GD 20
Rock Climbing, Part 1

We went from variable to arrays and then to classes and objects. In this project, we will combine all 3 topics together. This means that we will create one or many objects and then store the newly created object into an array. Since an array is a group of memory elements and it is like a line, the game objects are more organized and this is much better than having many variables scattered around. Next, we will use the square brackets to select one element of the array and then use the member access operator to access a private variable or function of the chosen object. We can efficiently update all game objects by combining all 3 together inside a "for" loop.

Rock Climbing. How well do you know California? The goal of Rock Climbing is to start at the bottom and climb up to the top. The Rock Climber makes it up to the top by matching the question with 1 of 4 possible answers. The Rock Climber can get into position to choose one of the answers by zip lining sideways or vertically.

1. arrays 2. class definition 3. "**this.**" versus object name

---

1. **Write the code below in between <script> </script>.** The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ======================================
* QA class
* ====================================== */
class QA
{
    constructor( question, answer)
    {
        this.quest = question;
        this.an    = answer;
    }

    check_if_matching( uAnswer )
    {
        console.log( "#1" );
        return (this.an == uAnswer) ? 1 : 0;

    }

    get_qa_pair()
    {
      console.log( "#2 );
      return { q: this.quest, a: this.an };

    }

    // -- JS Challenge 1

}
```

Explanation

The code above is the class definition for QA, which stands for question and answer. The class definition has 2 private variables and they are named `this.quest` and `this.an`. Remember that in Javascript, we define private variables inside the constructor and we put "`this.`" in front of the variable name.

The class definition also has two private functions (the constructor doesn't count as a private function). We know that `this.quest` and `this.an` and the two private functions belong to class QA because they **are all inside the open and closing curly braces of class QA.**

Since we are inside the class definition, we can use "`this.`" as a short cut instead of saying QA.

JS Challenge 1. Can a private function call another private function?

For this challenge, write the private function `call_all()`. Inside this private function, call the other two private functions.

What do you think will happen?

2. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ============================================
* Zipline class constructor
* ============================================ */
constructor( xPos1, yPos1, xPos2, yPos2, stepSize )
{
      this.xStPos   = xPos1;
      this.yStPos   = yPos1;
      this.xEnPos   = xPos2;
      this.yEnPos   = yPos2;
      this.stepSize = stepSize;
```

Explanation

In the Rock Climbing game, we must select one of the answers in order to climb up the rock. We select an answer by ziplining to one of the four possible answers. We may need to reposition ourselves to a different location by using a zipline and then using another zipline to choose one of the answers. This means that we have to create copies of the zipline and each zipline copy will be different. **This is the perfect situation for objects.**

We know that the "new" operator is used to create an object. How do we initialize the object with custom values? The answer is to use the **constructor** inside the class definition.

The **constructor** above has 5 input parameters and these are used to customize the starting and ending point of the zipline. Also, the last parameters are used to customize the speed of our zipline and this determines how fast we travel.

Inside the constructor, we create several private member attributes (ie. private variables). We know that they are private member attributes because of the word "**this**" and the period (.) that is in front of the private attributes.

Remember, the period is the **member access operator** and it shows ownership. To the **left** of the member access operator is the **owner** and to the **right** is **what is being owned**. This means that the owner is Zipline and Zipline owns the private member attributes.

3. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ======================================================
* Check if done zipping
* ====================================================== */
check_end_hb( endXPos, endYPos )
{
      // -- the "if" statement is all one line of code
      if( endXPos-10 <= this.xStPos && this.xStPos <= endXPos+10 && endYPos-10
<= this.yStPos && this.yStPos <= endYPos+10 )
      {
          console.log( "ready to stop zipping" );
          return 1;
      }
      else
      {
          return 0;
      }
}
```

Explanation

The code above assumes that the player has already pressed on the left button of the mouse and the game character is zipping towards the end of the zipline. How do we know when to stop the game character?

In game development, we have to check for collision and we do this by checking if the end of the zipline is within the hitbox of our game character. The long "**if**" statement uses the less than sign to check if the ending x and y points of the zipline are inside the hitbox. Notice that we have a lower boundary and the upper boundary. If the ending x and y points of the zipline are inside the upper and lower boundaries, then we have a collision.

4. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ========================================
* Create New Zipline
* ======================================== */
if( newLF )
{
       console.log( "ready for new zip" );
        zipLineArr = [];
        newLF      = 0;
        sbStill    = 0;

       // -- all one line of code
       var zLObj = new ZipLine( playerArr[playerWB].pXPos, playerArr[playerWB].pYPos,
mouseXEnd, mouseYEnd, 3 );

        zipLineArr.push( zLObj );
        zLNum++;
```

Explanation

In step 2, we created a class definition for a Zipline class. Inside the class definition, we used the constructor to initialize any newly created objects.

