**Develop a Web Application Using Front-end Stack**

**Day 1: HTML, CSS and JavaScript**

1. HyperText Markup Language (HTML)
   1. Tags
      * The document type <!DOCTYPE html>
      * Opening and closing html tags <html></html>
      * Head tag <head></head>
      * META tags

<meta name=“keywords” content=“code,html,programming,rob” />

<meta name=“description” content =“rob faves web” />

<meta http-equiv=“refresh” content=“5” />

<meta http-equiv=“refresh” content =“5; url=http://www.google.com” />

<meta name=“author” content=“rob imperial” />

* + - Link tags <link> (ex: getting css external links)
    - Script tags <script> (ex:getting external javascript files)
    - Title tag <title></title>
    - Body tag <body>
    - Adding comments and remarks <!-- -- >
    - Paragraph tag <p>
    - Break tags for adding new line <br />
    - Horizontal rule tag <hr />
    - Table tag <table><caption><tr><td or th>

Common attributes:

Cellpadding (space between contetnt and border)

Cellspacing (border thickness between cells)

Border

Align

Width

Bgcolor

Background

Bordercolor

<td rowspan=“2”…>

<td colspan=“3”…>

* + - Pre-formatted text tag <pre>
    - Text formatting tags

<font><u><b><i><strike><h1><h2><h3><h4><h5><sup><sub><q><code>

* + - List tags

<ul type=“square/disk/circle”><li></li>

<ol type=“1/i/I/a/A”><li></li>

* + - Image tag <img src=? alt=? width=? height=? align=? />
    - Hyperlink tag <a href=? Target=“\_blank/\_self”></a>
    - Video tag

<video width="320" height="240" controls>

<source src="movie.mp4" type="video/mp4">

Your browser does not support the video tag.

</video>

* + - Core attributes on majority of the tags:

Id (unique identifier for the html element)

Title (tooltip)

Class (associate element with a css class)

Style (select a css style for the element)

* + - Other attributes

Align

Bgcolor

Background

Width

Height

Name

Href

* + - Container tags <div><span>
    - Form and input tags

<form method=“post/get” action=““ enctype=“multipart/form-data” onSubmit=“?”>

<input type=“text” name=“?” pattern=“[0-9a-zA-Z]{3,8}” size=“?” maxlength=“?” required>

<input type=“password” name=“?”>

<input type=“email” name=“?”>

<input type=“radio” name=“?” value=“?” checked>

<input type=“checkbox” name=“?” value=“?” checked>

<input type=“button” name=“?” value=“?” onclick=“?”>

<input type=“submit” name=“?” value=“?”>

<input type=“reset” name=“?” value=“?”>

<textarea name=“?” rows=“?” cols=“?”></textarea>

<select name=“?” \*size=“4”>

<optgroup label=“?”>

<option value=“?” selected></option>

<option value=“?”></option>

<option value=“?”></option>

</optgroup>

</select>

<input type=“file” name=“?” accept=“images/\*”>

<input type=“hidden” name=“?” value=“?”>

1. Cascading Style-Sheets (CSS)

CSS is now the de-facto for implementing layout and design in elements and controls for web projects and applications.

* Syntax

selector{

property:value;

}

* Selectors
  + [type selector] h1{color:red;}
  + [universal selector] \*{color:blue;}
  + [descendant selector] ul li a{color:green;]
  + [class selector] .set1{color:yellow;}

Ex. <p class=“set1”>

* + [id selector] #set2{color:skyblue;}
  + [child selector] body>p{color:green;}
  + [attribute selector] input[“type=text”]{color:aqua;}
  + [multiple style rules] h1

{

color:red;

font-weight:normal;

text-transfer:lowercase;

}

* + [grouping selectors] h1, h2, h3

{

color:red;

font-weight:normal;

text-transfer:lowercase;

}

* Using css styles
  + - Embedded (declaring a style rule on the <head> tag)

Ex.

**<head>**

**<style type=“text/css”>**

**…**

**</style>**

**</head>**

* + - Inline (using the style attribute inside the html tag)

Ex.

**<div style=“background-color:yellow;border:2px solid black”>**

* + - External (create a separate .css file then include it on the page using the <link> in the <head>)

Ex.

**<head>**

**<link rel=“stylesheet” type=“text/css” href=“rules\_set1.css” />**

**</head>**

* + - Imported (importing an external .css file into the page)

Ex.

**<head>**

**<@import “mystyle.css”>**

**</head>**

* + - Overriding and prioritization
      1. Inline
      2. Embedded
      3. External or imported
* Comments (/\*…\*/)
* Measurement

%

cm

em (height of font, ex. Font = 12pts so, 2em=24pts)

ex (measurement relative to font’s lowercase x-height)

in

mm

pc (picas,1pc=12pts)

pt(points, smaller than pica)

px (pixels, based on screen resolution)

* Colors (see color sheet reference)

hex #rrggbb ex. p{color:#ff0000;}

short hex #rgb ex. p{color:#647;}

rgb% rgb(rrr%,ggg%,bbb%) ex. p{color:rgb(50%,50%,50%);}

rgb absolute rgb(rrr,ggg,bbb) ex. p{color:rgb(0,0,255);}

keyword aqua,black,etc… ex. p{color:yellow;}

* Background
  + - background-color:black
    - background-image:url(/images/pic1.jpg)
    - background-repeat:norepeat or repeat
    - background-position:100px 🡪100px from left
    - background-position:100px 200px 🡪left and right position
    - background-attachment:fixed or scroll
    - shorthand style

ex.

<p style=“background:url(/images/pic3.jpg) repeat fixed”>

* Font
* font-family:georgia
* font-style:normal|italic|oblique
* font-variant:normal|smallcaps
* font-weight:bold|bolder|number?
* font-size:xx-small|x-small|small|medium|large|x-large|xx-large|smaller|larger|pixels?|in?|%?
* font-stretch:normal|wider|narrower|ultra-condensed|extra-condensed|condensed|semi-condensed|semi-expanded|expanded|extra-expanded|ultra-expanded
* Text
  + color:red
  + direction:ltr|rtl
  + letter-spacing:normal|number?
  + word-spacing:normal|number?
  + text-indent:%|number?
  + text-align:right|center|left
  + text-decoration:underline|line-through|overline|blink
  + text-transform:capitalize|lowercase|uppercase
  + white-space:normal|pre|nowrap
  + text-shadow:4px 4px 8px blue
* Images
  + border:1px dashed red
  + height:100px
  + width:100px
* Links

[types}

a:link

a:visited

a:hover

a:active

ex:

a:link{color:yellow;}

* Tables
* border-collapse:collapse|separate

ex.

<style type="text/css">

table.one {border-collapse:collapse;}

table.two {border-collapse:separate;}

td.a {

border-style:dotted;

border-width:3px;

border-color:#000000;

padding: 10px;

}

td.b {border-style:solid;

border-width:3px;

border-color:#333333;

padding:10px;

}

</style>

<table class="one">

<caption>Collapse Border Example</caption>

<tr><td class="a"> Cell A Collapse Example</td></tr>

<tr><td class="b"> Cell B Collapse Example</td></tr>

</table>

<br />

<table class="two">

<caption>Separate Border Example</caption>

<tr><td class="a"> Cell A Separate Example</td></tr>

<tr><td class="b"> Cell B Separate Example</td></tr>

</table>

* + empty-cells:show|hide
* Border
  + border-color:green

border-bottom-color:?

border-top-color:?

border-left-color:?

border-right-color:?

* + border-style:none|solid|dotted|dashed|double|groove|ridge|inset|outset|hidden

border-bottom- style:?

border-top- style:?

border-left- style:?

border-right- style:?

* + border-width:number?

border-bottom-width:?

border-top- width:?

border-left- width:?

border-right- width:?

* Margins

Ex.

p{margin:15px;} 🡪all sides

p{margin:15px 2%;} 🡪top&bottom left&right

p{margin:15px 2% -10px auto;} 🡪top left bottom right

* Lists
  + list-style-path
    - 1. <ul> 🡪disc,circle,square
      2. <ol>
         1. decimal
         2. decimal-leading-zero
         3. lower-alpha
         4. upper-alpha
         5. lower-roman
         6. upper-roman
  + list-style-image:url(/mybullets/b1.gif)
  + marker-offset:2em
* Paddings (space between content and border, values:length,percent,inherit)
  + Ex.

<p style=“padding-bottom:5%;border:1px solid black”>

* + Ex. (all 4 paddings)

<p style=“padding:5%;border:1px solid black”>

* + Ex. (4 padding definition)

<p style=“padding:5% 2% 5% 2%;border:1px solid black”>

1. Bootstrap Quick Guide
2. Using bootstrap

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>My bootstrap test</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

<!-- jQuery library -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<!-- Latest compiled JavaScript -->

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

</head>

<body>

</body>

</html>

1. Popular bootstrap components
   1. Navigation

(bs 3.3.7)

<div class="navbar navbar-inverse">

<div class="container-fluid">

<div class="navbar-header">

<a class="navbar-brand" href="#">WebSiteName</a>

</div>

<ul class="nav navbar-nav">

<li class="active"><a href="#">Home</a></li>

<li class="dropdown">

<a class="dropdown-toggle" data-toggle="dropdown" href="#">Page 1

<span class="caret"></span></a>

<ul class="dropdown-menu">

<li><a href="#">Page 1-1</a></li>

<li><a href="#">Page 1-2</a></li>

<li><a href="#">Page 1-3</a></li>

</ul>

</li>

<li><a href="#">Page 2</a></li>

<li><a href="#">Page 3</a></li>

</ul>

</div>

</div>

* 1. Carousel

<!--slideshow-->

<div id="myCarousel" class="carousel slide" data-ride="carousel">

<!-- Indicators -->

<ol class="carousel-indicators">

<li data-target="#myCarousel" data-slide-to="0" class="active"></li>

<li data-target="#myCarousel" data-slide-to="1"></li>

<li data-target="#myCarousel" data-slide-to="2"></li>

</ol>

<!-- Wrapper for slides -->

<div class="carousel-inner">

<div class="item active">

<img src="la.jpg" alt="Los Angeles">

</div>

<div class="item">

<img src="chicago.jpg" alt="Chicago">

</div>

<div class="item">

<img src="ny.jpg" alt="New York">

</div>

</div>

<!-- Left and right controls -->

<a class="left carousel-control" href="#myCarousel" data-slide="prev">

<span class="glyphicon glyphicon-chevron-left"></span>

<span class="sr-only">Previous</span>

</a>

<a class="right carousel-control" href="#myCarousel" data-slide="next">

<span class="glyphicon glyphicon-chevron-right"></span>

<span class="sr-only">Next</span>

</a>

</div>

* 1. Grids

Bootstrap have 12 columns horizontally. Create a div for the row then distribute the number of columns to each div inside the row.

