BUILD AN INFINITE RUNNER IN JAVASCRIPT

Table of Contents

Introduction

Welcome to IT Girls 2017! At the end of this booklet, you'll have coded your own JavaScript infinite runner game! We'll learn the basics of JavaScript along with a few tools that help to make creating JavaScript games easier. This booklet goes with a website *p5.kayesspeadesign.com* there you will find source code for the whole project as well as code snippets to make building the game easier. Whenever you're asked to edit code, you can copy and paste the relevant text from this webpage.

How to read this book

This booklet will use a few patterns to make reading it a bit easier.

Bold

Bold text will signify a new term. New terms will shortly be followed by a contextual definition.

Blue Italic

Blue italic text will signify URLs.

`Constant Width`

Constant width text will signify elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

```
Highlighted text
```

Highlighted text blocks will signify code that needs to be added our edited. It is highlighted to make it easier to read and see it's parts. In this booklet, any new code will be highlighted. Gray code will signify code that has been added in a previous step and shouldn't be changed. Ellipses (…) will indicate that there is text in the code just before or after the code you are adding, but is not displayed again.

What is JavaScript?

JavaScript is the programming language of the web. Many of the websites you use every day employ JavaScript to add **interactivity**, or make them respond to you when you use them. JavaScript is a great programming language for beginners because of its ease of use as well as the fact that it's literally all over! We'll be using JavaScript along with a few JavaScript **libraries**.

What Is a Library?

A library is a convenient way to package code. JavaScript libraries have lots of helpful code snippets that make it easier to build programs by take care of some of the more difficult or time-consuming tasks programmers have t do.

What Are P5.js and P5.play?

One of the libraries we'll be using is called **p5.js**. It's designed for artists and beginners to be able to dive right in and make things! It's the perfect library for us to use because it takes the guess work out of making the pictures and movements we'll need for our game. You can find out more about p5.js at *www.p5js.org*.

P5.js focuses mainly on the **HTML canvas**, a place that allows us to draw right inside of the webpage's screen. The canvas is made specifically for drawing and displaying images on a page and is the perfect place for our game. P5.js also has extender libraries that will help us make our game even better in less time. One of these libraries is called **p5.play**. It handles most of the physics we need for our game to work. It handles collisions, physics, and managing all the parts of the game. You can find out more about p5.play at *http://p5play.molleindustria.org/*.

Let's Start Coding

Now that we've been introduced to the tools, let's dive right in.

Setting Up Our Coding Environment

For this class, we'll be coding on CodePen. CodePen is a place for web developers to show each other their work online. We'll be using it because it allows us to save our work on the cloud and get up and running in a flash. Codepen has its own built-in **text editor**, or program to write our code. Signing up for CodePen is easy, but if you're at the IT Girls class, we've set up a codepen account for you already. Your login information is in the back of this book.

Go to *www.codepen.io* and click "Sign-Up" in the top right-hand corner. Click the black "Join CodePen Free" button in the middle. Put in your name, select a username, provide your email address, and choose a password and click "Submit". You can skip the "profile" portion of the sign-up by scrolling down and clicking "Save & Continue". Click the "Let's Go" button to take the virtual tour of CodePen to familiarize yourself with the site.

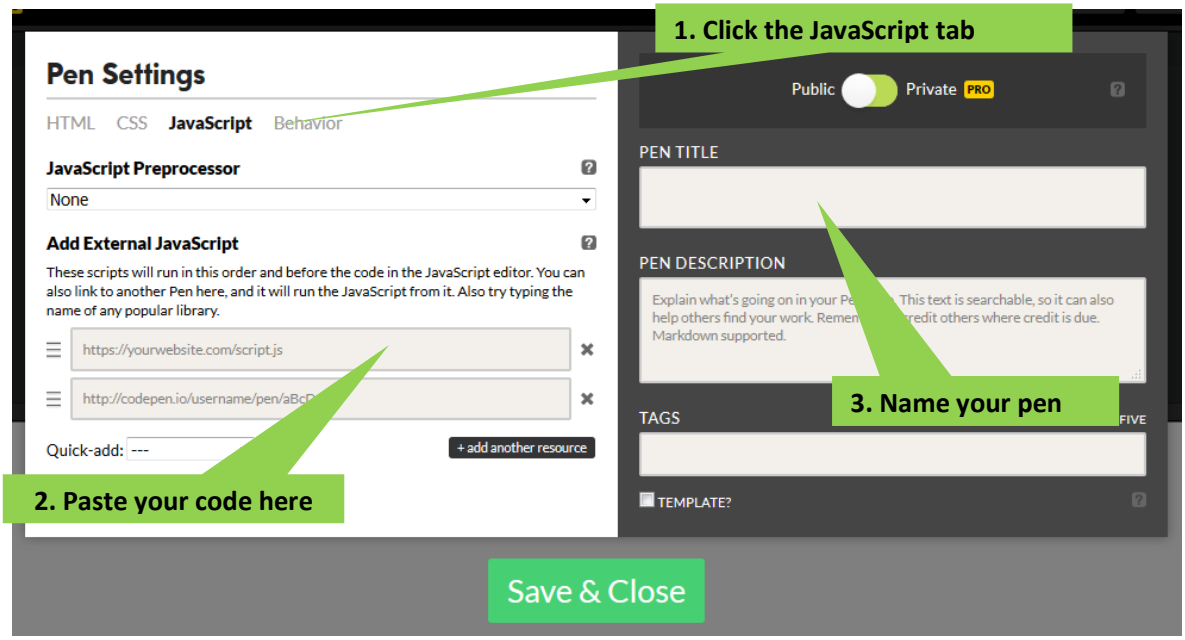
Now that we're familiar with CodePen, we're ready to begin coding, let's start a new "pen" or a new project. Click the "Create a New Pen" button at the end of the tour. For now, we'll be focusing on the JavaScript Section. You can make this section bigger by dragging its frame to the left. We won't be using the HTML and CSS section just yet, so feel free to make the JavaScript section the only visible section.

As we start coding, we'll sometimes make mistakes. CodePen has a feature that looks out for mistakes we might not catch. While you're coding look out for the red error indicator ( ) at the bottom of your code windows. When you see it, it means that CodePen has spotted something wrong. You can click on it to be taken to the error. Don't feel discouraged if you make mistakes, even veteran programmers have to **debug**, or fix, their work!

Linking To Our Libraries

When we use a library or framework, we have to make sure to include it in our page. There are several ways to link code to our webpages, but CodePen has a very convenient way of doing it. From your pen screen, click "Settings" in the top-right corner. When the box pops up, click "JavaScript" and paste the first line of code in the code snippets section of the webpage into the box at the bottom.

