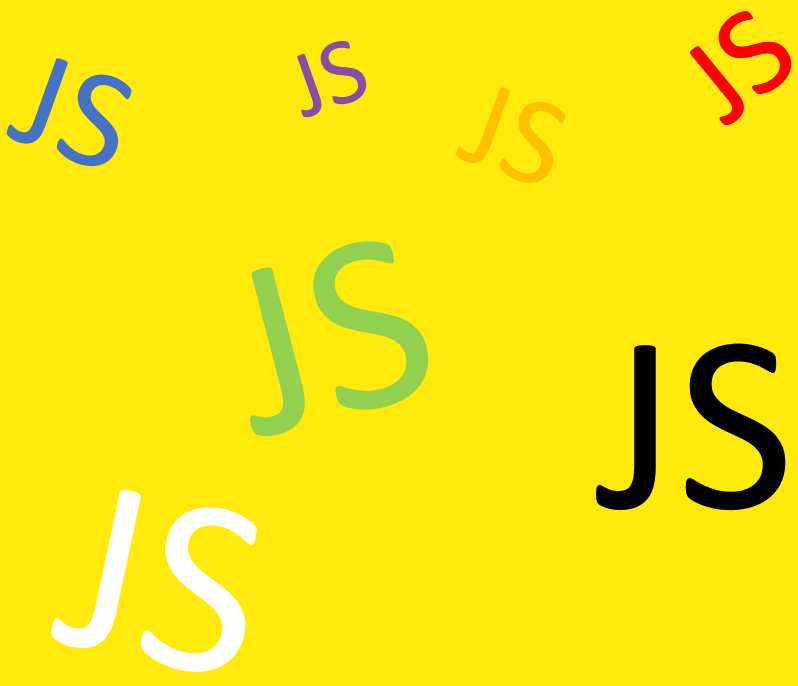
Free Tutorial

Intermediate JavaScript

to master before learning React



Satoshi Yoshida
Simona Yoshida

For more information, email address: react3001@gmail.com

Release History: FIRST EDITION September 2021

All men by nature desire to know.

(Aristotle 384-322BC)

About this book

React (also known as React.js or ReactJS) is a JavaScript library for building UIs. React was created and is maintained by Facebook, a community of individual developers and companies. Facebook, Instagram, Netflix, WhatsApp, Airbnb, and many startup companies are using React.

One of the advantages of React is its flexibility. React could be used for client-side rendering or server-side rendering. Although React is a UI library, many developers consider it a framework because React imposes the component-based architecture that developers must adopt. React has a considerable architectural impact on every aspect of an app that would use them.

Unlike Angular, React is not a complete framework, and advanced features require additional libraries for state management, routing, and interaction with an API. This makes the basic core React simple and the learning curve not so steep. However, depending on the path you take with additional libraries, the learning curve can be very steep.

One of the most important things to understand React is that it is fundamentally JavaScript. It means that the better you are with JavaScript, the more successful you will be with React. Many beginners drop out of React courses because their JavaScript skills are limited.

This tutorial helps you understand intermediate-level JavaScript such as closure, high-order functions, curry, arrow functions, promise, generator, async/await, and other ES6 features required for React.

This free tutorial is extracted from chapters 1 & 2 of the book "React Hooks Redux in 48 hours", released from Amazon.com in July 2021. Chapter 1 is the JavaScript React ecosystem overview. You could skip this chapter if you are busy.

This book is for:

- JavaScript beginners: some experience in Web design using HTML, CSS, and JavaScript.

- JavaScript programmers who do not know closure, functional programming, and ES6.

- Non-Web software engineers or system architects know at least one of the high-level languages.

TABLE OF CONTENTS

1. OVERVIEW

(one hour reading)

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

(René Descartes 1596-1650)

1.1 JavaScript

As React is not a full-fledged framework, it is relatively easy to learn. Instead of working in a preset pattern, a developer can add any library according to their preferences. For example, Redux is widely used by developers as state management.

JavaScript ecosystems come and go. But learning React also means you will have to learn JavaScript. Being a react developer will make you a good JavaScript developer automatically. Even if React became obsolete in the future, your JavaScript skills are still relevant and re-usable.

JavaScript is a dynamically typed programming language, meaning that the variables can be reassigned with a value of a different data type after initialization. JavaScript runs natively in the browser and is quite a strange language compared to current mainstream languages.

JavaScript is a broad language that continually evolves, and it is challenging to keep up with the changes. It is a multi-paradigm language whose syntax is based on the C language family. It supports the following programming paradigms:

- **Functional programming (FP)** –The most common pattern used in recently released React. Functions are objects, giving functions the capacity to hold executable code and be passed around like any other object. React's primary flavor is functional programming, which means that components are built through composition, not inheritance. By default, records (state) are immutable. FP consumes extra memory space. However, as memory costs have dropped drastically recently, it is not a problem any longer.

- **Prototype-based object-oriented programming** –You can use objects without defining a static class. JavaScript is the only mainstream language to support the Prototype-based OOP. As ES6 introduced Classical OOP, you can avoid it. One of the advantages of Prototype-based object-oriented programming is to save memory usage. However, considering memory costs these days, this advantage may not be relevant.

- **Classical object-oriented programming (OOP)** - ES6 introduced Class syntax, which is more understandable to many programmers. However, after Hooks released in 2019, React is shifting towards Functional programming.

- **Procedural programming** - It is still a commonly used technique for low-level, timing-critical embedded firmware. But not recommended to use this pattern in the JavaScript environment.

- **Metaprogramming -** ES6 supports the Proxy and Reflect objects, which allow you to intercept and define custom behavior for fundamental language operations. Metaprogramming is useful when developing a new language based on JavaScript.

