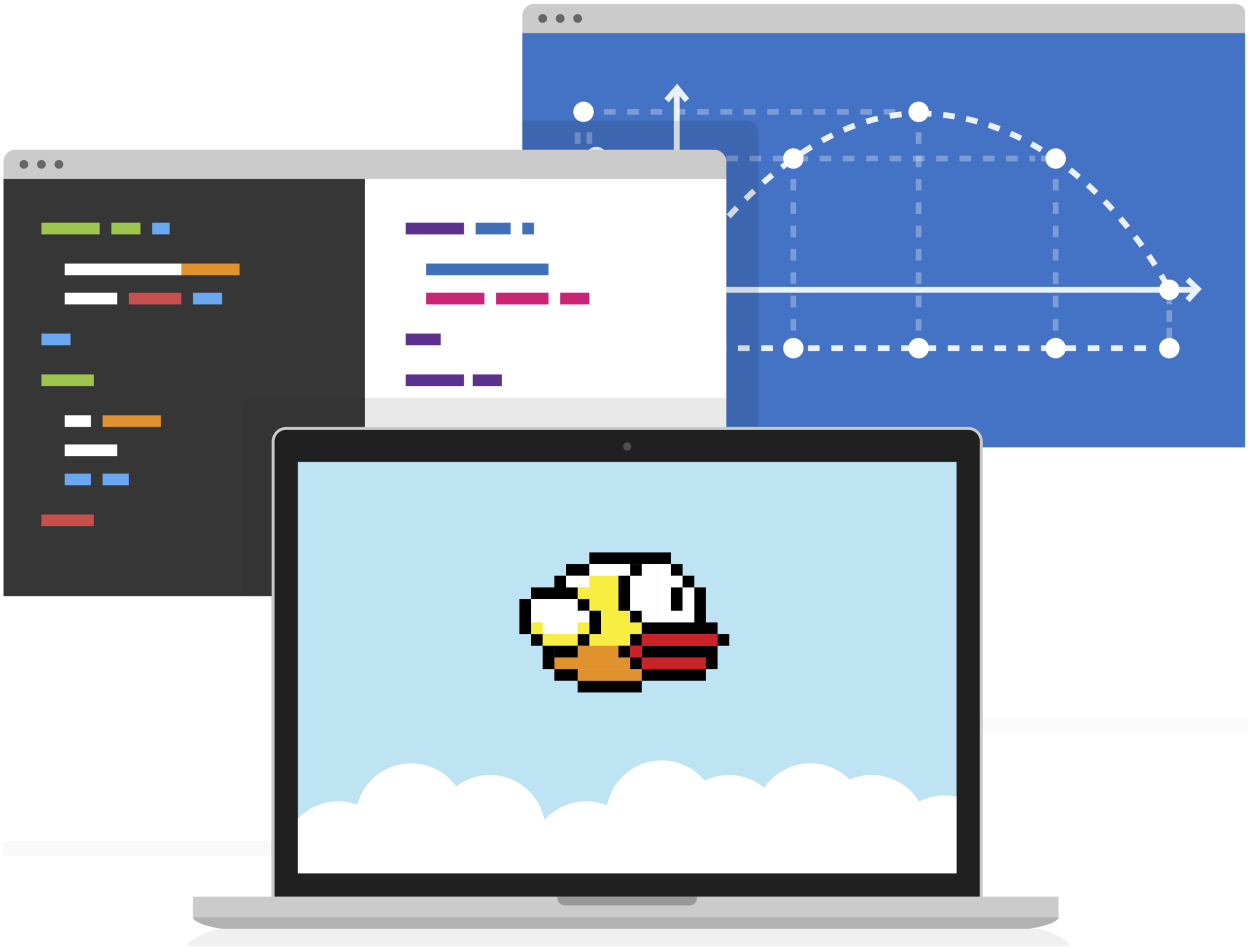
CAMBRIDGE
CODING ACADEMY

BUILD A GAME IN A DAY_



Build a Game in a day

by Cambridge Coding Academy Limited

Cambridge Coding Academy Limited 71-75 Shelton Street
Covent Garden
London
WC2H 9JQ
Email: contact@cambridgecoding.com

Contents

There are five modules in the Flappy Bird Workshop, each introducing one or two core programming concepts.

Set up

Start the day by going through some simple setup steps to. Don't proceed to Module 1 until you're done with this!

Module 1

Statements, functions and arguments - how to give instructions to computers.

Module 2

Event handling and user interaction - handling user input.

Module 3

Variables and objects - tracking changes to data.

Module 4

Loops and conditionals - controlling when commands are executed and how many times.

Module 5

Game physics and advanced event handling - emulating physical interactions and responding to events within the game.

Appendices

Complete code examples for each module.

Initial set-up 1

Here are instructions on how to download the code from the web to your machine. You will need to follow these steps before you can start to learn about coding with us.

Forking the project on github

Github is a website where you can store code and exchange it with others. It is very useful for collaborating with people on projects. We stored some code there and you need to pick it up. The first step is to *fork* it to your account. Forking a project just means copying it in your own github account.

Go on ***https://github.com*** and log in. Go to ***https://github.com/cambridgecoding/webapp-flappy*** (1) and click on "Fork" (2).

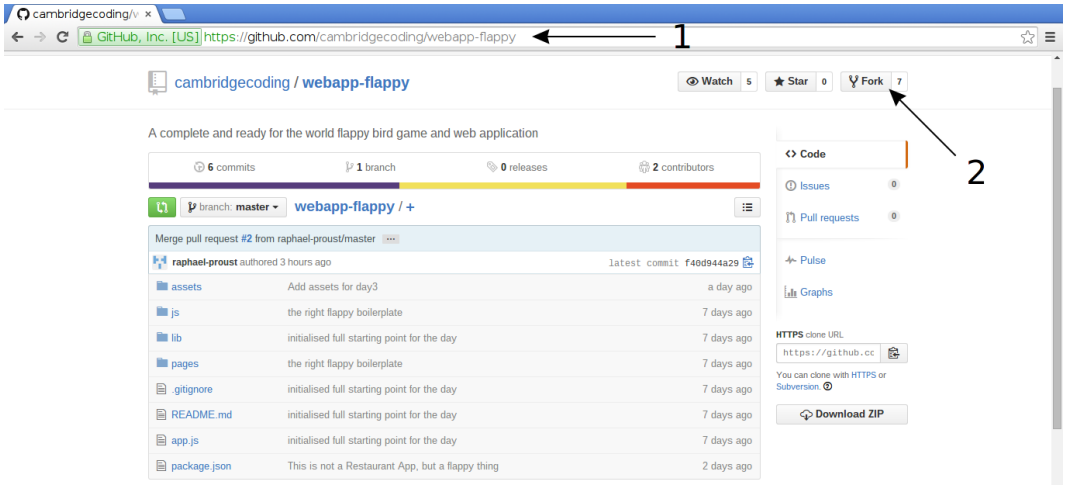


FIGURE 1-1

Forking on Github

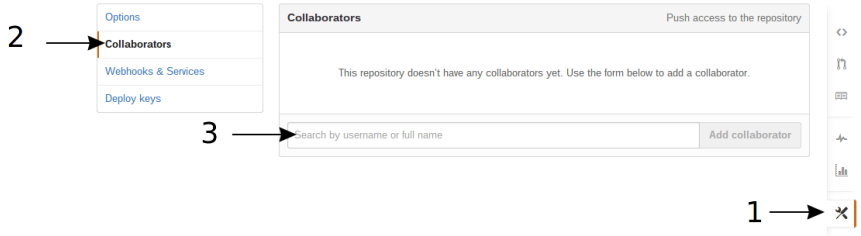
Adding collaborators

You will be working in pairs. That means you will be sharing the code with someone. In order to make that easier, you can add each others as collaborator of your project on github.

Go on your project page on github (the page you arrive to when you fork): `https://github.com/yourGithubLoginHereInstead/webapp-flappy`. Then, choose the setting icon on the right-hand side (1), choose "Collaborators" on the left-hand side menu (2), and type your pair's github login (3).

FIGURE 1-2

Adding Collaborators on Github



Opening webstorm

Open Webstorm. The aim is to reach the screen below.

FIGURE 1-3

Starting Webstorm

If Webstorm asks you to register, choose the free trial option. If Webstorm has already opened a project, you can go back to the project selection screen by closing the opened project from the menus.

Cloning the project on your machine

You can now get the code from your own github account to the machine you are working on. From Webstorm, choose “Check-out from Version Control” (1) and then “GitHub” (2).

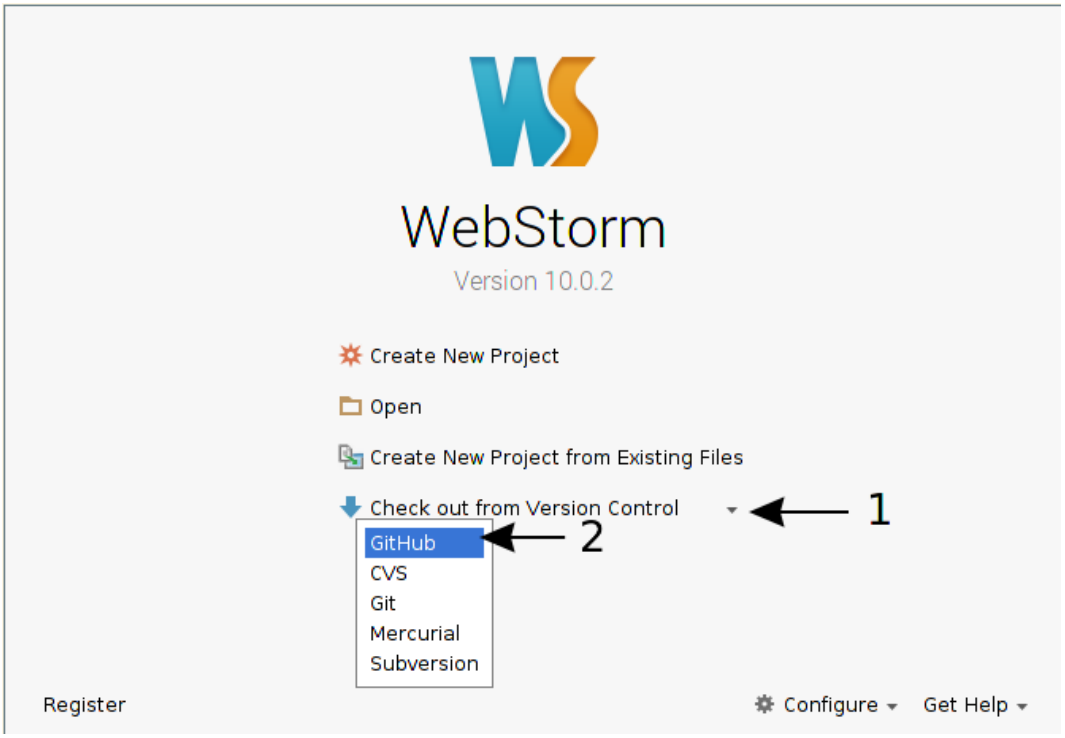


FIGURE 1-4

Choose "github"

Webstorm will walk you through the different steps of cloning. First enter your Github credentials and choose login.

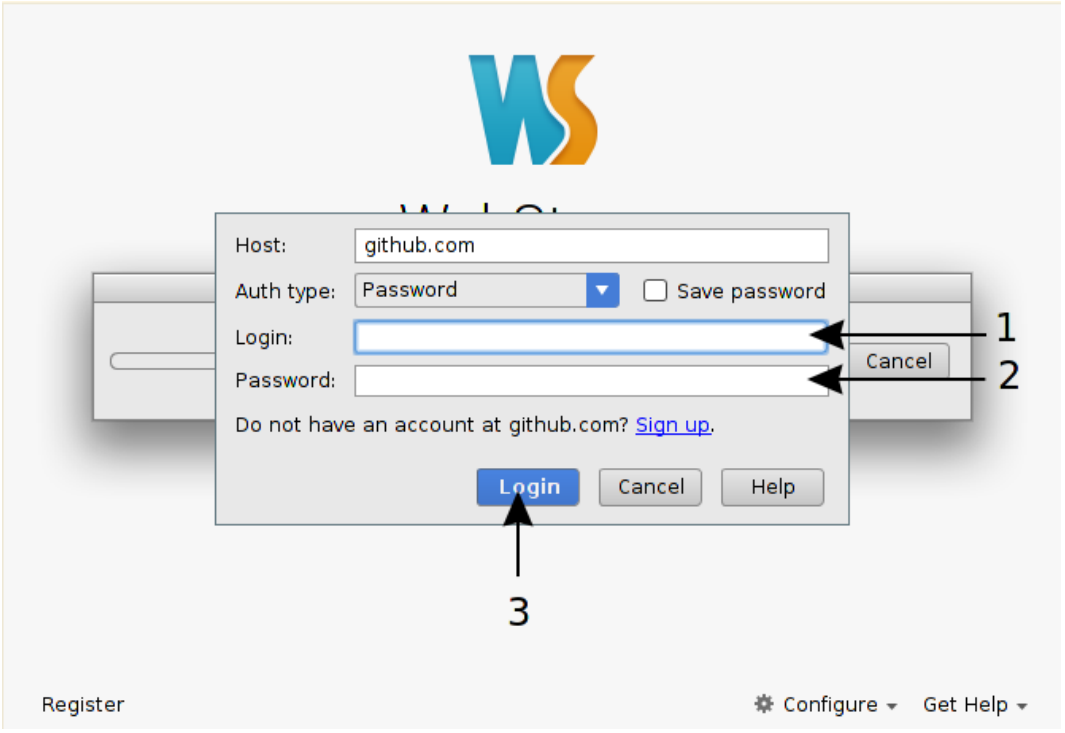


FIGURE 1-5

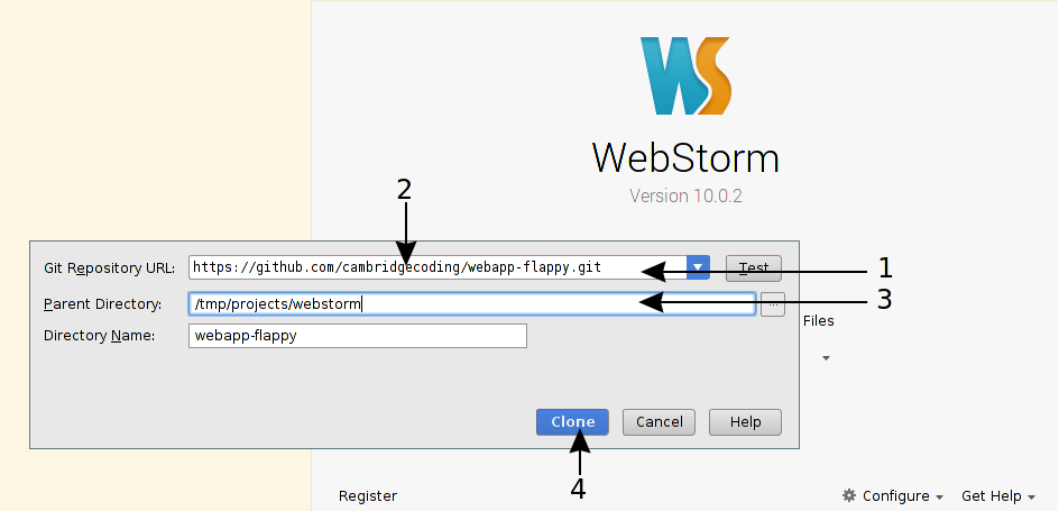
Enter github credentials

Then choose the “webapp-flappy.git” repository (1). Your account name will appear instead of (2) “cambridgecoding”. Make sure the parent directory (3) exists (otherwise create it). Finally, choose “Clone” (4).

Webstorm will ask you for a master password. After that, you will not need to enter your Github credentials again. Next time you push (you will learn what it is later) you will need to re-enter this password instead.

FIGURE 1-6

Clone your repository



Webstorm may ask you to set up a master password. This avoids having to retype your github password every time you want to synchronise the code.

Webstorm will then open to its main interface.

Webstorm’s interface

Here is a very quick introduction to Webstorm interface. Not everything will make sense right now: it’s okay, it’ll all make sense very soon.

The main interface has part for editing text (1) and another for browsing files (2).

