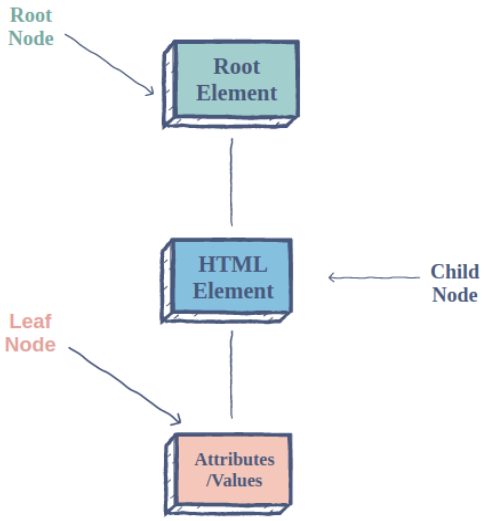
DOM vs. Virtual DOM

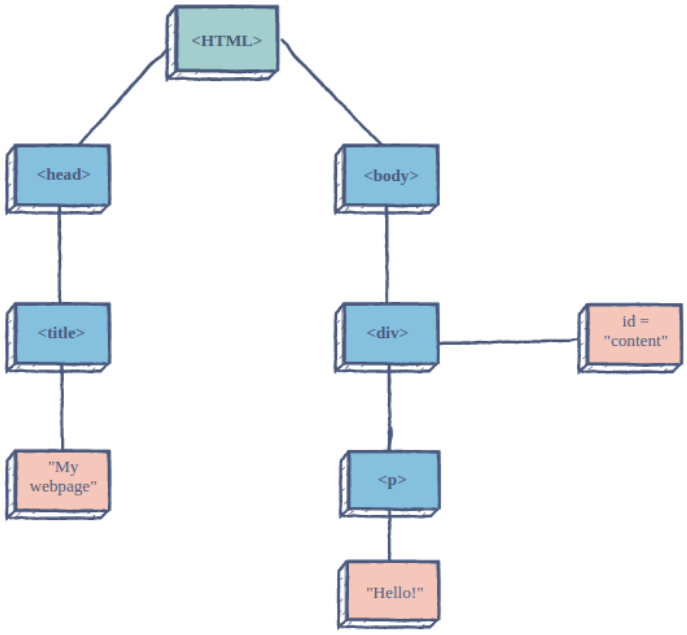
- **Document Object Model (DOM)**[1] is the platform and an interface allowing the program and the script to access and update the content, structure, and the style of the document (HTML web-pages).
- DOM contains nodes where each node represents an HMTL element.
- Javascript uses DOM to access and manipulate the document and its elements.
- DOM updates all objects even if only one of the objects was changed - inefficient.



- In React, for every DOM object, there is a corresponding **Virtual DOM (VDOM)**[2] object.
- VDOM has the same properties as DOM, but it lacks the power to directly change what is on the screen.
- When a VDOM object is changed, React only updates the real object (on actual DOM) that corresponds to the VDOM object that was changed.

How DOM looks like for the given HTML document

```html
<html>
  <head>
    <title> My webpage </title>
  </head>
  <body>
    <div id="content">
      <p> Hello!</p>
    </div>
  </body>
</html>
```



[1] For more info about DOM: The Document Object Model
[2] Virtual DOM and Internals – React, React: The Virtual DOM

React

- Front-end JS script library developed by Facebook. It uses a **component** based approach.
- Few information about react:

3. What are the features of React?

Major features of React are listed below:

i. It uses the **virtual DOM** instead of the real DOM.
ii. It uses **server-side rendering**.
iii. It follows **uni-directional data flow** or data binding.

4. List some of the major advantages of React.

Some of the major advantages of React are:

i. It increases the application's performance
ii. It can be conveniently used on the client as well as server side
iii. Because of JSX, code's readability increases
iv. React is easy to integrate with other frameworks like Meteor, Angular, etc
v. Using React, writing UI test cases become extremely easy

5. What are the limitations of React?

Limitations of React are listed below:

i. React is just a library, not a full-blown framework
ii. Its library is very large and takes time to understand
iii. It can be little difficult for the novice programmers to understand
iv. Coding gets complex as it uses inline templating and JSX

- **JSX** is a syntax extension to JS, used by React.
  - Instead of artificially separating technologies by putting markup and logic in separate files, React combines them into units called **components**.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- React follows uni-directional data flow or data binding.

  One-way Data Binding: ReactJS uses one-way data binding. In one-way data binding one of the following conditions can be followed:

  - **Component to View:** Any change in component data would get reflected in the view.
  - **View to Component:** Any change in View would get reflected in the component's data.

- Every component must have a **render()** function.
  - It returns a single React element, a representation of the DOM component.

- React vs Vue

  To recap, our findings, Vue's strengths are:

  - Flexible options for template or render functions
  - Simplicity in syntax and project setup
  - Faster rendering and smaller size

  React's strengths:

  - Better at scale, studier and more testable
  - Web and native apps
  - Bigger ecosystem with more support and tools available

  However, both React and Vue are exceptional UI libraries and have more similarities than differences. Most of their best features are shared:

  - Fast rendering with virtual DOM
  - Lightweight
  - Reactive components
  - Server-side rendering
  - Easy integration with router, bundler and state management
  - Great support and community

- React vs Angular

TOPIC	REACT	ANGULAR
1. ARCHITECTURE	Only the View of MVC	Complete MVC
2. RENDERING	Server-side rendering	Client-side rendering
3. DOM	Uses virtual DOM	Uses real DOM
4. DATA BINDING	One-way data binding	Two-way data binding
5. DEBUGGING	Compile time debugging	Runtime debugging
6. AUTHOR	Facebook	Google

- **References (refs)** in React is an attribute which helps to store a reference to a particular React element or component, which will be returned by the component's render configuration function.
    - That is, it is used to return references to a particular element or component returned by render().

Redux[3]

- **Redux** is one of the most trending libraries for front-end development.
- It is a predictable state container for JavaScript applications and is used for the entire application's state management.
- Applications developed with Redux are easy to test and can run in different environments showing consistent behavior.
- Follows **three principles**:
    1. **Single source of truth**:
       The state of the entire application is stored in an object/ state tree within a single **store**. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
    2. **State is read-only**:
       The only way to change the state is to trigger an **action**. An action is a plain JS object describing the change. Just like state is the minimal representation of data, the action is the minimal representation of the change to that data.
    3. **Changes are made with pure functions**:
       In order to specify how the state tree is transformed by actions, you need pure functions. **Pure functions** are those whose return value depends solely on the values of their arguments.

[3] Top 50 React Interview Questions and Answers For 2021

Components, Props, and State[4]

- **Components** are the building blocks of React App's UI. It breaks up the UI into small independent and reusable pieces.
- Two types of components: **function component** and **class component**.
  - **Function** is easier to test/debug and it is planned to be more optimized[5].
  - **Class** is useful if we need more intuitive and detailed controls with the life-cycle.
- **Properties (props)** is an object that contains JSX attributes and children that influences the output of render, and are passed to the components.
  - Acts as a component's configuration (options) and is immutable within the receiving component.

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

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- **State** is an object that also influences the output of the render, but is managed within the component (as opposed to being passed to the component like props).
  - Manages its own state internally and is mutable. It is also private.

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

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

[4] Component State – React docs
[5] 20 responses to "React Functional or Class Components: Everything you need to know"

- Mutating (updating) state is done using **setState**[6], an asynchronous operation.

```
incrementCount() {
  // Note: this will *not* work as intended.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // When React re-renders the component, `this.state.count` will be 1, but you
expected 3.

  // This is because `incrementCount()` function above reads from
`this.state.count`,
  // but React doesn't update `this.state.count` until the component is re-
rendered.
  // So `incrementCount()` ends up reading `this.state.count` as 0 every time,
and sets it to 1.

  // The fix is described below!
}
```

Update function allows access to the current state value inside the updater.

```
incrementCount() {
  this.setState((state) => {
    // Important: read `state` instead of `this.state` when updating.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // If you read `this.state.count` now, it would still be 0.
  // But when React re-renders the component, it will be 3.
}
```

[6] State and Lifecycle – React, Component State – React docs

- States could be introduced to function components using **State Hooks**[7].

```
1:  import React, { useState } from 'react';
2:
3:  function Example() {
4:    const [count, setCount] = useState(0);
5:
6:    return (
7:      <div>
8:        <p>You clicked {count} times</p>
9:        <button onClick={() => setCount(count + 1)}>
10:        Click me
11:       </button>
12:     </div>
13:   );
14: }
```

Equivalent class component would look like the following:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

