**CONTENTS**

**……………………………………………………………………………………………..**

Contents

[INTRODUCTION](#page8) [2](#page8)

1.2 TAGS IN HTML. 5

1.3 HTML LISTS 7

1.4 BASIC OPERATIONS 8

CHAPTER 2

2.1 BASIC THEORY OF CSS 9

2.2 CSS SYNTAX [9](#page13)

2.3 HOW TO ADD CSS [9](#page14)

[Chapter 3](#page19) [13](#page19)

3.1 INTRODUCTION TO JAVASCRIPT 25

3.2 APPLICATION OF JS PROGRAMMING [16](#page22)

3.3 SYNTAX [19](#page25)

[Chapter 4](#page29) [27](#page29)

4.1 DEFINITION OF ANGULAR JS 29

4.2 MVC ARCHITECTURE 29

4.3 ANGULARJS DATA BINDING 30

4.4 ANGULARJS CONTROLLER 30

4.5 ANGULARJS SCOPE 33

4.7 ANGULARJS TABLES 38

4.8 ANGULARJS FORMS 40

4.9 ANGULARJS FORM VALIDATION 48

ABOUT THE PROJECT 49

PLATFORM GAME 50

**Chapter 1**

**INTRODUCTION**

**Chapter 1. Introduction to HTML**

**1.1. What is HTML**

**HTML** stands for **Hyper Text Markup Language**, which is the most widely used language on Web to develop web pages. **HTML** was created by Berners-Lee in late 1991 but "HTML 2.0" was the first standard HTML specification which was published in 1995. HTML 4.01 was a major version of HTML and it was published in late 1999. Though HTML 4.01 version is widely used but currently we are having HTML-5 version which is an extension to HTML 4.01, and this version was published in 2012.

**Why to Learn HTML?**

Originally, **HTML** was developed with the intent of defining the structure of documents like headings, paragraphs, lists, and so forth to facilitate the sharing of scientific information between researchers. Now, HTML is being widely used to format web pages with the help of different tags available in HTML language.

**HTML** is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning HTML:

* **Create Web site** - You can create a website or customize an existing web template if you know HTML well.
* **Become a web designer** - If you want to start a carrer as a professional web designer, HTML and CSS designing is a must skill.
* **Understand web** - If you want to optimize your website, to boost its speed and performance, it is good to know HTML to yield best results.
* **Learn other languages** - Once you understands the basic of HTML then other related technologies like javascript, php, or angular are become easier to understand.

**HTML text Editors**

* An HTML file is a text file, so to create an HTML file we can use any text editors.
* Text editors are the programs which allow editing in a written text, hence to create a web page we need to write our code in some text editor.
* There are various types of text editors available which you can directly download, but for a beginner, the best text editor is Notepad (Windows) or TextEdit (Mac).
* After learning the basics, you can easily use other professional text editors which are, **Notepad++, Sublime Text, Vim, etc**.
* In our tutorial, we will use Notepad and sublime text editor. Following are some easy ways to create your first web page with Notepad, and sublime text.

**Hello World using HTML.**

Just to give you a little excitement about HTML, I'm going to give you a small conventional **HTML Hello World** program, You can try it using Demo link.

<!DOCTYPE html>

<html>

<head>

<title>This is document title</title>

</head>

<body>

<h1>This is a heading</h1>

<p>Hello World!</p>

</body>

</html>

**Applications of HTML**

As mentioned before, HTML is one of the most widely used language over the web. I'm going to list few of them here:

* **Web pages development** - HTML is used to create pages which are rendered over the web. Almost every page of web is having html tags in it to render its details in browser.
* **Internet Navigation** - HTML provides tags which are used to navigate from one page to another and is heavily used in internet navigation.
* **Responsive UI** - HTML pages now-a-days works well on all platform, mobile, tabs, desktop or laptops owing to responsive design strategy.
* **Offline support** HTML pages once loaded can be made available offline on the machine without any need of internet.
* **Game development**- HTML5 has native support for rich experience and is now useful in gaming developent arena as well.

**1.2. Tags In HTML**

**Heading Tags**

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements **<h1>, <h2>, <h3>, <h4>, <h5>,** and **<h6>**. While displaying any heading, browser adds one line before and one line after that heading.

**Paragraph Tag**

The **<p>** tag offers a way to structure your text into different paragraphs. Each paragraph of text should go in between an opening <p> and a closing </p> tag

**Line Break Tag**

Whenever you use the **<br />** element, anything following it starts from the next line. This tag is an example of an empty element, where you do not need opening and closing tags, as there is nothing to go in between them.

The <br /> tag has a space between the characters **br** and the forward slash. If you omit this space, older browsers will have trouble rendering the line break, while if you miss the forward slash character and just use <br> it is not valid in XHTML.

An **HTML element** is defined by a starting tag. If the element contains other content, it ends with a closing tag, where the element name is preceded by a forward slash as shown below with few tags −

|  |  |  |
| --- | --- | --- |
| **Start Tag** | **Content** | **End Tag** |
| <p> | This is paragraph content. | </p> |
| <h1> | This is heading content. | </h1> |
| <div> | This is division content. | </div> |
| <br /> |  |  |

**Table 1.1 Tags in HTML**

So here **<p>....</p>** is an HTML element, **<h1>...</h1>** is another HTML element. There are some HTML elements which don't need to be closed, such as **<img.../>**, **<hr />** and **<br />** elements. These are known as **void elements**.

HTML documents consists of a tree of these elements and they specify how HTML documents should be built, and what kind of content should be placed in what part of an HTML document.

**HTML Tag vs. Element**

An HTML element is defined by a starting tag. If the element contains other content, it ends with a *closing tag*.

For example, **<p>** is starting tag of a paragraph and **</p>** is closing tag of the same paragraph but **<p>**This is paragraph**</p>** is a paragraph element.

**1.3. HTML List**

HTML Lists are used to specify lists of information. All lists may contain one or more list elements. There are three different types of HTML lists:

* Ordered List or Numbered List (ol)
* Unordered List or Bulleted List (ul)
* Description List or Definition List (dl)

**HTML Ordered List or Numbered List**

In the ordered HTML lists, all the list items are marked with numbers by default. It is known as numbered list also. The ordered list starts with <ol> tag and the list items start with <li> tag.

**<ol>**

**<li>**Aries**</li>**

**<li>**Bingo**</li>**

**<li>**Leo**</li>**

**<li>**Oracle**</li>**

**</ol>**

**HTML Unordered List or Bulleted List**

In HTML Unordered list, all the list items are marked with bullets. It is also known as bulleted list also. The Unordered list starts with <ul> tag and list items start with the <li> tag.

**<ul>**

**<li>**Aries**</li>**

**<li>**Bingo**</li>**

**<li>**Leo**</li>**

**<li>**Oracle**</li>**

**</ul>**

**HTML Description List or Definition List**

HTML Description list is also a list style which is supported by HTML and XHTML. It is also known as definition list where entries are listed like a dictionary or encyclopedia.

The definition list is very appropriate when you want to present glossary, list of terms or other name-value list.

The HTML definition list contains following three tags:

* **<dl> tag** defines the start of the list.
* **<dt> tag** defines a term.
* **<dd> tag** defines the term definition (description).

**1.4. Basic Operation in HTML**

**HTML Links - Hyperlinks**

HTML links are hyperlinks.

You can click on a link and jump to another document.

When you move the mouse over a link, the mouse arrow will turn into a little hand.

**HTML Links - Syntax**

Hyperlinks are defined with the HTML <a> tag:

<a href="*url*">*link text*</a>

**HTML Images Syntax**

In HTML, images are defined with the <img> tag.

The <img> tag is empty, it contains attributes only, and does not have a closing tag.

The src attribute specifies the URL (web address) of the image:

<img src="*url*">

**The alt Attribute**

The alt attribute provides an alternate text for an image, if the user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader).

The value of the alt attribute should describe the image:

**Example**

<img src="img\_chania.jpg" alt="Flowers in Chania">

**HTML Comments**

Comments are some text or code written in your code to give an explanation about the code, and not visible to the user. Comments which are used for HTML file are known as HTML comments. Anything written between these tags will be ignored by the browser, so comments will not be visible on the webpage.

Comments of any code make code easy to understand and increase readability of code.

Comments are also part of the code, which gives an explanation of the code.

**How to add comment In HTML**

You can add comments in your HTML file using <! -- ... --> tag. So if you will write anything between theses comment tag that will be treated as comment and browser will not read it.

**Syntax**

<! -- Write commented text here -->

**Chapter 2. Introduction to CSS**

**2.1. Basic Theory Of CSS**

What is CSS

CSS stands for Cascading Style Sheets. It is a style sheet language which is used to describe the look and formatting of a document written in markup language. It provides an additional feature to HTML. It is generally used with HTML to change the style of web pages and user interfaces. It can also be used with any kind of XML documents including plain XML, SVG and XUL.

CSS is used along with HTML and JavaScript in most websites to create user interfaces for web applications and user interfaces for many mobile applications.

What does CSS do

* You can add new looks to your old HTML documents.
* You can completely change the look of your website with only a few changes in CSS code.

**2.2. Why use CSS**

These are the three major benefits of CSS:

**1) Solves a big problem**

Before CSS, tags like font, color, background style, element alignments, border and size had to be repeated on every web page. This was a very long process. For example: If you are developing a large website where fonts and color information are added on every single page, it will be become a long and expensive process. CSS was created to solve this problem. It was a W3C recommendation.

**2) Saves a lot of time**

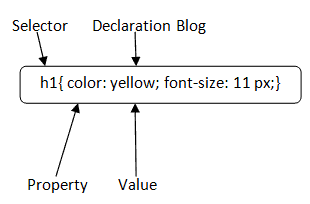
CSS style definitions are saved in external CSS files so it is possible to change the entire website by changing just one file.

**3) Provide more attributes**

CSS provides more detailed attributes than plain HTML to define the look and feel of the website.

