Introduction

Hello everyone and welcome to our Javascript 101 course! In this course we will learn the basics of the Javascript language and apply it toward game development. 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 a small 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, and even 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?

1. Javascript language basics:
    1. Variables and strings
    2. Operators
    3. Arrays
    4. Dictionaries
    5. Functions, parameters, and return values
    6. If, else-if, and else statements
    7. While loops
    8. Break, continue, and return statements
    9. For loops
    10. Classes, objects, inheritance, and scope
2. Game making:
    1. Canvas
    2. Drawing on the canvas
    3. Movement and logic
    4. Touch events and player interaction
    5. Collision detection and end game logic
    6. Adding images and creating sprites

Language Basics

Here we will learn to code using Javascript. We will cover the concepts and syntax we need to start writing real Javascript code. As a set up to the game development basics, we will use examples that pertain to video games.

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. Compilers such as **rextester.com** or **playcode.io** allow you to write and run Javascript code to see results in real time. Although both online compilers work fine, **rextester.com** does not support string interpolation – more specifically, the use of backticks (`), while **playcode.io** does.

Any time you want to print a value to see the results of some code, simply run the code:

```
print(<value>);
```

replacing <value> with the variable or value that you want to print.

Variables

Variables provide a way to store and keep track of values within a program. We simply assign a value to a name and 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 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;
```


Strings are slightly more complex as they represent arrays of characters and have more functionality attached to them. However, we can treat them like basic variables for now. They represent any text in a program such as with 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 in throug ${<expression>}. For example, if we want to print the message: “I am <age> years old.” but use the age variable, we could do so with:

```
var ageMessage = `I am ${age} years old.`; // ageMessage = “I am 1 years old.”
```

If we know a variable value will never change, we can use a constant instead. These work the same as variables except that we cannot change the value one assigned. Declare constants like this:

```
const pi = 3.14159;
let characterName = “Nimish”;
```


Operators

Javascript contains 5 types of operators that we will cover: assignment, arithmetic, comparion, logical, ternary. Assignment is the simplest and consists of just one: =. We have already seen it in action and 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 is used to raise something to the power of something else. These operations are generally performed between two numbers. For example, I could do perform basic addition like this:

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

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

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

Considering that number1 was reassigned the value of 5, 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.

Comparison operators operators return a boolean value. These include: >, >=, <, <=, ==, !=, and === and are used to compare numbers, strings, booleans, and other variables. All return a true or false value and some examples include:

```
var boolResult = 5 > 2; // true
boolResult = age == 5; // true
boolResult = isGameOver != true; // false
```

The // is just to indicate a comment and would be ignored by the compiler. The === returns true only 2 values and types are equivalent such as:

```
boolResult = 5 === “5”;
```

This returns false as the two are kind of the same value but not the same type.

Logical operators consist of !, &&, and || and all return a boolean value. The ! operator is the simplest as it only reverses the value of a boolean (turns true to false and false to true) like with:

```
var boolResult = !isGameOver;
```

This stores the value of false in boolResult as boolean1 contains true. The && operator is the and operator and || is the or operator. These allow you to test for additional cases. For example:

```
boolResult = 5 > 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 we will cover if-else statements later.

Arrays

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 of [] separated with commas. A simple example would be:

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

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

```
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 at index 0 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 length 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.

You can also create and use multidimensional arrays by specifying rows and columns in a matrix-like structure. To fetch or change items, specify the row, then column. An example can be seen below:

```
var levels = [[1.1, 1.2, 1.3], [2.1, 2.2, 2.3, 2.4], [3.1, 3.2, 3.3]];
var currentLevel = levels[1][2]; // currentLevel = 2.3
```

Dictionaries

Dictionaries, or maps, are similar to arrays except that instead of single values, each item is a key-value pair. This is beneficial as we can assign and retrieve values based on keys rather than positions in an array. The order in which the items are added to a dictionary is irrelevant so if you want to store items based on a specific position, use an array. If you want to associate each stored value with a key other than an index, use a dictionary. Create a dictionary like this:

```
var inventory = {“fruit”: 2, “helmet”: 1, “knife”: 3};
```

You access items based on the key rather than the index so if I want the value stored at “fruit” I fetch it like this:

```
var fruitQuantity = inventory[“fruit”]; // fruitQuantity = 2
```

To modify a value, simply access the key and assign a value. If the key doesn’t exist, it will create a key value pair, otherwise it replaces the value at that key. For example:

```
fruitQuantity[“fruit”] = 4;
fruitQuantity[“bread”] = 2; // inventory = {“fruit”: 4, “helmet”: 1, “knife”: 3, “bre
ad”: 2}
```


Functions

As of now, all of the code we write 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 age by 1. We could write something like this:

```
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 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?”
```


If, Else-if, and Else Statements

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 which returns true or false and execute 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 put this within a function. With this code, we will increase the position by 1 because keyPressed == "r" returns true. If keyPressed was anything else, we do not execute the code 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 is ignored. This is different with multiple if statements in a row without else if statements as in that case, all tests would be performed.

Finally, there is the else case. This is like a default; it provides some code to execute if every previous test fails. Just 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 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 so we could do this:

```
let endPosition = 5;

if (keyPressed == “r” && endPosition < 5) {
    position++;
}
```

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

While Loops

Sometimes in code, we want to execute the same code multiple times. However, it's bad practice to repeat code and sometimes, we need to run the code hundreds or thousands of times so 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 < endPosition) {
    position++;
}
```

This code runs 4 times, bringing our final position to 5. On each loop iteration, we check to see if position < endPosition. If true, we run the code in the loop and repeat the check. If false, we exit out of the loop and continue with the rest of the code.

