framework training
We love technology

React

John Coumbe

020 3137 3920 @FrameworkTrain
frameworktraining.co.uk

framework training
We love technology

React Framework

Section		Page
	Outline	1
A	Intro	3
B	Component	7
C	Modules	11
D	Props	13
E	State	15
F	Stateless	19
G	Basket	22
H	Form	25
I	Proptypes	30
J	Lifecycle	33
K	Keys	35
L	Router	38
M	Hocs	43
N	Redux	46
O	Appendix - JavaScript review	54

Framework Training
React
London July 2018
Course outline

OVERVIEW

- This course introduces the **React** Javascript framework from Facebook.
- The course is intended for web developers who have used **Javascript** or another programming language.

TOOLS, SETUP

- You will need admin rights on your machine.
- We will install the following tools during the course.
  - The Chrome **React DevTools**
  - **Atom** text editor.
  - **Node** and **NPM**
  - The **create-react-app** CLI tool

JAVASCRIPT REVIEW

- **ES5** fundamentals: variables, arrays, objects, functions.
- **Functional** programming: map, filter, reduce, forEach
- **Scope**: closure, global namespace, this, bind.
- **ES6**: arrow functions, constants, destructuring, classes, modules, template literals.

REACT

- Creating **Components**, the fundamental building blocks of a React application.
- Passing arguments into components using **props**.
- Managing **state** inside components.
- The advantages of **stateless** components.
- Using **map** and **keys** to iterate over arrays of data.
- Adding **prop-types** to component definitions.
- Building React apps as a **composition** of components.
- Using **Fetch** to read external **JSON** data.
- The **Container-Presentation** pattern
- Building **forms** with validation in React.
- Understanding the component **Life Cycle**.
- Creating a single page application with **React Router**.

- Developing **higher order** components (HOCS)
- Using the **Redux** state management pattern in React.

Framework Training
React
London July 2018
Exercise A-intro

- This exercise introduces **React components** which combine markup, logic, style and state.
- Components are a fundamental building block of the React framework.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

React components

- Components allow you to create custom HTML elements which combine markup, logic, style and state.
- This exercise builds a very simple React component.
- Review the starter files.
- **public/index.html** is an HTML file containing two empty sections.

```
  <section class="spain"></section>
  <section class="norway"></section>
```

- We will use React to create components and inject markup inside these sections.
- The entry point for our project is **index.js**. We will create instances of a React component here that are rendered in the HTML sections above.

Define a City component

- We will define a simple component which displays a message

```
Welcome to the city of Seville.
```

- The name of the city will be passed in to the component as an argument (prop).
- Open **src/City.js**
- Import the React library and its Component class.

```
import React, { Component } from 'react';
```

- Import the local stylesheet City.css

```
import "./City.css";
```

- A React component can be defined as an ES6 class.
- The class extends the React Component class.

```
class City extends Component {}
```

- Every component must have a **render** method.
- This method defines the view/UI of the component.
- The render method returns markup, written in **JSX**, the React HTML-like dialect.

```
render() {
    return <section>Hello</section>
    }
```

- We have now created a **definition** of a component.
- Export the component so that it is useable in other modules.

```
export default City;
```

Component instances

- To be visible on a web page, we need to create **instances** of this component.
- File **src/index.js** is the entry point for our React project where we will create component instances.
- React has been separated into two libraries.
- The core library is React.
- React-DOM handles rendering React code in the browser DOM.
- This separation allows React to also be deployed to browser-less server environments.

```
import React from "react";
import ReactDOM from "react-dom";
```

- We need to import the City component, and any local CSS.

```
import "./index.css";
import City from "./City";
```

- We can now use React-DOM to render one or more **instances** of the City component on the page.

```
ReactDOM.render(<City/>, document.querySelector(".spain"));
```

- Hello should now appear on the web page.

Component props

- We will extend the component to pass in the city name as an argument.
- To allow multiple lines of JSX in the return statement, we will wrap the code in parentheses.

```
    render() {
        return (
            <section>Hello</section>
        )
    }
```

- Change the markup to include a SPAN which can be styled.

```
<section>Welcome to the city of<span>name</span></section>
```

- We can pass a city name into the component instance as an attribute in **index.js**.

```
<City name="Seville"/>
```

- This argument is referred to as a **prop** in React.
- Inside the component **this.props** is in scope as an object containing the name/value pairs of the arguments passed in.

```
console.log(this.props);
// {name: "Seville"}
```

- We can use the expression **this.props.name** to display the city name.

```
<span>{ this.props.name }</span>
```

Multiple instances

- We can render **multiple instances** of a component.

```
let spain = document.querySelector(".spain");
let norway = document.querySelector(".norway");

ReactDOM.render(<City name="Seville" />, spain);
ReactDOM.render(<City name="Oslo" />, norway);
```

Different types of component syntax

- There are a number of alternative approaches to writing React components.
- The render method can be written as an **ES6 arrow** function.

```
    render = () => <section>Welcome to the city of<span>
{this.props.name}</span></section>
```

- *Later in the course we will review the scope advantages of using arrow functions.*

Stateless components

- The City component does not have any internal **state**.
- In this case we can rewrite the component more simple as a stateless function.

```
function City(props) {
    return (
        <section>Welcome to the city of<span>{ props.name }
</span></section>
    )
}
```

- Note two changes: we have to explicitly pass in props, and we refer to them without using this.
- We can refactor this component using ES6 arrow syntax.

```
let City = props => <section>Welcome to the city of<span>{
props.name }</span></section>
```

- All four variations of the component do the same thing.
- We will review use cases for when to use each approach later in the course.

Framework Training
React
London July 2018
Exercise B-component

- This exercise creates a component which uses **map** to **iterate** over an array of objects.

Installation

- Install the starter version of the project.
- Open the terminal at the correct folder.

```
npm install
npm start
```

Plain JS

- We can use plain Javascript to turn an array of strings into HTML markup. Reviewing this code will help gain an understanding of how React works.
- Review the code in **help/turn-array-into-markup.js**.

