforms, then you will jump right through it.

To fix this issue, you will need to add collision detection logic to the game.

Start by calling the `forEach` method on the `platforms` array. For the callback function pass in `platform` as the parameter.

Step 82:

Inside the callback function, create a new `const` variable called `collisionDetectionRules` and assign it an empty array.

Inside that array, add a boolean expression that checks whether the player's `y` position plus the player's height is less than or equal to the platform's `y` position.

Step 83:

Add another boolean expression that checks if the sum of the player's `y` position, height, and `y` velocity is greater than or equal to the platform's `y` position.

Step 84:

Below that boolean expression, add another boolean expression that checks if the player's `x` position is greater than or equal to the platform's `x` position minus half of the player's width.

Step 85:

Add one last boolean expression that checks if the player's `x` position is less than or equal to the sum of the platform's `x` position plus the platform's width minus one-third of the player's width.

Step 86:

Next, add an `if` statement that checks if every rule in the `collisionDetectionRules` array is truthy. Make sure to use the `every` method for this.

Inside the body of the `if` statement, assign the number 0 to the player's `y` velocity followed by a `return` statement.

Step 87:

Create a new `const` variable called `platformDetectionRules` and assign it an empty array.

Step 88:

Inside that array, add a boolean expression that checks if the player's x position is greater than or equal to the platform's x position minus half of the player's width.

Step 89:

Below that boolean expression, add another boolean expression that checks if the player's `x` position is less than or equal to the sum of the platform's `x` position plus the platform's width minus one-third of the player's width.

Step 90:

Add another boolean expression that checks if the player's `y` position plus the player's height is greater than or equal to the platform's `y` position.

Below that, add another boolean expression that checks if the player's `y` position is less than or equal to the sum of the platform's `y` position plus the platform's height.

Step 91:

Add an `if` statement that checks if every platform detection rule is `true`. Make sure to use the `every` method for this.

Step 92:

Inside the body of the if statement, assign platform.position.y + player.height to the player's y position.

Then, assign gravity to the player's y velocity.

Now, when you start the game, you will be able to jump underneath the platform and collide with it.

Step 93:

The last portion of the project is to add the logic for the checkpoints. When a player collides with a checkpoint, the checkpoint screen should appear.

Start by creating a new class called CheckPoint.

Step 94:

Inside that CheckPoint class, add a constructor with x, y and z parameters.

Step 95:

Inside the constructor, create an object with x and y parameters and assign it to the position.

Remember to use the this keyword to access the properties.

Step 96:

The next step is to add the width and height to the CheckPoint class.

The width and height should be proportionalSize(40) and proportionalSize(70) respectively.

Step 97:

Below the checkpoint's width and height properties, use the this keyword to add a new claimed property and assign it the value of false. This property will be used to check if the player has reached the checkpoint.

Step 98:

Now you need to create a draw method for the CheckPoint class.

Inside the draw method, assign the fillStyle property on the ctx object the hex color "#f1be32".

Below the fillStyle property, use the fillRect method on the ctx object and pass in the x, y, width, and height properties as arguments.

Step 99:

The last method you will need to add to the CheckPoint class is the claim method.

Inside the claim method, assign 0 to the width and height properties of the CheckPoint instance.

Below those properties, assign Infinity to the y position.

Lastly, assign true to the claimed property.

Step 100:

Use const to create a new array called checkpointPositions.

Inside that array, add an object for each of the following positions:

Example Code:

```
x: 1170, y: proportionalSize(80), z: 1

 x: 2900, y: proportionalSize(330), z: 2

 x: 4800, y: proportionalSize(80), z: 3
```

Step 101:

The next step is to create a list of new `checkpoint` instances using the `CheckPoint` class.

Start by creating a new `const` variable called `checkpoints` and assign it `checkpointPositions.map()`.

For the `map` callback function, pass in `checkpoint` for the parameter and implicitly return the creation of a new `CheckPoint` instance with the `checkpoint.x`, `checkpoint.y` and `checkpoint.z` values passed in as arguments.