Loops can run 0 times if the condition fails right off the bat. While loops can also run infinitely if we are not careful. If we write the code in such a way that the test will never fail, the loop will continue to run until the program runs out of memory and eventually crashes. We can control this using break and continue statements.

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 as 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 probably execute some end-game logic. This is very similar to a return statement in a function in that a return statement will exit and function. You can also call return within a loop to break out of the loop and its enclosing function if it is 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) {
    if (numbers[i] % 2 != 0) {
        continue;
    }

    print(numbers[i]);
}
```

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] % 2 == 0 and if so, print numbers[i] 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.

For Loops

While loops are great for running code when we don't know exactly how many times the loop will need to run. For example, we often use while loops in games to constantly check for collisions and user interactions and update movement and re-render 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 have to pair them up. For example, if we have an array of values and want to print them, we could do so like this:

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

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

Note the start point, end point, and iterator 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--) {
    print(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 has to 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) {
    print(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.

Classes

The class and object system can be observed in many languages, particularly the object oriented languages. Objects in code are constructs with state and behaviour; state or properties are maintained through variables (called fields) and behaviour is added through functions (called methods). Classes are the code implementation of objects specifying the properties and behaviours that they should have; they are kind of like a blueprint. We will focus on classes first.

Classes in Javascript can be created and used in a few ways. We will explore the basic ways first then move onto the more traditional way of doing things. The first way looks very similar to a dictionary; the main difference is how we access the items within it. Let's create an object that represents a basic game character:

```
var gameCharacter = {
    name: "Nimish",
    xPosition: 2,
    yPosition: 1,
    move: function(xAmount = 0, yAmount = 0) {
        this.xPosition += xAmount;
        this.yPosition += yPosition;
    }
};
```

There's a lot going on here so let's break it down. We're creating a variable called gameCharacter with the properties of name, xPosition, and yPosition. These represent the state. There is also a method called move that takes in an xAmount and yAmount and changes the xPosition and yPosition. This isn't a class like you would see in languages like Java or Swift with a constructor; it's a one-off class that is just a single container for values and the one function. The keyword "this" refers to the property that belongs to the current class.

To access or modify the values or function, just use syntax like this:

```
var name = gameCharacter.name;
gameCharacter.xPosition = 3;
gameCharacter.move(4, 2);
```

You can even create new properties and methods on the fly like this:

```
gameCharacter.color = 'blue';
gameCharacter.printName = function() {
    print("Hi my name is ${this.name)!"
};
```

With this type of class, we can assign values without creating variables. This is particularly useful when creating arrays of objects.

The above way of creating an object is interesting in that it is actually creating the class and the object at the same time. The second way of creating a class is different in that it separates the two processes. It provides the class implementation by putting it inside of a function and we call upon the function to create new instances of the class (objects). For example, we could create the same class as above by doing this:

```
var GameCharacter = function(name, xPosition, yPosition, move) {
    this.name = name;
    this.xPosition = xPosition;
    this.yPosition = yPosition;
    this.move = move;
};
```

This is a bit more abstract as in order to assign values, we have to call upon the function and pass in values. However, this way, we can create multiple instances of the gameCharacters and assign different values to each of them much more easily. The syntax is cleaner even though the functionality is essentially the same. To create an instance of the class, we could do something like this:

```
var move = function(xAmount = 0, yAmount = 0) {
    this.xPosition += xAmount;
    this.yPosition += yPosition;
};

var newCharacter = new GameCharacter(“Nimish”, 2, 1, move);
```

Note that we created a function expression (a variable that represents an entire function) called move and passed that in. To access or modify state or behaviour of an instance, we could do something like this:

```
newCharacter.name = “Zenva”;
newCharacter.move(0, 1);
```

We can add or modify state or behaviour of every GameCharacter by using the “prototype” keyword like this:

```
GameCharacter.prototype.color = “blue”;
GameCharacter.prototype.printName = function() {
    print(“Hi my name is ${this.name)!”
};
```

This only affects values that have no explicitly been assigned, however. In the above example, it has added the color property with the value of “blue” as well as the function to each GameCharacter instance. But if we do something like this:

```
GameCharacter.prototype.name = “Bob”;
```

newCharacter.name is still “Zenva” and not “Bob” as we have explicitly assigned a value for name to our instance.

The third and final way to create classes and objects is the most similar to other object oriented languages and involves creating a separate class with fields, methods, and constructor. The constructor is a special function (similar to the previous example) that is used to set up an instance of the class with initial values. For example, we could do the same as above by doing this:

```
class GameCharacter {
    constructor (name, xPosition, yPosition) {
        this.name = name;
        this.xPosition = xPosition;
        this.yPosition = yPosition;
    }

    move (xAmount = 0, yAmount = 0) {
        this.xPosition += xAmount;
        this.yPosition += yPosition;
    }
}
```

This syntax is much cleaner as it provides clear separation for each component of the class, even though it provides the same kind of construct as the previous 2 examples. Also note how methods in classes do not need the function keyword. To create instances of GameCharacter this way, do something like this:

```
var newCharacter = new GameCharacter(“Nimish”, 2, 1);
```

We can access the variables and functions by doing this:

```
var xPos = newCharacter.xPosition;
newCharacter.move(1, 0);
```

One of the major benefits of choosing this final way of declaring and using classes is that it is easy to inherit from other classes. Inheritance allows a class to take on the characteristics of another class along with its own, unique characteristics. We inherit from another class using the extends keyword.

Let's look at an example. It could be a non player character that shares the same basic characteristics as a regular GameCharacter but cannot move vertically which means its yPosition would always be 1. An example could be something like this:

```
class NonPlayerCharacter extends GameCharacter {
    constructor(name, xPosition, color) {
        this.color = color;
        super(name, xPosition, 1);
    }

    move(xAmount) {
        super.move(xAmount, 0);
    }
}
```

Note first that NonPlayerCharacter does not explicitly set up the name, xPosition, or yPosition. This is because it has inherited these properties from GameCharacter. The same would be true of any methods that GameCharacter has, except that we are adding in the move function because we are providing a slightly different implementation. It does however, set up a color. This is something unique to the NonPlayerCharacter and does not exist in the GameCharacter class.

Also note the use of the word super a couple of times. Super refers to the superclass implementation of something. The superclass is the class that is inherited from, in this case the GameCharacter. The NonPlayerCharacter class is considered a subclass in this situation because it is inheriting from the superclass. If we call upon super.<variable> or super.<method> it calls upon the superclass implementation of that but changes the values within the current class instance. That may be confusing so let's look at some examples. If we create an instance of a NonPlayerCharacter, we can do so like this:

```
var enemy = new NonPlayerCharacter("enemy1", 5, "blue");
```

This will explicitly set up the color because our constructor does that. For the others, however, it calls upon the superclass constructor which just sets the attributes up and passes in a value of 1 for the yPosition. A similar thing happens when we call upon the NonPlayerCharacter move function. It increases just the xPosition by the amount specified by using the GameCharacter implementation of the move function.

The last thing to note here is that NonPlayerCharacters can call upon all 4 of the properties and the move function. GameCharacters however, can only call upon 3 of the properties and the move function (and the move function has a slightly different implementation) as the color attribute was declared only in the NonPlayerCharacter class. Methods and fields/properties/attributes declared in a subclass cannot be accessed in a superclass.

At this point you might ask, why bother with inheritance when it seems like a complex topic. Why not just create the variables, functions, etc. twice? The answer lies in code maintainability. Rewriting the same code is bad for two reasons: one, it wastes time and memory and two: if you need to change the code, you need to change it in two or more spots in the program. If you write a superclass and

extend it from multiple subclasses and realize that you need to change some values, you only need to change the superclass values and it updates everywhere. This is much better practice and where possible, always try to avoid rewriting code.

The final topic to broach here is the idea of scope. Scope dictates when and where we can access variables. Take for example, some code structured like this:

```
var number1 = 1;

function addAllNumbers(number) {
    var number2 = 2;

    if (number > number2) {
        var number3 = 3;
        return number + number1 + number2 + number3;
    }

    return number + number1 + number2;
}
```

Here we have multiple levels of variables. Number1 is considered global and is accessible everywhere as it is declared outside of any function or class. Variables declared within a class are accessible within that class or any instance of that class. Number is a parameter and so is accessible anywhere within that function. Number2 is considered a local variable and is only accessible within the addAllNumbers function because it is declared in the body. Number3 is still a local variable but it is only accessible within the if statement because that is where it is declared. There are some exceptions but the general rule of thumb is that a variable is only accessible within the set of {} in which it is declared. Otherwise, it is as if the variable does not exist!

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.

Game Basics

By now we should have a pretty good grasp of the language and how to use it effectively. Now we will learn the pieces needed to create simple games using Javascript. We will start with the 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 but interactivity, logic, and good looking graphics.

How do I Follow Along?

Now we have to 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 and see the output, simply right click on the file and open with the browser of your choice.

Creating a Canvas

The first thing we need to do is 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 rectangle with a color or image as a background and draw everything on top of it. The great part is that HTML already has a canvas tag that contains special properties. The following setup will create a canvas with a red background and a black border:

```
<html>
<head>
<style>
    canvas {
        border: 1px solid black;
        background-color: red;
    }
</style>
</head>
<body>
    <canvas width="500" height="500" id="myCanvas"></canvas>
</body>
</html>
```

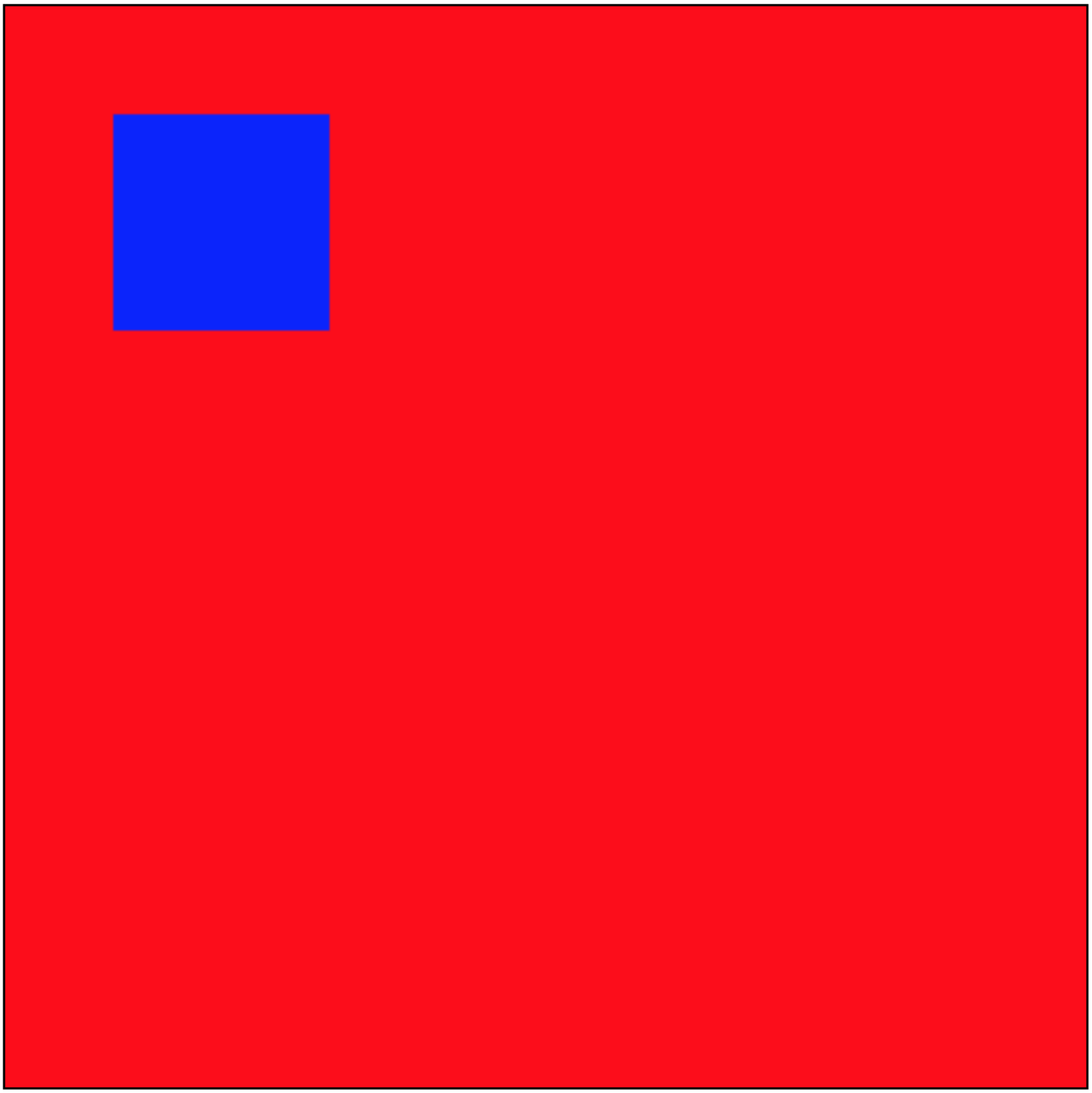
Note the canvas tag which requires a width and height. Now we will want to draw on the canvas at some point to set up obstacles, graphics, and any other non-sprite items that appear on screen. We first need to reference the canvas and can do so inside of a <script> tag in the body like so:

```
<body>
    <canvas width="500" height="500" id="myCanvas"></canvas>
<script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");
</script>
</body>
```


Drawing on the Canvas

Once we have the context, we can draw rectangles and other shapes on it. To draw a rectangle, you first have to add a fill style (in our case just the color), then draw the rectangle by specifying its x and y positions and width and height. 0 x and 0 y starts in the top left corner. Adding a rectangle like this:

```
<script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");

    ctx.fillStyle = "rgb(0, 0, 255)";
    ctx.fillRect(50, 50, 100, 100);
</script>
```



This is all still inside of the <script> tag. This is what is outputted when you run the code in a browser .

Adding Movement to a Canvas

Adding movement is as simple as changing x or y positions and updating the way that the canvas is drawn. We first need a function to update the positions of the items on the canvas. We next need a function to clear the canvas and draw the rectangle again at the new positions. We then animate the frames. We can start with a simple task: moving our rectangle from left to right. We only need to update the x position by some speed value. Here is a sample (written inside of the <script> tag:

```
var x = 50;
var speed = 1;
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");

var update = function() {
    x = x +  speed;
};

var draw = function() {
    ctx.clearRect(0,0,500,500);
    ctx.fillStyle = "rgb(0, 0, 255)";
    ctx.fillRect(x, 50, 100, 100);
};

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

step();
```

The square moves slowly from left to right. However, it does eventually go off screen because we have never given it a reason to switch direction. We should give it a boundary and once it reaches that boundary, we should send it back the way it came. This way we can have it bounce back and forth, all the while staying on screen. If we apply some logic to the update function to change speed to negative when it reaches the right hand side and do the same on the far left. Changing the update function to this achieves the desired result:

```
var update = function() {
    if (x > 350 || x < 50) {
        speed = -speed;
    }