[7] Using the State Hook – React

- Props is the go-to choice to pass data from the parent component to the child[8].
- From child to parent, we need to use **callbacks** and states to pass data.
    - **Callback function** is a function passed into another function as an argument.

```
class ToDoList extends React.Component {

    myCallback = (dataFromChild) => {
        [...we will use the dataFromChild here...]
    },

    render() {
        return (
            <div>
                 <ToDoItem callbackFromParent={this.myCallback}/>
            </div>
        );

    }
}
```

```
class ToDoItem extends React.Component{

    someFn = () => {
        [...somewhere in here I define a variable listInfo which    I
think will be useful as data in my ToDoList component...]

        this.props.callbackFromParent(listInfo);
    },

    render() {
        [...]
    }
};
```

- Mutability comparisons between props and states[9]:

	props	***state***
Can get initial value from parent Component?	Yes	Yes
Can be changed by parent Component?	Yes	No
Can set default values inside Component?*	Yes	Yes
Can change inside Component?	No	Yes
Can set initial value for child Components?	Yes	Yes
Can change in child Components?	Yes	No

[8] Passing Data Between React Components | by Ruth M. Pardee
[9] https://github.com/uberVU/react-guide/blob/master/props-vs-state.md

Pure Components[10]

- In Javascript, **pure functions** do not depend on or modify the state of variables outside their scope.
- In react, **pure components** do not re-render when the value of the state and props has been updated with the same values.
    - In react, a class could extend React.PureComponent class.
    - It's the same as a normal Component, but it takes care of shouldComponentUpdate() by itself through **shallow comparison**.
- **Shallow comparison** (equality checks) simply checks that two different variables reference the same object.
    - This is not going to re-render since the reference of userArray did not change.

```
1
2  class ShalloWCompareComponent extends React.PureComponent {
3    constructor() {
4      super();
5      this.state = {
6        userArray: [1, 2, 3, 4, 5]
7      }
8
9      // The value of Counter is updated to same value during continues interval
10
11     setInterval(() => {
12       this.setState({
13         userArray: userArray.push(6)
14       });
15     }, 1000);
16   }
17
18   render() {
19     return <b>Array Length is: {this.state.userArray.length}</b>
20   }
21 }
```

- **Spread syntax** (...) allows an iterable to be expanded in places where zero or more arguments/elements are expected.

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];

console.log(sum(...numbers));
// expected output: 6
```

[10] What are Pure Components in React | by Mayank Gupta | TechnoFunnel

- Using spread syntax, we can create a new array and set that as userArray. This will change the reference of the userArray and re-render.

```
class ShalloWCompareComponent extends React.PureComponent {
  constructor() {
    super();
    this.state = {
      userArray: [1, 2, 3, 4, 5]
    }
    setInterval(() => {

      // Here we are creating the new Array Object during setState using "Spread" Operator

      this.setState({
        userArray: [...this.state.userArray, 6]
      });
    }, 1000);
  }

  render() {
    return <b>Array Length is: {this.state.userArray.length}</b>
  }
}
```

- **React.memo**[11] is a **higher order component (HOC)** that you can wrap your component with, to achieve the similar effect.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

  - By memoizing the previous result, react will skip rendering the component or part of the component if the prop was unchanged.
  - This means if useState, useReducer, or useContext Hook is in the component, the component will re-render when the state or the context changes.
- You could pass a custom comparison function to prevent default shallow comparison.

```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}
export default React.memo(MyComponent, areEqual);
```

[11] React Top-Level API – React

Higher Order Components (HOC)

- Function that takes a component and returns another component.
- HOC doesn't modify the input component, nor use inheritance to copy its behavior.
    - HOC composes the original component by wrapping it in a container component.
- You should not modify a component's prototype inside a HOC.
    - Instead, use composition byt wrapping the input component in a container component.

```
function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

The above should be changed to ...

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

React Hooks[12]

- With Hooks, you can reuse **stateful logic** without changing the component.
    - Stateful logic: any code that uses state - behaviour created with the use of one or more Hooks.
- Hooks let you split one component into smaller functions based on the pieces that are related to each other.
    - For example, using Effect Hook, we can run specific functions if and only if there is a change in some dependencies.
- Hooks let you use more of React's features without having to use classes.
- Hooks should be called inside loops, conditions, or nested functions.
    - Ensure that Hooks are called in the same order each component renders.

useCallback[13]

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

- Takes a function as an argument and returns a cached/memoized version of it.
- The callback is only changed/updated if one of the dependencies has changed.
    - That is, we only create a new callback, if one of the dependencies changed.
- Reduces the resources by not creating a new callback when it does not need to.

useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- Unlike useCallback (which caches the provided function), **useMemo** invokes the function and caches the result.
    - Very similar to how React.memo works.
- The result is changed if and only if one of its dependencies change.
- But note that useMemo should be used more as a performance optimizer.

---

[12] Hooks API Reference – React,
State Management within React Functional Components with hooks | by Rajesh Naroth | Medium
[13] Previous React's useCallback and useMemo Hooks By Example

useContext

```
const value = useContext(MyContext);
```

- Used to access states outside of the component - used to share states with others.
- The current context value is determined by the value prop of the nearest context provider (<MyContext.Provider>) above the calling component in the tree.

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}
```

UseReducer

- Used to modify state via reducers and actions.
- Preferred over useState when working with a complex state logic involving multiple sub-values, or when the next state depends on the previous one.
- Accepts a reducer of type (state, action) => newState.
- Returns the current state paired with a dispatch method.

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

useState

```
const [state, setState] = useState(initialState);
```

```
setState(newState);
```

useRef[14]

```
const refContainer = useRef(initialValue);
```

- Returns a mutable ref object where .current property is initialized to the argument.
  - This means, unlike createRef, no new ref is returned on every render.
  - .current property survives on every render.

```
const usePrevious = (value) => {
    const previousUserRef = React.useRef()
   React.useEffect(() => {
      previousUserRef.current = value
   }, [value])

   return previousUserRef.current
}
```

- Changing the current value of the useRef does not refresh the DOM.

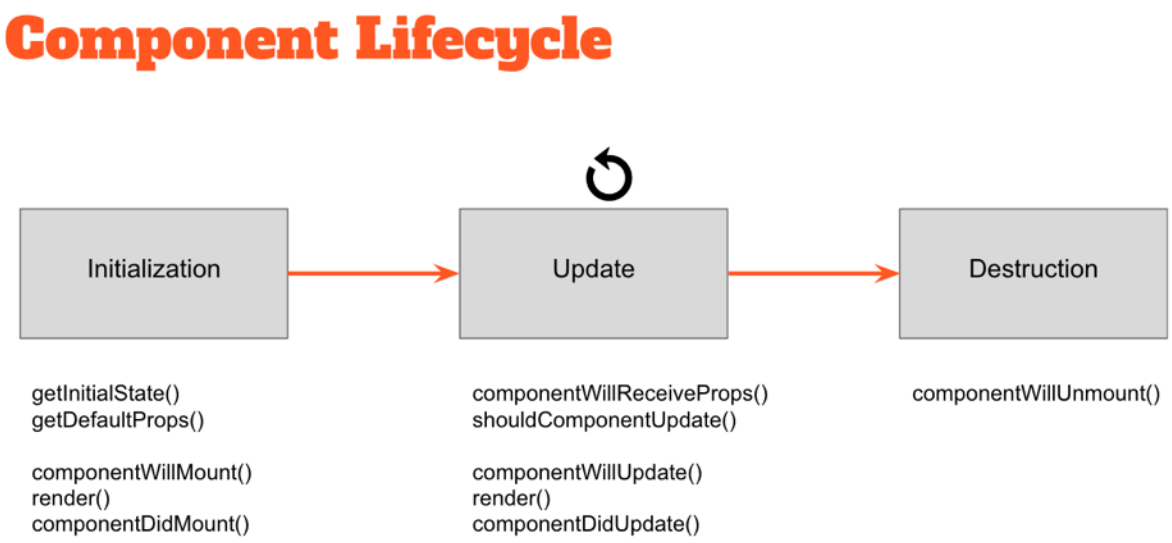
useEffect

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

[14] React useRef: Introduction to useRef Hook - DEV Community

Life-cycle of a React component

- Three main categories: **initialization, state/property updates, destruction**.



- Update happens multiple times whenever the property or the state of the component changes. Others happen only once.
    - In the case in which the component should not re-render, we could use shouldComponentUpdate().

```
class MyComponent extends React.Component {
  // only re-render if the ID has changed!
  shouldComponentUpdate(nextProps, nextState) {
    return nextProps.id === this.props.id;
  }
}
```

- We can introduce life-cycle control to the function components using the **Effect Hook**[15].

[15] React useEffect : A hook to introduce lifecycle methods in functional components, Using the Effect Hook – React

ComponentWillUnmount-

`componentWillUnmount()` is called when a component is being removed from the DOM.

For imitating the functionality of componentWillUnmount, we need to pass the cleanup function in this function.