[3 equal columns]

<div class="row">

<div class="col-sm-4">.col-sm-4</div>

<div class="col-sm-4">.col-sm-4</div>

<div class="col-sm-4">.col-sm-4</div>

</div>

[unequal columns]

<div class="row">

<div class="col-sm-4">.col-sm-4</div>

<div class="col-sm-8">.col-sm-8</div>

</div>

* 1. Table

[table hover, striped and responsive]

<table class="table table-hover table-responsive table-striped">

<thead>

<tr>

<th>Firstname</th>

<th>Lastname</th>

<th>Email</th>

</tr>

</thead>

<tbody>

<tr>

<td>John</td>

<td>Doe</td>

<td>john@example.com</td>

</tr>

<tr>

<td>Mary</td>

<td>Moe</td>

<td>mary@example.com</td>

</tr>

<tr>

<td>July</td>

<td>Dooley</td>

<td>july@example.com</td>

</tr>

</tbody>

</table>

* 1. Images

[rounded corners]

<img src="cinqueterre.jpg" class="img-rounded" alt="Cinque Terre">

[circular]

<img src="cinqueterre.jpg" class="img-circle" alt="Cinque Terre">

[thumbnail]

<img src="cinqueterre.jpg" class="img-thumbnail" alt="Cinque Terre">

[linked thumbnails with caption and grid]

<div class="row">

<div class="col-md-4">

<div class="thumbnail">

<a href="/w3images/lights.jpg">

<img src="/w3images/lights.jpg" alt="Lights" style="width:100%">

<div class="caption">

<p>Lorem ipsum...</p>

</div>

</a>

</div>

</div>

<div class="col-md-4">

<div class="thumbnail">

<a href="/w3images/nature.jpg">

<img src="/w3images/nature.jpg" alt="Nature" style="width:100%">

<div class="caption">

<p>Lorem ipsum...</p>

</div>

</a>

</div>

</div>

<div class="col-md-4">

<div class="thumbnail">

<a href="/w3images/fjords.jpg">

<img src="/w3images/fjords.jpg" alt="Fjords" style="width:100%">

<div class="caption">

<p>Lorem ipsum...</p>

</div>

</a>

</div>

</div>

</div>

* 1. Alerts

<div class="alert alert-success">

<strong>Success!</strong> Indicates a successful or positive action.

</div>

<div class="alert alert-info">

<strong>Info!</strong> Indicates a neutral informative change or action.

</div>

<div class="alert alert-warning">

<strong>Warning!</strong> Indicates a warning that might need attention.

</div>

<div class="alert alert-danger">

<strong>Danger!</strong> Indicates a dangerous or potentially negative action.

</div>

* 1. Buttons

<button type="button" class="btn">Basic</button>

<button type="button" class="btn btn-default">Default</button>

<button type="button" class="btn btn-primary">Primary</button>

<button type="button" class="btn btn-success">Success</button>

<button type="button" class="btn btn-info">Info</button>

<button type="button" class="btn btn-warning">Warning</button>

<button type="button" class="btn btn-danger">Danger</button>

<button type="button" class="btn btn-link">Link</button>

* 1. Button groups

<div class="btn-group">

<button type="button" class="btn btn-primary">Apple</button>

<button type="button" class="btn btn-primary">Samsung</button>

<button type="button" class="btn btn-primary">Sony</button>

</div>

* 1. Pagination

<ul class="pagination">

<li><a href="#">1</a></li>

<li class="active"><a href="#">2</a></li>

<li><a href="#">3</a></li>

<li><a href="#">4</a></li>

<li><a href="#">5</a></li>

</ul>

* 1. Dropdown

<div class="dropdown">

<button class="btn btn-primary dropdown-toggle" type="button" data-toggle="dropdown">Dropdown Example

<span class="caret"></span>

</button>

<ul class="dropdown-menu">

<li><a href="#">HTML</a></li>

<li><a href="#">CSS</a></li>

<li><a href="#">JavaScript</a></li>

<li class="divider"></li>

<li><a href="#">others</a></li>

</ul>

</div>

* 1. Collapse

<button data-toggle="collapse" data-target="#demo">Collapsible</button>

<div id="demo" class="collapse">

Lorem ipsum dolor text....

</div>

* 1. Modal

<!-- Trigger the modal with a button -->

<button type="button" class="btn btn-info btn-lg" data-toggle="modal" data-target="#myModal">Open Modal</button>

<!-- Modal -->

<div id="myModal" class="modal fade" role="dialog">

<div class="modal-dialog">

<!-- Modal content-->

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal">&times;</button>

<h4 class="modal-title">Modal Header</h4>

</div>

<div class="modal-body">

<p>Some text in the modal.</p>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal">Close</button>

</div>

</div>

</div>

</div>

1. Client-Side Script (Javascript)

JavaScript is a client-side scripting language that is popularly and widely used as part of most web applications. It is now also being used as the script for creating and managing a variety of other well-known web application and scripting libraries and frameworks such as JQuery, Node.js, AngularJS and many others.

JavaScript offers cross-browser compatibility, can easily be integrated to new and existing web projects, and provide behavioral and event-programming functions along with object-oriented programming capabilities.

JavaScript can be called directly from a web page since the browser will be the one interpreting the code.

JavaScript functions are case-sensitive. Use a semi-colon to terminate each statement.

**Example of a javascript in an html code:**

<html>

<head><title>page1</title></head>

<body>

<script type="text/javascript">

document.write("iam created using javascript");

</script>

</body>

</html>

**Example of a javascript in an html element event(“onclick”):**

<input type="button" name="btn1" id="btn01" value="click me" onclick="alert('hi');">

**Example of a javascript in a function on the same page:**

<html>

<head>

<title>page1</title>

<script type="text/javascript">

function showtext()

{

alert("iam created using javascript");

}

</script>

</head>

<body>

<input type="button" name="btn1" id="btn01" value="click me" onclick="showtext();">

</body>

</html>

**Example of referencing an external javascript page to use its functions:**

<head>

<script type=“text/javascript” src=“jscript1.js” />

</head>

* Displaying Alerts

//hi, iam a comment. I won’t appear on the output

//the ff code is an alert box

alert(“hi there!”);

//the following is an example of a confirm box

var x = confirm(“press a button:”);

if(x==true){ alert(“you pressed ok”); }

else{ alert(“you pressed cancel”); }

* Declaring Variables

Variables hold data for processing. Can also hold control and event references.

Primitive Data Types:

1. Number
2. String
3. Boolean
4. Null
5. Object
6. Symbol
7. Undefined

Sample variable declaration:

\*You don’t have to specifically choose a data type upon variable creation. The data type will be automatically determined while the program is being processed.

var firstname = "marian";

alert(firstname);

var numberx = 45;

alert(numberx);

* Operators and Math Expressions

Common arithmetic operators:

Parenthesis ()

Addition +

Subtraction -

Multiplication \*

Division /

Remainder %

Comparison operators:

Equality ==

Non-equality !=

Relational operators:

Greater than >

Greater than or equal >=

Less than <

Less than or equal <=

Logical operators:

And &&

Or ||

Not !

//sample math expression:

//note: follows PMDAS operator precedence

var ans=((var1+var2+var3)-var4)\*var5;

//set decimal places

var num = 12.455678;

num=num.toFixed(2);//decimal places

//number type conversion

parseInt(float/string variable)

parseFloat(int/string variable)

sample:

<html>

<head>

<title>page1</title>

<script type="text/javascript">

function addme()

{

var x=parseFloat(document.getElementById("text01").value);

var y=parseFloat(document.getElementById("text02").value);

var z=x+y;

alert(z);

}

</script>

</head>

<body>

<input type="text" name="text1" id="text01"><br />

<input type="text" name="text2" id="text02"><br />

<input type="button" name="btn2" id="btn02" value="add" onclick="addme();">

</body>

</html>

* Prompts

Sample:

var name = prompt(“enter name: “,defaultvalue);

alert(“you entered “ + name);

* Conditional Statements (IF and Switch)

Sample:

//if condition sample code

if(condition) { action; }

else if (condition) { action; }

else if (condition) { action; }

else {action; }

//switch statement sample code

switch(var to test)

{

case val1: action;break;

case val2: action;break;

case val3: action;break;

default:action;break;

}

* Arrays

//creating array

var mysubjects = [“database”,”web”,”network”,”mobile”];

//to use array elements, use their array index number. The first item is 0.

alert(“the second subject is “ + mysubjects[1]);

//creating an empty array

var myfriends = [];

//then adding elements to the array

myfriends[0]=“angel”;

myfriends[1]=“castro”;

myfriends[2]=“delilah”;

//editing an array element value

myfriends[1]=“cassie”;

//adding more elements at the end of the array

myfriends.push(“bernard”,”anna”);

//adding elements at the beginning of the array

myfrineds.unshift(“kevin”,”sven”);

//removing the last element of the array

myfriends.pop();

//removing the first element of the array

myfriends.shift();

//splicing elements in the array

myfriends.splice(3,2,”elsa”,”hans”); //1st param=where to start, 2nd param=how many to remove after the start position, succeeding params=array element to insert.

myfriends.splice(3,0,”troll”,”king”,”queen”); //spliced the elements into the array from the 3rd position without removing any other elements.

myfriends.splice(5,2); //removing two elements from the third element on the array without adding new elements.