Create a React component

- **Cities.js** defines a module which exports an array of objects.

```
let cities = [
{ name: "Seville", temp: 88.. },
{ name: "Trujillo", temp: 64.. }
];
```

- We will create a Spain component to display this data, and then create an instance of the component in index.js.
- In **Spain.js**, import React and its component class.

```
import React from "react";
import { Component } from "react";
```

- Import the CSS and the data.

```
import "./App.css";
import { cities } from "./cities";
```

- To create a minimal React component, it must contain one render method.
- That method must return one top level HTML element.

```
class Spain extends Component {
    render() {
        return <section>Spain</section>;
    }
}

export default Spain;
```

- A component instance can be created in index.js.
- Import the component definition.

```
import Spain from "./Spain";
```

- The instance needs to be injected into a specific element in the DOM in public/index.html:

```
    <section class="page"></section>
```

- In index.js store a reference to that element into a variable.

```
    let el = document.querySelector(".page");
```

- Use React to render an instance of the Spain component into that element.

```
    ReactDOM.render(<Spain />, el);
```

- The component instance should appear on the page.
- Create a heading in the render method and style it. Note the React JSX syntax uses className to avoid the reserved word class.

```
return (
    <section className="holiday">
        <h1>Spain</h1>
    </section>
);
```

Iterating over the cities data using map()

- We can use the Javascript **map function** to iterate over the cities array and apply a function to each element.

```
    import { cities } from "./cities";
    cities.map(city => console.log(city.name, city.region));
```

- We can use map and JSX inside the render method to create markup.

```
{cities.map(city => <p className="city">{city.name}</p>)}
```

- This works but generates a React error. React wants unique key attributes on each iterated element.
- Add a key attribute using the ID property in each city.
- Add the region as a SPAN within the paragraph.

```
        {cities.map(city => (
            <p key={city.id} className="city">
                {city.name}
                <span>{city.region}</span>
            </p>
        ))}
```

- Add a population property to each object. Display the population and temperature for each city

Review

- The Spain component is not **pure**: it relies on the external local variable cities being in scope.
- We have not defined any way of passing data into the component, so that it can be reused with different cities.

Stateless version

- The Spain component does not have any internal state, so it could be defined as a stateless ES6 arrow function.

```
let Spain = () => {

    return (
        <section className="holiday">
            <h1>Spain</h1>

            {cities.map(city => (
                <p key={city.id} className="city">
                    {city.name}
                    <span>{city.region}</span>
                </p>
            ))}
        </section>
    )
```

```
};
```

Framework Training
React
London July 2018
Exercise C-modules

- This exercise separates a project into **ES6 modules**.
- Install and run the starter project.

```
npm install
npm start
```

Review the project

- All the components and data are held in one file, **index.js**.
- This exercise will refactor the code into separate files using ES6 modules.

Data.js

- Move the array of objects into its own file data.js.
- Export the array named books.

```
var books = [....]
export { books }
```

- Import the data into index.js

```
import { books } from './data';
```

Book.js

- Create file book.js for the book component.
- Import the React library

```
import React from 'react'
let Book = ....

export { Book };
```

Bookshelf.js

- Create file bookshelf.js for the BookShelf component.

- It contains instances of Book and so needs to import it

```
import React from 'react'
import { Book } from "./book";
let BookShelf = ....
export { BookShelf };
```

index.js

- Import Bookshelf and the data.

```
import { BookShelf } from './bookshelf';
import { books } from './data';

import './style.css';

render( <BookShelf shelf={books}
/>,document.getElementById("root"))
```

- Test that the new code organisation works.

Framework Training
React
London July 2018
Exercise D-props

- This exercise turns the Spain component into a more reusable Nation component.
- Data is passed into **multiple instances** of the Nation component as **React props**.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the starter version of the project

- Data for Spanish and Japanese cities is imported into the main index.js file.

```
import { es } from "./cities/spain";
import { jp } from "./cities/japan";
```

- public/index.html contains two sections where the component instances will be injected.

```
<section class="spain"></section>
<section class="japan"></section>
```

- Create two React component instances in index.js

```
let spain = document.querySelector(".spain");
let japan = document.querySelector(".japan");

ReactDOM.render( <Nation/> , spain );
ReactDOM.render( <Nation/> , japan );
```

- We can pass the data into the component instances as props.

```
ReactDOM.render( <Nation country={es}/> , spain );
ReactDOM.render( <Nation country={jp}/> , japan );
```

- In the Nation component, props are passed in via the constructor.

```
    constructor( props ) {
    super( props );
    console.log( this.props.country );
    }
```

- We can display the country name in the render method:

```
<h1>{ this.props.country.name }</h1>
```

- We can use destructuring to write more concise syntax:

```
let { name,cities } = this.props.country;
<h1>{ name }</h1>
```

- Remove the React comments to re-enable the map code:

```
{ cities.map .. }
```

- View the component instances in the React DevTools.

Stateless components

- Nation can be redefined as a stateless ES6 arrow function.

```
let Nation = ( props ) => {
    let { name,cities } = props.country;
    return ( <section> .. )
}
```

Framework Training
React
London July 2018
Exercise E-state

- This exercise adds **state** to components.
- Install and run the starter project.

```
npm install
npm start
```

Review the current state of the project

- The Shop component render method maps over the basket props to draw up four panels within a FlexBox.

```
{this.props.basket.map((name, n) => {
    return (
        <section className="panel" key={n}>
            <h2>0</h2>
            <h4>{name}</h4>
            <p>Up</p>
            <p>Down</p>
        </section>
    );
})}
```

- We can use **destructuring** to write more concise syntax.

```
let {basket} = this.props;
    {basket.map((item, n) ....}
```

Component composition

- We can use **composition** by moving each item into its own Panel component.

```
<section className="shop">
    {basket.map((name,n) => <Panel key={n} desc={name} /> )}
</section>
```

- Note that **Panel is a stateless component**: a single function, not a class with methods.

```
    let Panel = (props) => {

        let {desc} = props;

        return (
            <section className="panel">
                <h2>0</h2>
                <h4>{desc}</h4>
                <p>Up</p>
                <p>Down</p>
            </section>
        )
    }
```

Generating a random key

- We should use true unique keys, not just a sequence of numbers for the key attribute.
- Add a getKey method which creates a unique key from the item name and a random number.

```
  getKey( s ) {
      return s + "-" + Math.floor( Math.random() * 1024 * 1024
);
  }
```

- Use this method in each Panel instance.

```
<Panel key={this.getKey(name)} desc={name} />
```

Add state to the Panel component.

