Introduction

Welcome to our Javascript 101 course! In this course, we will learn the basics of the Javascript language and apply our knowledge towards developing a game. We will start with the basics of the language, piecing together concepts while learning Javascript syntax. Then, we will learn the basics of building a game with Javascript including adding sprites, implementing movement and logic, and adding collision detection. Finally, we will put everything together and build our very own game.

Why Learn Javascript?

Javascript is one of the most commonly used languages today. All over the world, Javascript powers webpages, games, mobile and web apps, as well as data science and machine learning applications. It's an easy to use language that can run anywhere; all you need is a browser. Its flexible nature makes it easy for anyone to pick up, even if you have no previous coding experience. As it is used everywhere, there is a ton of support and solutions to almost any question you could ask.

What Topics will we Cover?

Javascript language basics:

- Variables and strings
- Operators
- Arrays
- Functions, parameters, and return values
- If, else-if, and else statements
- While loops
- Break, continue, and return statements
- For loops
- Objects and prototypes
- Classes and inheritance

Game making:

- Creating the canvas
- Drawing on the canvas
- Movement and logic
- Touch events and player interaction
- Collision detection and end game logic
- Adding images and creating sprites

Let's start with the basics of learning to code using Javascript. We will cover the concepts and syntax we need to start writing real Javascript code and, because we're going to be using our code to create a game, we will use relevant examples that pertain to video games. But first of all, let's make sure we're set up and ready to go.

How do I Follow Along?

For the language basics portion, we won't be running any complex programs. An online compiler usually works best for learning language basics as you don't have to download anything and can run code right in the browser. Compilers such as **rextester.com**, **jsbin.com** or **playcode.io** allow you to write and run Javascript code to see results in real time. Any time you want to print a value to see the results of some code, simply run the code...

```
console.log(value);
```

... replacing with the variable or value that you want to print. Another option is to use a website templating tool like jsbin. That allows you to write CSS and HTML code in addition to Javascript but we would only focus on the Javascript aspect. You can close the HTML, CSS, and Output windows and just use the Javascript window to write code and the console window to output code.

The restrictions of some systems such as rextester and jsbin is that some of them don't recognize the class keyword so you cannot create classes in that way and some don't recognize the backtick ("`") character. This is different from normal quotes (double or single) and is needed for string interpolation. **Playcode.io** allows you to create classes and use backticks, so that's an alternative.

Variables provide a way to store and keep track of values within a program. We simply assign a value to a unique name, also referred to as an identifier, and then we can retrieve and modify the value throughout a program's execution. In **Javascript**, as with most programming languages, variables have types which dictate the kind of data that they can hold. Unlike many "strongly typed" languages, we don't explicitly assign a type to a variable; we need only provide a value and the type is inferred.

The main **variable** types we will focus on are numbers, booleans, and strings. Numbers can be either decimal or whole numbers and are declared like this:

```
var age = 1;
var currentLevel = 2.2;
```

Booleans represent true or false values and are created like this:

```
var isGameOver = true;
var isPlayerAlive = false;
```

We can change their values once assigned but we have to have created the variable first. We can assign a literal value (such as a number or true/false) or we can assign the value of another variable. For example:

```
currentLevel = 2.3;
isGameOver = isPlayerAlive;
```

Declaring (Creating) Variables

When declaring or creating a variable, we need to type **var** in front of the unique name that we choose to give that variable. Once this is done, we can assign a value to our variable and use it throughout our program. After this declaration, we no longer need to type **var** each time we use or access the variable throughout our program.

Comments

When working in programming (everything from data science to game development), it is common to see **comments** littered throughout code. This is a great habit to develop as you grow into a developer because it allows you to document your intentions throughout your code. Comments can be used to remind yourself, or someone else who looks at your code, what a particular part of your code is meant to do. You can also use them to comment out lines that you are no longer wanting to use.

In Javascript, comments are written like this:

```
// This is a comment
```


The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

There are two additional ways of creating variables within our Javascript code using the **const** and **let** keywords. **Const** allows you to create constants which are variables that cannot change their value once assigned. If we try to change the value, it throws an error and causes the program to halt. For example:

```
const maxHealth = 100;
maxHealth = 50; // Not allowed, will throw an error
```

As a quick aside, anything we put after the // is a **comment** and is ignored by the compiler. It's not code that is run and is just there for coders to read.

We can also create variables (whose value *can* change) with the **let** keyword. The difference between **let** and **var** is that **let** can only be accessed *within* the scope or block in which it's declared. Essentially, when we create a **let** with { }, we can't access it outside of the { } without encountering an error, but we can with a **var**.

The following code has been updated, and differs from the video:

```
if (true) {
  var isGameOver = false;
  let isNotGameOver = true;
  isNotGameOver = false;
}
console.log(isGameOver); // false
console.log(isNotGameOver); // Throws an error because we cannot access a let outside
 of the { }
```

As **var** can be accessed **globally**, the code above prints **isGameOver** as **false** just fine. However, trying to access **isNotGameOver** will cause the code to stop running and throw an **error** as it is a **let** (not a var) thus being a **local variable**.

Strings are slightly more complex variables compared to what we've just looked at. They represent arrays of characters and have more functionality attached to them, though we can treat them like basic variables for now. They represent any text in a program such as messages, names, titles, etc. and are created like this:

```
var title1 = "Hello";

var title2 = 'World';
```

Note that you can use either "" or '' for strings as long as you don't mix the two and the string value is put between the quotes.

Strings come with extra functionality (we will cover some here but for more functionality search for Javascript string functions). We can retrieve properties or modify the string using functions. For example, we can get the length with:

```
var titleLength = title1.length; // 5
```

We can also turn a string to upper or lower case using:

```
var alternateTitle = title1.toUpperCase();

var alternateTitle = title1.toLowerCase();
```

Finally, we can take a slice of a string by calling the .slice(startIndex, endIndex) method like so:

```
var slice = title1.slice(0, 2); // slice = "He"
```

We can also feed values into strings via **string interpolation**. This converts the value of a variable into a string and feeds it into the existing string. We just feed the value or expression inside **${}** using **backticks**. For example, if we want to print the message: "I am years old.", using the age variable, we can do as follows:

```
var age = 26;
var ageMessage = `I am ${age} years old.`; // ageMessage = "I am 26 years old."
```

Note: If that doesn't work for you, try using the "+" operator instead (it varies accordingly to the editor you're using).

```
var age = 26;
var ageMessage = "I am " + age + " years old."; // ageMessage = "I am 26 years old."
```


Operators

Javascript contains 5 types of operators that we will cover: assignment, arithmetic, comparison, logical, ternary.

Assignment is the simplest and consists of just one: =. We have already seen it in action plenty of times and it is used to assign a value to a variable or to store a new value. For example, if we wanted to change the first number variable (number1) to 5, we could do this:

```
age = 5;
```

In order to reassign a value, the variable must already have been created.

Instead of just assigning a literal value (like with 5), we can assign the results of an expression, such as performing a mathematical operation. **Arithmetic** operators are used to perform some mathematical operation on numbers and return a number. The basic operations include: **+, -, *, /, %** and **. The % is the **modulus** operator and returns the remainder of a division. The ** is the **exponentiation** operator and is used to raise something to the power of something else. These operations are generally performed between two numbers. For example, I could perform basic addition like this:

```
var result = 1 + 2;
```

To yield the result of 3, I could also use an existing variable like this:

```
result = age + 5;
```

Considering that number1 variable was reassigned the value of 5, the result now contains 10. The other operators work in the same way. We can also use the addition operator to join strings together. I could work with the previous strings to do something like this:

```
var helloWorld = title1 + “, world!”;
```

As string1 contains “Hello” and I am appending “, world!” to it, I get the result “Hello, world!” stored in helloWorld. I can combine these operators in any way I want. Although order of operations is correctly observed, it is usually a good idea to include brackets to communicate intent to anyone who is reading your code as in the example:

```
result = (age * 3) + (4 / 2);
```

The result is the same with and without the brackets but the intent is clearer with the brackets. These operators return a number or a string depending on the variables used.

Arithmetic and assignment operators can be combined in a few ways using the operators: ++, –, +=, -=, *=, /=, and %=. The ++ and — operators are used to increment or decrement a value like so:


```
age++;
```

This stores the value of 3 in number2 as it contained 2 before. The others are used like so:

```
age -= 1;
```

This is the same as:

```
age = age - 1;
```

And stores the result of 2 back into number2. The others work in the same way and the += operator can be used with strings.

You might have noticed throughout these lessons we've been using these strokes**//**. This is just to indicate a **comment** and is a part of the code that is ignored by the compiler and works as a reminder for you, or someone else looking at your code, what your intentions were.

Comparison operators return a **boolean** value. These include: >, >=, 2 && 4 > 6; // false boolResult = 2 > 1 || isGameOver == true; // true

The first result is false as the **&&** operator returns true only if both cases on the left and right return true whereas only one of the two cases has to return true with the **||** operator.

There is only one **ternary** operator and it looks like this:

```
var score = isGameOver == true ? 0 : 1;
```

It works to test a condition (which returns true or false) and then store a value depending on the results. If the condition returns true, we store the first result to the left of **:** whereas if it returns false, we store the second value. It works like an if-else statement but more on that in a later lesson.

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

Arrays or **lists** provide a way to store multiple values inside a single variable. Generally, we try to restrict arrays to contain values of only one type, but we can use multiple types of values and variables within an array. Arrays are created like variables but we put the values inside square brackets **[]** separated with **commas**. A simple example would be:

```
var items = [“clothes”, “food”, “tool”];
```

Arrays can also be created **empty** (with nothing in the []) or they can contain a mix of numbers, strings, booleans, and even objects. Just like with variables, we can modify or access the entire array, like this:

```
items = [“shirts”, “fruits”, “axe”];
```

But sometimes we want to access or modify an *individual* element within an array. We can do this by accessing the **index** of the array. Indexing starts at 0, which means the first element is stored at index 0 rather than 1 and the last is at index (array.length – 1). Be careful not to try to access an index that doesn’t exist, otherwise your code won’t run. Some examples are:

```
var tool = items[2]; // tool = “axe”
items[0] = “coat”; // items = [“coat”, “fruits”, “axe”]
```

Arrays also come equipped with extra functionality to allow you to add, edit, or delete values within them as well as retrieve properties such as the length. For example, we can get the length of an array by calling the **.length** function. You can also do the same thing with strings:

```
var length = items.length; // 3
length = items[0].length; // 4
```

We can call upon functions to help us to add or remove items. Use the **.push()** function to add an item onto the end of an array and the **.pop()** function to remove the last element and return it:

```
items.push(“money”); // items = [“coat”, “fruits”, “axe”, “money”]
items.pop(); // returns “money” and items = [“coat”, “fruits”, “axe”]
```

There are other functions attached to arrays but we won’t go over all of them in this tutorial. Feel free to check out others by exploring Javascript array functions.

The following instruction has been updated, and differs from the video:

In the video lesson (at **3:30**), the presenter says “unless it’s a **constant**” while typing the keyword “**let**“. He actually meant to write “**const**” as a let variable can be reassigned and, therefore, is not a constant.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

Multidimensional arrays are basically **arrays of arrays**.

Let's take into account the **levels** of a game, we could represent them as follows:

```
var levels = [1, 2, 3, 4, 5]
```

However, we usually have **sublevels** in realistic games. So now we can represent them with a multidimensional array by specifying **rows** and **columns** in a **matrix-like** structure:

```
var levels = [
[1.1, 1.2, 1.3],
[2.1, 2.2, 2.3, 2.4],
[3.1, 3.2]
];
```

Each **element** of this new array is an array itself and, within each of these elements, we have the individual values for the levels.

Note that JavaScript supports different lengths for the arrays within an array.

To access the first array of "levels", we can do:

```
var firstWorld = levels[0]; //[1.1, 1.2, 1.3]
```

We can then perform operations on the resulting array normally, such as **add** values to it or **remove** values from it.

If we want to access a specific value of our "levels" array we go further by specifying its **index** (row and column) inside the element array:

```
var firstLevel = levels[0][1]; //1.2
```

Lastly, if we want to **change** a specific value in our "first-level" array, for instance, we **set** the new value to the same **position** by accessing its row and column **indexes** as shown below.

The following code has been updated, and differs from the video:

```
levels[0][1] = 1.4
```


Up until now, all of the code we have learned and written is executed any time you run a program containing it. However, sometimes we want finer control over when and where we want to execute the code we write. Using **functions**, we can write the code inside a function and **call** the function to **execute** the code exactly when we need it. This is beneficial as we can choose exactly when to execute the code (for example, on a button press), and we can write the code once and reuse it multiple times.

Functions have a **name**, a function **body** (where the code lies) and can also take **inputs** and produce **outputs**.

