Week 7

Reactive Programming

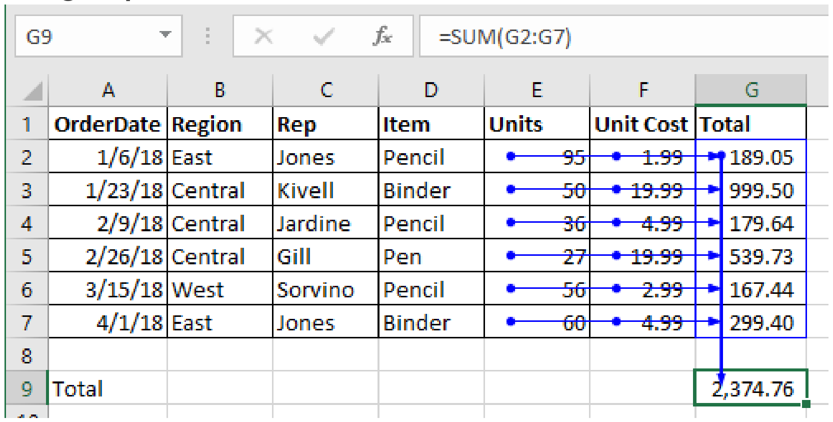
Overview

- Reactive Programming Theory
  - Functional reactive programming
- Reactive extensions (Rx or ReactiveX) http://reactivex.io/
  - Rx for .NET then RxJava, RxJS, …
- UI frameworks
  - https://reactjs.org/ (View Library UI for Javascript)
    - Redux library often used in React
    - React Native for Mobile apps
  - https://reactiveui.net/ (functional reactive model-view-viewmodel framework for .NET)

Reactive Programming Theory

- https://en.wikipedia.org/wiki/Reactive_programming
- Declarative programming paradigm
- *Something happens –> something else gets automatically updated*
- A graph of dependencies - change events are propagated along edges in the graph.
- Programming with asynchronous data streams

- E.g. Spreadsheet:



G9 =SUM(G2:G7)

	A	B	C	D	E	F	G
1	OrderDate	Region	Rep	Item	Units	Unit Cost	Total
2	1/6/18	East	Jones	Pencil	95	1.99	189.05
3	1/23/18	Central	Kivell	Binder	50	19.99	999.50
4	2/9/18	Central	Jardine	Pencil	36	4.99	179.64
5	2/26/18	Central	Gill	Pen	27	19.99	539.73
6	3/15/18	West	Sorvino	Pencil	56	2.99	167.44
7	4/1/18	East	Jones	Binder	60	4.99	299.40
8							
9	Total						2,374.76

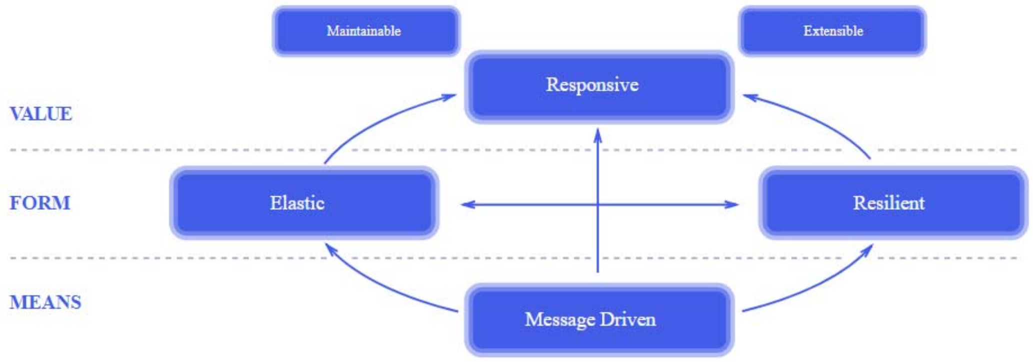
```
var a = 42;
var b = a + 1;
a++;
```

Functional Reactive Programming

- https://en.wikipedia.org/wiki/Functional_reactive_programming
- *“Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce and filter)”.*

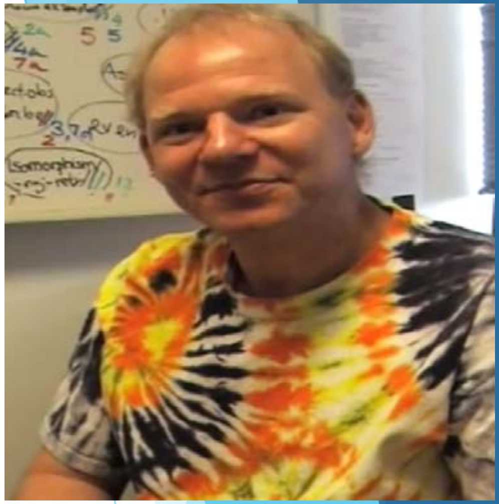
The Reactive Manifesto

- https://www.reactivemanifesto.org/
- Responsive – system responds in a timely manner
- Resilient – stay responsive in the face of failure
- Elastic – continues to work reliably even as the workload increases.
- Message Driven – asynchronous message passing to ensure loose coupling

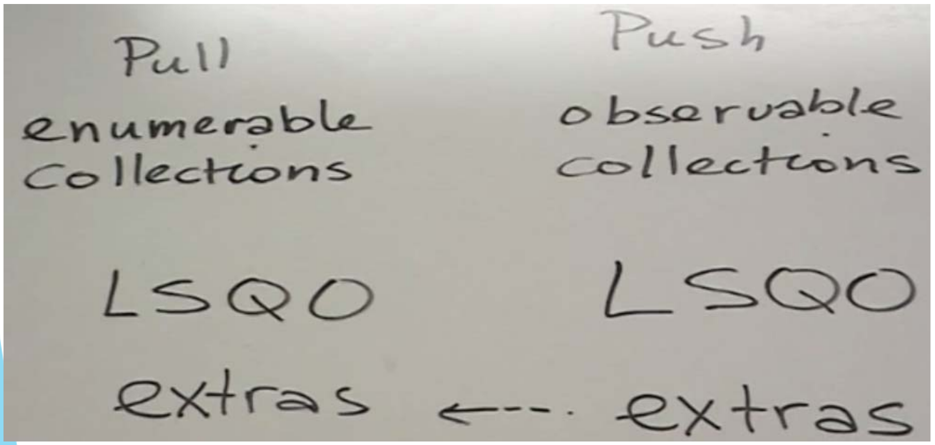


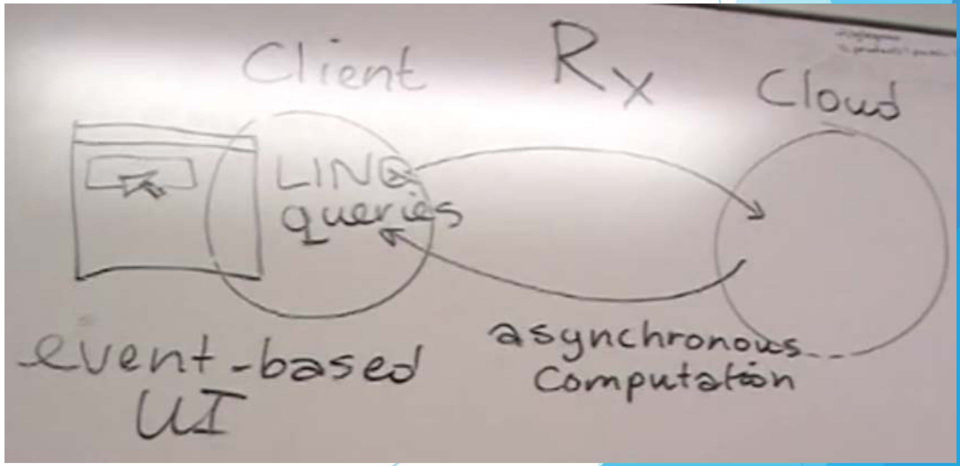
Reactive Extensions for .NET (Rx)