**2.3. CSS Syntax**

A CSS rule set contains a selector and a declaration block.



**Fig 2.1 CSS Syntax**

**Selector:** Selector indicates the HTML element you want to style. It could be any tag like <h1>, <title> etc.

**Declaration Block:** The declaration block can contain one or more declarations separated by a semicolon. For the above example, there are two declarations:

* color: yellow;
* font-size: 11 px;

Each declaration contains a property name and value, separated by a colon.

**Property:** A Property is a type of attribute of HTML element. It could be color, border etc.

**Value:** Values are assigned to CSS properties. In the above example, value "yellow" is assigned to color property.

Selector{Property1: value1; Property2: value2; ..........;}

**CSS Selectors**

CSS selectors are used to "find" (or select) the HTML elements you want to style.

We can divide CSS selectors into five categories:

* Simple selectors (select elements based on name, id, class)
* [Combinator HYPERLINK "https://www.w3schools.com/css/css\_combinators.asp" selectors](https://www.w3schools.com/css/css_combinators.asp) (select elements based on a specific relationship between them)
* [Pseudo-class selectors](https://www.w3schools.com/css/css_pseudo_classes.asp) (select elements based on a certain state)
* [Pseudo-elements selectors](https://www.w3schools.com/css/css_pseudo_elements.asp) (select and style a part of an element)
* [Attribute selectors](https://www.w3schools.com/css/css_attribute_selectors.asp) (select elements based on an attribute or attribute value)

This page will explain the most basic CSS selectors.

**The CSS element Selector**

The element selector selects HTML elements based on the element name.

**Example**

Here, all <p> elements on the page will be center-aligned, with a red text color:

p {  
  text-align: center;  
  color: red;  
}

**The CSS id Selector**

The id selector uses the id attribute of an HTML element to select a specific element.

The id of an element is unique within a page, so the id selector is used to select one unique element!

To select an element with a specific id, write a hash (#) character, followed by the id of the element.

**Example**

The CSS rule below will be applied to the HTML element with id="para1":

#para1 {  
  text-align: center;  
  color: red;  
}

**Note:** An id name cannot start with a number!

**The CSS class Selector**

The class selector selects HTML elements with a specific class attribute.

To select elements with a specific class, write a period (.) character, followed by the class name.

**Example**

In this example all HTML elements with class="center" will be red and center-aligned:

.center {  
  text-align: center;  
  color: red;  
}

You can also specify that only specific HTML elements should be affected by a class.

**Example**

In this example only <p> elements with class="center" will be center-aligned:

p.center {  
  text-align: center;  
  color: red;  
}

HTML elements can also refer to more than one class.

**Example**

In this example the <p> element will be styled according to class="center" and to class="large":

<p class="center large">This paragraph refers to two classes.</p>

**Note:** A class name cannot start with a number!

**The CSS Universal Selector**

The universal selector (\*) selects all HTML elements on the page.

**Example**

The CSS rule below will affect every HTML element on the page:

\* {  
  text-align: center;  
  color: blue;  
}

**The CSS Grouping Selector**

The grouping selector selects all the HTML elements with the same style definitions.

Look at the following CSS code (the h1, h2, and p elements have the same style definitions):

h1 {  
  text-align: center;  
  color: red;  
}  
  
h2 {  
  text-align: center;  
  color: red;  
}  
  
p {  
  text-align: center;  
  color: red;  
}

It will be better to group the selectors, to minimize the code.

To group selectors, separate each selector with a comma.

**Example**

In this example we have grouped the selectors from the code above:

h1, h2, p {  
  text-align: center;  
  color: red;  
}

**2.4. How To Add CSS**

When a browser reads a style sheet, it will format the HTML document according to the information in the style sheet.

**Three Ways to Insert CSS**

There are three ways of inserting a style sheet:

* External CSS
* Internal CSS
* Inline CSS

**External CSS**

With an external style sheet, you can change the look of an entire website by changing just one file!

Each HTML page must include a reference to the external style sheet file inside the <link> element, inside the head section.

**Example**

External styles are defined within the <link> element, inside the <head> section of an HTML page:

<!DOCTYPE html>  
<html>  
<head>  
<link rel="stylesheet" type="text/css" href="mystyle.css">  
</head>  
<body>  
  
<h1>This is a heading</h1>  
<p>This is a paragraph.</p>  
  
</body>  
</html>

An external style sheet can be written in any text editor, and must be saved with a .css extension.

The external .css file should not contain any HTML tags.

Here is how the "mystyle.css" file looks like:

**"mystyle.css"**

body {  
  background-color: lightblue;  
}  
  
h1 {  
  color: navy;  
  margin-left: 20px;  
}

**Internal CSS**

An internal style sheet may be used if one single HTML page has a unique style.

The internal style is defined inside the <style> element, inside the head section.

**Example**

Internal styles are defined within the <style> element, inside the <head> section of an HTML page:

<!DOCTYPE html>  
<html>  
<head>  
<style>  
body {  
  background-color: linen;  
}  
  
h1 {  
  color: maroon;  
  margin-left: 40px;  
}  
</style>  
</head>  
<body>  
  
<h1>This is a heading</h1>  
<p>This is a paragraph.</p>  
  
</body>  
</html>

**Inline CSS**

An inline style may be used to apply a unique style for a single element.

To use inline styles, add the style attribute to the relevant element. The style attribute can contain any CSS property.

**Example**

Inline styles are defined within the "style" attribute of the relevant element:

<!DOCTYPE html>  
<html>  
<body>  
  
<h1 style="color:blue;text-align:center;">This is a heading</h1>  
<p style="color:red;">This is a paragraph.</p>  
  
</body>  
</html>

**Multiple Style Sheets**

If some properties have been defined for the same selector (element) in different style sheets, the value from the last read style sheet will be used.

Assume that an **external style sheet** has the following style for the <h1> element:

h1 {  
  color: navy;  
}

Then, assume that an **internal style sheet** also has the following style for the <h1> element:

h1 {  
  color: orange;     
}

**Example**

If the internal style is defined **after** the link to the external style sheet, the <h1> elements will be "orange":

<head>  
<link rel="stylesheet" type="text/css" href="mystyle.css">  
<style>  
h1 {  
  color: orange;  
}  
</style>  
</head>

**Example**

However, if the internal style is defined **before** the link to the external style sheet, the <h1> elements will be "navy":

<head>  
<style>  
h1 {  
  color: orange;  
}  
</style>  
<link rel="stylesheet" type="text/css" href="mystyle.css">  
</head>

**2.5. Cascading Order**

What style will be used when there is more than one style specified for an HTML element?

All the styles in a page will "cascade" into a new "virtual" style sheet by the following rules, where number one has the highest priority:

* Inline style (inside an HTML element)
* External and internal style sheets (in the head section)
* Browser default

So, an inline style has the highest priority, and will override external and internal styles and browser defaults.

**2.6. CSS Comments**

CSS comments are generally written to explain your code. It is very helpful for the users who reads your code so that they can easily understand the code.

Comments are ignored by browsers.

Comments are single or multiple lines statement and written within /\*............\*/ .

**Chapter 3. Introduction to JavaScript**

**3.1. Basic Theory of JavaScript**

JavaScript is a lightweight, interpreted programming language. It is designed for creating network-centric applications. It is complimentary to and integrated with Java. JavaScript is very easy to implement because it is integrated with HTML. It is open and cross-platform.

**Why to Learn Javascript**

**Javascript** is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Javascript:

* Javascript is the most popular **programming language** in the world and that makes it a programmer’s great choice. Once you learnt Javascript, it helps you developing great front-end as well as back-end softwares using different Javascript based frameworks like jQuery, Node.JS etc.
* Javascript is everywhere, it comes installed on every modern web browser and so to learn Javascript you really do not need any special environment setup. For example Chrome, Mozilla Firefox , Safari and every browser you know as of today, supports Javascript.
* Javascript helps you create really beautiful and crazy fast websites. You can develop your website with a console like look and feel and give your users the best Graphical User Experience.
* JavaScript usage has now extended to mobile app development, desktop app development, and game development. This opens many opportunities for you as Javascript Programmer.
* Due to high demand, there is tons of job growth and high pay for those who know JavaScript. You can navigate over to different job sites to see what having JavaScript skills looks like in the job market.
* Great thing about Javascript is that you will find tons of frameworks and Libraries already developed which can be used directly in your software development to reduce your time to market.

There could be 1000s of good reasons to learn Javascript Programming. But one thing for sure, to learn any **programming language**, not only Javascript, you just need to code, and code and finally code until you become expert.

**Hello World using Javascript**

Just to give you a little excitement about **Javascript programming**, I'm going to give you a small conventional Javascript Hello World program, You can try it using Demo link

<html>

<body>

<script language = "javascript" type = "text/javascript">

<!--

document.write("Hello World!")

//-->

</script>

</body>

</html>

There are many useful **Javascript frameworks** and libraries available:

* Angular
* React
* jQuery
* Vue.js
* Ext.js
* Ember.js
* Meteor
* Mithril
* Node.js
* Polymer
* Aurelia
* Backbone.js

It is really impossible to give a complete list of all the available Javascript frameworks and libraries. The Javascript world is just too large and too much new is happening.

**3.2.Applications of Javascript Programming**

As mentioned before, **Javascript** is one of the most widely used **programming languages** (Front-end as well as Back-end). It has it's presence in almost every area of software development. I'm going to list few of them here:

* **Client side validation** - This is really important to verify any user input before submitting it to the server and Javascript plays an important role in validting those inputs at front-end itself.
* **Manipulating HTML Pages** - Javascript helps in manipulating HTML page on the fly. This helps in adding and deleting any HTML tag very easily using javascript and modify your HTML to change its look and feel based on different devices and requirements.
* **User Notifications** - You can use Javascript to raise dynamic pop-ups on the webpages to give different types of notifications to your website visitors.
* **Back-end Data Loading** - Javascript provides Ajax library which helps in loading back-end data while you are doing some other processing. This really gives an amazing experience to your website visitors.
* **Presentations** - JavaScript also provides the facility of creating presentations which gives website look and feel. JavaScript provides RevealJS and BespokeJS libraries to build a web-based slide presentations.
* **Server Applications** - Node JS is built on Chrome's Javascript runtime for building fast and scalable network applications. This is an event based library which helps in developing very sophisticated server applications including Web Servers.