Functions can also access variables and other functions created outside of them. Let's say we want a function to increase the value of the variable we've called **age** by **1**. We could write something like this:

```
var age = 26;

function increaseAge() {
  age++;
}
```

This is just the function **implementation** though. The code inside won't be run until we **call** the function to execute it. We can do so like this:

```
increaseAge();
```

This will **execute** the code and increase the value of **age** by **1**.

Our function right now has no inputs or outputs and it doesn't need to. However, we often will need inputs to make functions more generic. We call these inputs **parameters** and can treat them like variables in a function body. On the other hand, outputs are called **return values**. We specify an output through a **return statement** which ends a function execution and outputs a value. It is important to note that a function exits once it reaches a return statement and executes it – meaning any code written afterwards will never run.

Our **function** right now has no **inputs** or **outputs** and it doesn't need to. However, we often will need inputs to make functions more generic. We call these inputs **parameters** and can treat them as variables in a function body. On the other hand, outputs are called **return values**. We specify an output through a return statement which ends a function execution and outputs a value. It is important to note that a function exits once it reaches a return statement and executes it – any code written afterwards will never run.

These are the benefits of inputs and outputs. As an example, let's say we have a function that will take in a first name and output the message: "Hello, ! How are you today?". We want to be able to pass firstName in as a parameter because we could be dealing with many people's names. We also want to be able to output the final message as we may have other messages. We can create such a function like this:

```
function printWelcomeMessage(firstName) {
   return "Hello " + firstName + "! How are you today?";
}
```

This takes firstName as input via a parameter and outputs the message. If a function has parameters, we have to pass in values. We can also **retrieve** the output and use it like a variable, although we don't have to if we don't need to. We can call our function and retrieve output like this:

```
var welcomeMessage = printWelcomeMessage("Nimish");
```

Parameters can also have default values which allows us to choose between passing in an **explicit** value or using the **default** value. For example, we could change our function to this:

```
function printWelcomeMessage(firstName = "user") {
  return "Hello " + firstName + "! How are you today?";
}
```

This way, when we call the function, we don't have to pass in a value for the parameter; it will take on the value of "user" unless we pass in a different value:

```
var welcomeMessage = printWelcomeMessage("Nimish"); // "Hello Nimish! How are you tod
ay?"
welcomeMessage = printWelcomeMessage(); // "Hello user! How are you today?"
```


The code in the video for this particular lesson has been updated, please see the code below for the corrections:

```
var maxHealth = 100
var currentHealth = 50
function heal(healAmount = 10) {
    var newHealth = currentHealth + healAmount;
    currentHealth = newHealth > maxHealth ? maxHealth : newHealth;
    return calculatePercent(currentHealth, maxHealth);
}

function calculatePercent(dividend, divisor) {
    return (dividend / divisor) * 100;
}

var result = heal(); // currentHealth = 60, result = 60
result = heal(50); // currentHealth = 100, result = 100
```

Note that by calculating 'newHealth > maxHealth' instead of 'newHealth > 100' we guarantee that our code stays correct even if we update the value of maxHealth later in the future.

Our function right now has no inputs or outputs and it doesn't need to. However, we often will need inputs to make functions more generic. We call these inputs parameters and can treat them like variables in a function body. On the other hand, outputs are called return values. We specify an output through a return statement which ends a function execution and outputs a value. It is important to note that a function exits once it reaches a return statement and executes it so any code written afterwards will never run.

These are the benefits of inputs and outputs. As an example, let's say we have a function that will take in a first name and output the message: "Hello, <firstName>! How are you today?". We want to be able to pass firstName in as a parameter because we could be dealing with many people's names. We also want to be able to output the final message as we may have other messages. We can create such a function like this:

```
function printWelcomeMessage(firstName) {
  return "Hello " + firstName + "! How are you today?";
}
```

This takes firstName as input via a parameter and outputs the message. If a function has parameters, we have to pass in values. We can also retrieve the output and use it like a variable, although we don't have to if we don't need to. We can call our function and retrieve output like this:

```
var welcomeMessage = printWelcomeMessage("Nimish");
```

Parameters can also have default values which allows us to choose between passing in an explicit value or using the default value. For example, we could change our function to this:

```
function printWelcomeMessage(firstName = "user") {
```

```
  return "Hello " + firstName + "! How are you today?";
}
```

This way, when we call the function, we don't have to pass in a value for the parameter; it will take on the value of "user" unless we pass in a different value:

```
var welcomeMessage = printWelcomeMessage("Nimish"); // "Hello Nimish! How are you tod
ay?"
welcomeMessage = printWelcomeMessage(); // "Hello user! How are you today?"
```

Control flow is an important aspect of every software system as it provides mechanisms to implement logic. We can choose which parts of a program's code we want to execute based on the outcome of **conditional** tests and the **current state** (values and properties) of a program.

The simplest of these is the basic **if** statement. If statements perform a test that returns true or false and executes some code if the test returns true. If the test returns false, nothing happens and the program skips over the code inside the if statement body. For example, let's pretend that we are moving a character based on keystrokes from the user. We want to detect which key is pressed and move the character forward if the key == "r":

```
var keyPressed = "r";
var position = 1;

if (keyPressed == "r") {
  position++;
}
```

Note that the test is performed in the brackets and the code to execute is in the **{}**. Unlike a function, we don't need to call this code to execute it, although generally, we would likely put these statements in a function. With this code, we will increase the position by 1 because **keyPressed == "r"** returns true. If keyPressed was anything else, the code would not execute and position remains 1.

Now we want to move the character in a different way if different keys are pressed. This will require performing another test. We could just put another if statement after the first one, but it makes more sense to introduce an **else-if** statement. We could rewrite our code to do something like this:

```
var keyPressed = "l";
var position = 1;

if (keyPressed == "r") {
  position++;
} else if (keyPressed == "l") {
  position--;
}
```

This time, the position decreases by 1 and becomes 0 because **keyPressed == "l"**. With an else-if statement, the second test is only performed if the first test fails. If the first test passes and the code is executed, all subsequent tests and code are ignored. This is different if there are multiple if statements in a row without else if statements, in which case, all tests would be performed.

Finally, there is the **else** case. This is like a default, providing some code to execute if every previous test fails. To demonstrate the concept, let's say that any other keystroke returns our character to the start (position 0). We could rewrite the above code to this:

```
var keyPressed = "h";
var position = 1;

if (keyPressed == "r") {
  position++;
} else if (keyPressed == "l") {
```



```
  position--;
} else {
  position = 0;
}
```

This resets our position back to 0 as the keyPressed is not “r” or “l”. The order of events is to test the first if statement. If that returns true, execute the code and ignore the rest. If the first test fails, move onto the second and try that. If that test fails, move onto the third and so on. If all tests fail, execute the code in the else statement as the else statement does not actually test anything. It is important to order your if and else-if statements properly as you want to prioritize the important cases first.

If statements can also be **nested**, meaning we could structure our code something like this:

```
if (isGameOver == false) {
  if (keyPressed == “r”) {
    position++;
  }
}
```

We can also test for **multiple conditions** in a single if statement with the **&&** and **||** operators. For example, we would only move forward as long as we are not at the edge of the map. We could do this like so:

```
let endPosition = 5;
if (keyPressed == “r” && endPosition   position++;
}
```

This way, position only increases by 1 if we press the right key and we are *not* at the end.

Sometimes in code, we want to execute the same code multiple times. However, it's bad practice to repeat code. When we need to run the code hundreds or thousands of times, writing the same code over and over again is impractical. Instead, we can put the code inside of a **loop** body, choose how many times we want to run the loop, and execute the code.

One way of doing this is with a **while** loop. It acts very similarly to an **if** statement but instead of running the code in the body once, it continues to run the code until the test fails. Let's say we want to move a character forward until it reaches the end position. Ignoring the previous code, we could use a loop like this:

```
var position = 1;
let endPosition = 5;

while (position     position++;
}
```

This code runs 4 times, bringing our final position to 5. On each loop iteration, we check to see if position

Break, Continue, and Return Statements

Break statements are used to exit **loops** and **if statements** prematurely. They will halt a loop execution and exit out regardless of whether or not the while loop test returns true or false. Let's expand upon the previous example and say there is an enemy that will also cause us to break out of the loop if we collide with it. We could rewrite the above code to look like this:

```
var position = 1;
var enemyPosition = 3;
let endPosition = 5;

while (position < endPosition) {
    position++;

    if (position == enemyPosition) {
        break;
    }
}
```

This will only allow our loop to run twice because on the second execution, we reach position = 3 and the if statement returns true, executing the break statement. This exits out of the loop and we would then probably execute some end-game logic. This is very similar to a return statement in a function in that a return statement will exit the function. You can also call return within a loop to break out of the loop and its enclosing function if it is all in one.

Continue statements on the other hand, skip over any remaining code in the current loop iteration and move directly onto the next. For example, let's say we have an array of numbers and we only want to print out the even ones. We could do something like this:

```
let numbers = [1, 2, 3, 4, 5, 6]
var i = 0;

while (i < numbers.length) {
    i += 1;
    if (numbers[i-1] % 2 != 0) {
         continue;
    }
    console.log(numbers[i-1]);
}
```

Checking **x % 2 == 0** is a common way to see if a number is even. We could really have just checked to see if the **numbers[i-1] % 2 == 0** and if so, output numbers[i-1] but we wanted to demonstrate the continue statement. In the above code, if the number is not even, we execute the continue statement and skip the print statement, moving onto the next loop iteration.

While loops are great for running code when we don't know how many times the loop will need to run. For example, we often use while loops in games to constantly check for collisions, user interactions, updated movement and re-rendering graphics. This is because we want to keep the game code running as long as the player is still playing the game which could be a few seconds to a few hours or even longer. Sometimes, however, we want to run a loop a predefined number of times. We could simulate this using a while loop but there is a cleaner and easier way: using a **for loop**.

For loops have a defined **start** point, **end** point, and **iterator**. This means that they will always stop at some point and will never run infinitely. These factors make **for loops** and **arrays** couple very well; although we don't always have to pair them up. For example, if we have an array of values and want to output them, we could do so like this:

```
var items = [“axe”, “knife”, “rope”, “boots”];

for (var i = 0; i < items.length; i++) {
  console.log(items[i]);
}
```

Note the start point, end point, and iterator are respectively separated with **;**. We create a variable called **i** and run the loop as long as **i < items.length**, increasing its value by **1** with each iteration. Once **i >= items.length**, the loop exits.

We can set the start point, end point, and iterator to be whatever values we want. For example, we could reverse the order by doing this:

```
for (var i = items.length - 1; i >= 0; i--) {
  console.log(items[i]);
}
```

This runs backwards through the array. For loops can always be replaced by while loops if we want, but not necessarily vice versa. Otherwise, they work in almost the same way.

There is a variant on the for loop called the **forEach** loop. This **must** be coupled with an array and can be used to fetch each element and index. These loops implicitly start at the beginning, end at the end, and visit every member of an array without us having to define them. They work like this:

```
items.forEach(function(element, index) {
  console.log(element);
});
```

This will do the same thing as our first for loop. It's important to note that using forEach, we can modify the value of element within the function but it does not affect the original array elements.

Objects allow us to associate multiple values with a single variable. Unlike arrays that just store lists of values, objects use values to describe properties/state and functionality. We attach values and functions to **names/keys**, similar to **dictionaries** or **maps** in other languages. Essentially, objects are variables with multiple values that describe the state and functions that help to modify the state by changing the values.

There are a few ways to create objects, and the simplest form is to create a bunch of **key:value pairs. The values can be single variable values, arrays values, or function definitions. Let's say we wanted to create a game character object. A game character might have a name, health, x position, and a function to help move the character by changing the x position. We can do so like this:**

```
var gameCharacter = {
  name: "Nimish",
  xPosition: 2,
  health: 100,
  move: function(xAmount = 0) {
    this.xPosition += xAmount;
  }
};
```

Notice how we create it like a var (you can also use const or let) and we put the key-value pairs in the { }. Each key is separated from the value by the : so the first property is called name, and the value is "Nimish". We can think about the name/key as being the name of a variable that belongs to the gameCharacter object. The main benefit of doing this is that we don't have to create three separate variables and a function. We can store all of that information inside of the gameCharacter variables. Also note that we use the this keyword. We add that to refer to the xPosition that belongs to this object so any time we have a function that accesses a property of the current object, put this. in front of it.