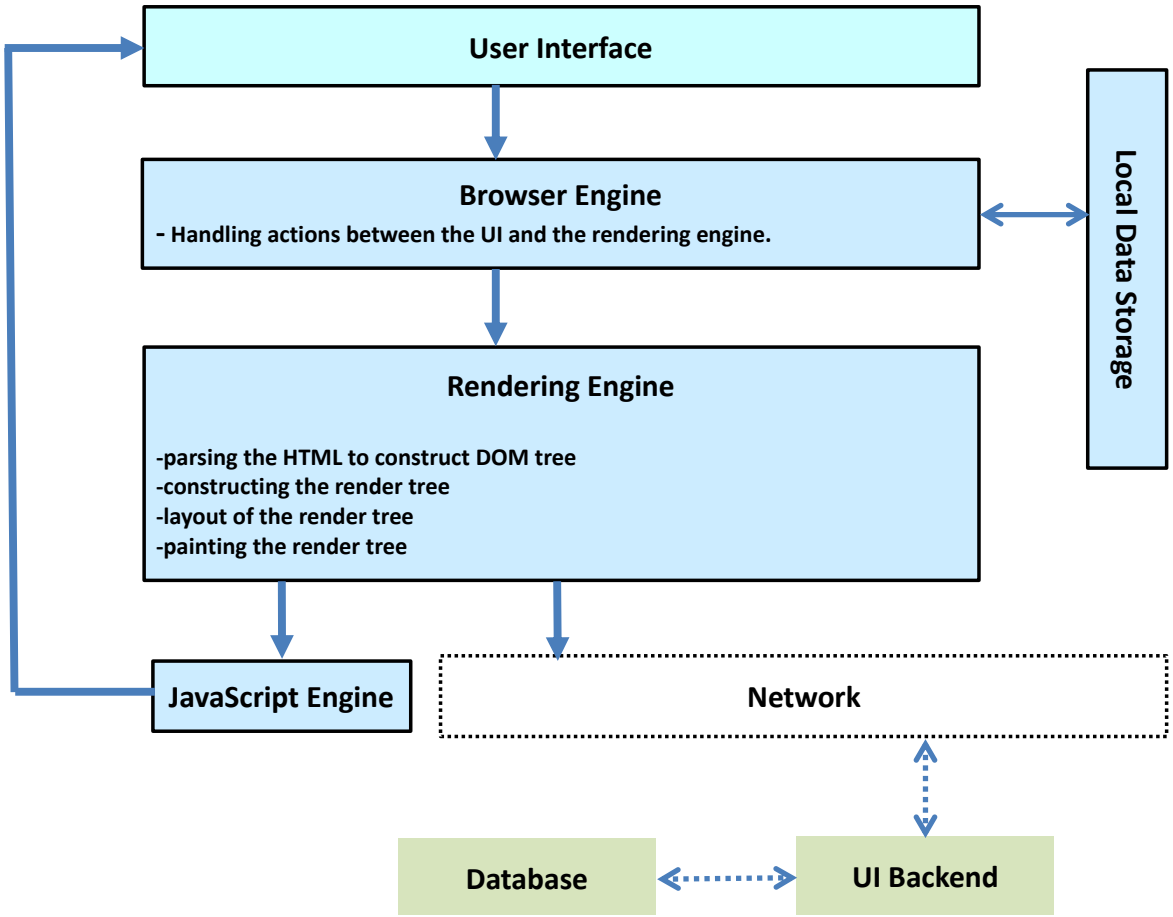
Recently there has been a growing trend toward Functional Programming. With libraries like React/Redux, you will achieve clean software by using immutable data structures. Immutability is a core concept of Functional Programming. Overall, there are many advantages to Functional Programming in JavaScript. Douglas Crockford says that JavaScript is "Lisp in C's Clothing." In fact, JavaScript may be a bit cheaply looking Functional Programming language. React Hooks and Redux are influenced by Functional Programming. The problem is that many programmers who come from an object-oriented background seem to have difficulties in adopting Functional Programming principles in the first place.

React relies heavily on JavaScript. It takes a component's state, JavaScript objects, and using JavaScript functions to manipulate the state. The page is then rendered to reflect those changes. The fact that React relies heavily on plain JavaScript can be good and bad, depending on how strong your JavaScript skills are. The closure, classes, event handling, importing, exporting, callback functions, higher-order functions, curry, etc., are used in React. If you don't know JavaScript, you will get stuck and return to JavaScript tutorials many times. In the end, you might end up wasting your time.

JavaScript Environment in Web

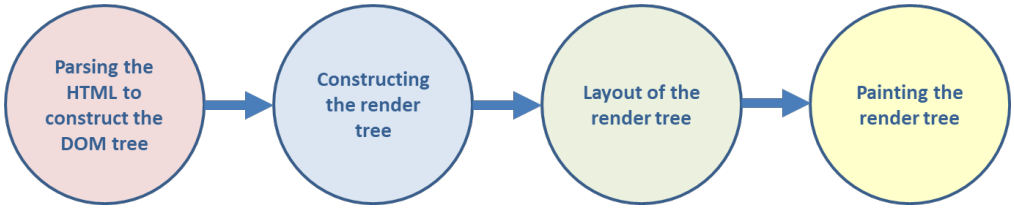
When you are developing web apps, you don't write isolated JavaScript code that runs on its own. JavaScript is for interacting with the user environment, so understanding this environment will allow you to build better apps and be well-prepared for potential issues that might arise once your apps are released commercially.

The following diagram shows the client-side architecture. For the browser, JavaScript is like a machine code that runs under the CPU.



The browser´s main components are:

- **User interface (UI)**: It includes the address bar, the back, forward buttons, text input, radio button, submit button, etc. It is every part of the browser display except for the window where you see the web page itself.
- **Browser engine**: The interactions between the user interface and the rendering engine.
- **Rendering engine**: It is for displaying the web page. The rendering engine parses the HTML and the CSS and displays the parsed content on the screen. The rendering engine receives the contents of the requested document from the networking layer.



- **JavaScript engine**: This is where the JavaScript gets executed. Modern browsers don't use a conventional interpreter. For example, Google Chrome's V8 compiles JavaScript directly to CPU machine code using just-in-time (JIT) compilation before executing it. The compiled code is additionally optimized dynamically at runtime.
- **Networking**: These are network protocols such as XHR requests, which are a platform-independent interface.
- **UI backend**: The backend exposes a generic interface that is not platform-specific. It uses operating system UI methods underneath.
- **Local Data storage**: The apps may need to store all data locally. The supported types of storage mechanisms include localStorage, indexDB, WebSQL, and FileSystem.

ECMAScript 2015 (or ES6) and TypeScript

ECMAScript is a standard that extracted only pure language specification from JavaScript and removed extras features such as DOM. ECMA-262 5th Edition (commonly known as ES5), released in 2009, had been used for a long time. ECMAScript up to ES5 had features that are not often found in other languages. Many developers had been suffering from these features, which were significant barriers for JavaScript novices. However, since the ECMAScript 2015 (ES6) was approved in 2015, it became a modern and practical language. Countless improvements have been made.

ECMAScript (ES2016, ES2017, ES2018, ES2019, and ES2020) has been updated every year. The good news is that no large-scale additions have been made at present because the change from ES5 to ES6 was too drastic. It has been released every year to improve it gradually. This makes it very easy for users to follow the new specifications.

ECMAScript does not have a static type, but the type checking is feasible using a superset of ECMAScript called TypeScript that adds a type definition to ECMAScript. The main benefit of TypeScript is that a compiler detects descriptive errors in case of type violations. However, if you are new to JavaScript, we recommend learning the native JavaScript first to understand JavaScript's runtime behaviors. Since TypeScript is compiled to JavaScript, you still have to learn how JavaScript works anyway.

Much like Java/C#, TypeScript emphasizes a class-based type system, which promotes creating classes with methods that you need to instantiate.

React also have their own way of enforcing type (prop-types to ensure correct types at component interfaces), unit testing for components and logic, so TypeScript is not mandatory. Typescript comes in handy when you develop large-scale apps.

1.2 React Overview

React, created by Facebook, is an open-source JavaScript library for building fast and interactive user interfaces for web and mobile applications. It is a component-based, front-end library responsible only for the application's view layer.

Why using React?

One-way data binding
React imposes a strict one-way data flow to the developer. React makes it possible to structure a project and defines how the data should flow across the UI by enforcing a clear structure and a strict one-way data flow.

A declarative approach to building UIs
React can build UIs a lot more declaratively than with classic HTML templates. You can build Web-based UIs without even accessing the DOM directly.

Component-Based
Components are the building blocks of any React application, and a single app usually consists of multiple components. React splits the UI into independent, reusable parts that can process separately. You can build encapsulated components that manage their own state, then compose them to make complex UIs.

Virtual DOM
The virtual DOM stores a representation of the UI in memory and is synchronized with the actual DOM. The virtual DOM compares the components' previous states and updates only the Real DOM items that were changed, instead of updating all of the components again, as conventional web applications. It makes web applications faster.

Supports both web and mobile apps
A framework called React Native, derived from React itself, can be used for creating mobile applications.

Supports server-side rendering
Server-side rendering renders the React components on the Node.js server. By combining Client-side rendering and Server-side rendering, you can create a universal application.