You can also name your game and add a description to your pen from this menu. If you ever want to rename it, come back to this menu to change the name or description.



The document we are linking to is a copy of all the libraries we will need to make our game. I've put them all in one document, but CodePen lets you add several.

Setting Up Our P5.Js Structure

P5.js has two main parts that make it work. They are the `setup()` and `draw()` **functions**. Functions are blocks of code that are designed to do a task whenever they are called upon. The `setup()` function is called once at the beginning of the program. We'll put things here that we want to happen only once, like how big we want our game screen to be. The `draw()` function is called over and over again. Our `draw()` function will be called 60 times every second! We'll put things in here that we want to happen over and over, like where our character is moving.

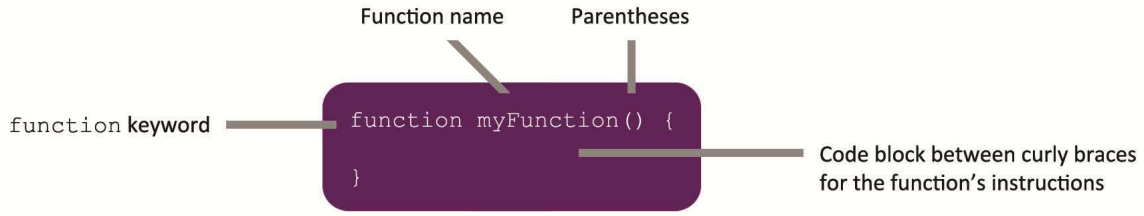
Add this to your JavaScript section.

```
function setup(){

}
function draw(){

}
```

This is what it looks like when we **define**, or create, a function. We first tell the computer that we're about to write a function with the `function` **keyword**. A keyword is a pre-defined word that let's a programming language know something is about to happen. The `function` keyword lets JavaScript know we are about to make create a function. After we've included the `function` keyword, we type the name of the function followed by parentheses. Next, we put curly brackets. These curly brackets tell the computer where our instructions start and stop.



Be sure to put all of the instruction inside of the proper brackets, or the computer may get confused. Computers need to be spoken to very specifically. The specific way we talk to computer is called **syntax**. Every programming language has its own syntax, or rules about how to talk to the computer.

Introducing P5.js

As we found out earlier, p5.js is a JavaScript library. It is full of helpful blocks of code that make things easier for us programmers.

Creating the Canvas

The canvas is the area we'll be drawing in. Let's put one on the screen.

Add the following in your `setup()` function.

```
function setup(){
  createCanvas(840,390);
  background(200, 200, 200);
}
function draw(){
}
// We'll define our own functions below here.
```

These are also functions, they're built into p5.js so we don't have to define them ourselves. They take two and one **arguments**, respectively. Arguments are specific bits of information that the function needs to know to work properly. The `createCanvas()` function needs to know how wide and how tall we'd like the canvas to be. In this case, we'd like the canvas to be 840 pixels wide and 390 pixels tall.

We've also told the `setup()` function that we'd like the background to be gray with the `background()` function.

Did you notice the line after the two slash marks? It's called a **comment**, or a line of code just for the developer that the computer ignores. Developers use code all the time to leave notes to themselves and others reading their code about what the code does and why they made a particular coding position. You should get in the habit of leaving notes. It'll make it much easier to read your code when you come back later. Try leaving yourself notes in the code explaining what it means.

Drawing a Circle

P5.js makes drawing very easy with built-in functions for creating shapes. One of these is the `ellipse()` function. The `ellipse()` function takes four arguments. The `ellipse()` function needs to know where we'd like to draw the circle vertically, where we'd like to draw horizontally, and how tall we'd like the circle to be, and how wide we'd like the circle to be.

Add this in the `draw()` function.

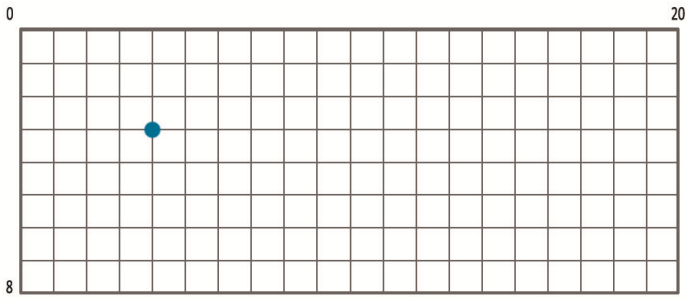
```
function setup(){
  createCanvas(840,390);
  background(200, 200, 200);
}
function draw(){
  ellipse(100,200,30,30);
}
```

We've just told the computer that we'd like to draw a circle 100px in from the left and 200px down from the top. We've also told the computer that we want it to be 30px wide and 30px tall.

Understanding the canvas grid

The first arguments in our ellipse() function tell us about where we'd like our circle to be drawn. Their values correspond to points on the canvas's **grid**. Our particular canvas has a grid that is 840px wide on the X axis (up and down) and 390px wide on the Y axis (left to right). Let's look at what a canvas grid looks like.

Each number represents a pixel in the canvas. The top-left corner is zero on both axes and the value goes up the farther you go to the right and the farther you go down. The grid to the right is only 20px wide and 8px tall. If we define that a circle should be drawn at 4px over and 3px down, we tell it to draw the circle's center at the dot above. Our game's grid is much bigger, but the concept is the same.



Understanding how to define things in relation to the canvas's grid will be important to placing elements on the screen in our game. Try playing around with these four numbers and see how it effects the circle you've made.

Giving the Circle Color

Now that we have a circle, we can give it some color. Computers can read colors in many ways, but we'll be using color that is defined in three parts: red, green, and blue (RGB). The computer will read the numbers we

give it and add a certain amount of red, a certain amount of green, and a certain amount of blue to our circle. The numbers for these colors can be from 0 to 255. Let's see it on our circle.

Add this to your code.

```
function draw(){
  fill(23,55,100);
  ellipse(100,200,30,30);
}
```

The `fill()` function tells the computer what color we'd like inside our circle. We just told the computer we want the circle to have a red value of 23, a green value of 55, and a blue value of 100. The higher the value, the more of a color is in the final color. The higher all the values get, the lighter the color gets. If you set all three values to 255, the color will be white. If you set all three values to 0, the color will be black.

Experiment with the three color arguments and see what happens. Remember: the first value is red, the second value is green, and the third value is blue.

Here's a tip!

There is a shorthand for using grayscale in RGB. Because all the numbers will be the same, you only have to write them once. `(255,255,255)` can be written as simply `(255)` and the computer will know you want all the values to be 255. You can try this out whenever you want a color to be some shade of gray.

