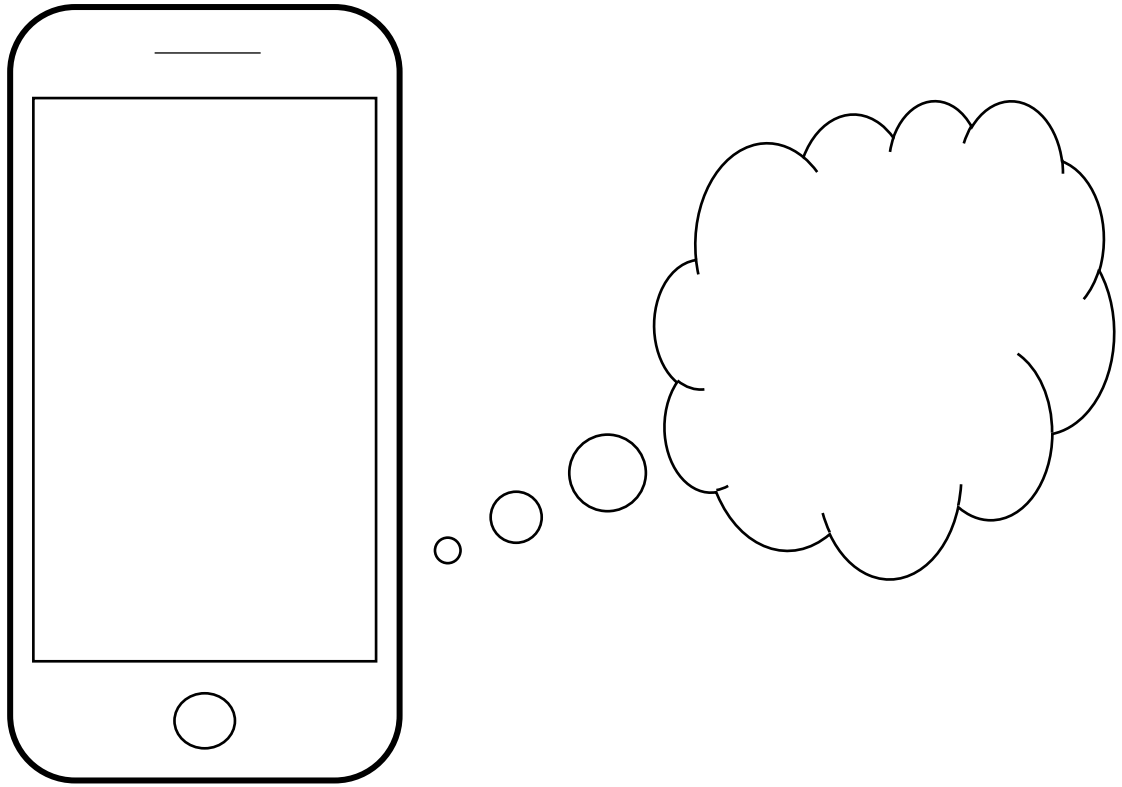
# 335 – The Other Half

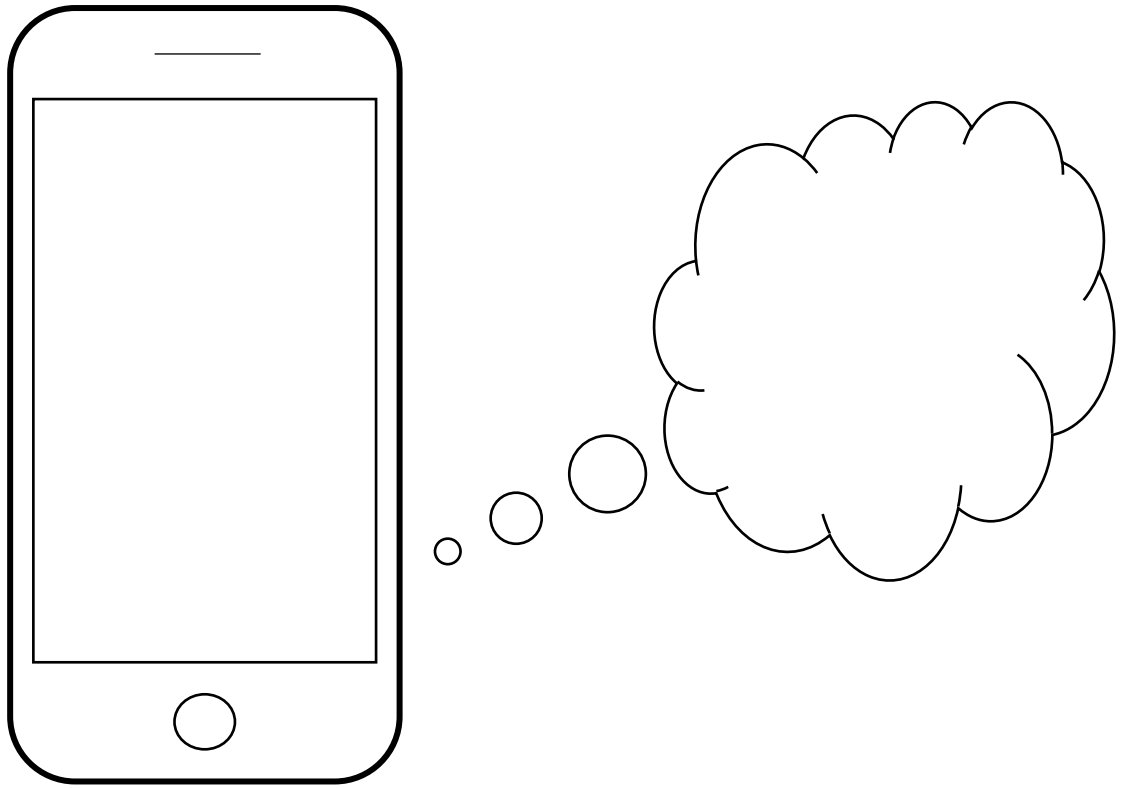
Tell me and I forget. Teach me and I remember. Involve me and I learn.

