Week 2 - JavaScript and Some HTML and JSX

This week we are going to cover a fair bit of ground, beginning with some more advanced JavaScript. Later on we will revisit HTML, and show how it is used in a shorthand syntax called JSX, which allows us quickly to create UI elements under React. We will work with JSX below. For now, we will continue with JavaScript.

Intermediate JavaScript

In this section we will remain in the REPL console - we do not initially connect this material with Web Computing - but there are elements of the language which we need to know to do anything meaningful. Some of these exercises reinforce concepts already covered in the lectures, but in other cases we are breaking new ground, so it is important to work through the examples. As before, you can go to this link for direct access to a Node.JS repl. If there are issues loading this one – sometimes you may hit capacity limitations - try the Babel REPL.

Arrays

Similar to Lists in Python and arrays in C, C# and other languages, JavaScript arrays are used to store multiple values in a single variable. They are a fundamental data structure and their syntax and operation is very similar to C and to a lesser extent like C# and Java.

Using an array literal is the easiest way to create a JavaScript Array.

```
let array_name = [item1, item2, ...];
```

Spaces and line breaks are not important. A declaration can span multiple lines:

```
let animals = [
 "Lion",
 "Tiger",
 "Giraffe",
 "Elephant",
 "Monkey",
 "Lemur",
 "Rhinoceros"
];
```

The most important thing to remember about arrays in JavaScript is that they are zero-indexed: ***array indexes start with 0. [0] is the first element. [1] is the second element.***

This statement accesses the value of the first element in animals:

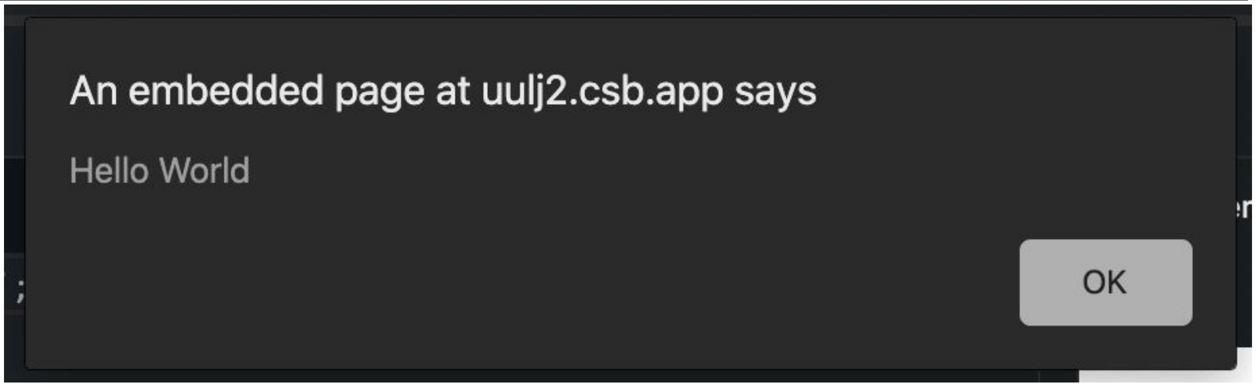
```
let firstAnimal = animals[0]; // Will be "Lion"
```

Arrays can contain basically any valid javascript expression such as strings, numbers, arrays, objects, even functions. Unlike in strongly typed languages such as C# and Java, this mixing of element types is legal in JavaScript:

```
const myWeirdArray = [
 "Text goes here",
 1,
 [0, 1, 2, 3],
 function myFunction() {
   alert('Hello World')
 }
];
```

As usual, all of these items work the same as they would in a normal variable if you use the expression to access that array item instead of the variable name. E.g. to call the above function `myFunction` refer to its position in the array followed by parentheses '()'.

```
myWeirdArray[3]()
```


An embedded page at uulj2.csb.app says

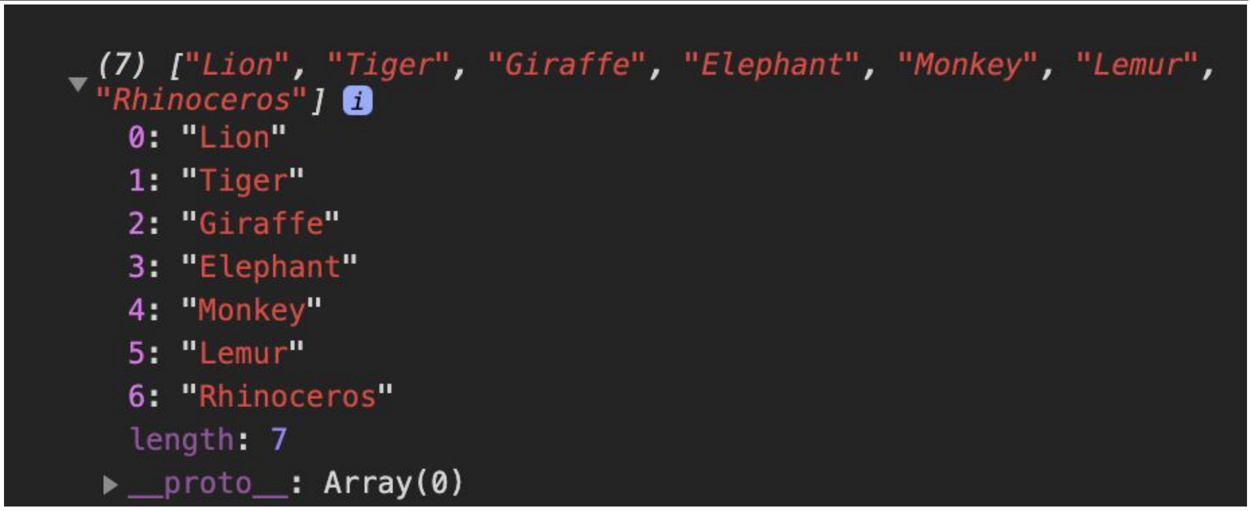
Hello World

Arrays in JavaScript also support many useful, standard properties and methods such as `length`, `includes()`, `push()`, `slice()` and so on. Many more can be found here. We will generally use these as we need to, explaining them in context, but most of them will be familiar from other languages.