React Elements

The browser DOM is made up of DOM elements. Similarly, the React DOM consists of React elements. They may look the same, but they are quite different in concept. **React element is a representation of a DOM element in the virtual DOM and is an instruction for how real DOM should be created.**

Let's say there is a somewhere in your HTML file:

We call this a "root" DOM node because everything inside will be managed by React DOM. Applications built with React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, two parameters are required in method **ReactDOM.render([*what*], [*where*]).** A React element using JSX syntax is like below. It looks just like HTML, but there is more than that:

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

<h1>Hello, world</h1> is transpiled to JavaScript code like this:

```
React.createElement("h1", null, "Hello world")
```

React.createElement is a method included in the React library. You can build React apps without using JSX. However, using JSX is strongly recommended for readability and maintenance.

An element is a plain object describing a component instance or DOM node and its desired properties. It contains only information about the component type, its properties, and any child elements inside it. A React element is not an actual instance. It is more like a way to instruct React what you want to see on the screen. You can't call any methods on the element.

React Components

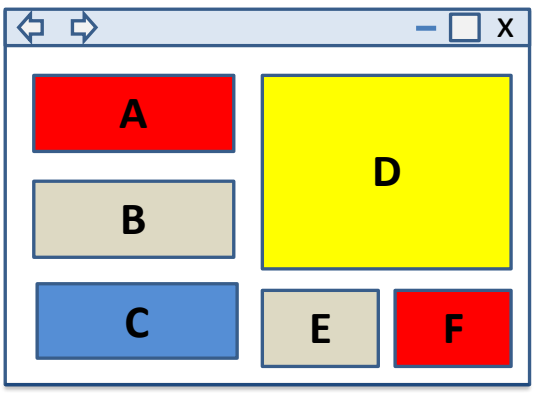
A component is a function or a class that optionally accepts input and returns a React element. A component is an abstraction over an element. They may carry some internal state, and they may behave differently depending on the props they receive. When asked to render, components return elements.

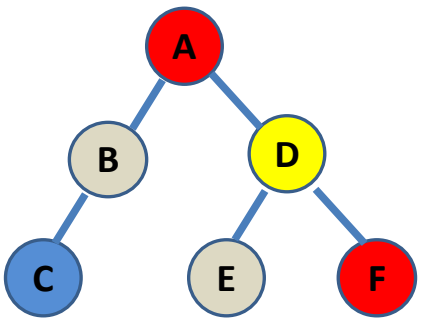
A component is an independent entity that describes a part of your UI. An application's UI can be split into smaller components. Each component has its code, structure, and API. Components encapsulate their state and then compose them to make complex UIs.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs and return React elements describing what should appear on the screen. For example, Facebook has thousands of pieces of components interfaced together when you view their web application. The component-based architecture allows you to think of each block in isolation. Each component can update everything in its scope without being concerned about how it affects other components.

React components are reusable and can be injected into interfaces if necessary. For a better understanding, consider the entire UI as a tree-like structure below. Here the starting component becomes the root, and each of the independent pieces becomes branches, which can be further divided into sub-branches.

React's mechanisms for communicating among components are simple and effective. Props allow data to flow down the component hierarchy from parent to child. When a child wants to talk back up to a parent, a callback function is passed through props.

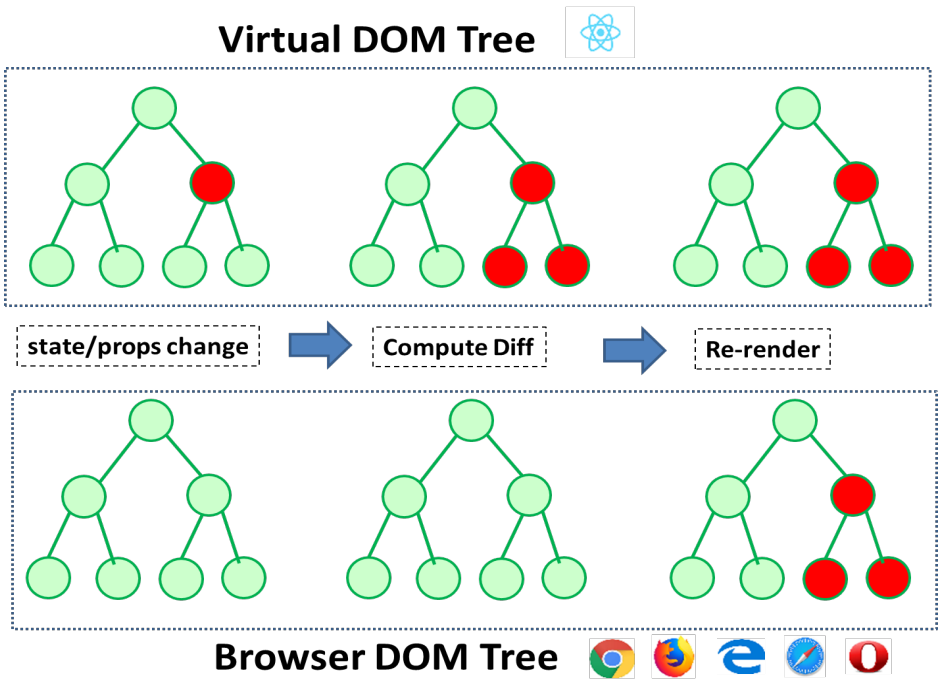




Virtual DOM

In React, instead of directly accessing the real DOM (Document Object Model), it manipulates the virtual DOM. The virtual DOM is a node tree listing of components created in memory. It is a memory area corresponding to the real DOM. The **render( )** method in React creates the Virtual DOM.

Each time the underlying data changes, new Virtual DOM is created, comparing the previous virtual DOM and the current virtual DOM, and only the changes are updated in real DOM. **ReactDOM.render( )** method updates DOM elements that have changed.



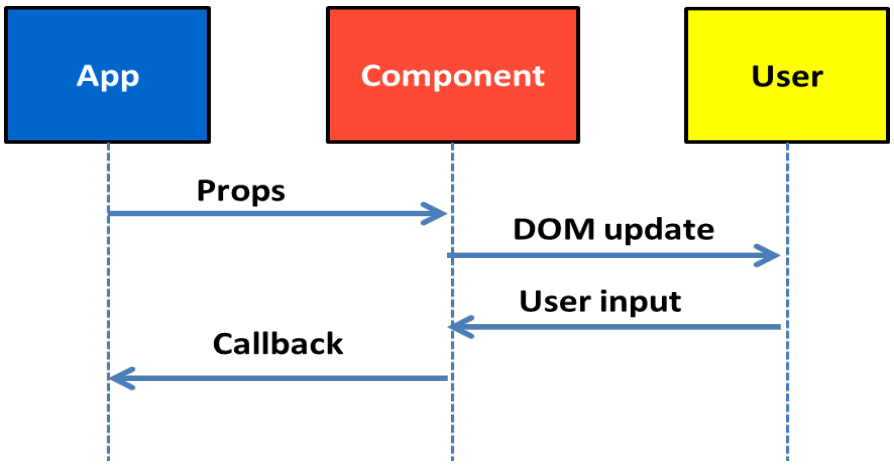
This process that React updates the DOM is referred to as Reconciliation.

How can data be exchanged among components?