```
useEffect(function() {
  window.addEventListener(eventName, eventListener);

  return function cleanup() {
    window.removeEventListener(eventName, eventListener);
  }
}, []);
```

useEventListener.jsx hosted with ❤ by GitHub view raw

ComponentDidMount-

`componentDidMount()` is invoked immediately after a component is mounted.

For imitating the functionality of componentDidMount, we need to pass an empty array(**[]**). When you pass an empty array as 2nd argument, the useEffect function will only get invoked once.

ComponentDidUpdate-

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.

For imitating the functionality of componentDidUpdate, we should **not pass the array**. It will then run the effect on every render.(It does invoke after the initial render also, but we will look into that later in the series).

shouldComponentUpdate-

`shouldComponentUpdate()` is used if a component's output is affected by the current change in state or props. The default behaviour is to re-render on every state change.

For imitating the functionality of shouldComponentUpdate, we should pass the values in the array through which the component's output gets affected.

```
1  import React, {useState, useEffect} from "react";
2
3  export default function useDarkMode() {
4    const [enabled, setEnabled] = useState(false);
5    const [otherItemsInComponent, setOtherItems] = useState({});
6
7    function toggleEnabled() {
8      setEnabled(!enabled);
9    }
10   //In this case, we only want to set use-dark-mode once the enabled value changes, and
11   // In this example, any changes happening to otherItemsInComponent, won't invoke the b
12   useEffect(function() {
13     localStorage.setItem("use-dark-mode", enabled);
14   }, [enabled]);
15
16   return (
17     //some components affecting otherItemsInComponent
18     <button onClick={() => toggleEnabled()}>Dark Mode</button>
19   );
20 }
```

useDarkMode.jsx hosted with ❤ by GitHub view raw

- Arrays could also be used to replace componentDidUpdate().

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

- Overview comparison of introducing life-cycle to function component.

```jsx
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```jsx
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

Promise

- A **promise** is an object that may produce a single value some time in the future.
  - This value could be a resolved value or unresolved reason (caused by an error).
- A promise may be in three different states: **fulfilled**, **rejected**, or **pending**.
  - Promises in fulfilled and rejected states are considered **settled**.
- Creating a new Promise object:

```
new Promise(executor)
```

  - **Executor** is a function to be executed by the constructor, which is a code that ties an outcome to the promise.
  - When the promise is resolved, resolutionFunc should be executed with the value of the promise as the first argument.
  - When the promise is rejected, rejectionFunc should be executed with the reason for the rejection as the first argument.
    - These rejections could be JS values, but Error objects are recommended.

```
const myFirstPromise = new Promise((resolve, reject) => {
  // do something asynchronous which eventually calls either:
  //
  //   resolve(someValue)        // fulfilled
  // or
  //   reject("failure reason")  // rejected
});
```

- To consume a promise, use **.then()**, which is a function that returns another promise.

```
promise.then(
  onFulfilled?: Function,
  onRejected?: Function
) => Promise
```

- Example usage of a promise:

```
let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.htm");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

- Since .then() returns a promise, we can chain promises together:

```
const wait = time => new Promise(
  res => setTimeout(() => res(), time)
);

wait(200)
  // onFulfilled() can return a new promise, `x`
  .then(() => new Promise(res => res('foo')))
  // the next promise will assume the state of `x`
  .then(a => a)
  // Above we returned the unwrapped value of `x`
  // so `.then()` above returns a fulfilled promise
  // with that value:
  .then(b => console.log(b)) // 'foo'
  // Note that `null` is a valid promise value:
  .then(() => null)
  .then(c => console.log(c)) // null
  // The following error is not reported yet:
  .then(() => {throw new Error('foo');})
  // Instead, the returned promise is rejected
  // with the error as the reason:
  .then(
    // Nothing is logged here due to the error above:
    d => console.log(`d: ${ d }`),
    // Now we handle the error (rejection reason)
    e => console.log(e)) // [Error: foo]
  // With the previous exception handled, we can continue:
  .then(f => console.log(`f: ${ f }`)) // f: undefined
  // The following doesn't log. e was already handled,
  // so this handler doesn't get called:
  .catch(e => console.log(e))
  .then(() => { throw new Error('bar'); })
  // When a promise is rejected, success handlers get skipped.
  // Nothing logs here because of the 'bar' exception:
  .then(g => console.log(`g: ${ g }`))
  .catch(h => console.log(h)) // [Error: bar]
;
```

- The **.catch()** block at the end handles errors. If an error rises from the resolution function or if the error rises from rejection of some kind, the error will be caught by .catch() block.

Asynchronous vs. Synchronous

- **Synchronous Operations** that run block the next operation until it completes.
- Making synchronous calls to resources can lead to long response times locking up the UI until the resource responds.
    - Let's say you want some requests to start as soon as possible but still allow the rest of the page to load what it can to enable the requests.
- **Asynchronous Operations** happen independently from the main program flow.
    - For example, if the main program is running on thread A, it runs on thread B.
    - A common use is querying a database and using the result.
    - For example, while the database loads and responds to the request, maybe we can make the rest of the page or other resources to load.

- A function could be converted to an **async function**, which expects the possibility of the await keyword being used inside of itself - to invoke asynchronous code.
    - An async function guarantees that the return values are converted to promises.
    - This means consuming the returned value requires .then() block.
- **await** can be put in front of any async promise-based function to pause the code on that line until the promise fulfills and actually returns a result.

- Consider the following code:

```
fetch('coffee.jpg')
.then(response => {
  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }
  return response.blob();
})
.then(myBlob => {
  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
})
.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

- Using async and await, we can make above code much cleaner:
    - Instead of needing to chain a .then() block on to the end of each promise-based method, you just need to add an await keyword before the method call, and then assign the result to a variable.

```
async function myFetch() {
  let response = await fetch('coffee.jpg');

  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }

  let myBlob = await response.blob();

  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

myFetch()
.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

- Combining .then(), async, and await to be more flexible:

```
async function myFetch() {
  let response = await fetch('coffee.jpg');
  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }
  return await response.blob();

}

myFetch().then((blob) => {
  let objectURL = URL.createObjectURL(blob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}).catch(e => console.log(e));
```

- **Promise.all()** takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises.
    - Returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises.

```
async function fetchAndDecode(url, type) {
  let response = await fetch(url);

  let content;

  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  } else {
    if(type === 'blob') {
      content = await response.blob();
    } else if(type === 'text') {
      content = await response.text();
    }
  }

  return content;


}

async function displayContent() {
  let coffee = fetchAndDecode('coffee.jpg', 'blob');
  let tea = fetchAndDecode('tea.jpg', 'blob');
  let description = fetchAndDecode('description.txt', 'text');

  let values = await Promise.all([coffee, tea, description]);

  let objectURL1 = URL.createObjectURL(values[0]);
  let objectURL2 = URL.createObjectURL(values[1]);
  let descText = values[2];

  let image1 = document.createElement('img');
  let image2 = document.createElement('img');
  image1.src = objectURL1;
  image2.src = objectURL2;
  document.body.appendChild(image1);
  document.body.appendChild(image2);

  let para = document.createElement('p');
  para.textContent = descText;
  document.body.appendChild(para);
}

displayContent()
.catch((e) =>
  console.log(e)
);
```

    - We see that there are three promises that are being created: coffee, tea, description.
    - By using await in Promise.all() we are able to wait and get all the results of the three promises returned into the values array.
    - This makes asynchronous operations more like a synchronous code.
- The problem with this is that we are making it look synchronous by actually making it into one - each async operation has to wait for the one before to be fulfilled (blocked).

Closures

- Whenever you declare a new function and assign it to a variable, you store the function definition, as well as a **closure**.
- The **closure** is a collection of all the variables in scope at the time of creation of the function.

```
1: function createCounter() {
 2:   let counter = 0
 3:   const myFunction = function() {
 4:     counter = counter + 1
 5:     return counter
 6:   }
 7:   return myFunction
 8: }
 9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

    - When createCounter() is declared, it creates a closure containing the counter.
    - Since createCounter() returns myFunction, which exists in the function scope of createCounter, whenever increment() is being used, it is increasing the counter of the closure associated with the scope of createCounter().
    - Therefore, above code logs 1, 2, 3.
- Closure is used to give access to the outer function's scope from an inner function.