console.log & console.table

We have implicitly worked with the `console` since last week, but here we talk about it a bit more. We use it mainly for simple testing purposes. Logging variables using `console.log()` shows the value of the variable at that point in your code, making it easy to ensure it is doing what it should:

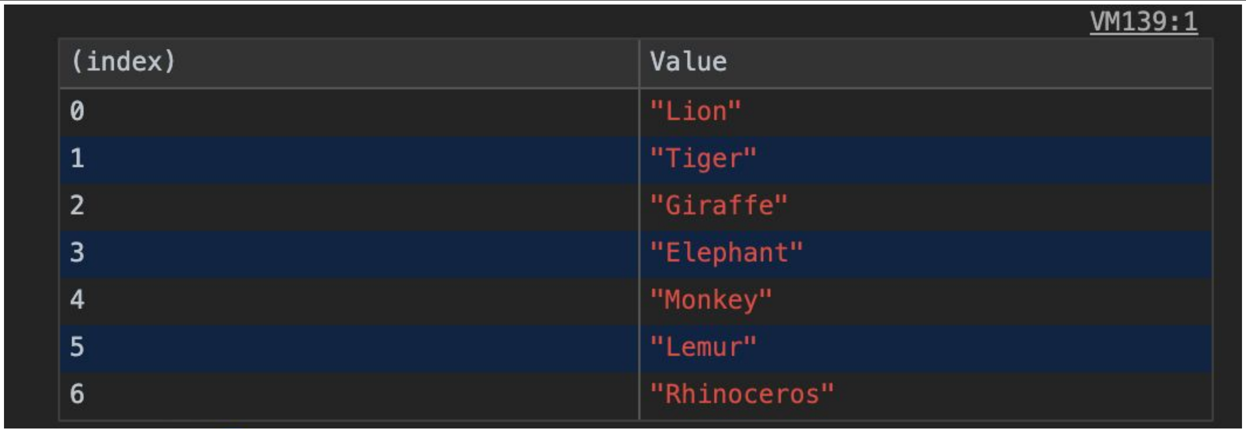
```
console.log(animals);
```



```
▼ (7) ["Lion", "Tiger", "Giraffe", "Elephant", "Monkey", "Lemur",
  "Rhinoceros"] i
    0: "Lion"
    1: "Tiger"
    2: "Giraffe"
    3: "Elephant"
    4: "Monkey"
    5: "Lemur"
    6: "Rhinoceros"
    length: 7
  ▶ __proto__: Array(0)
```

Console `table` outputs the variable in a table format in the console which can be a nicer way of viewing arrays and objects:

```
console.table(animals);
```



VM139:1

(index)	Value
0	"Lion"
1	"Tiger"
2	"Giraffe"
3	"Elephant"
4	"Monkey"
5	"Lemur"
6	"Rhinoceros"

Arrays are a special type of object. The typeof operator in JavaScript returns "object" for arrays.

```
console.log(typeof animals) // Prints "object"
```

But, JavaScript arrays are best described distinctly as they have different uses and syntax. In particular, we seldom use indexing styles when accessing objects.

Objects

JavaScript objects are generally defined in what is called 'JavaScript Object Notation' (JSON). JSON has become a standard format for transferring data which we will use a lot in later weeks. The basic JSON syntax is to have curly brackets (braces) '`{}`' surrounding the object's values. The values are written as key: value pairs (key and value separated by a colon).

```
const object_name = {
 key1: "Value1",
 key2: "Value2"
};
```

Arrays use index numbers to access their "elements" as shown in the examples above. Objects, however, use the key or property name to access their "properties" `(object_name.key1)`. In this example, `animal.name` returns Lion:

```
const animal = {
 name: "Lion",
 number: 3
};

console.log(animal.name) // Prints "Lion"
```

This is the preferred way of accessing object properties. You may encounter external data that has badly chosen keys containing spaces, periods or characters that this format doesn't allow. In these cases the more array-like syntax of `animal["name"]` is also valid.

Because each property has its own name, objects allow us to provide context around the elements of data. Arrays are better in situations where there is a list of similar elements. If you have multiple numbers, books, or students where the difference between each item doesn't need to be explained - each element of the list is very much of the same type - then an array should be used. Objects are useful for more varied data that explains more information about an entity or piece of data.

In JSON data, arrays and objects are frequently used inside each other to have arrays of objects and object properties that are arrays. This can be confusing at first glance, but pretend that you are a compiler and start matching up braces and square brackets and it will quickly become clear:

```
const animalData = [
 {
   name: "Lion",
   number: 3,
   eats: ["zebra", "antelope", "buffalo", "hippopotamus"]
 },
 {
   name: "Tiger",
   number: 5,
   eats: ["moose", "deer", "buffalo"]
 },
 //...
 {
   name: "Lemur",
   number: 15,
   eats: ["fruit", "leaves", "insects"]
 },
 {
   name: "Rhinoceros",
   number: 2,
   eats: ["grass", "shoots", "leaves", "berries"]
 }
];


console.log(animalData);
console.table(animalData);
```

Unsurprisingly, we are very keen to save JSON and to load it from external sources. When we want to pass it around in a convenient format, it is usual to convert a JSON object to a string, and to parse these strings to recover the JSON object. Fortunately JavaScript helps us out:

`JSON.stringify()` - The `JSON.stringify()` method converts a JavaScript object or value to a JSON string.

```
JSON.stringify(animalData)
```

```
'[{"name":"Lion","number":3,"eats":["zebra","antelope","buffalo","hippopotamus"]},{
"name":"Tiger","number":5,"eats":["moose","deer","buffalo"]},{"name":"Lemur","numbe
r":15,"eats":["fruit","leaves","insects"]},{"name":"Rhinoceros","number":2,"eats":[
"grass","shoots","leaves","berries"]}]'
```

JSON.parse()

The JSON.parse() method parses a JSON string into a JavaScript object. This can be necessary when receiving data from APIs and other sources.