    x = x +  speed;
};
```

You could also replace the 350 and 50 with startZone and endZone variables. Where possible, it is better to use variables than literal values as they are easier to change around later. The step function acts kind of like a game loop, executing the code within it many times each second.

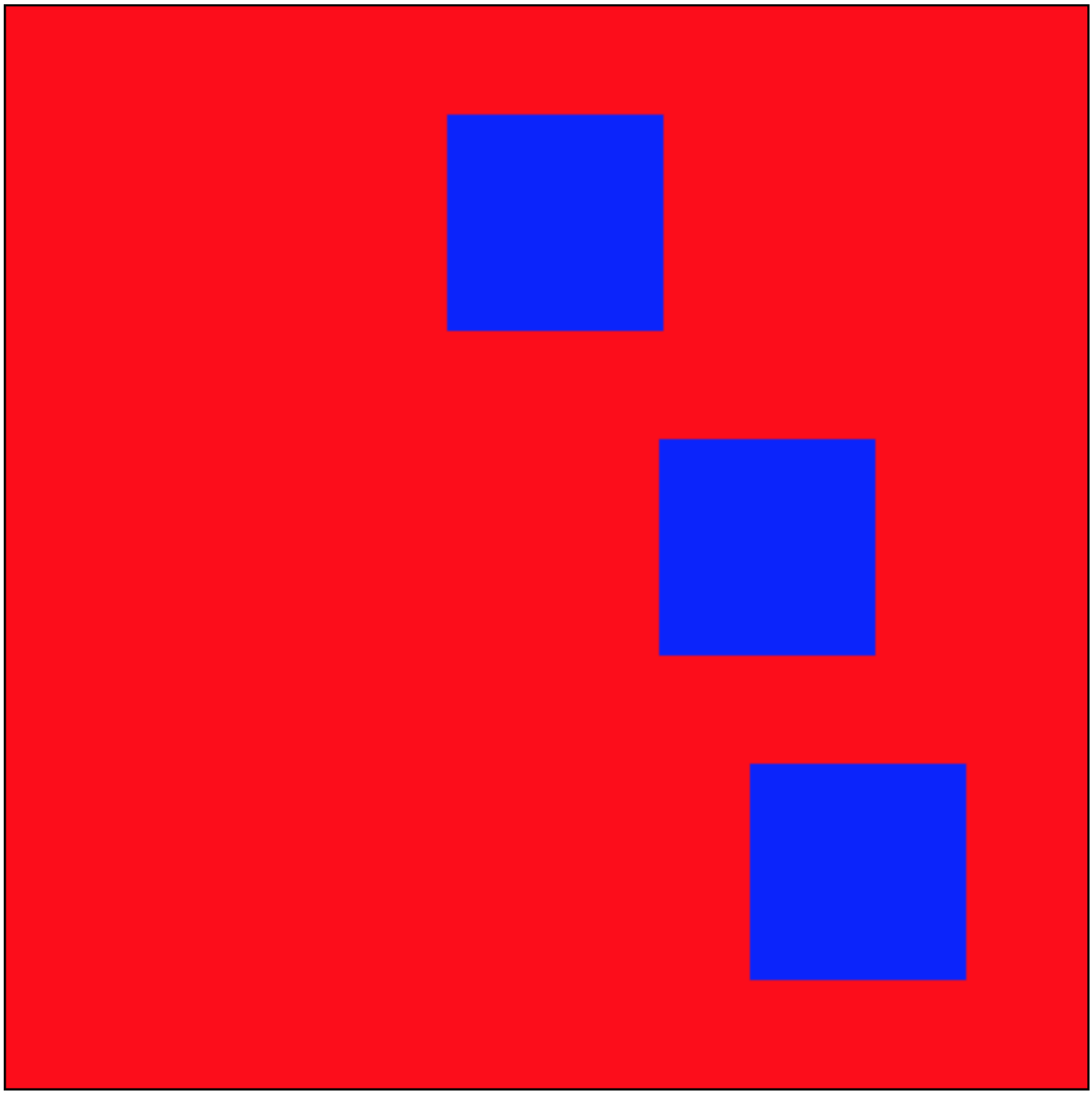
Updating Multiple Items

Our next step is to apply this same logic to multiple items on the screen. Games will more often than not have multiple components moving at the same time and each one will need to be updated based on factors such as x and y position, width and height, and speed as these may all be different for each component. We can accomplish this task quite easily with a couple of forEach loops and an array of values.

First, we will define an array of objects with x and y values and speed. We will keep it simple and modify our structure once we start adding in more attributes. Then we will loop through each of these objects when it comes time to update the x positions and redraw everything. We can create the array and update our update and draw functions like this:

```
var startPos = 50;
var endPos = 350;
var length = 100;
var rectangleColor = "rgb(0, 0, 255)";

var rectangles = [
    {
        x: 50,
        y: 50,
        speed: 1,
    },
    {
        x: 50,
        y: 200,
        speed: 2,
    },
    {
        x: 50,
        y: 350,
        speed: 3,
    },
];

var update = function() {
    rectangles.forEach(function(element, index) {
        if (element.x > endPos || element.x < startPos {
            element.speed = -element.speed;
        }

        element.x += element.speed;
    });
};

var draw = function() {
    ctx.clearRect(0,0,500,500);
    ctx.fillStyle = rectangleColor;
    rectangles.forEach(function(element, index){
        ctx.fillRect(element.x, element.y, length, length);
    });
};
```


The step function and canvas code stays the same. Essentially, we just loop through each of the items that need to be updated and individually update the x positions. We then loop again through each item and draw it based on its new x position. It should look something like this:



Add Player Controls

Now that we know how to add movement for multiple items, it is time we added player controlled movement. There is no point in playing a game if we as the users cannot interact with it. Let's clear the canvas (get rid of the rectangles) and just focus on movement from a player perspective. To make things easy, we will just focus on left to right movement. We can add listeners to listen for events such as key presses or mouse clicks but seeing as we are just dealing with a few arrow key presses, let's just add keyup and keydown methods. First we should add the player object like this:

```
var player = {
    x: 50,
    y: 50,
    maxSpeed: 2,
    currentSpeed: 0,
    color: "rgb(0,255,0)",
};
```

Note that I made the player green and we have a max and current speed. Then we can add the keyup and keydown functions like this:

```
document.onkeydown = function(event) {
    if(event.keyCode == 39) {
        player.currentSpeed = player.maxSpeed;
    } else if (event.keyCode == 37) {
        player.currentSpeed = -player.maxSpeed;
    }
};

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

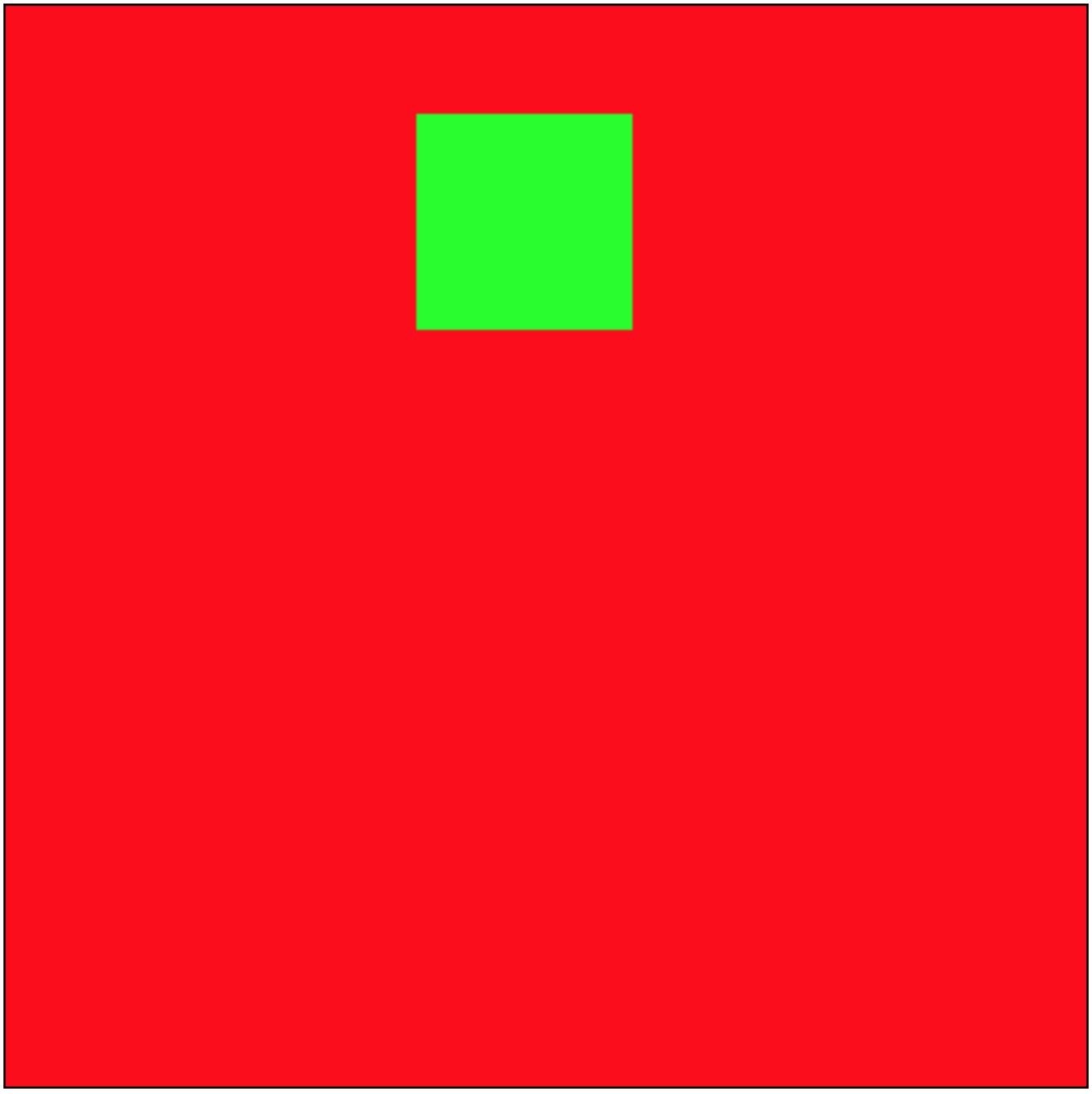
The keydown function is triggered anytime any key is pressed so we have to check to see which one has been pressed with the keyCode attribute. 39 is the right key and 37 is the left key. For a complete list of key codes, simply search online for javascript key codes. Here we are just setting the currentSpeed to positive if they right key is pressed, negative if the left is pressed, and 0 if the user lifts their finger. This ensures the player moves right, left, and stops moving respectively. Next we can add the update and draw functions and call them in the step function like this:

```
var updatePlayer = function() {
    player.x += player.currentSpeed;
}

var drawPlayer = function() {
    ctx.clearRect(0,0,500,500);
    ctx.fillStyle = player.color;
    ctx.fillRect(player.x, player.y, length, length);
}

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