* FOR loops

for(var counter=0;counter!=50;counter++)

{

action;

}

//looping though an array

var mypets=[“lion”,”tiger”,”panther”];

for(var x=0;x<=mypets.length;x++)

{

alert(mypets[x]);

}

//checking for a match in the array

var tosearch = document.getElementById(“text03”).value;

var mypets=[“lion”,”tiger”,”panther”];

var matchfound=“no”; //initializing a variable with a default value. This is called a flag.

for(var x=0;x<mypets.length;x++)

{

If(tosearch==mypets[x])

{

matchfound=“yes”;

alert(mypets[x]);

break;

}

}

* while loops

var counter=0;

while(counter<=50)

{

action;

counter++;

}

* do...while loops

var counter=0;

do

{

action;

counter++;

}while(counter!=50)

* Using Pre-built String Functions

//create a string then use the following properties and functions to manipulate the string.

var s = document.getElementById(“text04”).value;

s.length

s.charAt(int) 🡪gets character at specified position

s.indexOf(value) 🡪searches for an instance of a character or string,

returns -1 if not found

s.substr(start,#ofchars)

s.substring(start,end)

s.toLowerCase()

s.toUpperCase()

s.replace(“old”,”new”)

s.endswith(string) 🡪returns Boolean true if the string ends with the specified string

s.trim() 🡪removes leading and trailing whitespaces

* Rounding numbers

//round to the nearest whole number

var rn=Math.round(90.877654);

alert(rn); //returns “90”

//round to 2 decimal places

var rn=Math.round(90.877654\*100)/100;

alert(rn);

* Generating random numbers

//generates a random number from 1-10

Math.floor((Math.random() \* 10) + 1);

* Converting strings to integers and float, vice-versa

//check the data type of a variable

typeof variable;

//string to integer

var i=parseInt(“40”);

var f=parseFloat(“40”); //returns number with decimal places

var s=numbervar.toString();

* Getting the current date and time

//getting current system time and date

var mydate = new Date();

mydate.getDate(); //1-31

mydate.getMonth(); //0-11

mydate.getDay(); //0-6

mydate.getFullYear();

mydate.getHours(); //24-hr format

mydate.getMinutes();

mydate.getSeconds();

//date difference.

//Get the two dates

var startdate = new Date(); //gets current date

var enddate = new Date(“October 13, 2016”); //sets another date manually.

//you can also add time to the manually entered date.

//Ex: (var d=new Date(“January 13, 2020 14:30:00”))

var millisecdiff = enddate – startdate;

var hoursdiff = millisecdiff / 1000; //gets difference in hours

var daysdiff = millisecdiff / (1000\*60\*60\*24); //gets difference in days

* Functions:

function functionname(parameter1, parameter2)

{

var peso = (parameter 1\* 45)+parameter2;

return peso;

}

//to use

var answer = functionname(78,20);

document.write(answer);

//automate a function. Function name and parenthesis should be in quotes

setTimeout('functionname()',10000);

setInterval('functionname()',3000);

//call a function from control event

onClick=functionname()

onMouseOver=functionname()

onSubmit=functionname()

* Placing scripts

**Example of a javascript in an html code:**

<html>

<head><title>page1</title></head>

<body>

<script type="text/javascript">

document.write("iam created using javascript");

</script>

</body>

</html>

**Example of a javascript in an html element event(“onclick”):**

<input type="button" name="btn1" id="btn01" value="click me" onclick="alert('hi');">

**Example of a javascript in a function on the same page:**

<html>

<head>

<title>page1</title>

<script type="text/javascript">

function showtext()

{

alert("iam created using javascript");

}

</script>

</head>

<body>

<input type="button" name="btn1" id="btn01" value="click me" onclick="showtext();">

</body>

</html>

**Example of referencing an external javascript page to use its functions:**

<head>

<script type=“text/javascript” src=“jscript1.js” />

</head>

* Events

popular control events

* 1. Form control or html element events
     1. onClick
     2. onSubmit
     3. onMouseOver
     4. onMouseOut
     5. onMouseDown
     6. onMouseUp
     7. onBlur
     8. onFocus
     9. onChange
     10. onSelect
     11. onReset
     12. onSubmit
  2. body events
     1. onLoad
     2. onPageHide
     3. onPageShow
     4. onResize
     5. onUnload
* Events: link

//showing the use of javascript:void(0) to negate the action of the link then using an event “onClick” to trigger a javascript function

<a href="JavaScript:void(0)" onClick="alert('Hi');">Click</a>

* Events: button

<input type="button" name="btn1" id="btn01" value="click me"

onMouseOver="value='yes me'"

onMouseOut="value='click me'"

onClick="alert(‘singe click’);"

onDblClick=“alert(‘2x click’);”>

* Events: mouse

<img src=“sample1.jpg” onMouseOver=“src=’sample2.jpg’” onMouseOut=“src=’sample1.jpg’” />

Or by using javascript to manipulate html element properties

<input type=“button” onMouseOver=“document.getElementById(“img01”).src=’picture2.jpg’;”>

* Events: fields

Can be used for form input field validation and behavioral control.

<input type=“text”

onFocus=“this.style.backgroundcolor=’yellow’”

onBlur=“this.style.backgroundcolor=’white’”>

* Reading field values

//getting values from html input fields (input field not in a form)

function getinput(x)

{

var input1 = document.getElementById(x).value;

//this tests if field has data. Use if(!input){} to test it the other way around.

if(input1)

{

alert(input1);

}else{

document.getElementById(x).style.backgroundcolor=“green”;

alert(“please fill up the green colored fields”);

}

}

<input type=“text” name=“text1” id=“text01”>

<input type=“button” name=“button1” id=“button01” onClick=“getinput(‘text01’);”>

//getting values from html input fields (using a form for submission)

function getinput(x)

{

var input1 = document.getElementById(x).value;

//this tests if field has data. Use if(!input){} to test it the other way around.

if(input1)

{

alert(input1);

}else{

document.getElementById(x).style.backgroundcolor=“green”;

alert(“please fill up the green colored fields”);

}

}

<form onSubmit=“getinput(‘text01’);”>

<input type=“text” name=“text1” id=“text01”>

<input type=“button” name=“button1” id=“button01”>

</form>

//creating a reusable function that dynamically checks and gets the ID of whoever calls the function.

function get\_id(mycontrol)

{

var id = mycontrol.id;

//…codes that utilizes the id…

}

//on control event

…onClick=“get\_id(this)”;

//getting values from other html tag that act as container such as <p> or <div>

document.getElementById(“p1”).innerHTML

* Setting field values

Sample:

function fillfields(x)

{

var i=document.getElementById(x).value;

var newval="";

switch(i)

{

case "101":newval="lobby";break;

case "102":newval="reception";break;

case "103":newval="classroom";break;

default:newval="";

}

document.getElementById("text02").value=newval;

}

<input type="text" name="text1" id="text01" onBlur="fillfields('text01');"><br />

<input type="text" name="text2" id="text02"><br />

<input type="button" name="btn2" id="btn02" value="add" onClick="fillfields('text01');">

* Reading and setting paragraph text

\*note: using *document.getElementById(?).innerHTML=“value/text”* allows you to insert almost any content into other html containers such as a *<div>* or others.

//create a function to call your full paragraph

function fullparag1()

{

var fp1="What is JavaScript? JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well. It is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles.<br>JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behaviour.<br>Contrary to popular misconception, JavaScript is not \"Interpreted Java\". In a nutshell, JavaScript is a dynamic scripting language supporting prototype based object construction. The basic syntax is intentionally similar to both Java and C++ to reduce the number of new concepts required to learn the language. Language constructs, such as if statements, for and while loops, and switch and try... catch blocks function the same as in these languages (or nearly so.) <br>JavaScript can function as both a procedural and an object oriented language. Objects are created programmatically in JavaScript, by attaching methods and properties to otherwise empty objects at run time, as opposed to the syntactic class definitions common in compiled languages like C++ and Java. Once an object has been constructed it can be used as a blueprint (or prototype) for creating similar objects.<br>JavaScript's dynamic capabilities include runtime object construction, variable parameter lists, function variables, dynamic script creation (via eval), object introspection (via for ... in), and source code recovery (JavaScript programs can decompile function bodies back into their source text).";

document.getElementById("parag1").innerHTML=fp1;

}

//create a shortened version of your paragraph and a link to trigger the paragraph expansion.

<p id=“parag1”>

What is JavaScript?JavaScript® (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in <a href=“javascript:void();” onCLick=“expandparag1();”>read more…</a>

</p>

* Manipulating images and text

\*note: the ff activity will show how to use a CSS class-rule through event-trigger function in javascript.

//create a css class-rule(sample: make an HTML element invisible)

<style type="text/css">

.hidden {display:none;}

.showme{display:block;}

</style>

//create a javascript function to invoke the css-rule

function makedisappear(x)

{

document.getElementById(x).className="hidden";

}

function makeappear(x)

{

document.getElementById(x).className="showme";

}

//add events to the control to call the javascript functions

<img src="boy1.gif" id="pic01" height="60" width="70"

onmouseout="makeappear('pic01');"

onmouseover="makedisappear('pic01');">

//if the html element already has a class for css styling and you don’t want to override that class, you //can add additional class definition by editing the javascript code to “add” another class.

Sample:

function makedisappear(x)

{

document.getElementById(x).className+=" hidden"; //a space before the classname is needed.