# The Hottest Programming Model



Client and the Cloud

# The Hottest Programming Model



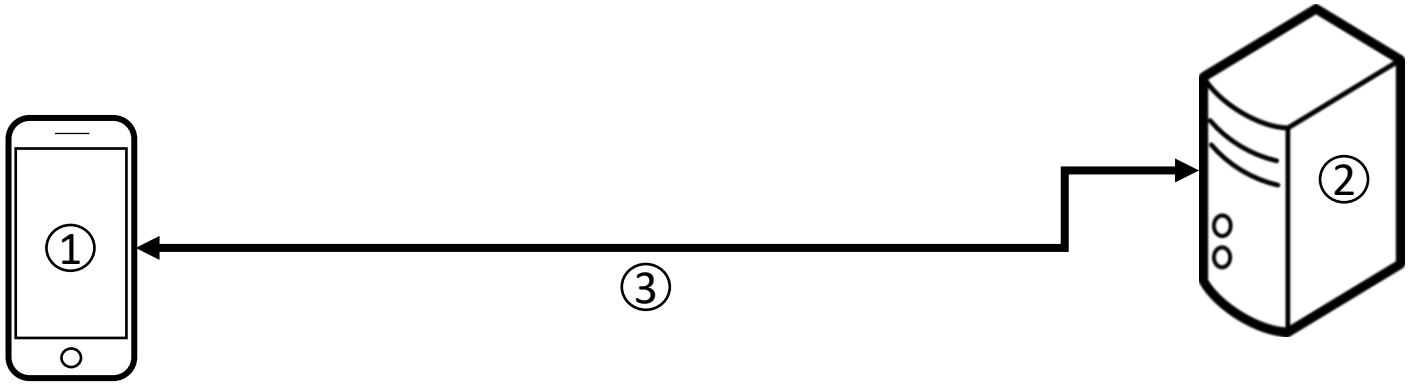
Client and the Cloud

# An Application

- Runs on a slick device
- Reactive and responsive – actions without wholesale screen refreshes
- Data stored locally and/or across on a server
- Asynchronous access to server if/when required
- Could be available off-line

# Parts of the Application

1. The client on the device
2. Server(s) offering services and data to support the client
   - Facility to store and retrieve (potentially large amounts of) data
   - Facility to compute results (e.g., data aggregation)
   - Facility to pre-compute results (e.g., search indices)
   - Deliver data and results
3. A network supporting the transfer of data and results



# Client Application

- Native Applications – iOS, Android, and Windows
- Web Applications – Using the native browser as the UI

# Server Backend

- Provides for
  - Storage
  - Heavy lifting – memory, CPU and/or storage intensive computations
  - Centralized handling of services for multiple platforms (iOS, Android, …)

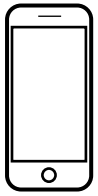
# Client vs. Server

The BIG question…
How to share the work?

# Client-Server Communication

- Common data exchange formats – JSON and XML
- Data transfer protocols – HTTP and HTTPS
- Security concerns
  - Who is tapping my wire?
  - Is the client really talking to the server … or an imposter?
  - Is the server really talking to the client … or an imposter?
- Performance concerns
  - What price data transfers? – cost, time delays, …

# The Big Picture

| Client | Network – the middle layer | Server |
| --- | --- | --- |
|  Web Applications<br>HTML<br>CSS<br>JavaScript<br>SVG | Data Exchange<br>JSON<br>XML<br>XML schema<br><br>Data Transfer Protocols<br>HTTP<br>HTTPS | Web Services<br>Web API<br>Data store<br>C# |
|  Native Applications | | |
| Security: Privacy, Trust, Authenticity, Integrity  | Security and performance testing. |  Performance: Latency reduction techniques Caching & compression |

¿?