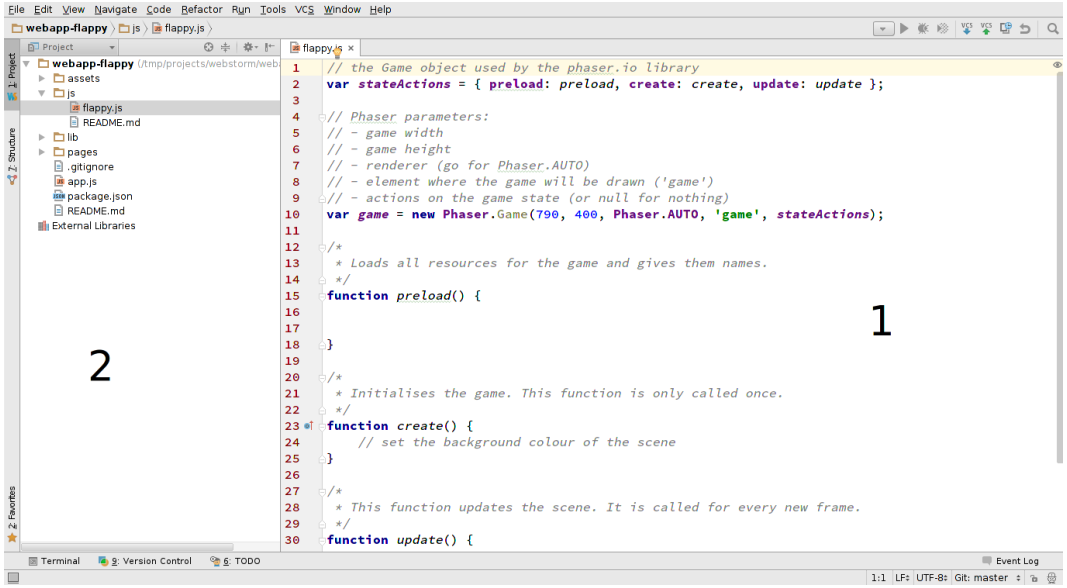


FIGURE 1-7

Main Webstorm interface

Features

Webstorm is more than a type writer. It does several things to make your life as a programmer easier.

1. Extra characters: Webstorm will try to guess and type some characters before you. For example, if you open a parenthesis ((), Webstorm will add a closing parenthesis ()). Try to look at the screen more than the keyboard and you will get used to it faster.
2. Auto-completion: Webstorm will try to finish words for you. When you start typing the name of a variable (you will learn what those are later) Webstorm will display a list of candidates for completion: use the arrows to select the one you want and Tab to validate your choice. (Or you can just finish writing the whole thing by hand.)

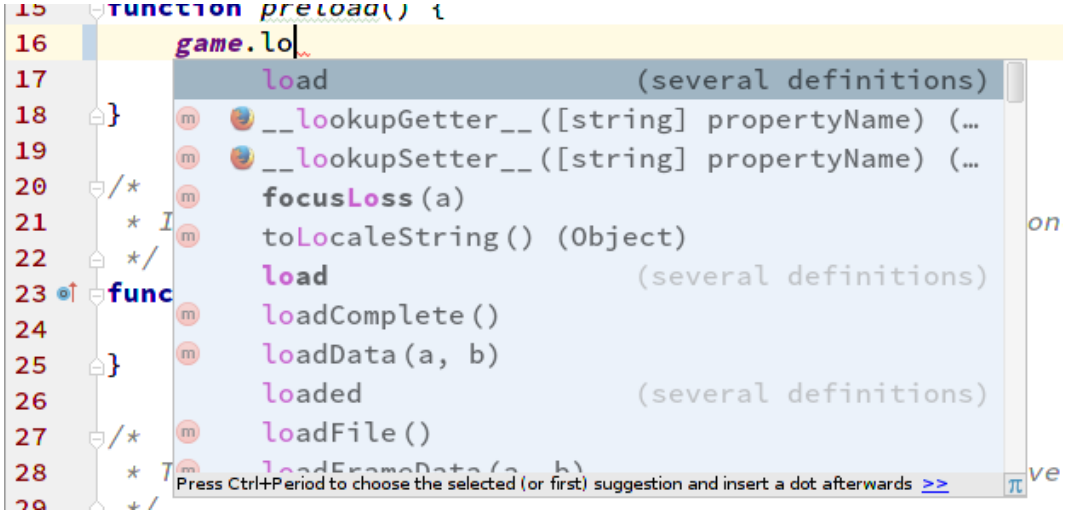


FIGURE 1-8

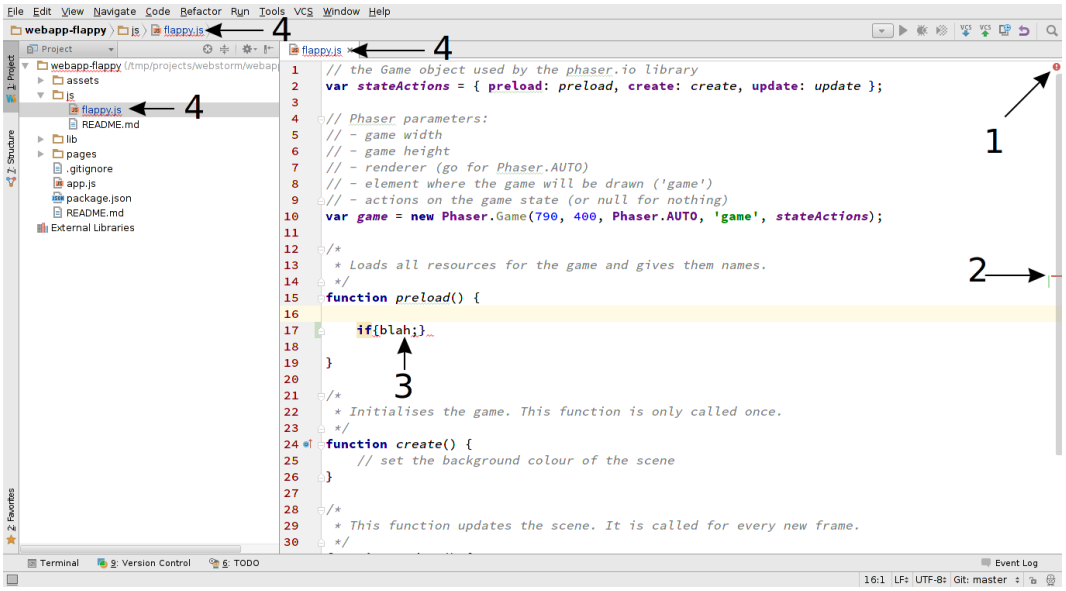
Auto-completion

1. Colouring: Webstorm will colour different parts of the program differently. This will help you recognise keywords from variables from values. You will learn what those are later.

2. Error detection: Webstorm indicates all the places it detected possible errors in your program. A red exclamation mark appears (1), a red marker shows on the scroll bar (2), the text is underlined in red (3) and the file name is underlined in red (4). Not all warnings will actually be errors and not all errors will be detected. Still, if your program doesn't work, check them.

FIGURE 1-9

Error hints



1. Auto-save: Webstorm does not have a save button, your files are automatically updated.

Statements, functions and arguments 2

In this module, you will create a background scene for your game while learning about statements and functions.

You are given a skeleton of the code in the file `flappy.js`, and during this workshop we will be adding code step by step. In Learning Activity 1 and 2, we will be adding code in the function `create()` block delimited by curly braces (`{}`). The programming language you will learn is called *JavaScript*.

```
var stateActions = { preload: preload, create: create, update: update };

//
var game = new Phaser.Game(790, 400, Phaser.AUTO, 'game', stateActions);

function preload() {


}

function create() {


}

function update() {

}
```

❶ The game object used by the phaser.io library

❷ Loads all resources for the game and gives them names.

❸ Initialises the game. This function is only called once. You will type your first line of code inside the `{}` of the function block `create()`

❹ This function updates the scene. It is called for every new frame.

Giving instructions to the computer

The first thing you will learn is how to tell the computer to do something for you!

Learning Activity 1: Set the game canvas background colour

In the function block `create()` type the code:

```
game.stage.setBackgroundColor("#F3D3A3");
```

Check the outcome of adding this code by right-clicking the file name `example.html` in the left navigation pane and selecting Open in browser > Chrome.

FIGURE 2-1

Opening in Chrome

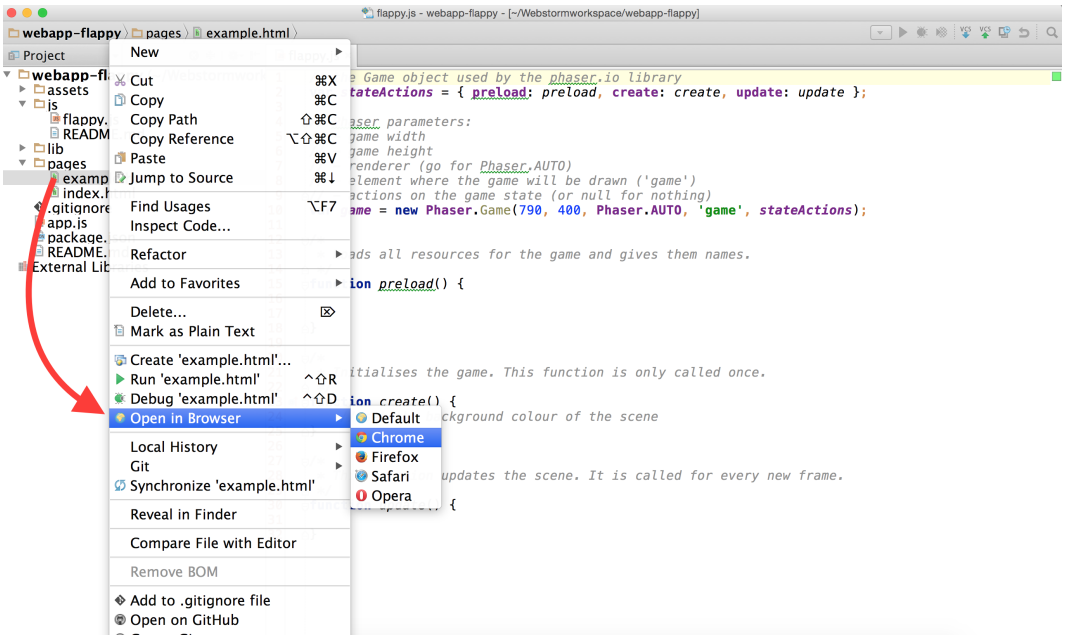
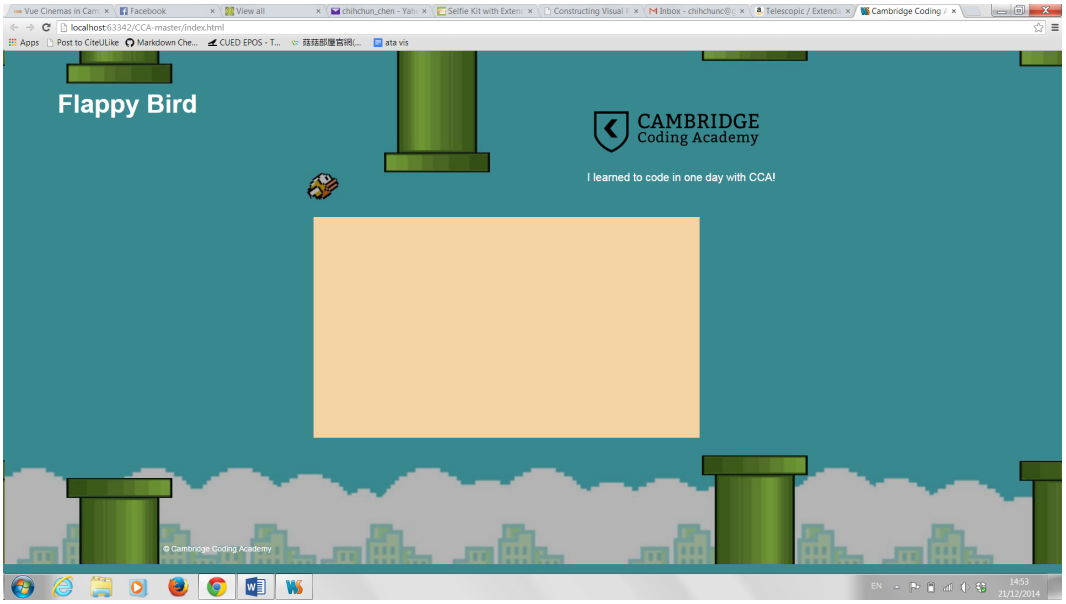


FIGURE 2-2

Setting the background colour



1. Modify the alphanumeric sequence inside the quotation marks, currently `#F3D3A3`. You can only use `A-F` and `0-9` and the length needs to be 6 characters (excluding #). Different character sequences correspond to different colours. When you have modified the value, simply refresh the page in the web browser to see the results.
2. Modify `#F3D3A3`, but this time use other characters outside the A-F, 0-9 range, e.g. Z, L etc. When you reload the page, there should be a black background (the default) because sequences containing these characters have no corresponding colour.

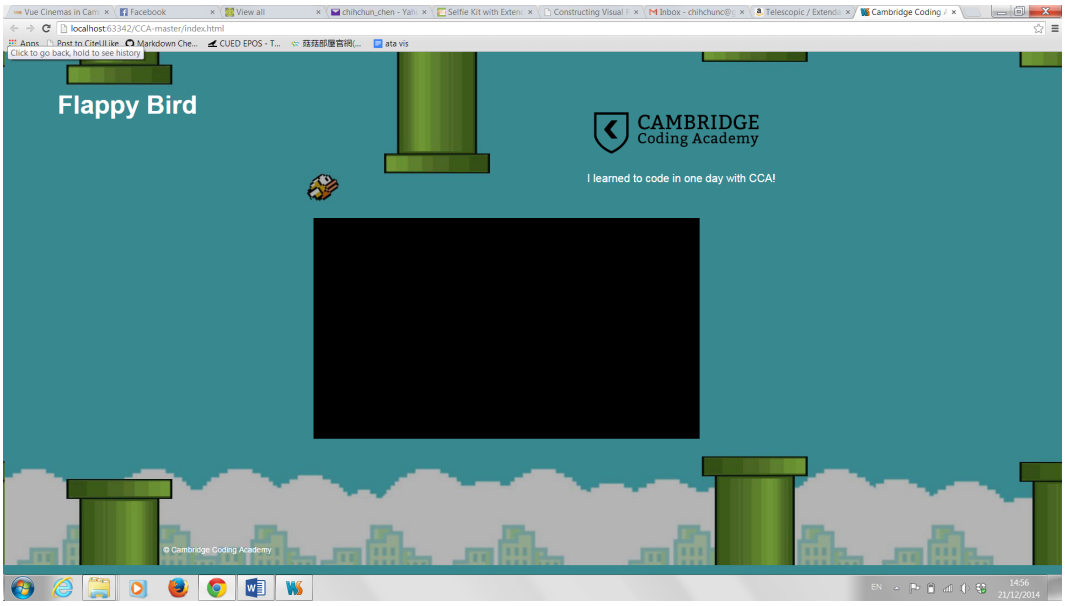


FIGURE 2-3

Setting the background colour

3. Choose your own colour using an HTML colour picker, e.g. ***http://www.w3schools.com/tags/ref_colorpicker.asp***

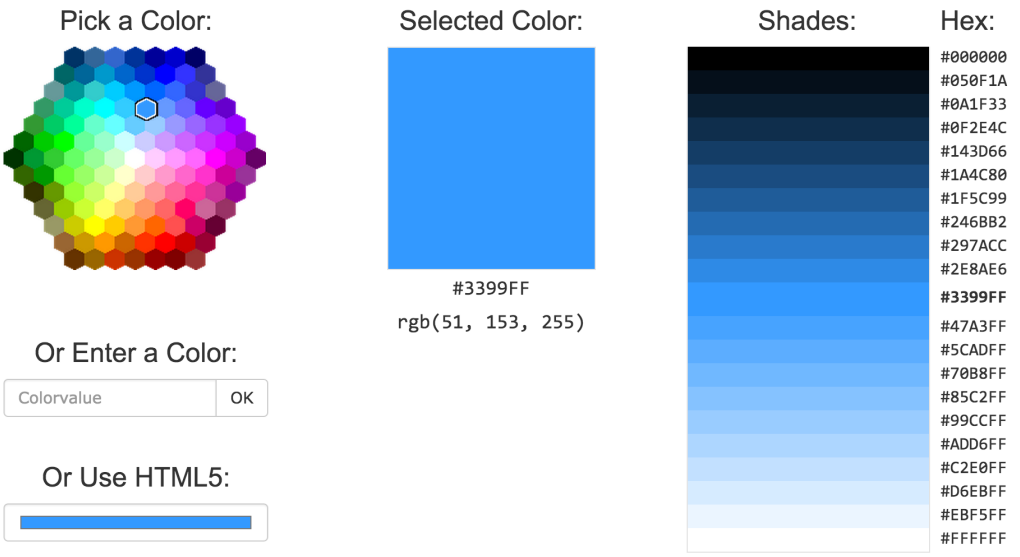


FIGURE 2-4

Colour pickers are one option for finding the colour codes you may want to use. One example is the W3C school's resource: http://www.w3schools.com/tags/ref_colorpicker.asp

Congratulations! You have given your first instruction (or command) to the computer. You have also learnt that colours are specially represented so the computer can interpret them. Let's now look at problems that may happen when you type your code.