**3.3. Syntax**

JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page.

You can place the **<script>** tags, containing your JavaScript, anywhere within your web page, but it is normally recommended that you should keep it within the **<head>** tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

<script ...>

JavaScript code

</script>

The script tag takes two important attributes −

* **Language** − This attribute specifies what scripting language you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
* **Type** − This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

So your JavaScript segment will look like −

<script language = "javascript" type = "text/javascript">

JavaScript code

</script>

Let us take a sample example to print out "Hello World". We added an optional HTML comment that surrounds our JavaScript code. This is to save our code from a browser that does not support JavaScript. The comment ends with a "//-->". Here "//" signifies a comment in JavaScript, so we add that to prevent a browser from reading the end of the HTML comment as a piece of JavaScript code. Next, we call a function **document.write** which writes a string into our HTML document.

This function can be used to write text, HTML, or both. Take a look at the following code.

<html>

<body>

<script language = "javascript" type = "text/javascript">

<!--

document.write("Hello World!")

//-->

</script>

</body>

</html>

This code will produce the following result −

Hello World!

**Whitespace and Line Breaks**

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

**Semicolons are Optional**

Simple statements in JavaScript are generally followed by a semicolon character, just as they are in C, C++, and Java. JavaScript, however, allows you to omit this semicolon if each of your statements are placed on a separate line. For example, the following code could be written without semicolons.

<script language = "javascript" type = "text/javascript">

<!--

var1 = 10

var2 = 20

//-->

</script>

But when formatted in a single line as follows, you must use semicolons −

<script language = "javascript" type = "text/javascript">

<!--

var1 = 10; var2 = 20;

//-->

</script>

**Note** − It is a good programming practice to use semicolons.

**Case Sensitivity**

JavaScript is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So the identifiers **Time** and **TIME** will convey different meanings in JavaScript.

**NOTE** − Care should be taken while writing variable and function names in JavaScript.

**Comments in JavaScript**

JavaScript supports both C-style and C++-style comments, Thus −

* Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
* Any text between the characters /\* and \*/ is treated as a comment. This may span multiple lines.
* JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
* The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

**Example**

The following example shows how to use comments in JavaScript.

<script language = "javascript" type = "text/javascript">

<!--

// This is a comment. It is similar to comments in C++

/\*

\* This is a multi-line comment in JavaScript

\* It is very similar to comments in C Programming

\*/

//-->

</script>

**Chapter 4. Introduction to Angular JS**

**4.1. What is Angular JS**

**AngularJS** is a very powerful JavaScript Framework. It is used in Single Page Application (SPA) projects. It extends HTML DOM with additional attributes and makes it more responsive to user actions. AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

**Why to Learn AngularJS?**

AngularJS is an open-source web application framework. It was originally developed in 2009 by Misko Hevery and Adam Abrons. It is now maintained by Google. Its latest version is 1.2.21.

* AngularJS is a efficient framework that can create Rich Internet Applications (RIA).
* AngularJS provides developers an options to write client side applications using JavaScript in a clean Model View Controller (MVC) way.
* Applications written in AngularJS are cross-browser compliant. AngularJS automatically handles JavaScript code suitable for each browser.
* AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

Overall, AngularJS is a framework to build large scale, high-performance, and easyto-maintain web applications.

**Applications of AngularJS**

The general features of AngularJS are as follows −

* AngularJS is a efficient framework that can create Rich Internet Applications (RIA).
* AngularJS provides developers an options to write client side applications using JavaScript in a clean Model View Controller (MVC) way.
* Applications written in AngularJS are cross-browser compliant. AngularJS automatically handles JavaScript code suitable for each browser.
* AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

Overall, AngularJS is a framework to build large scale, high-performance, and easyto-maintain web applications.

**Audience**

This tutorial is designed for software professionals who want to learn the basics of AngularJS and its programming concepts in simple and easy steps. It describes the components of AngularJS with suitable examples.

**Prerequisites**

You should have a basic understanding of JavaScript and any text editor. As we are going to develop web-based applications using AngularJS, it will be good if you have an understanding of other web technologies such as HTML, CSS, AJAX, etc.

Now let us write a simple example using AngularJS library. Let us create an HTML file *myfirstexample.html* shown as below −

<!doctype html>

<html>

<head>

<script src = "<https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js>"></script>

</head>

<body ng-app = "myapp">

<div ng-controller = "HelloController" >

<h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>

</div>

<script>

angular.module("myapp", [])

.controller("HelloController", function($scope) {

$scope.helloTo = {};

$scope.helloTo.title = "AngularJS";

});

</script>

</body>

</html>

Let us go through the above code in detail −

**Include AngularJS**

We include the AngularJS JavaScript file in the HTML page so that we can use it −

<head>

<script src = "<https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js>">

</script>

</head>

You can check the latest version of AngularJS on its official website.

**Point to AngularJS app**

Next, it is required to tell which part of HTML contains the AngularJS app. You can do this by adding the ng-app attribute to the root HTML element of the AngularJS app. You can either add it to the html element or the body element as shown below −

<body ng-app = "myapp">

</body>

**View**

The view is this part −

<div ng-controller = "HelloController" >

<h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>

</div>

*ng-controller* tells AngularJS which controller to use with this view. *helloTo.title*tells AngularJS to write the model value named helloTo.title in HTML at this location.

**Controller**

The controller part is −

<script>

angular.module("myapp", [])

.controller("HelloController", function($scope) {

$scope.helloTo = {};

$scope.helloTo.title = "AngularJS";

});

</script>

This code registers a controller function named HelloController in the angular module named *myapp*. We will study more about [modules](https://www.tutorialspoint.com/angularjs/angularjs_modules.htm) and [controllers](https://www.tutorialspoint.com/angularjs/angularjs_controllers.htm) in their respective chapters. The controller function is registered in angular via the angular.module(...).controller(...) function call.

The $scope parameter model is passed to the controller function. The controller function adds a *helloTo* JavaScript object, and in that object it adds a title field.

**Execution**

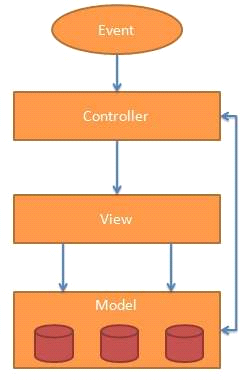
Save the above code as myfirstexample.html and open it in any browser. You get to see the following output

**4.2. AngularJS - MVC Architecture**

**M**odel **V**iew **C**ontroller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts −

* **Model** − It is the lowest level of the pattern responsible for maintaining data.
* **View** − It is responsible for displaying all or a portion of the data to the user.
* **Controller** − It is a software Code that controls the interactions between the Model and View.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



**Fig 4.1 MVC Model**

**The Model**

The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

**The View**

A presentation of data in a particular format, triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

**The Controller**

The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

AngularJS is a MVC based framework. In the coming chapters, we will see how AngularJS uses MVC methodology.

An AngularJS application consists of following three important parts −

* **ng-app** − This directive defines and links an AngularJS application to HTML.
* **ng-model** − This directive binds the values of AngularJS application data to HTML input controls.
* **ng-bind** − This directive binds the AngularJS Application data to HTML tags.

**4.3. AngularJS Data Binding**

Data binding in AngularJS is the synchronization between the model and the view.

**Data Model**

AngularJS applications usually have a data model. The data model is a collection of data available for the application.

**Example**

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.firstname = "John";  
  $scope.lastname = "Doe";  
});

HTML View

The HTML container where the AngularJS application is displayed, is called the view.

The view has access to the model, and there are several ways of displaying model data in the view.

You can use the ng-bind directive, which will bind the inner HTML of the element to the specified model property:

**Example**

<p ng-bind="firstname"></p>

You can also use double braces {{}} to display content from the model:

**Example**

<p>First name: {{firstname}}</p>

Or you can use the ng-model directive on HTML controls to bind the model to the view.

**The ng-model Directive**

Use the ng-model directive to bind data from the model to the view on HTML controls (input, select, textarea)

**Example**

<input ng-model="firstname">

The ng-model directive provides a two-way binding between the model and the view.

**Two-way Binding**

Data binding in AngularJS is the synchronization between the model and the view.

When data in the *model* changes, the *view* reflects the change, and when data in the *view* changes, the *model* is updated as well. This happens immediately and automatically, which makes sure that the model and the view is updated at all times.

**Example**

<div ng-app="myApp" ng-controller="myCtrl">  
  Name: <input ng-model="firstname">  
  <h1>{{firstname}}</h1>  
</div>  
  
<script>

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.firstname = "John";  
  $scope.lastname = "Doe";  
});

**4.4. AngularJS Controller**

Applications in AngularJS are controlled by controllers.

Because of the immediate synchronization of the model and the view, the controller can be completely separated from the view, and simply concentrate on the model data. Thanks to the data binding in AngularJS, the view will reflect any changes made in the controller.

**Example**

<div ng-app="myApp" ng-controller="myCtrl">  
  <h1 ng-click="changeName()">{{firstname}}</h1>  
</div>  
  
<script>

var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.firstname = "John";  
  $scope.changeName = function() {  
    $scope.firstname = "Nelly";  
  }  
});

</script>

**AngularJS Controllers**

AngularJS controllers control the data of AngularJS applications.

AngularJS controllers are regular JavaScript Objects.

AngularJS applications are controlled by controllers.

The **ng-controller** directive defines the application controller.