Adding a Bit of Movement

Follow-the-Cursor: Built-In Sprites

We know that the `ellipse()` function takes arguments for where we want the circle to be drawn. But what if we wanted the circle's location to change? We can do this with **variables**. A variable is a named value in your program. Whenever you use the name in the program, it's replaced with the value. You could create a variable called `fruit` and give it the value `apple`. Then, if you tell the computer to "display fruit," it will display `apple`.

Variables are like stand-ins for something that isn't set in stone, and they can change. So you could change the value of `fruit` to `pear` and tell the computer to display `fruit`, it will display `pear` instead of `apple`.

P5.js has some variables that it automatically keeps track of. Let's use two of them: `mouseX` and `mouseY`. The `mouseX` variable keeps up with where the mouse is on the X axis and `mouseY` keeps up with where the mouse is on the Y axis. Let's try it. Replace the ellipse's X and Y arguments like it is below.

```
function draw(){
  fill(23,55,100);
  ellipse(mouseX,mouseY,30,30);
}
```

Notice how the `draw()` function draws a circle wherever the mouse is. Even when you move the mouse, the circle follows. It draws a new circle every time it **executes**, or follows the instructions in the brackets. The result is that we see every circle that the `draw()` function ever draws until we reload the page.

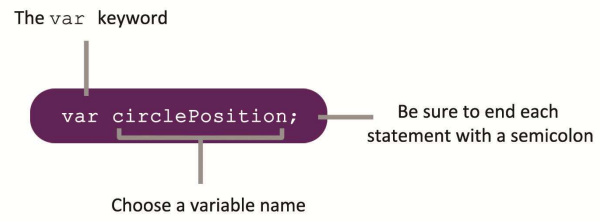
Automatic Movement: User-Defined Variables

So far we've used variables that are built-in. But what if we want to make our own? We can make and edit our own variables by remembering the three parts of using variables: declaration, assignment, and use.

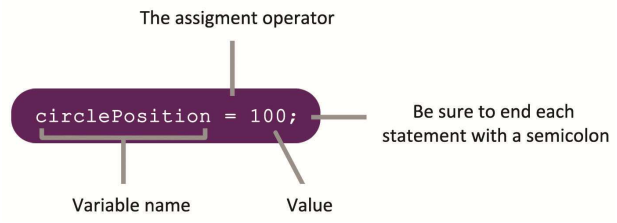
When we **declare** a variable, we just tell the computer that the variable exists. Think of declaring like putting a box on the table and letting the computer know it's there. When we **assign** a value to a variable, we tell the computer what value we'd like to give to the variable. Think of assigning like putting something in the box.

Whenever we use a variable, we're telling the computer we want the value from the box. Let's look at some syntax for declaring and assigning variables.

Declaring a variable



Assigning a value



Let's see this in action. Edit your code like this.

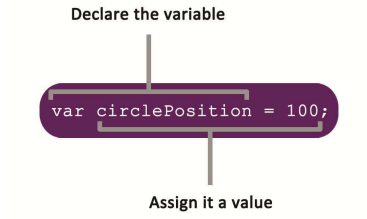
```
var circlePosition;

function setup(){
  ...
  circlePosition = 100;
}
function draw(){
  ...
  ellipse(circlePosition, 200,30,30);
}
```

We declare our new variable `circlePosition` at the very top with the `var` keyword. Then, inside the `setup()` function, we assign a value of 100. Finally, inside the `draw()` function, we tell the `ellipse()` function that we want the circle to be drawn at whatever value is inside the `circlePosition` variable (100). You can change the value of `circlePosition` to whatever you want it to be.

Often, declaration and initiation is done in one step like this. We'll be declaring and assigning values both ways in this lesson.

One-Step Declaration and Assignment



But we said we wanted the circle to move by itself, so let's make that happen. In the `draw()` function, edit your code like this.

```
function draw(){
  circlePosition = circlePosition + 1;
  ...
}
```

This new code tells the computer to add one to `circlePosition` every time the `draw()` function does its thing. This means that the value in `circlePosition` will be just a bit higher than the last time `draw()` finished executing its code, so the circle moves over just a bit more.

Great! Now we've got movement, but what if we don't want that tail left behind? That's a simple fix! Let's just move our `background()` instructions from the `setup()` function into the `draw()` function. Your code should now look like this.

```
function draw(){
  background(200,200,200);
  circlePosition = circlePosition + 1;
  ...
}
```

This will tell the `draw()` function to first paint a gray background, then add one to the total in `circlePosition`, and then paint the circle with a colored fill. The order that we put the instructions in is important.

And look! The circle is moving across the screen! This is because `draw()` updates the value for `circlePosition` then draws a new circle using that value. You can create variables to hold any information you want. Variables are especially good for any information you think might change in the future.

Introduction To P5.play Sprites

Now it's time to start building our game! We'll begin with another function. This one is built into **p5.play**, a library that adds even more features to p5.js. The first thing we'll need is a character to run in our game. The character will be a **sprite**.

Sprites are the main building blocks of p5.play. Sprites are **objects** that are able to store images or animations with a set of **properties**. Think of an object like a bag with other bags inside. Think of properties like the bags inside an object. They store facts about the object that we can edit and refer to, just like we edit and refer to variables. A sprite's properties can change in the same way that variables can change. This will be important for making our sprites behave the way we want them to. Just about everything we make in our game will be a sprite.

Here's a tip!

When a function executes, it goes down the function, following instructions one-by-one. We have to pay attention to what order we put things in because some instructions override others. In our example before, we want the computer to draw a new background before it draws a circle. That's why we put it at the top of the instructions.

Our First Sprite

Let's make our very first sprite in our game. This one will represent our character. We'll call the sprite "runner." Let's clear out our `setup()` and `draw()` functions and then add code like this.

```
var runner;

function setup(){
  createCanvas(840,390);
  runner = createSprite(50,100,25,40);
  runner.depth = 4;
}

function draw(){
  background(200);
  drawSprites();
}
```

Our sprite is just a box right now, but we'll give it some flare shortly. The `createSprite()` function takes four arguments. They tell where you'd like the sprite drawn on the X axis and Y axis, and how tall and wide you want it. Our sprite is drawn at 50px in from the left and 100px down from the top. The sprite is 25px wide and 40px tall. We created the sprite in the in the `setup()` function because we only need the sprite created once.

The Preload Function

Along with the `setup()` and `draw()` functions, p5.js has a function made especially for if you have a lot of images and files to load. This function is called `preload()`. This function loads everything inside it before it executes the rest of the code. This is where we'll tell our program where to find the images we'll be using in our game. Let's add the function to our code like this.

```
var runner;

function preload(){

}
function setup(){
  ...
}
function draw(){
  ...
}
```

Adding Images and Animations

All the images you need for this game are hosted on the accompanying website, *p5.kayesspeadesign.com*.