https://channel9.msdn.com/Blogs/Charles/Erik-Meijer-Rx-in-15-Minutes



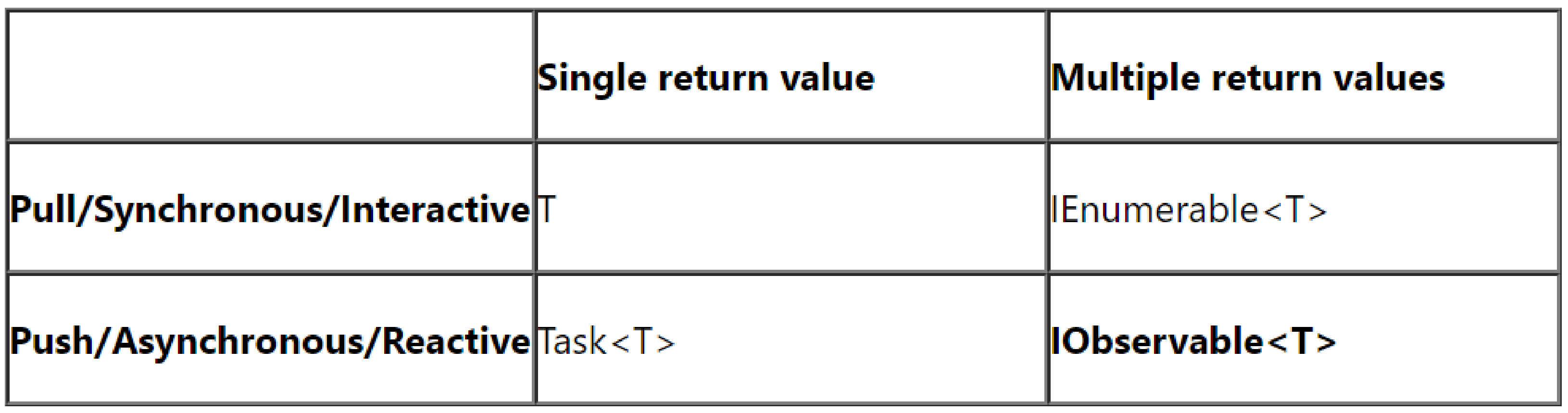
- (IEnumerable, IEnumerator) = "*pull based*" *Collections*
  - *Consumer pulls values from the collection as they are needed.*
- Rx (IObservable, IObserver) = "*push based*" *Collections*
  - *Producer pushes values at the consumer as they become available*
  - *E.g. event streams, asynchronous computations (or even normal IEnumerable collections)*





Rx.NET

- The Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.
- Because observable sequences are data streams, you can query them using standard LINQ query operators implemented by the Observable extension methods.
- Thus you can filter, project, aggregate, compose and perform time-based operations on multiple events easily by using these standard LINQ operators.



	Single return value	**Multiple return values**
Pull/Synchronous/Interactive	T	IEnumerable<T>
Push/Asynchronous/Reactive	Task<T>	**IObservable<T>**

“Push Collections” IObservable and IObserver

```
interface IObservable<out T>
{
    public IDisposable Subscribe (IObserver<out T> observer);
}

interface IObserver<in T>
{
    public void OnCompleted ();
    public void OnError (Exception error);
    public void OnNext (T value);
}
```

System.Reactive.LINQ

- **Create an observable sequence**

  `Create, Generate, Defer, Range`
- **Convert to or from Observable sequences**

  `FromAsyncPattern, FromEvent, FromEventPattern, ToObservable, ToEnumerable`
- **Combine multiple observable sequences into a single sequence**

  `Amb, Concat, StartWith, Merge, Repeat, Zip`
- **Functional:**

  `Let, Prune, Publish, Replay`
- **Mathematical operations on sequences**

  `Aggregate, Count, Min, Max, Sum`
- **Time-based operations:**

  `Delay, Interval, TimeInterval, Timestamp, Timeout`
- **Handling Exceptions:**

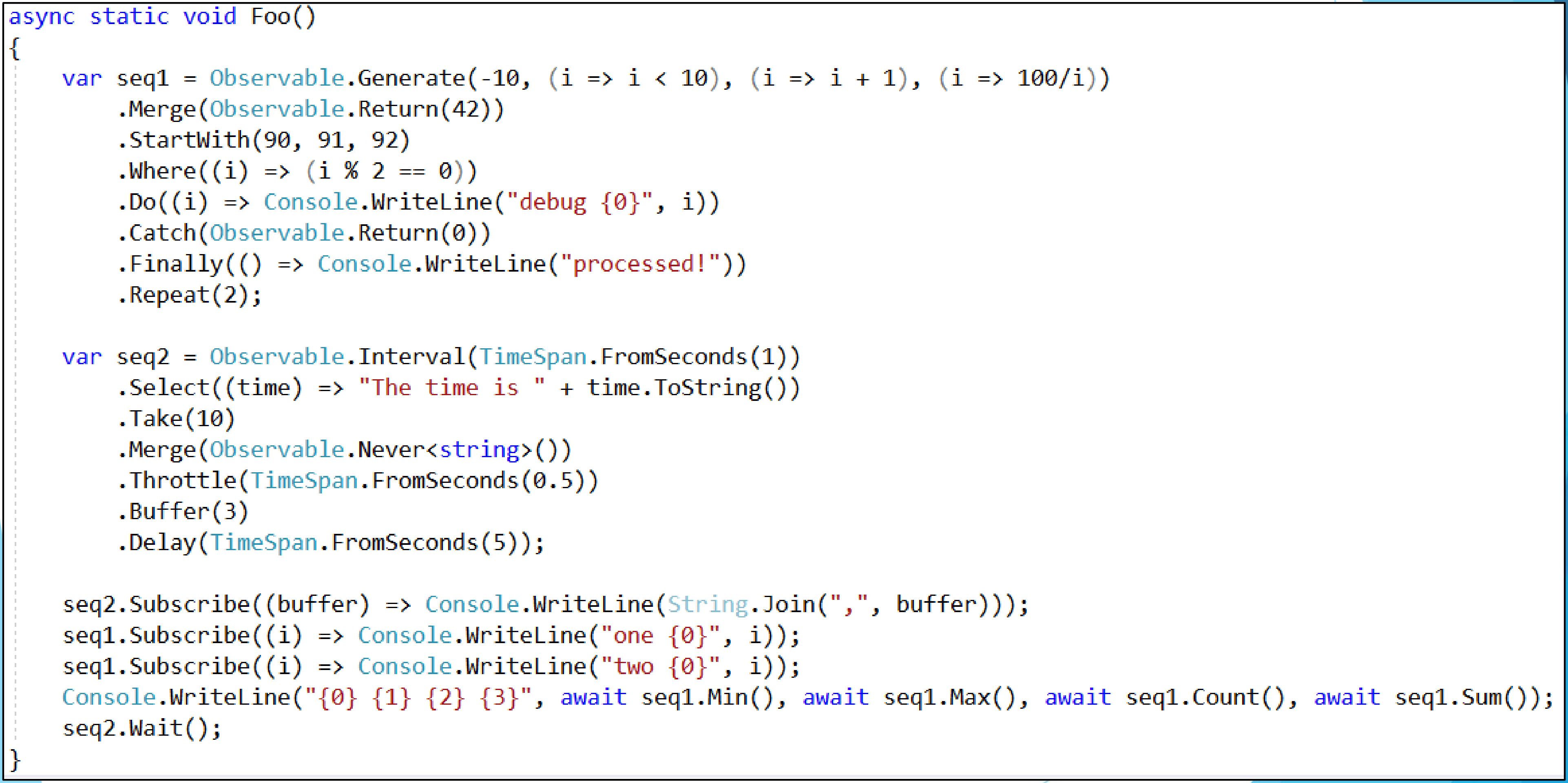
  `Catch, Finally, Retry, OnErrorResumeNext`
- **Miscellaneous operators**