}

//swapping pictures

function swappic(id,newpic)

{

document.getElementById(id).src=newpic;

}

//add an event on the control to trigger swapping the image

<img src=“pic1.jpg” id=“pic1” onMouseOver=“swappic(‘pic1’,’pic2.jpg’);”>

* Setting styles

//if used within a function

document.getElementById(?).style.backgroundColor=“yellow”;

document.getElementById(?).style.visibility=“hidden”;

//if used inline within the html tag.Sample:

<input type=“text” id=“text1” onFocus=“this.style.backgroundColor=’yellow’” onBlur=“this.style.backgroundColor=’white’”>

* Target all elements by tag name

//select the html tag that you wish to target. Example: <p>

<p>paragraph1. this is an example of paragraph1</p>

<p>paragraph2. this is an example of paragraph2</p>

<p>paragraph3. this is an example of paragraph3</p>

//prepare javascript function to loop through all instances of that html tag to set desired attribute

function setparagraphcolor()

{

//get an array of all the elements with the matching tag.example:<p>

//the first instance of the <p> will be assigned to p[0], then p[1]...

var pa=document.getElementsByTagName("p");

//loop to execute the action to all instances.

for(var i=0;i<pa.length;i++)

{

pa[i].style.color="yellow";

}

}

//prepare the html element to trigger the function

<input type="button" value="color it yellow" onclick="setparagraphcolor();">

* Target some elements by tag name

//as you loop through all the instances of the tag using document.getElementsByTagName(“?”), you can set condition(s) before executing the action.

function setparagraphcolor()

{

//get an array of all the elements with the matching tag.example:<p>

//the first instance of the <p> will be assigned to p[0], then p[1]...

var pa=document.getElementsByTagName("p");

//loop to execute the action to all instances.

for(var i=0;i<pa.length;i++)

{

//get attribute then set a condition

var ptext=pa[i].innerHTML;

if(ptext.endsWith("2"))

{

pa[i].style.color="yellow";

}

}

}

* Browser control: Filling the window with content

//open up a new window

function createwindow()

{

var newwindow=window.open();

}

//open up a new window then add content

function createwindow()

{

var newwindow=window.open();

var content1="<img src='boy1.gif' id='pic01' height='250' width='200' onmouseout='makeappear('pic01');' onmouseover='makedisappear('pic01');'><img src='dog.gif' id='pic01' height='250' width='200' ><img src='hula.gif' id='pic01' height='250' width='250' >";

newwindow.document.write(content1);

}

//create and open a window then function to close it

var newwindow;

function createwindow()

{

newwindow=window.open("http://www.facebook.com/");

}

function closedummy()

{

newwindow.close();

}

<input type="button" id="btn50" value="new window" onClick="createwindow();">

<input type="button" id="btn51" value="close new window" onClick="closedummy();">

* Browser control: Controlling the window's size and location

var newwindow=window.open(“url”,”name”,”width=200,height=200”,”left=200,top=100”)

* Browser control: Testing for popup blockers

//for browsers except IE. For IE use the test (testPop===“undefined”)

function checkForPopBlocker() {

var testPop = window.open("", "","width=100,height=100");

if (testPop === null) {

alert("Please disable your popup blocker.");

}

testPop.close();

}

<body onLoad ="checkForPopBlocker();">

* Form validation: text fields

//a sample function to test if a textfield in blank upon page submission or on the textfield’s own onBlur event

function checkForLastName() {

var targetField = document.getElementById("lastNameField");

if (targetField.value.length === 0) {

alert("Please enter your last name");

targetField.focus();

targetField.style.background = "yellow";

return false;

}

targetField.style.background = "white";

}

* Form validation: drop-downs

//sample markup for a combobox

<form onSubmit="return checkForSelection(‘states’);">

<select id="states">

<option value="" selected="selected">

SELECT A STATE</option>

<option value="AL">Alabama</option>

<option value="AK">Alaska</option>

<option value="AZ">Arizona</option>

<option value="AR">Arkansas</option>

</select>&nbsp;&nbsp;

<input type="submit" value="Submit Form">

</form>

//sample function for validation

function checkForSelection(selecID) {

var target = document.getElementById(selecID);

if (target.selectedIndex === 0) {

alert("Please select a state.");

return false;

}

}

* Form validation: radio buttons

//actually easier if you simply set the property “checked” on one of the radio buttons.

//sample markup for radio buttons

<form onSubmit="return validateRadios();">

<input type="radio" name="r1" value="cat"> Cat<br>

<input type="radio" name="r1" value="bat"> Bat<br>

<input type="radio" name="r1" value="hat"> Hat<br>

<input type="submit" value="Submit Form">

</form>

//sample function to check if a user have selected a radio button

function validateRadios() {

var radios = document.getElementsByName("r1");

for (var i = 0; i < radios.length; i++) {

if (radios[i].checked) {

return true;

}

}

alert("Please check one.");

return false;

}

* Form validation: email

//easier by using html5 <input type=“email”>

//here is the javascript function that uses regex to check for the correct email format

function validateEmail() {

var eEntered = document.getElementById("address").value;

var regex = /^[\w\-\.\+]+\@[a-zA-Z0-9\. \-]+\.[a-zA-z0-9]{2,4}$/;

if (!(eEntered.match(emailCorrectPattern))) {

alert("Please correct email address");

return false;

}

}

* Exceptions: try and catch

//trying a risky code then displaying the error as generated by the browser

function greetWorld() {

try {

var greeting = "Hello world!";

aler(greeting);

}

catch(err) {

alert(err);

}

}

* Exceptions: throw

//assigning custom errors using throw instead of browser-generated errors

//this is a sample function that checks for password complexity

function checkPassword() {

try {

var pass = document.getElementById("f1").value;

if (pass.length < 8) {throw "Please enter at least 8 characters.";}

if (pass.indexOf(" ") !== -1) {throw "No spaces in the password, please.";}

var numberSomewhere = false;

for (var i = 0; i < pass.length; i++)

{

if (isNaN(pass(i, i+1)) === false) {numberSomewhere = true;break;}

}

if (numberSomewhere === false) {throw "Include at least 1 number.";}

}

catch(err) {

alert(err);

}

}

* Handling events within JavaScript

//you can trigger an HTML event programmatically in javascript

//the event (ex.”onclick”) should be lowercase

document.getElementById("button1").onclick = sayHello;

**Day 2: An Introduction to Typescript**

1. Basic Types

**Boolean**

let isDone: boolean = false;

**Number**

let decimal: number = 6;

let hex: number = 0xf00d;

let binary: number = 0b1010;

let octal: number = 0o744;

let big: bigint = 100n;

**String**

Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

let color: string = "blue";

color = 'red';

You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

let fullName: string = `Bob Bobbington`;

let age: number = 37;

let sentence: string = `Hello, my name is ${fullName}.

I'll be ${age + 1} years old next month.`;

This is equivalent to declaring sentence like so:

let sentence: string =

"Hello, my name is " +

fullName +

".\n\n" +

"I'll be " +

(age + 1) +

" years old next month.";

**Array**

Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

let list: number[] = [1, 2, 3];

The second way uses a generic array type, Array<elemType>:

let list: Array<number> = [1, 2, 3];

**Tuple**

Tuple types allow you to express an array with a fixed number of elements whose types are known but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

// Declare a tuple type

let x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

Type 'number' is not assignable to type 'string'.

Type 'string' is not assignable to type 'number'.

When accessing an element with a known index, the correct type is retrieved:

// OK

console.log(x[0].substring(1));

console.log(x[1].substring(1));

Property 'substring' does not exist on type 'number'.

Accessing an element outside the set of known indices fails with an error:

x[3] = "world";

Tuple type '[string, number]' of length '2' has no element at index '3'.

console.log(x[5].toString());

Object is possibly 'undefined'.

Tuple type '[string, number]' of length '2' has no element at index '5'.

**Enum**

As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

enum Color {

Red,

Green,

Blue,

}

let c: Color = Color.Green;

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

enum Color {

Red = 1,

Green,

Blue,

}

let c: Color = Color.Green;

Or, even manually set all the values in the enum:

enum Color {

Red = 1,

Green = 2,

Blue = 4,

}

let c: Color = Color.Green;

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren't sure what that mapped to in the Color enum above, we could look up the

corresponding name:

enum Color {

Red = 1,

Green,

Blue,

}

let colorName: string = Color[2];

// Displays 'Green'

console.log(colorName);

**Unknown**

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API. In

these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

let notSure: unknown = 4;

notSure = "maybe a string instead";

// OK, definitely a boolean

notSure = false;

If you have a variable with an unknown type, you can narrow it to something more specific by doing typeof checks or comparison checks:

declare const maybe: unknown;

// 'maybe' could be a string, object, boolean, undefined, or other types

const aNumber: number = maybe;

Type 'unknown' is not assignable to type 'number'.

if (maybe === true) {

// TypeScript knows that maybe is a boolean now

const aBoolean: boolean = maybe;

// So, it cannot be a string

const aString: string = maybe;

Type 'boolean' is not assignable to type 'string'.

}

if (typeof maybe === "string") {

// TypeScript knows that maybe is a string

const aString: string = maybe;

// So, it cannot be a boolean

const aBoolean: boolean = maybe;

Type 'string' is not assignable to type 'boolean'.

}

Any

In some situations, not all type information is available or its declaration would take an inappropriate amount of effort. These may occur for values from code that has been written without TypeScript or a 3rd party library. In these cases,

we might want to opt-out of type checking. To do so, we label these values with the any type:

declare function getValue(key: string): any;

// OK, return value of 'getValue' is not checked

const str: string = getValue("myString");

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type checking during compilation.

Unlike unknown, variables of type any allow you to access arbitrary properties, even ones that don't exist. These properties include functions and TypeScript will not check their existence or type:

let looselyTyped: any = 4;

// OK, ifItExists might exist at runtime

looselyTyped.ifItExists();

// OK, toFixed exists (but the compiler doesn't check)

looselyTyped.toFixed();

let strictlyTyped: unknown = 4;

strictlyTyped.toFixed();

Object is of type 'unknown'.