# HTML

mano@cs.auckland.ac.nz

# An Application

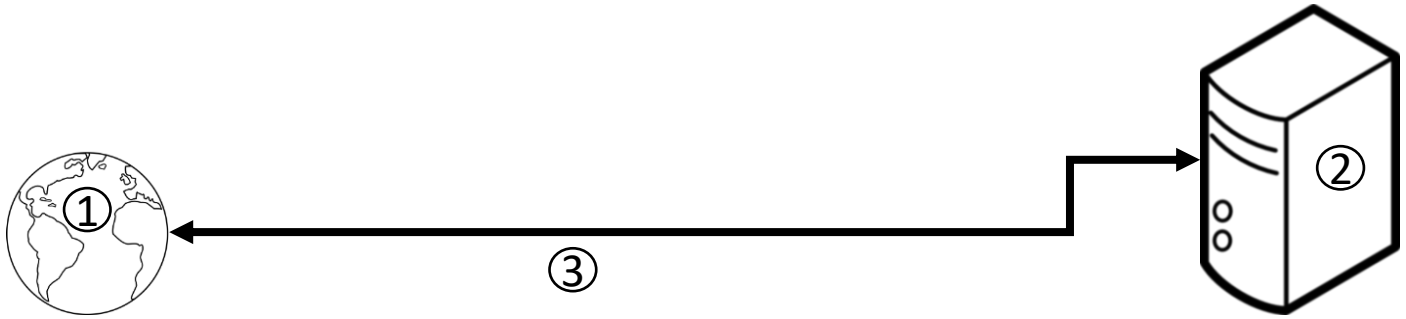
- Runs on a slick device
- Reactive and responsive – actions without wholesale screen refreshes
- Data stored locally and/or across on a server
- Asynchronous access to server if/when required
- Could be available off-line

# A Web Application

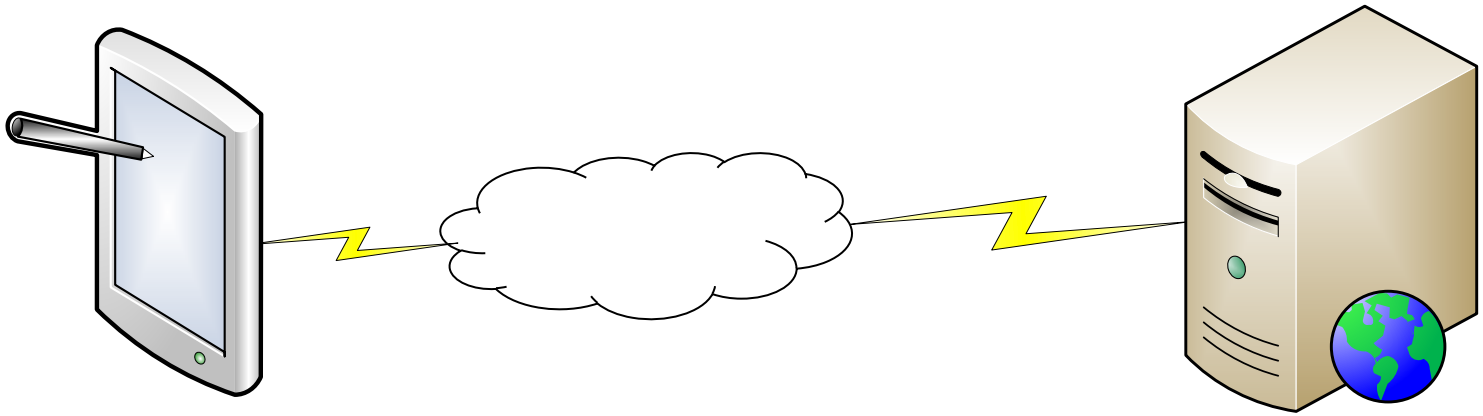
- Uses a web browser for its user-interface, and is platform-agnostic
- Reactive and responsive – actions without wholesale screen refreshes
- Data stored locally and/or across on a server
- Asynchronous access to server if/when required
- Could be available off-line

# Parts of a Web Application

1. The client application that runs on the browser is the user-facing part
2. Server(s) offer services and data to support the client
   - Facility to store and retrieve (potentially large amounts of) data
   - Facility to compute results (e.g., data aggregation)
   - Facility to pre-compute results (e.g., search indices)
   - Deliver data and results
3. A network supports the transfer of data and results