To access the properties, we can use dot syntax or [ ] syntax. For example, we can access and change the variables like this:

```
var name = gameCharacter.name; // name is "Nimish"
name = gameCharacter['name'];
gameCharacter.health = 120; // Now the health of gameCharacter is 120
```

There's no real difference between the two but I'd recommend using dot syntax as we will need that later. We can run functions in a similar way:

```
gameCharacter.move(5); // gameCharacter.xPosition = 7
```

We can also add functions or properties to these objects on the fly. Let's say we want to add a health function and type variable and we've already created the gameCharacter object. We can do so like this:

```
gameCharacter.heal = function(amount) {
  this.health += amount;
```



```
};
gameCharacter.type = “Human”;
```

We can then access them in the same way as with any other variable. One big downside to creating objects like this is that if we want to create other gameCharacter objects, we have to rewrite the same code but assign it to a different variable. For example we would do this to create multiple gameCharacters:

```
var gameCharacter = {
  name: “Nimish”,
  xPosition: 2,
  health: 100,
  move: function(xAmount = 0) {
    this.xPosition += xAmount;
  }
};
var otherGameCharacter = {
  name: “Zenva”,
  xPosition: 5,
  health: 120,
  move: function(xAmount = 0) {
    this.xPosition += xAmount;
  }
};
```

This is generally not great practice because we are repeating code even though we’re assigning different values. gameCharacter and otherGameCharacter both have the same properties and functionality but have different values assigned. Also, if we added extra properties or functions to one of the two characters, the other character does not get access to those properties or functions. It’s not a good long term solution if we want to create multiple gameCharacters. It’s really only meant as a one time use and there is a better way of creating objects if we want multiple of them.

In this lesson, we're going to cover object **getters** and **setters**.

Before that, however, let's talk about the difference between assigning a value and copying a **reference**.

Consider the following code:

```
var i = 5;
var j = i;
i = 10;
```

Here, we're setting **j** to take the value of **i** (so j is equal to 5), and then we set a new value to i. Var **i** is now equal to **10**, but note how that does not change anything for **j** (which had already been assigned a value).

However, things change when we're dealing with **objects** (instead of simple numeric primitives such as in the example above).

Take a look at this piece of code here (very similar to the one we discussed in our previous lesson):

```
var gameCharacter = {
    name: "Nimish",
    xPos: 0,
    items: ["Knife", "Food"],
    move: function(x) {
        this.xPos += x;
    }
};
```

Now, if we were to assign our gameCharacter to a new variable, we'd be simply creating a reference to the gameCharacter object we created before:

```
var gc = gameCharacter
```

This is important to understand because now changing **gc's name** attribute will also change the name attribute of our **gameCharacter** object:

```
gc.name = "Joseph"
```

Both **gc** and **gameCharacter** variables will be affected by any changes made to either of them. Our gameCharacter's name is "Joseph" now too (and no longer "Nimish").

Getters and setters

Getters and setters help us with a clean way to organize our code, although their use is not mandatory.

Getters act like functions and we use them to return **attributes** of an object (or a modified version of them).

Let's add a new attribute to our **gameCharacter** to see an example of this:

```
var gameCharacter = {
    name: "Nimish",
    class: "Human",
    xPos: 0,
    items: ["Knife", "Food"],
    move: function(x) {
        this.xPos += x;
    }
};
```

Class will tell us the **classification** of our character in a game with lots of different classes of creatures, for instance.

Say we want to have a **title** for each character, which will be their name and class in a simple string.

In this case, we can use a **get** attribute (getter) to retrieve it:

```
var gameCharacter = {
    name: "Nimish",
    class: "Human",
    get title() {
        return this.name + " the " + this.class;
    },
    xPos: 0,
    items: ["Knife", "Food"],
    move: function(x) {
        this.xPos += x;
    }
};
```

The syntax is the **get** keyword then the **name** of your getter followed by parentheses **()** and brackets **{}**. Inside the computation you want to make is stored (here being a string interpolation).

Be careful when using the word **class** as it is actually the **keyword** for a class in Javascript, so make sure you're using "**this.**" in front of it.

```
gameCharacter.title //"Nimish the Human"
```

Note that we **call** it like a variable (without the need to put the parentheses at the end as we do with functions).

Setters, on the other hand, are used to **modify** an attribute within an object.

Let's say we want to set the health of our game character every time we change the value of

maximum health:

```
var gameCharacter = {
    name: "Nimish",
    class: "Human",
    health: 100,
    get title() {
        return this.name + " the " + this.class;
    },
    set maxHealth(h) {
        this.health = h;
    },
    xPos: 0,
    items: ["Knife", "Food"],
    move: function(x) {
        this.xPos += x;
    }
};

gameCharacter.maxHealth = 150; //health = 150
var characterHealth = gameCharacter.maxHealth //undefined
```

The syntax is pretty similar to the get property we've just covered and now we are modifying the value of health for our gameCharacter object.

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected ones.

If we want to create multiple instances of an **object**, it helps if we can define an abstract version that specifies the properties and behaviours that the object should have. We can do that with a **constructor function**. We can pass in **any** values that need to be assigned as parameters and then use the function to create new instances of objects by passing in the required values. If we wanted to create the same GameCharacter as above, we can do something like this:

```
function GameCharacter(name, xPos, health) {
  this.name = name;
  this.xPosition = xPos;
  this.health = health;
  this.move = function(xAmount = 0) {
    this.xPosition += xAmount;
  }
}
```

This isn't actually creating an object; this is just building a template of the properties and behaviours that a GameCharacter object should have. To actually **create** some GameCharacter objects, we do this:

```
var gc = new GameCharacter("Nimish", 2, 100);
```

This will use the values that we're taking in and assign them to the appropriate variables/functions. We can also take in functions as parameters if we want. The great thing is that once the GameCharacter is defined with the function, we can create as many instances as we want with just a single line of code each rather than redefining the GameCharacter each time. We use this way of creating objects when we will need multiple objects with the same functionality. Also note the use of the "**new**" keyword. Just like before, we can access or modify values/functions using dot syntax:

```
var name = gc.name;
gc.move(5);
```

Also note how we are calling on the **instance** of GameCharacter (gc) rather than the GameCharacter function.

The following instructions have been updated, and differs from the video:

You can add a **new property** to your >object function as follows:

```
function GameCharacter(name, xPos, health) {
  this.name = name;
  this.xPosition = xPos;
  this.health = health;
  this.move = function(xAmount = 0) {
    this.xPosition += xAmount;
  }
}
```



```
var gc = new GameCharacter("Nimish", 2, 100);
gc.yPosition = 5;
```

However, note that **yPosition only exists for the gc instance** specifically because it's been set outside of the scope of the object function.

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

Sometimes, we want to change the definition of an object, meaning we want to add extra variables or functions. We saw that we can do exactly that with the **key-value** pair style of creating an object and we can do something very similar with **constructor functions**. For example, let's reuse the same GameCharacter definition but let's add a "type" variable. We can do that like this:

```
var gc1 = new GameCharacter("Nimish", 2, 100);
gc1.type = "Human";
```

That's fine, we just added another variable to **gc1** and can access or modify it just like before. However, if we created another GameCharacter, that GameCharacter does not know about the "type" variable.

The following instructions have been updated, and differ from the video:

```
var gc2 = new GameCharacter("Zenva", 5, 120);
var type = gc2.type; //undefined because gc2 has no 'type' attribute
```

That's because we have only assigned the new variable to an **instance** of GameCharacter. If we want all GameCharacters to have the type variable, we have to use **prototyping**.

Prototyping forms the basis of inheritance, an object oriented language which we will talk about in greater detail later. Essentially, this is a way to add extra variables or functions to provide a more specific and complete definition of an object. To add the "type" variable to all GameCharacters, we do this:

```
GameCharacter.prototype.type = "Human";
```

That adds the type attribute and assigns the value of "Human" to all GameCharacters, even if they were created before adding that line of code. So now gc2 has a type of "Human" and so would any other GameCharacters created before and any that will be created in the future. We can do the same thing with functions. For example, if we wanted to add a heal function, we could do so like this:

```
GameCharacter.prototype.heal = function(amount) {
  this.health += amount;
}
```

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

Almost all object oriented languages use a standard way of creating object templates and they do this through **class structures**. Javascript recently incorporated this syntax into its library but really, the class format is just syntactic sugar (syntax designed to make things easier) over the top of constructor functions. Within a constructor, we defined the variables, any default values, and functions that an object should have. With a class, we separate out the declarations of variables and functions but it accomplishes the same purpose. To create the GameCharacter class, we would do this:

```
class GameCharacter {

  constructor(name, xPos, health) {
    this.name = name;
    this.xPosition = xPos;
    this.health = health;
  }

  move(xPos) {
    this.xPosition += xPos;
  }

}
```

The only real difference here is that we moved the **move** function implementation outside of the constructor and have encompassed the entire structure in the **{ }**. Also, note that we can **drop** the function keyword when creating functions and that the constructor has taken on the constructor keyword. We can create instances in the same way as before.

The following code has been updated, and differs from the video:

```
var gc = new GameCharacter(“Nimish”, 0, 100);
```

The usage is exactly the same as when we create an object template with just a constructor function, so this is just different syntax for the same functionality. It should be noted that some online compilers like Rextester and JSBin don’t support class definitions in this way.

Inheritance

Given this definition of an object, the question might come to mind: can we add extra variables and functions like we do when prototyping? The answer is yes! In fact, we can do the exact same thing as with regular prototyping by doing the **className.prototype.propertyName = value**.

There is another way of accomplishing basically the same task, however, and that’s done through **inheritance**. This is a way in which one class inherits everything from another class (has the same variables and functions) but can also define additional variables and functions. In this case, the inheriting class is the **subclass** and the class being inherited from is the **superclass**. Although this is the only way to do this in object oriented languages, at the end of the day, this is just more syntactic sugar for prototyping in Javascript.

Let's say we want to create a HumanCharacter class with all the same variables and functions, but we also want a "type" variable and a "heal" function. We can extend (inherit from) the superclass GameCharacter by doing this (providing we use the previous GameCharacter class):

```
class HumanCharacter extends GameCharacter {

  constructor(name, xPos, health) {
    super(name, xPos, health);
    this.type = "Human";
  }

  heal(amount) {
    this.health += amount;
  }

}
```

By extending GameCharacter, HumanCharacter automatically gets the name, xPos, and health variables and the move function. We're providing an additional variable (type) and an additional function (heal) that the superclass (GameCharacter) does not know about. So we can do this:

```
var hc = new HumanCharacter("Nimish", 2, 100);
hc.heal(10);
```

But not this:

```
var gc = new GameCharacter("Zenva", 5, 120);
gc.heal(10);
```

Note how in the constructor, we are still taking in a name, xPos, and health but we are calling super(name, xPos, health). By calling super(), we are using the **superclass constructor** to set up the name, xPos, and health variables. It already knows what to do, so we let it do all the heavy lifting as there is no point in repeating code. If we are using a superclass constructor, we have to make sure we are passing all of the appropriate values to the superclass constructor.

Language Basics Summary

That is it for the Javascript language basics. You know enough to get started on the game development portion of the course. We will first cover and intro to the components needed to build games in Javascript and then move on to develop a playable minigame.

If you have followed the lesson notes (the summaries for each video lesson of the course) then you're already up to date with the code. Otherwise, take a look at the **fixes** done to the course's code below.