A controller is a **JavaScript Object**, created by a standard JavaScript **object constructor**.

**AngularJS Example**

<div ng-app="myApp" ng-controller="myCtrl">  
  
First Name: <input type="text" ng-model="firstName"><br>  
Last Name: <input type="text" ng-model="lastName"><br>  
<br>  
Full Name: {{firstName + " " + lastName}}  
  
</div>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
  $scope.firstName = "John";  
  $scope.lastName = "Doe";  
});  
</script>

Application explained:

The AngularJS application is defined by  **ng-app="myApp"**. The application runs inside the <div>.

The **ng-controller="myCtrl"** attribute is an AngularJS directive. It defines a controller.

The **myCtrl** function is a JavaScript function.

AngularJS will invoke the controller with a **$scope** object.

In AngularJS, $scope is the application object (the owner of application variables and functions).

The controller creates two properties (variables) in the scope (**firstName** and **lastName**).

The **ng-model** directives bind the input fields to the controller properties (firstName and lastName).

**Controller Methods**

The example above demonstrated a controller object with two properties: lastName and firstName.

A controller can also have methods (variables as functions):

**AngularJS Example**

<div ng-app="myApp" ng-controller="personCtrl">  
  
First Name: <input type="text" ng-model="firstName"><br>  
Last Name: <input type="text" ng-model="lastName"><br>  
<br>  
Full Name: {{fullName()}}  
  
</div>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('personCtrl', function($scope) {  
  $scope.firstName = "John";  
  $scope.lastName = "Doe";  
  $scope.fullName = function() {  
    return $scope.firstName + " " + $scope.lastName;  
  };  
});  
</script>

**Controllers In External Files**

In larger applications, it is common to store controllers in external files.

Just copy the code between the <script> tags into an external file named [personController.js](https://www.w3schools.com/angular/personController.js):

**AngularJS Example**

<div ng-app="myApp" ng-controller="personCtrl">  
  
First Name: <input type="text" ng-model="firstName"><br>  
Last Name: <input type="text" ng-model="lastName"><br>  
<br>  
Full Name: {{fullName()}}  
  
</div>  
  
<script src="personController.js"></script>

**Another Example**

For the next example we will create a new controller file:

angular.module('myApp', []).controller('namesCtrl', function($scope) {  
  $scope.names = [  
    {name:'Jani',country:'Norway'},  
    {name:'Hege',country:'Sweden'},  
    {name:'Kai',country:'Denmark'}  
  ];  
});

Save the file as [namesController.js](https://www.w3schools.com/angular/namesController.js):

And then use the controller file in an application:

**AngularJS Example**

<div ng-app="myApp" ng-controller="namesCtrl">  
  
<ul>  
  <li ng-repeat="x in names">  
    {{ x.name + ', ' + x.country }}  
  </li>  
</ul>  
  
</div>  
  
<script src="namesController.js"></script>

**4.5. AngularJS Scope**

The scope is the binding part between the HTML (view) and the JavaScript (controller).

The scope is an object with the available properties and methods.

The scope is available for both the view and the controller.

**How to Use the Scope?**

When you make a controller in AngularJS, you pass the $scope object as an argument:

**Example**

Properties made in the controller, can be referred to in the view:

<div ng-app="myApp" ng-controller="myCtrl">  
  
<h1>{{carname}}</h1>  
  
</div>  
  
<script>

var app = angular.module('myApp', []);  
  
app.controller('myCtrl', function($scope) {  
  $scope.carname = "Volvo";  
});

</script>

When adding properties to the $scope object in the controller, the view (HTML) gets access to these properties.

In the view, you do not use the prefix $scope, you just refer to a property name, like {{carname}}.

**Understanding the Scope**

If we consider an AngularJS application to consist of:

* View, which is the HTML.
* Model, which is the data available for the current view.
* Controller, which is the JavaScript function that makes/changes/removes/controls the data.

Then the scope is the Model.

The scope is a JavaScript object with properties and methods, which are available for both the view and the controller.

**Example**

If you make changes in the view, the model and the controller will be updated:

<div ng-app="myApp" ng-controller="myCtrl">  
  
<input ng-model="name">  
  
<h1>My name is {{name}}</h1>  
  
</div>  
  
<script>

var app = angular.module('myApp', []);  
  
app.controller('myCtrl', function($scope) {  
  $scope.name = "John Doe";  
});

</script>

**Know Your Scope**

It is important to know which scope you are dealing with, at any time.

In the two examples above there is only one scope, so knowing your scope is not an issue, but for larger applications there can be sections in the HTML DOM which can only access certain scopes.

**Example**

When dealing with the ng-repeat directive, each repetition has access to the current repetition object:

<div ng-app="myApp" ng-controller="myCtrl">  
  
<ul>  
  <li ng-repeat="x in names">{{x}}</li>  
</ul>  
  
</div>  
  
<script>

var app = angular.module('myApp', []);  
  
app.controller('myCtrl', function($scope) {  
  $scope.names = ["Emil", "Tobias", "Linus"];  
});

</script>

Each <li> element has access to the current repetition object, in this case a string, which is referred to by using x.

**Root Scope**

All applications have a $rootScope which is the scope created on the HTML element that contains the ng-app directive.

The rootScope is available in the entire application.

If a variable has the same name in both the current scope and in the rootScope, the application uses the one in the current scope.

**Example**

A variable named "color" exists in both the controller's scope and in the rootScope:

<body ng-app="myApp">  
  
<p>The rootScope's favorite color:</p>  
<h1>{{color}}</h1>  
  
<div ng-controller="myCtrl">  
  <p>The scope of the controller's favorite color:</p>  
  <h1>{{color}}</h1>  
</div>  
  
<p>The rootScope's favorite color is still:</p>  
<h1>{{color}}</h1>  
  
<script>

var app = angular.module('myApp', []);  
app.run(function($rootScope) {  
  $rootScope.color = 'blue';  
});  
app.controller('myCtrl', function($scope) {  
  $scope.color = "red";  
});

</script>  
</body>

**4.6.AngularJS Tables**

The ng-repeat directive is perfect for displaying tables.

**Displaying Data in a Table**

Displaying tables with angular is very simple:

**AngularJS Example**

<div ng-app="myApp" ng-controller="customersCtrl">  
  
<table>  
  <tr ng-repeat="x in names">  
    <td>{{ x.Name }}</td>  
    <td>{{ x.Country }}</td>  
  </tr>  
</table>  
  
</div>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('customersCtrl', function($scope, $http) {  
  $http.get("customers.php")  
  .then(function (response) {$scope.names = response.data.records;});  
});  
</script>

**Displaying with CSS Style**

To make it nice, add some CSS to the page:

**CSS Style**

<style>  
table, th , td {  
  border: 1px solid grey;  
  border-collapse: collapse;  
  padding: 5px;  
}  
  
table tr:nth-child(odd) {  
  background-color: #f1f1f1;  
}  
  
table tr:nth-child(even) {  
  background-color: #ffffff;  
}  
</style>

**Display with orderBy Filter**

To sort the table, add an **orderBy** filter:

**AngularJS Example**

<table>  
  <tr ng-repeat="x in names | orderBy : 'Country'">  
    <td>{{ x.Name }}</td>  
    <td>{{ x.Country }}</td>  
  </tr>  
</table>

**Display with uppercase Filter**

To display uppercase, add an **uppercase** filter:

**AngularJS Example**

<table>  
  <tr ng-repeat="x in names">  
    <td>{{ x.Name }}</td>  
    <td>{{ x.Country | uppercase }}</td>  
  </tr>  
</table>

**Display the Table Index ($index)**

To display the table index, add a <td> with **$index**:

**AngularJS Example**

<table>  
  <tr ng-repeat="x in names">  
    <td>{{ $index + 1 }}</td>  
    <td>{{ x.Name }}</td>  
    <td>{{ x.Country }}</td>  
  </tr>  
</table>

**4.7. AngularJS Forms**

Forms in AngularJS provides data-binding and validation of input controls.

**Input Controls**

Input controls are the HTML input elements:

* input elements
* select elements
* button elements
* textarea elements

**Data-Binding**

Input controls provides data-binding by using the ng-model directive.

<input type="text" ng-model="firstname">

The application does now have a property named firstname.

The ng-model directive binds the input controller to the rest of your application.

The property firstname, can be referred to in a controller:

**Example**

<script>  
var app = angular.module('myApp', []);  
app.controller('formCtrl', function($scope) {  
  $scope.firstname = "John";  
});  
</script>

It can also be referred to elsewhere in the application:

**Example**

<form>  
  First Name: <input type="text" ng-model="firstname">  
</form>  
  
<h1>You entered: {{firstname}}</h1>

**Checkbox**

A checkbox has the value true or false. Apply the ng-model directive to a checkbox, and use its value in your application.

**Example**

Show the header if the checkbox is checked:

 <form>  
  Check to show a header:  
  <input type="checkbox" ng-model="myVar">  
</form>  
  
<h1 ng-show="myVar">My Header</h1>

**Radiobuttons**

Bind radio buttons to your application with the ng-model directive.

Radio buttons with the same ng-model can have different values, but only the selected one will be used.

**Example**

Display some text, based on the value of the selected radio button:

<form>  
  Pick a topic:  
  <input type="radio" ng-model="myVar" value="dogs">Dogs  
  <input type="radio" ng-model="myVar" value="tuts">Tutorials  
  <input type="radio" ng-model="myVar" value="cars">Cars  
</form>

The value of myVar will be either dogs, tuts, or cars.

**Selectbox**

Bind select boxes to your application with the ng-model directive.

The property defined in the ng-model attribute will have the value of the selected option in the selectbox.

**Example**

Display some text, based on the value of the selected option:

<form>  
  Select a topic:  
  <select ng-model="myVar">  
    <option value="">  
    <option value="dogs">Dogs  
    <option value="tuts">Tutorials  
    <option value="cars">Cars  
  </select>  
</form>

The value of myVar will be either dogs, tuts, or cars.