Select a character, and environment that you'd like for your game. Once you've chosen, click the "Get the Code" button for each and then copy and paste it into your `preload()` function. Don't forget to declare your variables at the top!

```
var runner;
var runningAnimation;
var jumpingAnimation;
var gameBackground;
var platformBackground;
var gameFont;

function preload(){
 [The code you copied from the website]
}
```

This code is telling the computer to load all the images we need to animate our character and store them in the variables `playerRunning` and `playerJumping`. We'll put all the media we need for our game into the preload function so that we can be sure everything is loaded before the game starts.

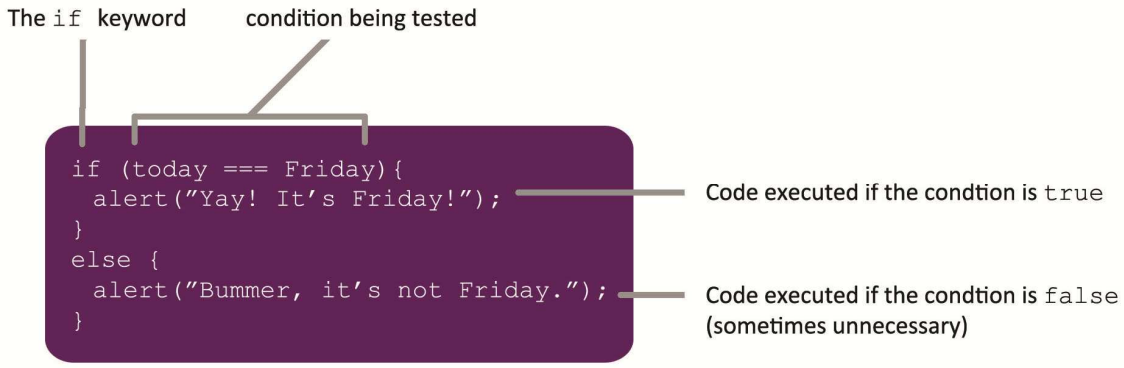
The code we just put in the preload has all the images we'll use to make the character appear to run and jump. Let' add the animations to the character's sprite, kind of like putting the animations in the character's bag. Add this to your `setup()` function.

```
function setup(){
  createCanvas(840,390);
  runner = createSprite(50,100,25,40);
  runner.addAnimation('jump', jumpingAnimation);
  runner.addAnimation('run', runningAnimation);
  runner.setCollider("rectangle", 0,0,10,41);
  runner.depth = 4;
}
```

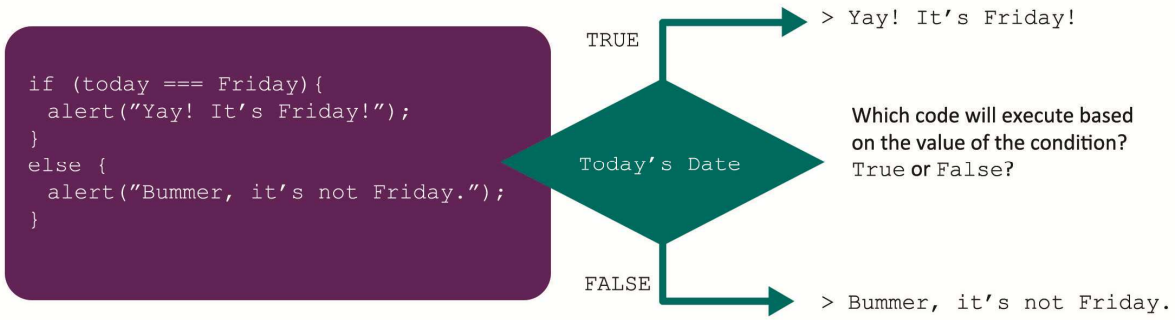
Do you see how the character never stops running? This is because the `playerRunning` animation is set to loop forever. We can stop this, but we don't want to just yet because we want him to just keep running.

Conditional Statements

Sometimes we want to know whether or not something is true before we do an activity. For instance, we may want to know whether or not it is cold outside before we put on a coat and leave our house. This happens in coding, too. When we'd like to check things, we add **conditional statements**—statements that ask if something is true—to our code. We'll be using an `if` statement, a type of conditional statement. `If` statements usually look like this:



`If` statements begin with the keyword `if` followed by parentheses. Inside the parentheses, we put whatever it is we want to check. Then, we add a set of curly brackets with instructions inside. The computer will only do the things inside the brackets if the things inside the parentheses are true.



Condtional statements use **comparison and logical operators**, symbols that say what they are testing. The `if` statement above, for instance, uses three equal signs `===` to show that it's looking for a value that is exactly `Friday`. Below is a chart of comparison and logical operator symbols used to define conditional staments. You'll find yourself using them often when you program.

Comparison Operators			
Operator	Name	Example	Evaluates to
>	Greater than	**5 > 3** 3 > 10 7 > 7	**true** false **false**
>=	Greater than or equal to	**5 >= 3** 3 >= 10 **7 >= 7**	**true** false **true**
<	Less than	**5 < 3** 3 < 10 **7 < 7**	**false** true **false**
<=	Less than or equal to	**5 <= 3** 3 <= 10 **7 <= 7**	**false** true **true**
===	Strictly equal to	**5 === 3** 7 === 7 **7 === "7"**	**false** true **false**
!	Not	**5 !< 4** 12 !== 12 **1 !< 4**	**true** false **false**
!==	Not strictly equal to	**5 !== 3** 7 !== 7 **7 !== "7"**	**true** false **true**

Let's see an `if` statement in action. Edit your `draw()` function like this.

```
function draw(){
  if(!gameOver){
    background(200);
    drawSprites();
  }
  if(gameOver){
  }
}
```

Okay, here are two new conditional statements. The second one asks if `gameOver` is true. Notice how the first conditional statement has a `!` (called a bang) just before `gameOver`. A bang means that whatever is behind it is the opposite. So the second statement is asking if gameOver is true while the first statement is asking if gameOver is not true. See it in the comparison operators chart? Note: The background function has moved to *inside* the `!gameOver` condition. We only need that background while the game is still being played.

But how will the computer know if game over is true or not? We'll tell it! Let's create a new variable at the top and initialize its value to `false` In `setup()`. `True` and `false` are called a **Booleans**, or values that are either `true` or `false`. Booleans are great for setting conditions that have only two options.

Define and assign the variable like this at the top of your code.

```
var gameOver = false;
```

What happens if you set it to `true` instead? The animations stop. This is because we've only told the computer to draw sprites if `gameOver` is set to `true`.