- We want to add state to the Panel component.
- Clicking the UP or DOWN buttons should change the number displayed.
- The component is currently a stateless function.
- Convert it to a component class with methods.

```
class Panel extends Component {
    render() {
        let {desc} = this.props;
        return (
```

```
            <section className="panel">..</section>
        )
    }
}
```

- Add a constructor method.

```
constructor( props ) {
    super( props );
    console.log( this.props );
}
```

Component state

- State can be only defined in the **constructor**.
- It can be changed indirectly using **setState** in other methods.
- State is defined as an object.
- It is then visible in all methods as **this.state**.

```
constructor( props ) {
super( props );
this.state = { total:0 };
```

}

- We can define an UP method to increase the total.

```
up() {
    let n = this.state.total + 1;
    this.setState({ total: n });
}
```

- Define a DOWN method to decrease the total and avoid minus numbers.

```
  down() {
    let n = Math.max(this.state.total - 1, 0);
    this.setState({ total: n });
  }
```

- Add event-based code to the render method to call these methods when the user clicks UP or DOWN.

```
<p onClick={this.up}>Up</p>
```

Down

- Clicking UP or DOWN causes a runtime error.

- Javascript changes the runtime value of THIS to undefined.
- Expressions like this.state.total cause a run-time error.
- One solution is to **explicitly bind** the run-time value of THIS in the constructor.

```
this.up = this.up.bind( this );
this.down = this.down.bind( this );
```

- To see changes in state, we need to update the render method.

```
        <h2>{ this.state.total }</h2>
```

- We can make this more concise with **destructuring**

```
let {total} = this.state;
<h2>{total}</h2>
```

Framework Training
React
London July 2018
Exercise F-stateless

- This exercise works with **stateless** components.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the starter version of the project

- Stateless components are just one single function.
- They take in data as **props**.
- They return a view as markup.
- A stateless component has no other methods.
- It has no **state**.
- Its simplicity makes it reliable and easily testable.
- Keyword **this** is not available within a stateless component.

Interate over the books using map

- Iterate over the books array and pass each book down to a separate Book component.

```
let BookShelf = props =>

    <section>
        <h2>Man Booker Prize Winners</h2>
        { props.books.map((b,n) => <Book key={n} book={b} /> )}
    </section>

let Book = props =>

    <section className="book">
        <h4>{ props.book.author }</h4>
        <p>{ props.book.title }</p>
```

```
    </section>
```

Destructuring in the Book component

- We can add **destructuring** in the Book component.
- Once the arrow function contains multiple statements, we need to add brackets and an explicit return statement.

```
let Book = props => {

    let {author,title} = props.book;

    return (
        <section className="book">
            <h4>{author}</h4>
            <p>{title}</p>
        </section>
    )

}
```

Spread operator

- We can use the ES6 spread operator in the parent component.
- This will pass down each property of the book object (title,author) to the Book component as individual props.

```
<Book key={n} {...b} />
```

- This is equivalent to writing

```
<Book key={n} author={b.author} title={b.title} />
```

- By passing down both author and title as props, we can simplify the Book component.
- The two components have become one-line ES6 arrow functions.

```
import React from 'react';

let BookShelf = props =>

    <section><h2>Booker Prize</h2>{ props.books.map((b,n) =>
<Book key={n} {...b} /> )}</section>
```

```
let Book = ({author,title}) =>

    <section className="book"><h4>{author}</h4><p>{title}</p>
</section>

export default BookShelf;
```

Framework Training
React
London July 2018
Exercise G-basket

- This exercise passes **functions** to components as props. This allows components to invoke functions in their parents.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the starter version of the project

- The fruitVeg array is passed into the main component Shop.

```
<Shop fruit={fruitVeg} />
```

- The Shop component render method uses map to iterate over the array.

```
let {fruit} = this.props;
{fruit.map((item,n) => <Item key={n} name={item} /> )}
```

Add items to a basket

- We want to select items and add them to a basket.
- Define an empty basket array as component **state** in the **constructor**.

```
        this.state = { basket:[] };
```

- We can define a buyItem method in the Shop component and pass this down as a prop to the Item component.

```
        buyItem = e => console.log(e);

        <Item key={n} name={item} select={this.buyItem}/>
```

- Inside the Item component, we can add an event handler which points at the select prop.

```
        onClick={props.select}
```

- Clicking on an item will log a React **synthetic event** to the browser console.
- Expression e.target points at the DOM element clicked on.

```
buyItem = e => console.log(e.target);
```

- Store the name of the selected item to a variable.

```
let item = e.target.textContent;
```

- We want to add this name to the basket, but we should not attempt to directly change state.
- This code makes a copy of the basket array using the spread operator and adds in the new item.

```
let item = e.target.textContent;
let copy = [ ...this.state.basket, item ];
```

- We can sort the array of item names.

```
copy = copy.sort( (a,b) => a > b);
```

- Use setState to update the basket.

```
this.setState( { basket:copy } )
```

- Look at the basket state in the React DevTools.

Display the state basket

- Add another FlexBox in the Shop component render method to iterate over the basket array.

```
let {basket} = this.state;

<section className="shelf">
        {basket.map((item,n) => <Item key={n} name={item}/> )}
</section>
```

- This creates a problem. A React render method should return only one top-level element.
- Restructure the JSX: wrap both FlexBoxes in a containing section.
- *Note: both the fruit and basket arrays share the same Item component.*

Conditional styling

- To style the basket items differently, we could create a new component, or use conditional styling.
- This example uses conditional styling.
- Add a type prop to both instances of the Item component

```
<Item .. type="shelf">
<Item .. type="basket">
```

- Use that prop to apply conditional styling.

```
className={ props.type === "basket" ? "basket" : null }
```

Remove items from the basket

- We want to be able to remove items from the basket.
- Define a new method in the main component.

```
removeItem = e => console.log( e.target.textContent );
```

- Pass that function down as a prop.

```
<Item .... select={this.removeItem} />
```

- RemoteItem needs to make a copy of the basket

```
let copy = [ ...this.state.basket ];
```

- Then search the basket for the first element with this name

```
let position = copy.findIndex( (f) => f=== name );
```

- Remove that element from the copy.

```
copy.splice( position,1 );
```

- Use setState to update the state basket.

```
this.setState( { basket : copy } );
```

- Test this: clicking an item in the basket should remove it.

Framework Training
React
London July 2018
Exercise H-form

- This exercise builds a form with validation in React.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the existing project

- The Form component defines a form without any validation.
- We want the state of the form to reflect the component state.
- The first time the form is displayed, it should use values from the component state.
- When the user types into a field, an event handler should capture this value and update component state.
- When component state changes, the render method is called.
- It should update the form to reflect state.
- This approach to keeping form and component state in sync is called a **Controlled Component**.

Validation

- The component will apply **validation** to the form.
- The city field will allow **two or more letters, but no digits**.
- The passport field will only allow exactly **eight digits**.
- **Warning feedback** will be displayed adjacent to each field.
- Field contents will be **styled red** when a field is incorrect.
- **Regular expressions** will be used to style the field.