Pass by Value vs. Pass by Reference

- Passing by value happens when assigning primitives while passing by “reference” when assigning objects.
    - Technically, it is still passed by value where the value is the memory address.

```
let a = 1;
let b = a;

b = b + 2;

console.log(a); // 1
console.log(b); // 3
```

```
let x = [1];
let y = x;

y.push(2);

console.log(x); // [1, 2]
console.log(y); // [1, 2]
```

- Using strict comparison operator ===, we can tell the difference even further:
- When comparing primitives, the value is strictly evaluated:

```
const one = 1;
const oneCopy = 1;

console.log(one === oneCopy); // true
console.log(one === 1);       // true
console.log(one === one);     // true
```

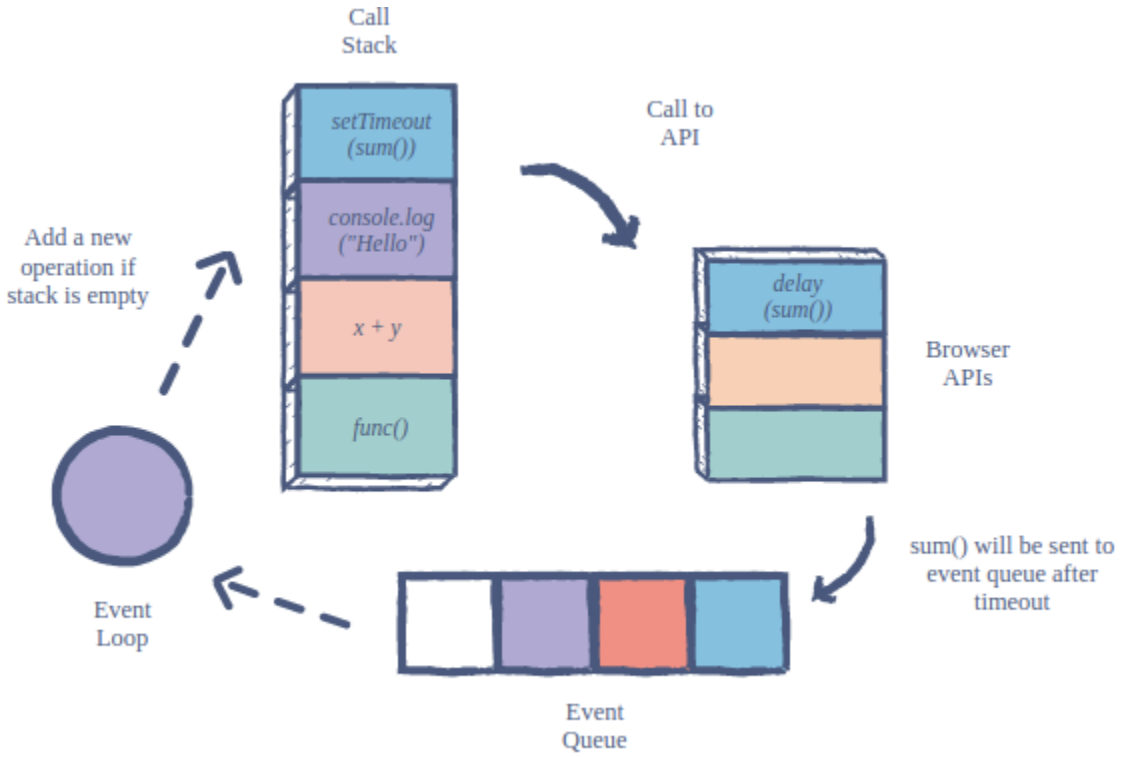
- When comparing objects or any references, their memory address is compared:

```
const ar1 = [1];
const ar2 = [1];

console.log(ar1 === ar2); // false
console.log(ar1 === [1]);  // false

const ar11 = ar1;
console.log(ar1 === ar11); // true
console.log(ar1 === ar1);  // true
```

Event Loop



- The **call stack** is responsible for keeping track of all the operations in line to be executed. Whenever a function is finished, it is popped from the stack.
- The **event (task) queue** is responsible for sending new functions to the track for processing.
    - It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.
- Whenever an async function is called, it is sent to a browser API.
    - These are APIs built into the browser.
    - The API starts its own single-threaded operation to execute the command received from the call stack.
    - After execution, API sends the operation to the event queue.
    - The language itself is single-threaded, but the browser APIs act as separate threads.
    - The **event loop** facilitates this process; it constantly checks whether or not the call stack is empty. If it is empty, new functions are added from the event queue. If it is not, then the current function call is processed.

Functional Programming (FP)

- **Functional programming** is the process of building software by composing **pure functions**, **avoiding shared state**, **avoiding mutable data**, and **avoiding side-effects**.
- Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code.
- **Pure function** is a function which, given the same inputs, always returns the same output, and has no **side-effects**.
- **Shared state** is any variable, object, or memory space that exists in a shared scope, or as the property of an object being passed between scopes.
    - Using shared state could lead to bugs as it may create race conditions.
- An **immutable object** is an object that can't be modified after it's created. Conversely, a **mutable object** is any object which can be modified after it's created.
    - **const** creates a variable name binding which can't be reassigned after creation. const does not create immutable objects.
    - JavaScript has a **.freeze()** method that freezes an object one-level deep, but only top level primitive properties of a frozen object can't change. Any property which is also an object (including arrays, etc…) can still be mutated.
    - In many functional programming languages, there are special immutable data structures called **trie data structures** (pronounced "tree") which are effectively deep frozen.
    - There are several libraries in JavaScript which take advantage of tries, including Immutable.js and Mori.
- A **side effect** is any application state change that is observable outside the called function other than its return value.
    - Modifying any external variable or object property.
    - Logging to the console/ writing to the screen, file, or network.
    - Triggering any external process
    - Calling any other functions with side-effects
- Functional programming is a **declarative paradigm**, meaning that the program logic is expressed without explicitly describing the flow control.
    - **Imperative programs** spend lines of code describing the specific steps used to achieve the desired results.

```
1  const doubleMap = numbers => {
2    const doubled = [];
3    for (let i = 0; i < numbers.length; i++) {
4      doubled.push(numbers[i] * 2);
5    }
6    return doubled;
7  };
8
9  console.log(doubleMap([2, 3, 4])); // [4, 6, 8]
```

- **Declarative programs** abstract the flow control process, and instead spend lines of code describing the data flow.

```
1  const doubleMap = numbers => numbers.map(n => n * 2);
2
3  console.log(doubleMap([2, 3, 4])); // [4, 6, 8]
```

This[16]

- **this** keyword refers to the object it belongs to.
  - The value of **this** is determined by how a function is called (runtime binding). That is, it may be different each time the function is called.

It has different values depending on where it is used:

In a method, `this` refers to the **owner object**.
Alone, `this` refers to the **global object**.
In a function, `this` refers to the **global object**.
In a function, in strict mode, `this` is `undefined`.
In an event, `this` refers to the **element** that received the event.
Methods like `call()`, and `apply()` can refer `this` to **any object**.

- **call()** or **apply()** could be used to set the value of this to a particular value when calling a function.
  - The difference between two is that call() requires the parameters to be listed explicitly using comma (e.g., list all elements in the array explicitly).
  - apply() lets you use an argument (e,g., array directly).

```
// An object can be passed as the first argument to call or apply and this will be bound to it.
var obj = {a: 'Custom'};

// We declare a variable and the variable is assigned to the global window as its property.
var a = 'Global';

function whatsThis() {
  return this.a;  // The value of this is dependent on how the function is called
}

whatsThis();          // 'Global' as this in the function isn't set, so it defaults to the global
whatsThis.call(obj);  // 'Custom' as this in the function is set to obj
whatsThis.apply(obj); // 'Custom' as this in the function is set to obj
```

[16] this - JavaScript | MDN

- **bind()** method is used to set the value of a function's **this** regardless of how it's called.
  - Calling **f.bind(someObject)** creates a new function with the same body and scope as f, but where this occurs in the original function, in the new function this is permanently bound to the first argument of bind, regardless of how the function is being used.

```
function f() {
  return this.a;
}

var g = f.bind({a: 'azerty'});
console.log(g()); // azerty

var h = g.bind({a: 'yoo'}); // bind only works once!
console.log(h()); // azerty

var o = {a: 37, f: f, g: g, h: h};
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

- In **arrow functions**, this retains the value of the enclosing lexical context's this.
  - In global code, it will be set to the global object.