  `Do, Run, Remotable`
- **Filtering and selecting values in a sequence:**

  `Take, TakeUntil, TakeWhile, Select, SelectMany, Skip, SkipUntil, SkipWhile`
- **Primitives:**

  `Never, Empty, Return, Throw`

Observable Operator Examples

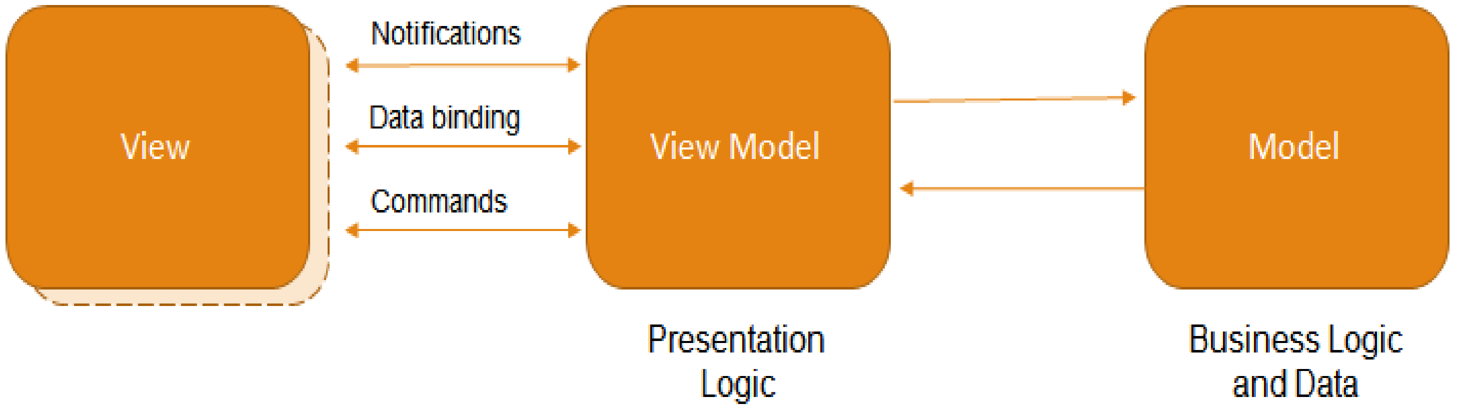


```
async static void Foo()
{
    var seq1 = Observable.Generate(-10, (i => i < 10), (i => i + 1), (i => 100/i))
        .Merge(Observable.Return(42))
        .StartWith(90, 91, 92)
        .Where((i) => (i % 2 == 0))
        .Do((i) => Console.WriteLine("debug {0}", i))
        .Catch(Observable.Return(0))
        .Finally(() => Console.WriteLine("processed!"))
        .Repeat(2);

    var seq2 = Observable.Interval(TimeSpan.FromSeconds(1))
        .Select((time) => "The time is " + time.ToString())
        .Take(10)
        .Merge(Observable.Never<string>())
        .Throttle(TimeSpan.FromSeconds(0.5))
        .Buffer(3)
        .Delay(TimeSpan.FromSeconds(5));

    seq2.Subscribe((buffer) => Console.WriteLine(String.Join(",", buffer)));
    seq1.Subscribe((i) => Console.WriteLine("one {0}", i));
    seq1.Subscribe((i) => Console.WriteLine("two {0}", i));
    Console.WriteLine("{0} {1} {2} {3}", await seq1.Min(), await seq1.Max(), await seq1.Count(), await seq1.Sum());
    seq2.Wait();
}
```

ReactiveUI

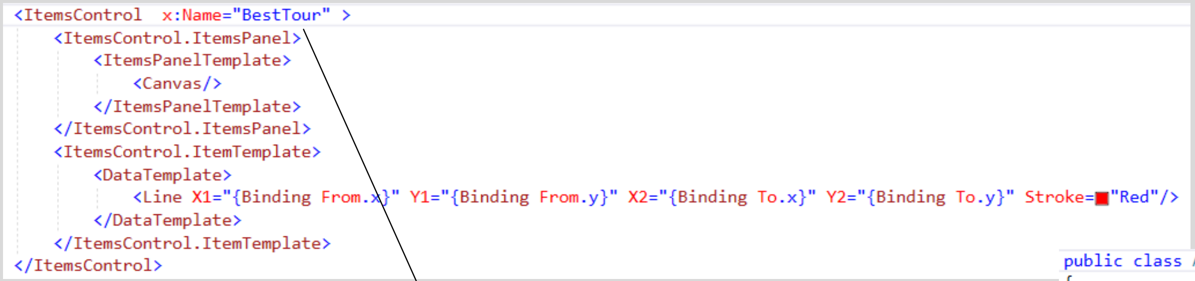
- https://reactiveui.net/
- A functional reactive Model-View-ViewModel (MVVM) framework for .NET
- Supports:
  - Windows Presentation Foundation (WPF)
  - Windows Forms
  - Universal Windows Platform (UWP)
  - Xamarin (IOS, Android and Mac)

MVVM (Model-View-ViewModel)



https://documentation.devexpress.com/WPF/15112/MVVM-Framework

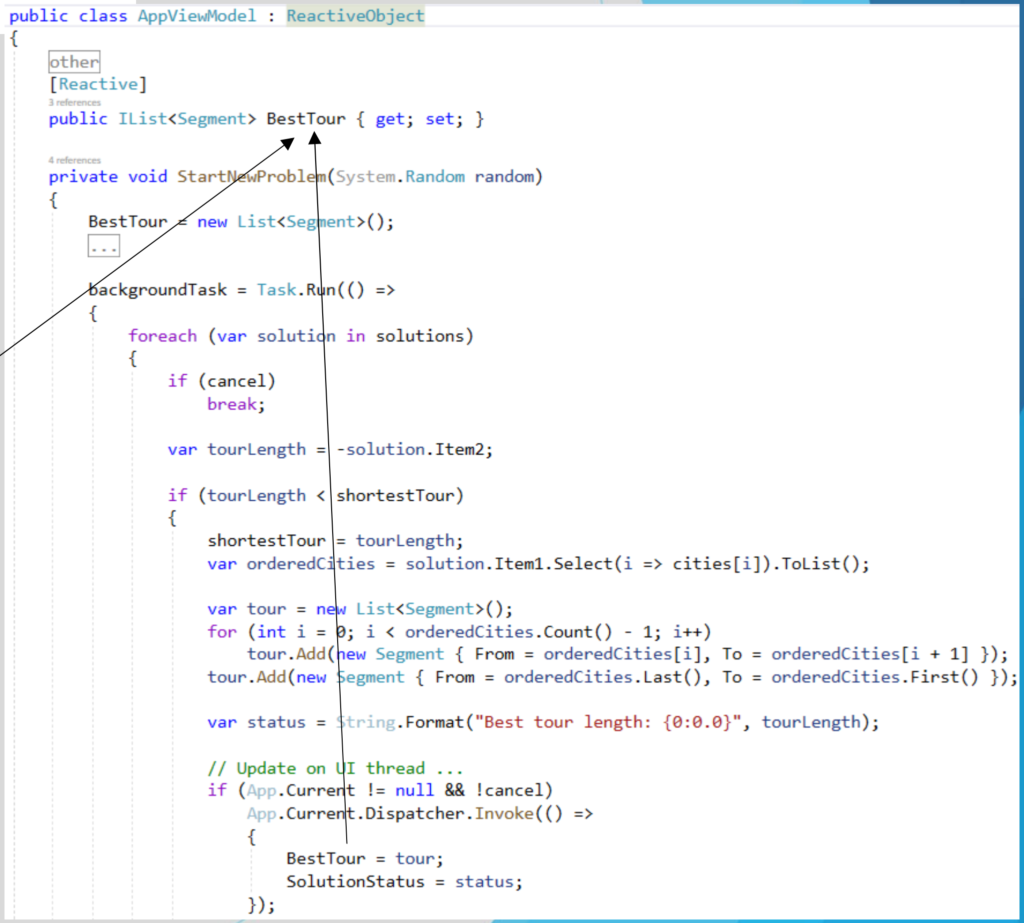
- ViewModel is an abstracted version of the View
  - Observable properties expose the data that should be presented (somehow)
  - Reactive commands expose actions that the user might perform (somehow)
  - Can be tested in isolation (via scripts) as no actual GUI interaction is required.



```
<ItemsControl  x:Name="BestTour" >
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <Canvas/>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Line X1="{Binding From.x}" Y1="{Binding From.y}" X2="{Binding To.x}" Y2="{Binding To.y}" Stroke=🟥"Red"/>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

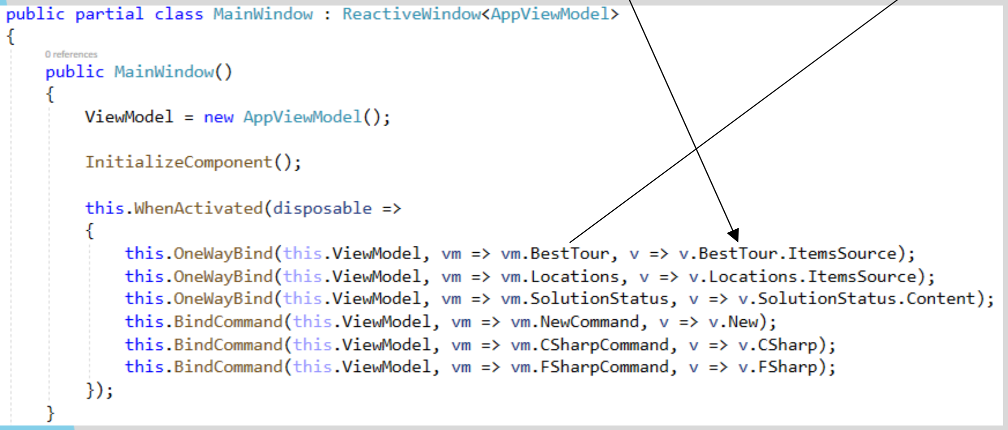
MainWindow.xaml

ViewModel.cs