In React, there are various ways we can pass data to/from components:

1. Render Props
2. Context API
3. React Hooks (useContext API)
4. React-Redux/Redux
5. Other state management libraries (i.e., XState, MobX)

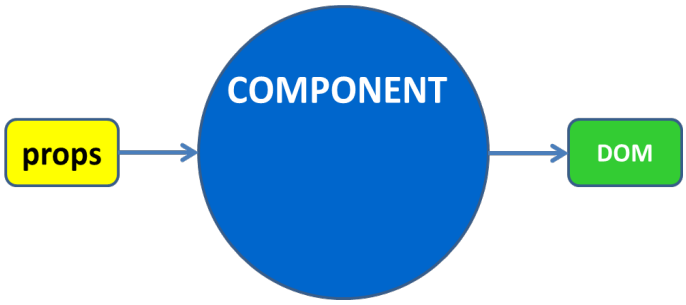
In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props.



Props and State

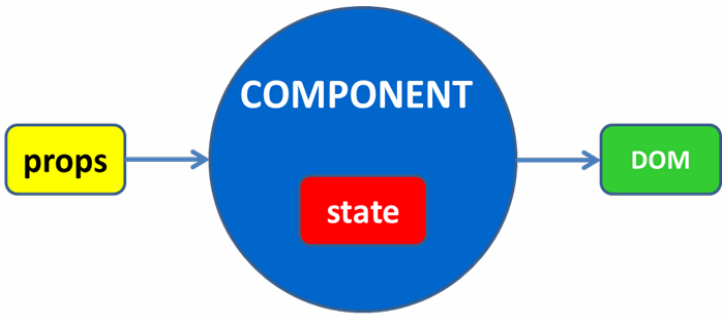
In a React component, *state* holds data that can be rendered to the user. *State* is a structure that keeps track of how data changes over time in your application. *Props* stand for properties and are being used for passing data from one component to another.

Data is updated and manipulated by using *props* and *state*. *Props* and *state* determine how a component renders and how it behaves. If components are plain JavaScript functions, then props would be the function input. A component accepts an input (what we call props), processes it, and then renders some JSX code.



props is passed from a parent component and read-only.

In principle, *props* should be immutable data and top-down direction. This means that the parent component can pass on whatever data it needs to its child component as *props*, but the child component cannot modify its *props*.



State is used for internal communication inside a component.

On the other hand, *state* is an object owned by the component where it is declared. Unlike props, state is local to the component and can only be initialized and updated within the component. **When the state object changes, the component re-render.** When *state* is passed out of the current scope, it is referred to as *props***.** The state of the parent component usually ends up being props of the child component.

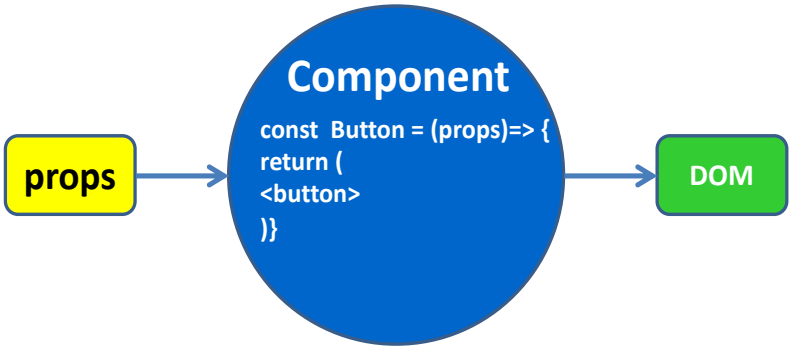
Local State and Global State

Local state encapsulates the dataflow within the React component. In other words, the local state in React holds information in a component that might affect its rendering. The local state is much less manageable and testable in a complicated application.

Global state in React means our state is accessible by every element/component of the application. But the important fact is that it could pollute the whole app since every component can access and update it.

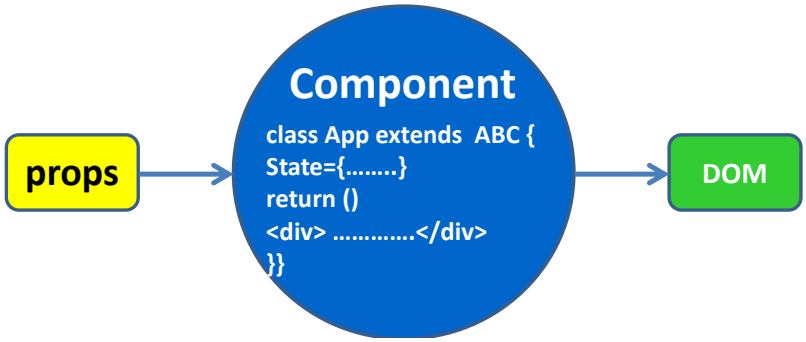
Functional Components

A component is a JavaScript ES6 class or function that optionally accepts inputs (*props*) and returns a React element that describes how a section of the UI should appear. Functional components are just JavaScript functions. They take in an optional input (*props*) and return some JSX directly for rendering. **The Functional component does not support *state***.



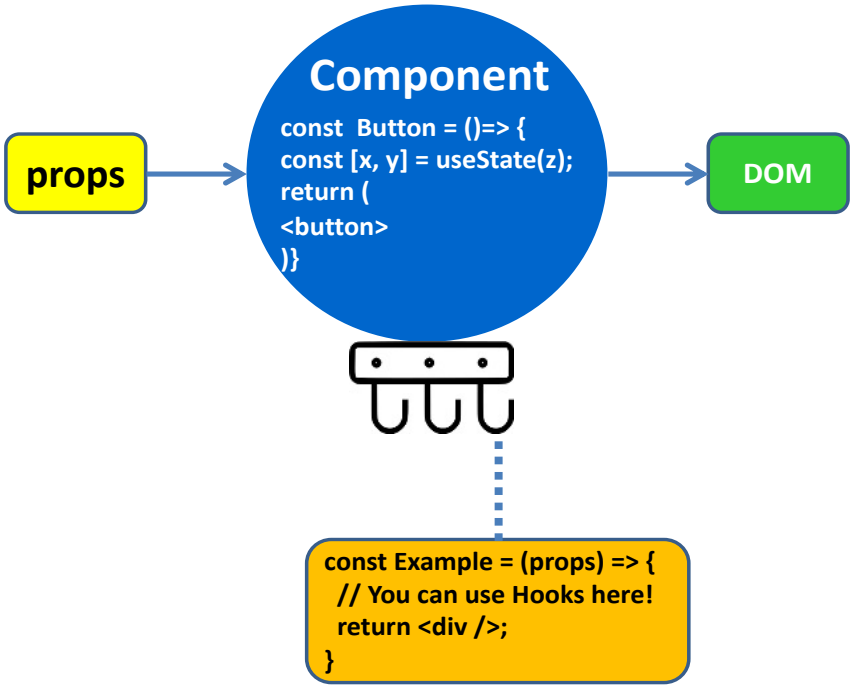
ES6 Class-based Components

ES6 class components have more features. The primary reason to choose Class-based components over Functional components is that they can have *state*. Class components can be used without *state* too. Class components implement a render function to return some JSX.