Making Our Own Functions

We've used functions that were already and part of p5.js and p5.play. But just like variables, sometimes we'll need to make our own. We've already seen how functions look, but now we'll be creating some for ourselves.

Remember, functions begin with the `function` keyword and then the function's name followed by parentheses. Remembering this syntax is important to making our functions work. If we didn't include the function keyword or didn't put the parentheses, the computer would become confused about what we wanted it to do.

Here's a tip!

Have you noticed that almost every line we type ends in a semicolon? This is a part of JavaScript's syntax. JavaScript pays no attention to spaces or line breaks, so semicolons are its signal that a command is over.

Adding Platforms

So far, we've got a character running in place, but what will our character run on? We'll need some platforms. Let's create a new **group**. In p5.play, groups are **arrays** that hold sprites. Arrays are very similar to objects, the main difference between the two is that the things inside of an array go in a specific order. We use arrays when we want to keep all the things inside in order. Instead of them being bags with contents loose inside, arrays are like orderly boxes. Their content has a specific order and a specific place. We'll use a group array to hold all the platforms we'll be making. Edit your `setup()` function like this.

```
var platformsGroup;
function setup(){
  ...
  platformsGroup = new Group
}
```

We've just told the computer to make an array called "platformsGroup" to hold our platforms. Let's add a `function` with an `if` statement that will add platforms to our game. We'll add the function just below the `draw()` function. Edit your code like this.

```
var currentPlatformLocation;

function setup(){
  ...
  currentPlatformLocation =  -width;
}

function draw(){
  ...
}
function addNewPlatforms(){
  if(platformsGroup.length < 5){
    var currentPlatformLength = 1132;
    var platform = createSprite((currentPlatformLocation * 1.3),
random(300,400), 1132, 336);

    currentPlatformLocation += currentPlatformLength;
    platform.addAnimation('default', platformBackground)
    platform.depth = 3;
    platformsGroup.add(platform);
  }
}
```

We've just declared a new function that uses an `if` statement to determine when we'd like to add a new platform. The function only executes if the amount of platforms in the `platformsGroup` is less than 5. If at any time there are five or more platforms, the function will not execute its instructions.

We've given the computer a lot of instructions inside this function. Let's list them in order using **pseudocode**. Pseudocode is notation that somewhat resembles code, but is not necessarily in computer-readable language. We use pseudocode when designing programs because it helps us to think about what we'd like to tell the computer before we focus on syntax. Here is the `addNewPlatforms()` function in pseudocode.

```
If there are less than five platforms inside of platformsGroup{
      1. The value for 'currentPlatformLength' is 1132.
      2. Create a platform sprite and put it in the variable 'platform'
            a. It should be 1.3 times as far from the left as the one before
               it
            b. It should be anywhere between 300 and 400 pixels down from the
               top
            c. It should be 1132 pixels wide and 336 pixels tall
      3. Add the value of currentPlatformLength to the value of
         currentPlatformLocation
      4. Add an animation to the platform sprite called default that uses the
         platformBackground variable
      5. Put the platform at level 2 in layer order
      6. And finally, add the new sprite to the platformGroup array.
```

Did you notice the strange notation when we told the computer to add the value of `currentPlatformLength` to the value of `currentPlatformLocation`? This is called a **compound assignment operator** and it is a shorter way of saying "don't erase the value in this variable, just add another value to it." There are other compound assignment operators, not just for addition. You can subtract, multiply, divide and more, but we'll only be using the compound assignment operator for addition in our game. In

programming, the need to add to the total inside of a variable comes up a lot, so much that this shortcut was made so programmers could code it easier.

Let's see this concept in practice. If you were to write something like this…

```
var total = 10;
total += 5;
```

The value would be 15, not 5. We'll be using this operation more as we continue to make our game. Remember how we used the long form of this operation when we first made the circle move? You'll find that you use it often when you code your own programs.

Global vs Local Variables (Scope)

Notice that we declare and initialize `currentPlatformLength` at the same time inside of the `addNewPlatforms()` function. This is because we only need the variable inside the function. The variables we are declaring at the very top are called **global variables**, or variables that are accessible to all of the program's code. The variable `currentPlatformLength` is only accessible to the code inside of the `addNewPlatforms()` function. It is a **local variable**, or a variable that is only available in the code block it was declared in. We make decisions on where to define variables based on if we want all the code to have access to it or if we just want some of the code to have access to it. The rules that determine which functions have access to a variable is called the variable's **scope**.

We don't need to use `currentPlatformLength` anywhere but inside of the `addNewPlatforms()` function, so we define it inside there. We'll need variables like `currentPlatformLocation` in a few functions, so we make sure to declare it on the outside, so all the functions can use it.

Also notice that we've initialized `currentPlatformLocation` as `–width.` The variable `width` is a built-in variable in p5.js. It is the width of our canvas. There is also a `height` variable that we'll be using later. Here, we've told the computer that the value for `currentPlatformLength` is the negative of whatever our canvas's width is. This way, the first platform will be generated under the runner with lots of space for him to land.

Calling Functions

You may have noticed that we've been including functions by typing their names followed by parentheses wherever we want to execute them. This is called **calling** a function, or instructing the computer to use the function, along with any arguments the function needs. We've called the `ellipse()`function, the `creatCanvas()` function, and others. Now we'll call the `addNewPlatforms()` function we just wrote.

Edit your `setup()` function like this.

```
function draw(){
  ...
if(!gameOver){
    background(200);
    addNewPlatforms();
    drawSprites();
  }
}
```

Now the `draw()` function's instructions include all of the instructions inside of the `addNewPlatforms()` function! We could have just written everything directly into the `draw()` function, but putting it into a separate function makes it easier to use the instructions in multiple places and move the instructions around.

Adding Physics

We've got some basic building blocks of our game: platforms and a runner. But our runner is just floating there, running on nothing. Let's add some gravity and jumping power to make the game really work!

Sprite Velocity

In p5.js, every sprite has a property called velocity. The velocity determines if a sprite should move when the `draw()` function executes. The velocity property has two parts: X and Y. These correspond the sprite's position on the X axis and the Y axis. If a runner has an X velocity of 1, it will move one pixel to the right every time the draw function executes. We saw something similar to this when we made our little circle move across the screen at the beginning of the lesson.

In order to simulate gravity, we will use the velocity property of our `runner` sprite. Edit your code like this.

```
var gravity;

function setup(){
  ...
  gravity = 1;
}

function draw(){
  if(!gameOver){
    background(200);
    runner.velocity.y += gravity;
    drawSprites();
  }
}
```

Above, we declare gravity as a global variable. Then we initialize gravity with a value of 1. Then, we tell the draw function to add to our runner's X velocity by a rate of whatever the value of gravity is (1). This way the characters falls faster the longer he falls, similar to how objects fall in real life.