The any will continue to propagate through your objects:

let looselyTyped: any = {};

let d = looselyTyped.a.b.c.d;

// ^ = let d: any

After all, remember that all the convenience of any comes at the cost of losing type safety. Type safety is one of the main motivations for using TypeScript and you should try to avoid using any when not necessary.

**Void**

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

function warnUser(): void {

console.log("This is my warning message");

}

Declaring variables of type void is not useful because you can only assign null (only if --strictNullChecks is not specified, see next section) or undefined to them:

let unusable: void = undefined;

// OK if `--strictNullChecks` is not given

unusable = null;

**Null and Undefined**

In TypeScript, both undefined and null actually have their types named undefined and null respectively. Much like void, they're not extremely useful on their own:

// Not much else we can assign to these variables!

let u: undefined = undefined;

let n: null = null;

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number. However, when using the --strictNullChecks flag, null and undefined are only assignable to unknown, any and their respective types (the one exception being that undefined is also assignable to void). This helps avoid many common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type string | null | undefined.

**Never**

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns. Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

Some examples of functions returning never:

// Function returning never must not have a reachable end point

function error(message: string): never {

throw new Error(message);

}

// Inferred return type is never

function fail() {

return error("Something failed");

}

// Function returning never must not have a reachable end point

function infiniteLoop(): never {

while (true) {}

}

**Object**

object is a type that represents the non-primitive type, i.e. anything that is not number, string, boolean, bigint, symbol, null, or undefined. With object type, APIs like Object.create can be better represented. For example:

declare function create(o: object | null): void;

// OK

create({ prop: 0 });

create(null);

create(undefined); // Remember, undefined is not a subtype of null

// Argument of type 'undefined' is not assignable to parameter of type 'object | nullcreate(42);

// Argument of type '42' is not assignable to parameter of type 'object | null'.

create("string");

// Argument of type '"string"' is not assignable to parameter of type 'object | null'create(false);

// Argument of type 'false' is not assignable to parameter of type 'object | null'.

// Generally, you won't need to use this.

**Type assertions**

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually, this will happen when you know the type of some entity could be more specific than its current type. Type assertions are a way to tell the compiler "trust me, I know what I'm doing."

A type assertion is like a type cast in other languages, but it performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms.

One is the as-syntax:

let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;

The other version is the "angle-bracket" syntax:

let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;

The two samples are equivalent. Using one over the other is mostly a choice of reference; however, when using TypeScript with JSX, only as-style assertions are allowed.

**A note about let**

You may have noticed that so far, we've been using the let keyword instead of JavaScript's var keyword which you might be more familiar with. The let keyword is actually a newer JavaScript construct that TypeScript makes available.

**About Number, String, Boolean, Symbol and Object**

It can be tempting to think that the types Number, String, Boolean, Symbol, or Object are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be

used as a type.

function reverse(s: String): String {

return s.split("").reverse().join("");

}

reverse("hello world");

Instead, use the types number, string, boolean, object and symbol.

function reverse(s: string): string {

return s.split("").reverse().join("");

}

reverse("hello world");

**Interfaces**

One of TypeScript's core principles is that type checking focuses on the shape that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

**Our First Interface**

The easiest way to see how interfaces work is to start with a simple example:

function printLabel(labeledObj: { label: string }) {

console.log(labeledObj.label);

}

let myObj = { size: 10, label: "Size 10 Object" };

printLabel(myObj);

The type checker checks the call to printLabel. The printLabel function has a single parameter that requires that the object passed in has a property called label of type string. Notice that our object actually has more properties than

this, but the compiler only checks that at least the ones required are present and match the types required. There are some cases where TypeScript isn't as lenient, which we'll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the label property that is a string:

interface LabeledValue {

label: string;

}

function printLabel(labeledObj: LabeledValue) {

console.log(labeledObj.label);

}

let myObj = { size: 10, label: "Size 10 Object" };

printLabel(myObj);

The interface LabeledValue is a name we can now use to describe the requirement in the previous example. It still represents having a single property called label that is of type string. Notice we didn't have to explicitly say that

the object we pass to printLabel implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

**Optional Properties**

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like "option bags" where you pass an object to a function that

only has a couple of properties filled in.

Here's an example of this pattern:

interface SquareConfig {

color?: string;

width?: number;

}

function createSquare(config: SquareConfig): { color: string; area: number

let newSquare = { color: "white", area: 100 };

if (config.color) {

newSquare.color = config.color;

}

if (config.width) {

newSquare.area = config.width \* config.width;

}

return newSquare;

}

let mySquare = createSquare({ color: "black" });

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a ? at the end of the property name in the declaration. The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the color property in createSquare, we would get an error message letting us know:

interface SquareConfig {

color?: string;

width?: number;

}

function createSquare(config: SquareConfig): { color: string; area: number

let newSquare = { color: "white", area: 100 };

if (config.clor) {

// Property 'clor' does not exist on type 'SquareConfig'. Did you mean 'color'?

// Error: Property 'clor' does not exist on type 'SquareConfig'

newSquare.color = config.clor;

// Property 'clor' does not exist on type 'SquareConfig'. Did you mean 'color'?

}

if (config.width) {

newSquare.area = config.width \* config.width;

}

return newSquare;

}

let mySquare = createSquare({ color: "black" });

**Readonly properties**

Some properties should only be modifiable when an object is first created. You can specify this by putting readonly before the name of the property:

interface Point {

readonly x: number;

readonly y: number;

}

You can construct a Point by assigning an object literal. After the assignment, x and y can't be changed.

let p1: Point = { x: 10, y: 20 };

p1.x = 5; // error!

// Cannot assign to 'x' because it is a read-only property.

TypeScript comes with a ReadonlyArray<T> type that is the same as Array<T> with all mutating methods removed, so you can make sure you don't change your arrays after creation:

let a: number[] = [1, 2, 3, 4];

let ro: ReadonlyArray<number> = a;

ro[0] = 12; // error!

// Index signature in type 'readonly number[]' only permits reading.

ro.push(5); // error!

// Property 'push' does not exist on type 'readonly number[]'.

ro.length = 100; // error!

// Cannot assign to 'length' because it is a read-only property.

a = ro; // error!

The type 'readonly number[]' is 'readonly' and cannot be assigned to the mutable tyOn the last line of the snippet you can see that even assigning the entire ReadonlyArray back to a normal array is illegal. You can still override it with a

type assertion, though:

let a: number[] = [1, 2, 3, 4];

let ro: ReadonlyArray<number> = a;

a = ro as number[];

**readonly vs const**

The easiest way to remember whether to use readonly or const is to ask whether you're using it on a variable or a property. Variables use const whereas properties use readonly.

**Excess Property Checks**

In our first example using interfaces, TypeScript lets us pass { size: number; label: string; } to something that only expected a { label: string; }. We also just learned about optional properties, and how they're useful when

describing so-called "option bags".

However, combining the two naively would allow an error to sneak in. For example, taking our last example using createSquare:

interface SquareConfig {

color?: string;

width?: number;

}

function createSquare(config: SquareConfig): { color: string; area: number

return {

color: config.color || "red",

area: config.width ? config.width \* config.width : 20,

};

}

let mySquare = createSquare({ colour: "red", width: 100 });

Argument of type '{ colour: string; width: number; }' is not assignable to parameteObject literal may only specify known properties, but 'colour' does not exist inNotice the given argument to createSquare is spelled colour instead of color. In

plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the width properties are compatible, there's no color property present, and the extra colour property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code. Object literals get special treatment and undergo excess property checking when assigning them to other variables, or passing them as arguments. If an object

literal has any properties that the "target type" doesn't have, you'll get an error:

let mySquare = createSquare({ colour: "red", width: 100 });

Argument of type '{ colour: string; width: number; }' is not assignable to parameteObject literal may only specify known properties, but 'colour' does not exist inGetting around these checks is actually really simple. The easiest method is to

just use a type assertion:

let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way. If SquareConfig can have color and width properties with the above types,

but could also have any number of other properties, then we could define it like so:

interface SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

We'll discuss index signatures in a bit, but here we're saying a SquareConfig can have any number of properties, and as long as they aren't color or width, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since squareOptions won't undergo excess property checks, the compiler won't give you an error.

let squareOptions = { colour: "red", width: 100 };

let mySquare = createSquare(squareOptions);

The above workaround will work as long as you have a common property between squareOptions and SquareConfig. In this example, it was the property width. It will however, fail if the variable does not have any common object property. For example:

let squareOptions = { colour: "red" };

let mySquare = createSquare(squareOptions);

// Type '{ colour: string; }' has no properties in common with type 'SquareConfig'.

**Function Types**

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and

type.

interface SearchFunc {

(source: string, subString: string): boolean;

}

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

let mySearch: SearchFunc;

mySearch = function (source: string, subString: string): boolean {

let result = source.search(subString);

return result > -1;

};

For function types to correctly type check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

let mySearch: SearchFunc;

mySearch = function (src: string, sub: string): boolean {

let result = src.search(sub);

return result > -1;

};

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not wantto specify types at all, TypeScript's contextual typing can infer the argument types since the function value is assigned directly to a variable of type SearchFunc. Here, also, the return type of our function expression is implied by the values it returns (here false and true).

let mySearch: SearchFunc;

mySearch = function (src, sub) {

let result = src.search(sub);

return result > -1;

};

Had the function expression returned numbers or strings, the type checker would have made an error that indicates return type doesn't match the return type described in the SearchFunc interface.

let mySearch: SearchFunc;

mySearch = function (src, sub) {

// Type '(src: string, sub: string) => string' is not assignable to type 'SearchFunc'Type 'string' is not assignable to type 'boolean'.

let result = src.search(sub);

return "string";

};

**Indexable Types**

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like a[10], or ageMap["daniel"]. Indexable types have an index signature that describes the types we can use to

index into the object, along with the corresponding return types when indexing.

Let's take an example:

interface StringArray {

[index: number]: string;

}

let myArray: StringArray;

myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];

Above, we have a StringArray interface that has an index signature. This index signature states that when a StringArray is indexed with a number, it will return a string.