Functional Components with Hooks

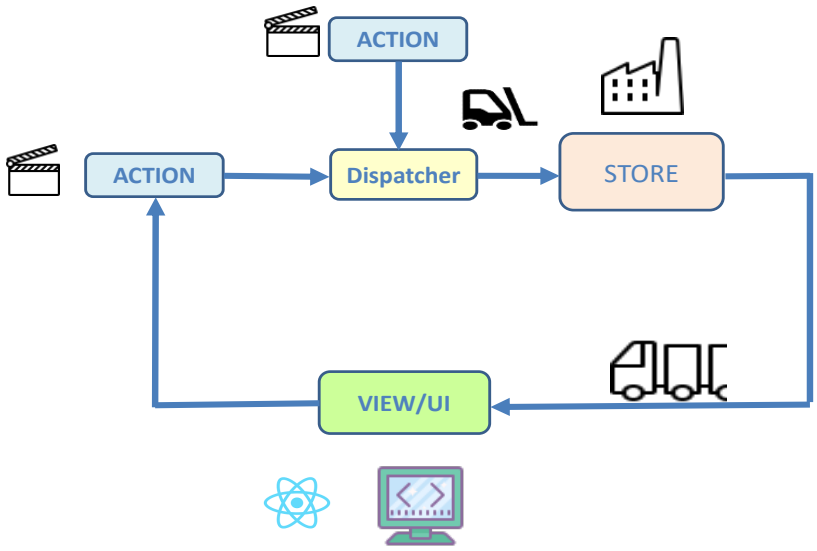
Hooks is a new addition in React 16.8 that lets you use *state* and other React features in the functional component without writing a class component. Hooks brings all previously available features in class components into functional components.



Flux

After having learned the complexity of the MVC (Model View Controller) architecture, the Facebook team developed Flux as an alternative to MVC architecture. Flux follows unidirectional data flow, which supports a composable view of React's components. The Flux architecture is based on the following blocks:

- **Action:** Every change to state starts with an action, which is a JavaScript object describing an event in your application. The actions are typically generated by user interaction or by a server event, such as an HTTP response.
- **Dispatcher**: coordinates actions and updates the store.
- **Store:** manages the application's state and decides how the state will be updated. An application has multiple state portions, and each of them is dependent on the numerous stores of the application.
- **View:** the user interface component, which is responsible for rendering the user interface and handling the user interaction.



Redux

Redux was created by Dan Abramov and Andrew Clark in 2015. Redux is a library based on Flux architecture. The major difference between Flux and Redux is that Flux can include multiple Stores per app, but Redux includes a single Store per app.

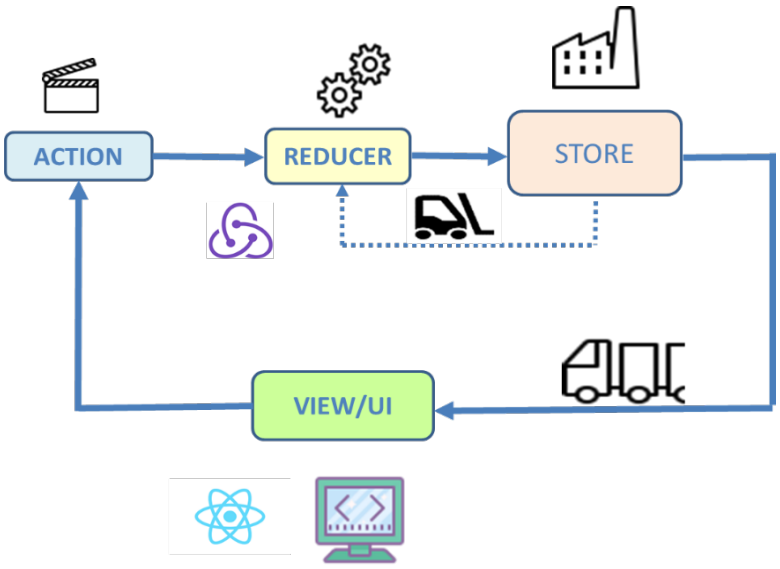
Redux is a predictable state management library to store your app state in one place and manipulate it from anywhere in your app. It helps us write applications that behave consistently, run in different environments (client, server, and native), build more maintainable code, and much easier track down the root cause when something goes wrong. Redux architecture is based on the following components:

- **Action creators:** The actions are typically generated by user interaction or by a server event, such as a server response. The action creators are functions that return actions. Actions are JavaScript objects containing a type and an optional payload. For example, the format of action is like this:
  {**type**: 'ADD_ToDoList', **payload**: 'study Redux'}

- **Reducer**: A reducer is a pure function that determines changes to an application state (based on the new action). It returns a new state object rather than mutating the original state. As our app grows, a single reducer will be split into smaller reducers that manage certain parts of the state tree.

- **Store:** A store is a state container that stores the whole state of the apps in an immutable object tree. Whenever the store is updated, it will update the React components which are subscribed to it. Redux enforces a single global store.

- **View:** The user interface component, which is responsible for rendering the user interface and for handling the user interaction. The views emit actions that describe what happened. Your app reads the state from the store.



As the data flow in redux is predefined, you have to stick with it. In smaller applications, the amount of boilerplate seems to be excessive. On the other hand, in larger applications, the predefined factory-like data flow helps us. With Redux, you have a centralized location where your application's global state is stored, and the state can only be changed by dispatching actions. When an action is dispatched, the state is updated, and components rerender accordingly. It is easy to reason about and to keep track.

You may not always need Redux for every app. If your app consists of a single view, does not save or load state, and has no complicated asynchronous I/O, you may not find a good reason to add the boilerplate of Redux. Some people think that Hooks (useContext and useReducer) will replace Redux. But Redux is application-level global state management. Hooks are mostly for managing state in components themselves, so usages are completely different. As the complexity of an app increases, more data needs to be managed across the app. In many cases, the state management with one global state container can simplify your application.

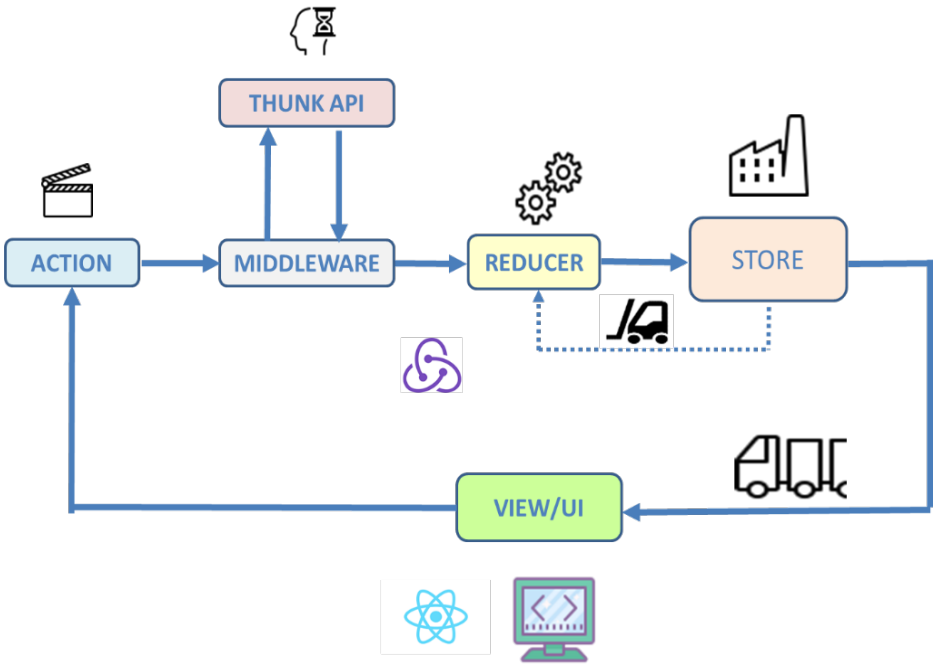
According to the “State of Frontend 2020" survey, 48% of React apps use Redux. If you are working on your own project, Redux is not mandatory for simple react applications. However, if you are working with other React developers or preparing for a job interview, you might as well know Redux, even if you decide not to use it.