Form state

- Define state in the constructor for the two form fields.

```
    constructor( props ) {
```

```
super( props );
this.state = { city:"", passport:"" };
}
```

- Listen for change events on the city field as the user types.

```
<input type="text" name="city"
onChange={this.changeField} />

changeField = e => console.log(e);
```

- React passes a **synthetic event** to the function changeField.
- The expression **e.target** points at the form field.
- We can use this to update state for the city field.

```
changeField = e => {
    let el = e.target;
    this.setState( { city:el.value })
}
```

- Use React DevTools to confirm that this changes state.

More generic event handlers

- This approach works but we want to avoid writing separate event handlers for each field.
- Expression e.target.name contains the name of each form field.
- We can refactor the event handler to use this expression.

```
this.setState( { [el.name]:el.value })
```

- We can then call the same method from the passport field.

```
<input type="text" name="passport"
 onChange={this.changeField} />
```

- Use React DevTools to confirm that this changes state for both fields.

Form values

- We also want to ensure that form fields and state remain in sync, and we may want to initially set form values in the constructor.
- Set the form field values equal to their state.

```
.... value={city}
.... value={passport}
```

Validation

- We will use regular expressions to validate the fields.
- This expression will return true if the city contains two or more letters and no digits.

```
/^[a-zA-Z]{2,}$/.test( city )
```

- This expression returns true if the passport is exactly eight digits.

```
/^[0-9]{8}$/.test( passport )
```

- This validate function creates an object that contains the current validation state of the form.

```
    validate = () => {

        let {city,passport} = this.state;

        return {
            city : {
                test : /^[a-zA-Z]{2,}$/.test( city )
            } ,
            passport: {
                test : /^[0-9]{8}$/.test( passport )
            }
        }
    }
```

- Call the function in the render method.

```
    let valid = this.validate();
    console.log(valid);
```

- Once the form is valid, this object will contain:

```
{ city:{ valid:true }, passport: { valid:true }}
```

Conditional styling

- We can conditionally style form fields using the valid object.

```
className={ valid.city.test ? null : "form-error" }
className={ valid.passport.test ? null : "form-error" }
```

Error spans

- Adjacent to each field is a span containing an error warning message if the field is incorrect.
- We can add warning messages to our validate method.

```
    city : {
    test : /^[a-zA-Z]{2,}$/.test( city ),
    warn : "Only letters"
} ,
passport: {
    test : /^[0-9]{8}$/.test( passport ),
    warn : "8 digits"
}
```

- We can use conditional styling to hide/reveal these messages based on the state of the form.

```
    <span className=
    { valid.city.test ? null : "form-warning" }>{
valid.city.warn }</span>

    <span className=
    { valid.passport.test ? null : "form-warning" }>{
valid.passport.warn }</span>
```

- Test the behaviour of the span messages.

Submit the form

- We need an event handler that is called when the user submits the form.
- The default behaviour of web forms is to refresh the web page on submit. The first task is to turn off this behaviour.

```
<form id="holiday" onSubmit={this.buyHoliday}>

buyHoliday = (e) => {
    e.preventDefault();
    console.log( this.state );
}
```

- This works but the submit button is active all the time, even when the form contains invalid fields.
- We need a boolean function which returns true if every field is valid.

```
  isValid = () => {
    let valid = this.validate();
```

```
    return Object.keys(valid).every( j => valid[j].test );
}
```

- We can use this function to control the submit button state.

```
disabled={!this.isValid()}
```

- Once the form has been submitted, we want to clear the form.

```
this.setState( { city:"", passport:"" } );
```

- We have now created a Controlled component which manages the state of a form.

Framework Training
React
London July 2018
Exercise I-prop-types

- We pass arguments into React components using **props**.
- We can **validate** these props to ensure that component instances are used correctly.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the existing project

- The project displays two instances of a Nutrition component.

```
    <Nutrition name="broccoli" calories={28}
    protein={87} type="cruciferous"/>
    <Nutrition name="cucumber" calories={42}
    protein={56} type="marrow" />
```

- **PropTypes** deal with one common source of bugs: using components with the wrong/missing props.
- The **propTypes** property defines the correct type for each prop.
- The **defaultProps** property defines default values for these props.
- *These properties can be defined using two styles of syntax. This exercise uses static properties inside the class.*

PropTypes

- We import the **propTypes** React library.

```
import PropTypes from 'prop-types';
```

- We define propTypes as a **static** property within the class.

```
static propTypes = {}
```

- We can define the name prop as a string.

```
static propTypes = {
name : PropTypes.string
```

}

- If we pass a number-expression to the component instance this will trigger a run-time error.

```
<Nutrition name={45} .... />
Invalid prop `name` of type `number`
```

- Passing a correct string value will fix this.

```
<Nutrition name="broccoli" .. />
```

- We can make name a **compulsory** prop by adding **isRequired**.

```
name : PropTypes.string.isRequired
```

- Add a propType for the calories field.

```
calories : PropTypes.number
```

PropType functions

- We can perform **custom validation** by defining a function within the propTypes object.
- This function checks that the protein prop is in the range 0–100.

```
protein : (props, name) => {
    let n = Number( props[name]);
    return (n >=0 && n <=100) ? null : new Error("Protein range
0-100")
},
```

Range of allow numbers

- We can define an array of acceptable prop values for the type prop.

```
type: PropTypes.oneOf(
['leafy', 'cruciferous','root','marrow','allium'])
```

Default props

- We can define **default values** for component instances.

- Component instances can be created with certain props omitted.

```
    static defaultProps = {
        calories: 50,
        protein: 10
    }
```

- We can then create a second instance which uses the default values.

```
        <Nutrition name="cucumber" type="marrow" />
```

Framework Training
React
London July 2018
Exercise J-lifecycle

- This exercise reviews a working example to understand how the React Component **Life Cycle** works.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the existing project

- **Index.js** file contains a Demo component instance.
- Its **render** method contains an instance of a Life component.

```
    <Life json={file}/>
<p onClick={this.changeData}>Change data</p>
```

- The **prop** named json contains a filename “spain–2017.json”
- Clicking on the adjacent button changes this filename.

The LIFE component life-cycle

- The first method to run is the **constructor**.
- It sets the component state.

```
this.state = { data: [], letters:[] }
```

- The **render** method is called for the first time.
- Once the component has been rendered on stage, the **componentDidMount** method is called.
- It calls getJson() to load data using Fetch.

```
this.getJson("componentDidMount");
```

- When Fetch returns data, it updates state:

```
this.getJson("componentDidMount");
```

- When the state changes, the **render** method is called.
- The **componentDidUpdate** method is also called when the component updates.