SEMI-COLONS IN JAVASCRIPT

Note that instructions in JavaScript and many other programming languages terminate with the symbol `;` (a semi-colon). Even though many browsers nowadays might still execute the code without these semi-colons, this can't be guaranteed, and when you start using JavaScript for more heavy duty programming, sticking strictly to the right form of code will become very important.

Syntax

JavaScript is one among many programming languages. "Syntax" refers to the code you write in order to express what you want the program to do in a particular programming language. It is easy to make mistakes by misspelling commands. For example, you may have noticed that in JavaScript, the American spelling of "color" is used in `setBackgroundColor`. If you wrote instead `setBackgroundColour` (with the English spelling of "colour"), the code would not run. You can notice Webstorm hints at the error.

Learning Activity 2: Experiment with different syntax errors

1. What happens if you miss out the opening bracket of the command in Learning Activity 1, i.e. instead of

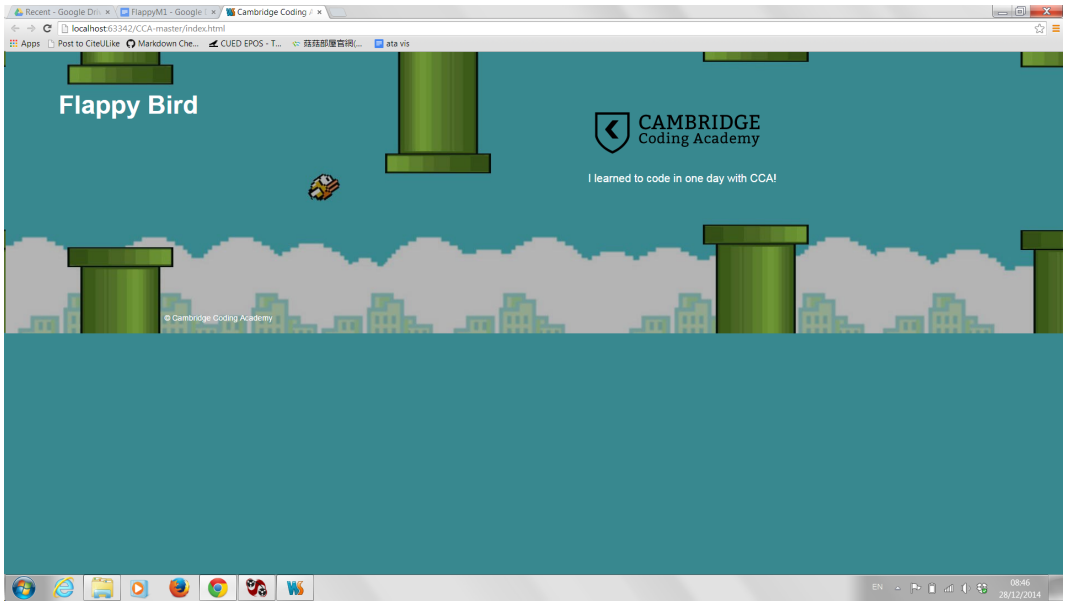
   ```
   game.stage.setBackgroundColor("#F3D3A3");
   ```

   you have:

   ```
   game.stage.setBackgroundColor"#F3D3A3");
   ```

FIGURE 2-5

Syntax errors cause the program to not work



In this case, only half the page is displayed because one of the instructions in the program is not expressed in valid syntax. When the program reaches this line, it stops working because it has reached a statement that can not be understood by the web browser.

2. What happens if you accidentally include an additional dot, e.g.

```
game.stage..setBackgroundColor("#F3D3A3");
```

Again, as above, only half the page is displayed because the instruction contains a syntax error.

3. What happens if you mistype `setBackgroundColor` so that it contains a lowercase `b` instead of an uppercase `B`, i.e.

```
game.stage.setbackgroundColor("#F3D3A3");
```

Interestingly, this results in the canvas being set to the default black background colour, which suggests that the web browser is still able to recognise that a command is being attempted. This is due to the fact that the code

```
game.stage.setbackgroundColor("#F3D3A3");
```

still has the correct format (this is similar to non-sensical but grammatically correct sentences, e.g. “green ideas sleep furiously.”)

QUIZ: WILL THIS CODE WORK?

```
game.stage.setBaCkgRouNdColOr("#F3DEEE");
```

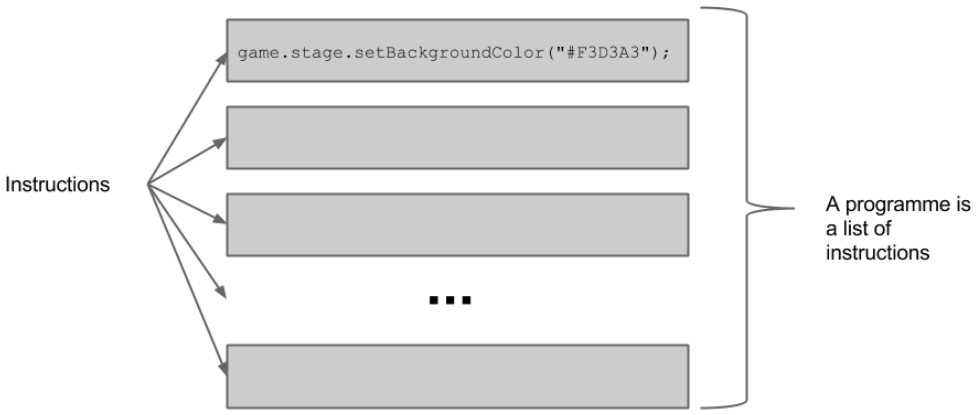
Answer: No. The instruction name is case sensitive. Consequently, the computer can not interpret the code because `setBaCkgRouNdColOr` is not a valid instruction.

Statements and function calls

A *computer program* is a list of instructions or commands, which is also known as *statements* in computing jargon. The code you typed: `game.stage.setBackgroundColor("#F3D3A3");` is a statement that determines the background colour of the game canvas.

FIGURE 2-6

A program is a list of instructions



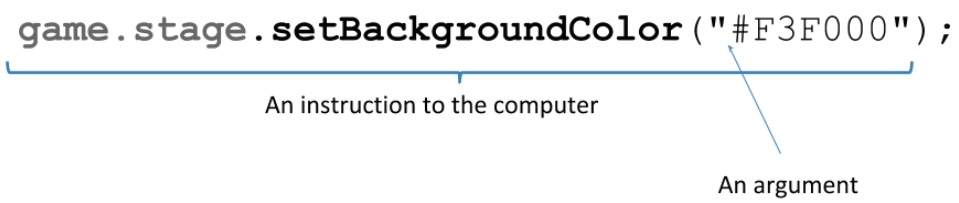
There are many different kinds of statements and the statement

```
game.stage.setBackgroundColor("#F3D3A3");
```

is specifically what we call a *function call*. You will generally recognise function calls from the opening and closing parentheses `()`. This type of statement tells the computer to execute a section of code, the function (you will also sometime hear the terminology "method"), which we "call" using its name. In this case, the function's name is `setBackgroundColor`. You provide fine-grained detail to the function by passing *arguments* to it. These are given in the parentheses following the function name. Passing arguments allows you to make more specific demands on the function. For example, by passing in the six-character code to `setBackgroundColor()`, you are telling the program that you want the background to be the specific colour corresponding to the code.

FIGURE 2-7

An argument to a function call



Note that functions can have any number of arguments, including none, in which case the parentheses are simply left empty. You will learn more about this in a moment.

QUIZ: HOW MANY ARGUMENTS?

```
game.add.text(20, 20, "Hello");
```

Answer: There are three arguments passed to `game.add.text()`

A *function block* is the entire specification of the function (its name and arguments) with its code. In other words, a function block describes what a function is meant to do. For example, the `create()` function block currently consists of the function name, followed by empty parentheses (indicating that there are no arguments to be supplied), and a single line of code setting the background colour inside curly braces (`{}`):

```
function create() {
    game.stage.setBackgroundColor("#F3D3A3");
}
```

Function blocks help to organise code by wrapping up instructions so that they can be called using the function call (as described above). You will recognise function blocks because they start with the `function` keyword.

Arguments and data types

Let's now look in more details at what you can do with arguments.

Learning Activity 3: Set and position the welcome text

In the function block `create()`, type:

```
game.add.text(20, 20, "Welcome to my game");
```

The first two arguments `20, 20` specify the location of the text on the game canvas; the first argument specifies the `x` coordinate while the second specifies the y coordinate of the top left corner of the text. The third argument `"Welcome to my game"` specifies the text itself, which is enclosed in quotation marks.

The game canvas has a coordinate system with its origin in the top left hand corner and which runs to the right (x coordinate increases on moving towards the right) and downwards (y coordinate increases on moving downwards).

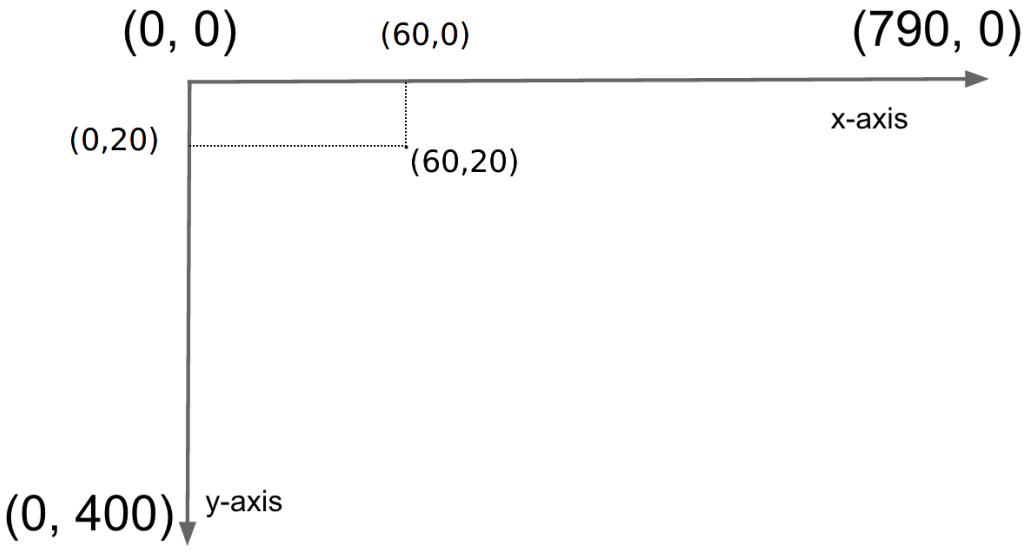


FIGURE 2-8

The game canvas coordinate system

1. Change the text `Welcome to my game` to your own welcome text.
2. Modify the `x` and `y` coordinates (i.e. the first two arguments) to alter the position of the text.
3. Place the text in the top right corner. (Remember the x coordinate increases on moving to the right and that the y coordinate has a value of 0 at the top of the canvas.)
4. Place the text in the bottom left corner. (Remember the y value increases on moving downwards.)

You may have noticed that different types of data can be given to functions as arguments. For example, in

```
game.add.text(20, 20, "Welcome to my game");
```

the first two arguments are numbers and the third argument is text (indicated by the fact that it is enclosed in quotation marks).

It is very important that values passed in as arguments have the correct data types. For example, the code

```
game.add.text(20, 20, Welcome to my game);
```

is incorrect since the text is not enclosed in quotation marks. Sometimes the web browser is forgiving (designed to be foolproof) and still produces the correct result even with these errors, but you should never count on this being the case as different browsers react differently, and the only way to ensure consistency is to write the correct code.

Learning Activity 4: Format the welcome text to your choice of font style, size and colour

1. Add an extra argument to the `game.add.text` function call by typing

   ```
   game.add.text(20, 20, "Welcome to my game",
       {font: "30px Arial", fill: "#FFFFFF"});
   ```

   The extra argument itself has two parts enclosed in curly braces (`{}`); the first part specifies the font (in this case size `30px Arial` font); the second part specifies the colour of the text. Note that `px` means “pixels” and is the basic unit measure for the size of things displayed on a screen. Based on your knowledge of how colour codes work from Learning Activity 1, modify the colour of the text by changing `#FFFFFF` to some other value.
2. Modify the font size and style by changing `30px Arial` to some other value. Note that you can only use font styles that are available in your computer. (In Chrome, you can see the list of fonts installed on your system by going to “Settings > Advanced settings > Customize fonts” and checking what is available in the drop down menus.)

Code structure and the Phaser framework

Now that you know how to add text to your game canvas and know how to give instructions to the computer using functions and function calls, let’s have a closer look at the code structure so you have a better understanding of how to develop your game. The file `flappy.js` contains the bulk of the program and controls its behaviour. In this workshop, all your coding will be done in this file.

In `flappy.js`, there are three empty function blocks:

1. `preload()`: This function is called once at the very beginning. It is used to load resources (e.g., images, sounds) for the game. In this function, resources are also given a name so they can be referred to in the rest of the code.
2. `create()`: This function is called once after `preload()` to set up the elements of the game. As soon as the function ends, the game starts.
3. `update()`: This function is called for each frame of the game. It lets you set up tests and actions for every instant of your game.

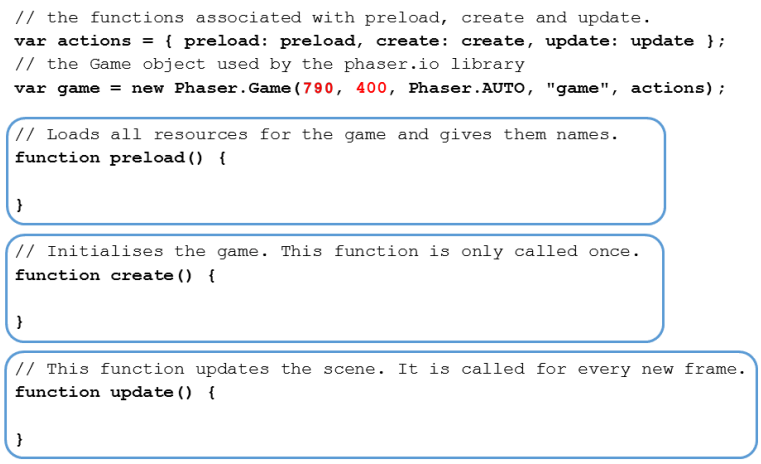


FIGURE 2-9

The code structure

A *framework* (also known as a "library") is a toolbox of predefined instructions. It's code that someone else has written and that you can use by invoking function calls, just as you have been doing with your own code. In this workshop you will be using a framework called Phaser, which has predefined instructions that are particularly useful for building online games, e.g. displaying images, setting colour, handling user input, playing sound, movement, physics emulation.

Making images available to the game

Now for the final task in this module: how do you add images to the game canvas?

Learning Activity 5: Produce a game canvas with an image in all four corners, e.g.:

FIGURE 2-10

A game scene with images

Before you are able to display images, you need to first make them available to the game. The `preload()` function handles this. More generally, `preload()` loads the resources you wish to include and associates them with aliases (names) so that you can refer to them easily in the rest of the code.