```
// Create obj with a method bar that returns a function that
// returns its this. The returned function is created as
// an arrow function, so its this is permanently bound to the
// this of its enclosing function. The value of bar can be set
// in the call, which in turn sets the value of the
// returned function.
var obj = {
  bar: function() {
    var x = (() => this);
    return x;
  }
};

// Call bar as a method of obj, setting its this to obj
// Assign a reference to the returned function to fn
var fn = obj.bar();

// Call fn without setting this, would normally default
// to the global object or undefined in strict mode
console.log(fn() === obj); // true

// But caution if you reference the method of obj without calling it
var fn2 = obj.bar;
// Calling the arrow function's this from inside the bar method()
// will now return window, because it follows the this from fn2.
console.log(fn2()() == window); // true
```

Hoisting

- **Hoisting** is Javascript's default behavior of moving all declarations to the top of their functional scope.

Array.prototype.map() vs. Array.prototype.forEach()

- The **map** method receives a function as a parameter.
  - Then it applies some kind of operation on each element and returns an entirely new array populated with the results of calling the provided function.
  - This means we can chain another array operation to the map().
  - map() will not mutate the original array.
- The **forEach()** method receives a function as an argument and executes it once for each array element. However, instead of returning a new array like map, it returns undefined.
  - This means we cannot chain array operations as there is no value (undefined).
  - forEach() may mutate the original array depending on the callback (argument).
- Example comparing the two:

```
const myAwesomeArray = [1, 2, 3, 4, 5]
myAwesomeArray.forEach(x => x * x)
//>>>>>>>>>>>>>>return value: undefined

myAwesomeArray.map(x => x * x)
//>>>>>>>>>>>>>>return value: [1, 4, 9, 16, 25]
```

- Mutability example:

```
arr.forEach((num, index) => {
    return arr[index] = num * 2;
});
```

```
// arr = [2, 4, 6, 8, 10]
```

Hosted Objects vs. Native Objects

- **Host Objects** are objects supplied by a certain environment.
- They are not always the same because each environment differs and contains host objects that accommodate execution of ECMAScript.
  - Example, the browser environment supplies objects such as window.
  - Node.js/server environment supplies objects such as NodeList.
- **Native Objects** or **Built-in Objects** are standard built-in objects provided by Javascript.
  - Native objects are sometimes referred to as 'Global Objects' since they are objects Javascript has provided natively available for use.

Function Declaration, Expression, Constructor

- **Function Declaration** declares a function statement (statements perform an action) but does not execute, however, it does get registered into the global namespace.

```
function calcRectArea(width, height) {
  return width * height;
}

console.log(calcRectArea(5, 6));
// expected output: 30

```

- **Function Expression** defines a variable and this variable contains a value reference to the function.
  - Any JavaScript Expressions (including Function Expressions) returns a value.
  - This may also be an **Anonymous function** if no name has been assigned to a function but wrapped in parenthesis to be interpreted as an expression.
  - Function expressions are loaded right away when the interpreter reaches the expression (as opposed to declaration before the code is executed).
  - Benefits may include use in closures, use in arguments to other functions.

```
const getRectArea = function(width, height) {
  return width * height;
};

console.log(getRectArea(3, 4));
// expected output: 12

```

- **Function Constructor** instantiates a new object of the class constructor.
  - A function declaration is just a regular function unless it has been instantiated, it then becomes a class constructor.

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
```

Let, Var, Const

- **var** declarations are globally scoped or function scoped and can be updated and re-declared within its scope.
    - It can be declared without initialization.
- **let** is block scoped and can be updated but not re-declared.
    - It can be declared without initialization.
- **const** is block scoped and can neither be updated nor re-declared.
    - It cannot be declared without initialization.

Event Bubbling and Capturing[17]

```
target.addEventListener(type, listener);
target.addEventListener(type, listener, options);
target.addEventListener(type, listener, useCapture);
target.addEventListener(type, listener, useCapture, wantsUntrusted)
```

- When an event is fired on an element that has parent elements, modern browsers run three different phases:
    1. **Capturing phase**
        - The browser checks to see if the element's outermost ancestor (<html>) has an event handler registered on it for the capturing phase, and runs it.
        - Then it does the same thing, then the next one, and so on until it reaches the direct parent of the element that was actually selected.
    2. **Target phase**
        - The browser checks to see if the target property has an event handler for the event registered on it, and runs it if so.
        - Then, if bubbles is true, it propagates the event to the direct parent of the selected element, then the next one, and so on until the <html> element.
        - Otherwise, if bubbles is false, it doesn't propagate the event to any ancestors of the target.
    3. **Bubbling phase**
        - The browser checks if the direct parent of the element selected has an event handler registered on it for the bubbling phase, and runs it if so.
        - Then it moves on to the next ancestor element and does the same thing, then the next one, and so on until it reaches the <html> element.
- By default, all event handlers are registered for bubbling phase (i.e., useCapture = false)
    - We can prevent bubbling (events being propagated upwards) from a specific ancestor using **stopPropagation()**.

```
video.onclick = function(e) {
  e.stopPropagation();
  video.play();
};
```

[17] Introduction to events - Learn web development | MDN

Same-Origin Policy

- The **same-origin policy** is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin.
    - Two URLs have the same origin if the protocol, port (if specified), and host are the same for both.
- It helps isolate potentially malicious documents, reducing possible attack vectors.
- The same-origin policy controls interactions between two different origins (**Cross-origin**), such as when you use XMLHttpRequest or an <img> element.
    - These interactions are typically placed into three categories:
        1. Cross-origin writes are typically allowed.
        2. Cross-origin embedding is typically allowed.
        3. Cross-origin reads are typically disallowed, but read access is often leaked by embedding.
    - Examples of embedding include <img>, <video>, <object>, <embed>, etc.

Strict Mode[18]

- With "**use strict;**" at the top of your code/function, the JS is evaluated in **strict mode**.
    - Strict mode throws more errors and disables some features in an effort to make your code more robust, readable, and accurate.
    - Catches more common coding errors and prevents "unsafe" actions being taken.
    - It disables features that are confusing or poorly thought out.
- Disadvantages would arise when mixing strict mode with normal mode.
    - If the developer is using normal mode while the library is using strict mode, there may be some actions that the library will not allow, leading to confusion.

== vs. ===

- **Double equals (==)** is a comparison operator, which transforms the operands having the same type before comparison.
    - Compare string with a number, JavaScript converts any string to a number.
- **Triple equals (===)** is a strict equality comparison operator in JavaScript, which returns false for the values which are not of a similar type.

[18] Lucy | JS: use strict

Typing

- Six **Data Types** that are primitives, checked by `typeof` operator:
  - `undefined` : `typeof instance === "undefined"`
  - Boolean : `typeof instance === "boolean"`
  - Number : `typeof instance === "number"`
  - String : `typeof instance === "string"`
  - BigInt : `typeof instance === "bigint"`
  - Symbol : `typeof instance === "symbol"`
- **Structural Types**:
  - Object : `typeof instance === "object"`. Special non-data but **Structural type** for any constructed object instance also used as data structures: `new Object` , `new Array` , `new Map` , `new Set` , `new WeakMap` , `new WeakSet` , `new Date` and almost everything made with `new` keyword;
  - Function : a non-data structure, though it also answers for `typeof` operator: `typeof instance === "function"`. This is merely a special shorthand for Functions, though every Function constructor is derived from `Object` constructor.
- **Structural Root** Primitive:
  - `null` : `typeof instance === "object"`. Special primitive type having additional usage for its value: if object is not inherited, then `null` is shown;

- Note that an instance is **undefined** when it is declared, but not initialized.
- Note that an instance is **null** when it is declared and given an object **null** as its value.
- To check if an instance is a type of a specific object, use **instanceof**.

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
const auto = new Car('Honda', 'Accord', 1998);

console.log(auto instanceof Car);
// expected output: true

console.log(auto instanceof Object);
// expected output: true
```

Destructuring Assignment

```
let a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]

({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20

// Stage 4(finished) proposal
({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}
```

Spread vs. Rest

- **Spread** syntax "expands" an array into its elements, while **rest** syntax collects multiple elements and "condenses" them into a single element.

While vs. Do While

- Use **while loops** when testing for a condition before the first iteration of the loop.

```
> let n = 4
  while (n < 5){
    n++;
    console.log(n)
  }
```

- Use **do…while loops** when testing for a condition after the iteration of the loop.

```
let i = 1;
do {
    console.log("My number is " + i );
    i++;
}
while(i <= 5);
```