```
public class AppViewModel : ReactiveObject
{
    other
    [Reactive]
    3 references
    public IList<Segment> BestTour { get; set; }

    4 references
    private void StartNewProblem(System.Random random)
    {
        BestTour = new List<Segment>();
        ...

        backgroundTask = Task.Run(() =>
        {
            foreach (var solution in solutions)
            {
                if (cancel)
                    break;

                var tourLength = -solution.Item2;

                if (tourLength < shortestTour)
                {
                    shortestTour = tourLength;
                    var orderedCities = solution.Item1.Select(i => cities[i]).ToList();

                    var tour = new List<Segment>();
                    for (int i = 0; i < orderedCities.Count() - 1; i++)
                        tour.Add(new Segment { From = orderedCities[i], To = orderedCities[i + 1] });
                    tour.Add(new Segment { From = orderedCities.Last(), To = orderedCities.First() });

                    var status = String.Format("Best tour length: {0:0.0}", tourLength);

                    // Update on UI thread ...
                    if (App.Current != null && !cancel)
                        App.Current.Dispatcher.Invoke(() =>
                        {
                            BestTour = tour;
                            SolutionStatus = status;
                        });
```

MainWindow.xaml.cs



```
public partial class MainWindow : ReactiveWindow<AppViewModel>
{
    0 references
    public MainWindow()
    {
        ViewModel = new AppViewModel();

        InitializeComponent();

        this.WhenActivated(disposable =>
        {
            this.OneWayBind(this.ViewModel, vm => vm.BestTour, v => v.BestTour.ItemsSource);
            this.OneWayBind(this.ViewModel, vm => vm.Locations, v => v.Locations.ItemsSource);
            this.OneWayBind(this.ViewModel, vm => vm.SolutionStatus, v => v.SolutionStatus.Content);
            this.BindCommand(this.ViewModel, vm => vm.NewCommand, v => v.New);
            this.BindCommand(this.ViewModel, vm => vm.CSharpCommand, v => v.CSharp);
            this.BindCommand(this.ViewModel, vm => vm.FSharpCommand, v => v.FSharp);
        });
    }
```

React.JS

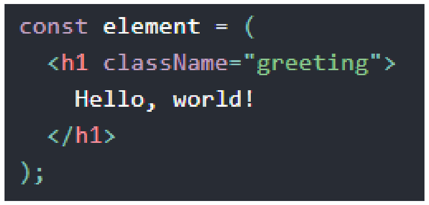
- https://reactjs.org/
- A JavaScript library for building user interfaces

React Components

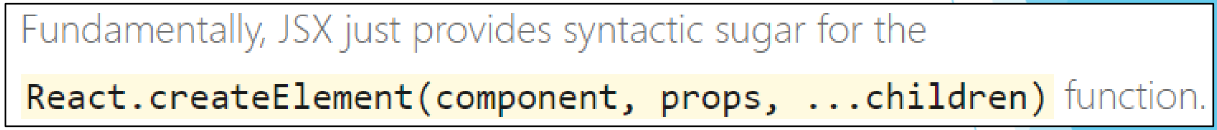
- Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) *separate concerns* by separating Presentation and Logic
- React *separates concerns* by creating separate components (widgets) that each combine presentation and logic.
  - Rendering logic (how the data is displayed)
  - How events are handled
  - How the state changes over time

React Elements (JSX)

- https://reactjs.org/docs/introducing-jsx.html
- https://facebook.github.io/jsx/
- JSX Syntax:

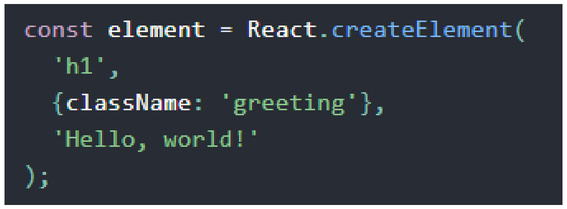


```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```



Fundamentally, JSX just provides syntactic sugar for the `React.createElement(component, props, ...children)` function.

- At build time the JSX is compiled/translated into imperative JavaScript code:



```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React Components

- https://reactjs.org/docs/components-and-props.html
- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Function components:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- Class components:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- Nesting components:

```
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

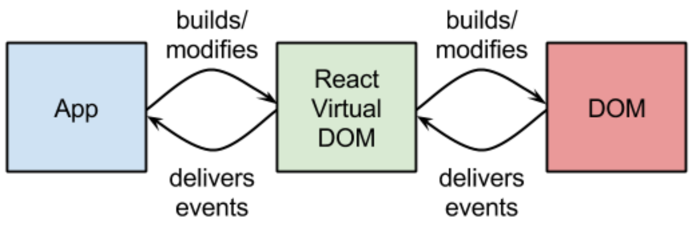
- Rendering Components:

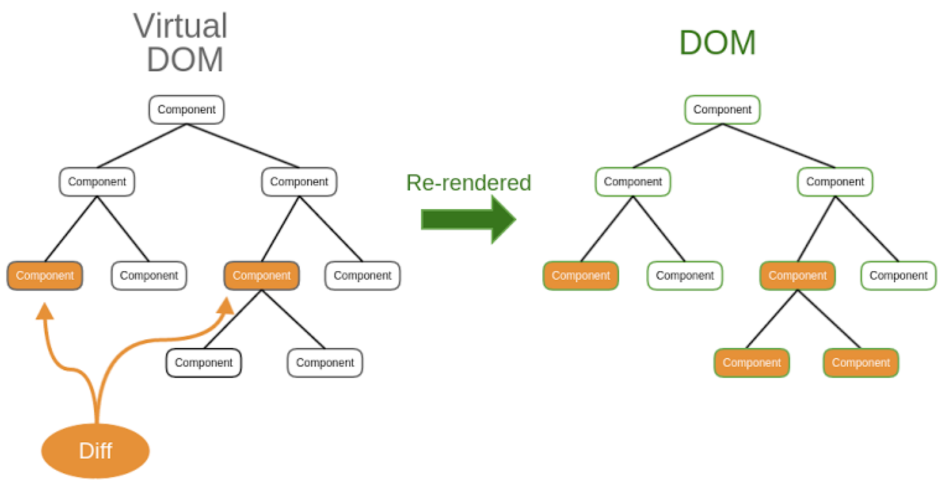
```
<div id="root"/>
```

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

React Virtual DOM and Re-rendering

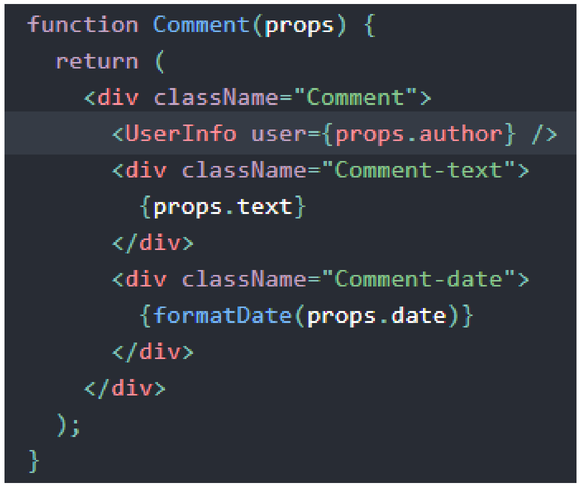
- React Elements are light-weight, they are not HTML DOM objects
- Render method compares the tree of React objects to the tree of HTML DOM objects and only changes the parts of the DOM that have actually changed.





Immutable Properties

- Properties are provided when React Component is created and cannot be changed.

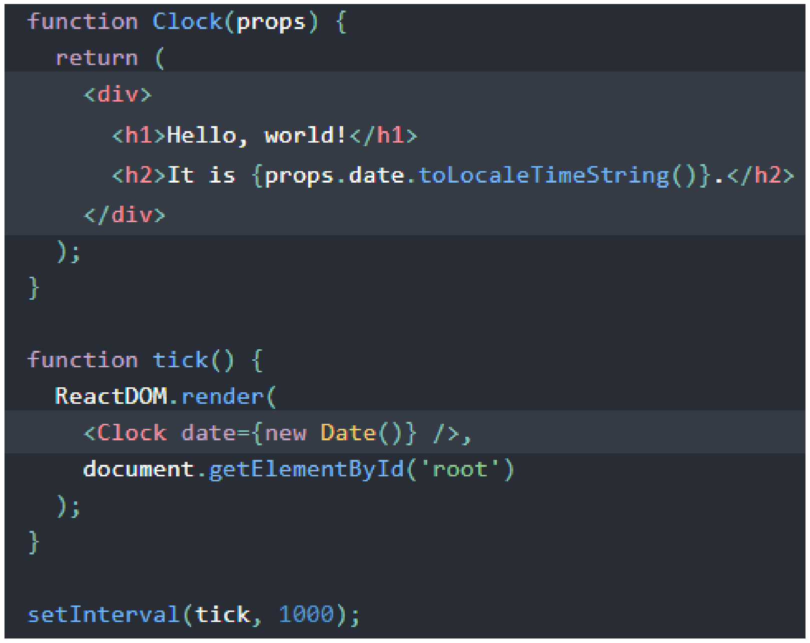


```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

- **All React components must act like pure functions with respect to their props.**

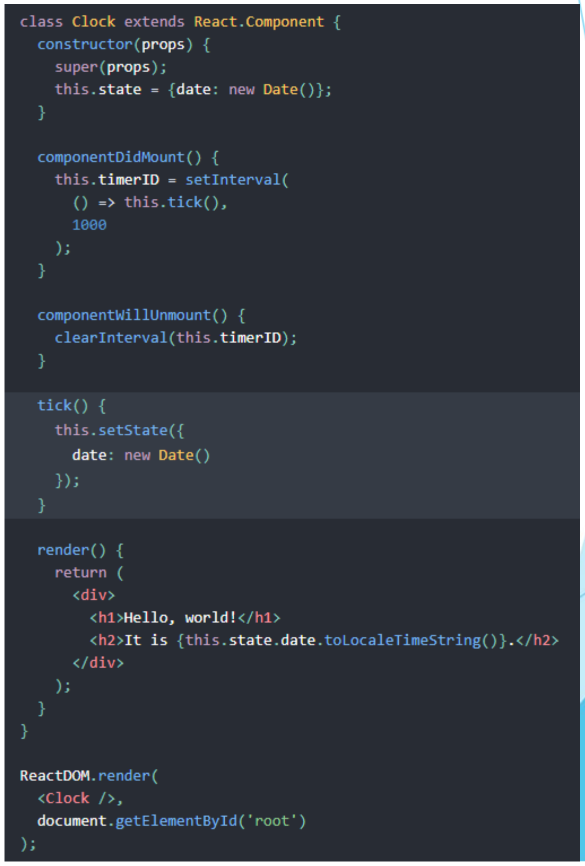
Mutable State

- Re-render a new instance of the component:



```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

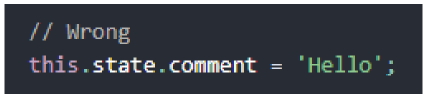
- Or, change the state of the existing instance:



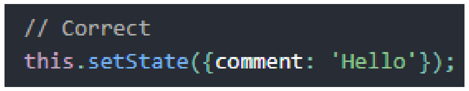
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Updating the State

- Do Not Modify State Directly!

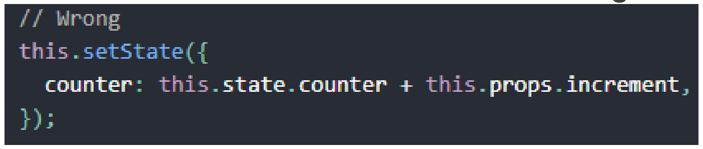


```
// Wrong
this.state.comment = 'Hello';
```

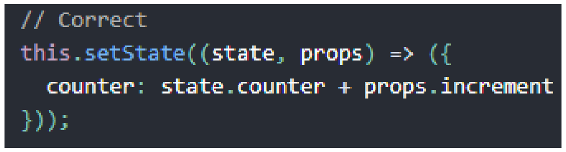


```
// Correct
this.setState({comment: 'Hello'});
```

- React may batch multiple `setState()` calls into a single update for performance.
- Because props and state may be updated asynchronously, you should not rely on their current values for calculating the new state:

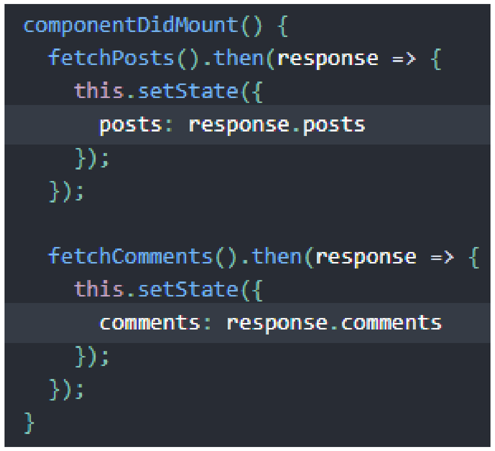


```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```