Change data

- If the user clicks the **Change Data** button, it flips the order of this array of filenames in the state of the Demo component

```
json:[ "spain-2017.json","spain-2018.json" ]
```

- When the state changes its render method is called again.

```
<Life json={file}/>
```

- Because the value of the prop has changed, the **componentDidUpdate** method in the Life component is called.
- It compares old versus new props. If they have changed it calls getJson to load new JSON data.

```
if(prevProps.json !== this.props.json) {
    this.getJson("componentDidUpdate");
}
```

- **getJson** loads the data and changes component state, which triggers another call to **render**.
- Review the Life Cycle using the browser console and React DevTools.

Framework Training
React
London July 2018
Exercise K-keys

- **Keys** are used in React when iterating over arrays of data using map.
- This exercise examines potential problems with using keys.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the project.

- This project creates a TO DO list.
- The list is held in component **state** as an array of objects.

```
[ { desc:"Large apples"}, { desc:"Small pears"} ]
```

- When the user clicks the **add-item** button, a new object is pushed into the **end** of the array.
- This works but this is a **run-time error** in the browser console.

```
Each child in an array or iterator should have a unique "key"
prop.
```

- React uses **keys** to identify changes in groups of related items.
- Here we map over the list. The error occurs because we need to give each list item (LI) a **unique key**.

```
<ul>{ props.list.map((item,n) =>
    <li><label>{item.desc}</label>
    <input type="text"/></li> )}
</ul>
```

- Map passes in two arguments: the item and an index. We can use this index as the key.

```
    <li key={n}>
```

- This approach clears the error message, but using this index value is **problematic**.

Problems with the item index as a key

- To demonstrate the problem, we will change the code to insert new items at the **start** of the list in method addItem.

```
list.unshift( this.createItem());
```

- Test this new version.
- Add some items, then edit the first input field.
- When you then add further items, the first input field **falls out of sync** with its associated label.
- We are using index keys that are numbered from 0 to N.
- When we insert a new item at the front of the list, we replace element with a key of 0 with another element with a key of 0.
- At this point React fails to reorder the elements correctly.

Unique keys

- We can avoid bugs with keys by using **unique** non-repeating IDs for each item.
- Add a method which creates unique keys based on the item desc.
- This method uses a regular expression to create a four letter code from the description.
- So “Organic large English plums” becomes “olep”

```
    createId = desc =>
desc.match(/\b(\w)/g).join("").toLowerCase()
```

- This will only create 256 different keys so their is a chance of duplicates.
- We can add a random number to the key to make duplicates very rare.

```
        createId = desc =>
desc.match(/\b(\w)/g).join("").toLowerCase() + "-" +
Math.floor(Math.random()*1000000)
```

- This will create keys like **“lsfp–564512”**
- Update **createItem** to use this method.

```
    createItem = () => {
        let desc = this.props.getFruit();
        let id = this.createId(desc);
        return { desc:desc, id:id }
    }
```

- Use this unique key in the **render** method.

```
    key={item.id}
```

- Test and confirm that this solves the bug.
- Review the code in React DevTools.

Framework Training
React
London July 2018
Exercise L-router

- This exercise uses **React Router** to manage navigation in a Single Page React Application.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

- The React Router was installed after creating the project with create-react-app: **npm install react-router-dom**

BrowserRouter

- The **BrowserRouter** component manages routing and maps changes of url to changes of component on the page.
- The **Route** component will render a component on the page when the current URL matches a defined path.
- Here, the Team component will appear on the page when the url is “/team”

```
<Route path="/team" component={Team}
```

- The **Switch** component is wrapped around multiple Routes, matching only the first suitable route.
- In the Hike component render method, create four routes:

```
<BrowserRouter>
    <section>
        <NavBar/>
        <Switch>
            <Route path='/team'     component={Team}/>
            <Route path='/contact'  component={Contact}/>
            <Route path='/packs' component={Packs}/>
            <Route path='/admin' component={Admin}/>
        </Switch>
```

```
    </section>
</BrowserRouter>
```

- Add a **home path** at the start of the Switch construct.
- Note this requires an **exact attribute** to prevent it matching routes like “/team” which contain “/”.

```
<Route exact={true} path='/' component={Home}/>
```

- Add an **error route** (which has no defined path) to catch any other route.

```
<Route component={Error}/>
```

Link components

- The navigation bar needs to be updated to use React Router Link components rather than HTML link elements.
- This will prevent that page reloading when the user navigates around the application.

```
    <li><Link to='/'>Home</Link></li>
    <li><Link to='/team'>Team</Link></li>
    <li><Link to='/contact'>Contact</Link></li>
    <li><Link to='/packs'>Packs</Link></li>
    <li><Link to='/admin'>Admin</Link></li>
```

Route component render attributes

- The Route component can define a render attribute. This contains an **inline function** that executes instead of instantiating a name Component.
- This code defines a “/version” route which renders version information in a header.

```
<Route path="/version" render= { () => <h2>v14.78</h2> }/>
```

Private Routes

- We want to **limit access** to the “/admin” route which opens an Admin-Tools page.
- We can define a render attribute which checks some boolean value.
- If true, the Admin component is rendered.
- If false, the Router redirects the user back to the home page.
- In the browser web tools define a **localStorage** property called admin.

```
localStorage.admin = 1
```

- Note, localStorage stores strings. To test for this value use an expression like:

```
Number( localStorage.admin ) === 1
```

- Define a new method in the Hike component.

```
  getAuth() {
  return (Number( localStorage.admin ) === 1);
  }
```

- In the Hike component render method, define an object which will be used by the Router Redirect component.

```
let goHome = { pathname: '/',state: { from: this.props.location
}} ;
```

- Define a Route which uses getAuth() to decide whether to render the Admin component or redirect the user back to the home page.

```
<Route
    path='/admin'
    render= { () => this.getAuth() ? <Admin/> :
    <Redirect to={goHome}/> }
/>
```

- Test that this functionality works.

Style the ADMIN link

- The Admin link can be **conditionally styled** to show if access is available.
- The NavBar component needs to use the getAuth() method defined in the Hike component.
- Pass this method down as a prop.

```
<NavBar getAuth={this.getAuth}/>
```

- Conditionally add the “no-admin” CSS class to the Admin link if getAuth returns false.

```
<Link to='/admin'
className={props.getAuth() ? null : "no-admin"} >
```

Packs component

- Packs.js defines an array of objects containing the backpacks that are available in the shop.