**An AngularJS Form Example**

First Name :  
  
Last Name :  
  
  
RESET

form = {"firstName":"John","lastName":"Doe"}

master = {"firstName":"John","lastName":"Doe"}

**Application Code**

<div ng-app="myApp" ng-controller="formCtrl">  
  <form novalidate>  
    First Name:<br>  
    <input type="text" ng-model="user.firstName"><br>  
    Last Name:<br>  
    <input type="text" ng-model="user.lastName">  
    <br><br>  
    <button ng-click="reset()">RESET</button>  
  </form>  
  <p>form = {{user}}</p>  
  <p>master = {{master}}</p>  
</div>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('formCtrl', function($scope) {  
  $scope.master = {firstName: "John", lastName: "Doe"};  
  $scope.reset = function() {  
    $scope.user = angular.copy($scope.master);  
  };  
  $scope.reset();  
});  
</script>

**4.8. AngularJS Form Validation**

AngularJS can validate input data.

**Form Validation**

AngularJS offers client-side form validation.

AngularJS monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state.

AngularJS also holds information about whether they have been touched, or modified, or not.

You can use standard HTML5 attributes to validate input, or you can make your own validation functions.

Client-side validation cannot alone secure user input. Server side validation is also necessary.

**Required**

Use the HTML5 attribute required to specify that the input field must be filled out:

**Example**

The input field is required:

<form name="myForm">  
  <input name="myInput" ng-model="myInput" required>  
</form>  
  
<p>The input's valid state is:</p>  
<h1>{{myForm.myInput.$valid}}</h1>

**E-mail**

Use the HTML5 type email to specify that the value must be an e-mail:

**Example**

The input field has to be an e-mail:

<form name="myForm">  
  <input name="myInput" ng-model="myInput" type="email">  
</form>  
  
<p>The input's valid state is:</p>  
<h1>{{myForm.myInput.$valid}}</h1>

**Form State and Input State**

AngularJS is constantly updating the state of both the form and the input fields.

Input fields have the following states:

* $untouched The field has not been touched yet
* $touched The field has been touched
* $pristine The field has not been modified yet
* $dirty The field has been modified
* $invalid The field content is not valid
* $valid The field content is valid

They are all properties of the input field, and are either true or false.

Forms have the following states:

* $pristine No fields have been modified yet
* $dirty One or more have been modified
* $invalid The form content is not valid
* $valid The form content is valid
* $submitted The form is submitted

They are all properties of the form, and are either true or false.

You can use these states to show meaningful messages to the user. Example, if a field is required, and the user leaves it blank, you should give the user a warning:

**Example**

Show an error message if the field has been touched AND is empty:

<input name="myName" ng-model="myName" required>  
<span ng-show="myForm.myName.$touched && myForm.myName.$invalid">The name is required.</span>

**CSS Classes**

AngularJS adds CSS classes to forms and input fields depending on their states.

The following classes are added to, or removed from, input fields:

* ng-untouched The field has not been touched yet
* ng-touched The field has been touched
* ng-pristine The field has not been  modified yet
* ng-dirty The field has been modified
* ng-valid The field content is valid
* ng-invalid The field content is not valid
* ng-valid-*key* One *key* for each validation. Example: ng-valid-required, useful when there are more than one thing that must be validated
* ng-invalid-*key* Example: ng-invalid-required

The following classes are added to, or removed from, forms:

* ng-pristine No fields has not been modified yet
* ng-dirty One or more fields has been modified
* ng-valid The form content is valid
* ng-invalid The form content is not valid
* ng-valid-*key* One *key* for each validation. Example: ng-valid-required, useful when there are more than one thing that must be validated
* ng-invalid-*key* Example: ng-invalid-required

The classes are removed if the value they represent is false.

Add styles for these classes to give your application a better and more intuitive user interface.

**Example**

Apply styles, using standard CSS:

<style>

input.ng-invalid {  
  background-color: pink;  
}  
input.ng-valid {  
  background-color: lightgreen;  
}

</style>

Forms can also be styled:

**Example**

Apply styles for unmodified (pristine) forms, and for modified forms:

<style>

form.ng-pristine {  
  background-color: lightblue;  
}  
form.ng-dirty {  
  background-color: pink;  
}

</style>

**Custom Validation**

To create your own validation function is a bit more tricky; You have to add a new directive to your application, and deal with the validation inside a function with certain specified arguments.

**Example**

Create your own directive, containing a custom validation function, and refer to it by using my-directive.

The field will only be valid if the value contains the character "e":

<form name="myForm">  
<input name="myInput" ng-model="myInput" required my-directive>  
</form>  
  
<script>

var app = angular.module('myApp', []);  
app.directive('myDirective', function() {  
  return {  
    require: 'ngModel',  
    link: function(scope, element, attr, mCtrl) {  
      function myValidation(value) {  
        if (value.indexOf("e") > -1) {  
          mCtrl.$setValidity('charE', true);  
        } else {  
          mCtrl.$setValidity('charE', false);  
        }  
        return value;  
      }  
      mCtrl.$parsers.push(myValidation);  
    }  
  };  
});

</script>

**Example Explained:**

In HTML, the new directive will be referred to by using the attribute my-directive.

In the JavaScript we start by adding a new directive named myDirective.

Remember, when naming a directive, you must use a camel case name, myDirective, but when invoking it, you must use - separated name, my-directive.

Then, return an object where you specify that we require  ngModel, which is the ngModelController.

Make a linking function which takes some arguments, where the fourth argument, mCtrl, is the ngModelController,

Then specify a function, in this case named myValidation, which takes one argument, this argument is the value of the input element.

Test if the value contains the letter "e", and set the validity of the model controller to either true or false.

At last, mCtrl.$parsers.push(myValidation); will add the myValidation function to an array of other functions, which will be executed every time the input value changes.

**Validation Example**

<!DOCTYPE html>  
<html>  
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>  
<body>  
  
<h2>Validation Example</h2>  
  
<form  ng-app="myApp"  ng-controller="validateCtrl"  
name="myForm" novalidate>  
  
<p>Username:<br>  
  <input type="text" name="user" ng-model="user" required>  
  <span style="color:red" ng-show="myForm.user.$dirty && myForm.user.$invalid">  
  <span ng-show="myForm.user.$error.required">Username is required.</span>  
  </span>  
</p>  
  
<p>Email:<br>  
  <input type="email" name="email" ng-model="email" required>  
  <span style="color:red" ng-show="myForm.email.$dirty && myForm.email.$invalid">  
  <span ng-show="myForm.email.$error.required">Email is required.</span>  
  <span ng-show="myForm.email.$error.email">Invalid email address.</span>  
  </span>  
</p>  
  
<p>  
  <input type="submit"  
  ng-disabled="myForm.user.$dirty && myForm.user.$invalid ||  
  myForm.email.$dirty && myForm.email.$invalid">  
</p>  
  
</form>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('validateCtrl', function($scope) {  
  $scope.user = 'John Doe';  
  $scope.email = 'john.doe@gmail.com';  
});  
</script>  
  
</body>  
</html>

**6.4 About the Project**

A Platform (stages) game

Picture of a game character jumping over lava



**The game**

The dark box represents the player, whose task is to collect the yellow boxes (coins) while avoiding the red stuff (lava). A level is completed when all coins have been collected.

The player can walk around with the left and right arrow keys and can jump with the up arrow. Jumping is a specialty of this game character. It can reach several times its own height and can change direction in midair. This may not be entirely realistic, but it helps give the player the feeling of being in direct control of the on-screen avatar.

The game consists of a static background, laid out like a grid, with the moving elements overlaid on that background. Each field on the grid is either empty, solid, or lava. The moving elements are the player, coins, and certain pieces of lava. The positions of these elements are not constrained to the grid—their coordinates may be fractional, allowing smooth motion.

The technology

I will use the browser DOM to display the game, and we’ll read user input by handling key events.

The screen- and keyboard-related code is only a small part of the work we need to do to build this game. Since everything looks like colored boxes, drawing is uncomplicated: we create DOM elements and use styling to give them a background color, size, and position.

In games and other programs that should animate graphics and respond to user input without noticeable delay, efficiency is important. Although the DOM was not originally designed for high-performance graphics, it is actually better at this than you would expect. You saw some animations in Chapter 14. On a modern machine, a simple game like this performs well, even if we don’t worry about optimization very much.

Levels

Its a human-readable, human-editable way to specify levels. Since it is okay for everything to start out on a grid, we could use big strings in which each character represents an element—either a part of the background grid or a moving element.

The plan for a small level might look like this:

edit & run code by clicking itlet simpleLevelPlan = `

......................

..#................#..

..#..............=.#..

..#.........o.o....#..

..#.@......#####...#..

..#####............#..

......#++++++++++++#..

......##############..

......................`;

Periods are empty space, hash (#) characters are walls, and plus signs are lava. The player’s starting position is the at sign (@). Every O character is a coin, and the equal sign (=) at the top is a block of lava that moves back and forth horizontally.

We’ll support two additional kinds of moving lava: the pipe character (|) creates vertically moving blobs, and v indicates dripping lava—vertically moving lava that doesn’t bounce back and forth but only moves down, jumping back to its start position when it hits the floor.

A whole game consists of multiple levels that the player must complete. A level is completed when all coins have been collected. If the player touches lava, the current level is restored to its starting position, and the player may try again.

Reading a level

The following class stores a level object. Its argument should be the string that defines the level.

class Level {

constructor(plan) {

let rows = plan.trim().split("\n").map(l => [...l]);

this.height = rows.length;

this.width = rows[0].length;

this.startActors = [];

this.rows = rows.map((row, y) => {

return row.map((ch, x) => {

let type = levelChars[ch];

if (typeof type == "string") return type;

this.startActors.push(

type.create(new Vec(x, y), ch));

return "empty";

});

});

}

}

The trim method is used to remove whitespace at the start and end of the plan string. This allows our example plan to start with a newline so that all the lines are directly below each other. The remaining string is split on newline characters, and each line is spread into an array, producing arrays of characters.