In our final JavaScript topic we will look at a series of methods that sit on top of arrays, allowing us to process data in a functional style. To make this code more compact, we will often use the JavaScript arrow notation for the declaration of anonymous functions. The terms and the syntax seem challenging, but they are not that bad once you understand the purpose. Sometimes we want a function to do some work for us in a specific context - here we will use it to filter, select or transform the elements of an array. Such functions are normally trivial, and we don't want to use them again. In modern languages we can integrate the functions more closely with the code and not even bother to give them a name. These are what we call anonymous functions, and many languages have a special syntax for them. This is considered below.

Arrow and Anonymous functions

In the example below, we will show you different ways to write a function to double a number - so that if the value of num is passed as 4, the function will return a value of 8.

All of the following examples are valid ways to write this doubling function. In each case, on the left hand side we will provide a template format showing the style of declaration. On the right hand side, we will see the corresponding example.

So here is the basic function we are familiar with, it has a function keyword, the name of the function, parentheses surrounding any parameters and curly braces around the function body.

```
function name (parameters) {
  statements
}
```

```
function double (num) {
  return num * 2;
}
```

As you saw in week 1, we can also declare a function without a name, assigning it to an identifier as shown below. We won't consider scope here, but the rules still apply. Note the use of const rather than let - once we assign a function to a variable we seldom wish to change it to something else.

```
const name = function (parameters) {
  statements
}
```

```
const double = function (num) {
  return num * 2;
}
```

This formal way of defining an anonymous function is often used in parameter lists and elsewhere, but eventually someone found it tiresome and proposed a shorter syntax.

We now simplify our declaration to get an *arrow function*. This is done by removing the `function` keyword and instead putting an arrow like syntax `=>` in between the parameters and the function body. This works exactly the same as in the example above, but it is widely seen as a more convenient way to declare anonymous functions.

```
const name = (parameters) => {
  statements
}
```

```
const double = (num) => {
  return num * 2;
}
```

If it is a basic function that only needs one line for a return statement, we don't even need the curly braces or return keyword. We can write the same thing like this:

```
const name = (parameters) => expression
```

```
const double = (num) => num * 2;
```

Note: on the left 'statement' has been changed to 'expression' because it now has to be a single valid JavaScript expression that will be returned.

This syntax makes it really easy to write a simple compact function if all it needs to do is return an expression. This is particularly common in higher order functions like map, filter and reduce. It may be a very small amount of code and not look much like a function but it is still its own function.

Map, Filter, Reduce

Map, Filter and Reduce are useful array methods that are known as higher order functions. Higher-order functions are functions that take other functions as arguments or return functions as their results. The resulting code is said to adopt a functional style.

Map

The `map()` function creates a new array populated with the results of calling a provided function on every element in the calling array. The map function is used extensively in React to generate JSX from an array. Here we show a simple example where the anonymous arrow function from before is used to double every element in the specified array:

```
const array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1);
```

```
// expected output: Array [2, 8, 18, 32]
```

Filter

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Here is a simple example where each word in the array is tested to see if it has a length of greater than six characters.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

Reduce

The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. This may appear more confusing than the others. The second parameter of the reduce method is an initial value for the 'accumulator'.

In the example below, the initial value given is 5, and the accumulator becomes the additive total of the numbers in the array.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Introduction to React

We are now going to move on and connect some of our JavaScript knowledge with some work on the client side using React. We are going to start slowly, with a revision of HTML, but you will quickly see that tags that appear in HTML can also find their way into JSX and that is how construct a lot of the UI elements in React.

As we said in the lecture, we expect you to learn or revise HTML with limited assistance. It is a very simple syntax, and mostly it works very well. You don't have to memorise it, and there won't be a test, but if you can't use it you will struggle with some of the more complex material which relies on its use.

Pure HTML without styling is like a formatted text document, but there is a set of common tags which underpin most web pages. HTML is simple, but permissive. It is often trivial and seen as old hat, but it still provides the scaffolding for modern responsive web apps. So we will revise HTML and take our first steps towards React, learning to embed HTML in JS using JSX.

Revision of HTML

If you are new to HTML, then W3Schools has a decent introduction. Most importantly, their site allows you to work interactively with the tags and later with the page styling. You can edit their examples and see what happens. In the lecture, we showed you a simple HTML document, and you can work with that if you need to make some of this general discussion more concrete.

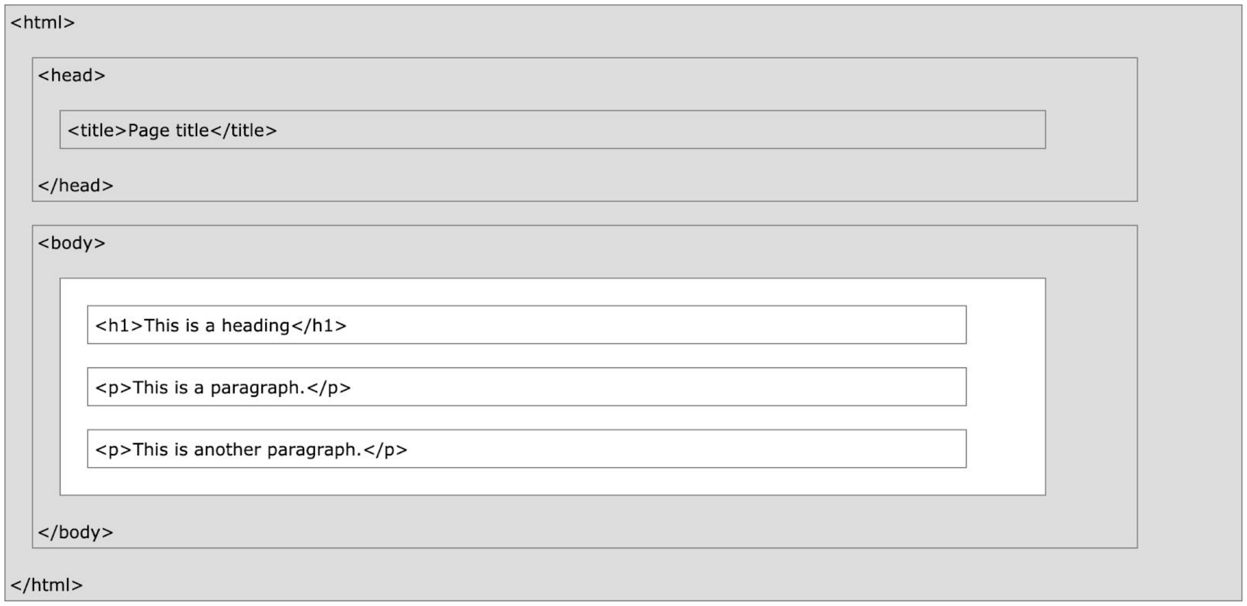
In HTML every tag has a corresponding closing tag, denoted by a forward slash. A few empty tags can be self-closing - for example images and text inputs don't need to wrap around other content. This is also the case for React components, which we will consider later.