```
    drawPlayer();
    window.requestAnimationFrame(step);
};
```

This results in a green square that moves from left to right or stops when the user presses the correct keys or lifts their finger. The final result looks something like this (but with movement):



Adding Collision Detection

Now that we can move our player about and know how to automate bot movement, we can start making our game more fun. Let's move on to implementing object collision detection. To do so, we just need to check the moving components every time the positions are updated and see if anything is touching the player. In concept, this is fairly simple; we get the x and y position and width and height of the items and see if the x positions and y positions overlap. If they do, we can implement the end game logic.

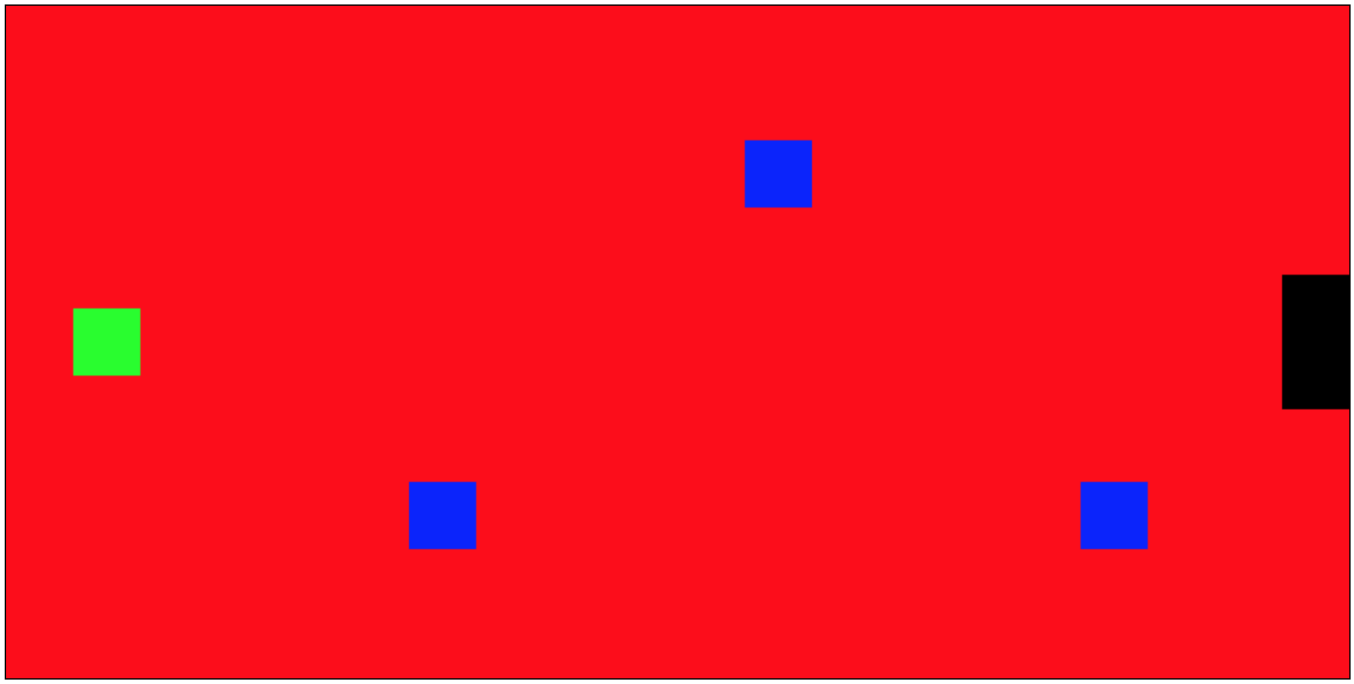
To start, let's rearrange our game so that enemies are moving up and down and the player is moving left to right. We should also make the screen bigger and the enemies smaller. I also started 2 enemies at the top and 1 at the bottom and centered the player vertically. We can do so with this code:

```
<canvas id="myCanvas" width="1000" height="500" ></canvas>
<script>
    var topEdge = 50;
    var bottomEdge = 400;
    var length = 50;
    var rectangleColor = "rgb(0, 0, 255)";

    var rectangles = [
        {
            x: 300,
            y: 50,
            speed: 1,
        },
        {
            x: 550,
            y: 400,
            speed: 2,
        },
        {
            x: 800,
            y: 50,
            speed: 3,
        },
    ];

    var player = {
        x: 50,
        y: 225,
        maxSpeed: 2,
        currentSpeed: 0,
        color: "rgb(0,255,0)",
    };

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

    var update = function() {
        rectangles.forEach(function(element, index) {
            if (element.y > bottomEdge || element.y < topEdge {
                element.speed = -element.speed;
            }
```



```
            element.x += element.speed;
        });
    };

    var draw = function() {
        ctx.clearRect(0,0,1000,500);
        ctx.fillStyle = rectangleColor;

        rectangles.forEach(function(element, index){
            ctx.fillRect(element.x, element.y, length, length);
        });

        ctx.fillStyle = player.color;
        ctx.fillRect(player.x, player.y, length, length);
    };

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

    step();
</script>
```

Note the switches in x and y positions and the reintegration of player and enemy code all in one. You should see something like this:



With the enemies moving up and down. Now we want the player movement so add the keyDown and keyUp functions from before:


```
document.onkeydown = function(event) {
    if(event.keyCode == 39) {
        player.currentSpeed = player.maxSpeed;
    } else if (event.keyCode == 37) {
        player.currentSpeed = -player.maxSpeed;
    }
};

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

This allows the player to move left to right. Finally we need the collision detection. We will build a function that takes in 2 entities and checks to see if there is any x and y overlap like this:

```
var checkCollision = function(rect1, rect2) {
var xOverlap = Math.abs(rect1.x - rect2.x) <= length;

var yOverlap = Math.abs(rect1.y - rect2.y) <= length;
    return xOverlap && yOverlap;
};
```