There are two types of supported index signatures: string and number. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is

because when indexing with a number, JavaScript will actually convert that to a string before indexing into an object. That means that indexing with 100 (a number) is the same thing as indexing with "100" (a string), so the two need to

be consistent.

interface Animal {

name: string;

}

interface Dog extends Animal {

breed: string;

}

// Error: indexing with a numeric string might get you a completely separate typeinterface NotOkay {

[x: number]: Animal;

Numeric index type 'Animal' is not assignable to string index type 'Dog'.

[x: string]: Dog;

}

While string index signatures are a powerful way to describe the "dictionary" pattern, they also enforce that all properties match their return type. This is because a string index declares that obj.property is also available as

obj["property"]. In the following example, name's type does not match the string index's type, and the type checker gives an error:

interface NumberDictionary {

[index: string]: number;

length: number; // ok, length is a number

name: string; // error, the type of 'name' is not a subtype of the indexer

// Property 'name' of type 'string' is not assignable to string index type 'number'.

}

However, properties of different types are acceptable if the index signature is a union of the property types:

interface NumberOrStringDictionary {

[index: string]: number | string;

length: number; // ok, length is a number

name: string; // ok, name is a string

}

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

interface ReadonlyStringArray {

readonly [index: number]: string;

}

let myArray: ReadonlyStringArray = ["Alice", "Bob"];

myArray[2] = "Mallory"; // error!

// Index signature in type 'ReadonlyStringArray' only permits reading.

// You can't set myArray[2] because the index signature is readonly.

**Class Types**

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

interface ClockInterface {

currentTime: Date;

}

class Clock implements ClockInterface {

currentTime: Date = new Date();

constructor(h: number, m: number) {}

}

You can also describe methods in an interface that are implemented in the class, as we do with setTime in the below example:

interface ClockInterface {

currentTime: Date;

setTime(d: Date): void;

}

class Clock implements ClockInterface {

currentTime: Date = new Date();

setTime(d: Date) {

this.currentTime = d;

}

constructor(h: number, m: number) {}

}

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance. Difference between the static and instance sides of classes When working with classes and interfaces, it helps to keep in mind that a class

has two types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

interface ClockConstructor {

new (hour: number, minute: number);

}

class Clock implements ClockConstructor {

// Class 'Clock' incorrectly implements interface 'ClockConstructor'.

// Type 'Clock' provides no match for the signature 'new (hour: number, minute: numbcurrentTime: Date;

constructor(h: number, m: number) {}

}

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, ClockConstructor for the constructor and ClockInterface for the instance methods. Then, for convenience, we define a

constructor function createClock that creates instances of the type that is passed to it:

interface ClockConstructor {

new (hour: number, minute: number): ClockInterface;

}

interface ClockInterface {

tick(): void;

}

function createClock(

ctor: ClockConstructor,

hour: number,

minute: number

): ClockInterface {

return new ctor(hour, minute);

}

class DigitalClock implements ClockInterface {

constructor(h: number, m: number) {}

tick() {

console.log("beep beep");

}

}

class AnalogClock implements ClockInterface {

constructor(h: number, m: number) {}

tick() {

console.log("tick tock");

}

}

let digital = createClock(DigitalClock, 12, 17);

let analog = createClock(AnalogClock, 7, 32);

Because createClock's first parameter is of type ClockConstructor, in createClock(AnalogClock, 7, 32), it checks that AnalogClock has the correct constructor signature.

Another simple way is to use class expressions:

interface ClockConstructor {

new (hour: number, minute: number): ClockInterface;

}

interface ClockInterface {

tick(): void;

}

const Clock: ClockConstructor = class Clock implements ClockInterface {

constructor(h: number, m: number) {}

tick() {

console.log("beep beep");

}

};

let clock = new Clock(12, 17);

clock.tick();

**Extending Interfaces**

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

interface Shape {

color: string;

}

interface Square extends Shape {

sideLength: number;

}

let square = {} as Square;

square.color = "blue";

square.sideLength = 10;

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

interface Shape {

color: string;

}

interface PenStroke {

penWidth: number;

}

interface Square extends Shape, PenStroke {

sideLength: number;

}

let square = {} as Square;

square.color = "blue";

square.sideLength = 10;

square.penWidth = 5.0;

**Functions**

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of

functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

// Named function

function add(x, y) {

return x + y;

}

// Anonymous function

let myAdd = function (x, y) {

return x + y;

};

Just as in JavaScript, functions can refer to variables outside of the function body. When they do so, they're said to capture these variables.

let z = 100;

function addToZ(x, y) {

return x + y + z;

}

**Function Types**

Typing the function

Let's add types to our simple examples from earlier:

function add(x: number, y: number): number {

return x + y;

}

let myAdd = function (x: number, y: number): number {

return x + y;

};

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at each piece of the function type.

let myAdd: (x: number, y: number) => number = function (

x: number,

y: number

): number {

return x + y;

};

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each

parameter a name and a type. This name is just to help with readability. Wecould have instead written:

let myAdd: (baseValue: number, increment: number) => number = function (

x: number,

y: number

): number {

return x + y;

};

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type. The second part is the return type. We make it clear which is the return type by

using an arrow (=>) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use void instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the "hidden state" of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type even if you only have types on one side of the equation:

// The parameters 'x' and 'y' have the type number

let myAdd = function (x: number, y: number): number {

return x + y;

};

// myAdd has the full function type

let myAdd2: (baseValue: number, increment: number) => number = function

return x + y;

};

This is called "contextual typing", a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given null or undefined, but rather, when the function is called, the compiler will check that the user has provided a value for

each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

function buildName(firstName: string, lastName: string) {

return firstName + " " + lastName;

}

let result1 = buildName("Bob"); // error, too few parameters

// Expected 2 arguments, but got 1.

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

// Expected 2 arguments, but got 3.

let result3 = buildName("Bob", "Adams"); // ah, just right

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is undefined. We can get this functionality in TypeScript by adding a ? to the end of parameters we want to be optional. For

example, let's say we want the last name parameter from above to be optional:

function buildName(firstName: string, lastName?: string) {

if (lastName) return firstName + " " + lastName;

else return firstName;

}

let result1 = buildName("Bob"); // works correctly now

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

Expected 1-2 arguments, but got 3.

let result3 = buildName("Bob", "Adams"); // ah, just right

Any optional parameters must follow required parameters. Had we wanted to make the first name optional, rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place.

These are called default-initialized parameters. Let's take the previous example and default the last name to "Smith".

function buildName(firstName: string, lastName = "Smith") {

return firstName + " " + lastName;

}

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"

let result2 = buildName("Bob", undefined);

// still works, also returns "Bob Smith

let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

// Expected 1-2 arguments, but got 3.

let result4 = buildName("Bob", "Adams"); // ah, just right

Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters, can be omitted when calling their respective function.

function buildName(firstName: string, lastName?: string) {

// ...

}

and

function buildName(firstName: string, lastName = "Smith") {

// ...

}

This means optional parameters and trailing default parameters will share commonality in their types, so bothshare the same type (firstName: string, lastName?: string) => string. The default value of lastName disappears in the type, only leaving behind the fact that the parameter is optional.

Unlike plain optional parameters, default-initialized parameters don't need to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass undefined to get the default initialized value. For example, we could write our last example with only a default initializer on firstName:

function buildName(firstName = "Will", lastName: string) {

return firstName + " " + lastName;

}

let result1 = buildName("Bob"); // error, too few parameters

// Expected 2 arguments, but got 1.

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

// Expected 2 arguments, but got 3.

let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"

let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"

Rest Parameters

Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the arguments variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

}

// employeeName will be "Joseph Samuel Lucas MacKinzie"

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (...), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName

**this**

Learning how to use this in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use this and how to spot when it's not being used correctly. Fortunately,

TypeScript lets you catch incorrect uses of this with a couple of techniques

**this and arrow functions**

In JavaScript, this is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously

confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

let deck = {

suits: ["hearts", "spades", "clubs", "diamonds"],

cards: Array(52),

createCardPicker: function () {

return function () {

let pickedCard = Math.floor(Math.random() \* 52);

let pickedSuit = Math.floor(pickedCard / 13);

return { suit: this.suits[pickedSuit], card: pickedCard % 13 };

};

},

};

let cardPicker = deck.createCardPicker();

let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

Notice that createCardPicker is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the this being used in the function created by createCardPicker

will be set to window instead of our deck object. That's because we call cardPicker() on its own. A top-level non-method syntax call like this will use window for this. (Note: under strict mode, this will be undefined rather than

window).

We can fix this by making sure the function is bound to the correct this before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original deck object. To do this, we change the

function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the this where the function is created rather than where it is invoked:

let deck = {

suits: ["hearts", "spades", "clubs", "diamonds"],

cards: Array(52),

createCardPicker: function () {

// NOTE: the line below is now an arrow function

thisreturn () => {

let pickedCard = Math.floor(Math.random() \* 52);

let pickedSuit = Math.floor(pickedCard / 13);

return { suit: this.suits[pickedSuit], card: pickedCard % 13 };

};

},

};

let cardPicker = deck.createCardPicker();

let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

**this parameters**

Unfortunately, the type of this.suits[pickedSuit] is still any. That's because this comes from the function expression inside the object literal. To fix this, you can provide an explicit this parameter. this parameters are fake parameters that

come first in the parameter list of a function:

function f(this: void) {

// make sure `this` is unusable in this standalone function

}

Let's add a couple of interfaces to our example above, Card and Deck, to make the types clearer and easier to reuse:

interface Card {

suit: string;

card: number;

}

interface Deck {

suits: string[];

cards: number[];

createCardPicker(this: Deck): () => Card;

}

let deck: Deck = {

suits: ["hearts", "spades", "clubs", "diamonds"],

cards: Array(52),

// NOTE: The function now explicitly specifies that its callee must be of type D

createCardPicker: function (this: Deck) {

return () => {

let pickedCard = Math.floor(Math.random() \* 52);

let pickedSuit = Math.floor(pickedCard / 13);

return { suit: this.suits[pickedSuit], card: pickedCard % 13 };

};

},

};

let cardPicker = deck.createCardPicker();

let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

Now TypeScript knows that createCardPicker expects to be called on a Deck object. That means that this is of type Deck now, not any, so --noImplicitThis will not cause any errors.