Redux-Thunk

By default, actions in Redux are dispatched synchronously, which is a problem for any app that needs to communicate with an external API or perform side effects. Redux-Thunk provides a good way of solving the complexity of asynchronous nature in JavaScript via dispatch chaining.

Thunk is a concept in programming where a function is used to delay the evaluation/calculation of an operation. Redux-Thunk acts as a middleware that allows developers to separate any business logic, like Ajax, data manipulation, or any other asynchronous operations, which may not seem to be appropriate in reducers.

In Redux, actions are defined with simple objects or action creator functions that return an object. **Redux-Thunk lets us call action creators that return a callback function instead of an action object.** The Thunk function receives the dispatch method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have been completed.

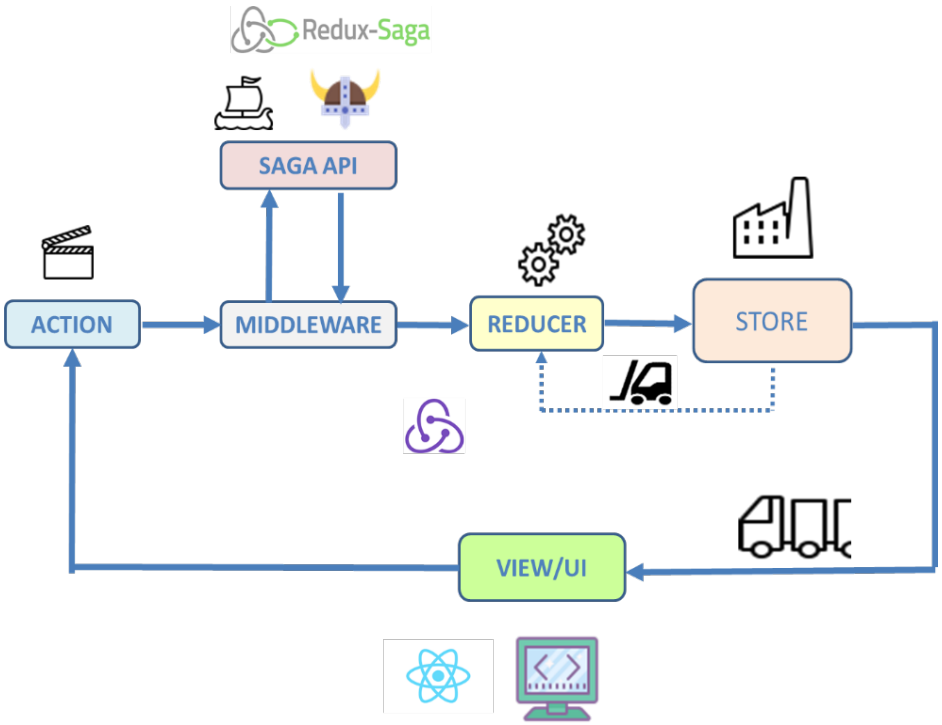


Redux-Saga

Redux-Saga is a library bringing the concept of "task" to Redux. If you have experience in embedded software with an RTOS environment, it is a relatively straightforward concept.

Redux-Saga acts as a middleware that allows developers to separate any business logic, Ajax, data manipulation, or any other asynchronous operations, which may not seem appropriate to processing in reducers directly.

In comparison to Thunk, the benefit of Saga is that we can avoid callback hell. We could avoid passing in functions and calling them inside. Additionally, we can test our asynchronous data flow more easily. Saga can be used to set up complex control flows within our state management system. Saga is suited for complex/large-size applications. The downside is that a bit steep learning curve is expected.



Recoil

Recoil is an experimental state management library introduced by the Facebook team. Just like Redux, Recoil is not an official state management library for React.

Recoil provides several capabilities that are difficult to achieve with React alone. Recoil lets us create a data-flow graph that flows from atoms (shared state) through selectors (pure functions) and down into our React components. Atoms are pieces of React state that can be subscribed to by any component inside the root component. Selectors are pure functions that can receive atoms and selectors and return derived state (computed values or transformation of state). The Recoil hooks are designed to be easily incorporated into custom hooks.

The concept of atoms is to isolate the state in small pieces, making it very flexible. Using selectors, we can easily combine pieces of state and move between being synchronous and asynchronous without modifying the components.

In a nutshell, Recoil is like a global version of React's useState hook. It also supports Concurrent Mode, which is still in the works at the time of writing.

Concurrency in rendering

React, and its libraries are written in JavaScript that runs on the main thread. The main thread also runs DOM updates, browser paints, etc. JavaScript runs on a single main thread, which blocks the react rendering and updating.

In Concurrent Mode, React can work on several state updates concurrently. It means that React can render multiple trees simultaneously, interrupt, delay, or dismiss an ongoing render and defer state updates.

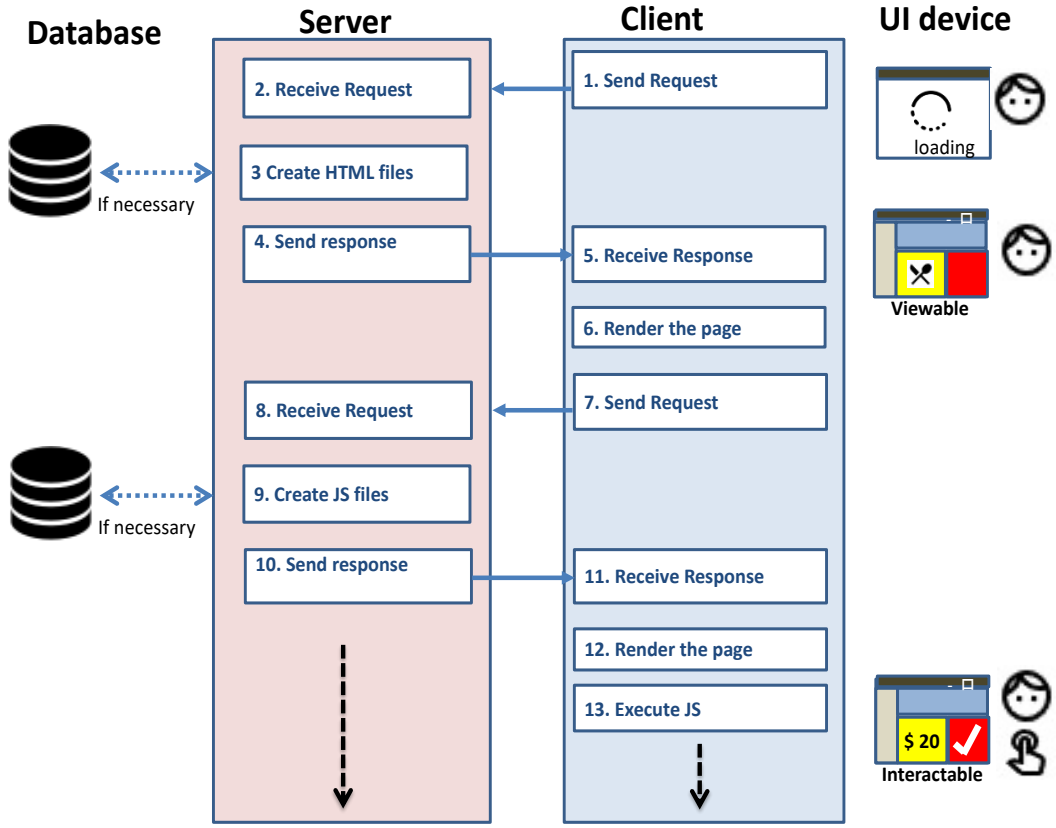
- For CPU-bound updates (such as creating DOM nodes and running component code), concurrency means that a more urgent update can “interrupt” rendering that has already started.
- For IO-bound updates (such as fetching code or data from the network), concurrency means that React can start rendering in memory even before all the data arrives and skip showing empty loading states.