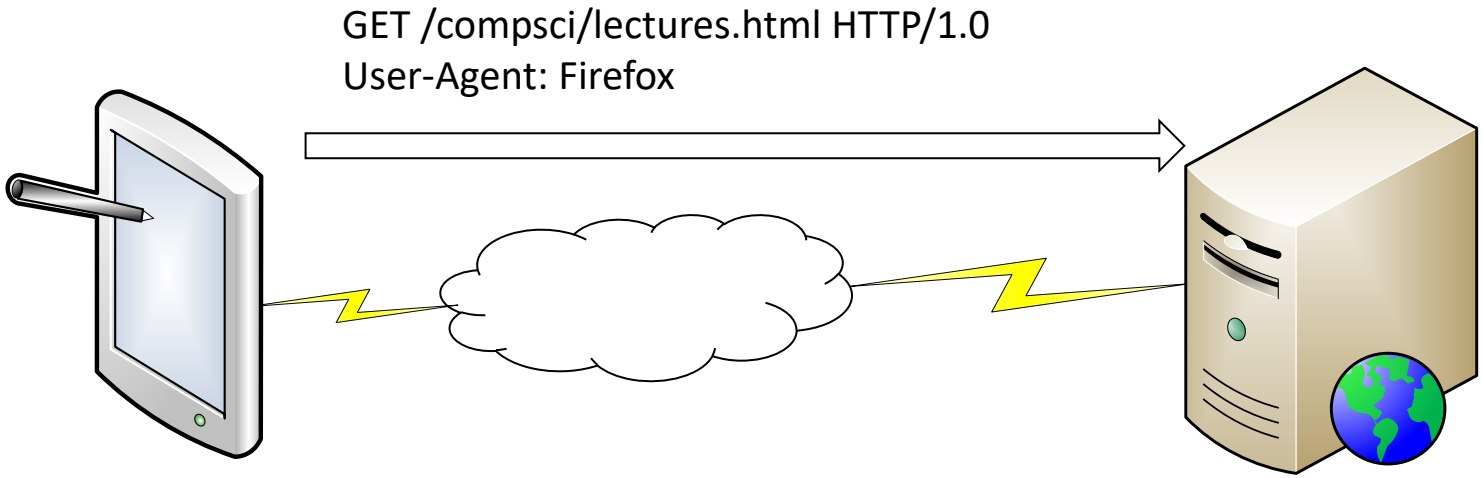


# HTTP Essentials



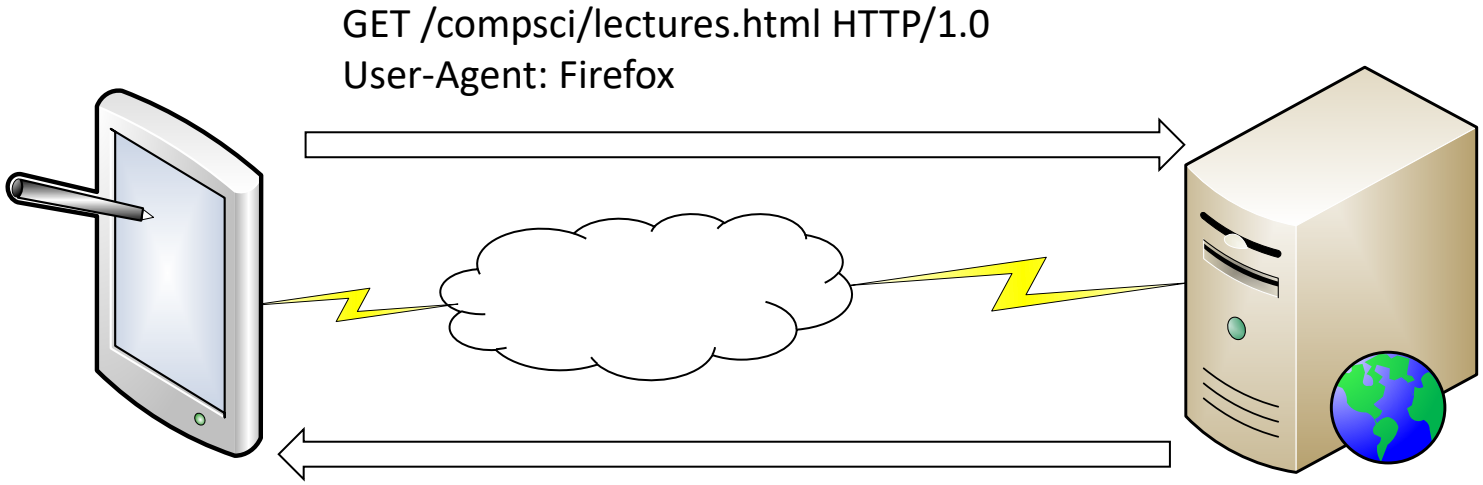
http://www.auckland.ac.nz:8080/compsci/lectures.html

# HTTP Essentials




http://www.auckland.ac.nz:8080/compsci/lectures.html

# HTTP Essentials




HTTP/1.1 200 OK
Date: Sun, 20 Jan 2020 20:20:20 GMT
Server: Apache
Expires: Sun, 20 Jan 2020 22:20:20 GMT
Content-Type: text/html
Connection: close

{Contents of /compsci/lectures.html}

http://www.auckland.ac.nz:8080/compsci/lectures.html

# A Web Application – The Client Part

- Three primary building blocks
  1. Information – HTML
  2. Presentation – CSS (cascading style sheets)
  3. Interaction – JavaScript