In the example on the next page, we see the overall structure of an HTML document. Note that this one is *well-formed* - every opening tag is paired with its closing tag. Browsers probably should enforce this, but often don't. We *will*.

The surrounding `<html>` tags specify that this is an HTML document. The `<head>` tags typically contain things such as links to styles, scripts and meta information not viewed on the web page. Generally you should not link to large script files in the header as they take too long to load and are not immediately available to your users. Web pages have now been around since the early 1990s and there are a range of styling guidelines which we explore in later exercises. Those wishing to see some really bad examples can start here:

https://www.mockplus.com/blog/post/bad-web-design

In HTML documents, the content is placed in the `<body>` section - highlighted in the figure below. This is what is actually shown on the page and this is generally all you will need to worry about when we work with React, as the application itself will organise the content programmatically.


<html>
<head>
<title>Page title</title>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
</body>
</html>

You will see that the page is organised using headings and paragraphs, but you will also expect to see unordered lists `<ul>` and list items `<li>` and widespread use of `<div>` or *division* tags which help us to subdivide the document into regions that can be more easily styled or controlled using CSS or JavaScript.

As discussed at the lecture, these tags form an implicit tree structure, the so-called DOM or Document Object Model. This is available within the browser, and it can be manipulated programmatically using JavaScript. We will not look at this in detail here, but you should understand that React works with its own version of this structure. Just as we can add elements to the overall Document, so we can and will update the structure of the *React DOM*. We will see more of this below.

React

React was introduced briefly in the lecture, and we will look at it properly in the coming weeks. As you now know, React is a view creation library, built in JavaScript by the team at Facebook. If you have written websites using Express before, it is similar to Pug or EJS, but you don't need to have this background to understand it. React is designed to handle only the layout of your data - it does not have any strong opinions on how that data is retrieved, stored, or manipulated, though there are one or two caveats. Contrast this with a *framework* such as Angular, which *does* express strong opinions on the flow of your data through the application, on how you perform HTTP requests, on how your data is stored, and so on.

It is important to understand at the start that React is a view layer *only*. A 'presenter' before anything else. Here we will give you a hint of what is to come with some example JSX before covering a bit more background. We will then use some very basic JSX and React to get you started with the approach.

Some Background - JSX

JSX is the HTML-style syntax that React uses to declare elements. You can read a lot more about it here. It is the building block for every React application. JSX is an extension to JavaScript (hence JavaScripteXtension) that allows for a much more familiar and natural syntax to be used. It is not, however, magic.

Take the following element as an example:

```
const element = <p>Hello, World!</p>;
```

When rendered with ReactDOM (take a look at the lecture example if you are not sure of this) an almost identical element will appear in the browser. However, the actual function call is:

```
const element = React.createElement('p', null, 'Hello, World!');
```

You can see that the JSX is much more similar to the regular HTML you might write, and is just some Syntactic Sugar over the underlying function calls. We use it because it saves us work. In practice, you'll almost *never* need to use the createElement function directly.

Note: We do not have the time to cover it too deeply, but if you're curious about what a particular piece of JSX compiles to in JavaScript, you can look at the Babel REPL. Babel is the *de facto* compiler for JavaScript, and as before, the REPL is an online compiler. You can paste (or type) your code in and see what the output of the compiler is.

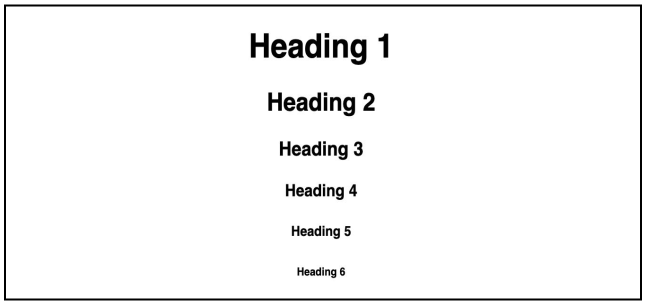
Commonly used HTML/JSX tags you will need to know

In the following text we will take you through some commonly used HTML tags,. The point here is that they are not just HTML tags, but as you will be starting to realise, they are also JSX tags that can be used to help construct React user interfaces. We will see this in practice below. If you know your HTML well then you can skip over this section pretty quickly, but if the material is new then you probably should take your time.

Heading tag `<h[1-6]>`

It is used to define the heading of html document.

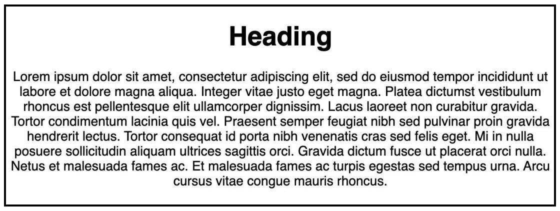
```
<h1>Heading 1 </h1>
<h2>Heading 2 </h2>
<h3>Heading 3 </h3>
<h4>Heading 4 </h4>
<h5>Heading 5 </h5>
<h6>Heading 6 </h6>
```



Paragraph tag `<p>`

It is used to define paragraph or basic text content in html document.

```html
<h1>Heading</h1>
<p>Lorem ipsum dolor sit amet ...</p>
```



Heading

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Integer vitae justo eget magna. Platea dictumst vestibulum rhoncus est pellentesque elit ullamcorper dignissim. Lacus laoreet non curabitur gravida. Tortor condimentum lacinia quis vel. Praesent semper feugiat nibh sed pulvinar proin gravida hendrerit lectus. Tortor consequat id porta nibh venenatis cras sed felis eget. Mi in nulla posuere sollicitudin aliquam ultrices sagittis orci. Gravida dictum fusce ut placerat orci nulla. Netus et malesuada fames ac. Et malesuada fames ac turpis egestas sed tempus urna. Arcu cursus vitae congue mauris rhoncus.