- The Packs component should read this array and render a submenu for each pack found:

```
// { code:"2806", desc:"Yellow" },
// <Link to="/packs/2806">Yellow</Link>
```

- We need to pass the pack data into the Packs component as a prop. *Note, the Router also passes additional props into the Pack component.*
- The Route component does not allow this, but we can achieve this using the render attribute.

```
<Route
exact path='/packs'
render={ props =>
<Packs packs={MountainShed} {...props}/>} />
```

- Inside the Packs component, the data is now in scope.

```
let {packs} = props;
```

- We can map over the packs array to create a NavBar local to the Packs page.

```
{ packs.map( (p,n) =>
<li key={n}>
<Link to={"/packs/"+p.code}>{p.desc}</Link>
</li> )}
```

Pack routes

- Clicking on the Yellow link in the Packs page generates a 404 error for route **localhost:4016/packs/2806**.
- We need to define a variable route in the Hike component.

```
    <Route exact path='/packs/:code'
    render={ props =>
    <Packs packs={MountainShed} {...props}/>}
    />
```

- This variable route will be passed down to the Packs component in **props.match.params.code**

```
    console.log( props.match.params.code );
```

- We can search the array of packs for the matching pack code.

```
packs.filter( p => p.code === props.match.params.code )
```

- This will return an array of one object if the pack is found.
- Invalid codes will return an empty array.
- Store this first pack to a variable.
- The variable will be undefined for invalid codes.

```
let pack = packs.filter( p => p.code === props.match.params.code
)[0];

console.log(pack); // e.g. {code: "4765", desc: "Orange"}
```

- Using the pack variable, we can conditionally render a matching image with the correct ALT attribute.

```
<section>
{ pack ? <img src={"../images/" + pack.code + ".png"} alt=
{pack.desc} /> : null }
```

- Test that the packs page works.

Framework Training
React
London July 2018
Exercise M-hocs

- This exercise creates a **Higher Order** component that works with different data and views.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Higher Order functions

- Higher Order functions (HOFS) return another function not just a value.
- double is an ordinary function which doubles numbers

```
let double = n => n*2;
```

- mult is a HOF which returns another function that multiplies numbers by some parameter.

```
let mult = a => b => a*b
```

- mult(2) returns a function that doubles numbers.

```
let double = mult(2)
double(2);
```

  let quad = mult(4)

```
quad(2)
```

- HOFs can be immediately invoked

```
mult(2)(2);
```

Review the existing project

- Component DataComponent reads a local JSON file using the Fetch API.
- Its render method contains an instance of the stateless view component called Regions.

```
render() { return <Regions data={this.state.data} /> }
```

- An instance of DataComponent is created in index.js.

```
<DataComponent json="spain-2017.json" />
```

- We can change DataComponent into a Higher Order component which will work with different views and data.
- CreateDataComponent is a function which takes one JSON url as its argument.
- It returns another function which takes a View component as its argument.

```
let CreateDataComponent = jsonData => ViewComponent => (
    class DataComponent extends Component { .. }
)
```

- Change the render method. ViewComponent will be passed in as an argument.

```
render() { return <ViewComponent data={this.state.data} />;}
```

- Invoking CreateDataComponent return a new component that reads a specific JSON file.

```
let Spain2017 = CreateDataComponent("spain-2017.json");
```

- We can pass a View component to Spain2017 to create a new component that displays the JSON data in a specific view.

```
let Spain2017Regions = Spain2017(Regions);
```

- We can create an instance of this new component in index.js

```
import { Spain2017Regions } from "./Hofs";
render(<Spain2017Regions />, document.querySelector("#a"));
```

Exercises

- Create a new stateless view that displays temperature information.

```
const Temps = ({ data:spain }) => (
  <ul className="flex">
    {spain.map((city,n) => (
      <li key={n}>{city.name}<span>{city.temp}</span></li>
```

```
    ))}
  </ul>
)
```

- Create a new component using the same 2017 data with this view.

```
let Spain2017 = CreateDataComponent("spain-2017.json");

let Spain2017Regions = Spain2017(Regions);
let Spain2017Temps = Spain2017(Temps);

export { Spain2017Regions,Spain2017Temps }
```

- Export and import the new component names into index.js

```
import {
    Spain2017Regions,
    Spain2017Temps
} from "./Hofs";

render(<Spain2017Regions />, document.querySelector("#a"));
render(<Spain2017Temps />, document.querySelector("#b"));
```

- Create component instances that work with the 2018 data for both views using immediately-invoked syntax.

```
let Spain2018Regions = CreateDataComponent("spain-2018.json")
(Regions);

let Spain2018Temps =
CreateDataComponent("spain-2018.json")(Temps);
```

Framework Training
React
London July 2018
Exercise N-redux

- This exercise uses the **Redux** state management pattern.
- Install and run the starter project.

```
npm install
npm start
```

- This project was built with create-react-app.
- Then the **redux** and **react-redux** packages were installed with npm.

Redux

- **Redux is a software design pattern** which aims to isolate the state of your application into one separate **store**.
- The store object lives separate from the hierarchy of React components.
- The components communicate indirectly with the store.
- This approach avoids the complexity of passing information between nested components using props.
- Redux defines a **store** where the application state is held. Our application will hold two things, the name of the airport and the star rating of the hotel.

```
{
airport : "gatwick",
hotel : "four",
}
```

- When components want to notify the store of a change, they **dispatch actions**.
- An action contains a type and some data.
- Only two types of action are needed in this example.

```
{type:AIRPORT, data:"Gatwick"}
{type:HOTEL, data:"four"}
```

- The type properties are strings defined as constants.

```
const AIRPORT = "airport" ;
const HOTEL = "hotel" ;
```

- When an action is dispatched, it is sent to a **reducer** function.
- The reducer is a custom function that you write which describes how the store should change when a specific action happens.
- The reducer does not directly change the store.
- It returns a new updated copy of the state, which the store updates.
- Whenever the store updates itself, it then notifies any **subscribers** that it has changed.