So rows holds an array of arrays of characters, the rows of the plan. We can derive the level’s width and height from these. But we must still separate the moving elements from the background grid. We’ll call moving elements actors. They’ll be stored in an array of objects. The background will be an array of arrays of strings, holding field types such as "empty", "wall", or "lava".

To create these arrays, we map over the rows and then over their content. Remember that map passes the array index as a second argument to the mapping function, which tells us the x- and y-coordinates of a given character. Positions in the game will be stored as pairs of coordinates, with the top left being 0,0 and each background square being 1 unit high and wide.

To interpret the characters in the plan, the Level constructor uses the levelChars object, which maps background elements to strings and actor characters to classes. When type is an actor class, its static create method is used to create an object, which is added to startActors, and the mapping function returns "empty" for this background square.The position of the actor is stored as a Vec object.

As the game runs, actors will end up in different places or even disappear entirely (as coins do when collected). We’ll use a State class to track the state of a running game.

class State {

constructor(level, actors, status) {

this.level = level;

this.actors = actors;

this.status = status;

}

static start(level) {

return new State(level, level.startActors, "playing");

}

get player() {

return this.actors.find(a => a.type == "player");

}

}

The status property will switch to "lost" or "won" when the game has ended.

This is again a persistent data structure—updating the game state creates a new state and leaves the old one intact.

Actors

Actor objects represent the current position and state of a given moving element in our game. All actor objects conform to the same interface. Their pos property holds the coordinates of the element’s top-left corner, and their size property holds its size.

Then they have an update method, which is used to compute their new state and position after a given time step. It simulates the thing the actor does—moving in response to the arrow keys for the player and bouncing back and forth for the lava—and returns a new, updated actor object.

A type property contains a string that identifies the type of the actor—"player", "coin", or "lava". This is useful when drawing the game—the look of the rectangle drawn for an actor is based on its type.

Actor classes have a static create method that is used by the Level constructor to create an actor from a character in the level plan. It is given the coordinates of the character and the character itself, which is needed because the Lava class handles several different characters.

This is the Vec class that we’ll use for our two-dimensional values, such as the position and size of actors.

class Vec {

constructor(x, y) {

this.x = x; this.y = y;

}

plus(other) {

return new Vec(this.x + other.x, this.y + other.y);

}

times(factor) {

return new Vec(this.x \* factor, this.y \* factor);

}

}

The times method scales a vector by a given number. It will be useful when we need to multiply a speed vector by a time interval to get the distance traveled during that time.

The different types of actors get their own classes since their behavior is very different. Let’s define these classes. We’ll get to their update methods later.

The player class has a property speed that stores its current speed to simulate momentum and gravity.

class Player {

constructor(pos, speed) {

this.pos = pos;

this.speed = speed;

}

get type() { return "player"; }

static create(pos) {

return new Player(pos.plus(new Vec(0, -0.5)),

new Vec(0, 0));

}

}

Player.prototype.size = new Vec(0.8, 1.5);

Because a player is one-and-a-half squares high, its initial position is set to be half a square above the position where the @ character appeared. This way, its bottom aligns with the bottom of the square it appeared in.

The size property is the same for all instances of Player, so we store it on the prototype rather than on the instances themselves. We could have used a getter like type, but that would create and return a new Vec object every time the property is read, which would be wasteful. (Strings, being immutable, don’t have to be re-created every time they are evaluated.)

When constructing a Lava actor, we need to initialize the object differently depending on the character it is based on. Dynamic lava moves along at its current speed until it hits an obstacle. At that point, if it has a reset property, it will jump back to its start position (dripping). If it does not, it will invert its speed and continue in the other direction (bouncing).

The create method looks at the character that the Level constructor passes and creates the appropriate lava actor.

class Lava {

constructor(pos, speed, reset) {

this.pos = pos;

this.speed = speed;

this.reset = reset;

}

get type() { return "lava"; }

static create(pos, ch) {

if (ch == "=") {

return new Lava(pos, new Vec(2, 0));

} else if (ch == "|") {

return new Lava(pos, new Vec(0, 2));

} else if (ch == "v") {

return new Lava(pos, new Vec(0, 3), pos);

}

}

}

Lava.prototype.size = new Vec(1, 1);

Coin actors are relatively simple. They mostly just sit in their place. But to liven up the game a little, they are given a “wobble”, a slight vertical back-and-forth motion. To track this, a coin object stores a base position as well as a wobble property that tracks the phase of the bouncing motion. Together, these determine the coin’s actual position (stored in the pos property).

class Coin {

constructor(pos, basePos, wobble) {

this.pos = pos;

this.basePos = basePos;

this.wobble = wobble;

}

get type() { return "coin"; }

static create(pos) {

let basePos = pos.plus(new Vec(0.2, 0.1));

return new Coin(basePos, basePos,

Math.random() \* Math.PI \* 2);

}

}

Coin.prototype.size = new Vec(0.6, 0.6);

In Chapter 14, we saw that Math.sin gives us the y-coordinate of a point on a circle. That coordinate goes back and forth in a smooth waveform as we move along the circle, which makes the sine function useful for modeling a wavy motion.

To avoid a situation where all coins move up and down synchronously, the starting phase of each coin is randomized. The phase of Math.sin’s wave, the width of a wave it produces, is 2π. We multiply the value returned by Math.random by that number to give the coin a random starting position on the wave.

We can now define the levelChars object that maps plan characters to either background grid types or actor classes.

const levelChars = {

".": "empty", "#": "wall", "+": "lava",

"@": Player, "o": Coin,

"=": Lava, "|": Lava, "v": Lava

};

That gives us all the parts needed to create a Level instance.

let simpleLevel = new Level(simpleLevelPlan);

console.log(`${simpleLevel.width} by ${simpleLevel.height}`);

// → 22 by 9

The task ahead is to display such levels on the screen and to model time and motion inside them.

Encapsulation as a burden

Most of the code in this chapter does not worry about encapsulation very much for two reasons. First, encapsulation takes extra effort. It makes programs bigger and requires additional concepts and interfaces to be introduced. Since there is only so much code you can throw at a reader before their eyes glaze over, I’ve made an effort to keep the program small.

Second, the various elements in this game are so closely tied together that if the behavior of one of them changed, it is unlikely that any of the others would be able to stay the same. Interfaces between the elements would end up encoding a lot of assumptions about the way the game works. This makes them a lot less effective—whenever you change one part of the system, you still have to worry about the way it impacts the other parts because their interfaces wouldn’t cover the new situation.

Some cutting points in a system lend themselves well to separation through rigorous interfaces, but others don’t. Trying to encapsulate something that isn’t a suitable boundary is a sure way to waste a lot of energy. When you are making this mistake, you’ll usually notice that your interfaces are getting awkwardly large and detailed and that they need to be changed often, as the program evolves.

There is one thing that we will encapsulate, and that is the drawing subsystem. The reason for this is that we’ll display the same game in a different way in the next chapter. By putting the drawing behind an interface, we can load the same game program there and plug in a new display module.

Drawing

The encapsulation of the drawing code is done by defining a display object, which displays a given level and state. The display type we define in this chapter is called DOMDisplay because it uses DOM elements to show the level.

We’ll be using a style sheet to set the actual colors and other fixed properties of the elements that make up the game. It would also be possible to directly assign to the elements’ style property when we create them, but that would produce more verbose programs.

The following helper function provides a succinct way to create an element and give it some attributes and child nodes:

function elt(name, attrs, ...children) {

let dom = document.createElement(name);

for (let attr of Object.keys(attrs)) {

dom.setAttribute(attr, attrs[attr]);

}

for (let child of children) {

dom.appendChild(child);

}

return dom;

}

A display is created by giving it a parent element to which it should append itself and a level object.

class DOMDisplay {

constructor(parent, level) {

this.dom = elt("div", {class: "game"}, drawGrid(level));

this.actorLayer = null;

parent.appendChild(this.dom);

}

clear() { this.dom.remove(); }

}

The level’s background grid, which never changes, is drawn once. Actors are redrawn every time the display is updated with a given state. The actorLayer property will be used to track the element that holds the actors so that they can be easily removed and replaced.

Our coordinates and sizes are tracked in grid units, where a size or distance of 1 means one grid block. When setting pixel sizes, we will have to scale these coordinates up—everything in the game would be ridiculously small at a single pixel per square. The scale constant gives the number of pixels that a single unit takes up on the screen.

const scale = 20;

function drawGrid(level) {

return elt("table", {

class: "background",

style: `width: ${level.width \* scale}px`

}, ...level.rows.map(row =>

elt("tr", {style: `height: ${scale}px`},

...row.map(type => elt("td", {class: type})))

));

}

As mentioned, the background is drawn as a <table> element. This nicely corresponds to the structure of the rows property of the level—each row of the grid is turned into a table row (<tr> element). The strings in the grid are used as class names for the table cell (<td>) elements. The spread (triple dot) operator is used to pass arrays of child nodes to elt as separate arguments.

The following CSS makes the table look like the background we want:

.background { background: rgb(52, 166, 251);

table-layout: fixed;

border-spacing: 0; }

.background td { padding: 0; }

.lava { background: rgb(255, 100, 100); }

.wall { background: white; }

Some of these (table-layout, border-spacing, and padding) are used to suppress unwanted default behavior. We don’t want the layout of the table to depend upon the contents of its cells, and we don’t want space between the table cells or padding inside them.

The background rule sets the background color. CSS allows colors to be specified both as words (white) or with a format such as rgb(R, G, B), where the red, green, and blue components of the color are separated into three numbers from 0 to 255. So, in rgb(52, 166, 251), the red component is 52, green is 166, and blue is 251. Since the blue component is the largest, the resulting color will be bluish. You can see that in the .lava rule, the first number (red) is the largest.