In the function block `preload()`, type:

```
game.load.image("playerImg", "../assets/jamesBond.gif");
```

The first argument (in this case `"playerImg"`) is the alias that will be associated with the image and which you use to refer to the image in the rest of the code. The second argument (in this case `"../assets/jamesBond.gif"`) specifies the file location of the image. If you wish to, you can modify this second argument so that a different image is used to represent your player.

Next, associate the image with your player, a `sprite`. *Sprite* is gaming terminology for a visual element that can have additional properties, e.g. size, velocity, gravity. Many of these additional properties are used for other gaming functions such as animation and physics.

In the function block `create()`, type

```
game.add.sprite(10, 270, "playerImg");
```

As with `game.add.text()`, the first two arguments (in this case `10, 270`) specify the location of the image, while the third argument (in this case `"playerImg"`) specifies the image (using its alias as defined in `preload()`) to associate with the sprite.

FURTHER READING

- *http://phaser.io/examples* – Example code using the Phaser framework
- *http://phaser.io/docs* – The documentation for the Phaser framework

Wrap up of Module 1

- A *program* is a sequence of statements which tell the computer what to do.
- A *statement* is a command or instruction to the computer.
- A *function* is a section of code with a name. This name is used to "call" the function from other parts of the program.
- A *function block* is the entire specification of the function (its name and arguments) with its code. For example, the `create()` function block currently consists of the function name, followed by empty parentheses (indicating that there are no arguments to be supplied), and a few instructions inside curly braces (`{}`).
- *Function blocks* help to organise code by wrapping up instructions so that they can be called using a single statement called the *function call*.
- A *framework* is a toolbox of predefined instructions (functions), which you can use in your code by calling them (just as you do with your own code).
- *Arguments* (also known as *"parameters"*) are a way of "customising" functions. The values passed in must have the correct data types (e.g. text, number) for the function to make sense of them.

Final code for this module can be found in Appendix A

Some frequently asked questions

HOW CAN I CHANGE AN IMAGE WITH WHITE BACKGROUND SO IT BECOMES TRANSPARENT?

You can find websites on google which can help you with that. For example, ***https://www.lunapic.com/editor/?action=transparent***

CAN I SET THE ENTIRE BACKGROUND TO AN IMAGE RATHER THAN A COLOUR?

Yes. All you need to do put an image in the asset folder (e.g., `myAwesomeBackground.png`) and reuse what you learnt in the final module. You first load your image in the `preload()` function block, e.g.

```
game.load.image("backgroundImg", "../assets/myAwesomeBackground.png");
```

Then you make the image appear as an object in the game canvas in the `create` function, e.g.

```
game.add.image(0, 0, "backgroundImg"); ❶
```

❶ The first two arguments (0, 0) specify the location of the top left hand corner of the image, while the third argument specifies the image itself using the alias you assigned to it in the `preload()` function.

You may also need to do some scaling of your image to make it to fill the canvas.

HOW DO I SCALE MY BACKGROUND IMAGE?

You can control the size of an image by changing its height and width, but to do this you first need to assign it to a variable (which we'll cover in Module 3). Instead of the simple `game.add.image(0,0,"backgroundImg")` as above, you can write:

```
var background = game.add.image(0, 0, "backgroundImg"); ❶
background.width = 100; ❷
background.height = 50; ❸
```

❶ `background` is the variable to which your image is being assigned.

❷ Set the width of the image.

❸ Set its height.

CAN I HAVE AN ANIMATED BACKGROUND BEFORE THE GAME STARTS?

Yes. This will be covered later on. One thing to bear in mind is that if you want an image to move even before the game has started, the code you use to do this should go in the `create()` block.

HOW DO I CHANGE THE SIZE OF THE CANVAS?

If you look at the code outside the three functions right at the top of your `flappy.js` file, you'll see the initialisation of the game (we'll explain the term "initialisation" in module 3):

```
var game = new Phaser.Game(790, 400, Phaser.AUTO, 'game', stateActions);
```

The first two arguments (in this case 790 and 400) specify the width and height (in pixel units) of the game canvas. By changing them, you can modify the size of your canvas.

HOW DO I CHANGE THE POSITION OF THE CANVAS?

The positioning of the game canvas is controlled in a file called `cca.css` in the `assets` folder. If you open this file and scroll to the bottom, you will find a block called `#game`. This is the block that controls the placement of the game canvas:

```
#game {
    margin: 20px;
    margin-left: auto;
    margin-right: auto;
    width: 800px;
}
```

The values that you might need to modify in order to move the game canvas are `margin`, `margin-left` and `margin-right`. You can find out more about the use of margins for positioning here: ***http://www.w3schools.com/css/css_margin.asp***

CAN I CHANGE THE BACKGROUND IMAGE OUTSIDE THE CANVAS?

The background image outside the canvas is specified in a file called `cca.css` in the `assets` folder. If you open this file, you will see that in the `#header` block, you have an image specified for the image at the top:

```
#header {
    background-size: cover;
    -webkit-background-size: cover;
    -moz-background-size: cover;
    -o-background-size: cover;
    background: url('flappy-header.png') top center no-repeat; ❶
    background-size: 100%;
}
```

❶ *flappy-header.png* is the image file for the header.

Similarly in the `#footer` block:

```
#footer {
    background-size: cover;
    -webkit-background-size: cover;
    -moz-background-size: cover;
    -o-background-size: cover;
    background: url('flappy-footer.png') top center no-repeat; ❶
    background-size: 100%;
}
```

❶ *flappy-footer.png* is the image file for the footer.

You can find these image files in the `assets` folder along with the other images. If you were to change these files in the `cca.css` file, you would be able to have a different background outside the canvas.

Event handling and user interaction 3

You now know how to create a program by providing a list of instructions to the computer. You are now also able to use functions to organise your code. Now you need to learn how to control the player of your game with inputs like key strokes or mouse presses. From the game's perspective, these inputs are *events*. Let's look at what this means.

Events

Learning Activity 6: Make the program display the message "click!" if there is a user input (e.g. mouse click, keyboard key press).

The `alert()` function pauses the program and displays a message to the user.

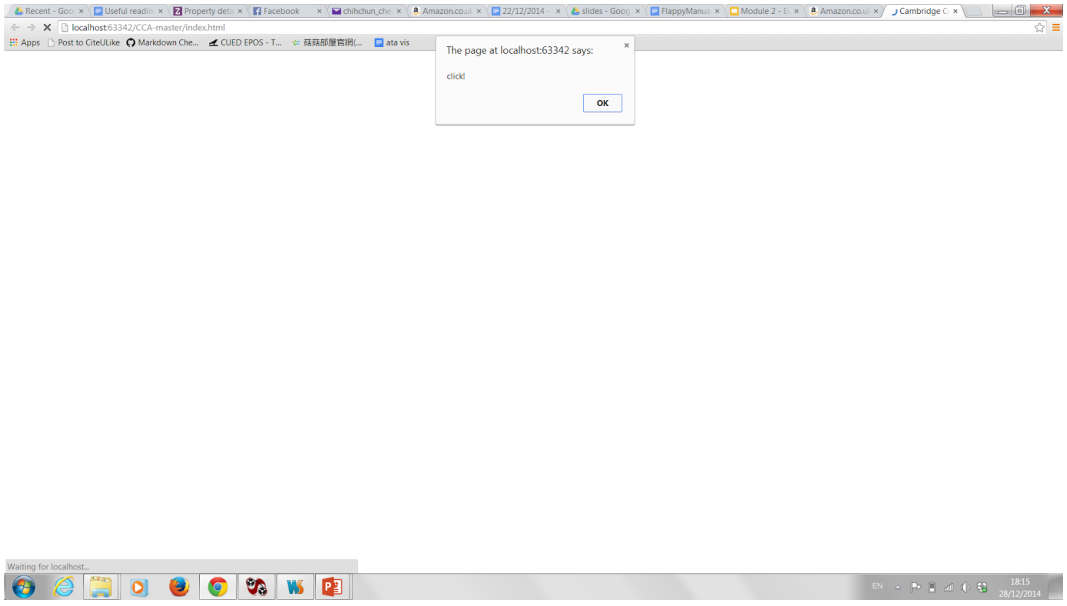


FIGURE 3-1

Alert function pauses the program and displays a message to the user.

1. Write a function which will be called when there's a click event. Let's call it `clickHandler` and type it outside of `create()` (before or after is not important) because it's separate to it (i.e. outside its associated curly braces `{}`). Remember function blocks start with the `function` keyword:

```
function clickHandler(event) {
    alert("click!");
}
```

2. Add the following code to the function block `create()`:

```
function create() {
    game.input              ❶
        .onDown             ❷
        .add(clickHandler); ❸
}
```

What is going on here?

❶ `input` is the Phaser input manager for any type of input, e.g. mouse button, touch screen, keyboard. It is the most general input manager since it "listens" for events from any input device.

❷ `onDown` is the event associated with the `input`, (a mouse button being clicked, key being pressed down etc.).

❸ `add` associates the function `clickHandler` (which is a "user-defined function" that you write yourself) with the input event, e.g. a function that makes an alert box appear might be associated with the click. The user-defined function `clickHandler` is known as the *event handler*.

LINE BREAKS IN JAVASCRIPT

Note that you can write code in JavaScript across multiple lines when you connect a set of operations with dots (.)! You will learn more of that in the following modules.

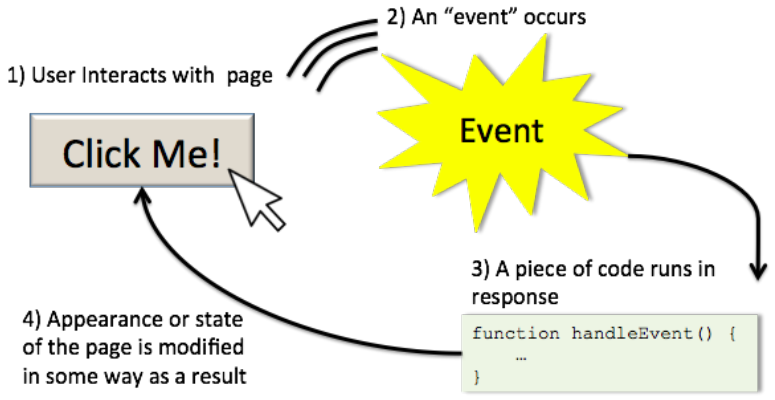
Since the function `clickHandler` you wrote is associated with the event `input.onDown`, every time a key or pointer is pressed down, an alert message with the text "click" will be displayed.

For most purposes, the event input.onDown is too general. We usually want to distinguish between different types of input, e.g. touch vs keyboard (`input.touch`, `input.keyboard`), different keyboard keys (`input.keyboard.SPACEBAR`, `input.keyboard.RIGHT`). This is what we will be exploring in the learning activities which follow.

In JavaScript, user inputs (e.g. mouse clicks, keyboard presses) are *events*. An *event* is a change that occurs when the program is running. It might be brought about by the program itself (i.e. the code itself produces the change), or it might come from something outside the program, e.g. user input, interaction with another program.

FIGURE 3-2

How user interaction triggers events in your program



Learning Activity 7: Make the program pause when the mouse is clicked and display the x and y coordinates of the click location.

Events often have parameters associated with them, e.g. a mouse click will have an `x` coordinate and `y` coordinate associated with it to represent its location. You can access these coordinates with `event.x` and `event.y` respectively. (Some events do not have parameters associated with them, e.g. a key being pressed.)

When an event has parameters associated with it, these parameters can be used in the handling of the event. How would you modify your `clickHandler` function so that it uses the `x` and `y` coordinates to display the location of the click using `alert`?

```
function clickHandler(event) {
    alert("The position is: " + event.x + "," + event.y);
}
```

This results in the `x` and `y` coordinates of the click location (delimited by a comma ",") being displayed, e.g. "`The position is: 100,200`", rather than the text "`click`". Note that the `+` operator is used to "join" bits of text together. Also note that Javascript converts numbers to their text representation when needed.

Learning Activity 8: Make an image appear where the mouse is clicked

How would you modify your `clickHandler` function to display your player image in the location of the click?

1. Remember the function introduced in the previous module used to display an image in a specified location:

   ```
   game.add.sprite(x, y, "playerImg");
   ```

2. A click event has `x` and `y` coordinates which you can access using `event.x` and `event.y`.

   ```
   function clickHandler(event) {
       game.add.sprite(event.x, event.y, "playerImg");
   }
   ```

Learning Activity 9: Make a sound play if the spacebar is pressed

1. Remember how you made an image available to the game in the `preload()` function? Now use the same principle to make the sound file available to the game by adding the following code to the `preload()` function block:

   ```
   game.load.audio("score", "../assets/point.ogg");
   ```

2. Write an event handler called `spaceHandler` which plays the sound:

   ```
   function spaceHandler() {
       game.sound.play("score");
   }
   ```

3. In the `create()` function block, add the spacebar to the game's input manager as well as associate the `spaceHandler` function:

```
game.input
    .keyboard.addKey(Phaser.Keyboard.SPACEBAR)
    .onDown.add(spaceHandler);
```

WHITE-SPACE

Notice that the instruction above is spread over several lines. It is customary to limit the length of lines of code to make sure it fits on any screen. As a result instructions are sometime split.

Line breaks count as white-space: character that separates other characters but are not visible themselves. This include spaces, line breaks and tabulation characters. Wherever there are two syntactical elements in your programs you can put as much white space as you want. Syntactical elements are variable names, dots, operators, separations characters (semi-colon, comma, brackets, etc.).

The following fragments are equivalent (but some are more readable than other):

```
function plusTimes(x,y,z){x.foo+=y*z;}
```

❶

```
function
        plusTime
( x
,y  , z)
   {
x
 . foo +=
          y  *z;
         }
```

❷

```
function plusTime(x, y, z){
    x.foo += y*z;
}
```

❸

❶ Overly compact code.

❷ Random white-space makes code difficult to read.

❸ Some white-space helps readability.

Try to make your code readable: separate your functions by line-breaks, indent the code within blocks, etc. And try to be consistent in your use of white-space and line breaks: it helps readability by giving a common style to all parts of your code.

Creating your own functions

Both `clickHandler` and `spaceHandler` are functions that you defined yourself. Here's a more general recipe for creating your own functions:

1. Use the `function` keyword to tell the computer to create a new function. This makes all the code associated with the function block available when the function is called.
2. Give the function a name. The name is used to call the function and invoke the code associated with it, i.e. when the function is called, the code is executed (analogy: when we call "make tea", we are actually invoking "add water to kettle", "close kettle lid", "switch kettle on"....."pour boiling water over teabag in mug"). In the above example, the function's name is `clickHandler`.

3. Write a list of arguments in the parentheses, "`()`" following the function name; separate by commas if there is more than one argument. The list might also be empty. In the `clickHandler` function above, there is only one argument, event.
4. Write the code for the function inside curly braces "`{}`". This is very important! Any code outside of the curly braces will not be part of your function block.
5. You can call the function with the pattern `functionName(arguments)`. (Arguments are also known as parameters.)

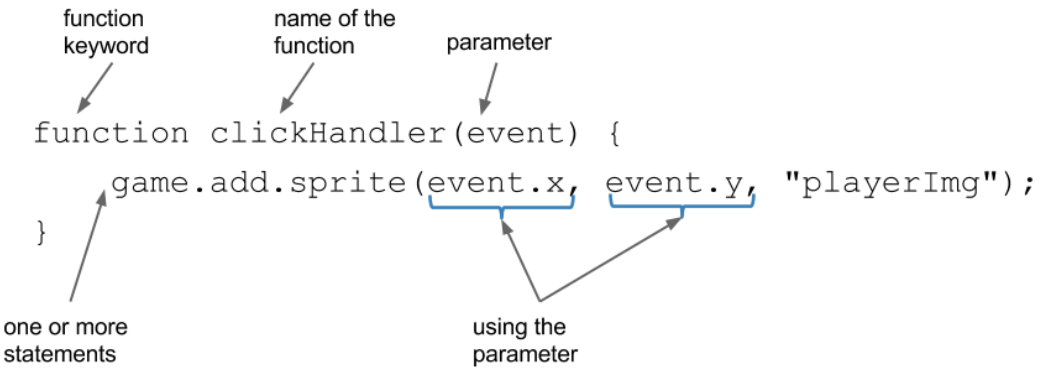


FIGURE 3-3

Structure of a function

QUIZ: WHAT'S WRONG WITH THIS FUNCTION BLOCK?

```
function moveLeft {

}
```

Answer: It is missing the empty parentheses `()` to indicate that it is a function that has zero parameter. The correct syntax is:

```
function moveLeft() {

}
```

Wrap up

- An *event* is a change that occurs when the program is running. It might be brought about by the program itself (i.e. the code itself produces the change), or it might come from something outside the program, e.g. user input, interaction with another program.
- A user input (e.g. mouse click, keyboard press) is an *event*.
- Some events have parameters associated with them. These parameters may be used when handling them.
- A *function* is a named section of code that you can refer to in other parts of the program using its name.
- *Arguments* of a function specify the data that needs to be passed into the function so that it can be *customised*.