Collision Detection

Great! Now our character has the force of gravity pulling him down. But he just fell right on behind the platform! We need to add what's called **collision detection**. Collision detection is the process by which a computer detects the intersection of two or more objects, or how the computer knows when objects are touching.

Collision detection can sometimes be very complicated to code, and there are many ways to do it. Fortunately for us, we're using p5.play! P5.play has built-in collision detection that is great for making games. Let's add it to our code. Edit your `draw()` function like this.

```
function draw(){
  if(!gameOver){
    ...
    runner.collide(platformsGroup, solidGround);
    addNewPlatforms();
    drawSprites();
  }
}
```

Methods and Callbacks

Methods

In p5.play, `collide()` is a **method** that all sprites have. `Collide()` detects if the sprite collides with a particular sprite or group of sprites. A method is a function that belongs to a particular object. Remember our bag analogy? Methods are functions that live inside of one of the bags in the object and can only be called if

you're using it on that specific object. P5.play makes sure that every sprite we create can use the `collide()` method.

Let's take a look at the arguments that the `collide()` method needs from us. The first argument is the sprite or group of sprites we want to detect collisions with. We've told the computer that we want to detect any collisions `runner` makes with any sprites in the `platformGroup` array.

Callbacks

The second argument is special. It is the name of a function we want to call whenever the computer detects a collision between `runner` and `platformGroup.` When a function or method is used as an argument in another function or method, it is called a **callback**. Basically, we use callbacks when we want to "call back" to a function when a particular thing happens. The calling is always done by another function. Callbacks have lots of uses in programming, but we're using this one to tell the computer what we want to happen the moment `runner` and `platformGroup` collide. We want it to execute the `solidGround()` function.

So now the computer knows that we want it to run `solidGround()` as soon as it detects a collions. But we haven't defined that function yet. No worries, we'll do it now! First, let's think about what a function like `solidGround()` would need to tell the computer to do. Let's write it out in pseudocode.

```
1. Stop the runner sprite from moving down.
2. Change the animation to make the runner look like he's running.
3. Check if the runner sprite has anything touching it on the right side,
   because that would mean it hit a wall.
      a. If it does, make sure the runner sprite doesn't move forward.
      b. Also, make sure the runner sprite starts to falls
```

These instructions will be just what our game needs to make sure our character interacts with the platforms correctly. Now let's code it. Add this function just after the `addNewPlatforms()` function at the bottom of your code.

```
function addNewPlatforms(){
  ...
}
function solidGround(){
  runner.velocity.y = 0;
  runner.changeAnimation("run");
  if(runner.touching.right){
    runner.velocity.x = 0;
    runner.velocity.y+= gravity;
  }
}
```

Giving the Character Movement

Jumping

So now our character is running in place on the platform. But we know he'll need to jump. Let's add some keyboard inputs so that when we hit a specific key, he jumps. We'll put the instructions in a function called `jumpDetection()` As usual with when we design functions, let's think about it in pseudocode first so we make sure we get the behavior right.

```
If someone presses the UP_ARROW key…
   1. Change the animation so that it looks like he's jumping
   2. Once the animation has been played once, go back to the beginning, stop
   3. Change the runner's velocity on the X axis by a certain amount (we'll
      define a variable for his jumping power).
```

This function will allow our character to jump. We're going to use a built-in p5.play method called `keyWentDown()`. This function detects if someone pressed a key and returns a value of true or false. By putting it in the `draw()` function with the key we are watching for (UP_ARROW) as an argument, we tell the computer to check for the UP_ARROW being pressed every time it the function executes. Add this to your code.

```
var jumpPower = 15;

function draw(){
  if(!gameOver){
    ...
    jumpDetection();
    drawSprites();

  }
}

function jumpDetection(){
  if(keyWentDown(UP_ARROW)){
    runner.changeAnimation("jump");
    runner.animation.rewind();
    runner.velocity.y = -jumpPower;
  }
}
```

Great! Now when we press the UP_ARROW key, our character jumps! Notice how his animations change to a jumping animation when he's in the air, but go back to a running animation once he lands on a platform. This is because we already specified this behavior in our `solidGround()` function.

Running

So far our character has been running in place. Let's make him run for real. (We are building an infinite runner after all!) Adding forward velocity will be easy. It just takes a couple lines of code. Edit your `draw()` function like this to get your character moving.

```
var runnerSpeed = 15;

function draw(){
  if(!gameOver){
    runner.velocity.y += gravity;
    runner.velocity.x = runnerSpeed;
    ...

  }
}
```

Using the Virtual Camera

Whoa! Our character ran right off of the screen! We need to make sure we can see him at all times. This is where the p5.play virtual camera comes into play. The virtual camera takes care of scrolling and zooming for scenes extending beyond the canvas. Telling the camera to follow our character will be easy. We'll just need one line of code. Edit your `draw()` code like this.

```
function draw(){
  if(!gameOver){
    ...
    camera.position.x = runner.position.x + 300;
    drawSprites();
  }
}
```

Garbage Collection

Now we have a somewhat playable game. But there's another problem. We don't have enough platforms. Our `addNewPlatforms()` function only generated five platforms and stopped. We need to make more platforms. Since we don't want to have an infinite number of platforms clogging up our computer's processing power, we'll have to do something called **garbage collection**.

Garbage collection is the process of collecting and removing objects that are no longer being used by a program. Garbage collection is important to almost all programming applications. Our garbage collection will focus on removing platform sprites once they are no longer being used. We'll write a function called `removeOldPlatforms()` to take care of this for us. Time for some pseudocode!