The code in **green** uses the "**new**" operator to create an object. Next, we put data inside the open and closing parenthesis and **this data is given** to the constructor. If we look at the **constructor of step 2**, the data inside the open and closing parenthesis match up.

For example,

`playerArr[playerWB].pXPos` matches up with `xPos1 of the constructor`

`playerArr[playerWB].pYPos` matches up with `yPos1 of the constructor`

JS Challenge 2. Write "for" loop for `zipLineArr`

Write a "**for**" loop using the following
1. create a digital key named z and load the data 0 into it
2. keep going until the length of the array
3. jump z by 1 position

```
/* ============================================
* Zip line Arr
* ============================================ */
```

JS Challenge 3. Write an "if" statement to check hit box. This code will combine array with objects

Wrote an "**if**" statement. Inside the "**if**" statement
1. select position z of `zipLineArr.`
2. next, use the member access operator to call the private function `check_end_hb();`

Inside the open and closing parenthesis of the private function, give it the input
2.1. `zipLineArr[z].xEnPos`
2.2. `zipLineArr[z].yEnPos`

```
/* =========================================
* Check Hit Box Zip
* ========================================= */
```

5. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* =========================================
* User moves mouse
* ========================================= */
if( canonMF  )
{

        cContext.beginPath()
        cContext.lineWidth   = 3;
        cContext.strokeStyle = "#ffffff";
        cContext.stroke();

       // -- code should be single line
        var pObj = new Path2D( "M " + (playerArr[selectedPlayerId].pXPos+4) + " " +
(playerArr[selectedPlayerId].pYPos+10) + " L " + mouseXEnd + " " + mouseYEnd );

        cContext.stroke( pObj );

}
```

Explanation

Before the player can zip the game character from one place to another, we have to give the user the ability to see where they potentially want to go. This will help the user visualize their next move.

The code above will create a zipline with a starting position of the player's x and y position. The code above uses the "**M** " command to tell the game to "**move to**" the player's x and y position.

The ending position will be with the "L " command and it will tell the game "line to" x and y position of the mouse.

So, the code above translates into "Move To player's x and y position, Line To mouse's x and y position ".

Finally, we actually draw the line by using the `cContext.stroke( pObj );`

6. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ============================================
* Draw player in their new position
* ============================================ */

for( var p = 0; p < playerArr.length; p++ )
{

	// -- move player using zipline
	if( playerArr[p].zipLineObj.check_end_hb( playerArr[p].zipLineObj.xEnPos,
playerArr[p].zipLineObj.yEnPos ) )
	{
		playerArr[p].zipLineObj.load_new_step_size( 0 );

	}
	else
	{
		playerArr[p].zipLineObj.move_one_step();
		playerArr[p].update_xy_pos();
	}

	// -- direction control
	playerArr[p].pXPos += playerArr[p].playerDirArr[playerArr[p].moveXIndex];
	playerArr[p].pYPos += playerArr[p].playerDirArr[playerArr[p].moveYIndex];

	playerArr[p].playerDirArr[playerArr[p].moveXIndex] = 0;
	playerArr[p].playerDirArr[playerArr[p].moveYIndex] = 0;

	// -- draw new position
	cContext.beginPath();
	cContext.fillStyle = playerArr[p].pColor;

	// -- code should all fit on one line
	cContext.drawImage( playerArr[p].pImage, playerArr[p].pXPos, playerArr[p].pYPos,
playerArr[p].pWidth, playerArr[p].pHeight );

	cContext.fill();

}
```

Explanation

The code above uses a "for" loop to loop through playerArr. Right now, we only have 1 player and so the loop will only run once. However, in case we would like to have more players, the code above doesn't need to be changed.

The code starts off by moving the player until the player reaches the end. Remember, we use hit box calculation to check if the player has reached the end.
1. If we are within the hit box, we then load the step size to 0 and this stops the player from moving.
2. else, we are not done moving and so move one step

Since the game character is in a different position from the previous position, we have to redraw the game character again and the code in **blue** does that.

JS Challenge 4 - Type solution inside Zoom chat
1. create a digital key by creating a variable named **j** and load the data 2 into it
2. Inside the open and closing parenthesis of the "**for**" loop, select position **j** of **wheelsArr**
3. variable **j** should then jump by 2 positions
3. send an alert. Inside the open and closing parenthesis of the alert, select position **j** of **wheelsArr**

JS Challenge 5 - Go to JS 1 Website
1. write a class definition of **FoodItem**
2. write a constructor that accepts **2 inputs**, **foodTitle** and **foodCost**
3. inside the constructor, create **2 private variables** named **fTitle** and **fCost**.
4. write **one private function** named **show_food()**
4.1. inside the private function **show_food()**, send an alert. Inside the open and closing parenthesis of the alert, call the **private variables fTitle** and **fCost**.