- To write a your own function, you use the keyword function, followed by the function name followed by the arguments (aka. parameters) inside parentheses, followed by the code you want to be run when you call the function inside curly braces (**Figure 3-3**)
- An *event handler* is a function that is associated with an event and produces the response to the event, e.g. sound, change in the game scene display.

Final code for this module can be found in Appendix B

Some frequently asked questions

Can I create responses to any key?

In general, the answer is yes, but some keyboards are unable to process certain combinations of keys. Full details here: ***http://www.html5gamedevs.com/topic/4876-impossible-to-use-more-than-2-keyboard-input-buttons-at-the-same-time/***

You can also learn more about input via the keyboard in Phaser's documentation: ***http://phaser.io/docs/Phaser.Keyboard.html/Phaser.Keyboard.html*** (on Phaser.Keyboard) ***http://phaser.io/docs/Phaser.Keyboard.html/Phaser.Key.html*** (on Phaser.Key)

Can I get the game to respond to voice input?

At the moment, Phaser does not have an input manager for voice input (only for mouse, keyboard, touch and gamepads). With time though, it may be that someone does develop one for voice, so keep your eyes peeled.

Is there a list of Phaser's keyboard key codes?

You can find a list here: ***http://www.html5gamedevs.com/topic/10139-phaser-keyboard-codes-cheatsheet/*** It is also in the code for `Phaser.Keyboard`, towards the bottom: ***https://github.com/photonstorm/phaser/blob/v2.3.0/src/input/Keyboard.js***

Which sound files can I use?

Theoretically you can use any type of sound file, but it is important to consider what will be compatible with different browsers and platforms. For example, if you use a `.wav` file, it may not be able to play through all browsers, and newer file types may not be compatible with older browsers.

For more details on sound file compatibilities with different browsers, see: ***https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats#Browser_compatibility***

4 Variables and objects

Variables are named containers which hold information (i.e. data). The name of a variable should be meaningful and reflect what it contains, e.g. the variable name `score` is better than the variable name `a` (the same is true of functions, whose name should reflect its purpose or what it does). Variables allow elements of a program to persist while undergoing change. For example, in the game, the player's score changes throughout the game, and the player moves.

You already know how to create a program that interacts with the player. But can you tell the program to remember things? For example, how would you get the program to remember the current score of the game or the location of the player? To do this you need to use *variables*!

Declaration, Initialisation and assignment

Learning Activity 10: Declare and initialise a variable called `score` to hold the score throughout the game.

1. Where in the code should the `score` variable be declared? Since the `score` needs to be updated throughout the game, it should be available to the `update()` function. It should therefore be a *global* variable. A global variable is a variable that is available everywhere in the code: it is available globally (as opposed to locally). In Javascript, to make a global variable, declare it right at the top outside all the function blocks.
2. What value does `score` have before it is initialised? Use `alert()` to check.

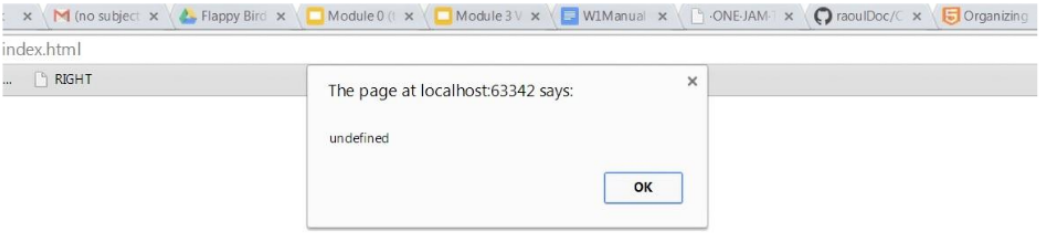
```
var score; ❶

function preload() {
    // your previous code for preload is here
}

function create() {
    // your previous code for create is here

    alert(score); ❷
}
```

❶ Declares `score` as a global variable.

❷ Uses the `score` variable inside `create()`.

FIGURE 4-1

An `alert` message showing that the `score` variable is `undefined`.



3. Assign an initial value to `score` and check what happens, e.g.

```
var score;
score = 0;
```

4. Now rewrite the above as one line of code.

```
var score = 0;
```

To create a new variable, you first need to *declare* it. Declaring a variable means giving a container a name so that the program knows that this name will be associated with the data in the container.

After the variable is declared, you can *assign* data values to it using the `=` operator. *Initialisation* refers to the first time you do this, e.g.

```
var score;          ❶
score = 0;          ❷
```

❶ Declaration

❷ Initialisation (first assignment)

It is also possible to combine declaration and initialisation into a single step, e.g.

```
var score = 0;
```

This can be used when you already know what the initial value of the variable should be. However, for cases where the initial value is dependent on other parts of the program, declaration and initialisation should be separated.

After initialisation, you can change the value of the variable by assigning new values to it, e.g.

```
var score = 0; ❶
score = 1;     ❷
score = 2;     ❸
```

❶ Declaration and initialisation

❷ Assignment

❸ Assignmment (again)

At the end of the code above, the value of `score` will be 2.

Learning Activity 11: Write a function that changes the value of score.

Do you remember how to declare your own functions? First, declare a function `changeScore` that takes no parameters.

```
function changeScore() {

}
```

Now you can modify the function so it increments the score by one:

```
function changeScore() {
        score = score + 1;
}
```

To see the score on the game display (rather than using `alert()`, which interrupts the program), you need to introduce another global variable, `labelScore`:

```
var labelScore;
```

This needs to be declared outside all the function blocks but initialised inside the `create()` function block.

```
var score = 0;                                  ❶
var labelScore;                                 ❷

function preload() {
    // your previous code for preload is here
}

function create() {
    // your previous code for create here

    labelScore = game.add.text(20, 20, "0"); ❸
}
```

❶ declare a global variable called `score` and initialise it to 0

❷ declare a global variable called `labelScore`

❸ initialise the `labelScore` variable with the graphical text object returned by `game.add.text`. The first two parameters of this function specify the location of the text (x and y coordinates, in this case both 20), while the third argument specifies the value of the text (in this case "0").

Note that the final argument in `game.add.text` (in this case "0") must be text. Since the score variable is a number, when you want to update this text, you need to use the `toString()` function to convert `score` to text format, e.g.

```
labelScore.setText(score.toString());
```

You can add this conversion statement to the `changeScore` function:

```
function changeScore() {
    score = score + 1;
    labelScore.setText(score.toString());
}
```

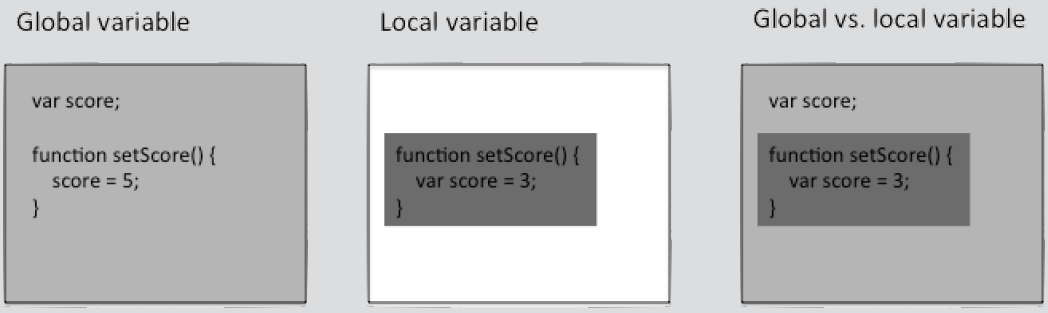
SCOPE

Something that is very important to consider when declaring a variable is where you declare it. This has implications for which bits of your code can use it. With JavaScript, when you refer to a variable by name the program looks for the variable in the closest level of nesting with that name.

If you declare and initialise the variable score right at the top of your code, outside all the function blocks, it will be available to all the functions in the program. Variables that are declared outside all the function blocks are called *global variables*, while variables declared inside function blocks (or further levels of nesting, e.g. loops, conditionals) are called *local variables*.

FIGURE 4-2

Global vs local variable.



In the left example, the function `setScore()` uses the *global* `score` variable and sets it to the value 5. In the middle example, the function `setScore()` itself declares and initialises a *local* variable called `score`, but this variable is not available outside the function so after `setScore()` has been executed, the `score` variable inside it is discarded. In the right example, the function `setScore()` declares and initialises a *local* variable called `score` which is independent from the *global* variable score defined outside.

To illustrate, the following code gives an alert with a value of 2:

```
var score = 0;                ❶

function create () {
    // your previous code for create() here

    changeScore();            ❷
    changeScore();            ❷
    alert(score);             ❸
}

function changeScore() {
    score = score + 1;        ❷
}
```

❶ The `score` variable is global.

❷ The function `changeScore()` is applied twice to the global score variable.

❸ The value displayed will be 2.

By contrast, the following code gives an output value of 0:

```
var score = 0;

function create () {
```

```
    // your previous code for create() here

    changeScore();     ❶
    changeScore();     ❶
    alert(score);      ❷
}

function changeScore() {
    var score = 0;
    score = score + 1;
}
```

❶ The function `changeScore()` already has a variable called `score`, which is declared and initialised each time the function is called and not available outside of the function.

❷ The value displayed will be 0.

For the purposes of our game you will be using the following definition of `changeScore()` to make sure the score label is updated whenever the score is updated:

```
function changeScore() {
        score = score + 1;
        labelScore.setText(score.toString());
}
```

Objects, methods and members

You may have noticed that certain program elements were more special than others. For example, in the previous module you could access the `x` coordinate of a click event with: `event.x`. What is this `.` about? It is a way to access the member called `x` of the event object. Let's explore in more detail what this means.

Learning Activity 12: Declare and initialise a variable called `player` to hold the sprite representing the player.

1. Where should `player` be declared? As with `score`, the `player` variable needs to be updated throughout the game so it should be a *global* variable and declared outside all the function blocks.
2. Where should `player` be initialised? Since `player` is a *sprite*, you need to associate it with an image. The `preload()` function makes images available to the game; therefore player can only be initialised after `preload()` has executed. As a result, `player` should be initialised in the `create()` function.

The initial value of player is therefore simply a sprite associated with an image at a given position in the game canvas, e.g.

```
var score;
var player; ❶

function preload() {
    // your previous code for preload is here
}
```

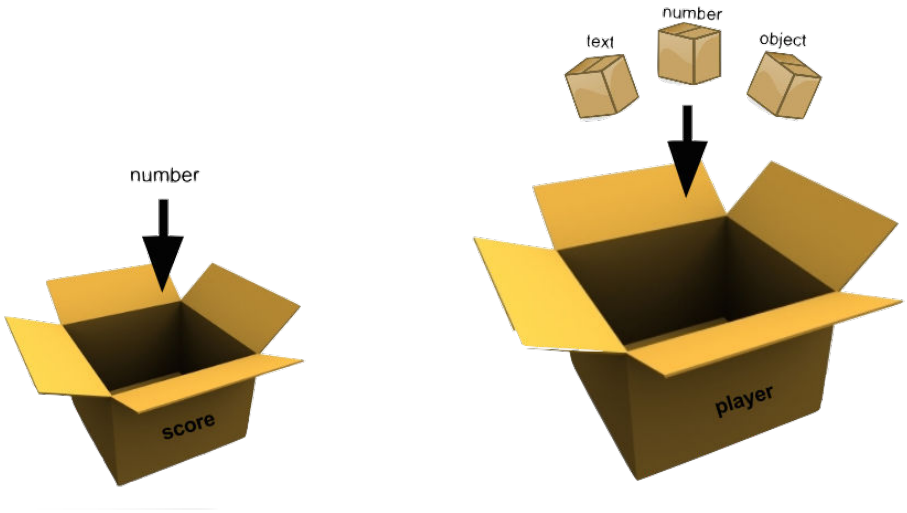
```
function create() {
    // your previous code for create is here
    player = game.add.sprite(100, 200, "playerImg");  ❷
}
```

❶ Declares `player` as a global variable.

❷ Initialises `player` to a game sprite.

FIGURE 4-3

`score` holds numerical values while `player` holds a sprite object.



The `score` variable in the previous learning activities holds number values (more precisely integers, `int`). In JavaScript, numbers, text and booleans (true/false values) are "primitive" data types, in that the language already has built-in operations and functions for dealing with them, e.g. adding numbers together, concatenating text, testing for truth conditions.

Objects are more complex data types. Just as functions are a way of grouping together commands, objects are a way of grouping together variables and the data they contain. A variable associated with an object is known as a member variable. For example, sprite objects have *member* variables `x` and `y` to hold the coordinates of their position. These are accessed by first referring to the object, followed by a ".", and then the variable name, i.e. the "." specifies that the `x` you are referring to is the one that is "local" to the object, and not some other `x` variable in the program, e.g.

```
// player is the name of the sprite
player.x;
```

Objects can also have functions associated with them. A function that is associated with an object is called a *method* of the object, e.g. in Phaser, a sprite has a method called `kill()` which stops rendering the sprite. You can call it like this:

```
player.kill();
```

The specifications for the members and methods of an object are known as its *class*. You can also see the class as the data *type*, while an object is an *instance* of the type. So the player object you have above is an object of the class `Sprite` (names of classes tend to be in capital letters).

The Phaser documentation (which you can find at: ***http://docs.phaser.io/index.html)*** lists the members and methods of the different classes in the framework.

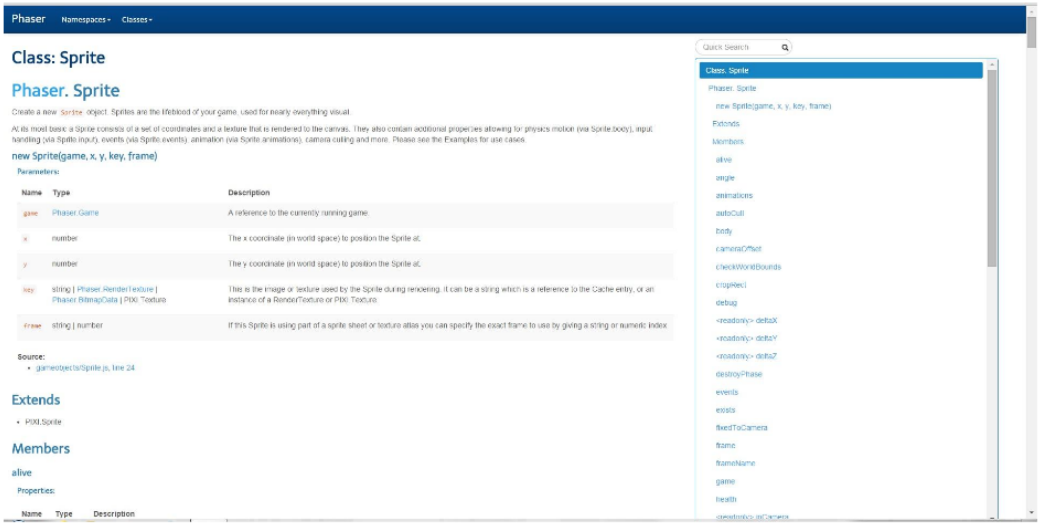


FIGURE 4-4

Phaser documentation.