React-Redux

- **React-Redux** is a library which implements the Redux pattern in React.
- The starter version of this project has installed these libraries:

```
    npm install redux react-redux
```

Implementing Redux

- Create a **redux folder** in the src folder.
- Define two constants which are the types of action that will be dispatched in new file **redux/actions.js**

```
const AIRPORT    = "airport" ;
const HOTEL      = "hotel" ;

export { AIRPORT, HOTEL } ;
```

- Define a minimal reducer function in new file **redux/reducer.js**

```
const DEFAULT_STATE = {
    airport : "gatwick",
    hotel : "four",
}

export let reducer = (state=DEFAULT_STATE, action) => {
    switch (action.type) {
        default:
            return state;
    }
}
```

- Create a file **redux/store.js** which defines the Redux store.

```
import { createStore } from 'redux';
import { reducer } from "./reducer";
```

```
let store = createStore( reducer );

export { store }
```

Redux web tools

- Install the **Redux WebTools** as a Chrome browser extension
- This will aid debugging of the React Redux app.

```
https://chrome.google.com/webstore/detail/redux-
devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
```

- We need to add a 2nd parameter to the createStore function in redux/store.js in order to communicate with the Redux WebTools.

```
let store = createStore(reducer,
window.__REDUX_DEVTOOLS_EXTENSION__());
```

Test Redux

- In src/index.js add code to test dispatching an action to the store.
- Import the store and the AIRPORT constant.

```
import { store } from './redux/store';
import { AIRPORT } from './redux/actions';
```

- Dispatch a test action to the store.
- This code is wrapped in a two second time delay.

```
window.setTimeout( () => store.dispatch( { type:AIRPORT,
data:"heathrow"}) , 2000 );
```

- Open the Redux-WebTools to confirm if this works.

Dispatching actions from components

- We want to adapt our Airport and Hotel components to communicate with the Redux store.
- They need to **send** it **actions**.
- They need to **receive notification** when the store has changed.
- We will define two functions.
- **mapDispatchToProps** is a function which defines an event handler that dispatches actions.
- **mapStateToProps** is a function that contains the parts of the store object that a component needs to know about.

- React-Redux includes a **connect** function.
- This higher-order function takes mapDispatchToProps and mapStateToProps as its two arguments.
- It returns a new function.
- We then invoke the new function passing it the Airport or Hotel component as an argument.
- It then returns a new enhanced component which can communicate with the Redux store via its props.
- Open **components/airport.js**
- Import the connect function and the AIRPORT constant.

```
import { connect } from 'react-redux';
import { AIRPORT } from '../redux/actions';
```

- Define a **mapStateToProps** function

```
function mapStateToProps( state ) {
    return {
        airport : state.airport
    }
}
```

- Define a **mapDispatchToProps** function.
- This creates a new event-handler which dispatches actions.

```
function mapDispatchToProps( dispatch ) {
    return {
        setAirport( a ) {
            dispatch( { type:AIRPORT, data:a } )
        }
    }
}
```

- Wire up the two radio buttons to call this event-handler when they are selected.

```
onChange={ () => setAirport('Gatwick')}
onChange={ () => setAirport('Heathrow')}
```

- The function setAirport is passed into this component as a prop. Use **destructuring** to gain access to it.

```
let {setAirport} = this.props;
```

- Create a new component using **connect**.

```
let AirportR = connect( mapStateToProps, mapDispatchToProps ) (
Airport ) ;

export { AirportR }
```

- In index.js change the import statement and instance.

```
import { AirportR } from './components/airport';
<AirportR />
```

- In its current state, the run-time error occurs.
- To pass the Redux store between all components implicitly the top-level view is wrapped in a **Provider**.

```
import { Provider } from "react-redux";

<Provider store={store}> .. </Provider>
```

- Test the application: clicking the Gatwick or Heathrow radio buttons should dispatch actions to the store.

Add custom logic to the reducer

- Add a case statement to the reducer.
- This describes how the store should be updated when an AIRPORT action is dispatched.
- Note the reducer is a pure function. It does not directly change the store.
- Instead it makes a new copy of the state object, and modifies the copy.
- Redux will then update the store with this copy.
- This code shows two approaches using Object.assign and the spread operator to make a separate object.

```
        case AIRPORT:
        return { ...state , airport: action.data };
        // return Object.assign( {} , state, { airport:
action.data });
```

Hotel component

- Implement the Hotel component using the same approach.
- Here are some code fragments which need to be added to multiple files.

```
import { connect } from 'react-redux';
import { AIRPORT } from '../redux/actions';
```

```
function mapStateToProps() {}
function mapDispatchToProps() {}

onChange={ () => setHotel('four')}
onChange={ () => setHotel('five')}

let {setHotel} = this.props;

let HotelR = connect( mapStateToProps, mapDispatchToProps ) (
HotelR ) ;

export { HotelR }

import { HotelR } from './components/hotel';
<HotelR />

case HOTEL:
return { ...state , hotel: action.data };
// return Object.assign( {} , state, { hotel: action.data });
```

Holiday component

- The Holiday component should display a sentence which updates as the state changes.
- Use connect to adapt the component to talk to the Redux store.

```
function mapStateToProps( state ) {
    return {
        hotel : state.hotel,
        airport : state.airport
    }
}

function mapDispatchToProps( dispatch ) {
    return {}
}

let HolidayR = connect( mapStateToProps, mapDispatchToProps ) (
Holiday ) ;

export { HolidayR }
```

- Note this component does not dispatch any actions.
- The state is passed in as props to the view.

```
    let { hotel, airport } = this.props;
    <p>{ hotel } { airport }</p>
```

- Update the new HolidayR name in trips.js

```
import { HotelR } from './components/hotel';
<HolidayR/>
```

Review the application in Redux WebTools

- Note that changes of state cause new props to be sent to the HolidayR component.
- Add a lifecycle method in this component to log this.

```
componentWillReceiveProps( nextProps ) {
    console.log( "componentWillReceiveProps" ,
nextProps,"*",this.props );
}
```

Reset button

- Create a new Reset component which dispatches a RESET action and sets the store back to its default values.
- Define and export a new constant in **actions.js.**

```
const RESET = "reset" ;
```

- Add a new case to the **reducer**.

```
case RESET:
return DEFAULT_STATE;
```

- Create a new component which consists of a button that calls a reset method.

```
class Reset extends Component {
  render() {
      let {reset} = this.props;
      return (
          <section className="button" onClick=
{reset}>Reset</section>
      )
  }
```

```
}
```

- Add functions to dispatch a RESET action to the store.

```
    function mapStateToProps( state ) {
        return {}
    }

    function mapDispatchToProps( dispatch ) {
        return {
            reset() {
                dispatch( { type:RESET } )
            }
        }
    }

    let ResetR = connect( mapStateToProps, mapDispatchToProps )
( Reset ) ;

    export { ResetR }
```

- Import the new component into index.js and create an instance.

```
import { ResetR } from './components/reset';
<ResetR/>
```

- Clicking the reset button should restore the default state of the store and reset the UI in the Airport/Hotel components.
- The Redux pattern means that we can restructure our components and the application continues to work. Move ResetR inside Trips to test this concept.