**this parameters in callbacks**

You can also run into errors with this in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, this will be undefined. With some work you can

use this parameters to prevent errors with callbacks too. First, the library author needs to annotate the callback type with this:

interface UIElement {

addClickListener(onclick: (this: void, e: Event) => void): void;

}

// this: void means that addClickListener expects onclick to be a function that does not require a this type.

// Second, annotate your calling code with this:

class Handler {

info: string;

onClickBad(this: Handler, e: Event) {

// oops, used `this` here. using this callback would crash at runtime

this.info = e.message;

}

}

let h = new Handler();

uiElement.addClickListener(h.onClickBad); // error!

// Argument of type '(this: Handler, e: Event) => void' is not assignable to parameter

// The 'this' types of each signature are incompatible.

// Type 'void' is not assignable to type 'Handler'.

With this annotated, you make it explicit that onClickBad must be called on an instance of Handler. Then TypeScript will detect that addClickListener requires a function that has this: void. To fix the error, change the type of this:

class Handler {

info: string;

onClickGood(this: void, e: Event) {

// can't use `this` here because it's of type void!

console.log("clicked!");

}

}

let h = new Handler();

uiElement.addClickListener(h.onClickGood);

Because onClickGood specifies its this type as void, it is legal to pass to addClickListener. Of course, this also means that it can't use this.info. If you want both then you'll have to use an arrow function:

class Handler {

info: string;

onClickGood = (e: Event) => {

this.info = e.message;

};

}

This works because arrow functions use the outer this, so you can always pass them to something that expects this: void. The downside is that one arrow function is created per object of type Handler. Methods, on the other hand, are only created once and attached to Handler's prototype. They are shared between all objects of type Handler.

**Overloads**

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: any): any {

// Check to see if we're working with an object/array

// if so, they gave us the deck and we'll pick the card

if (typeof x == "object") {

let pickedCard = Math.floor(Math.random() \* x.length);

return pickedCard;

}

// Otherwise just let them pick the card

else if (typeof x == "number") {

let pickedSuit = Math.floor(x / 13);

return { suit: suits[pickedSuit], card: x % 13 };

}

}

let myDeck = [

{ suit: "diamonds", card: 2 },

{ suit: "spades", card: 10 },

{ suit: "hearts", card: 4 },

];

let pickedCard1 = myDeck[pickCard(myDeck)];

alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);

alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

Here, the pickCard function will return two different things based on what the user has passed in. If the users passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card

they've picked. But how do we describe this to the type system?

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our pickCard accepts and what it

returns.

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: { suit: string; card: number }[]): number;

function pickCard(x: number): { suit: string; card: number };

function pickCard(x: any): any {

// Check to see if we're working with an object/array

// if so, they gave us the deck and we'll pick the card

if (typeof x == "object") {

let pickedCard = Math.floor(Math.random() \* x.length);

return pickedCard;

}

// Otherwise just let them pick the card

else if (typeof x == "number") {

let pickedSuit = Math.floor(x / 13);

return { suit: suits[pickedSuit], card: x % 13 };

}

}

let myDeck = [

{ suit: "diamonds", card: 2 },

{ suit: "spades", card: 10 },

{ suit: "hearts", card: 4 },

];

let pickedCard1 = myDeck[pickCard(myDeck)];

alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);

alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

With this change, the overloads now give us type checked calls to the pickCard function. In order for the compiler to pick the correct type check, it follows a similar process to the underlying JavaScript. It looks at the overload list and, proceeding with the first overload, attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, it's customary to order overloads from most specific to least specific.

Note that the function pickCard(x): any piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling pickCard with any other parameter types would cause an error.

**Literal Types**

A literal is a more concrete sub-type of a collective type. What this means is that "Hello World" is a string, but a string is not "Hello World" inside the type system.

There are three sets of literal types available in TypeScript today: strings, numbers, and booleans; by using literal types you can allow an exact value which a string, number, or boolean must have.

**Literal Narrowing**

When you declare a variable via var or let, you are telling the compiler that there is the chance that this variable will change its contents. In contrast, using const to declare a variable will inform TypeScript that this object will never

change.

// We're making a guarantee that this variable

// helloWorld will never change, by using const.

// So, TypeScript sets the type to be "Hello World", not string

const helloWorld = "Hello World";

// On the other hand, a let can change, and so the compiler declares it a string

let hiWorld = "Hi World";

The process of going from an infinite number of potential cases (there are an infinite number of possible string values) to a smaller, finite number of potential case (in helloWorld's case: 1) is called narrowing.

**String Literal Types**

In practice string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings.

type Easing = "ease-in" | "ease-out" | "ease-in-out";

class UIElement {

animate(dx: number, dy: number, easing: Easing) {

if (easing === "ease-in") {

// ...

} else if (easing === "ease-out") {

} else if (easing === "ease-in-out") {

} else {

// It's possible that someone could reach this

// by ignoring your types though.

}

}

}

let button = new UIElement();

button.animate(0, 0, "ease-in");

button.animate(0, 0, "uneasy");

// Argument of type '"uneasy"' is not assignable to parameter of type 'Easing'.

**Numeric Literal Types**

TypeScript also has numeric literal types, which act the same as the string literals above.

function rollDice(): 1 | 2 | 3 | 4 | 5 | 6 {

return (Math.floor(Math.random() \* 6) + 1) as 1 | 2 | 3 | 4 | 5 | 6;

}

const result = rollDice();

A common case for their use is for describing config values:

interface MapConfig {

lng: number;

lat: number;

tileSize: 8 | 16 | 32;

}

setupMap({ lng: -73.935242, lat: 40.73061, tileSize: 16 });

**Boolean Literal Types**

TypeScript also has boolean literal types. You might use these to constrain object values whose properties are interrelated.

interface ValidationSuccess {

isValid: true;

reason: null;

}

interface ValidationFailure {

isValid: false;

reason: string;

}

type ValidationResult = ValidationSuccess | ValidationFailure;

**Classes**

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality

and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach. In TypeScript, we allow

developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return "Hello, " + this.greeting;

}

}

let greeter = new Greeter("world");

The syntax should look familiar if you've used C# or Java before. We declare a new class Greeter. This class has three members: a property called greeting, a constructor, and a method greet. You'll notice that in the class when we refer to one of the members of the class we prepend this.. This denotes that it's a member access.

In the last line we construct an instance of the Greeter class using new. This calls into the constructor we defined earlier, creating a new object with the Greeter shape, and running the constructor to initialize it.

**Inheritance**

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance. Let's take a look at an example:

class Animal {

move(distanceInMeters: number = 0) {

console.log(`Animal moved ${distanceInMeters}m.`);

}

}

class Dog extends Animal {

bark() {

console.log("Woof! Woof!");

}

}

const dog = new Dog();

dog.bark();

dog.move(10);

dog.bark();

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, Dog is a derived class that derives from the Animal base class using the extends keyword. Derived classes are often called

subclasses, and base classes are often called superclasses.

Because Dog extends the functionality from Animal, we were able to create an instance of Dog that could both bark() and move().

Let's now look at a more complex example.

class Animal {

name: string;

constructor(theName: string) {

this.name = theName;

}

move(distanceInMeters: number = 0) {

console.log(`${this.name} moved ${distanceInMeters}m.`);

}

}

class Snake extends Animal {

constructor(name: string) {

super(name);

}

move(distanceInMeters = 5) {

console.log("Slithering...");

super.move(distanceInMeters);

}

}

class Horse extends Animal {

constructor(name: string) {

super(name);

}

move(distanceInMeters = 45) {

console.log("Galloping...");

super.move(distanceInMeters);

}

}

let sam = new Snake("Sammy the Python");

let tom: Animal = new Horse("Tommy the Palomino");

sam.move();

tom.move(34);

This example covers a few other features we didn't previously mention. Again, we see the extends keywords used to create two new subclasses of Animal: Horse and Snake.

One difference from the prior example is that each derived class that contains a constructor function must call super() which will execute the constructor of the base class. What's more, before we ever access a property on this in a

constructor body, we have to call super(). This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both Snake and Horse create a move method that overrides the move from Animal, giving it functionality specific to

each class. Note that even though tom is declared as an Animal, since its value is a Horse, calling tom.move(34) will call the overriding method in Horse:

Slithering...

Sammy the Python moved 5m.

Galloping...

Tommy the Palomino moved 34m.

**Public, private, and protected modifiers (Public by default)**

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, youmay have noticed in the above examples we haven't had to use the word public to accomplish this; for instance, C# requires that each member be explicitly labeled public to be visible. In TypeScript, each member is public by default.

You may still mark a member public explicitly. We could have written the Animal class from the previous section in the following way:

class Animal {

public name: string;

public constructor(theName: string) {

this.name = theName;

}

public move(distanceInMeters: number) {

console.log(`${this.name} moved ${distanceInMeters}m.`);

}

}

**ECMAScript Private Fields**

With TypeScript 3.8, TypeScript supports the new JavaScript syntax for private fields:

class Animal {

#name: string;

constructor(theName: string) {

this.#name = theName;

}

}

new Animal("Cat").#name;

Property '#name' is not accessible outside class 'Animal' because it has a privateThis syntax is built into the JavaScript runtime and can have better guarantees about the isolation of each private field.

**Understanding TypeScript's private**

TypeScript also has its own way to declare a member as being marked private, it cannot be accessed from outside of its containing class. For example:

class Animal {

private name: string;

constructor(theName: string) {

this.name = theName;

}

}

new Animal("Cat").name;

Property 'name' is private and only accessible within class 'Animal'. TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible,

then we say the types themselves are compatible.