Learning Activity 13: Make `player` move in response to the arrow keys

Try combining your understanding of objects and variables with what you learnt in the previous module on input event handling to make the player move in response to the arrow keys. When the right arrow key is pressed, the player should move to the right in the game scene, when the left arrow key is pressed, the player should move to the left in the game scene, when the up arrow key is pressed, the player should move upwards in the game scene, and when the down arrow is pressed, the player should move downwards in the game scene.

1. The position of a sprite is given by two of its member variables, `x` and `y`, representing respectively the x and y coordinates of the location in the game canvas where its image is displayed. Therefore, to specify the position of player, you change the value of its `x` and `y` member variables, e.g.

```
player.x = 150;
player.y = 200;
```

2. Movement is simply a change in position. Therefore to get the player to move, you need to take the values of its x and y member variables, apply an operation to these values, and then assign the respective variables with these new post-operation values, e.g.

```
player.x = player.x + 1; ❶
```

❶ Here, the current value of `player.x` is being replaced by a new value which is equal to the current value plus 1. So if the value of `player.x` was 150, it would be 151 after the operation.

3. If you wish to associate different move operations with different inputs (in this case different arrow keys), you need to put these operations in event handlers for the different inputs. For example, to get the player to move right in response to the right arrow key, you write an event handler function specifically for this right-directed movement:

```
function moveRight() {
        player.x = player.x + 1;
}
```

As you learnt in the previous module, you then need to make sure the right arrow key is associated with this event handler function in the `create()` function block:

```
function create() {
    // your previous code for create is here
    game.input.keyboard.addKey(Phaser.Keyboard.RIGHT)
                       .onDown.add(moveRight);

}
```

4. You can do the same for the other keys. In Phaser, the arrow keys are referred to using the following names:
    - Right arrow key: `Phaser.Keyboard.RIGHT` (as in the example above)
    - Left arrow key: `Phaser.Keyboard.LEFT`
    - Up arrow key: `Phaser.Keyboard.UP`
    - Down arrow key: `Phaser.Keyboard.DOWN`

Learning Activity 14: Alter the step size in your move operations

Rather than simply incrementing or decrementing the x and y coordinates by 1, try changing the operations inside your `moveRight()`, `moveLeft()`, `moveUp()` and `moveDown()` functions so that the step size is greater, e.g.

```
player.x = player.x + 10;
```

Syntactic "sugar"

The term *syntactic sugar* refers to code that provides a shorter or more elegant way of expressing a longer or more complex piece of code.

Learning Activity 15: Express the movement operations more succinctly

How can we write the following more succinctly (this is the operation for the `moveRight()` function)?

```
player.x = player.x + 1;
```

You can use the `+=` operator:

```
player.x += 1;
```

Or even more succinctly, the "++" operator:

```
player.x++;
```

In JavaScript (and most other programming languages), you can express the same command in different ways (just as in human languages you often have synonyms or equivalent expressions). One important example is the application of operations to variable values. E.g., the following three expressions do exactly the same thing.

```
player.x = player.x + 1;
player.x += 1;
player.x++;
```

Wrap up of Module 3

- A *variable* is a named container for data.
- Variables can be *global* so that they are available to all the code in the program, or they can be *local* to a particular function block or other nested code structure.
- An *object* is a data structure that groups together a set of variables and functions. The variables of the object are called its *members* while the functions of the object are called its *methods*.
- The *class* of an object is the type or template that specifies which members and methods the object has.

Code for this module can be found in Appendix C.

Some frequently asked questions

Can I have more than one player?

Yes, you can have as many as you like, and you can assign different controls to each of them. The important thing to ensure is that you name them different things (the computer is not telepathic). So, for example, if you wanted two players, instead of declaring and initialising just one player, you would declare and initialise two. You would also want them each to have their own score, e.g.

```
var score1; ❶
var score2; ❷
var player1;  ❸
var player2;  ❹

function preload() {
    // your previous code for preload is here
}

function create() {
    // your previous code for create is here
    score1 = 0; ❺
    score2 = 0; ❻
    player1 = game.add.sprite(100, 200, "player1Img");  ❼
    player2 = game.add.sprite(150, 200, "player2Img");  ❽
}
```

❶ Declares `score1` as a global variable. `score1` will hold the score for the first player.

❷ Declares `score2` as a global variable. `score2` will hold the score for the second player.

❸ Declares `player1` as a global variable.

❹ Declares `player2` as a global variable.

❺ Initialises `score1` to 0.

❻ Initialises `score2` to 0.

❼ Initialises `player1` to a game sprite associated with “player1Img”.

❽ Initialises `player2` to a game sprite associated with “player2Img”.

You would then use the same principles as you have learned in this module to control their behaviour and the input interactions with them. For example, you might use the up arrow to move `player1` up and the spacebar to move `player2` up. These keys therefore have to be associated with two distinct event handler functions:

```
function movePlayer1Up() {
        player1.x = player1.x + 1
}

function movePlayer2Up() {
        player2.x = player2.x + 1
}
```

As you learnt in the previous module, you then need to make sure the right key is associated with the right event handler function in the `create()` function block:

```
function create() {
    // your previous code for create is here
    game.input.keyboard.addKey(Phaser.Keyboard.UP)
                       .onDown.add(movePlayer1Up);
    game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR)
                       .onDown.add(movePlayer2Up);

}
```

5 Loops and conditionals

Now you have a game background and player, you will be adding pipes to your game scene, which will be the obstacles that your `player` sprite has to avoid.

Pipes are made out of repeated blocks that have a gap. How are you going to make the pipes? You will learn about *loops* to repeatedly execute some code and *conditionals* to execute some code depending on a condition.

Scaffolding

Make a new function called `generatePipe` which takes no parameter. For now the function is empty.

```
function generatePipe() {

}
```

In the `create` function, call `generatePipe` so you can see the effect it has.

```
generatePipe();
```

Now that the scaffolding has been set-up, you can start making pipes.

Repeating a command a number of times with a for-loop

Learning Activity 16: Use a for-loop to produce a vertical pipe

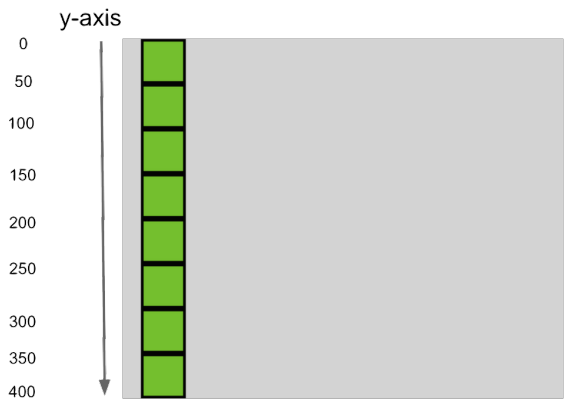


FIGURE 5-1

A vertical pipe generated with a for-loop

The image for each block of the pipe is the file `../assets/pipe.png` so the following line should be added to the `preload()` function block:

```
game.load.image("pipe","../assets/pipe.png");
```

What you now need to do is to repeat this block vertically a certain number of times to create a pipe. The height (and width) of this pipe block is 50, so to create a vertical column of stacked blocks to represent a pipe, you need to increase the y coordinate by 50 each time and display the pipe.

You could write the following code inside `generatePipe`.

```
game.add.sprite(20, 0, "pipe");
game.add.sprite(20, 50, "pipe");
game.add.sprite(20, 100, "pipe");
game.add.sprite(20, 150, "pipe");
game.add.sprite(20, 200, "pipe");
game.add.sprite(20, 250, "pipe");
game.add.sprite(20, 300, "pipe");
game.add.sprite(20, 350, "pipe");
```

However, this is very cumbersome. Imagine you want to change the coordinate or the image: you need to change many instructions. Thankfully, there's a nicer way to achieve the same result. Here's the equivalent to the code above using a for-loop:

```
for(var count=0; count<8; count+=1){
    game.add.sprite(20, 50*count, "pipe");
}
```

What is happening with this code? A for-loop consists of three elements:

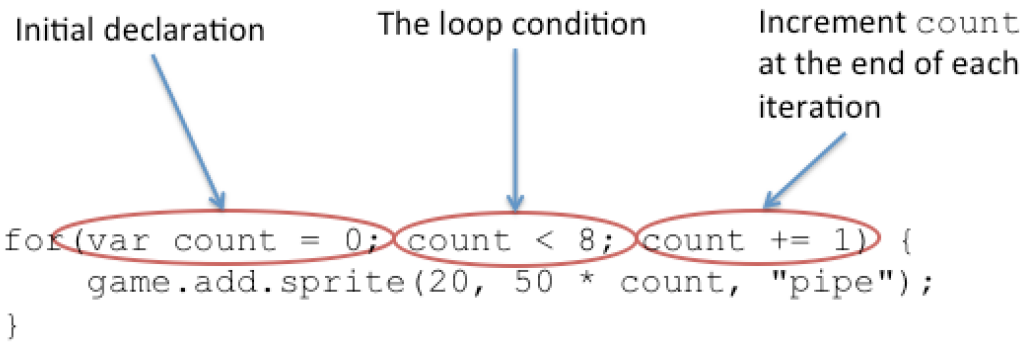


FIGURE 5-2

Components of a for-loop

- The first element declares and initialises the counter variable, e.g. `var count=0;` declares and initialises a variable called `count` to represent the counter.
- The second element specifies the condition that needs to be met for the code inside the loop to execute, e.g. `count<8` specifies that the code inside the loop is only executed when the value of count is less than 8.
- The third element is an operation that is applied to the counter variable with each loop, e.g. `count += 1` increments the value of count by 1 at the end of each loop.

You can see that the variable `count` is quite important! It holds the number of times that the code inside the for-loop block (i.e. inside the curly braces `{}`) has been repeated.

So the code above is equivalent to the following:

```
var count = 0; ❶
game.add.sprite(20, 0, "pipe");
count += 1;
game.add.sprite(20, 50, "pipe");
count += 1;
game.add.sprite(20, 100, "pipe");
count += 1;
game.add.sprite(20, 150, "pipe");
count += 1;
game.add.sprite(20, 200, "pipe");
count += 1;
game.add.sprite(20, 250, "pipe");
count += 1;
game.add.sprite(20, 300, "pipe");
count += 1;
game.add.sprite(20, 350, "pipe");
count += 1;    ❷
```

❶ Starts the iteration.

❷ The for-loop now stops because `count` contains the value 8 which invalidates the condition `count<8`.

CONTROL-FLOW OF FOR-LOOPS

The control-flow is a description of the order in which instructions are executed. Within a code block, instructions are executed one after the other: the control-flow is top to bottom. In a for-loop, the control flow is a bit more complex.

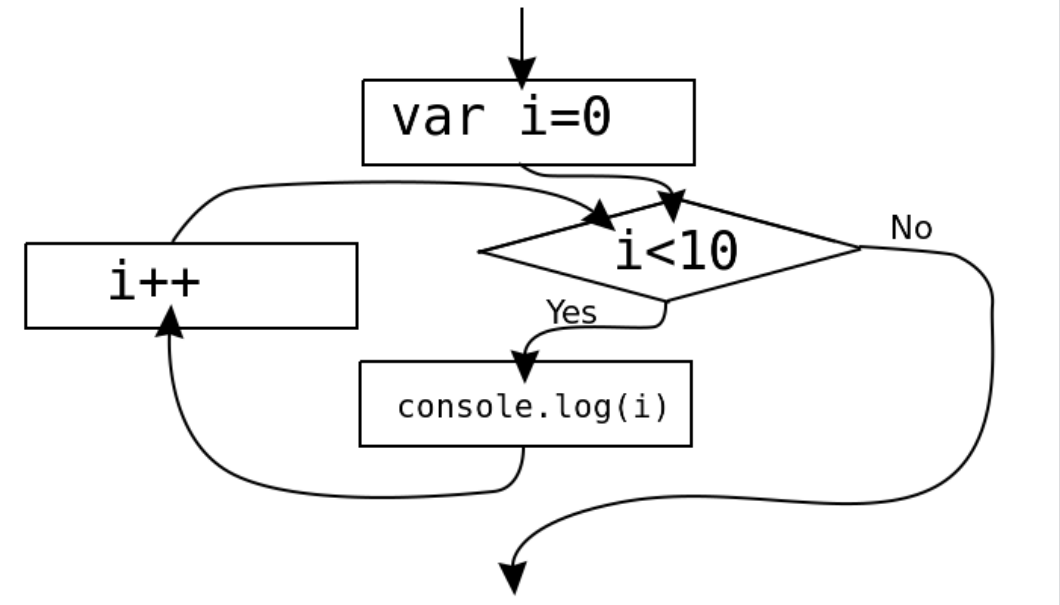
Consider the following for-loop:

```
for(var i=0; i<10; i++) {
    console.log(i);
}
```

The control flow is described in the figure Figure 5-3. You can see the four elements of the for-loop: the initialisation (`var i=0`), the test (`i<10`), the increment (`i++`) and the body (`console.log(i)`). The arrows represent transition from one instruction to the next. The diamond is a test with two possible exits: going into the loop if the test is true, or out if the test is false.

FIGURE 5-3

Control flow in a for-loop



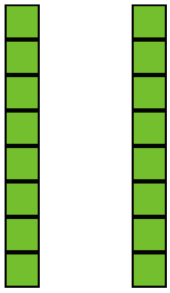
What happens if the test is never false? The loop goes on and on forever and ever. This is called an infinite loop and can result in stalled programs.

Learning Activity 17: Write a for-loop which produces two vertical pipes at each iteration

How would you modify your code above that uses a for-loop so it now displays two vertical pipes at the same time?

FIGURE 5-4

Two vertical pipes generated with a for-loop



In order for two vertical pipes to be displayed, we simply need to include two `game.add.sprite` commands in the for-loop block (remember it's delimited by the curly braces `{}`) with different x locations (in this case 20 and 150).

```
for(var count=0; count<8; count++){
    game.add.sprite(20, 50 * count, "pipe");
    game.add.sprite(150, 50 * count, "pipe");
}
```

Learning Activity 18: Write a for-loop that adds a pipe part at regular intervals

FIGURE 5-5

Pipe blocks at regular intervals generated with a for-loop

The operation in the third component of the for-loop determines the amount by which `count` is incremented with each loop. In the previous learning activity, count was incremented by 1 each time and multiplied with the ***height*** of the block (50) to give the *y* coordinate so that a continuous ***vertical*** column was formed.

To display the image above, the first thing to note is that the repetition occurs in the *horizontal* rather than vertical direction. Therefore you need to multiply `count` by the width of the block (which is also 50) to give the x coordinate. Secondly, to make the blocks occur at intervals rather than continuously, you need to control the amount by which `count` is incremented with each loop, e.g.

```
for (var count=2; count<10; count+=2) {
    game.add.sprite(count * 50, 200, "pipe");
} ❶
```

❶ The `count` variable is updated *after* the code inside the loop is executed.

Conditionals

Let's now look at how to display a pipe with a gap. The `if` statement is used to check whether a condition is true, and if it is, the code enclosed in the curly braces following it is executed, e.g.

```
if (count<5) {
    alert("count is smaller than 5!");
}
```

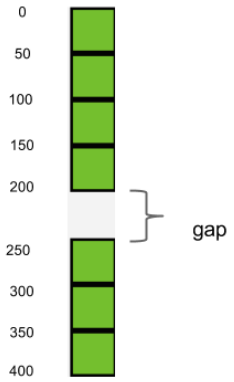
This code alerts a message if the `count` variable has a value less than 5.

There are other tests you can do: . `<` tests if something is smaller than something else, . `>` tests if something is bigger than something else, . `==` tests if two things are equal, . `!=` tests if two things are different, . Can you guess what `<=` and `>=` test? Try them on a few values…

Learning Activity 19: Write a for-loop that produces a pipe with a gap

FIGURE 5-6

Vertical pipe with a gap



In order to produce a vertical pipe with a gap, you again use a for-loop, but this time you want to skip displaying one of the pipe blocks. You can do this by using a conditional so that with each iteration, you check whether the `count` value satisfies the condition that it is not the block you wish to skip, e.g. to skip the 5th pipe block:

```
for(var count=0; count<8; count++) {
    if(count != 4){ ❶
        game.add.sprite(0, 50 * count, "pipe");
    }
}
```

❶ `count != 4` skips the 5th block because `count` starts at 0.