Note: The Concurrent mode is still in the experimental stage at the time of writing in March 2021.

1.3 Server-Client Architecture

Legacy Server Side Rendering (Multiple-Page Application)

Before the time of JavaScript frameworks like Angular, React, Vue…., all the web applications were server-side rendered. When a visitor enters the URL on a browser or visits the website through a link, a request is sent to the Web server. The server gathers the required data through necessary database queries and generates an HTML file for our browser to display. Every page is coming from the Web server, every time we click something, a new request is made to the server, and the browser subsequently loads the new page.



Conventional Server Side Rendering has the following advantages and disadvantages.

PROS

- Advantage in SEO (Search Engine Optimization).
- Initial load duration is shorter.

CONS

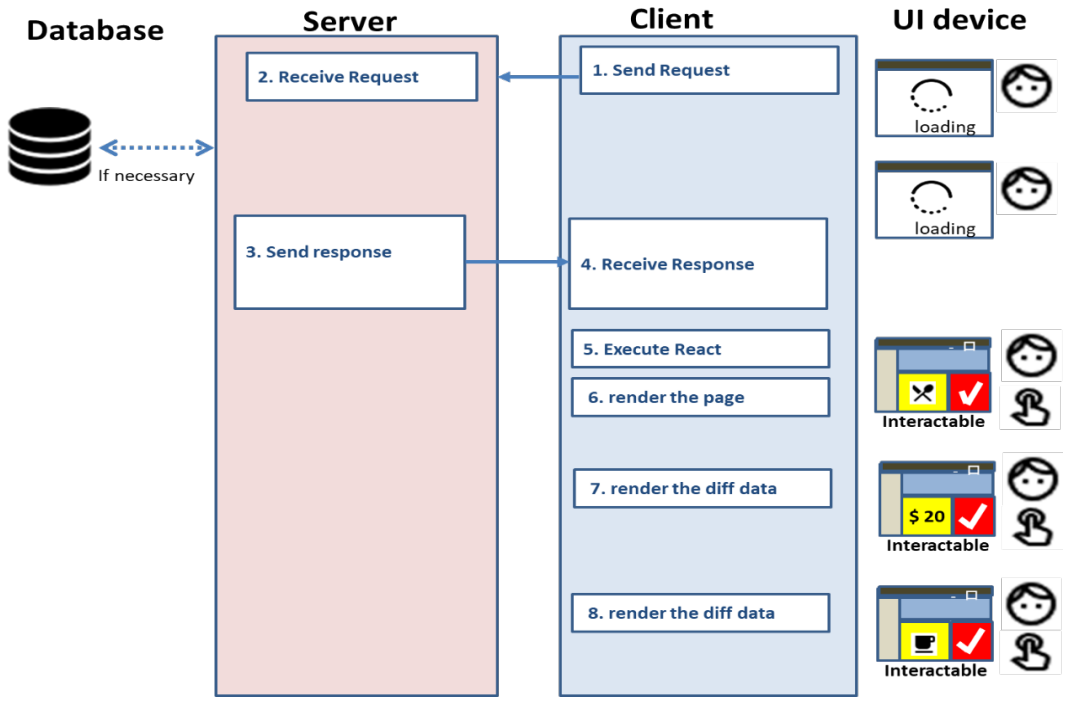
- Second and further page load takes time.
- Difficult to handle heavy user interactions.
- Partial real-time updates can be difficult.
- Synchronous rendering, which blocks other processes.
- In the case of a large number of requests, performance will decrease due to the data communication delay and data traffic.
- Achieving natural UX may be difficult.
- Much longer time to develop mobile applications.

Single-Page Application (SPA)

A Single-Page Application (SPA) is an app that runs under the browser and does not require page reloading during use. For instance, Gmail, Google Maps are typical SPAs.

SPA is just one web page and then loads all other content using JavaScript. SPA can serve an outstanding UX (User Experience) since there are no entire page reloads, no extra waiting period. Rather than re-loading each page entirely, a SPA application loads content dynamically. The fundamental code (HTML, CSS, and JavaScript) of the website is loaded just once.

In the client-side rendering, JavaScript renders the content in the browser. Instead of having the content inside the HTML file, we are getting an HTML document with a JavaScript file that will render the rest of the site using the browser. When we interact with the application, JavaScript intercepts the browser events. Instead of making a new request to the server for a new document, the client requests JSON or process on the server. Since the page that the user sees is not entirely updated, the users feel more like a desktop application.



SPA has the following advantages and disadvantages.

PROS

- Handle heavy user interactions.
- Once the initial load is done, the second and further page loading duration is short.
- Partial real-time updates are possible.
- Asynchronous rendering that does not block other processes.
- Less server-client traffic.
- Excellent UX
- Mobile-friendliness
- Easy to transform into Progressive Web Apps with local caching and offline mode.

CONS

- Disadvantage in SEO (Search Engine Optimization).
- Long initial load duration.
- Performance issues on old mobile devices/slow networks.

Universal Server-Side Rendering (or simply referred to as SSR)

Server-Side Rendering (SSR) is not mandatory to build a React app. React is known as a client-side JavaScript library, but React supports server-side rendering as well. In the conventional SSR, every page is rendered and loaded from the server. However, with the introduction of Universal React SSR, things are different.

A typical pattern in Universal React SSR is that server renders a page and then (re)hydrates client-side DOM through a serialized version of a UI's dependencies. React SSR allows you to pre-render the initial state of your React components server-side. This speeds up initial page loads as users do not need to wait for all the JavaScript to load before seeing the web page.

Since the server renders the initial page and the subsequent pages load directly from the client, you have the best of both worlds — the power of the initial server-side content and the speedy subsequent loads, which requests just the content needed for subsequent requests. Your application becomes more visible for Google crawlers to find your page in addition to the above benefit.

The downside is that implementation can be very complicated. In many cases, you might end up creating systems having the worst of Server Side Rendering and Client-Side Rendering. Therefore, using a framework like Next.js is highly recommended. Next.js for React, Nuxt.js for Vue, Angular Universal for Angular are frameworks using Universal Rendering.

JAMstack and Static Rendering

JAMstack (JavaScript, APIs, and Markup stack) is the revolutionary architecture designed to make the web faster, more secure, and easier to scale. Its deployment doesn't run on a traditional setup of origin servers. Instead, automated deployments are used to push sites directly to CDNs (Content Delivery Network).

With various tools and workflows, pre-rendered content is served to a CDN and made dynamic pages through APIs and serverless functions. Technologies in the JAMstack include JavaScript frameworks, Static Site Generators, Headless CMSs, and CDNs.

The static site generators generate the content during build time. Then, once loaded, React takes over, and you have a single-page application. Static rendering achieves a consistently fast Time-to-First-Byte since the HTML for a page doesn't have to be generated on the fly. With HTML responses being generated in advance, static renders can be deployed to multiple CDNs.