Image tag `<img />`

It is a self-closing tag used to include an image and sometimes some local styling, though this is not preferred.

It requires a src parameter that contains the url of where to find the image.

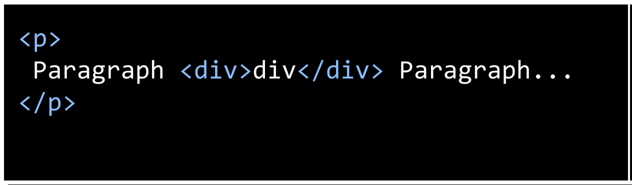
It also has an alt parameter that is used as a description of the image. This is important in cases where the image doesn't load as well as for screen readers and accessibility.

```html
<img
 src="https://images.pexels.com/photos/2220336/pexels-photo-2220336.jpeg"
 alt="Lion's Head"
/>
```

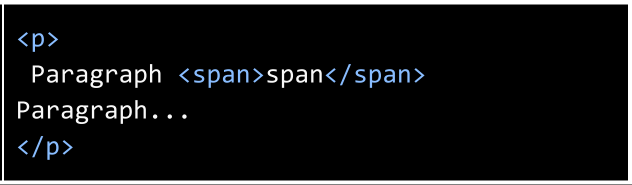
Division tag `<div>` & Span tag `<span>`

These tags are about dividing and grouping content to allow us to apply the same controlling code or styles. They have no intrinsic meaning in the way that a paragraph or a heading tag does.

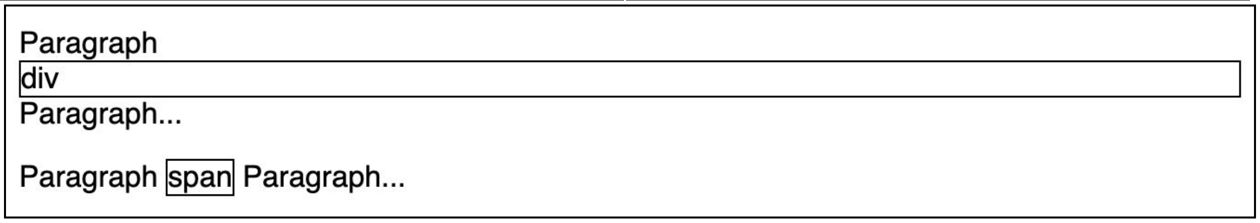
The key difference, at least initially, is that a div will put a box around the contents that by default is the maximum width of the page or enclosing element. A span, however, is an inline tag that does not naturally alter the page flow at all. In the examples below I have added borders around the elements to show the difference more easily.



```
<p>
 Paragraph <div>div</div> Paragraph...
</p>
```



```
<p>
 Paragraph <span>span</span>
Paragraph...
</p>
```



Paragraph
div
Paragraph...

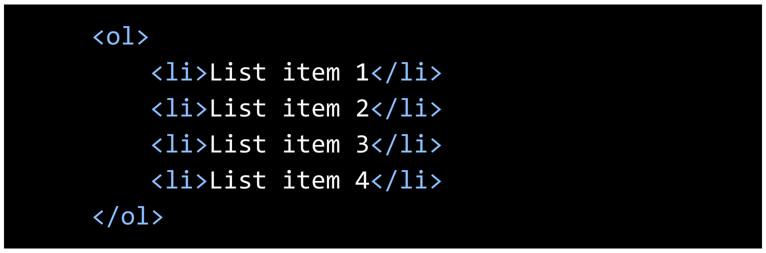
Paragraph span Paragraph...

Lists

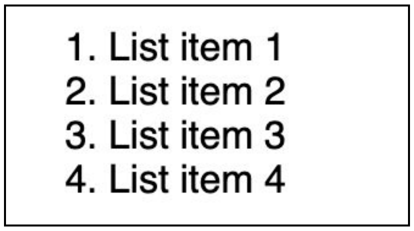
The unordered List tag `<ul>` is used to list the content without order (dot points).

The ordered List tag `<ol>` is used to list the content in a particular order (numbered list).

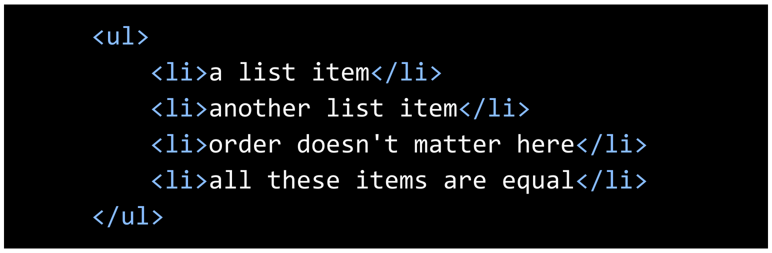
List Item `<li>` is used for each item in these lists.



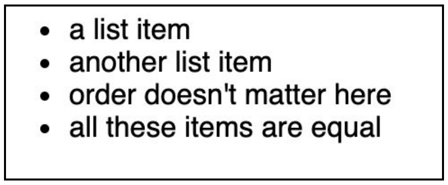
```
<ol>
    <li>List item 1</li>
    <li>List item 2</li>
    <li>List item 3</li>
    <li>List item 4</li>
</ol>
```



1. List item 1
2. List item 2
3. List item 3
4. List item 4



```
<ul>
    <li>a list item</li>
    <li>another list item</li>
    <li>order doesn't matter here</li>
    <li>all these items are equal</li>
</ul>
```



- a list item
- another list item
- order doesn't matter here
- all these items are equal

Icons `<i>`

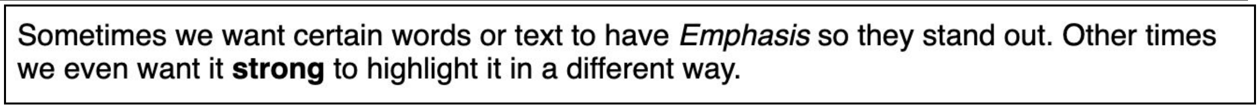
Technically an italics tag, the <i> tag has become the standard tag used to include icon fonts.

```
<i class="fab fa-js"></i>
```

Emphasis tag `<em>` & Strong tag `<strong>`

These tags are used to add emphasis in text, by default <em> shows text in italics and <strong> bolds text. Although the effects can be customised with styling.

```
<p>Sometimes we want certain words or text to have <em>Emphasis</em> so
they stand out. Other times we even want it <strong>strong</strong> to
highlight it in a different way.</p>
```


Sometimes we want certain words or text to have *Emphasis* so they stand out. Other times we even want it **strong** to highlight it in a different way.