I draw each actor by creating a DOM element for it and setting that element’s position and size based on the actor’s properties. The values have to be multiplied by scale to go from game units to pixels.

function drawActors(actors) {

return elt("div", {}, ...actors.map(actor => {

let rect = elt("div", {class: `actor ${actor.type}`});

rect.style.width = `${actor.size.x \* scale}px`;

rect.style.height = `${actor.size.y \* scale}px`;

rect.style.left = `${actor.pos.x \* scale}px`;

rect.style.top = `${actor.pos.y \* scale}px`;

return rect;

}));

}

To give an element more than one class, we separate the class names by spaces. In the CSS code shown next, the actor class gives the actors their absolute position. Their type name is used as an extra class to give them a color. We don’t have to define the lava class again because we’re reusing the class for the lava grid squares we defined earlier.

.actor { position: absolute; }

.coin { background: rgb(241, 229, 89); }

.player { background: rgb(64, 64, 64); }

The syncState method is used to make the display show a given state. It first removes the old actor graphics, if any, and then redraws the actors in their new positions. It may be tempting to try to reuse the DOM elements for actors, but to make that work, we would need a lot of additional bookkeeping to associate actors with DOM elements and to make sure we remove elements when their actors vanish. Since there will typically be only a handful of actors in the game, redrawing all of them is not expensive.

DOMDisplay.prototype.syncState = function(state) {

if (this.actorLayer) this.actorLayer.remove();

this.actorLayer = drawActors(state.actors);

this.dom.appendChild(this.actorLayer);

this.dom.className = `game ${state.status}`;

this.scrollPlayerIntoView(state);

};

By adding the level’s current status as a class name to the wrapper, we can style the player actor slightly differently when the game is won or lost by adding a CSS rule that takes effect only when the player has an ancestor element with a given class.

.lost .player {

background: rgb(160, 64, 64);

}

.won .player {

box-shadow: -4px -7px 8px white, 4px -7px 8px white;

}

After touching lava, the player’s color turns dark red, suggesting scorching. When the last coin has been collected, we add two blurred white shadows—one to the top left and one to the top right—to create a white halo effect.

We can’t assume that the level always fits in the viewport—the element into which we draw the game. That is why the scrollPlayerIntoView call is needed. It ensures that if the level is protruding outside the viewport, we scroll that viewport to make sure the player is near its center. The following CSS gives the game’s wrapping DOM element a maximum size and ensures that anything that sticks out of the element’s box is not visible. We also give it a relative position so that the actors inside it are positioned relative to the level’s top-left corner.

.game {

overflow: hidden;

max-width: 600px;

max-height: 450px;

position: relative;

}

In the scrollPlayerIntoView method, we find the player’s position and update the wrapping element’s scroll position. We change the scroll position by manipulating that element’s scrollLeft and scrollTop properties when the player is too close to the edge.

DOMDisplay.prototype.scrollPlayerIntoView = function(state) {

let width = this.dom.clientWidth;

let height = this.dom.clientHeight;

let margin = width / 3;

// The viewport

let left = this.dom.scrollLeft, right = left + width;

let top = this.dom.scrollTop, bottom = top + height;

let player = state.player;

let center = player.pos.plus(player.size.times(0.5))

.times(scale);

if (center.x < left + margin) {

this.dom.scrollLeft = center.x - margin;

} else if (center.x > right - margin) {

this.dom.scrollLeft = center.x + margin - width;

}

if (center.y < top + margin) {

this.dom.scrollTop = center.y - margin;

} else if (center.y > bottom - margin) {

this.dom.scrollTop = center.y + margin - height;

}

};

The way the player’s center is found shows how the methods on our Vec type allow computations with objects to be written in a relatively readable way. To find the actor’s center, we add its position (its top-left corner) and half its size. That is the center in level coordinates, but we need it in pixel coordinates, so we then multiply the resulting vector by our display scale.

Next, a series of checks verifies that the player position isn’t outside of the allowed range. Note that sometimes this will set nonsense scroll coordinates that are below zero or beyond the element’s scrollable area. This is okay—the DOM will constrain them to acceptable values. Setting scrollLeft to -10 will cause it to become 0.

It would have been slightly simpler to always try to scroll the player to the center of the viewport. But this creates a rather jarring effect. As you are jumping, the view will constantly shift up and down. It is more pleasant to have a “neutral” area in the middle of the screen where you can move around without causing any scrolling.

We are now able to display our tiny level.

<link rel="stylesheet" href="css/game.css">

<script>

let simpleLevel = new Level(simpleLevelPlan);

let display = new DOMDisplay(document.body, simpleLevel);

display.syncState(State.start(simpleLevel));

</script>

The <link> tag, when used with rel="stylesheet", is a way to load a CSS file into a page. The file game.css contains the styles necessary for our game.

Now we’re at the point where we can start adding motion—the most interesting aspect of the game. The basic approach, taken by most games like this, is to split time into small steps and, for each step, move the actors by a distance corresponding to their speed multiplied by the size of the time step. We’ll measure time in seconds, so speeds are expressed in units per second.

Moving things is easy. The difficult part is dealing with the interactions between the elements. When the player hits a wall or floor, they should not simply move through it. The game must notice when a given motion causes an object to hit another object and respond accordingly. For walls, the motion must be stopped. When hitting a coin, it must be collected. When touching lava, the game should be lost.

Solving this for the general case is a big task. You can find libraries, usually called physics engines, that simulate interaction between physical objects in two or three dimensions. We’ll take a more modest approach in this chapter, handling only collisions between rectangular objects and handling them in a rather simplistic way.

Before moving the player or a block of lava, we test whether the motion would take it inside of a wall. If it does, we simply cancel the motion altogether. The response to such a collision depends on the type of actor—the player will stop, whereas a lava block will bounce back.

This approach requires our time steps to be rather small since it will cause motion to stop before the objects actually touch. If the time steps (and thus the motion steps) are too big, the player would end up hovering a noticeable distance above the ground. Another approach, arguably better but more complicated, would be to find the exact collision spot and move there. We will take the simple approach and hide its problems by ensuring the animation proceeds in small steps.

This method tells us whether a rectangle (specified by a position and a size) touches a grid element of the given type.

Level.prototype.touches = function(pos, size, type) {

var xStart = Math.floor(pos.x);

var xEnd = Math.ceil(pos.x + size.x);

var yStart = Math.floor(pos.y);

var yEnd = Math.ceil(pos.y + size.y);

for (var y = yStart; y < yEnd; y++) {

for (var x = xStart; x < xEnd; x++) {

let isOutside = x < 0 || x >= this.width ||

y < 0 || y >= this.height;

let here = isOutside ? "wall" : this.rows[y][x];

if (here == type) return true;

}

}

return false;

};

The method computes the set of grid squares that the body overlaps with by using Math.floor and Math.ceil on its coordinates. Remember that grid squares are 1 by 1 units in size. By rounding the sides of a box up and down, we get the range of background squares that the box touches.

Finding collisions on a grid

We loop over the block of grid squares found by rounding the coordinates and return true when a matching square is found. Squares outside of the level are always treated as "wall" to ensure that the player can’t leave the world and that we won’t accidentally try to read outside of the bounds of our rows array.

The state update method uses touches to figure out whether the player is touching lava.

State.prototype.update = function(time, keys) {

let actors = this.actors

.map(actor => actor.update(time, this, keys));

let newState = new State(this.level, actors, this.status);

if (newState.status != "playing") return newState;

let player = newState.player;

if (this.level.touches(player.pos, player.size, "lava")) {

return new State(this.level, actors, "lost");

}

for (let actor of actors) {

if (actor != player && overlap(actor, player)) {

newState = actor.collide(newState);

}

}

return newState;

};

The method is passed a time step and a data structure that tells it which keys are being held down. The first thing it does is call the update method on all actors, producing an array of updated actors. The actors also get the time step, the keys, and the state, so that they can base their update on those. Only the player will actually read keys, since that’s the only actor that’s controlled by the keyboard.

If the game is already over, no further processing has to be done (the game can’t be won after being lost, or vice versa). Otherwise, the method tests whether the player is touching background lava. If so, the game is lost, and we’re done. Finally, if the game really is still going on, it sees whether any other actors overlap the player.

Overlap between actors is detected with the overlap function. It takes two actor objects and returns true when they touch—which is the case when they overlap both along the x-axis and along the y-axis.

function overlap(actor1, actor2) {

return actor1.pos.x + actor1.size.x > actor2.pos.x &&

actor1.pos.x < actor2.pos.x + actor2.size.x &&

actor1.pos.y + actor1.size.y > actor2.pos.y &&

actor1.pos.y < actor2.pos.y + actor2.size.y;

}

If any actor does overlap, its collide method gets a chance to update the state. Touching a lava actor sets the game status to "lost". Coins vanish when you touch them and set the status to "won" when they are the last coin of the level.

Lava.prototype.collide = function(state) {

return new State(state.level, state.actors, "lost");

};

Coin.prototype.collide = function(state) {

let filtered = state.actors.filter(a => a != this);

let status = state.status;

if (!filtered.some(a => a.type == "coin")) status = "won";

return new State(state.level, filtered, status);

};

Actor updates

Actor objects’ update methods take as arguments the time step, the state object, and a keys object. The one for the Lava actor type ignores the keys object.

Lava.prototype.update = function(time, state) {

let newPos = this.pos.plus(this.speed.times(time));

if (!state.level.touches(newPos, this.size, "wall")) {

return new Lava(newPos, this.speed, this.reset);

} else if (this.reset) {

return new Lava(this.reset, this.speed, this.reset);

} else {

return new Lava(this.pos, this.speed.times(-1));

}

};

This update method computes a new position by adding the product of the time step and the current speed to its old position. If no obstacle blocks that new position, it moves there. If there is an obstacle, the behavior depends on the type of the lava block—dripping lava has a reset position, to which it jumps back when it hits something. Bouncing lava inverts its speed by multiplying it by -1 so that it starts moving in the opposite direction.