```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

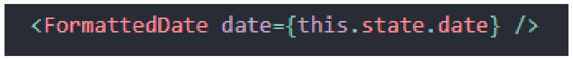
- A state update might only update part of the state, the remaining state stays the same:



```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

Sharing Data between React components

- Each React component encapsulates its own state.
- A parent component shouldn't know whether its child components are stateful or stateless
  - that should be an implementation decision for the child components.
- Data flows down from parent components to child components via properties
  - The properties of a child component might be set based on either the properties or the state of the parent.



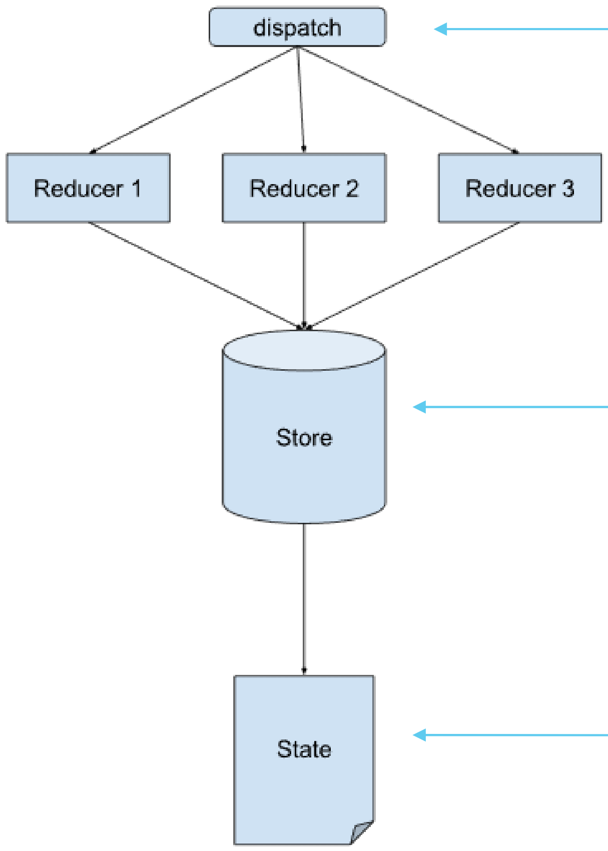
```
<FormattedDate date={this.state.date} />
```

  - The child component can use its properties to initialize its own internal state.
- Encapsulating State in each component works well for simple applications, however
  - If two siblings components both need to update the same state, the state needs to be "*lifted*" and stored in the parent and the parent needs to provide a call back function for updating this state (passed to the children via a property).

Redux

- https://redux.js.org/
- Instead of encapsulating state in each separate component – store it all together in one big state container.
  - Makes it easier to share state between components
  - When multiple components are (asynchronously) updating the same state, a centralized view makes it easier to reason about what is happening and ensure everything remains consistent, up to date and invariants are preserved.
  - Easier to test and debug – we can obtain a trace all the updates that occurred and the order in which they were applied. So, if we ever get into an invalid state we can debug by replaying the trace to discover where our logic went wrong.

Redux Store



Action requesting update part of the state in the store

Pure functions that apply an action to the current store state to create the new store state.

Contents of the store cannot be directly accessed by the application

store.getState() returns a snapshot copy of the current store state.

Redux Actions

- **Actions** are plain old JavaScript objects that have a type property and an optional action specific “payload” describing the details of the state change to be performed.

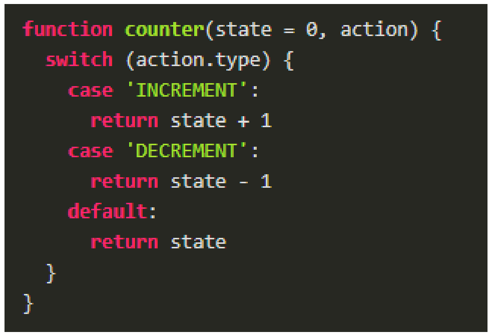
```
const ADD_TODO = 'ADD_TODO'
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

- Actions are passed to the store by calling `store.dispatch(action)`

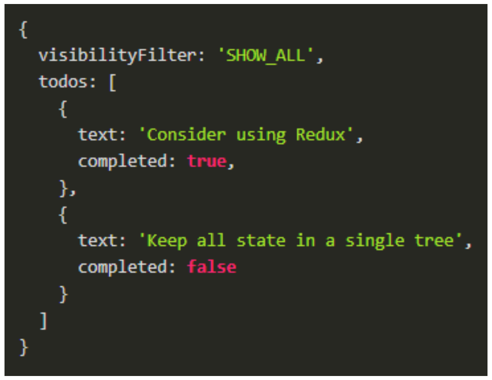
Redux Reducers

- The only way to update the state of the store is via a reducer.
- A reducer is a pure function that takes the current state of the store and an action to be performed and returns what will become the new updated state of the store:



```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

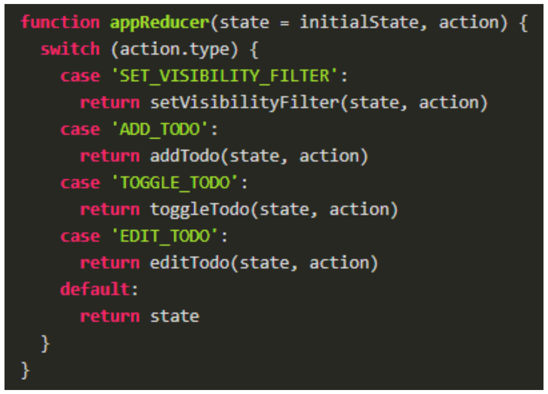
- In general, the state stored in the store will be a single object that contains many different properties that can be independently updated:



```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

Splitting up Reducer Logic

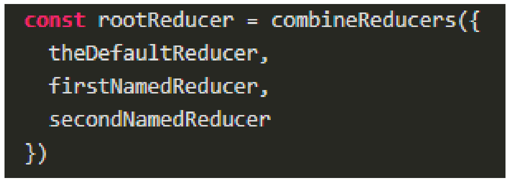
- Since the state store contains a variety of different data used by different React components in the App, rather than having just one big reducer containing all logic, there is typically a root reducer that delegates to a collection of sub-reducers, for example based on the action type:



```
function appReducer(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return setVisibilityFilter(state, action)
    case 'ADD_TODO':
      return addTodo(state, action)
    case 'TOGGLE_TODO':
      return toggleTodo(state, action)
    case 'EDIT_TODO':
      return editTodo(state, action)
    default:
      return state
  }
}
```

- Another common way of decomposing a reducer is by dividing the state into slices and having a separate reducer for each slice. A given action might therefore involve several slice reducers updating different parts of the state.

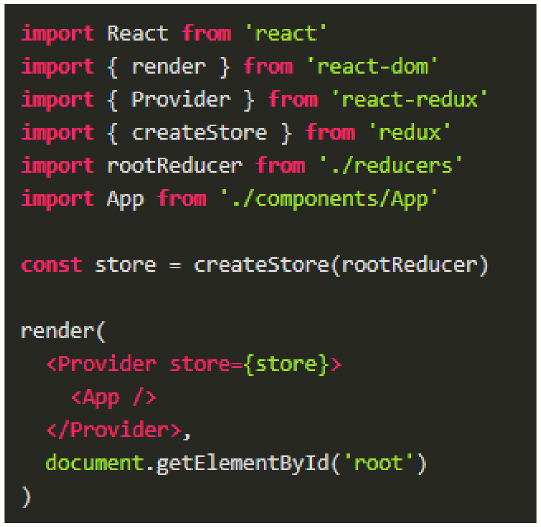
- `combineReducer` is an example of a higher-order reducer – it takes as input a list of slice reducers and returns a new reducer function:



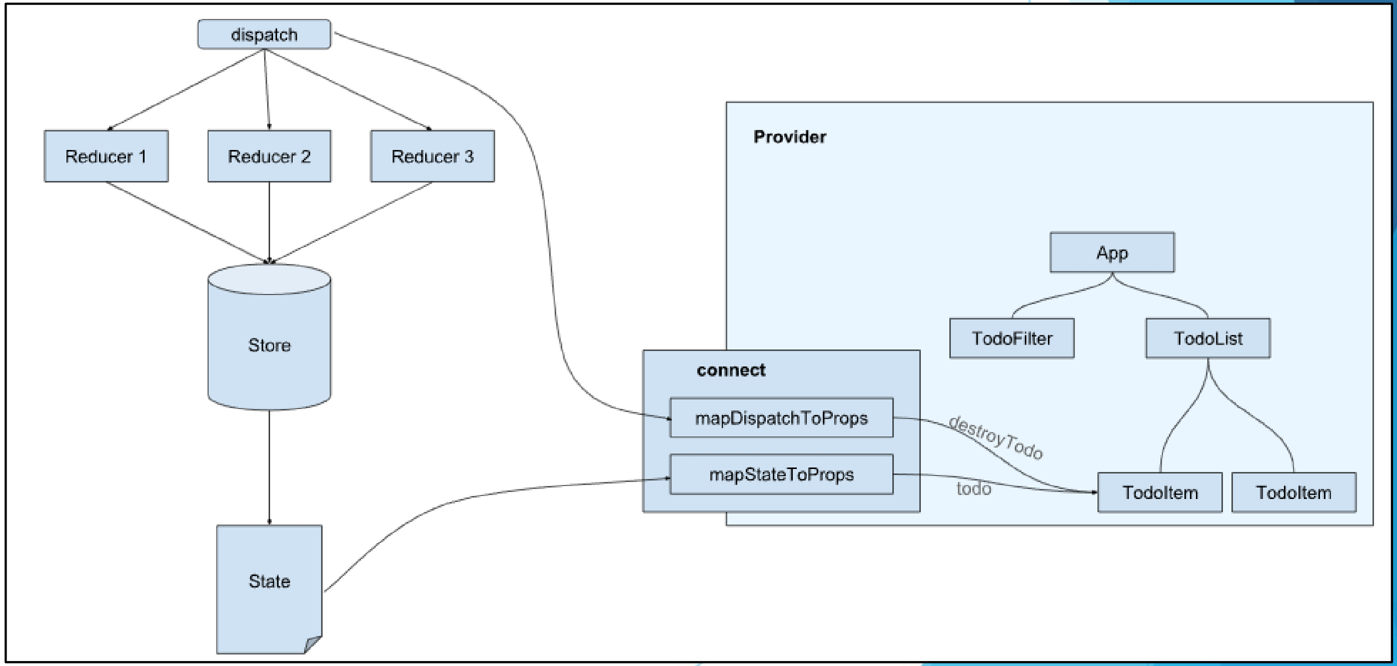
```
const rootReducer = combineReducers({
  theDefaultReducer,
  firstNamedReducer,
  secondNamedReducer
})
```

Creating and accessing the Store

- The store must be created somewhere, typically in the root component.
- The `createStore` function takes as argument the root reducer function.
- The `Provider` component allows all nested components convenient access to the store (without having to explicitly pass it down via properties.

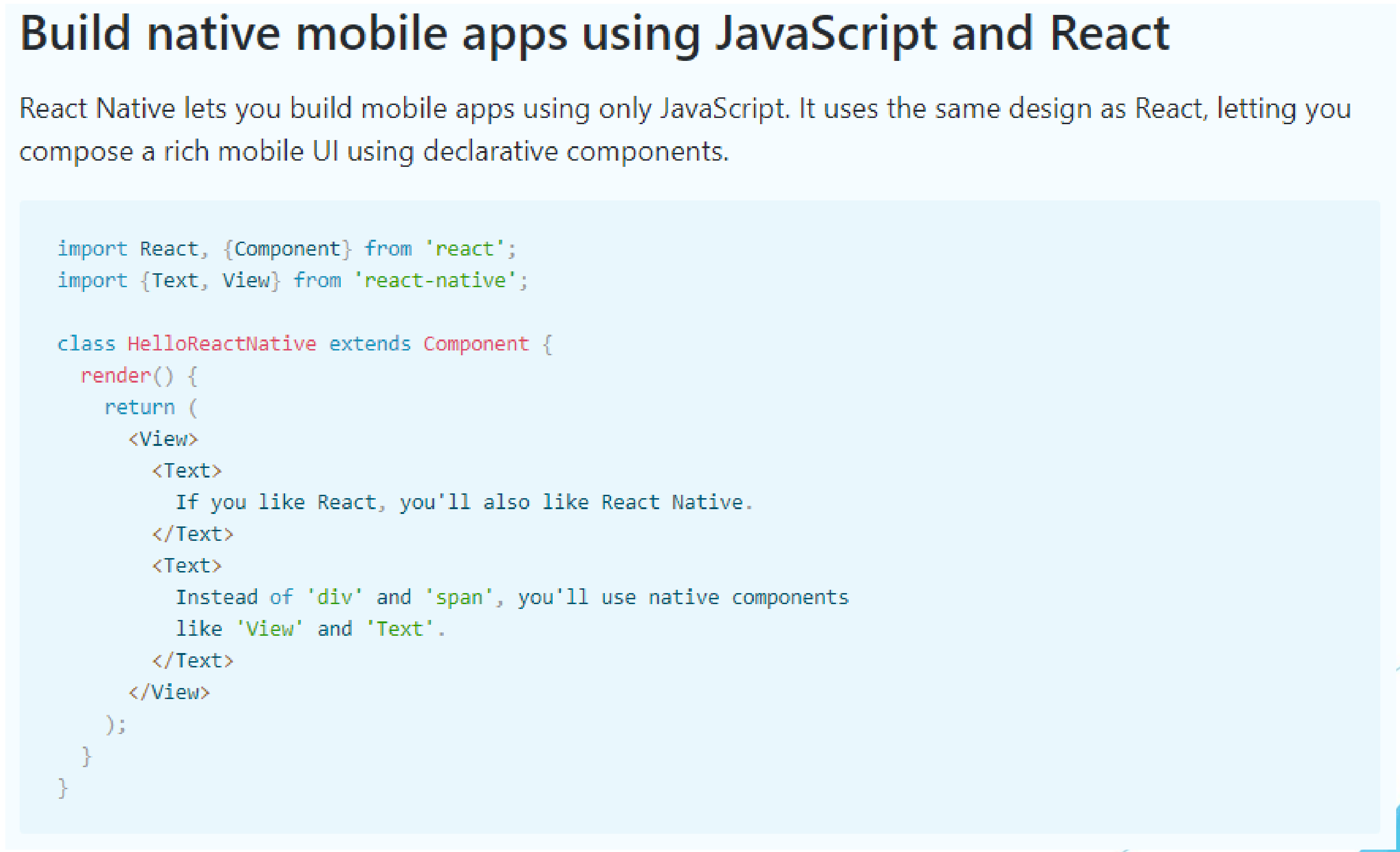


```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import rootReducer from './reducers'
import App from './components/App'

const store = createStore(rootReducer)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```



React Native

- https://facebook.github.io/react-native/



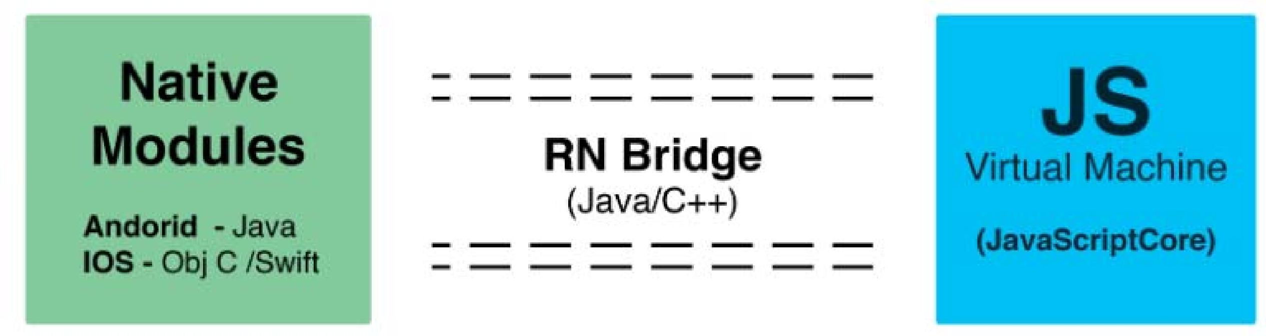
Build native mobile apps using JavaScript and React

React Native lets you build mobile apps using only JavaScript. It uses the same design as React, letting you compose a rich mobile UI using declarative components.