Refer to W3Schools or other online resources to learn more about more specific tags as needed. For example for adding tables, meta information, or semantic tags for more meaningful HTML. W3Schools is an excellent resource to learn more or to refresh your knowledge.

Putting it all together

Now we are going to combine the JSX and the JavaScript we have been talking about. For the practicals involving React, we recommend using CodeSandbox. It is a completely free, completely open-source online editor for JavaScript in general. CodeSandbox allows you to create a bunch of different applications online, so you can test out Vue, Angular or even Reason.

Using CodeSandbox means that you don't have to install any new software on your personal machines, don't have to handle differences between macOS, Linux and Windows (a lot of packages on NPM, especially for front-end code, aren't very friendly to Windows), and allow you to take your application with you to other machines by logging in with GitHub. We also know that everyone is working on the same platform, so it makes it much easier for your tutor to help you debug any issues.. It also lets you experiment freely with a disposable sandbox if you want to try something out, and you can 'fork' your existing work if you want to try something new and not mess with your existing code. What's more, you can export a .zip file from CodeSandbox with everything required to run your application somewhere else.

It uses create-react-app under the hood to create its React applications, which is the *defacto* standard for creating new React applications. You can get a pre-prepared React application by going to this link, and can start hacking on it straight away with a live reloading setup.

We will post a document on BB for those who are eager to run things on their own machines (or for those interested) using create-react-app - we *won't* be covering Webpack or similar. Consider that an exercise for the reader.

As you go through this and later pracs, you'll see some **'Exercises'**. Make sure you try these out before moving on (where there will be solutions). You won't pick up much from this work if you just copy the code directly. We already know React, but you putting the code directly into your application without thinking about it first doesn't help *you* learn React.

I have created a starter kit here. The first thing you're going to want to do is *fork* the sandbox using the button in the top left of the page. If you're signed in to CodeSandbox, your changes will be associated with your account automatically. Otherwise, you can edit anonymously (but you can't keep your work - hold on to the link tightly). If you want to work on it on your own computer (and you've got your computer set up for it) you can just download the starter directly from CodeSandbox and unzip it on your machine.

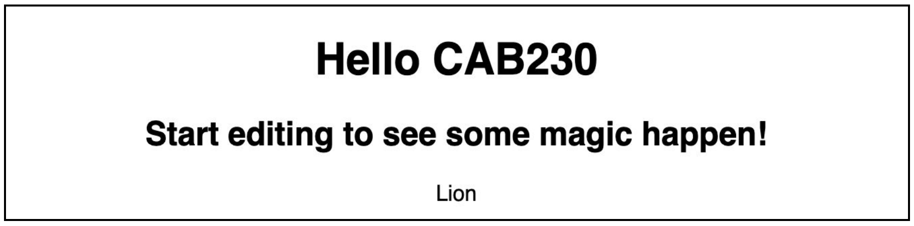
Initially you'll be greeted by a simple React application. The App function in App.js should look like this. Ignore the other files for now.

```
export default function App() {
 return (
   <div className="App">
     <h1>Hello CAB230</h1>
     <h2>Start editing to see some magic happen!</h2>
   </div>
 );
}
```

Initially, you will see a very simple message created by the Header tags in the app. Our first change is to add a paragraph after the second heading:

```
<p>Lion</p>
```

On the display, it should now have added the text 'Lion' underneath the heading. We have started editing and some very basic magic has happened.


Hello CAB230

Start editing to see some magic happen!

Lion

This may seem incredibly boring and it may appear pointless, but bear with me. So far we have added some very basic static content. In order to execute JavaScript code (like variables and expressions) in JSX we need to place curly braces `{}` around it.

We will begin by including a data source, the list of animals we considered earlier. This will allow us to work with all the animals. Above the App function, add our basic animals array from before.

```
const animals = [
 "Lion",
 "Tiger",
 "Giraffe",
 "Elephant",
 "Monkey",
 "Lemur",
 "Rhinoceros"
];
```

If we replace "Lion" with the code for the first element of the array, nothing much will change in the output, but we are already writing much more interesting code:

```
<p>{animals[0]}</p>
```