Coins use their update method to wobble. They ignore collisions with the grid since they are simply wobbling around inside of their own square.

const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {

let wobble = this.wobble + time \* wobbleSpeed;

let wobblePos = Math.sin(wobble) \* wobbleDist;

return new Coin(this.basePos.plus(new Vec(0, wobblePos)),

this.basePos, wobble);

};

The wobble property is incremented to track time and then used as an argument to Math.sin to find the new position on the wave. The coin’s current position is then computed from its base position and an offset based on this wave.

That leaves the player itself. Player motion is handled separately per axis because hitting the floor should not prevent horizontal motion, and hitting a wall should not stop falling or jumping motion.

const playerXSpeed = 7;

const gravity = 30;

const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {

let xSpeed = 0;

if (keys.ArrowLeft) xSpeed -= playerXSpeed;

if (keys.ArrowRight) xSpeed += playerXSpeed;

let pos = this.pos;

let movedX = pos.plus(new Vec(xSpeed \* time, 0));

if (!state.level.touches(movedX, this.size, "wall")) {

pos = movedX;

}

let ySpeed = this.speed.y + time \* gravity;

let movedY = pos.plus(new Vec(0, ySpeed \* time));

if (!state.level.touches(movedY, this.size, "wall")) {

pos = movedY;

} else if (keys.ArrowUp && ySpeed > 0) {

ySpeed = -jumpSpeed;

} else {

ySpeed = 0;

}

return new Player(pos, new Vec(xSpeed, ySpeed));

};

The horizontal motion is computed based on the state of the left and right arrow keys. When there’s no wall blocking the new position created by this motion, it is used. Otherwise, the old position is kept.

Vertical motion works in a similar way but has to simulate jumping and gravity. The player’s vertical speed (ySpeed) is first accelerated to account for gravity.

We check for walls again. If we don’t hit any, the new position is used. If there is a wall, there are two possible outcomes. When the up arrow is pressed and we are moving down (meaning the thing we hit is below us), the speed is set to a relatively large, negative value. This causes the player to jump. If that is not the case, the player simply bumped into something, and the speed is set to zero.

The gravity strength, jumping speed, and pretty much all other constants in this game have been set by trial and error. I tested values until I found a combination I liked.

Tracking keys

For a game like this, we do not want keys to take effect once per keypress. Rather, we want their effect (moving the player figure) to stay active as long as they are held.

We need to set up a key handler that stores the current state of the left, right, and up arrow keys. We will also want to call preventDefault for those keys so that they don’t end up scrolling the page.

The following function, when given an array of key names, will return an object that tracks the current position of those keys. It registers event handlers for "keydown" and "keyup" events and, when the key code in the event is present in the set of codes that it is tracking, updates the object.

function trackKeys(keys) {

let down = Object.create(null);

function track(event) {

if (keys.includes(event.key)) {

down[event.key] = event.type == "keydown";

event.preventDefault();

}

}

window.addEventListener("keydown", track);

window.addEventListener("keyup", track);

return down;

}

const arrowKeys =

trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);

The same handler function is used for both event types. It looks at the event object’s type property to determine whether the key state should be updated to true ("keydown") or false ("keyup").

Running the game

The requestAnimationFrame function, which we saw in Chapter 14, provides a good way to animate a game. But its interface is quite primitive—using it requires us to track the time at which our function was called the last time around and call requestAnimationFrame again after every frame.

Let’s define a helper function that wraps those boring parts in a convenient interface and allows us to simply call runAnimation, giving it a function that expects a time difference as an argument and draws a single frame. When the frame function returns the value false, the animation stops.

function runAnimation(frameFunc) {

let lastTime = null;

function frame(time) {

if (lastTime != null) {

let timeStep = Math.min(time - lastTime, 100) / 1000;

if (frameFunc(timeStep) === false) return;

}

lastTime = time;

requestAnimationFrame(frame);

}

requestAnimationFrame(frame);

}

I have set a maximum frame step of 100 milliseconds (one-tenth of a second). When the browser tab or window with our page is hidden, requestAnimationFrame calls will be suspended until the tab or window is shown again. In this case, the difference between lastTime and time will be the entire time in which the page was hidden. Advancing the game by that much in a single step would look silly and might cause weird side effects, such as the player falling through the floor.

The function also converts the time steps to seconds, which are an easier quantity to think about than milliseconds.

The runLevel function takes a Level object and a display constructor and returns a promise. It displays the level (in document.body) and lets the user play through it. When the level is finished (lost or won), runLevel waits one more second (to let the user see what happens) and then clears the display, stops the animation, and resolves the promise to the game’s end status.

function runLevel(level, Display) {

let display = new Display(document.body, level);

let state = State.start(level);

let ending = 1;

return new Promise(resolve => {

runAnimation(time => {

state = state.update(time, arrowKeys);

display.syncState(state);

if (state.status == "playing") {

return true;

} else if (ending > 0) {

ending -= time;

return true;

} else {

display.clear();

resolve(state.status);

return false;

}

});

});

}

A game is a sequence of levels. Whenever the player dies, the current level is restarted. When a level is completed, we move on to the next level. This can be expressed by the following function, which takes an array of level plans (strings) and a display constructor:

async function runGame(plans, Display) {

for (let level = 0; level < plans.length;) {

let status = await runLevel(new Level(plans[level]),

Display);

if (status == "won") level++;

}

console.log("You've won!");

}

Because we made runLevel return a promise, runGame can be written using an async function, as shown in Chapter 11. It returns another promise, which resolves when the player finishes the game.

There is a set of level plans available in the GAME\_LEVELS binding in this chapter’s sandbox. This page feeds them to runGame, starting an actual game.

<link rel="stylesheet" href="css/game.css">

<body>

<script>

runGame(GAME\_LEVELS, DOMDisplay);

</script>

</body>

See if you can beat those. I had quite a lot of fun building them.

Exercises

Game over

It’s traditional for platform games to have the player start with a limited number of lives and subtract one life each time they die. When the player is out of lives, the game restarts from the beginning.

Adjust runGame to implement lives. Have the player start with three. Output the current number of lives (using console.log) every time a level starts.

<link rel="stylesheet" href="css/game.css">

<body>

<script>

// The old runGame function. Modify it...

async function runGame(plans, Display) {

for (let level = 0; level < plans.length;) {

let status = await runLevel(new Level(plans[level]),

Display);

if (status == "won") level++;

}

console.log("You've won!");

}

runGame(GAME\_LEVELS, DOMDisplay);

</script>

</body>

Pausing the game

Make it possible to pause (suspend) and unpause the game by pressing the Esc key.

This can be done by changing the runLevel function to use another keyboard event handler and interrupting or resuming the animation whenever the Esc key is hit.

The runAnimation interface may not look like it is suitable for this at first glance, but it is if you rearrange the way runLevel calls it.

When you have that working, there is something else you could try. The way we have been registering keyboard event handlers is somewhat problematic. The arrowKeys object is currently a global binding, and its event handlers are kept around even when no game is running. You could say they leak out of our system. Extend trackKeys to provide a way to unregister its handlers and then change runLevel to register its handlers when it starts and unregister them again when it is finished.

<link rel="stylesheet" href="css/game.css">

<body>

<script>

// The old runLevel function. Modify this...

function runLevel(level, Display) {

let display = new Display(document.body, level);

let state = State.start(level);

let ending = 1;

return new Promise(resolve => {

runAnimation(time => {

state = state.update(time, arrowKeys);

display.syncState(state);

if (state.status == "playing") {

return true;

} else if (ending > 0) {

ending -= time;

return true;

} else {

display.clear();

resolve(state.status);

return false;

}

});

});

}

runGame(GAME\_LEVELS, DOMDisplay);

</script>

</body>

A monster

It is traditional for platform games to have enemies that you can jump on top of to defeat. This exercise asks you to add such an actor type to the game.

We’ll call it a monster. Monsters move only horizontally. You can make them move in the direction of the player, bounce back and forth like horizontal lava, or have any movement pattern you want. The class doesn’t have to handle falling, but it should make sure the monster doesn’t walk through walls.

When a monster touches the player, the effect depends on whether the player is jumping on top of them or not. You can approximate this by checking whether the player’s bottom is near the monster’s top. If this is the case, the monster disappears. If not, the game is lost.

<link rel="stylesheet" href="css/game.css">

<style>.monster { background: purple }</style>

<body>

<script>

// Complete the constructor, update, and collide methods

class Monster {

constructor(pos, /\* ... \*/) {}

get type() { return "monster"; }

static create(pos) {

return new Monster(pos.plus(new Vec(0, -1)));

}

update(time, state) {}

collide(state) {}

}

Monster.prototype.size = new Vec(1.2, 2);

levelChars["M"] = Monster;

runLevel(new Level(`

..................................

.################################.

.#..............................#.

.#..............................#.

.#..............................#.

.#...........................o..#.

.#..@...........................#.

.##########..............########.

..........#..o..o..o..o..#........

..........#...........M..#........

..........################........

..................................

`),

DOMDisplay);

</script>

</body>

**References**

* <https://www.guru99.com/introduction-to-database-sql.html>
* <https://www.tutorialspoint.com/sql/sql-distinct-keyword.htm>
* <https://www.w3schools.com/sql/default.asp>
* <https://www.w3schools.com/html/default.asp>
* <https://www.tutorialspoint.com/html/index.htm>
* <https://www.w3schools.com/css/default.asp>
* <https://docs.angularjs.org/tutorial>
* <https://www.w3schools.com/angular/>
* [https://www.tutorialspoint.com/javascript/iHYPERLINK "https://www.tutorialspoint.com/javascript/index.htm"ndex.htm](https://www.tutorialspoint.com/javascript/iHYPERLINK%20%22https://www.tutorialspoint.com/javascript/index.htm%22ndex.htm)