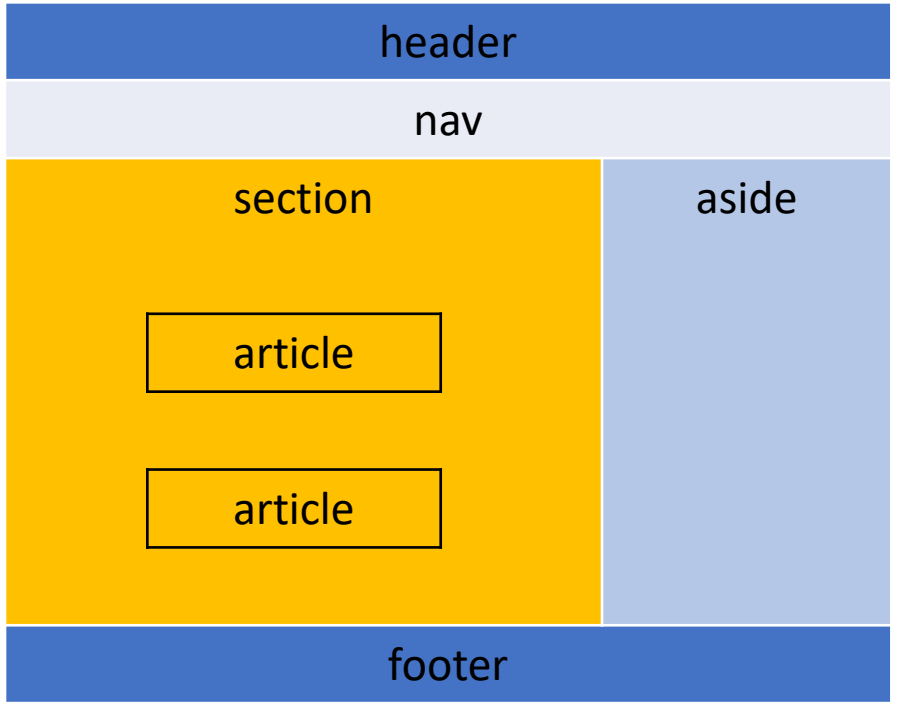
- We have other media – such as images – that too are part of these blocks

Try https://developer.mozilla.org/en-US/Learn/Getting_started_with_the_web

# HTML – An Example

```
<!DOCTYPE html>
<html>
<head>
   <title>Page Title</title>
</head>
<body>
   <h1>This is a heading</h1>
   <p>This is a paragraph.</p>
</body>
</html>
```

# HTML Semantic Elements




The non-semantic elements  and  do not tell anything about their content.

```
<!DOCTYPE html>
<html>
<head>
   <title></title>
</head>
<body>
   <header></header>
   <nav></nav>
   <section>
      <article></article>
      <article></article>
   </section>
   <aside></aside>
   <footer></footer>
</body>
</html>
```

# Examples

- `<header><h1>Welcome Amigos</h1></header>`
- `<footer><p>&copy; UoA</p></footer>`
- `<footer></footer>`
- `<footer/> <!– Self-closing not OK here -->`
- `<hr/> <!– Self-closing OK here -->`

- There are more semantic elements: `<details>`, `<figure>`, `<figcaption>`, `<summary>`, and `<time>`.

Try the HTML tutorial at http://www.w3schools.com/html/

¿?

# Cascading Stylesheets (CSS)

mano@cs.auckland.ac.nz

# Cascading Style Sheets

- CSS is a language that describes the presentation aspects (look & format) of an HTML document.
- This allows separation of concerns between the document's content and the document's presentation.

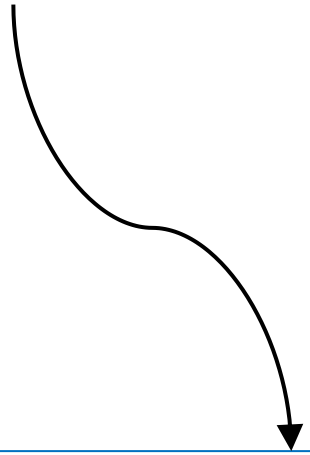
Try the CSS tutorial at http://www.htmldog.com/guides/css/ and the online HTML/CSS/JavaScript playground at http://jsfiddle.net/

# Cascading Style Sheets – An Internal Example

```
<!DOCTYPE html>
<html>
<head>
   <title>Page Title</title>
   <style>
      h1 { font-family: 'Times New Roman'; color: red; }
      p  { font-family: 'Times New Roman'; color: black; }
   </style>
</head>
<body>
   <h1>This is a heading</h1>
   <p>This is a paragraph.</p>
</body>
</html>
```

# Cascading Style Sheets – An External Example

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



```
h1 { font-family: 'Times New Roman'; color: red; }
p  { font-family: 'Times New Roman'; color: black; }
```

# Cascading Style Sheets

- Format:

```
selector { property: value; property: value; ... }
```

- Example:

```
h1 { font-family: 'Times New Roman'; color: red; }
p  { font-family: 'Times New Roman'; color: black; }
```

Note : Some values require units. E.g., `font-size: 12px; width: 90%;`.