```
My number is 1
My number is 2
My number is 3
My number is 4
My number is 5
```

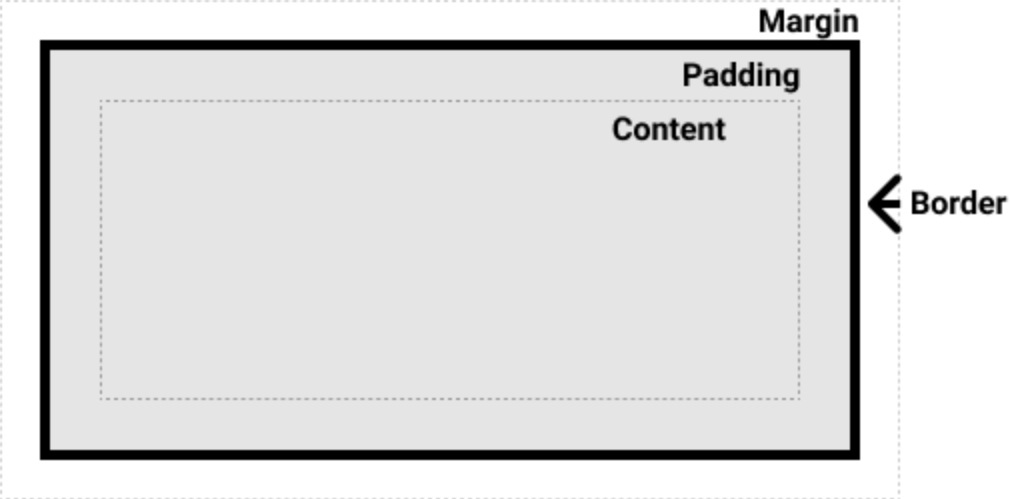
CSS

CSS Specificity

- **Specificity** is a weight that is applied to a given CSS declaration, determined by the number of each selector type in the matching selector.
- There are four categories which define the specificity level of a selector:
    1. **Inline styles**: 1000
        - An inline style is attached directly to the element to be styled.
        - Example: <h1 style="color: #ffffff;">.
    2. **IDs**: 100 for each
        - An ID is a unique identifier for the page elements.
        - Such as #navbar.
    3. **Classes, attributes and pseudo-classes**: 10 for each
        - This category includes .classes, [attributes] and pseudo-classes.
        - Such as :hover, :focus etc.
    4. **Elements and pseudo-elements**: 1 for each
        - This category includes element names and pseudo-elements.
        - Such as h1, div, :before and :after.
- When multiple declarations have equal specificity, the last declaration found in the CSS is applied to the element.

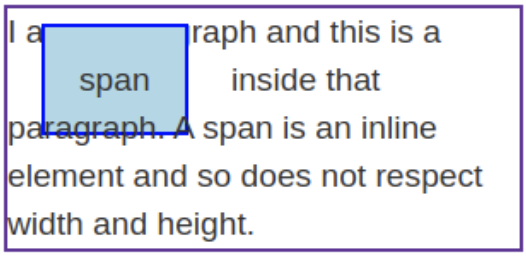
CSS Box

- In CSS we broadly have two types of boxes: **block boxes** and **inline boxes**.
    - Boxes also have an **inner display type** and an **outer display type**.
    - **Outer display type** details whether the box is block or inline.
    - **Inner display type** dictates how elements inside that box are laid out.
        - By default, the elements inside a box are laid out in **normal flow**, which means that they behave just like any other block and inline elements.
        - We can change the inner display type by using display values like flex.
- If a box has an **outer display type of block**, it will behave in the following ways:
    1. The box will break onto a new line.
    2. The box will extend in the inline direction to fill the space available in its container.
        - In most cases this means that the box will become as wide as its container, filling up 100% of the space available.
    3. The width and height properties are respected.
    4. Padding, margin and border will cause other elements to be pushed away from the box.
- If a box has an **outer display type of inline**, it will behave in the following ways:
    1. The box will not break onto a new line.
    2. The width and height properties will not apply.
    3. Vertical padding, margins, and borders will apply but will not cause other inline boxes to move away from the box.
    4. Horizontal padding, margins, and borders will apply and will cause other inline boxes to move away from the box.
- **CSS box model**:

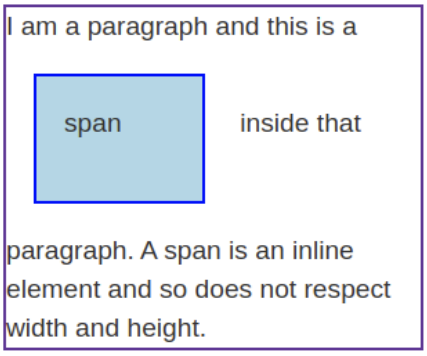


- A key thing to understand about **margins** is the concept of **margin collapsing**.

    - If you have two elements whose margins touch, and both margins are positive, those margins will combine to become one margin.
    - The size of the combined margin is the larger of the margins that were combined.
- Some of the properties can apply to inline boxes too, such as those created by a element.

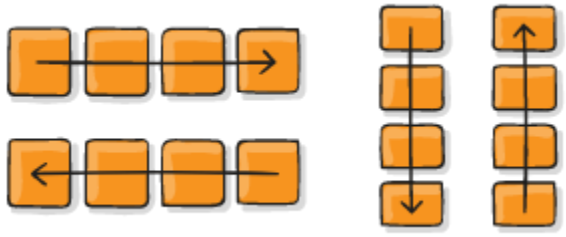


- There is a value of display, **display: inline-block**, which provides a middle ground between inline and block.
    - This is useful when you do not want an item to break onto a new line, but do want it to respect width and height and avoid the overlapping seen above.

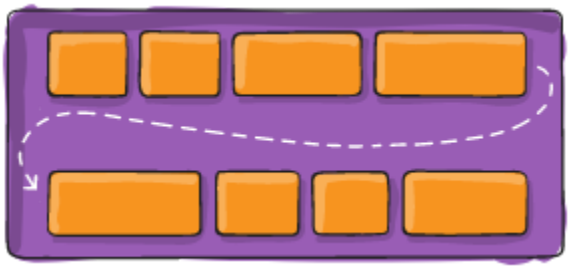


Flexbox[19]

- The main idea behind the **flex layout** is to give the container the ability to alter its items' width/height (and order) to best fill the available space.
    - A flex container expands items to fill available free space or shrinks them to prevent overflow.
- **Flex container (parent) properties**:
    1. **display: flex;**
        - Defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children.
    2. **flex-direction: row | row-reverse | column | column-reverse;**



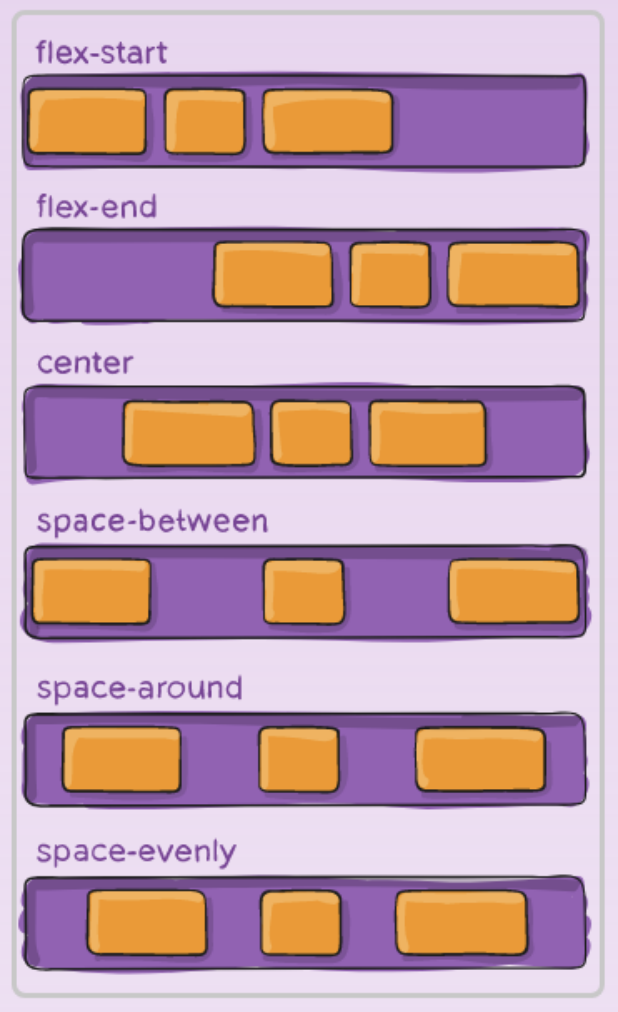
        - This establishes the main-axis, thus defining the direction flex items are placed in the flex container.
    3. **flex-wrap: nowrap (default) | wrap | wrap-reverse;**



        - By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.
    4. **flex-flow: [direction] [wrap];**
        - Combination of flex-direction and flex-wrap.

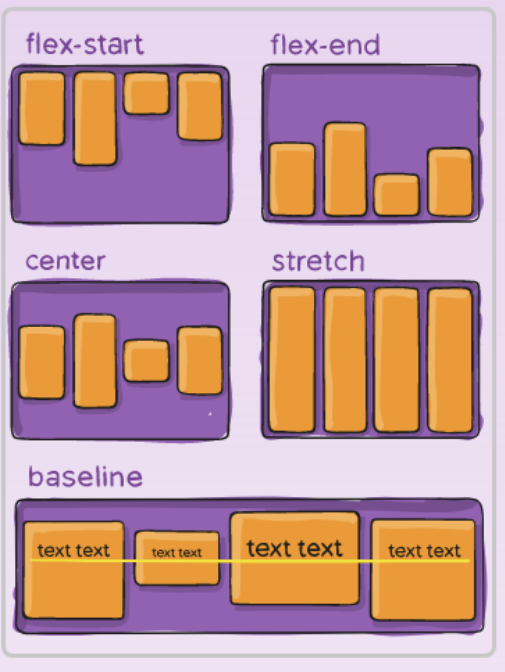
[19] Direct link to the article A Complete Guide to Flexbox

5. **Justify-content**



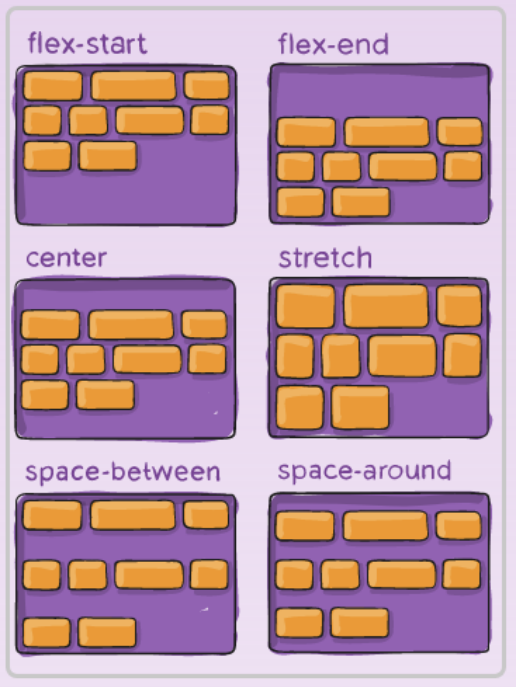
- Note that browser support for these values is nuanced, and some of them are not supported in some browsers.
- The safest values are flex-start, flex-end, and center.
- There are also two additional keywords you can pair with these values: **safe** and **unsafe**.
- Using **safe** ensures that however you do this type of positioning, you can't push an element such that it renders off-screen (e.g. off the top) in such a way the content can't be scrolled too (called "data loss").

6. **align-items**



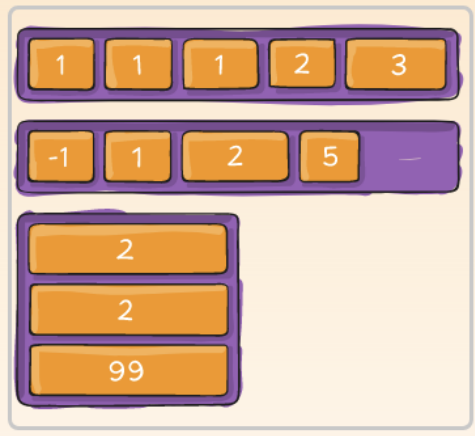
- The **safe** and **unsafe** modifier keywords can be used in conjunction with all the rest of these keywords.

7. **align-content**

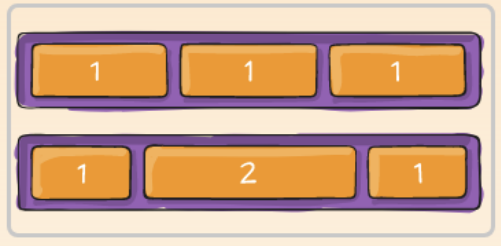


- The **safe** and **unsafe** modifier keywords can be used in conjunction with all the rest of these keywords.

- **Flex item (children) properties**:
    1. **order**

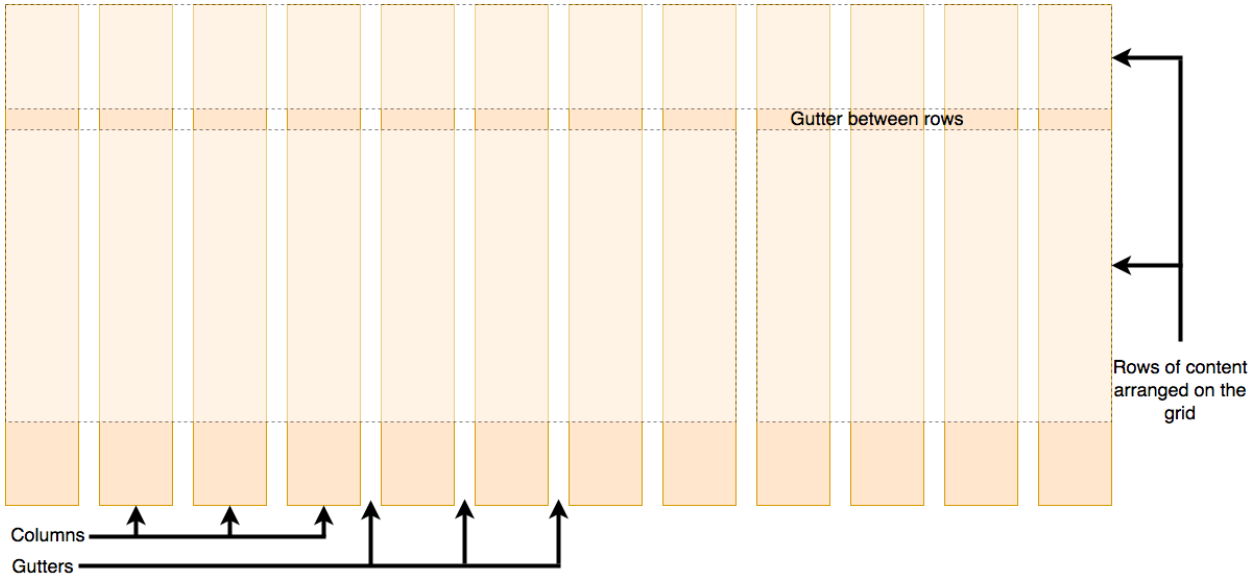


        - Gives an order number to each item determining where they appear.
    2. **flex-grow**



        - This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion.
    3. **flex-shrink**
        - This defines the ability for a flex item to shrink if necessary.
        - Only positive numbers are allowed and it determines how much to shrink.
    4. **flex-basis**
        - This defines the default size of an element before the remaining space is distributed.
        - The **auto** keyword means “look at my width or height property”.
        - The **content** keyword means “size it based on the item’s content”.
            - This keyword isn’t well supported yet.
    5. **flex: none | [<'flex-grow'> <'flex-shrink'>? || <'flex-basis'>];**
        - This is the shorthand for flex-grow, flex-shrink and flex-basis combined.
        - The second and third parameters (flex-shrink and flex-basis) are optional.
    6. **align-self: auto | flex-start | flex-end | center | baseline | stretch;**
        - This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.
        - Uses the same keywords as align-items.

CSS Grid



- A **grid** will typically have columns, rows, and then gaps between each row and column, commonly referred to as gutters.
- To define a grid we use the **grid** value of the **display property**.