Step 102:

Inside the animate function, you will need to draw each of the `checkpoints` onto the canvas.

Add a `forEach` loop that iterates through the `checkpoints` array.

Inside the callback function, add a `checkpoint` parameter and for the body of the function call the `draw` method on each `checkpoint`.

Step 103:

Inside your condition, add a `forEach` loop to iterate through the `checkpoints` array. Use `checkpoint` as the parameter name for the callback function.

Inside the loop, use the subtraction assignment operator to subtract 5 from the checkpoints's `x` position.

Step 104:

Inside your `else if` statement, add a `forEach` loop to iterate through the `checkpoints` array. Use `checkpoint` as the parameter name for the callback function.

Inside the loop, use the addition assignment operator to add 5 to the checkpoints's `x` position.

Step 105:

The next step is to create a function that will show the checkpoint message when the player reaches a checkpoint.

Create a new arrow function called `showCheckpointScreen` that takes in a `msg` parameter.

Step 106:

Inside the `showCheckpointScreen` function, set the `checkpointScreen` `style.display` property to `"block"`.

Step 107:

Set the `checkpointMessage`'s `textContent` property to the `msg` parameter.

Step 108:

Create an `if` statement that checks if `isCheckpointCollisionDetectionActive` is true.

Inside the `if` statement, add a `setTimeout()` that takes in a callback function and a delay of 2000 milliseconds.

For the callback function, it should set the `checkpointScreen` `style.display` property to `"none"`.

Step 109:

The last few steps involve updating the `animate` function to display the checkpoint screen when the player reaches a checkpoint.

Start by adding a `forEach` to the `checkpoints` array. For the callback function, use `checkpoint`, `index` and `checkpoints` for the parameters.

Step 110:

Create a new `const` variable called `checkpointDetectionRules` and assign it an empty array.

Inside that array, add a boolean expression that checks if the player's `position.x` is greater than or equal to the checkpoint's `position.x`.

Step 111:

Add another boolean expression that checks if the player's `position.y` is greater than or equal to the checkpoint's `position.y`.

Below that statement, add another boolean expression that checks if the player's `position.y` plus the player's `height` is less than or equal to the checkpoint's `position.y` plus the checkpoint's `height`.

Below that statement, add the isCheckpointCollisionDetectionActive variable.

Step 112:

You will need to add two more checkpoint detection rules to the checkpointDetectionRules array.

The first rule should check if the player's x position minus the player's width is less than or equal to the checkpoint's x position minus the checkpoint's width plus the player's width multiplied by 0.9. This will ensure that the player is close enough to the checkpoint to claim it.

The second rule should check if index is strictly equal to 0 or if the previous checkpoint(checkpoints[index - 1].claimed) is true. This will ensure that the player can only claim the first checkpoint or a checkpoint that has already been claimed.

Step 113:

Next, add an if statement that checks if every rule in the checkpointDetectionRules array is true.

Make sure to use the every method for this.

Step 114:

Inside the if statement, call the claim method on the checkpoint object.

Step 115:

The next step is to write a condition that checks if the player has reached the last checkpoint.

Start by adding an `if` statement that checks if the `index` is equal to the length of the `checkpoints` array minus one.

Step 116:

Inside the condition, you want to first set the `isCheckpointCollisionDetectionActive` to false.

Then you will need to call the `showCheckpointScreen` function and pass in the string `"You reached the final checkpoint!"` as an argument.

Lastly, you will need to call the `movePlayer` function and pass in the string `"ArrowRight"` as the first argument, the number `0` as the second argument, and the boolean `false` as the third argument.

Step 117:

The last thing you will need to do is add an `else if` statement.

Your condition should check if the player's `x` position is greater than or equal to the checkpoint's `x` position and less than or equal to the checkpoint's `x` position plus `40`.

Inside the body of the `else if` statement, you will need to call the `showCheckpointScreen` function and pass in the string `"You reached a checkpoint!"` as an argument.

Congratulations! You have completed the platformer game project!