# Cascading Style Sheets – IDs

- Instances of an HTML element (e.g., h1) can be separately styled.
- Such instances are uniquely identified by an `id` attribute of the HTML element.
  - `<h1 id="title">Courses</h1>`
  - `#title { font-family: 'Times New Roman'; color: red; }`

# Cascading Style Sheets – IDs

```
<!DOCTYPE html>
<html>
<head>
   <title>Page Title</title>
   <link rel="stylesheet" href="mystyle.css">
</head>
<body>
   <h1 id="title">This is a heading</h1>
   <p>This is a paragraph.</p>
   <h1>Here's another heading</h1>
   <p>And another paragraph.</p>
</body>
</html>
```

```
<h1 style="font-family: 'Times New Roman'; color: red;">...</h1>
```

```
#title { font-family: 'Times New Roman'; color: red; }
p  { font-family: 'Times New Roman'; color: black; }
```

# Cascading Style Sheets – Classes

- While IDs allow us to style an instance of an HTML element (e.g., h1) separately, classes allow us to style related instances of HTML elements together.
- Such instances are uniquely identified by a `class` attribute of the HTML element.
  - The class attribute categorizes these instances into some user-defined entity.
  - `<h1 class="course">Courses</h1>`
  - `<p class="course">COMPSCI 110</p>`
  - ```
    .course { font-family: 'Times New Roman';
              color: #4099FF; }
    ```

Note: See http://html-color-codes.info/ for colour codes.
    ```

# Styling with Id and Class Selectors

```
<h1 class="course">All Courses</h1>
<div id="allCourses">
   <p class="course">COMPSCI 101</p>
   <p class="course">COMPSCI 110</p>
   <p class="course">COMPSCI 120</p>
   <p class="course">COMPSCI 130</p>
   <p>Note: Not all courses are offered every semester.</p>
</div>
<h1 class="course">Selected Courses</h1>
<div id="selectedCourses">
      <p class="course">COMPSCI 110</p>
</div>
```

The `<div>` containers identify block sections. The `id` attribute allows us to give an element a unique id. The `class` attribute allows us to specify a (user-defined) category for an element. While `id` is used to uniquely identify a single element, `class` is used to identify more than one element.

# Styling with Id and Class Selectors

```
<h1 class="course">All Courses</h1>
<div id="allCourses">
   <p class="course">COMPSCI 101</p>
   <p class="course">COMPSCI 110</p>
   <p class="course">COMPSCI 120</p>
   <p class="course">COMPSCI 130</p>
   <p>Note: Not all courses are offered every semester.</p>
</div>
<h1 class="course">Selected Courses</h1>
<div id="selectedCourses">
      <p class="course">COMPSCI 110</p>
</div>
```

```
#allCourses      { background-color: cornflowerblue; }
#selectedCourses { background-color: yellow; transform: rotate(180deg); }
.course          { color: blue; }
```

# Styling to Hide

```
<p style="display: none">you never see this one!!</p>
```

# Styling to Show

```
<p style="display: block">you'd see this one!!</p>
<p style="display: block">... and this one!!!!</p>
```

Exercise: Change the display type to inline (rather than block) and see what difference it makes.

# Exercise

```
#main {
   width: 97%;
}

   #main p, #main div {
      font-size: 2em;
   }

      #main p a, #main div a {
         font-weight: bold;
      }

   #main pre {
      font-size: 3em;
   }
```

Work out what this style sheet says.

# Exercise

```
<html>
<head>
   <title>Picture with Caption</title>
</head>
<body style="background: linear-gradient(to right, #4099FF, lightblue);">
   <div style="width: 90%; margin: auto; background: #424242;">
      <img style="width: 100%;" alt="pic" src="panorama.jpg" />
      <p style="font-family: arial, helvetica, serif; color: white;">
         Scenic New Zealand in Summer. &copy; Us &#xa9;.
      </p>
   </div>
</body>
</html>
```

Work out what this HTML page says.

# Exercise

- Try the CSS selector workout at https://flukeout.github.io/
- Watch the CSS tutorial at https://www.youtube.com/watch?v=CUxH_rWSI1k

¿?

# JavaScript – Objects

mano@cs.auckland.ac.nz

# JavaScript

- JavaScript brings life to a Web application, adding interactivity and dynamism.

Try the online HTML/CSS/JavaScript playground at http://jsfiddle.net/ and the free JavaScript book at http://eloquentjavascript.net/

# JavaScript – Example #1

On the button click, select the <h1> element, and change its inner HTML content to the return value of the (built-in) function Date(). The argument to `querySelector` is a CSS selector.