As there have been changes in the code from the version presented on the video lessons, here's the **updated code** of this first part of the course:

```
// Variables
var maxHealth = 100;
var currentHealth = 50.5;
currentHealth = maxHealth;
//maxHealth = currentHealth;
var isGameOver = false;
isGameOver = true;

// Const, let, var
const maxHealth = 100;
var currentHealth = 50;
if (true) {
   // var isGameOver = false;
   let isNotGameOver = true;
}
// console.log(isGameOver) // false
console.logisNotGameOver) // error

// Strings
var characterName = "Nimish";
characterName = "Zenva"
var age = 26;
let message = `I am ${age} years old.`; // message = "I am 26 years old."
let ageMessage = "Hi, my name is " + characterName + " and my age is " + age; // age
Message = "I am 26 years old."

let length = characterName.length;
let upperName = characterName.toUpperCase();
characterName = characterName.toLowerCase();

let zen = characterName.slice(0, 3);

var level = 1;
var title = `Level ${level}`;

// Operators
var health = 50;

// + - * / % **
health = health + 10; // 60
health = health % 50; // 10
// health % 2 == 0
health = health ** 2; // 100

// +=, -=, *=, /=
health -= 20; // health = health - 20

// ++ --
```

```
health++; // 81

// health = (health * 2) + (health / 5);

health++;
health = health + 1;
health += 1;

// > >= < <= == !=
var number1 = 5;
var number2 = 10;

var result = number1 > number2; // false
result = number1 != number2; // true

var string1 = "hello"
var string2 = "world"
result = string1 == string2; // false

// !, ||, &&
result = !result; // true
result = number1 > number2 && string1 == string2; // false
number2 = number1;
result = number1 == number2 || string1 == string2; // true

// ? :
result = number1 > number2 ? number1 : number2;
var numberOfLives = 3;

var isGameOver = !(numberOfLives > 0);

// Arrays
var inventory = ["shirt", "axe", "bread"];
// inventory = ["water", "pants"];

let shirt = inventory[0];
inventory[2] = "cheese";

var length = inventory.length // 3
length = inventory[0].length; // 5

inventory.push("water"); // ["shirt", "axe", "cheese", "water"]
var water = inventory.pop();

// 2D Arrays
var levels = [
   [1.1, 1.2, 1.3],
   [2.1, 2.2, 2.3, 2.4],
   [3.1, 3.2],
];

var firstWorld = levels[0]; // [1.1, 1.2, 1.3]
var firstLevel = levels[0][1]; // 1.2
// levels[0][1] = 1.4
```

```
levels[1].pop()
levels[2].push(3.3)

// Functions
var currentHealth = 50
var healAmount = 10
function heal() {
   currentHealth += healAmount;
   // function nested() {

   // }
}
heal(); // currentHealth = 60
// var func = function heal() {
//    currentHealth += healAmount;
// }

// Function Parameters
var currentHealth = 50
function heal(healAmount = 10) {
   currentHealth += healAmount;
}
heal(); // currentHealth = 60
heal(40); // currentHealth = 100

// Function return values
var maxHealth = 100
var currentHealth = 50
function heal(healAmount = 10) {
   var newHealth = currentHealth + healAmount;
   currentHealth = newHealth > 100 ? maxHealth : newHealth;
   return calculatePercent(currentHealth, maxHealth);
}
function calculatePercent(dividend, divisor) {
   return (dividend / divisor) * 100;
}
var result = heal(); // currentHealth = 60, result = 60
result = heal(50); // currentHealth = 100, result = 100

// If statements
var keyPressed = " ";
var xPos = 0;

if (keyPressed == "l") {
   xPos -= 1;
} else if (keyPressed == "r") {
   xPos += 1;
} else {
   xPos = 0;
}

var keyPressed = "l";
var xPos = 0;
let endPos = 5;
let startPos = 0;
```

```
// if (keyPressed == "r" && xPos < endPos) {
//    xPos += 1;
// }

if (keyPressed == "r") {
   if (xPos < endPos) {
    xPos += 1;
   }
} else if (keyPressed == "l") {
   if (xPos > startPos) {
    xPos -= 1;
   }
} else {
   xPos = 0;
}

// var someNum = 5;

// if (someNum > 4) {
//    xPos += 1;
// }
// if (someNum > 2) {
//    xPos -= 1;
// }

// While Loops
let endPos = 10;
var xPos = 0;
var enemyPos = 4;
var isGameOver = false

// while(pos < endPos) {
//    pos++;
// }

// Control Statements
// while(!isGameOver) {
//    xPos++;
//    if (xPos >= endPos || xPos == enemyPos) {
//     isGameOver = true;
//    }
// }

// break
// while(xPos < endPos) {
//    xPos++;
//    if (xPos == enemyPos) {
//     break;
//    }
// }

// while(xPos < endPos) {
//    if (xPos % 2 == 1) {
//     xPos += 2;
```

```
//     continue;
//    }
//    xPos++;
//    if (xPos == enemyPos) {
//     break;
//    }
// }

function movePlayer() {
   while(xPos < endPos) {
    xPos++;
    if (xPos == enemyPos) {
     return;
    }
   }
}

// For loops
var items = ["Axe", "Shirt", "Knife"]
var finalString = "In my inventory, I have "
// for(var i = 0; i < items.length; i++) {
//    var itemName = items[i];
//    finalString += itemName + " ";
// }
items.forEach(function(element) {
   finalString += element + " ";
});

// Objects
var gameCharacter = {
   name: "Nimish",
   xPos: 0,
   items: ["Knife", "Food"],
   move: function(x) {
    this.xPos += x;
   }
};
var name = gameCharacter.name;
// var name = gameCharacter['name'];
gameCharacter.items = ["Axe", "Bread"];
gameCharacter.move(5);
var x = gameCharacter.xPos;
gameCharacter.yPos = 0;
gameCharacter.move = function(x,y) {
   this.xPos += x;
   this.yPos += y;
}

// Getters and Setters
var gameCharacter = {
   name: "Nimish",
   class: "Human",
   health: 100,
   get title() {
    return this.name + " the " + this.class;
```

```
    },
    set maxHealth(h) {
     this.health = h;
    }
    xPos: 0,
    items: ["Knife", "Food"],
    move: function(x) {
     this.xPos += x;
    }
};
gameCharacter.title; // "Nimish the Human"
gameCharacter.maxHealth = 150; // health = 150
var health = gameCharacter.maxHealth; // undefined
// var i = 5
// var j = i;
// i = 10; // i = 10, j = 5
// var gc = gameCharacter
// gc.name = "afsdfasdf" // gc.name = "afsdfasdf", gameCharacter.name = "afsdfasdf"

// Object Constructors
function gameCharacter(name, xPos, health) {
    this.name = name;
    this.xPos = xPos;
    this.health = health;
    this.move = function(x) {
     this.xPos += x;
    }
    this.class = "Human"
}
var gc1 = new gameCharacter("Nimish", 0, 100);
var name = gc1.name;
gc1.health = 150;
gc1.move(10);
var gc2 = new gameCharacter("Zenva", 5, 150);
// var gc1 = {
//    name: "Nimish",
//    xPos: 0,
//    health: 100
// }
// gc1.yPos = 5;
// var gc2 = {
//    name: "Zenva",
//    xPos: 5,
//    health: 150
// }

// Prototyping
function gameCharacter(name, xPos, health) {
    this.name = name;
    this.xPos = xPos;
    this.health = health;
    this.move = function(x) {
     this.xPos += x;
    }
}
```

```
var gc1 = new gameCharacter("Nimish", 0, 100);
gc1.yPos = 5; // 5
var gc2 = new gameCharacter("Zenva", 5, 150);
// gc2.yPos; // //undefined because gc2 has no 'yPos' attribute
gameCharacter.prototype.class = "Human";
gc1.class = "afsdfasdf";
gc2.class; // Human
var heal = function(amount) {
   this.health += amount;
}
gameCharacter.prototype.heal = heal;
gc1.heal(5);
gc2.heal(10);

 Classes
class GameCharacter {
   constructor(name, xPos, health) {
    this.name = name;
    this.xPos = xPos;
    this.health = health;
   }
   move(x) {
    this.xPos += x;
   }
}
class HumanCharacter extends GameCharacter {
   constructor(name, xPos, health) {
    super(name, xPos, health);
    this.classification = "Human";
   }
}
var gc1 = new GameCharacter('Nimish', 0, 100);
gc1.move(5);
gc1.name;
var hc1 = new HumanCharacter('Zenva', 5, 150);
```

Now that we've completed the Javascript basics, you know enough to get started on the game development portion of the course. We will first cover an intro to the components needed to build games in Javascript and then move on to develop your very first playable minigame.

Game Basics

By now we should have a pretty good grasp of the language and how to use it effectively. From now on, we will learn the pieces needed to create simple games using Javascript. We will start with the various components used in games, and then learn how to add logic and movement. We will finish with player interaction and collision handling. By the end, we will have a game built from start to finish interactively and with good looking graphics.

Let's get our hands dirty with some HTML and CSS as well as Javascript code. An online compiler won't cut it as we need to be able to display HTML output. You will need some kind of text editor (I use **Sublime Text** although even Notepad would work) and a **browser** to run the code. To run an HTML file in the browser, simply right click on the file and open with the browser of your choice.

Getting started

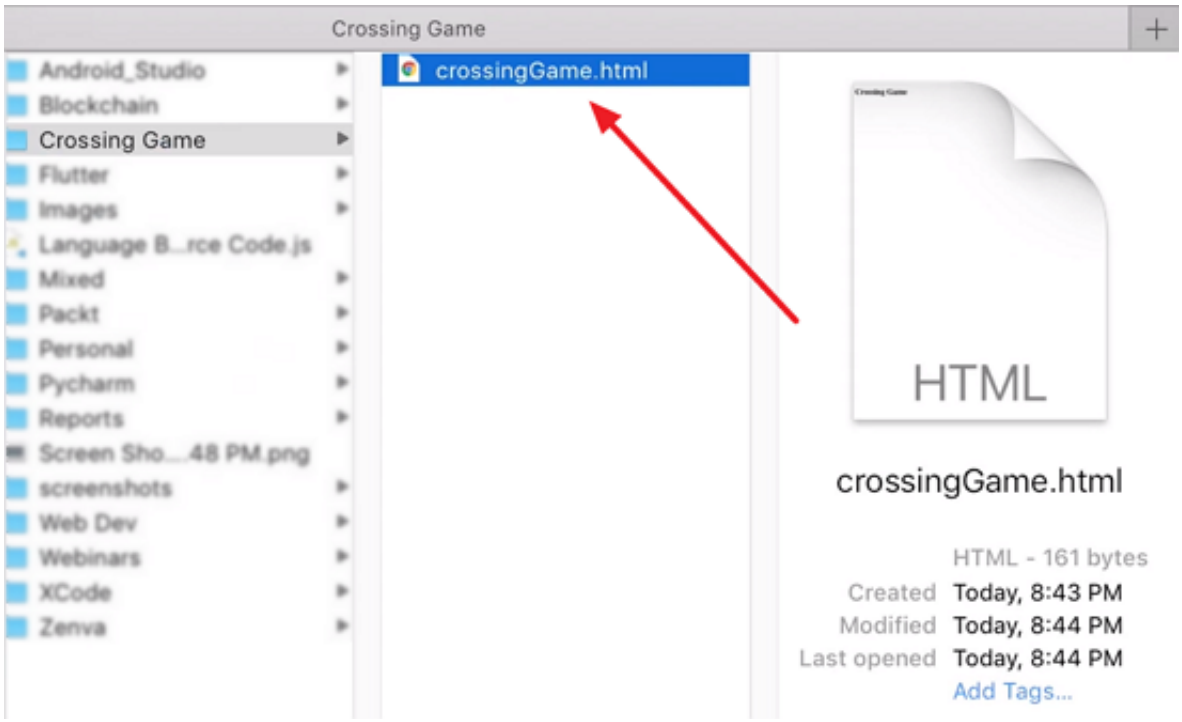
In your text editor, open up a **new** file. Let's name it "**crossingGame.html**".

Now, let's write a basic HTML **setup** to it:

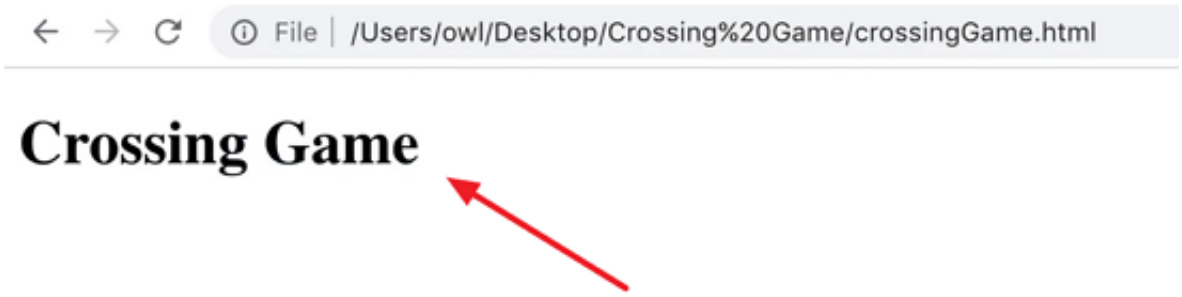
```
Crossing Game
```

We basically just have the **title** and the **heading** for our HTML page. The javascript part will come later on.

Go to the folder where you saved our HTML file and open it:



You should now see the **title** of our game:



In the next lesson, we'll start working with our **canvas**.

The first thing we need to do is to create a game **canvas**. This represents the game board or the **screen** on which we play the game. For our purposes, we will use a simple colored rectangle as a background and draw everything on top of it. The good news is that HTML already has a canvas **tag** that contains special properties.

We start by creating the canvas **inside** the **body** tag. We give it an **id**, a **width** and a **height** as follows:

Crossing Game

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');
```

We find our **canvas** element by **Id** to be able to get the **context** of the canvas and specify that we're going to be drawing only in **2D** (as it's not a 3-dimensional game).

This concludes the setup of our canvas. Feel free to alter the dimensions of the canvas, making it bigger or smaller as preferred.

Once you determined the **height** and **width** you want for your canvas, add them to the **script** as well:

The following instructions have been updated, and differ from the video:

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

const screenWidth = 1000;
const screenHeight = 500;
```

Now we're moving to the start of entering the code for the logic of our game loop. The **game loop** checks for updates in our game, such as characters' **movements** and **collisions**. It is executed many times per second depending on the **frames per second** (FPS) of the device running it.

For now, we're just **declaring** it and allowing the loop to be simulated through the call of **requestAnimationFrame** which will keep calling for another **iteration** of our step function. So, add the following code to your **script** tag below the screen width and height variables:

```
var step = function() {
    window.requestAnimationFrame(step);
}

step();
```

Don’t forget to **call** our **step** function after its declaration to **initiate** the game loop.

In the next lesson, we’ll start drawing onto our canvas.

Now that we have the **context** for our canvas, we can **draw** rectangles and other shapes on it.