To associate different commands with different conditions, the `else if` statement can be used any number of times:

```
if (count<5) {
    alert("count is smaller than 5!");
}
else if (count==5) { ❶
    alert("count is equal to 5!");
}
else if (count==6) { ❶
    alert("count is equal to 6!");
}
else {
    alert("count is something else (i.e. greater than 6)");
}
```

❶ In Javascript, the `==` operator tests equality (the `=` symbol is reserved for setting variables).

(This type of else-if conditional is not used in the game but only shown here for reference.)

Learning Activity 20: Write a function to generate a pipe with random gaps

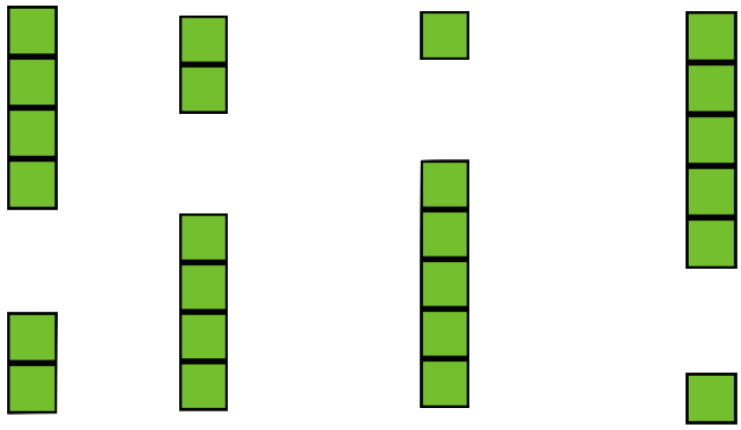


FIGURE 5-7

Multiple pipes with randomly placed gaps.

Modify the code in the `generatePipe` function to produce pipes with a randomly placed gap that is two blocks in size.

In the previous learning activity, you specified which pipe block you wanted to skip and "hard-coded" it into the loop. This time, you are going to let a random generator decide which pipe block to skip. You can use the function `game.rnd.integerInRange(lowerBound, UpperBound);` to generate a random integer (whole number) with a particular range, e.g. between 1 and 5:

```
var gapStart = game.rnd.integerInRange(1, 5);
```

You also need to consider the fact that unlike in the previous learning activity, the size of the gap is now 2 rather than 1, so you will have to skip two blocks counting from the place where the gap starts. Can you figure out the condition for this? Hint: you can say "and" using the `&&` operator.

Here's the code which generates a random pipe with a gap of size 2:

```
function generatePipe(){
    var gapStart = game.rnd.integerInRange(1, 5);          ❶
    for (var count=0; count<8; count=count+1) {    ❷
        if(count != gapStart && count != gapStart + 1) { ❸
            game.add.sprite(0, count * 50, "pipe");
        }
    }
}
```

❶ A random number between 1 and 5 is assigned to the variable `gapStart`.

❷ A for-loop which starts with `count = 0` and stops when `count` is greater or equal to 8.

❸ Leaves a gap of size 2.

Awesome — you can now generate pipes with random gaps! There's an additional step you need to undertake to make it easier to work with pipes in the future such as for collision detection.

As you can see from the code above, a pipe is made up of several pipe blocks, each of which is a sprite. To more easily test for collisions against each of them in the next module, you should group them into different blocks. For this purpose you will use arrays.

ARRAYS

Arrays are collections of values. Elements can be added to and removed from them. It's a bit like a shelf of books where a book is an item and the shelf is the array. Each book in that shelf can be accessed using an index from left to right. The benefit of using arrays is that you can store more than one element in a variable.

The empty array is `[]`. It's a way to initialise an array and say it contains nothing yet. If you want an array with elements you separate them with commas: `[1,2,3,4]`. To add an element to an existing array, use the method `push`:

```
var arr = [1,2,3,4];
arr.push(5);
arr.push(6);
```

To access an element of an array use square brackets: `arr[0]`, `arr[1]` and so on. Be careful: the first element of the array is at index 0! (The second one at index 1 and so on.) Finally, to get the number of element in an array use `arr.length`.

Open the Javascript console in your browser (in Chrome, go to the menu and select "More tools > JavaScript console") and try the following lines:

```
var arr = [1,2,"three",4];
arr.push(2+3);
arr.push(6);
console.log(arr[0]);
console.log(arr[2]);
arr[ arr.length ]; ❶
arr[ arr.length - 1 ]; ❷
```

❶ Because the indexes start at 0, there are no elements at `arr.length`.

❷ The last element of an array is stored at index `arr.length - 1`.

There are many more things that you can do with arrays. You will learn only some of them here. To explore arrays further, visit *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array*.

1. Declare a global variable called `pipes` to hold the column of pipe blocks. Initialise this global variable to an empty array:

   ```
   var pipes = []; ❶
   ```

   ❶ The value `[]` is an empty array.

2. Write a function called `addPipeBlock()` that takes two parameters `x` and `y` and adds a new pipe block, e.g.:

   ```
   function addPipeBlock(x, y) {
       // create a new pipe block
       var block = game.add.sprite(x,y,"pipe"); ❶
       // insert it in the 'pipes' array
       pipes.push(block); ❷
   }
   ```

❶ Adds a sprite on the game canvas at the `x` and `y` coordinates.

❷ Inserts the pipe sprite into the `pipes` array.

3. Now modify your `generatePipe` function so that it calls `addPipeBlock` to build up the pipe, still with two blocks skipped to represent a gap, e.g.:

```
function generatePipe() {
    // calculate a random position for the gap
    var gap = game.rnd.integerInRange(1 ,5);
    // generate the pipes, except where the gap should be
    for (var count=0; count<8; count++) {
        if (count != gap && count != gap+1) {
            addPipeBlock(40, count*50);
        }
    }
}
```

Wrap up

- A *for-loop* can be used to repeat a particular command so long as a particular condition is met with respect to its counter variable. With each loop, the value of the counter variable is changed according to a particular operation.
- *if* and *else if* can be used to make the execution of a particular command dependent on a certain condition being met so that different commands are executed when different conditions are satisfied.

In this module, you learnt how to make the execution of an instruction dependent on a particular condition. In the case of the *for-loop*, this condition is related to the value of the counter, while in the case of *if* and *else if*, the condition is given explicitly.

Final code for this module can be found in Appendix D.

Some frequently asked questions

Does JavaScript have any other types of loops?

Yes, JavaScript also has `while` and `do while` loops. You can find out more about these and how they relate to the `for` loop you have just learnt about here: ***http://www.kirupa.com/html5/loops_in_javascript.htm***

Can I change the value of the counter variable inside the for-loop?

Yes, you can, e.g.

```
for (var count=0; count<5; count+=1) {
    count = 10; ❶
    console.log(count); ❷
} ❸
```

❶ The `count` variable is set to value 10.

❷ Print the value of `count` in the Javascript console.

❸ This loop will only execute once since after the first loop, value of the `count` variable would be 10 and the condition `count<5` would no longer hold.

Can the conditions in `if` and `else if` statements overlap?

Syntactically they can, but in the case of an `if` statement followed by one or more `else if` statements, only the first of the blocks satisfying the conditions will be executed. For example:

```
var score = 0;
if (score < 1) {
  score += 1;
}
else if (score <= 1) {
  score += 2;
}
```

After execution, `score` would have a value of 1 because only the block associated with `if (score < 1)` would be executed even though `score <= 1` is also satisfied.

What's the difference between using another `if` and using `else if` after `if`?

Using `else if` implies an either/or situation so that *either* the code inside the `if` block is executed *or* the code inside the `else if` block is executed (or neither if neither of their conditions are satisfied). In the case that the conditions are not mutually exclusive (ie. both can be true at the same time), the block associated with the first one to be satisfied will be executed. On the other hand, if you have two `if` conditions following each other, each will be checked and they could both potentially be executed. For example:

```
var score = 0;
if (score < 1) {
  score += 1;
}
else if (score >= 1 && score <= 100) {
  score += 2;
}
```

After execution, `score` would have a value of 1 because only the block associated with `if (score < 1)` would be executed.
On the other hand:

```
var score = 0;
if (score < 1) {
  score += 1;
} ❶
if (score >= 1 && score <= 100) {
  score += 2;
}
```

❶ After the first block is executed, score now has a value of 1, which fulfils the condition `score >= 1` in `if (score >= 1 && score <= 100)` in the next `if` statement.

After execution, `score` would have value 3 because the blocks associated with both the `if` conditions would be executed.

Can I have `else if` or `else` without `if`?

No, this would be a syntax error. You need to have an `if` before the `else if`. If you think about it, this makes logical sense since `else` indicates contrast with the preceding `if` condition.

Game physics and advanced event handling

6

You now have almost all the components for your game! You are just missing the physics components like gravity and velocity to move your player and the pipes on the screen.

Learning Activity 21: Start the Phaser physics engine in your game.

Phaser comes with a physics engine that works "out of the box" to make your life easier. It has everything you need to emulate gravity and velocity in your game.

One of the most popular physics engine is known as the `ARCADE` physics engine. To start the `ARCADE` physics engine in your game, you need to add the following in the `create()` function block:

```
game.physics.startSystem(Phaser.Physics.ARCADE);
```

Learning Activity 22: Enable game physics for your player sprite.

On its own, starting the physics engine has no impact on the sprites in the game. Enabling physics for a sprite has to be done in a separate line of code in the `create()` function block. You must ensure this separate line of code occurs somewhere after the code that starts the physics engine (and also after the sprite has been declared and initialised).

```
player = game.add.sprite(80, 200, "playerImg"); ❶
game.physics.arcade.enable(player);             ❷
```

❶ You already wrote this (possibly bit different arguments), no need to repeat it. Simply add the second line.

❷ Enable physics for the player sprite.

A physics engine is a set of functions that helps to emulate a model of the laws of physics. These determine how sprites in the game behave and interact with each other physically, e.g. how much a player bounces when it jumps, how collisions between sprites affect their velocity. Note that there are different physics engine (e.g. `NINJA` and `P2`) with different characteristics. For your game you will be using `ARCADE`, which is the simplest.

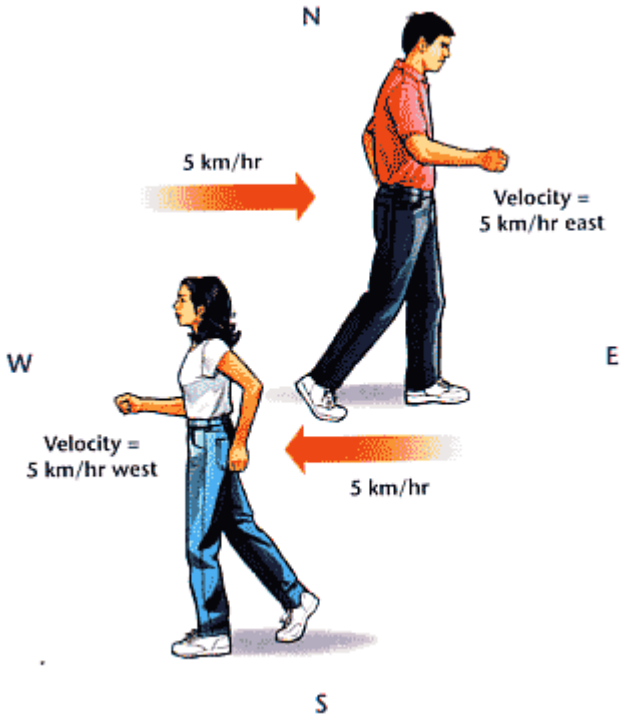
Velocity and Gravity

Let's now look at how to add velocity and gravity to your player.

Learning Activity 23: Set player velocity.

FIGURE 6-1

Velocity describes the speed and direction of movement of an object. It tells us how the position of an object changes with time.



1. To get your player to move to the right, set the velocity x value to a positive value, e.g.

```
player.body.velocity.x = 100;
```

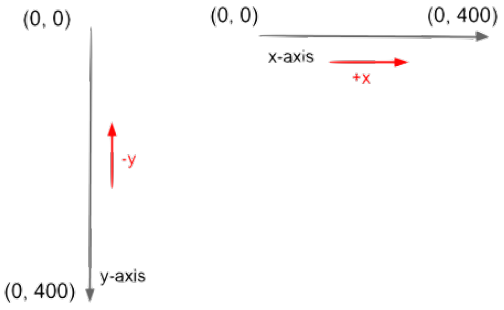
2. To get your player to move upwards, set the velocity y to a negative value (remember the y coordinate decreases on moving upwards), e.g.

```
player.body.velocity.y = -100;
```

3. Tweak the values to see how this affects the velocity of the player.

FIGURE 6-2

Velocity of a body, vertically and horizontally. Note that negative velocity means moving upwards!



Learning Activity 24: Set player gravity.

Gravity is the force of attraction between objects, including the “ground”, which is downwards.

1. You can set the amount of vertical gravity pulling the player down by setting the gravity y value, e.g.

```
player.body.gravity.y = 200;
```

2. Experiment with different values to make the player fall more quickly and more slowly, e.g.

```
player.body.gravity.y = 400;    ❶
player.body.gravity.y = 50;     ❷
```

❶ A lot of gravity - body falls more quickly.

❷ Less gravity - body falls more slowly.

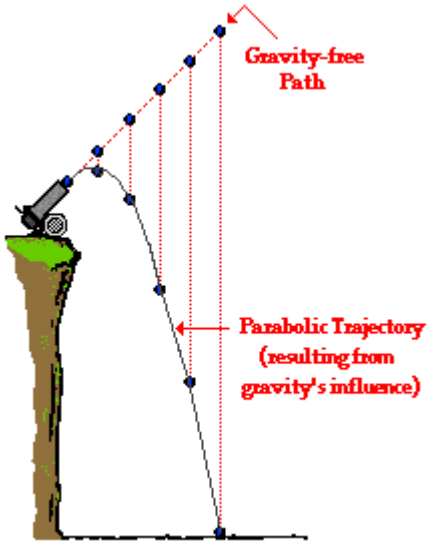


FIGURE 6-3

Gravity describes the force of attraction between two objects pulling at each other. The strength of gravity is determined by the size and distance of objects.

The heavier and/or closer an object, the stronger its gravitational pull. In Phaser, sprites can be assigned values for their gravity in different directions, representing the relative strength of the pull (not yet taking into account object weight or distance). For example, the y value determines the strength of the vertical force pulling the player down.

In Phaser, physics applies to the physical aspect of the sprite so it is the body that holds the member variables representing velocity and gravity, e.g.

```
player.body.velocity.x = 100;    ❶
player.body.gravity.y = 100;     ❷
```

❶ Sets the player to move to the right at speed 100.

❷ Subjects the player to a gravitational pull downwards of 100.

Learning Activity 25: Make the player jump when the spacebar is pressed.

1. In Module 2, you already wrote code to play sound in response to the spacebar being pressed, i.e. the event handler was a function that played sound. Therefore to make the player jump in response to the spacebar being pressed, you simply need to replace this event handler with a function that makes the player jump, i.e. move upwards with a given speed, e.g.

```
function playerJump() {
    player.body.velocity.y = -200;  ❶
}
```

❶ Try different numbers: the more negative the value the higher the player jumps.

2. You then need to associate this function with the event of the spacebar being press by adding the following in the `create()` function block:

```
game.input.keyboard
    .addKey(Phaser.Keyboard.SPACEBAR) ❶
    .onDown.add(playerJump);
```

❶ Associates the spacebar with the `playerJump` function.

Learning Activity 26: Write code that generates moving pipes.

One way of creating the impression of the player moving forwards is (towards the right) is to make the pipes move backwards (towards the left).

1. Modify the function `addPipeBlock` so that it adds a new pipe block that moves backwards (towards the left), e.g.:

```
function addPipeBlock(x, y) {
    var pipeBlock = game.add.sprite(x,y,"pipe"); ❶
    pipes.push(pipeBlock);                       ❷
    game.physics.arcade.enable(pipeBlock);       ❸
    pipeBlock.body.velocity.x = -200;            ❹
}
```

❶ Adds a sprite to the game canvas.

❷ Inserts the sprite in the `pipes` array.

❸ Enables physics for each individual pipe part.

❹ Sets the pipe's horizontal velocity to a negative value, which causes it to go left.