```
<!DOCTYPE html>
<html>
<head><title></title></head>
<body>
    <button onclick='document.querySelector("h1").innerText = Date()'>
        Click me.
    </button>
    <h1></h1>
</body>
</html>
```

Question: what happens if there are two <h1> elements?

# JavaScript – Example #2

```
<!DOCTYPE html>
<html>
<head>
   <title></title>
</head>
<body>
   <h2>Our first heading</h2>
   <h2>And our second heading</h2>
   <script>
      document.querySelector("h2").innerText = Date();
   </script>
</body>
</html>
```

# JavaScript – Example #2a

```
<!DOCTYPE html>
<html>
<head>
   <title></title>
   <script>
      document.querySelector("h2").innerText = Date();
   </script>
</head>
<body>
   <h2>Our first heading</h2>
   <h2>And our second heading</h2>
</body>
</html>
```

# JavaScript – Example #3

```
<!DOCTYPE html>
<html>
<head>
   <title></title>
</head>
<body>
   <h2>Our first heading</h2>
   <h2 id="pickThis">And our second heading</h2>
   <script>
      document.getElementById("pickThis").innerText = Date();
   </script>
</body>
</html>
```

# JavaScript Declarations

- With keyword let to declare a block-scoped, local variable, optionally initializing it to a value
- With keyword const to declare a block-scoped, read-only named constant

- Do NOT use the keyword var to declare a variable – Legacy & must avoid

# JavaScript Variables

- Variables are dynamically typed: the runtime infers the type.
- A variable can have different types during its life.

```
let me = "say hello"
me = 42
me += 24
me = false
alert(me)
```

Read http://blogs.perl.org/users/ovid/2010/08/what-to-know-before-debating-type-systems.html

# JavaScript Variables (2)

- If a variable is not assigned a value, the variable has the type undefined and its value is undefined as well.

```
let q;
alert(q); // will show 'undefined'
alert(typeof(q)); // will show 'undefined'
```

- A variable can be assigned a null value. It represents an intentional absence of any object value.

```
let q = null;
alert(q); // will show 'null'
alert(typeof(q)); // will show 'object'
```

# JavaScript – Objects

```
const pi = 3.142; // simple object

const user = {    // complex object
          name: "Bond",
          age: 30,
          spy: true,
          address: null
};
```

Fields of a complex object can be accessed using the familiar dot notation. E.g., `user.name`.

One can add new fields on the fly. E.g.,

```
user.phone = "+44 20 7601 2407";
user.swear = () => { alert("$@1^"); }; // lambda!
// properties of const objects are not protected from mods.
```

# JavaScript – Equality

```
if ("4" == 4) { alert("\'4\' == 4"); }
else { alert('not! "4" == 4'); }

if ("4" === 4) { alert("\'4\' === 4"); }
else { alert('not!! "4" === 4'); }
```

The operator == checks if the compared values are equal (converting both values to a common type), while the operator === checks if the compared values *and* their associated types are equal.

# JavaScript

mano@cs.auckland.ac.nz

# JavaScript – Functions

```
const pi = 3.142; // simple object

function square(n) { return n * n; } // classic function

function circleArea(r) { return pi * square(r); }

const circumference = function (r) { return 2 * pi * r; } // func expression

const circumference2 = (r) => { return 2 * pi * r; } // arrow func expression

const circumference3 = (r) => 2 * pi * r // arrow func expression
```

# JavaScript – Functions

```
const pi = 3.142; // simple object

const temp = circleArea(pi); // OK: classic functions are 'hoisted'
alert(temp);

function square(n) { return n * n; }

function circleArea(r) { return pi * square(r); }

const circumference = (r) => 2 * pi * r // arrow func expression
```

# JavaScript – Functions

```
const pi = 3.142; // simple object

const temp = circumference(pi);
             // Not OK: objects & func expressions aren't 'hoisted'
alert(temp);

function square(n) { return n * n; }

function circleArea(r) { return pi * square(r); }

const circumference = (r) => 2 * pi * r // arrow func expression
```

# JavaScript & JSON

mano@cs.auckland.ac.nz

# JSON – Representing Objects

- A simple format for representing objects using key-value pairs
  - { "courseId": "COMPCI 110", "title": "Computer Systems" }
  - { "courseId": "COMPCI 110", "isOffered": true, "year" : 2020 }

  - Can have arrays of objects in [ ]
  - "courses" : [

  { "courseId": "COMPCI 110", "title": "Computer Systems" },
  { "courseId": "COMPCI 120", "title": "Math for Computing" }
  ]

Question: How does one handle circular references?

# JSON types

- Boolean – true or false
- Null – Where the value defined to be nothing. There is no type to be associated with null.
- Number – This is a signed decimal number such as 42, 42.42, or 42.42e42.
- String – a sequence of characters delimited by double quotes.
- Object – name-value pairs with ‘:’ separating the two. The name is a string serving as a key
- Array – A comma-separated list of objects within square brackets.

Refer to ECMA-404: The JSON Data Interchange Syntax

# XML – An Alternative Object Representation