Let's first create a **game character class** (inside the **script tag**) as follows:

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

let screenWidth = 1000;
let screenHeight = 500;

class GameCharacter {
    constructor (x, y, width, height, color){
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.color = color;
    }
}

var step = function() {
    window.requestAnimationFrame(step);
}

step();
```

In our constructor, we receive **x** and **y** positions, **width**, **height**, and **color** as parameters and we **save** them in our class.

Now, we're going to **instantiate** our first game character **instance**, which is going to be a simple **rectangle**. To do that, add the following line to your code (below the game character class):

```
var rectangle = new GameCharacter (50, 50, 50, 50, "rgb(0, 0, 255)");
```

Note that we're passing the **color** as an **RGB string** in this case. We're passing **0** for the value of red, **0** for green, and **255** for blue.

Let's move on to the implementation of the **draw** function:

```
var draw = function() {
    ctx.clearRect(0, 0, screenWidth, screenHeight);

    ctx.fillStyle = rectangle.color;
    ctx.fillRect(rectangle.x, rectangle.y, rectangle.width, rectangle.height);
}
```

The first thing we do on a draw function is **clear** the entire **canvas**. We do that by using our canvas

context to call the **clearRect** function. This way we update our canvas in real-time, redrawing everything again on top of it at each new iteration.

Our next goal is to provide the **fill** style for the **context** (we're just coloring it here). After that, we pass the **shape** to be filled (in this case, we use the **attributes** of the **rectangle** we created before, which is the object we want to draw).

We then **call** our **draw** function in our **step function**:

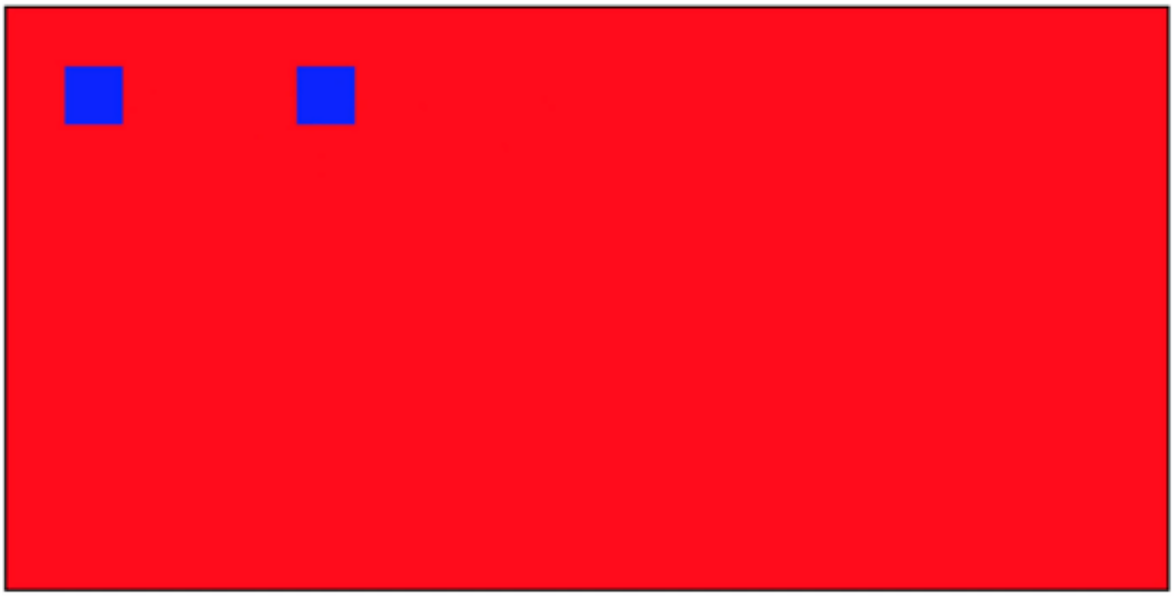
```
var step = function() {
    draw();
    window.requestAnimationFrame(step);
}
```

And there we have our rectangle (more precisely, a square):

If we want to add yet another rectangle, we simply add a similar block of code:

```
ctx.fillStyle = rectangle.color;
ctx.fillRect(rectangle.x + 200, rectangle.y, rectangle.width, rectangle.height);
```

Here, we're spacing them by **200** on the **x-axis**:



As an **exercise**, have **three rectangles** evenly distributed across the canvas (try varying their positions). In our final game, we'll have a rectangle for our **player** and these three other rectangles will be the **enemies**. The idea is that the enemies will move up and down while the player will move horizontally, aiming to reach the other side of the canvas. We'll see the **solution** for this exercise at the beginning of the next video!

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

Let's start by checking the changes needed to implement the **exercise** proposed in our previous lesson.

For our **enemies**, we replace our rectangle with an **array** of **three rectangles** spaced through our canvas:

```
var enemies = [
    new GameCharacter (200, 50, width, width, "rgb(0, 0, 255)"),
    new GameCharacter (450, screenHeight - 100, width, width, "rgb(0, 0, 255)"),
    new GameCharacter (700, 50, width, width, "rgb(0, 0, 255)")
];
```

For this, you need to set let width = 50; **above** the declaration of our **GameCharacter** class.

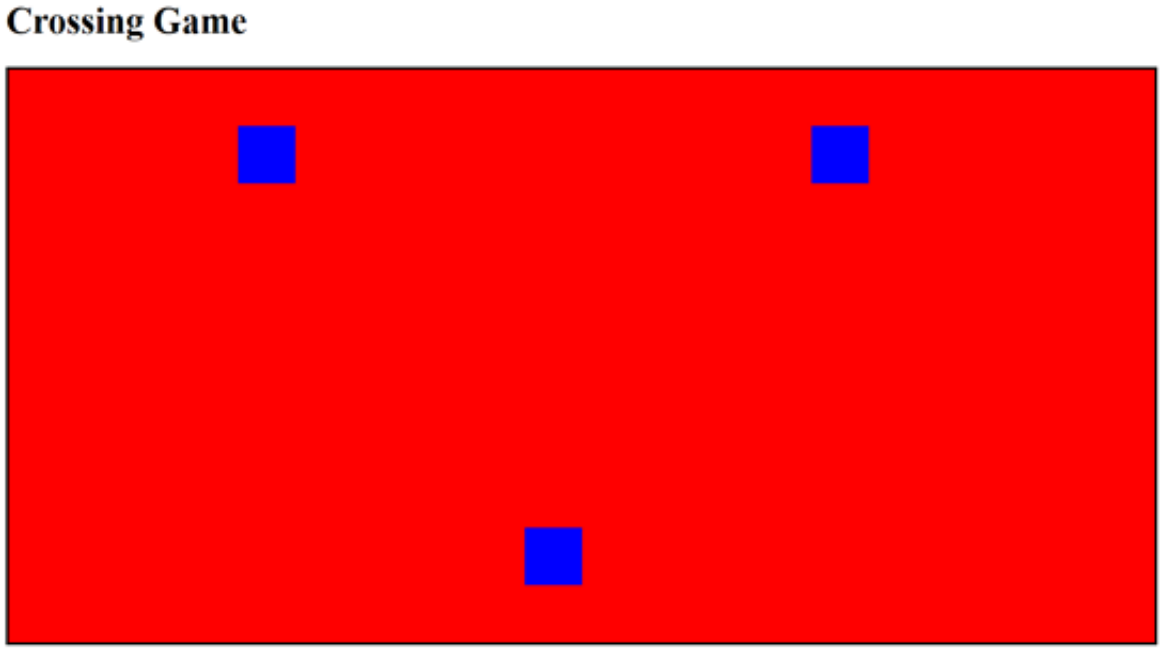
You can see that **two** of our enemies start from the **top** and the **middle** one from the **bottom** (at screenHeight – 100 on y). **All** enemies have the same size of **50** for both width and height.

Next, in our **draw function**, instead of calling fillStyle and fillRect for our only rectangle as before, we **iterate** through our enemies array to draw each of them in turn:

```
var draw = function() {
    ctx.clearRect(0, 0, screenWidth, screenHeight);

    enemies.forEach(function(element){
        ctx.fillStyle = element.color;
        ctx.fillRect(element.x, element.y, element.width, element.height);
    });
}
```

Our canvas looks like this now:



If your solution is different, it's ok as long as you have the three enemies drawn on your canvas.

Moving the enemies on the canvas

Let's move only **one** of the rectangles first. We'll start with the **leftmost** rectangle on the canvas and we'll bring it down on the y-axis by **increasing** its **y** position. To do that, we basically increase its y by 1 (or some other given amount) and redraw the rectangle over and over again.

In the **GameCharacter** class, **add** a **move function**:

```
move(xAmount, yAmount){
    this.x += xAmount;
    this.y += yAmount;
}
```

We also need an **update** function that we're going to use to update the **movements** of everything on the canvas.

Add the update function **below** the draw function as follows:

```
var update = function() {
    enemies[0].move(0,2);
}
```

We're taking the **first** rectangle of our enemies array and making it move only on the **y-axis** by **2** units at a time.

Now we call our **update** function from within our **step** function, before we call **draw()**:

```
var step = function() {
```



```
    update();
    draw();
    window.requestAnimationFrame(step);
}
```

If you open the page of our game, you'll see the rectangle is going down as expected but it's **going off** the canvas and not resurfacing again. We can fix that by making the rectangle **bounce** once it reaches the boundary of our canvas and go up to the top again.

Let's first add a parameter "**speed**" for our **GameCharacter** class **constructor** as this allows us to vary the speed of each of our rectangles (it's going to be useful later on):

```
constructor (x, y, width, height, color, speed){
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.color = color;
    this.speed = speed;
}
```

Our **enemies** array will also change, as now we need to set a **speed** for each of them. Let's set their speed to **2** for the time being:

```
var enemies = [
    new GameCharacter (200, 50, width, width, "rgb(0, 0, 255)", 2),
    new GameCharacter (450, screenHeight - 100, width, width, "rgb(0, 0, 255)", 2),
    new GameCharacter (700, 50, width, width, "rgb(0, 0, 255)", 2)
];
```

Let's now **split** our movement logic into **moving horizontally** and **moving vertically** as our player will move horizontally and the enemies will move vertically. For that, replace the **move function** on the **GameCharacter** class as follows.

The following code has been updated, and differs from the video:

```
moveVertically() {
    this.y += this.speed;
}

moveHorizontally() {
    this.x += this.speed;
}
```

Note that we **do not** need **xAmount** and **yAmount** parameters anymore as we're always **increasing** the position of the rectangles by **their speed** (in this case, 2 units at a time).

To make the rectangle **bounce** once it reaches close to the bottom of the canvas we can add an **if statement** that will **invert** the **speed** so the rectangle will start going **up** if it reaches **lower** than the screenHeight – 100:

```
moveVertically() {
    if (this.y > screenHeight - 100) {
        this.speed = -this.speed;
    }
    this.y += this.speed;
}
```

We'll update the moveHorizontally function further on, so don't worry about it for now.

Lastly, we need to change the **call** to our now called **moveVertically** function in the **update** function:

```
var update = function() {
    enemies[0].moveVertically();
}
```

Reloading our game page, we can see that our logic **works** as the rectangle is now bouncing back at the bottom.

As a **challenge** for the next lesson, try to implement the bouncing of the rectangle **at the top** as well, so it'll only move **inside** our canvas screen area instead of moving out of it.

Let's start by looking at a **solution** to the **challenge** of our last lesson:

```
moveVertically() {
    if (this.y > screenHeight - 100 || this.y         this.speed = -this.speed;
    }
    this.y += this.speed;
}
```

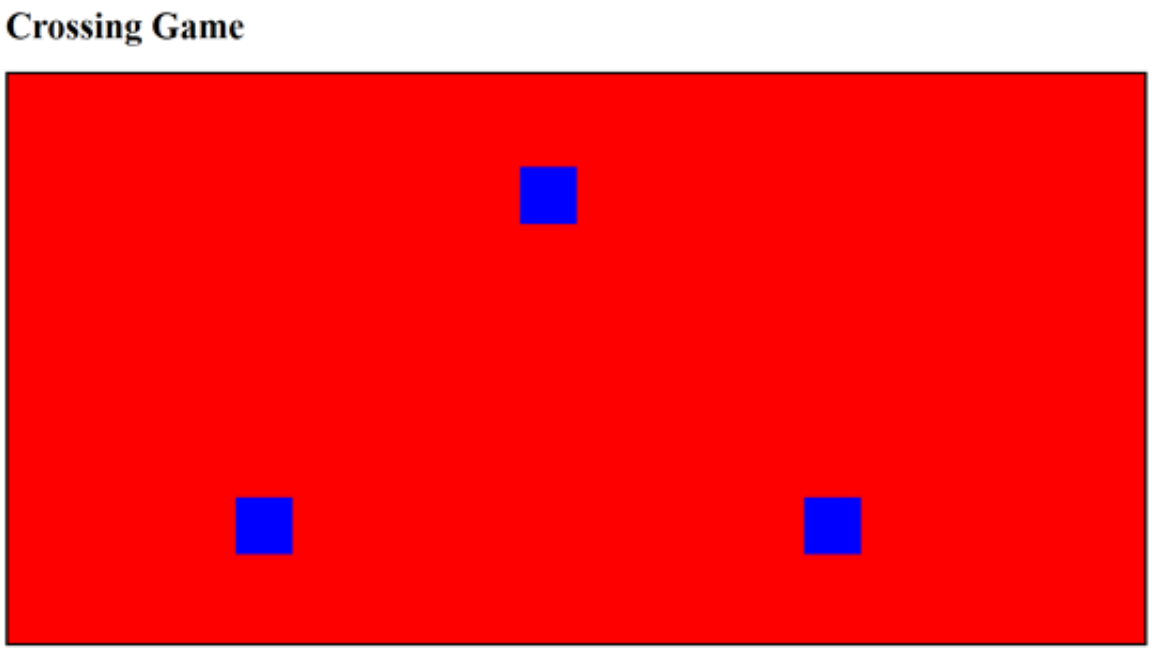
We basically just need to add an extra check to see if **y** is smaller than **50** (for instance), meaning that the rectangle is reaching the top of our canvas, and if that is the case, then we **invert** the speed again so that it'll start going **down** again.

Moving the enemies

Now, we need to apply the same logic to the other rectangles in our canvas.