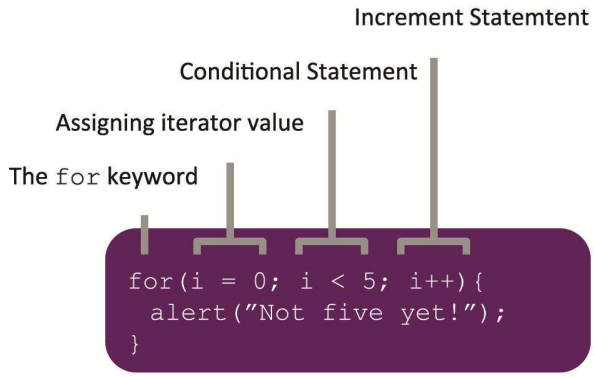
```
Check every platform in platformGroup
  If any of them have already been passed up
   1.   Remove the platform from the game.
```

For Loops

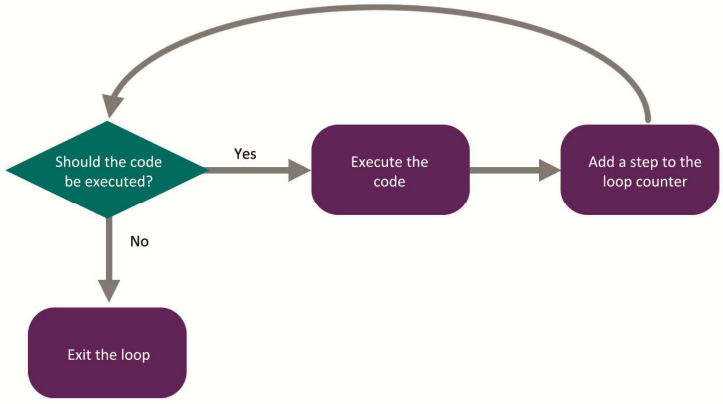
We want the computer to check every platform in the `platformGroup` array. When we want to go over every item in a certain array or we want to do something over and over again, we use **loops**. A loop is a series of instructions that are repeated until a certain situation is met. In our loop, we will repeat the command to check to see if the platform has been passed. We'll repeat it until we run out of platforms to check.

The particular type of loop we will use to do this is called a **for** loop. A `for` loop is used to repeat a section of code certain number of times. Sometimes you know exactly how many times you want it repeated. Sometimes it is the computer that knows how many times, not you. But either way, there's a definite stopping point for the loop.

Let's look at what a for loop looks like.



These loops begin with the `for` keyword followed by parentheses. Inside of the parentheses we had some instructions. Let's look at how `for` loops work.



The above loop will first check if the variable `i` is less than 20. If it is, the loop will execute the code. When it finishes executing, it will add 1 to the value of `i`. Let's add a function that contains a `for` loop that will create platforms for us. We'll add it just below the draw function.

```
function draw(){
  if(!gameOver){
    ...
    removeOldPlatforms();
    drawSprites();
  }
}
function removeOldPlatforms(){
  for(var i = 0; i<platformsGroup.length; i++){
      if((platformsGroup[i].position.x) < runner.position.x-width){
        platformsGroup[i].remove();
    }
  }
}
```

Here's a tip!

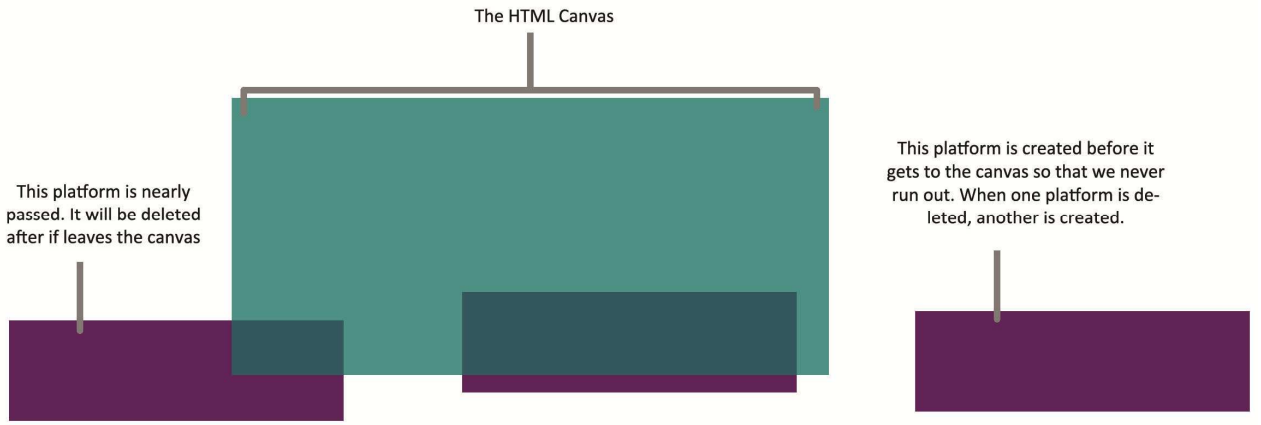
The variable we've defined as `i` can have any name you want, but most people use 'i' which is short for 'iterator.' This is what's called a coding **convention**, or a way that most people do it, even though there are no rules saying you have to. We use conventions to help us understand each other's code better.

Our `for` loop tells the computer, "For as long as the iterator is less than the amount of things inside `platformGroup`, check to see if the platform's position is at least as far away from the runner as the width of the canvas. When you find one that is, remove that particular platform from the group."

Notice how we put `i` in brackets following `platformGroup`. This is because putting a number in brackets after the name of an array accesses the item in that position on the array. Because `i` has a value of 0, and a value of one higher every time the loop executes, by putting `i` in the square brackets, we are asking the loop to check each platform one by one.

So with our loop deleting platforms that have been passed up, the total number of platforms falls below 5. This triggers our `addNewPlatforms()` function to add a new platform. The cycle goes on forever with one function deleting and the other function creating. Now we have infinite platforms without slowing down our game.

Don't forget to call the function in the `draw()` function!



Background Art

So far the background has been blank; let's fix that. We'll add the background in looping panels the same way we added our platforms. We'll change some values to match the background better. For instance, we won't be using a random number for the sprites' locations. We know that we always want the background in the same place. We'll also be using a different width for the background sprites because the image we'll be using has a different background.

Just like we deleted the old platforms, we'll need to delete background sprites that have been passed up. So we'll be adding two new functions: `addNewBackgroundTiles()` and `removeOldBackgroundTiles()` Declare your variables and call the functions like this.

```
var currentBackgroundTilePosition;
var backgroundTiles;

function setup(){
  ...
    backgroundTiles = new Group;
    currentBackgroundTilePosition = -width;
}

function draw(){
  if(!gameOver){
    addNewBackgroundTiles();
    removeOldBackgroundTiles();
    drawSprites();
  }
}
```

Then add the two functions like this. Can you see how they are very similar to the plaform functions?

```
function addNewBackgroundTiles(){
  if(backgroundTiles.length < 3){
    currentBackgroundTilePosition += 839;
    var bgLoop = createSprite(currentBackgroundTilePosition, height/2, 840,
390);
    bgLoop.addAnimation('bg', gameBackground);
    bgLoop.depth = 1;
    backgroundTiles.add(bgLoop);
  }
}
function removeOldBackgroundTiles(){
  for(var i = 0; i < backgroundTiles.length; i++){
      if((backgroundTiles[i].position.x) < runner.position.x-width){
        backgroundTiles[i].remove();
    }
  }
}
```

Adding a Lose Condition

So far, if our character fell below the game screen, he just kept falling. But now it's time put some functionality into the `if(gameOver)` section of our code! We want to show the player that the game is over and allow them to start over without refreshing the page. Let's think about how we'd make this happen.