```
.container {
    display: grid;
}
```

  - Declaring display: grid gives you a one column grid, so your items will continue to display one below the other as they do in normal flow.

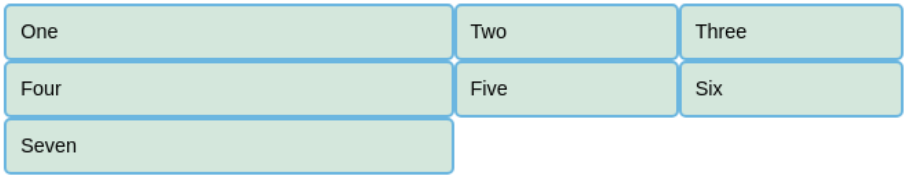
- To make things more grid-like, we will need to add some **columns** to the grid.
  - You can use any length unit, or percentages to create these column tracks.

```
.container {
    display: grid;
    grid-template-columns: 200px 200px 200px;
}
```

One	Two	Three
Four	Five	Six
Seven		

- We can also use the **fr unit** to flexibly size grid rows and columns.

```
.container {
    display: grid;
    grid-template-columns: 2fr 1fr 1fr;
}
```

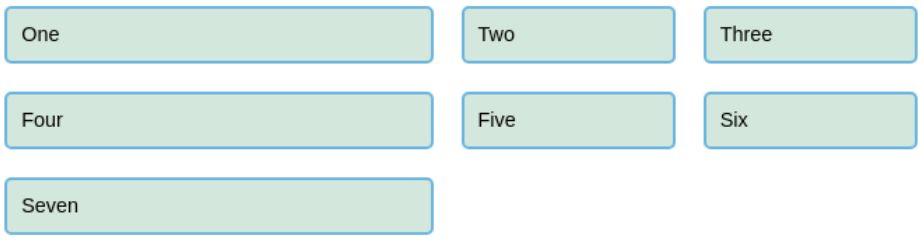


- You can **repeat** all, or a section of your track listing using repeat notation.

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 20px;
}
```

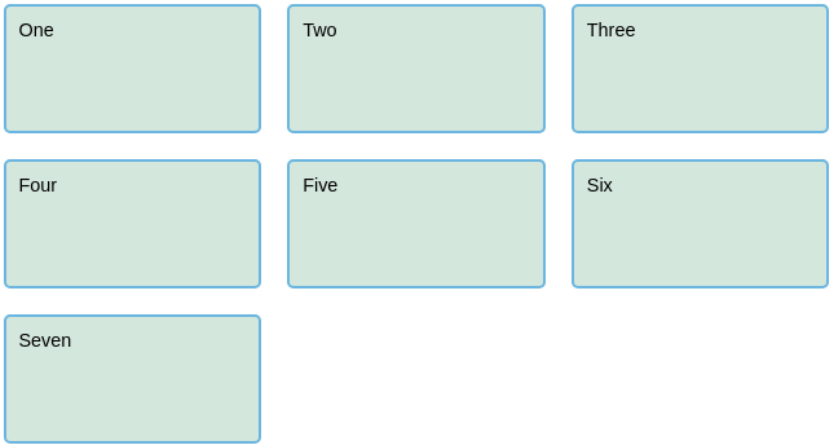
- You can add **gaps** between tracks.
  - We use the properties **column-gap** for gaps between columns,
  - **row-gap** for gaps between rows,
  - **gap** to set both at once.

```
.container {
    display: grid;
    grid-template-columns: 2fr 1fr 1fr;
    gap: 20px;
}
```



- The **explicit grid** is the one that you create using grid-template-columns or grid-template-rows.
- The **implicit grid** is created when content is placed outside of that grid.
    - For example, in all of the examples above, we've only specified columns.
    - Yet, rows are automatically being created - this is an example of implicit grid.
    - By default, tracks created in the implicit grid are **auto** sized (large enough to fit their content). This is adjustable.