2. Modify the `generatePipe` function so the score is incremented each time a pipe is generated, e.g.:

```
function generatePipe() {
    var gap = game.rnd.integerInRange(1 ,5);     ❶
    for (var count = 0; count < 8; count++) {    ❷
        if (count != gap && count != gap+1) {    ❸
            addPipeBlock(750, count * 50);
        }
    }
```

```
    changeScore();                                    ❹
}
```

❶ Selects a random position within the specfied range for the gap.

❷ Adds the pipe blocks to form a pipe…

❸ … except where the gap should be.

❹ Increments the score each time a new pipe is generated. By calling `changeScore()` each time a pipe is generated, you track the player's progress in the game.

3. So far the code only generates a single moving pipe, but you need to continuously generate pipes at regular intervals throughout the game. To do this, you need to associate the `generatePipe()` function with a time loop event. The time loop event is added in the `create()` function block, e.g.

```
var pipeInterval = 1.75;
game.time.events
    .loop(pipeInterval * Phaser.Timer.SECOND,     ❶
        generatePipe);                            ❷
```

❶ The first argument (`pipeInterval * Phaser.Timer.SECOND`) determines the rate at which pipes are generated. `pipeInterval` is a variable that we can manipulate to alter the speed while `Phaser.Timer.SECOND` is a Phaser constant.

❷ The second argument associates the loop event with the `generatePipe` function.

In the same way that user input events have event handlers, the function `generatePipe` is the event handler for the time loop event.

Collision detection

Collision occurs when two objects or parts of objects occupy the same location on the game canvas.

Learning Activity 27: Detect collisions between the player sprite and a pipe, and terminate the game when this happens.

Since both the player and pipes are moving throughout the game, their respective positions are changing and need to be checked throughout the game. This is done in the `update()` function, which is called with each frame of the game and is responsible for changes in the state of the game.

To check whether there is a collision between the player and a pipe, we can use one of Phaser's built-in functions, called `overlap()`. This checks whether or not two game objects are overlapping and is associated with an event handler which deals with cases where there is overlap.

```
function update() {
  game.physics.arcade
        .overlap(player,     ❶
                 pipes,      ❶
                 gameOver); ❷
}

function gameOver(){
```

```
    game.destroy(); ❸
}
```

❶ The first two arguments (in this case `player` and `pipes`) in this function specify which two objects we are checking for overlap (i.e. collision)

❷ The final argument (in this case `gameOver`) is the event handler function for the event of there being overlap between the two objects. You will define this function shortly.

❸ in the `gameOver` function, stop the game.

The `overlap()` function accepts both sprites and arrays as arguments. When an array is given as an argument, what is going on "under the hood" is that overlap is being checked for each object in the array (in this case each pipe part). So another way of implementing `update()` would be to check each element of the `pipes` array in turn. Fortunately you already know about for-loops!

```
function update() {
    for(var index=0; index<pipes.length; index++){ ❶
        game.physics.arcade
            .overlap(player,    ❷
                pipes[index],   ❷
                gameOver);      ❸
    }
}
```

❶ Loops over all the elements of the array.

❷ The first two arguments (in this case `player` and `pipes[index]`) in this function specify which two objects we are checking for overlap (i.e. collision)

❸ The final argument (in this case `gameOver`) is the event handler function for the event of there being overlap between the two objects.

An alternative to `gameOver` is to automatically restart the game. You can do that by reloading the page:

```
function gameOver() {
    location.reload();      ❶
}
```

❶ Reloads the game.

Wrap up of Module 5

- A *physics engine* is responsible for emulating a model of the laws of physics for the objects that are associated with it.
- *Velocity* describes the speed and direction of movement of an object. It tells us how the position of the object changes with time.
- *Gravity* describes the force of attraction between two objects as a function of their size and distance from each other.
- *Event handlers* can be written for events which occur as a result of the game having a particular state, e.g. when two objects overlap.

Final code for this module can be found in Appendix E

Some frequently asked questions

What are the differences between the different physics engines?

Phaser has three built-in physics engines, `ARCADE` (which you used to add gravity, velocity and collision detection to your game), `NINJA`, and `P2`. Of these, `ARCADE` is computationally the cheapest since it treats game objects as rectangles, but this also means that it gives coarser results, e.g. imprecise collision detection. `NINJA` is more sophisticated in also allowing rotations of the objects. The most sophisticated (and correspondingly more computationally expensive) is `P2`, which is a full body physics system, which allows you to model springs and more complex objects. The best way to learn about both of these is to consult the documentation (***http://phaser.io/docs#physics***) and to experiment.

Can you have more than one physics engine in a game and when might you want to do this?

In some cases, in order to trade off the fact that the more sophisticated the model underlying the physics engine, the more computationally expensive it is (see question above), it may be beneficial to have one physics engine for simpler scenarios, and a different one for more demanding scenarios.

For example, in a shooting game you might use an `ARCADE` physics engine to handle the bullets hitting targets (straightforward collision detection), and `P2` to handle movement of the player across a bumpy landscape (which is able to model the slopes in the landscape). This would be more efficient than using `P2` to handle everything, since collision detection (for the bullets hitting targets) would be much cheaper/faster with `ARCADE` than it would be with `P2`, yet you would still be able to benefit from the `P2` model for player movement on the landscape.

What's the difference between overlap() and collide()?

Both `overlap()` and `collide()` detect overlap between two game objects. The main difference between these two functions is that `collide` automatically separates the objects when an overlap is detected (as if they are colliding), while overlap does not.

Final code for Module 1: an initial game scene with the background colour, player image and welcome text that you want.

```
// the functions associated with preload, create and update.
var actions = { preload: preload, create: create, update: update };
// the Game object used by the phaser.io library
var game = new Phaser.Game(790, 400, Phaser.AUTO, "game", actions);
// Loads all resources for the game and gives them names.
function preload() {
    // make image file available to game and associate with alias playerImg
    game.load.image("playerImg","../assets/jamesBond.gif");
}

// Initialises the game. This function is only called once.
function create() {
    // set the background colour of the scene
    game.stage.setBackgroundColor("#F3D3A3");
    // add welcome text
    game.add.text(20, 20, "Welcome to my game",
        {font: "30px Arial", fill: "#FFFFFF"});
    // add image associated with player
    game.add.sprite(80, 200, "playerImg");
}

// This function updates the scene. It is called for every new frame.
function update() {
}
```

Final code for Module 2: Images on a mouse click and sounds in your game

B

```
// the functions associated with preload, create and update.
var actions = { preload: preload, create: create, update: update };
// the Game object used by the phaser.io library
var game = new Phaser.Game(790, 400, Phaser.AUTO, "game", actions);
// Loads all resources for the game and gives them names.
function preload() {
    // make image file available to game and associate with alias playerImg
    game.load.image("playerImg","../assets/jamesBond.gif");
    // make sound file available to game and associate with alias score
    game.load.audio("score", "../assets/point.ogg");
}

// Initialises the game. This function is only called once.
function create() {
// set the background colour of the scene
    game.stage.setBackgroundColor("#F3D3A3");
    // add welcome text
    game.add.text(20, 20, "Welcome to my game",
        {font: "30px Arial", fill: "#FFFFFF"});
    // add image associated with player
    game.add.sprite(80, 200, "playerImg");
    // associate click input with clickHandler function
    game.input.onDown.add(clickHandler);
    // associate spacebar with spaceHandler function
    game.input
        .keyboard.addKey(Phaser.Keyboard.SPACEBAR)
        .onDown.add(spaceHandler);
}

// This function updates the scene. It is called for every new frame.
function update() {
}

// Executed when a click input is detected.
function clickHandler(event) {
    //Displays image (alias playerImg) at the click location.
    game.add.sprite(event.x, event.y, "playerImg");
}

// Executed when the spacebar is hit.
function spaceHandler() {
    //Plays sound (alias score).
    game.sound.play("score");
}
```

Final code for Module 3: move the player around on the screen with arrow keys

C

```
// the functions associated with preload, create and update.
var actions = { preload: preload, create: create, update: update };
// the Game object used by the phaser.io library
var game = new Phaser.Game(790, 400, Phaser.AUTO, "game", actions);
// Global score variable initialised to 0.
var score = 0;
// Global variable to hold the text displaying the score.
var labelScore;
// Global player variable declared but not initialised.
var player;

// Loads all resources for the game and gives them names.
function preload() {
    // make image file available to game and associate with alias playerImg
    game.load.image("playerImg","../assets/jamesBond.gif");
    // make sound file available to game and associate with alias score
    game.load.audio("score", "../assets/point.ogg");
}

// Initialises the game. This function is only called once.
function create() {
    // set the background colour of the scene
    game.stage.setBackgroundColor("#F3D3A3");
    // add welcome text
    game.add.text(20, 20, "Welcome to my game",
        {font: "30px Arial", fill: "#FFFFFF"});
    // add score text
    labelScore = game.add.text(20, 60, "0",
        {font: "30px Arial", fill: "#FFFFFF"});
    // initialise the player and associate it with playerImg
    player = game.add.sprite(80, 200, "playerImg");
    // associate right arrow key with moveRight function
    game.input.keyboard.addKey(Phaser.Keyboard.RIGHT).onDown.add(moveRight);
    // associate left arrow key with moveLeft function
    game.input.keyboard.addKey(Phaser.Keyboard.LEFT).onDown.add(moveLeft);
    // associate up arrow key with moveUp function
    game.input.keyboard.addKey(Phaser.Keyboard.UP).onDown.add(moveUp);
    // associate down arrow key with moveDown function
    game.input.keyboard.addKey(Phaser.Keyboard.DOWN).onDown.add(moveDown);
    //To test the changeScore function
    changeScore();
    changeScore();
}

// This function updates the scene. It is called for every new frame.
function update() {
}
```

```
// Executed when right arrow key is hit.
function moveRight() {
    //Moves player one step to the right of the game canvas.
    player.x = player.x + 1;
}

// Executed when left arrow key is hit.
function moveLeft() {
    //Moves player one step to the left of the game canvas.
    player.x = player.x - 1;
}

// Executed when up arrow key is hit.
function moveUp() {
    //Moves player one step upwards on the game canvas.
    //Remember the y coordinate values decrease on going up.
    player.y = player.y - 1;
}

// Executed when down arrow key is hit.
function moveDown() {
    //Moves player one step downwards on the game canvas.
    player.y = player.y + 1;
}

// Function to change the score
function changeScore() {
    //increments global score variable by 1
    score++;
    // updates the score label
    labelScore.setText(score.toString());
}
```

Final code for Module 4: generate a game scene with pipes

D

```
// the functions associated with preload, create and update.
var actions = { preload: preload, create: create, update: update };
// the Game object used by the phaser.io library
var game = new Phaser.Game(790, 400, Phaser.AUTO, "game", actions);
// Global score variable initialised to 0.
var score = 0;
// Global variable to hold the text displaying the score.
var labelScore;
// Global player variable declared but not initialised.
var player;
// Global pipes variable initialised to the empty array.
var pipes = [];

// Loads all resources for the game and gives them names.
function preload() {
    // make image file available to game and associate with alias playerImg
    game.load.image("playerImg","../assets/jamesBond.gif");
    // make sound file available to game and associate with alias score
    game.load.audio("score", "../assets/point.ogg");
    // make image file available to game and associate with alias pipe
    game.load.image("pipe","../assets/pipe.png");
}

// Initialises the game. This function is only called once.
function create() {
    // set the background colour of the scene
    game.stage.setBackgroundColor("#F3D3A3");
    // add welcome text
    game.add.text(20, 20, "Welcome to my game",
        {font: "30px Arial", fill: "#FFFFFF"});
    // add score text
    labelScore = game.add.text(20, 60, "0",
        {font: "30px Arial", fill: "#FFFFFF"});
    // initialise the player and associate it with playerImg
    player = game.add.sprite(80, 200, "playerImg");
    // generate a pipe
    generatePipe();
}

// This function updates the scene. It is called for every new frame.
function update() {
}

// Adds a pipe part to the pipes array
function addPipeBlock(x, y) {
    // make a new pipe block
    var block = game.add.sprite(x, y, "pipe");
```

```
    // add it to the pipes array
    pipes.push(block);
}

// Generate pipe with random gap
function generatePipe() {
    // Generate  random integer between 1 and 5. This is the location of the
    // start point of the gap.
    var gapStart = game.rnd.integerInRange(1, 5);
    // Loop 8 times (8 is the height of the canvas).
    for (var count = 0; count < 8; count = count + 1) {
        // If the value of count is not equal to the gap start point
        // or end point, add the pipe image.
        if(count != gapStart && count != gapStart + 1){
            addPipeBlock(40, count * 50);
        }
    }
}

// Function to change the score
function changeScore() {
    //increments global score variable by 1
    score++;
    // updates the score label
    labelScore.setText(score.toString());
}
```

Final code for Module 5: Movement of objects and physical interactions

E

```
// the functions associated with preload, create and update.
var actions = { preload: preload, create: create, update: update };
// the Game object used by the phaser.io library
var game = new Phaser.Game(790, 400, Phaser.AUTO, "game", actions);
// Global score variable initialised to 0.
var score = 0;
// Global variable to hold the text displaying the score.
var labelScore;
// Global player variable declared but not initialised.
var player;
// Global pipes variable initialised to the empty array.
var pipes = [];
// the interval (in seconds) at which new pipe columns are spawned
var pipeInterval = 1.75;

// Loads all resources for the game and gives them names.
function preload() {
    // make image file available to game and associate with alias playerImg
    game.load.image("playerImg","../assets/jamesBond.gif");
    // make sound file available to game and associate with alias score
    game.load.audio("score", "../assets/point.ogg");
    // make image file available to game and associate with alias pipe
    game.load.image("pipe","../assets/pipe.png");
}

// Initialises the game. This function is only called once.
function create() {
    // set the background colour of the scene
    game.stage.setBackgroundColor("#F3D3A3");
    // add welcome text
    game.add.text(20, 20, "Welcome to my game",
        {font: "30px Arial", fill: "#FFFFFF"});
    // add score text
    labelScore = game.add.text(20, 60, "0",
        {font: "30px Arial", fill: "#FFFFFF"});
    // initialise the player and associate it with playerImg
    player = game.add.sprite(80, 200, "playerImg");
    // Start the ARCADE physics engine.
    // ARCADE is the most basic physics engine in Phaser.
    game.physics.startSystem(Phaser.Physics.ARCADE);
    // enable physics for the player sprite
    game.physics.arcade.enable(player);
    // set the player's gravity
    player.body.gravity.y = 200;
    // associate spacebar with jump function
    game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR).onDown.add(playerJump);
    // time loop for game to update
```

```
    game.time.events.loop(pipeInterval * Phaser.Timer.SECOND, generatePipe);
}

// This function updates the scene. It is called for every new frame.
function update() {
    // Call gameOver function when player overlaps with any pipe

    game.physics.arcade
        .overlap(player,
            pipes,
            gameOver);

}

// Adds a pipe part to the pipes array
function addPipeBlock(x, y) {
    // make a new pipe block
    var block = game.add.sprite(x,y,"pipe");
    // insert it in the pipe array
    pipes.push(block);
    // enable physics engine for the block
    game.physics.arcade.enable(block);
    // set the block's horizontal velocity to a negative value
    // (negative x value for velocity means movement will be towards left)
    block.body.velocity.x = -200;
}

// Generate moving pipe
function generatePipe() {
    // Generate  random integer between 1 and 5. This is the location of the
    // start point of the gap.
    var gapStart = game.rnd.integerInRange(1, 5);
    // Loop 8 times (8 is the height of the canvas).
    for (var count = 0; count < 8; count++) {
        // If the value of count is not equal to the gap start point
        // or end point, add the pipe image.
        if(count != gapStart && count != gapStart+1){
            addPipeBlock(750, count * 50);
        }
    }
    // Increment the score each time a new pipe is generated.
    changeScore();
}

function playerJump() {
    // the more negative the value the higher it jumps
    player.body.velocity.y = -200;
}

// Function to change the score
function changeScore() {
    //increments global score variable by 1
    score++;
    // updates the score label
    labelScore.setText(score.toString());
}

function gameOver() {
    // stop the game (update() function no longer called)
    game.destroy();
}
```


CAMBRIDGE
CODING ACADEMY