Checking for Falls

The first thing we would need is to tell the computer is what to do if `runner` falls too far down.

```
If the runner sprite falls below the canvas
   1.   The game is over; set gameOver variable to true
```

This is what that function will look like. We'll name the function `fallCheck()`

```
function draw(){
  if(!gameOver){
    ...
    fallCheck();
    drawSprites();
  }
}
function fallCheck(){
  if(runner.position.y > height){
      gameOver = true;
  }
}
```

Adding Game Over Text

Next, we need to tell the person playing the game that the game is over. We'll use p5.js text objects for this.

1. Darken the canvas
2. Display in large letters, "Game Over"
3. Display in smaller letters, "Jump to restart"

This is what that function would look like. We'll name the function `gameOverText()`. We'll also tell the computer to stop drawing sprites.

```
Function draw{
  if(gameOver){
    gameOverText();
    updateSprites(false);
  }
}

function gameOverText(){
    background(0,0,0,3)
    fill('white');
    stroke('black')
    textAlign(CENTER);
    textFont(gameFont);

    strokeWeight(2);
    textSize(90);
    strokeWeight(10);
    text("GAME OVER", camera.position.x, camera.position.y);

    textSize(15);
    text("Jump to try again", camera.position.x, camera.position.y + 100);
}
```

When defining text in p5.js we have to remember to be very aware of the order we define things in. The rules for `stroke()`, `fill()`, and `textSize()` effect any text that follows them. So when we want to change those values, we have to be sure to only put the things we want to change beneath them. Otherwise, the text would look different than we intended.

Restarting the Game

Now we need to write a function for restarting the game after the player loses.

```
If the player presses the UP_ARROW key
   1.   Remove all the platforms
   2.   Start animating again
   3.   Reset the runnerSpeed variable
   4.   Reset the runner's position
   5.   Reset the location of the platforms
```

This is what that function would look like, we'll call it `newGame()`.

```
function newGame(){
  if((keyWentDown(UP_ARROW)) ){
    platformsGroup.removeSprites();
    backgroundTiles.removeSprites();
    gameOver = false;
    updateSprites(true);
    runnerSpeed = 15;
    runner.position.x = 50;
    runner.position.y = 100;
    runner.velocity.x = runnerSpeed;
    currentPlatformLocation = -width;
    currentBackgroundTilePosition = -width;
  }
}
```

This function resets any variables that may have changed while someone was playing the game. It clears all the sprites we no longer need, and starts the game over again. Be sure to call it in the `if (gameOver)` section.

```
Function draw(){
  if(gameOver){
    gameOverText();
    updateSprites(false);
    newGame();
  }
}
```

Adding the Finishing Touches

Point Counter

Most people want a way to keep up with their score when they play a game. In our game, progress will be measured in yards. We'll need a function that keeps up with how many yards our little character has run before falling and displays it to anyone playing the game. Edit your code like this.

```
var playerScore = 0;

function draw(){
  if(!gameOver){
    drawSprites();
    updateScore();
  }
}
function updateScore(){
  if(frameCount % 60 === 0){
    playerScore++;
  }
  fill('white');
  textFont(gameFont);
  strokeWeight(2);
  stroke('black');
  textSize(20);
  textAlign(CENTER);
  text(playerScore, camera.position.x + 350, camera.position.y + 160);
}
```

This function tells the computer to add one to the playerScore every 30 frames. And because the draw() function creates a frame 60 times every second, players get one point every second the character is still running.

Updating Previous Functions

Now that we have a `playerScore` variable, we can display it at the end of the game at the game over screen. Edit your `gameOverText()` function like this.

```
function gameOverText(){
  fill('white');
    ...
    textSize(20);
    text("You ran " + playerScore + ' yards!', camera.position.x,
camera.position.y + 50);
}
```

We'll also need to reset it when a player restarts the game. Edit your `newGame()` function like this.

```
function newGame(){
  if((keyWentDown(UP_ARROW)) ){
    ...
    playerScore = 0;
  }
}
```

Loading Screen

So far, our game has been displaying the default loading screen that comes with p5.js. Adding our own loading screen is very easy, we'll just need to add a few lines of code to the HTML and CSS sections of our pen. Edit your HTML like this.

```
<div id="p5_loading" class='loading-screen'><p>LOADING</p></div>
```

This tells p5.js that you would like to display something other than the default loading screen. Edit you CSS section like this to tell the computer exactly how you'd like the loading screen and webpage to look.

```
body {
  background: #777;
}
canvas {
  position: absolute;
  top: 50%;
  left: 50%;
  -webkit-transform: translate(-50%, -50%);
          transform: translate(-50%, -50%);
}
.loading-screen {
  background: black;
  min-width: 840px;
  min-height: 390px;
  color: white;
  position: absolute;
  top: 50%;
  left: 50%;
  font-size: 2rem;
  -webkit-transform: translate(-50%, -50%);
          transform: translate(-50%, -50%);
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-pack: center;
      -ms-flex-pack: center;
          justify-content: center;
  -webkit-box-align: center;
      -ms-flex-align: center;
          align-items: center;
  align-text: center;
}
.loading-screen p {
  -webkit-animation: pulse 1s linear 2s infinite alternate;
          animation: pulse 1s linear 2s infinite alternate;
}
@-webkit-keyframes pulse {
  0% { opacity: 0; }
  100% { opacity: 1; }
}
@keyframes pulse {
  0% {
    opacity: 0;
  }
  100% {
    opacity: 1;
  }
}
```

Further Challenges

You've completed your game! Congratulations! If you're still looking for a challenge, how about these additions? You can find some solutions (though there is always more than one way to solve a code problem) on the website.

Progressive Difficulty.

Right now, our character runs at a steady speed. But what if we wanted him to run a little faster as time goes on? We know how to add to a variable, and we know that our player's speed is a variable. So how would you add to the speed variable as time went on? Tip: `runnerSpeed` doesn't need to change much for you to notice the effect. Try increasing it by very small numbers.

Mobile-Friendly Play

We built this game for computers with keyboards. But suppose you want to show your friends on the run? P5.play has a function call `touchStarted()` that works a lot like `keyWentDown()` but detects touchscreen touches. How would you use `touchStarted()` to make your game take touchscreen inputs?

More sound effects

We've already added a song to our game. But what if we wanted to add other sounds to give the player more feedback? How would we add jumping and game over sounds to our game?

Resources for Further Learning