```
<course>
   <courseId>COMPCI 101</courseId>
   <title>Programming Intro 1</title>
</course>
```

```
<course courseId="COMPCI 101">
   <title>Programming Intro 1</title>
</course>
```

```
<course courseId="COMPCI 101" title="Programming Intro 1">
</course>
```

```
<course courseId="COMPCI 101" title="Programming Intro 1" />
```

# Consuming JSON

```
const someJsonString =
    '{ "courseId": "COMPCI 101", "title": "Programming Intro 1" }';
const myObj = JSON.parse(someJsonString);

alert(myObj.title);
```

# Producing JSON

```
const jsonString = JSON.stringify(myObj);
alert(jsonString);
```

# JavaScript – Fetch

mano@cs.auckland.ac.nz

# Fetch API – Gateway to the network

- Fetch API in JavaScript provides communication to the outside network (e.g., servers).
- This sends requests asynchronously and take some action upon receipt of the response.
  - E.g., update just the required part of the page (rather than refreshing the entire page).
- Forms the basis of network interaction.

Read the API specification at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

# Fetch

```
const fetchPromise =
    fetch('https://dividni.com/cors/version');




const streamPromise =
    fetchPromise.then((response) => response.text());

streamPromise.then((data) => alert(data));
```

# Fetch (2)

```
const fetchPromise =
    fetch('https://dividni.com/cors/version',
          {
              headers : {
                  "Accept" : "application/json",
              },
          });
const streamPromise =
    fetchPromise.then((response) => response.json());

streamPromise.then((data) => alert(data));
```

# Fetch – POST data

```
const fetchPromise =
    fetch(serverUri,
          {
              headers : {
                  "Content-Type" : thisContentsType,
              },
              method : "POST",
              body : contentToSend
          });
```

# JavaScript – Arrays & Higher-order Functions

mano@cs.auckland.ac.nz

# Operations on Arrays – forEach

```
let towns = ['Auckland', 'Wellington', 'Dunedin', 'ChCh', 'Hamilton']


for (let i = 0; i < towns.length; ++i) {
   alert("Hello " + towns[i])
}

for (const element of towns) {
   alert("Hello " + element)
}

towns.forEach((element) => alert("Hello " + element))
```

forEach is a higher-order function which takes another function (typically a lambda – i.e., an anonymous function) as its argument – the lambda is applied to each element of the target array ‘towns’.

# Operations on Arrays – forEach (2)

```
const towns = ['Auckland', 'Wellington', 'Dunedin', 'ChCh', 'Hamilton']
const greet = name => alert("Hello " + name)

for (let i = 0; i < towns.length; ++i) {
   greet(towns[i])
}

for (const element of towns) {
   greet(element)
}

towns.forEach(greet)
```

forEach is a higher-order function which takes another function (in this case a function expression, i.e., a pointer to a function) as its argument – the function is applied to each element of the target array ‘towns’.

# Operations on Arrays – filter

```
const towns = ['Auckland', 'Wellington', 'Dunedin', 'ChCh', 'Hamilton']
const greet = name => alert("Hello " + name)

const someTowns = towns.filter((element) => element.length == 8)
someTowns.forEach(greet)
```

filter is one another fundamental higher-order function.

# Operations on Arrays – filter (2)

```
const towns = ['Auckland', 'Wellington', 'Dunedin', 'ChCh', 'Hamilton']
const greet = name => alert("Hello " + name)

towns
   .filter((element) => element.length == 8)
   .forEach(greet)
```

filter and forEach are pipelined (often known as method chaining).

# Operations on Arrays – map

```
const towns = ['Auckland', 'Wellington', 'Dunedin', 'ChCh', 'Hamilton']
const addHello = name => ("Hello " + name)

towns.map(addHello).forEach((element) => alert(element))
```

map and forEach are pipelined here.

# More methods of Array

- The method `every` checks to see if every element in the array passes the test given by a function passed to `every`.
- The `push` method adds the given elements to the end of the array.
- The method `reduce` applies a reducer function on each element of the array, and returns a single value which is the result of the reduction.
- The `join` method concatenates all the elements of the array separated a user-supplied string (which by default is the comma).
- There are more – be familiar with them all.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

# JavaScript – Drag & Drop

mano@cs.auckland.ac.nz

# Drag & Drop

- We can indicate if an HTML element is draggable or not by setting the draggable property.
  - Images and links are draggable by default and the others are not draggable by default.
- Elements do not accept drops by default. To let them accept drops, we need to disable this default behaviour.
  - This is done by calling *preventDefault* on the dragover event handler.

# Drag & Drop

- The data we transfer and the action we carry out during drag & drop are (typically) controlled by the ondragstart and ondrop event handlers.
- Use the ondragstart event handler of the drag element to set the data we want to transfer from the drag element to the drop element.
  - The dataTransfer object of the event carries this data.
- Use the ondrop event handler of the drop element to get the data sent from the drag element.
  - Recall that the dataTransfer object of the event carries this data.