However, when comparing types that have private and protected members, we treat these types differently. For two types to be considered compatible, if one of them has a private member, then the other must have a private member that originated in the same declaration. The same applies to protected members.

Let's look at an example to better see how this plays out in practice:

class Animal {

private name: string;

constructor(theName: string) {

this.name = theName;

}

}

class Rhino extends Animal {

constructor() {

super("Rhino");

}

}

class Employee {

private name: string;

constructor(theName: string) {

this.name = theName;

}

}

let animal = new Animal("Goat");

let rhino = new Rhino();

let employee = new Employee("Bob");

animal = rhino;

animal = employee;

// Type 'Employee' is not assignable to type 'Animal'.

Types have separate declarations of a private property 'name'.

In this example, we have an Animal and a Rhino, with Rhino being a subclass of Animal. We also have a new class Employee that looks identical to Animal in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because Animal and Rhino share the private side of their shape from the same declaration of private name: string in Animal, they are compatible. However, this is not the case for Employee. When we try to assign from an Employee to Animal we get an error that these types are not compatible. Even though Employee also has a private member called name, it's not the one we declared in Animal.

**Understanding protected**

The protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed within deriving classes. For example,

class Person {

protected name: string;

constructor(name: string) {

this.name = name;

}

}

class Employee extends Person {

private department: string;

constructor(name: string, department: string) {

super(name);

this.department = department;

}

public getElevatorPitch() {

return `Hello, my name is ${this.name} and I work in ${this.department

}

}

let howard = new Employee("Howard", "Sales");

console.log(howard.getElevatorPitch());

console.log(howard.name);

// Property 'name' is protected and only accessible within class 'Person' and its subc

Notice that while we can't use name from outside of Person, we can still use it from within an instance method of Employee because Employee derives from Person.

A constructor may also be marked protected. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

class Person {

protected name: string;

protected constructor(theName: string) {

this.name = theName;

}

}

// Employee can extend Person

class Employee extends Person {

private department: string;

constructor(name: string, department: string) {

super(name);

this.department = department;

}

public getElevatorPitch() {

return `Hello, my name is ${this.name} and I work in ${this.department

}

}

let howard = new Employee("Howard", "Sales");

let john = new Person("John");

// Constructor of class 'Person' is protected and only accessible within the class dec

**Readonly modifier**

You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

class Octopus {

readonly name: string;

readonly numberOfLegs: number = 8;

constructor(theName: string) {

this.name = theName;

}

}

let dad = new Octopus("Man with the 8 strong legs");

dad.name = "Man with the 3-piece suit";

// Cannot assign to 'name' because it is a read-only property.

**Parameter properties**

In our last example, we had to declare a readonly member name and a constructor parameter theName in the Octopus class. This is needed in order to have the value of theName accessible after the Octopus constructor is executed. Parameter properties let you create and initialize a member in one place. Here's a further revision of the previous Octopus class using a parameter property:

class Octopus {

readonly numberOfLegs: number = 8;

constructor(readonly name: string) {}

}

let dad = new Octopus("Man with the 8 strong legs");

dad.name;

Notice how we dropped theName altogether and just use the shortened readonly name: string parameter on the constructor to create and initialize the name member. We've consolidated the declarations and assignment into one location. Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or readonly, or both. Using private for a parameter property declares and initializes a private member; likewise, the same is done for public, protected, and readonly.

**Accessors**

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use get and set. First, let's start with an example without getters and setters.

class Employee {

fullName: string;

}

let employee = new Employee();

employee.fullName = "Bob Smith";

if (employee.fullName) {

console.log(employee.fullName);

}

While allowing people to randomly set fullName directly is pretty handy, we may also want enforce some constraints when fullName is set.

In this version, we add a setter that checks the length of the newName to make sure it's compatible with the max-length of our backing database field. If it isn't we throw an error notifying client code that something went wrong.

To preserve existing functionality, we also add a simple getter that retrieves fullName unmodified.

const fullNameMaxLength = 10;

class Employee {

private \_fullName: string = "";

get fullName(): string {

return this.\_fullName;

}

set fullName(newName: string) {

if (newName && newName.length > fullNameMaxLength) {

throw new Error("fullName has a max length of " + fullNameMaxLength

}

this.\_fullName = newName;

}

}

let employee = new Employee();

employee.fullName = "Bob Smith";

if (employee.fullName) {

console.log(employee.fullName);

}

To prove to ourselves that our accessor is now checking the length of values, we can attempt to assign a name longer than 10 characters and verify that we get an error.

**A couple of things to note about accessors:**

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downleveling to ECMAScript 3 is not supported. Second, accessors with a get and no set are automatically inferred to be readonly. This is helpful when

generating a .d.ts file from your code, because users of your property can see that they can't change it.

**Static Properties**

Up to this point, we've only talked about the instance members of the class, those that show up on the object when it's instantiated. We can also create static members of a class, those that are visible on the class itself rather than on the

instances. In this example, we use static on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending this. in front of instance accesses, here we

prepend Grid. in front of static accesses.

class Grid {

static origin = { x: 0, y: 0 };

calculateDistanceFromOrigin(point: { x: number; y: number }) {

let xDist = point.x - Grid.origin.x;

let yDist = point.y - Grid.origin.y;

return Math.sqrt(xDist \* xDist + yDist \* yDist) / this.scale;

}

constructor(public scale: number) {}

}

let grid1 = new Grid(1.0); // 1x scale

let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));

console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));

**Abstract Classes**

Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members. The abstract keyword is used

to define abstract classes as well as abstract methods within an abstract class.

abstract class Animal {

abstract makeSound(): void;

move(): void {

console.log("roaming the earth...");

}

}

Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes. Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body. However, abstract methods must include the abstract keyword and may optionally include access modifiers.

abstract class Department {

constructor(public name: string) {}

printName(): void {

console.log("Department name: " + this.name);

}

abstract printMeeting(): void; // must be implemented in derived classes

}

class AccountingDepartment extends Department {

constructor() {

super("Accounting and Auditing"); // constructors in derived classes must call}

printMeeting(): void {

console.log("The Accounting Department meets each Monday at 10am.");

}

generateReports(): void {

console.log("Generating accounting reports...");

}

}

let department: Department; // ok to create a reference to an abstract type

department = new Department(); // error: cannot create an instance of an abstractCannot create an instance of an abstract class.

department = new AccountingDepartment(); // ok to create and assign a non-abstractdepartment.printName();

department.printMeeting();

department.generateReports(); // error: department is not of type AccountingDepartProperty 'generateReports' does not exist on type 'Department'.

**Using a class as an interface**

A class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

class Point {

x: number;

y: number;

}

interface Point3d extends Point {

z: number;

}

let point3d: Point3d = { x: 1, y: 2, z: 3 };

**Enums**

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript. Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

**Numeric enums**

We'll first start off with numeric enums, which are probably more familiar if you're coming from other languages. An enum can be defined using the enum keyword.

enum Direction {

Up = 1,

Down,

Left,

Right,

}

Above, we have a numeric enum where Up is initialized with 1. All of the following members are auto-incremented from that point on. In other words, Direction.Up has the value 1, Down has 2, Left has 3, and Right has 4. If we wanted, we could leave off the initializers entirely:

enum Direction {

Up,

Down,

Left,

Right,

}

Here, Up would have the value 0, Down would have 1, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

enum UserResponse {

No = 0,

Yes = 1,

}

function respond(recipient: string, message: UserResponse): void {

// ...

}

respond("Princess Caroline", UserResponse.Yes);

Numeric enums can be mixed in computed and constant members. The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn't allowed:

enum E {

A = getSomeValue(),

B,

Enum member must have initializer.

}

**String enums**

In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

enum Direction {

Up = "UP",

Down = "DOWN",

Left = "LEFT",

Right = "RIGHT",

}

**Computed and constant members**

Each enum member has a value associated with it which can be either constant or computed. An enum member is considered constant if:

It is the first member in the enum and it has no initializer, in which case it's assigned the value 0:

// E.X is constant:

enum E {

X,

}

It does not have an initializer and the preceding enum member was a numeric constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.

// All enum members in 'E1' and 'E2' are constant.

enum E1 {

X,

Y,

Z,

}

enum E2 {

A = 1,

B,

C,

}

The enum member is initialized with a constant enum expression. A constant enum expression is a subset of TypeScript expressions that can be fully evaluated at compile time. An expression is a constant enum expression if it is:

1. a literal enum expression (basically a string literal or a numeric literal)

2. a reference to previously defined constant enum member (which can originate from a different enum)

3. a parenthesized constant enum expression

4. one of the +, -, ~ unary operators applied to constant enum expression

5. +, -, \*, /, %, <<, >>, >>>, &, |, ^ binary operators with constant enum expressions as operands

It is a compile time error for constant enum expressions to be evaluated to

**NaN or Infinity.**

In all other cases enum member is considered computed.

enum FileAccess {

// constant members

None,

Read = 1 << 1,

Write = 1 << 2,

ReadWrite = Read | Write,

// computed member

G = "123".length,

}

**Reverse mappings**

In addition to creating an object with property names for members, numeric enums members also get a reverse mapping from enum values to enum names.

For example, in this example:

enum Enum {

A,

}

let a = Enum.A;

let nameOfA = Enum[a]; // "A"

TypeScript compiles this down to the following JavaScript:

"use strict";

var Enum;

(function (Enum) {

Enum[Enum["A"] = 0] = "A";

})(Enum || (Enum = {}));

let a = Enum.A;

let nameOfA = Enum[a]; // "A"

In this generated code, an enum is compiled into an object that stores both forward (name -> value) and reverse (value -> name) mappings. References to other enum members are always emitted as property accesses and never inlined.

Keep in mind that string enum members do not get a reverse mapping generated at all.