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-auto-rows: 100px;
  grid-gap: 20px;
}
```



    - You can also use **minmax([min], [max])** to set the minimum and the maximum size for the track.

- Grid provides an ability to position them based on the **line** (**line-based placement**).
    - Grid always has lines, and where line 1 is placed depends on the writing mode of the document (English have it on the left, while Arabic have it on the right).
    - We can place things according to these lines by specifying the start and end line.
    - We do this using the following properties:
        - **grid-column-start, grid-column-end, grid-row-start, grid-row-end**
    - You can combine above properties using **grid-column** and **grid-row**.
        - We would need to specify start and end using [start]/[end].
        - Note that grid "frameworks" tend to be based around 12 or 16 column grids.

- You can also create **template areas** to position each grid item.
  - You need to have every cell of the grid filled.
  - To span across two cells, repeat the name.
  - To leave a cell empty, use a . (period).
  - Areas must be rectangular — you can't have an L-shaped area for example.
  - Areas can't be repeated in different locations.

```
.container {
  display: grid;
  grid-template-areas:
      "header header"
      "sidebar content"
      "footer footer";
  grid-template-columns: 1fr 3fr;
  gap: 20px;
}

header {
  grid-area: header;
}

article {
  grid-area: content;
}

aside {
  grid-area: sidebar;
}

footer {
  grid-area: footer;
}
```

  - Grid items with specific elements will fill in specific "grid-area" in the div with class .container.

Pseudo-classes[20]

```
selector:pseudo-class {
  property: value;
}
```

- A **CSS pseudo-class** is a keyword added to a selector that specifies a special state of the selected element(s).
- Pseudo-classes let you apply a style to an element not only in relation to the content of the document tree, but also in relation to external factors like…
    - The history of the navigator (:visited, for example),
    - The status of its content (like :checked on certain form elements),
    - The position of the mouse (like :hover, which lets you know if the mouse is over an element or not).

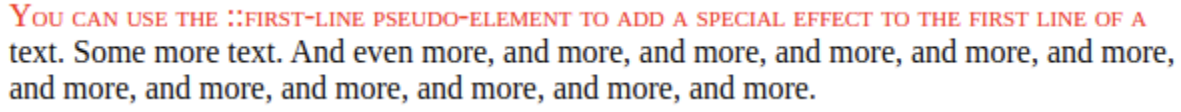
CSS Pseudo-elements[21]

```
selector::pseudo-element {
  property: value;
}
```

- A **CSS pseudo-element** is used to style specified parts of an element.
    - For example, style the first letter, or line, of an element.
    - Insert content before, or after, the content of an element
    - Etc.
- For example, the following style will result in…

```
p::first-line {
  color: #ff0000;
  font-variant: small-caps;
}
```

```
<p>You can use the ::first-line pseudo-element to add a special
effect to the first line of a text. Some more text. And even
more, and more, and more, and more, and more, and more, and more,
and more, and more, and more, and more, and more.</p>
```

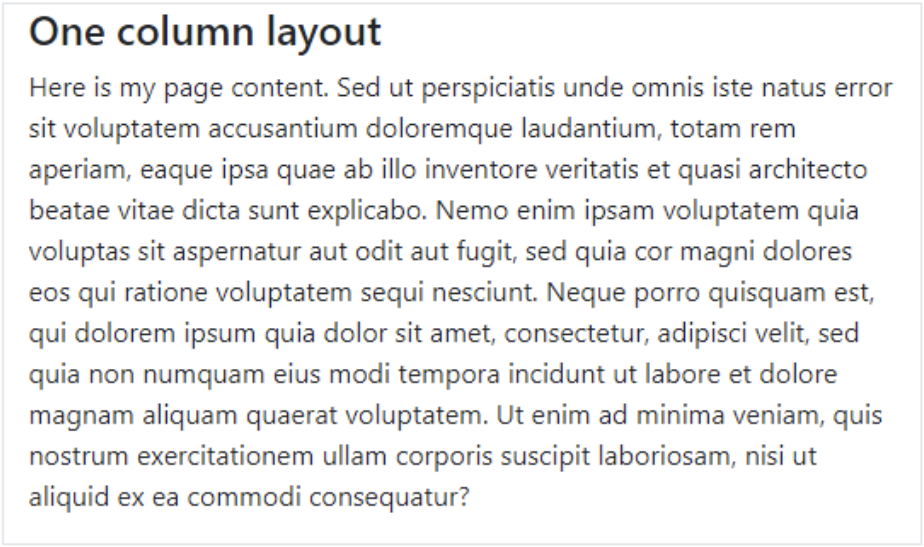


[20] Pseudo-classes - CSS: Cascading Style Sheets | MDN
[21] CSS Pseudo-elements

Bootstrap

- Front-end toolkit that is popular.
- **Bootstrap's grid system** uses a series of containers, rows, and columns to layout and align content. It's built with flexbox and is fully responsive.

```
<div class="container">
   <div class="row">
      <div class="col">I'm your content inside the grid!</div>
   </div>
</div>
```




```
<div class="container">
   <div class="row">
      <div class="col">Left column</div>
      <div class="col">Center column</div>
      <div class="col">Right column</div>
   </div>
</div>
```

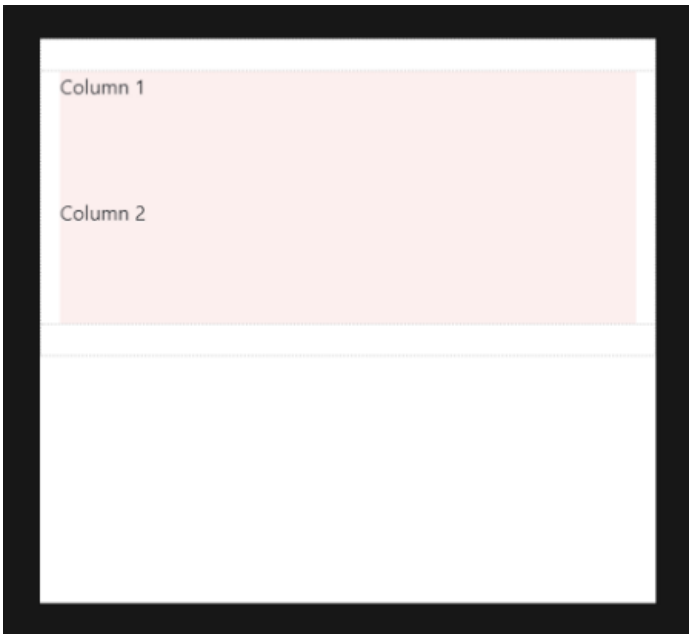
- **Rules of the Grid**:
    1. **Columns** must be the immediate child of a Row.
    2. **Rows** are only used to contain Columns, **nothing else**.
    3. Rows should be placed inside a **Container**.
- **Containers** can be used to hold any elements and contents.
- Columns create horizontal divisions across the viewport called "**gutter**".
    - Classic Bootstrap grid has 12 column units in total, which means the columns can be evenly divided up into factors of 12.

- **Responsive Breakpoints**, based on screen width:
    - (xs) — screen width < **576**px (This is the "default" tier)
    - sm — screen width ≥ **576**px
    - md — screen width ≥ **768**px
    - lg — screen width ≥ **992**px
    - xl — screen width ≥ **1200**px
        - Let's say we have 2 columns, each with 50% width:

```
<div class="container">
   <div class="row">
      <div class="col-sm-6">Column 1</div>
      <div class="col-sm-6">Column 2</div>
   </div>
</div>
```

- The col-sm-6 means use 6 of 12 columns wide (50%), on a typical small device width (greater than or equal to 768 px):

- On less than 768px, the 2 columns become 100% width and stack vertically:



- Larger breakpoints override smaller breakpoints.