```
var update = function() {
    enemies.forEach(function(element) {
        element.moveVertically();
    });
}
```

And if we open our page, it sure is working fine:



For a change, let's have the **first** rectangle start from the **middle**, and let's also make each of the rectangles have a **different speed**. To make that happen, we need only to update our enemies array to this:

```
var enemies = [
    new GameCharacter (200, 225, width, width, "rgb(0, 0, 255)", 2),
```



```
    new GameCharacter (450, screenHeight - 100, width, width, "rgb(0, 0, 255)", 3),
    new GameCharacter (700, 50, width, width, "rgb(0, 0, 255)", 4)
];
```

You can see that the first rectangle starts at the very middle of our canvas height (at position **225**) and that the second rectangle gained a speed of **3** and the last one of **4**.

If you **refresh** the page, you're going to notice that they are moving quite a lot**faster** than before, thus being more challenging to pass through them in our game.

Creating the player

Let's move on to creating our **player** character now.

We can use our **GameCharacter** class all the same, the only difference is that the player will move **horizontally** instead of **vertically** as the enemies do.

You can create the instance of the **player** below the enemies array with this line here:

```
var player = new GameCharacter(50, 225, width, width, "rgb(0, 255, 0)", 2);
```

It's very similar to how we created each of the enemies, but now we're setting its color to **green** so we can differentiate the player from the enemies.

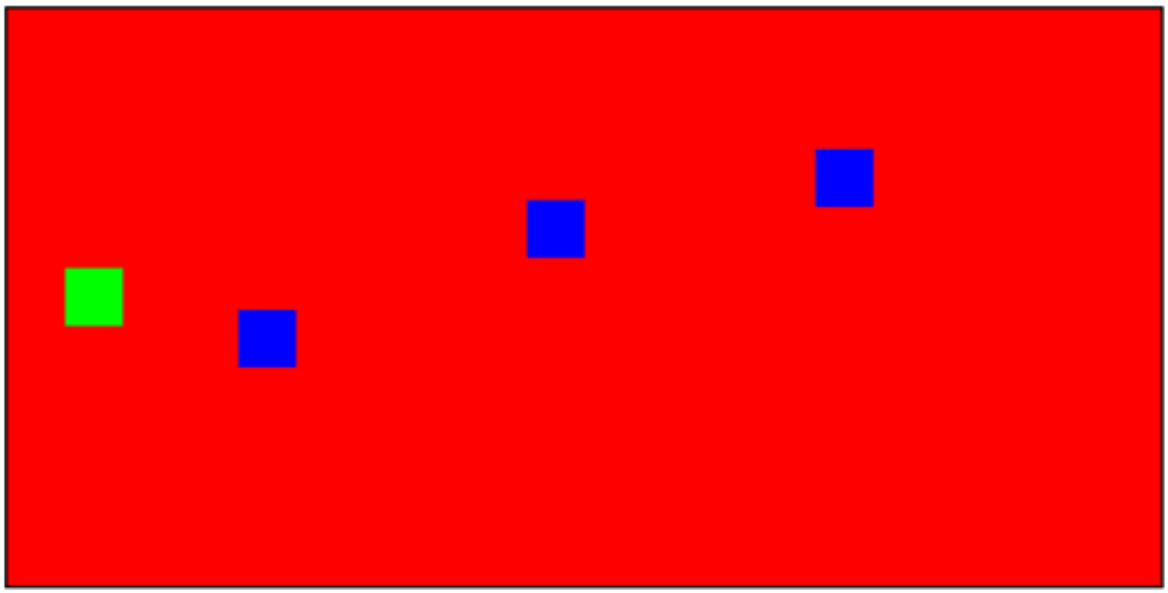
Next, we have to add the **player** to our **draw function** that now looks as follows:

```
var draw = function() {
    ctx.clearRect(0, 0, screenWidth, screenHeight);

    ctx.fillStyle = player.color;
    ctx.fillRect(player.x, player.y, player.width, player.height);

    enemies.forEach(function(element){
        ctx.fillStyle = element.color;
        ctx.fillRect(element.x, element.y, element.width, element.height);
    });
}
```

And there is our **player**:



In the next lesson, we'll make the player move horizontally across the canvas.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

In this lesson, we're going to add **events listeners** so that we can listen to events such as **mouse clicks** or **button presses**(important to our player movement implementation).

The player will be moving **right** with positive speed, **left** with a negative speed, or will stay in place with **zero** speed.

We're going to add a **maxSpeed** attribute in our **GameCharacter** constructor:

```
constructor (x, y, width, height, color, speed){
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.color = color;
    this.speed = speed;
    this.maxSpeed = 4;
}
```

We're setting it to **4** for now.

Adding event listeners

The next step is to add the **event listener** for when the user presses down on a key. The listener detects which key was pressed, and it's up to us to determine what to do as a **response** to that.

Add the **onkeydown** listener above the **draw function** in your code:

```
document.onkeydown = function(event) {
}
```

The **event** parameter has the info on which key was pressed down. We will treat it with a **switch statement** as follows.

The following code has been updated, and differs from the video:

```
document.onkeydown = function(event) {
    switch(event.key) {
        case 'ArrowRight':
            player.speed = player.maxSpeed;
            break;
    }
}
```

Note that *event.keyCode* is deprecated and has been replaced by *event.key*.

The **event.key** is going to get the corresponding key code to the **key** that the user pressed down.

Each key on the keyboard has its own separate code. We can search online to check the codes and their corresponding keys. Here, we're checking if the key pressed was "**ArrowRight**" and, if so, we set the player's speed to be equal to **maxSpeed**.

We also need to **update** the player's movement so that we'll see the green rectangle moving across the canvas. We do that on the update function by calling **moveHorizontally** for our player:

```
var update = function() {
    player.moveHorizontally();

    enemies.forEach(function(element) {
        element.moveVertically();
    });
}
```

If we refresh now, we'll notice that the player's rectangle is moving by itself and disappearing at the other side of the canvas. That is because we set its **initial speed to 2** when we instantiated it. When the **update** function is called it then calls **player.moveHorizontally** that we just added and sets the player into movement.

We actually want the player to start off **stationary**. For that, we need to changethe **speed** attribute of the player to **0** when we're creating it as follows:

```
var player = new GameCharacter(50, 225, width, width, "rgb(0, 255, 0)", 0);
```

This way, when the update is called and it tries to move the player by its speed, it's going to remain on the same spot because it's speed is now set to **0**.

Refreshing the page again, we have the player starting still and then moving when we **press** the **right arrow down**. Nevertheless, it starts off and ends up disappearing off the screen just the same.

To fix that, we need to add an **onkeyup** event listener to our code as well to make the player **stop** moving once we **release** the right arrow key:

```
document.onkeyup = function(event) {
    player.speed = 0;
}
```

So we now have implemented both **onkeydown** and **onkeyup**, the two event listeners needed to control the player's **movement** based on whether the user is pressing down on the arrow right key or releasing it. If we refresh the page once more, we see that it's working properly now.

As a **challenge**, try figuring out how to make the player **move left** because at the moment we're only moving to the right. You need basically to identify the key pressed down and then **inverting** the direction. We'll see a solution to this in the next lesson.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

In this lesson, we're going to implement the detection of **collisions** between our objects and **responses** to such collisions.

A collision occurs when there's an **overlap** on both **x and y axes** on some point of the involved objects. Every time a movement is updated on our canvas, we're going to check if there's an overlap between any of our objects (in our case, player, enemies, and goal). If so, we'll **verify** whether the user collided with the **goal** (for a **winning** logic) or with an **enemy** (**end game** logic).

Let us first quickly see the **solution** to the challenge of our last lesson.

The following code has been updated, and differs from the video:

```
document.onkeydown = function(event) {
    switch(event.key) {
        case 'ArrowRight':
            player.speed = player.maxSpeed;
            break;
        case 'ArrowLeft':
            player.speed = -player.maxSpeed;
            break;
    }
}
```

Basically, to move to the **left** as well, we're simply checking if the pressed key is left and then we **reverse** the direction of the movement.

Note that event.keyCode is deprecated and has been replaced by the event.key property.

Implementing collisions

Here we have the classic implementation of collision between rectangles.

Add the following code **above** the **draw function**.

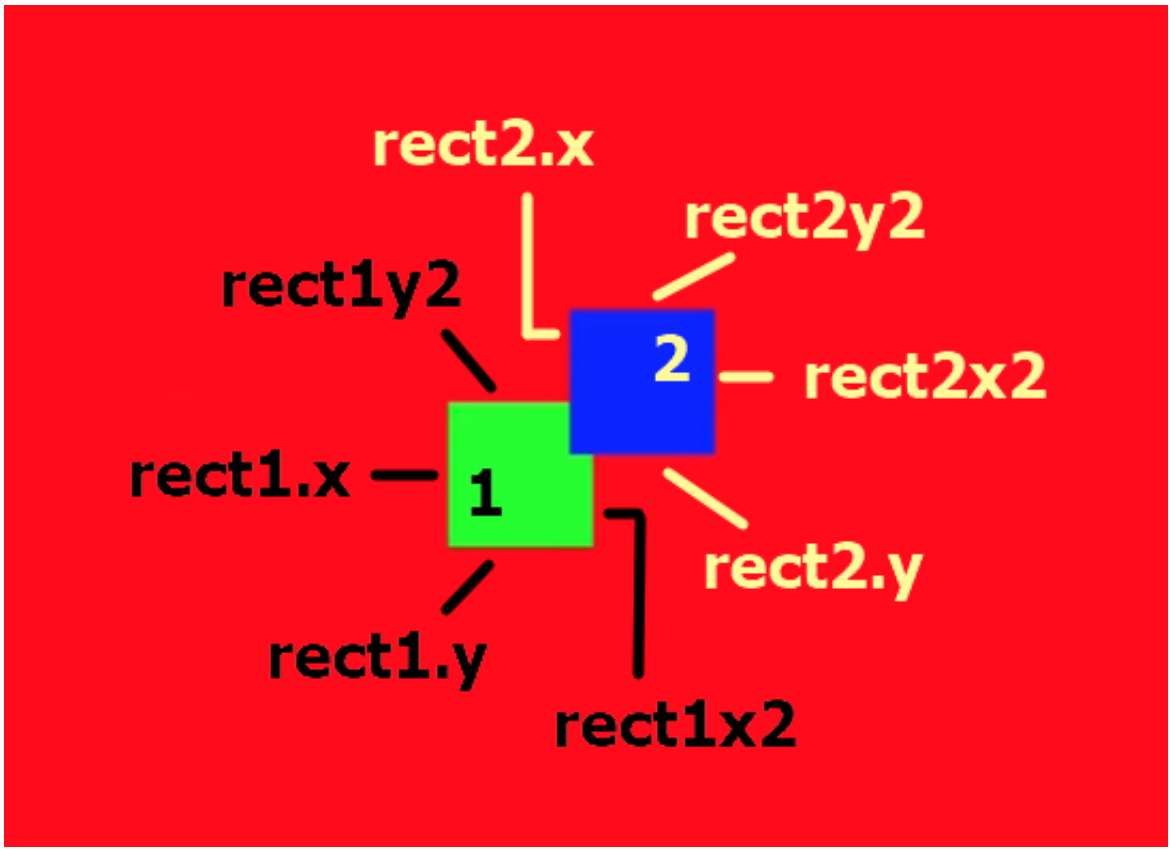
The following code has been updated, and differs from the video:

```
var checkCollisions = function(rect1, rect2) {
    let rect1x2 = rect1.x + rect1.width;
    let rect2x2 = rect2.x + rect2.width;
    let rect1y2 = rect1.y + rect1.height;
    let rect2y2 = rect2.y + rect2.height;

    return rect1.x  rect2.x && rect1.y  rect2.y;
}
```

The order of the rectangles here does not matter as the formula considers collisions from **rect1** coming from both **left** or **right** and **up** or **down**, and so all possibilities of collision are already included.

This code works by ensuring there is **no gap** between any of the **4 sides** of the rectangles (any gap means a collision does not exist). If the **four conditions** listed above are satisfied, then we have a **collision** for sure:



Now we need to **call** our **collision** function from the **update** function:

```
var update = function() {
    player.moveHorizontally();

    enemies.forEach(function(element) {
        if (checkCollisions(player, element)) {
            alert("collision detected");
        }
        element.moveVertically();
    });
}
```

We add an **if** statement to check whether there was any **collision** between the **player** and one of the **enemies**. If so, we just pop up a **message**.

In the next lesson, we're going to implement our **end game** logic.

As an **exercise** try adding a final **rectangle** at the other end of the canvas (**opposite** to the player), which is going to be the game **goal**. Besides that, also add **collision detection** for this goal rectangle as well, in this case a **win** rather than a loss (as with the enemies so far).

If you have followed the lesson notes (the summaries for each video lesson of the course) then you're already up to date with the code. Otherwise, take a look at the **fixes** done to the course's code below.

As there have been changes in the code from the version presented on the video lessons, here's the **updated code** so far:

```
<!DOCTYPE html>
<html>
<head>
    <title>Crossing Game</title>
    <style type="text/css">
        canvas {
            border: 2px solid black;
            background-color: red;
        }
    </style>
</head>
<body>
    <h1>Crossing Game</h1>
    <canvas id='myCanvas' width='1000' height='500'></canvas>

    <script type="text/javascript">
        var canvas = document.getElementById('myCanvas');
        var ctx = canvas.getContext('2d');

        let screenWidth = 1000;
        let screenHeight = 500;
        let width = 50;

        class GameCharacter {
            constructor (x, y, width, height, color, speed){
                this.x = x;
                this.y = y;
                this.width = width;
                this.height = height;
                this.color = color;
                this.speed = speed;
                this.maxSpeed = 4;
            }

            moveVertically() {
                if (this.y > screenHeight - 100 || this.y < 50) {
                    this.speed = -this.speed;
                }
                this.y += this.speed;
            }

            moveHorizontally() {
                this.x += this.speed;
            }
        }

        var enemies = [
            new GameCharacter (200, 225, width, width, "rgb(0, 0, 255)", 2),
```

```
            new GameCharacter (450, screenHeight - 100, width, width, "rgb(0, 0, 255)
", 3),
            new GameCharacter (700, 50, width, width, "rgb(0, 0, 255)", 4)
        ];
        var player = new GameCharacter(50, 225, width, width, "rgb(0, 255, 0)", 0);

        document.onkeydown = function(event) {
            switch(event.key) {
                case 'ArrowRight':
                    player.speed = player.maxSpeed;
                    break;
                case 'ArrowLeft':
                    player.speed = -player.maxSpeed;
                    break;
            }
        }

        document.onkeyup = function(event) {
            player.speed = 0;
        }

        var checkCollisions = function(rect1, rect2) {
            let rect1x2 = rect1.x + rect1.width;
            let rect2x2 = rect2.x + rect2.width;
            let rect1y2 = rect1.y + rect1.height;
            let rect2y2 = rect2.y + rect2.height;

            return rect1.x < rect2x2 && rect1x2 > rect2.x && rect1.y < rect2y2 && rec
t1y2 > rect2.y;
        }

        var draw = function() {
            ctx.clearRect(0, 0, screenWidth, screenHeight);

            ctx.fillStyle = player.color;
            ctx.fillRect(player.x, player.y, player.width, player.height);

            enemies.forEach(function(element){
                ctx.fillStyle = element.color;
                ctx.fillRect(element.x, element.y, element.width, element.height);
            });
        }

        var update = function() {
            player.moveHorizontally();

            enemies.forEach(function(element) {
                if (checkCollisions(player, element)) {
                    alert("collision detected");
                }
                element.moveVertically();
            });
        }

        var step = function() {
```

```
            update();
            draw();
            window.requestAnimationFrame(step);
        }

        step();
    </script>
</body>
</html>
```

Note that **event.keyCode** is **deprecated** and has been **replaced** by the **event.key**. Besides, we now have the classic implementation of **collision** between **rectangles** in place. It works by **ensuring** there is **no gap** between **any** of the **4 sides** of the rectangles (in regardless on their sizes).

We're going to start by **implementing** the **exercise** proposed during our last lesson.

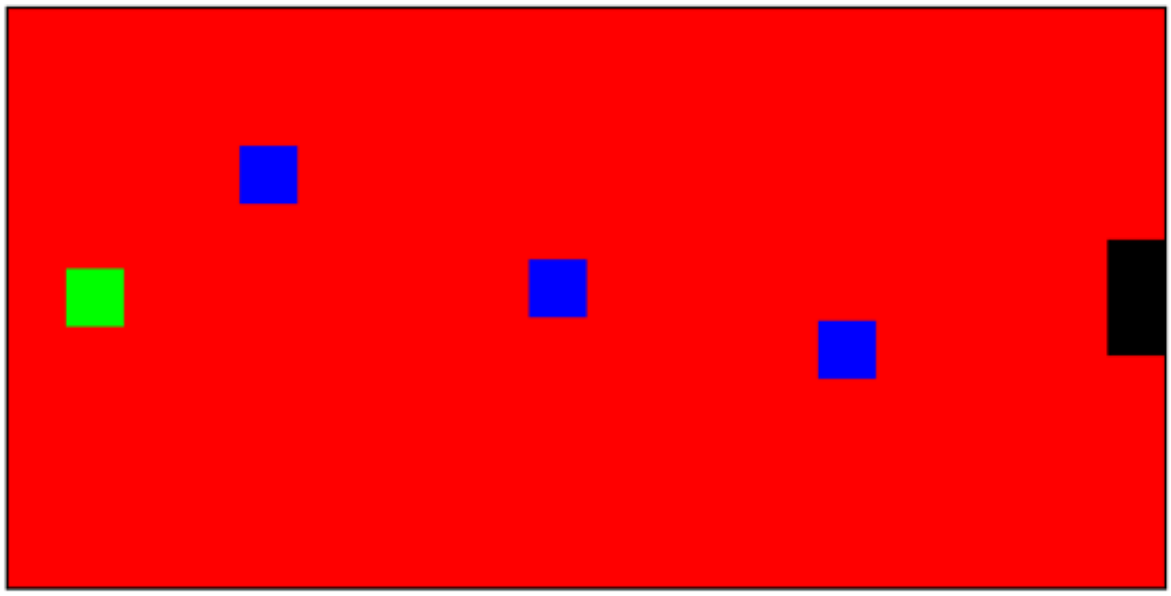
Let's start by **instantiating** the **goal** object:

```
var goal = new GameCharacter(screenWidth - width, 200, width, 100, "rgb(0, 0, 0)", 0)
;
```

Next, we **draw** our goal in the canvas. Add this to your **draw function**:

```
ctx.fillStyle = goal.color;
ctx.fillRect(goal.x, goal.y, goal.width, goal.height);
```

There we have our goal now in **black** at the other edge of our canvas:



To implement the **collision** detection, it's going to be pretty similar to how we did it for the enemies. Just add the following code to your **update** function:

```
if (checkCollisions(player, goal)) {
    alert("You win!");
}
```

We see that's working just fine:

However, it keeps popping up the message to us just as it does when the player collides with the enemies.

To fix that, add this **function** here **above** the **step** function:

```
var endGameLogic = function(text) {
    alert(text);
    window.location = "";
}
```

It's displaying the **alert** just the same, but also **closing** it after we hit "OK".

Now, we need to change our **calls** in the **update function** to use our newly implemented function **endGameLogic**:

```
var update = function() {
    if (checkCollisions(player, goal)) {
        endGameLogic("You win!");
    }

    player.moveHorizontally();

    enemies.forEach(function(element) {
    if (checkCollisions(player, element)) {
        endGameLogic("Game Over!");
    }
    element.moveVertically();
    });
}
```


If we **refresh** the page, we continue to get more than one window with the messages, though.

A quick solution is to create a **variable** that keeps track of whether the game is **live** or not:

```
var isGameLive = true;
```

The game is no longer live when the user **loses** or **wins** the game. So we need to change our **endGameLogic** to:

```
var endGameLogic = function(text) {
    isGameLive = false;
    alert(text);
    window.location = "";
}
```

Then, to guarantee that the game will only be animated if the game is still live, we add an extra check in our **step function**:

```
var step = function() {
    update();
    draw();

    if (isGameLive) {
        window.requestAnimationFrame(step);
    }
}
```

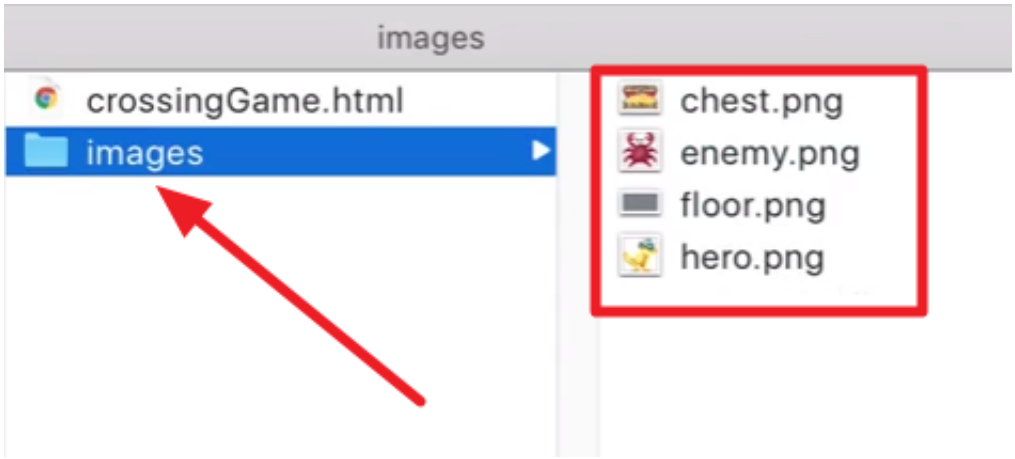
Now it’s working properly!

Feel free to **increase** the **speed** of the enemies as the user keeps **winning** the game as an exercise to **improve** the game. In the next lesson, we’re going to add the **sprites** to the screen so we don’t just have colored rectangles to play with.

In this lesson, we're going to take a look at **sprites** and use them in our game.

A sprite is an **entity** in the game with an **image** and **properties**, such as x and y positions, width, and height. We've been using rectangles so far, but we'll turn all of them into sprites (the **player**, **goal**, and **enemies**). We will even put the **background** as a sprite as well.

The **images** we're going to use are in the **images folder** of the **source material** provided for the course (see tab "**Course Files**"):



The sprites we're going to create will take the shape of these images rather than just being simple rectangles.

Let's add the code for our **sprites** like so:

```
var sprites = {}

var loadSprites = function() {
    sprites.player = new Image();
    sprites.player.src = 'images/hero.png';

    sprites.background = new Image();
    sprites.background.src = 'images/floor.png';

    sprites.enemy = new Image();
    sprites.enemy.src = 'images/enemy.png';

    sprites.goal = new Image();
    sprites.goal.src = 'images/chest.png';
}
```

You can place this code **below** our **GameCharacter** class.

We start by **instantiating** the sprites **variable** which will hold all of our sprites. It's just an empty **object** for now. Then we move to **loading** the sprites and **adding** them to our **sprites** object. That is, **assigning** the proper image to each of our characters and also to the background. We simply create an image for each one of them and pass the **path** to the image file we want to use.

We need to **call** our **load** function, right before we call on **step()** at the very end of our code:


```
loadSprites();
```

Right now we're **not** using our sprites because we're still **drawing** the colored rectangles on the canvas. We're going to **replace** the code in our **draw function** by doing the following:

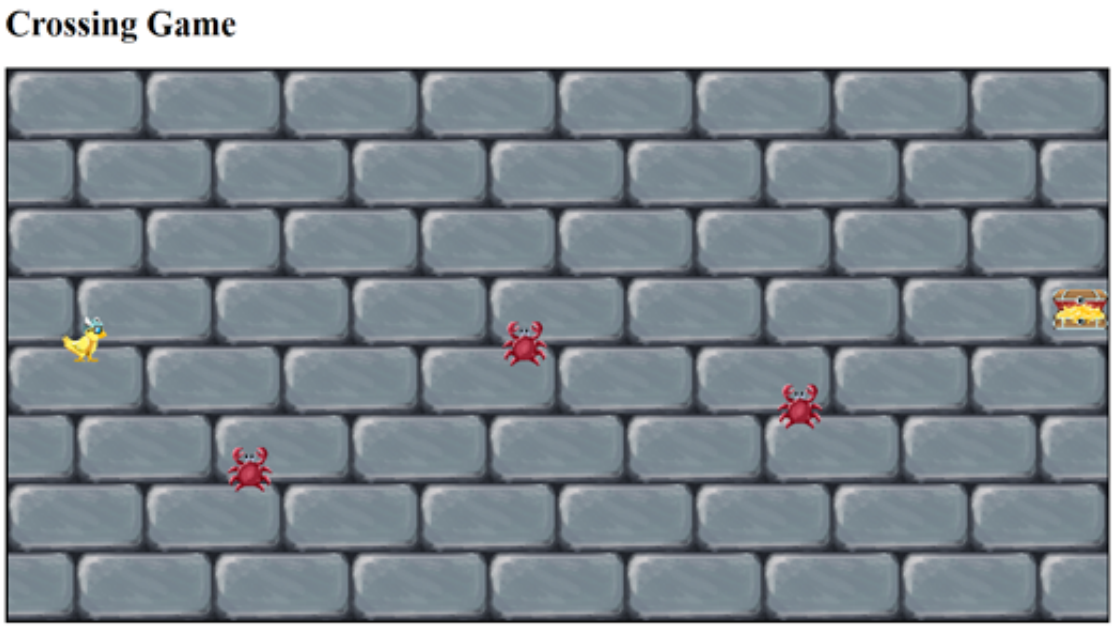
```
var draw = function() {
    ctx.clearRect(0, 0, screenWidth, screenHeight);

    ctx.drawImage(sprites.background, 0, 0);
    ctx.drawImage(sprites.player, player.x, player.y);
    ctx.drawImage(sprites.goal, goal.x, goal.y);

    enemies.forEach(function(element){
        ctx.drawImage(sprites.enemy, element.x, element.y)
    });
}
```

Here, we need to start by drawing the **background**, because the **drawImage** function will draw things on top of each other according to the **order** we enter them. For the **background**, we pass **x** and **y** positions as **0** so that it'll be drawn starting from the **top left corner** of the screen. We can **reference** the x and y positions for the other characters as we have them **instantiated** in our code. For the enemies **loop**, we're using the **same sprite** (sprites.enemy) but we **change** x and y positions according to **each** element of the **enemies array**.

Reloading our page:

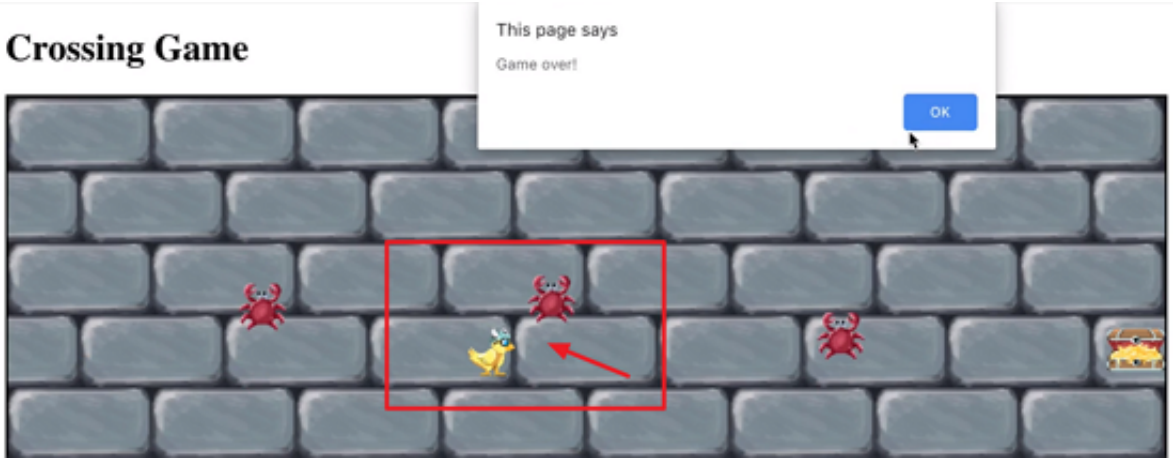


Let's just **align** the **player** with the **treasure chest** (the goal of our game). We just need to change the **player**'s **y** position to **200**:

```
var player = new GameCharacter(50, 225, width, width, "rgb(0, 200, 0)", 0);
```


Now our game is looking good!

You'll note that the character's rectangles are **bigger** than their actual sprites:



So there's room for improvement, of course. Feel free to add more stuff and change the game as you'd like!

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

At this point, you may have noticed that there is a bug in the game. If the **player** is moving and **any** key is pressed and released, the player **stops** moving because our **onkeyup function** sets the speed to **0** regardless of which key was released.

We can add some extra logic to make sure that isn't the case. In fact, we can go one step further and keep track of whether the user is pressing the **right** or **left** key at any time and **change direction** depending on which keys are being released.

For example, let's say we're moving to the **right** (by pressing the right key) and then we decide to move **left** by pressing the left key **without** releasing the right key (thus having **2** keys pressed **at the same time**, left being the most **recent** one). In this case, we should stop moving right and start moving **left**. However, when we **lift up** on the left key, because the right key is still pressed, we should start moving **right** again. Essentially, the idea is that we should only **stop** moving when **both** left and right keys are **not** being pressed.

To accomplish this, we have to create two **variables** to detect whether the **right** or **left** keys are being pressed at any time. They start out **false** and we set them to **true** when the keys are pressed down.

Add these two **boolean** variables at the **top** of our code (just below the **isGameLive** variable):

```
var isRightKeyPressed = false;
var isLeftKeyPressed = false;
```

Set them to **true** in the **onkeydown** function as follows.

The following code has been updated, and differs from the video:

```
document.onkeydown = function(event) {
    switch(event.key) {
        case 'ArrowRight':
            isRightKeyPressed = true;
            player.speed = player.maxSpeed;
            break;
        case 'ArrowLeft':
            isLeftKeyPressed = true;
            player.speed = -player.maxSpeed;
            break;
    }
};
```

Now, in the **onkeyup** function, we need to check which key is pressed because we won't respond to any keys except for the **left** and **right** ones. We then set the isRightKeyPressed or isLeftKeyPressed to **false** when appropriate. Finally, when one of those keys is released, we need to check if the other one is still pressed and change the speed appropriately (such that we move left if the left key is still pressed and so on).

The following code has been updated, and differs from the video:


```
document.onkeyup = function(event) {
    switch(event.key) {
        case 'ArrowRight':
            isRightKeyPressed = false;
            if (isLeftKeyPressed) {
                player.speed = -player.maxSpeed;
            } else {
                player.speed = 0;
            }
            break;
        case 'ArrowLeft':
            isLeftKeyPressed = false;
            if (isRightKeyPressed) {
                player.speed = player.maxSpeed;
            } else {
                player.speed = 0;
            }
            break;
    }
};
```

Basically, if the **keyUp** (the key that was released) is the arrow-right key, we only **stop** moving if the left key is **not** pressed or we move left, otherwise. We do the same thing with the arrow-left key. This will ensure that the user **switches** directions correctly or just stops moving whatsoever if none of these keys are pressed.

The following instructions have been updated, and differ from the video:

The code as-is allows the **player** to **disappear** from the screen to the **left** as we are not restricting the boundaries there. An interesting extra improvement we can do to our code then is to **check** whether the player is about to go off of the screen and set it back to the beginning of our canvas.

To do that, add an **if** statement to the **moveHorizontally function** in the **GameCharacter** class as seen below:

```
moveHorizontally() {
    this.x += this.speed;
    if (this.x         this.x = 0;
    }
}
```

Game Development Summary

That’s it for our game development portion of the course! We have successfully learned the basics of game development while building a game from scratch. The next step would be to improve the game by adding new features such as better collision detection, different difficulty levels, vertical movement, horizontal movement for the enemies, and so on.

If you have followed the lesson notes (the summaries for each video lesson of the course) then you're already up to date with the code. Otherwise, take a look at the **fixes** done to the course's code below.

As there have been changes in the code from the version presented on the video lessons, here's the **updated final code**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Crossing Game</title>
    <style type="text/css">
        canvas {
            border: 2px solid black;
            background-color: red;
        }
    </style>
</head>
<body>
    <h1>Crossing Game</h1>
    <canvas id='myCanvas' width='1000' height='500'></canvas>

    <script type="text/javascript">
        var canvas = document.getElementById('myCanvas');
        var ctx = canvas.getContext('2d');

        let screenWidth = 1000;
        let screenHeight = 500;
        let width = 50;
        var isGameLive = true;

        var isRightKeyPressed = false;
        var isLeftKeyPressed = false;

        class GameCharacter {
            constructor(x, y, width, height, color, speed) {
                this.x = x;
                this.y = y;
                this.width = width;
                this.height = height;
                this.color = color;
                this.speed = speed;
                this.maxSpeed = 4;
            }

            moveVertically() {
                if (this.y > screenHeight - 100 || this.y < 50) {
                    this.speed = -this.speed;
                }
                this.y += this.speed;
            }

            moveHorizontally() {
                this.x += this.speed;
                if (this.x < 0) {
```

```
                this.x = 0;
            }
        }
    }

    var enemies = [
        new GameCharacter(200, 225, width, width, "rgb(0, 0, 255)", 2),
        new GameCharacter(450, screenHeight - 100, width, width, "rgb(0, 0, 255)"
, 3),
        new GameCharacter(700, 50, width, width, "rgb(0, 0, 255)", 4),
    ];

    var player = new GameCharacter(50, 200, width, width, "rgb(0, 255, 0)", 0);
    var goal = new GameCharacter(screenWidth - width, 200, width, 100, "rgb(0, 0,
 0)", 0);
    var sprites = {};

    var loadSprites = function() {
        sprites.player = new Image();
        sprites.player.src = 'images/hero.png';

        sprites.background = new Image();
        sprites.background.src = 'images/floor.png';

        sprites.enemy = new Image();
        sprites.enemy.src = 'images/enemy.png';

        sprites.goal = new Image();
        sprites.goal.src = 'images/chest.png';
    }

    document.onkeydown = function(event) {
        switch(event.key) {
            case 'ArrowRight':
                isRightKeyPressed = true;
                player.speed = player.maxSpeed;
                break;
             case 'ArrowLeft':
                isLeftKeyPressed = true;
                player.speed = -player.maxSpeed;
                break;
            }
    }

    document.onkeyup = function(event) {
        switch(event.key) {
            case 'ArrowRight':
                isRightKeyPressed = false;
                if (isLeftKeyPressed) {
                    player.speed = -player.maxSpeed;
                } else {
                    player.speed = 0;
                }
                break;
            case 'ArrowLeft':
```

```
                    isLeftKeyPressed = false;
                    if (isRightKeyPressed) {
                        player.speed = player.maxSpeed;
                    } else {
                        player.speed = 0;
                    }
                    break;
                }
        }

        var checkCollisions = function(rect1, rect2) {
            let rect1x2 = rect1.x + rect1.width;
            let rect2x2 = rect2.x + rect2.width;
            let rect1y2 = rect1.y + rect1.height;
            let rect2y2 = rect2.y + rect2.height;

            return rect1.x < rect2x2 && rect1x2 > rect2.x && rect1.y < rect2y2 && rec
t1y2 > rect2.y;
        }

        var draw = function() {
            ctx.clearRect(0, 0, screenWidth, screenHeight);

            ctx.drawImage(sprites.background, 0, 0);
            ctx.drawImage(sprites.player, player.x, player.y);
            ctx.drawImage(sprites.goal, goal.x, goal.y);

            enemies.forEach(function(element) {
                ctx.drawImage(sprites.enemy, element.x, element.y);
            });

            // ctx.fillStyle = element.color;
            // ctx.fillRect(element.x, element.y, element.width, element.height);
            // ctx.fillStyle = player.color;
            // ctx.fillRect(player.x, player.y, player.width, player.height);
            // ctx.fillStyle = goal.color;
            // ctx.fillRect(goal.x, goal.y, goal.width, goal.height);
        }

        var update = function() {
            if (checkCollisions(player, goal)) {
                endGameLogic("You win!");
            }

            player.moveHorizontally();

            enemies.forEach(function(element) {
                if (checkCollisions(player, element)) {
                    endGameLogic("Game over!");
                }
                element.moveVertically();
            });
        }

        var endGameLogic = function(text) {
```

```
            isGameLive = false;
            alert(text);
            window.location = "";
        }

        var step = function() {
            update();
            draw();

            if (isGameLive) {
                window.requestAnimationFrame(step);
            }
        }

        loadSprites();
        step();
    </script>
</body>
</html>
```

Congratulations on making it to the end of the course!

You have learned how to code using **Javascript** and have built a game from scratch while learning how to also use the Javascript game engine.

You can confidently go forth and develop applications and web apps knowing that you have a good grasp of the Javascript language and game development concepts. You also have a small game to put on your resume and can improve upon it.

I hope you enjoyed the course – I'm very happy to be the one that taught you all how to code with Javascript and I look forward to seeing you in future courses!