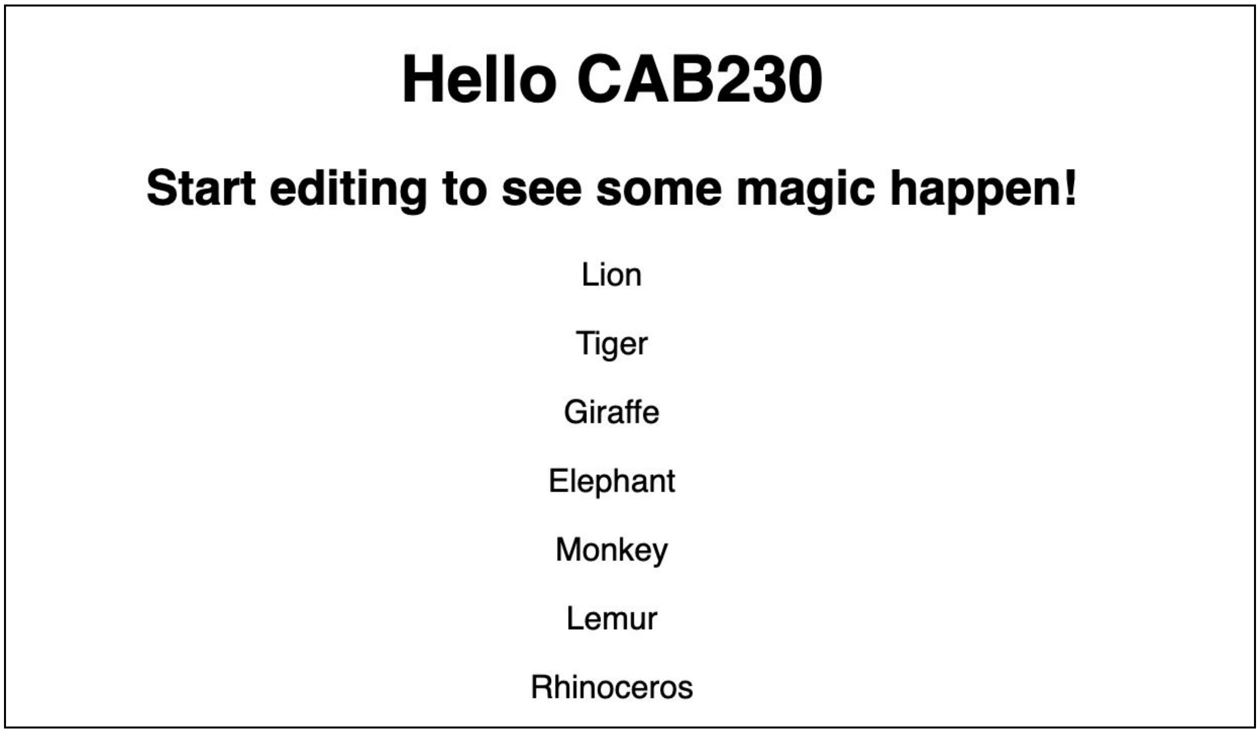
To list all the animals we need to work with a map function. We will write an anonymous function to transform each element using a tag. So for example, we will take the array element "Lion" and return something like `<p>Lion</p>`. So, instead of a simple reference like `<p>{animals[0]}</p>`, we will need something more like this:

```
     { animals.map(animal => (
       <p>{animal}</p>
     )) }
```

First, note the `=>` syntax that shows we have an anonymous arrow function. This function takes in each element of the array as 'animal' and returns the animal surrounded by paragraph tags.

Secondly, note that we wrap the element we've returned in parentheses. This is technically optional, but is very good practice to avoid the problems that can arise from Automatic Semicolon Insertion. You don't need to think about it too much - just remember to wrap your return statements in parentheses if they might span multiple lines. Just do it.

This should now be printing out all our animals.



Hello CAB230

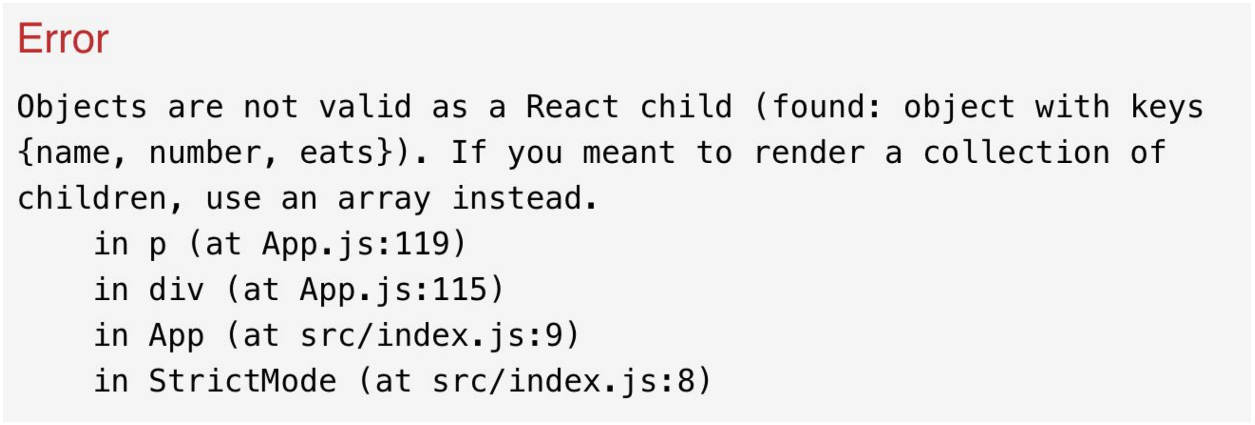
Start editing to see some magic happen!

```
Lion
Tiger
Giraffe
Elephant
Monkey
Lemur
Rhinoceros
```

Now let's add some more information about each of our animals. First we will need our object animal array instead of the basic one (shown on the following page).

```
const animals = [
 {
   name: "Lion",
   number: 3,
   eats: ["zebra", "antelope", "buffalo", "hippopotamus"]
 },
 {
   name: "Tiger",
   number: 5,
   eats: ["moose", "deer", "buffalo"]
 },
 {
   name: "Giraffe",
   number: 6,
   eats: ["leaves", "twigs"]
 },
 {
   name: "Elephant",
   number: 4,
   eats: ["grass", "leaves", "flowers", "fruit"]
 },
 {
   name: "Monkey",
   number: 10,
   eats: ["fruit", "leaves", "vegetables", "insects"]
 },
 {
   name: "Lemur",
   number: 15,
   eats: ["fruit", "leaves", "insects"]
 },
 {
   name: "Rhinoceros",
   number: 2,
   eats: ["grass", "shoots", "leaves", "berries"]
 }
];
```

Now once that is in, there will be an error (next page).