Variables

- **Case sensitive**.
- **Uninitialised** variables are undefined.

```
var year = 2018;
var city = "Seville"
var smoker = false;
var town; // undefined
var project = null;
```

- **Weakly typed**: the type is not defined and can change.
- Typescript allows enforcing of strict types

```
var town = "Sandwell";
town = -45;
```

- Variables are globally scoped if defined outside a function, or object.

```
window.city === city; // true if global variable
```

Using let in ES6.

- Variables declared with let use **block scope**.
- Here variable j only exists within the for loop.

```
for( let j=0; j<10; j++) { console.log(j); }
```

- Variables can only be defined once with the current scope;

```
let city = "Oslo";
let city = "Copenhagen"; // run-time error
```

Equality

- Loose equality returns true if two values are the same but not of the same type.

```
2+2 == "4" ; // true
```

- Strict equality requires two values to be the same value and the same type

```
2+2 === "4"; // false
```

Truthy, falsy

- Javascript uses a loose boolean concept.
- Almost every expression evaluates to true except for a short list of values:

```
undefined, null, "", false, 0, NaN
```

Arrays

- Arrays are zero-indexed lists.
- Typically, the items in the list are of the same type.

```
let capitals = [];
capitals.push( "Paris" );
capitals.push( "Madrid" );

// Move from back to front
let last = capitals.pop();
capitals.unshift( last );

// Move from front to back
let first = capitals.shift();
capitals.push( first )
console.log( capitals );
```

- We can iterate over an array using **forEach**.
- Define a function that displays one capital.

```
function show( city ) { console.log( city ); }
```

- Call that function for every item in the array.

```
capitals.forEach( show );
```

Objects

- Objects define structured data in a self-documenting way.

```
let fred = { age:64, name:"Fred Smith" };
fred.job = "Postman";
fred.holiday = { city:"Paris", year:2017 };
```

```
fred.smoker = false;
delete fred.smoker;
```

- We can create arrays of objects.

```
let people = [ fred,jane ];
console.table( people );
```

- We can iterate over an array using forEach

```
function getAge( p ) { console.log( p.age ); }
people.forEach( getAge );
```

- **JSON** is the string representation of objects.

```
let s = JSON.stringify( fred );
// "{"age":64,"name":"Fred Smith","job":"Postman","holiday":
{"city":"Paris","year":2017}}"

let ob = JSON.parse( s );
```

Copy by reference/value.

- Numbers and strings are primitive values. Assignment will create new independent copies.

```
let cityA = "Lisbon";
let cityB = cityA;
cityB = "Madrid";
console.log( cityA, cityB ); // "Lisbon","Madrid"
```

- Arrays and objects are complex. Assignment will create two pointers to the same value.

```
let personA = { name:"Bert", age:54 };
let personB = Object.assign( {} , personA );

let lottery = [ 4,5,6,7,8 ];
let lotto = Object.assign( [] , lottery );
```

Functions

- Functions define local scope.
- Functions are hoisted to the top of their containing scope.

```
function double( n ) {
```

```
        let result = n*2; // local variable
        return result;
}
```

ES6 arrow functions

- ES6 introduces arrow functions.
- They omit this syntax: function, (), {}, return
- Here double is the function name.
- “n” is the argument passed in.
- “n*2” is the return value.

```
let double = n => n*2 ;
```

- If we pass in 2 arguments, we need additional parentheses.

```
let calcArea = (a,b) => a*b ;
```

- A function with NO arguments needs paretheses

```
let getYear = () => "2016";
```

- To return an object, wrap {} in ()

```
var createCity = ( c,n ) => ( {city:c, nation:n })
```

Functional Javascript

- Functional JS uses forEach, map, filter and reduce to transform arrays of data.
- **forEach** runs a function for each item in an array but does not create a new array.

```
let lotto = [ 4,10,20,40,45 ];

function show( n ) { console.log( n ) }

lotto.forEach( show );
```

- **map** runs a function on each item in an array and creates/returns a new array.

```
function double( n ) { return n*2 }

let newLotto = lotto.map( double );
```

- **filter** runs a boolean function against each item in an array and returns a new filtered array

```
function getBig( n ) { return n > 20 }

let bigLotto = lotto.filter( getBig );
```

- **reduce** applies a function to adjacent pairs in an array and returns a single value.

```
function add( a,b, ) { return a+b }

let total = lotto.reduce( add );
```

- Functional techniques can be combined with ES6 arrow functions.

```
let double = n => n*2;
let newLotto = lotto.map( double );
```

ES6 constants

- ES6 constants cannot be re-assigned.

```
const YEAR = 2018;
YEAR = 2017; // run-time error
```

- Note, the properties of complex constants can be changed.

```
const HOLIDAY = { city:"Paris",year:2014 };
HOLIDAY.year++; // This works and changes year.
```

ES6 Destructuring

- ES6 destructuring allows variables to be created from complex objects.

```
let { city, year } = { city:"Paris",year:2014 };
```

- Variables can be assigned new names.

```
let { city:c, year:y } = { city:"Paris",year:2014 };
console.log( c,y );
```

ES6 classes

- ES6 introduces class syntax.
- Methods do not need to be prefixed with the word function.
- The **constructor** function is called at instantiation.
- Classes can use inheritance.

```
class Rectangle {

    constructor(length, width) {
        console.log( "Rectangle" );
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

var rect = new Rectangle(6,4);
console.log( rect.getArea());
```

ES6 modules

- ES6 allows us to define modules: separate .JS files which have their own scope.
- Everything with a module is privately scoped unless it is explicitly made visible with the **export** keyword.

```
// utils.js
let halve = n => n/2; // private
export let double = n => n*2; // public
```

- Exported functions and variables can be imported into other files

```
import { double } from "./utils.js" ;
double(2);
```

- This feature is not available in all browsers: **https://caniuse.com/#feat=es6-module**
- To use this code, add the **type="module"** attribute to the script tag:

```
<script type="module" src="utils.js"></script>
```

- An alternative approach uses **Webpack** to bundle and transpile ES6 module code back to standard ES5.

ES6 template strings

- Multiple line strings can be defined using the back-tick character.

```
let city = "Oxford";
let markup = `<section>${ city }</section>`
```