This returns true if the x and y both overlap. We take the absolute values because we don't know which rectangle is overlapping with the other. Now we need to add the checks to each enemy when it is moved. We can add the code in the update function so update should look like this:

```
var update = function() {
    rectangles.forEach(function(element, index) {
        if(checkCollision(player, element)) {
            gameOverLogic();
        }

        if (element.y > bottomEdge || element.y < topEdge {
            element.speed = -element.speed;
        }

        element.x += element.speed;
    });
};
```

Note the collision detection at the top. That calls a function called gameOverLogic which we haven't yet created. This will basically freeze the screen and pop up an alert that says game over. We will also need a variable called gameLive to keep track of whether or not the game is still running. This helps us determine whether to update anything or not. Create a

```
var gameLive = true;
```


variable at the top with the other variables and update the step function to contain this code:

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

Finally, implement the gameOverLogic function to contain this code:

```
var gameOverLogic = function() {
    gameLive = false;
    alert('Game Over!');
    window.location = "";
};
```

This executes the proper end game logic when a player and enemy collide. This ensures that the game stops running and an alert pops up. It also resets the game once the player dismisses the alert.

Sprites

The final task that we have left is to change our boring rectangles to interesting images. We can create what are called Sprites to represent our characters, enemies, background, and goal. Sprites are interesting in that they take the shape of the image rather than just being a rectangle. They are also library objects with built in functionality and properties.

We first need to obtain some images and put them in a folder in the same directory as the JS file. Let's call this directory images. There should be 4 images provided: a chest (the goal), an enemy, a hero (the player), and a floor (the background). If not, feel free to provide your own images and scale them accordingly. Now we want to load the images and create sprites. First create a dictionary to hold them:

```
var sprites = {};
```

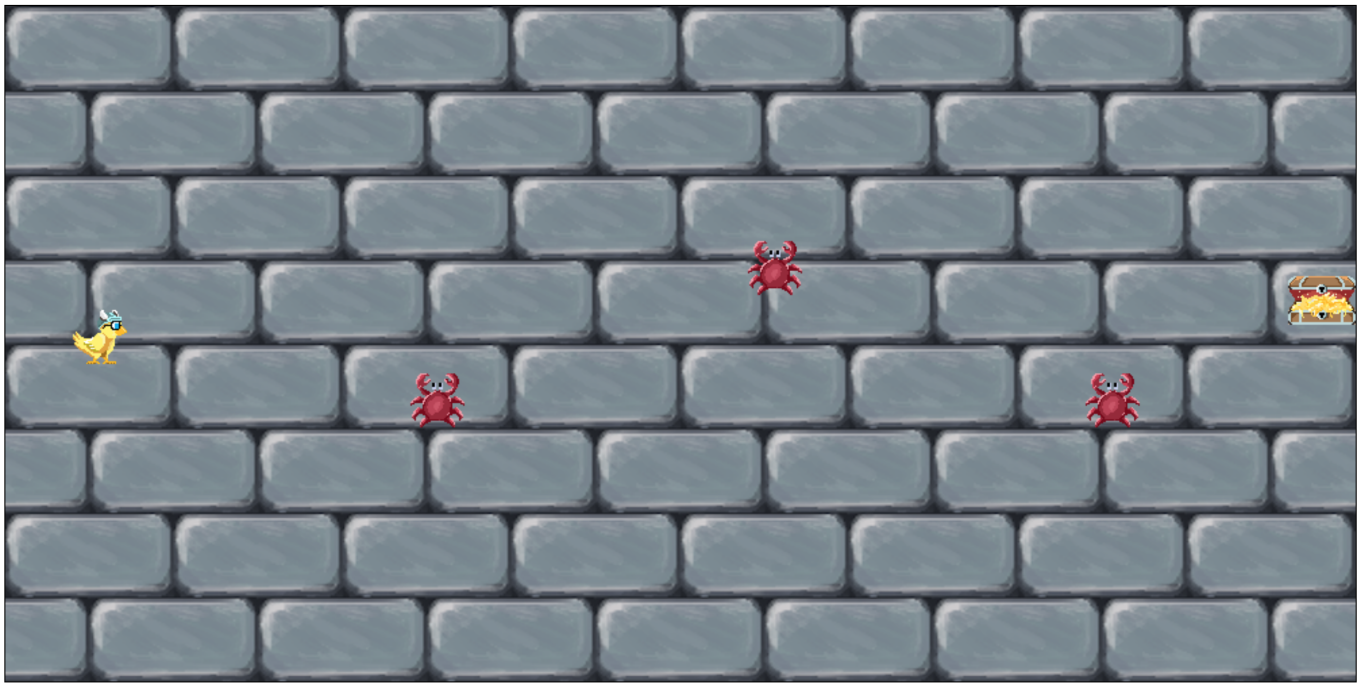
Then create a function to load the sprites based on images. Something like this will do:

```
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';
}
```

This creates sprites for each of the images and adds them to the sprites dictionary. Call loadSprites() just before you call step(). Finally, we need to modify the draw function to draw the sprites instead of the rectangles. Change it to this:

```
var draw = function() {
    ctx.clearRect(0,0,1000,500);
    ctx.drawImage(sprites.background, 0, 0);
    ctx.drawImage(sprites.player, player.x, player.y);

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

    ctx.drawImage(sprites.goal, goal.x, goal.y);
};
```

This sets Sprites in place of the boring rectangles, adds a nice background, and a fun chest as an end goal. This is what the final product looks like:



Game Development Summary

That’s it for our game development portion! 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 from the enemies, and so on.

Course Summary

Congratulations on making it to the end of the course! You have learned how to code using Javascript and have build a game from scratch while learning how to 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!