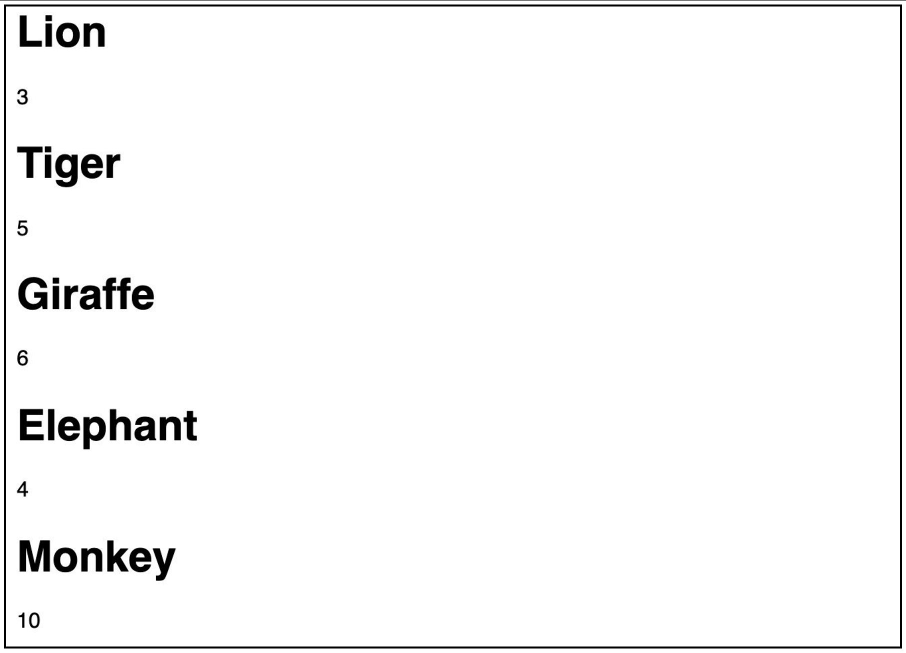


```
Error
Objects are not valid as a React child (found: object with keys
{name, number, eats}). If you meant to render a collection of
children, use an array instead.
    in p (at App.js:119)
    in div (at App.js:115)
    in App (at src/index.js:9)
    in StrictMode (at src/index.js:8)
```

The error is because each animal is now an object rather than a string and we can't just output an object in React like that. So to fix this we need to change our map to match our new animal objects. Let's just do something relatively basic for now and print out the name of the animal and its number.

Keep in mind that we can only return *one top level* element, so we'll need to wrap our text in something. For now, we'll use a div, because it doesn't *really* change what anything looks like.

```
     {animals.map(animal => (
       <div>
         <h1>{animal.name}</h1>
         <p>{animal.number}</p>
       </div>
     ))}
```



Lion

3

Tiger

5

Giraffe

6

Elephant

4

Monkey

10

Reusable Components

Despite how little information we are printing out, our map function is already starting to get a bit big and clunky already. It will get *much* worse if we extend it to more useful applications. With normal JavaScript code we would pull it into a new function and call it. We can do the same with JSX. This is called a reusable component.

A component is a nice, simple unit of code that is essentially independent from anything else. We isolate components by making a new function and including the relevant pieces within that function body.

First things first, let's make our component. As mentioned earlier, a component is just a function that returns some React code. App was our first component, and now we're going to make a second component called AnimalComponent.

To do this, create a new function above App that looks like this:

```
function AnimalComponent() {
 // our code will go in here
}
```

We need some way of passing in our object for the current animal. This is called a *prop* in React. A prop is something you pass into a React component. You can think about it almost like the parameter to a function (because it is!).

Every React component you create has all its props passed to it as an object in the first argument. You then access the fields of the prop in the usual way.

```
function AnimalComponent(props) {
 return (
   <div>
     <h1>{props.name}</h1>
     <p>{props.number}</p>
   </div>
 );
}
```

Now we want to change our map function to use our new component instead. The syntax for this uses the same syntax as our HTML and JSX tags. General practice is to use a capital letter at the start of components which differentiates them from actual html and normal functions.

The basic component looks like this (**Note: we still need a closing slash at the end):**

```
<AnimalComponent />
```

We need to pass in our animal data, this is done the same way as html attributes.

Note: remember to put curly braces around the javascript expressions.

```
{animals.map(animal => (
  <AnimalComponent name={animal.name} number={animal.number}/>
))}
```

Now we have the same output of animal information, but the resusible component keeps our App function simpler. In the remainder of this prac, we will show you a few tricks to make life a little simpler, at least once you master them.

JavaScript Shorthands

JavaScript is a very flexible language in many respects, as part of this, it contains many shorthands and alternate syntaxes for things. Here are a few that are commonly used in React.

Spread Syntax

Spread syntax (`...`) is a JavaScript notation that allows us to expand out an array or object in a way that is the equivalent of specifying them all individually. Essentially it saves a lot of clumsy typing.

More technically, we say that it allows an iterable type such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected. Similarly, it lets us expand an object expression in places where zero or more key-value pairs (for object literals) are expected. This is all much better understood in practice.

In our example from above, we can use this in our map function to pass in all the animal information easily.

```
{animals.map(animal => (
  <AnimalComponent {...animal} />
))}
```

This is equivalent to passing in all of the properties in our animal parameter individually, whatever they are.

Rest Parameter Syntax

The reverse of spreading, the rest parameter syntax (...) is used for collecting the rest of the parameters in a function.

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) {
     sum += arg;
  }

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

In this example, the args array will contain whatever is passed into the function. So for the subsequent calls it will equal `[1]`, `[1, 2]` and `[1, 2, 3]`.

Note: The rest parameters must be at the end

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error
}
```

The ...rest must always be last.

```
function f(arg1, arg2, ...rest) {
  // this is valid. Yay!
}
```

Destructuring assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const [a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // expected output: 10
console.log(b); // expected output: 20
console.log(rest); // expected output: Array [30,40,50]
```

This can be useful for our props back in the animal example.

```
function AnimalComponent(props) {
 let {name, number, eats} = props

 return (
   <div>
     <h1>{name}</h1>
     <p>{number}</p>
   </div>
 );
```

Now we can just use name, number and eats as variables without needing to specify props first. This syntax also works directly inside the function parentheses:

```
function AnimalComponent({name, number, eats}) {

 return (
   <div>
     <h1>{name}</h1>
     <p>{number}</p>
   </div>
 );
```

The same thing as an arrow function would be something like this:

```
const AnimalComponent = ({ name, number, eats }) => (
   <div>
     <h1>{name}</h1>
     <p>{number}</p>
   </div>
);
```

These sorts of things can be quite frequent in react code so it is good to recognise and understand them when they appear.

For bonus points, extend the AnimalComponent to display the eats array nicely.

Next week, we will look at some more substantial examples.