```
import React, {Component} from 'react';
import {Text, View} from 'react-native';

class HelloReactNative extends Component {
  render() {
    return (
      <View>
        <Text>
          If you like React, you'll also like React Native.
        </Text>
        <Text>
          Instead of 'div' and 'span', you'll use native components
          like 'View' and 'Text'.
        </Text>
      </View>
    );
  }
}
```

React Native Internals

- Can write most of the application in pure JavaScript
  - same code base works on both IOS and Android!
- JavaScript code drives the underlying native application via a JS ⇔ Native bridge:

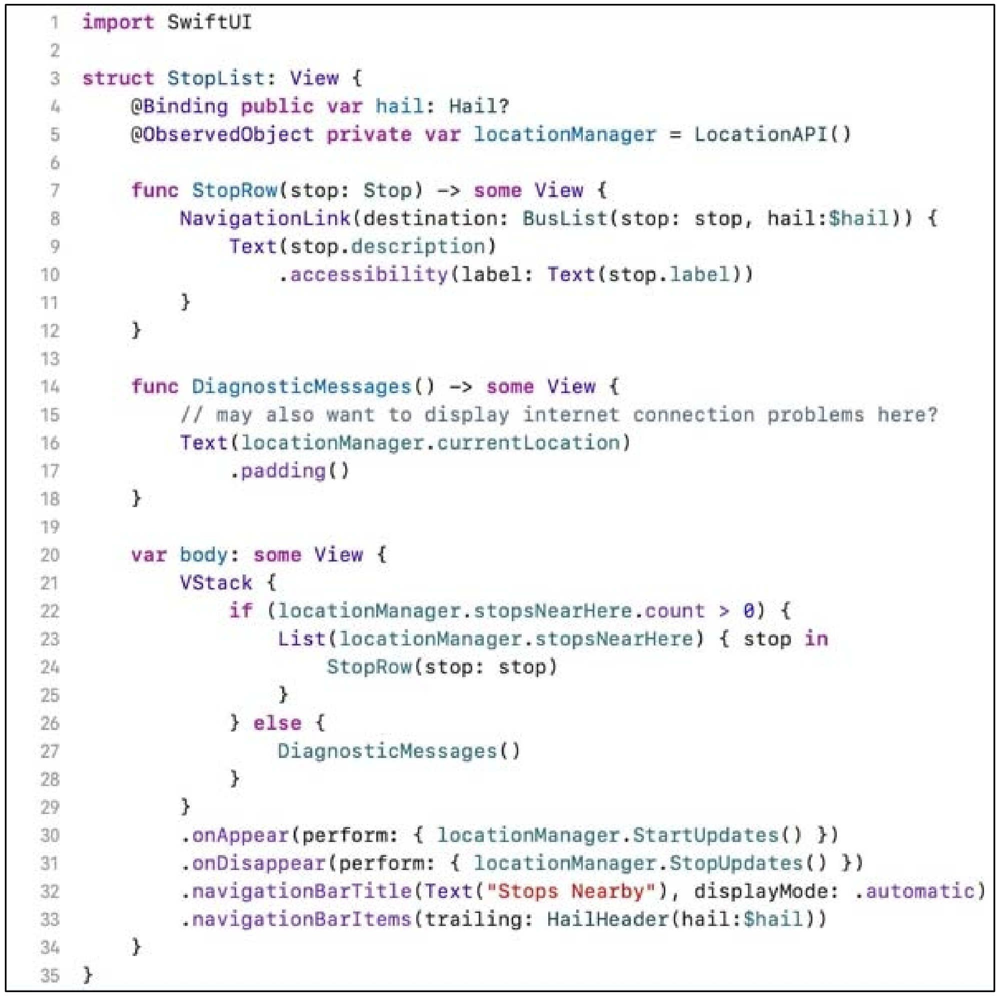


- Can also hand write native code (Java for Android or Swift for IOS) directly for sections where platform specific behaviour is required.

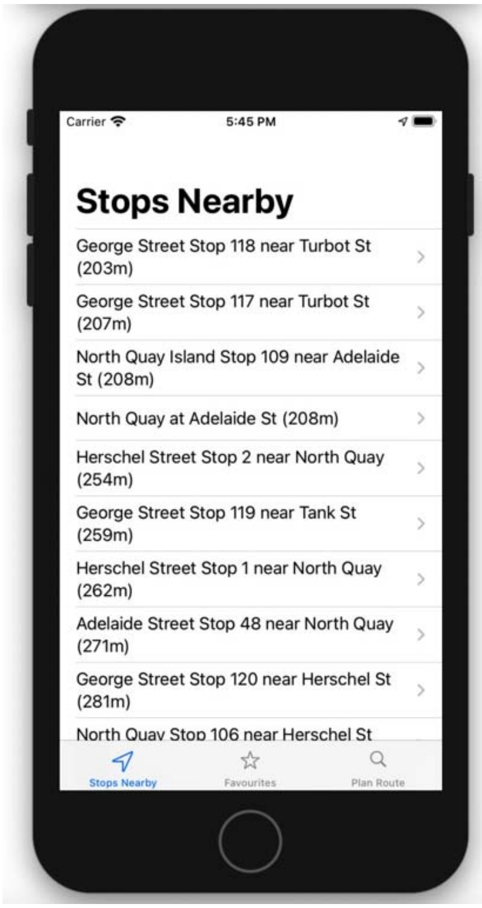
Native Mobile Development on Android

- Java being replaced by Kotlin due to better support for Functional and Reactive Programming.

Functional Reactive Programming using Swift UI for IOS Development



```swift
import SwiftUI

struct StopList: View {
    @Binding public var hail: Hail?
    @ObservedObject private var locationManager = LocationAPI()

    func StopRow(stop: Stop) -> some View {
        NavigationLink(destination: BusList(stop: stop, hail:$hail)) {
            Text(stop.description)
                .accessibility(label: Text(stop.label))
        }
    }

    func DiagnosticMessages() -> some View {
        // may also want to display internet connection problems here?
        Text(locationManager.currentLocation)
            .padding()
    }

    var body: some View {
        VStack {
            if (locationManager.stopsNearHere.count > 0) {
                List(locationManager.stopsNearHere) { stop in
                    StopRow(stop: stop)
                }
            } else {
                DiagnosticMessages()
            }
        }
        .onAppear(perform: { locationManager.StartUpdates() })
        .onDisappear(perform: { locationManager.StopUpdates() })
        .navigationBarTitle(Text("Stops Nearby"), displayMode: .automatic)
        .navigationBarItems(trailing: HailHeader(hail:$hail))
    }
}
```



Stops Nearby

George Street Stop 118 near Turbot St (203m)

George Street Stop 117 near Turbot St (207m)

North Quay Island Stop 109 near Adelaide St (208m)

North Quay at Adelaide St (208m)

Herschel Street Stop 2 near North Quay (254m)

George Street Stop 119 near Tank St (259m)

Herschel Street Stop 1 near North Quay (262m)

Adelaide Street Stop 48 near North Quay (271m)

George Street Stop 120 near Herschel St (281m)

North Quay Stop 106 near Herschel St