Gatsby and Next.js (version 9.3 onwards) are the popular frameworks supporting the Static Site Generator.

JAMstack official docs: https://jamstack.org/

1.4 Mobile Apps

React Native

React Native is an open-source mobile application framework created by Facebook. It is for developing applications for Android and iOS. Developers can use React along with native platform capabilities.

The working principles of React Native are identical to React. The significant difference is that React Native does not manipulate the Browser DOM. React Native invokes Object-C APIs to render to iOS components or Java APIs to render to Android components instead of rendering to the Browser DOM. This is possible because of the bridge, which provides React with an interface into the host platform´s native UI elements.

React components return markup from their render function, which describes how they should look. With React for the Web, this translates directly to the Browser DOM. As for React Native, this markup is translated to suit the host platform, so a <View> becomes an iOS-specific UI component or Android-specific UI component. React components wrap existing native code and interact with native APIs via React's declarative UI and JavaScript.

If you are comfortable with Web-based React, learning React Native is easy. Thousands of apps are using React Native.

React Native official site:
https://reactnative.dev/

PWA (Progressive Mobile Application)

PWA is considered to be the future of multi-platform development because of its application on several devices, improved speed, and requires no installation. It is intended to work on any platform that uses a standards-compliant browser, including both desktop and mobile devices.

PWA is delivered through a browser. It behaves like a native application, offering gestures and app-like style, but based on the Web. PWA is responsive and can be browsed on any mobile device, desktop, or tablet. Twitter, Uber, Instagram, Financial Times, Forbes, and many organizations use the PWAs.

PWA is not a specification but a design pattern. There is no official document.
There are three things you need to provide before your site turns into a valid PWA:

- S**ervice worker:** A service worker is a JavaScript file that runs in the background, separately from the main browser thread, intercepting network requests, caching or retrieving resources from the cache, and delivering push messages.
- **JSON manifest file:** The JSON file contains information on how your PWA should appear and function.
- **HTTPS:** PWAs only work on secured connections.

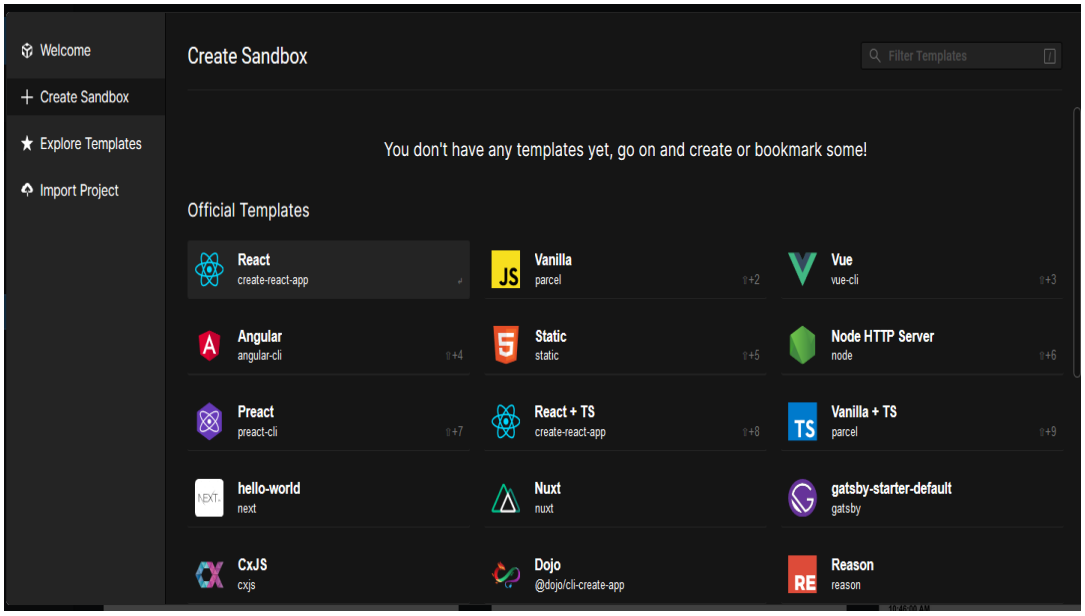
If your goal is to target mobile devices (Android and iOS) only via the app store, use React Native. If you want to reach a broader range of users and devices via the browser, go with PWAs. If you wish for the lowest-cost option for your development team, then PWAs wins.

1.5 Tools used in this book

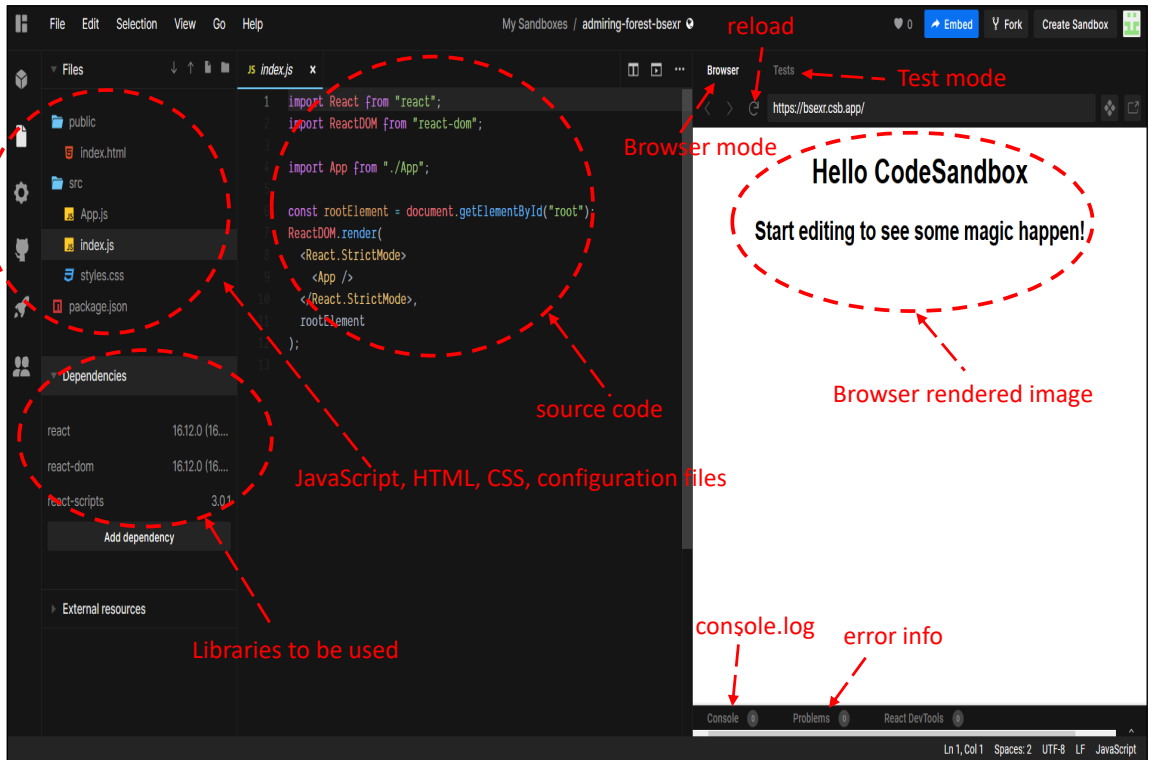
CodeSandbox

CodeSandbox is a free Online editor/playground/Cloud IDE tool for JavaScript framework/library. CodeSandbox allows developers to focus on writing code while it handles all the necessary background processes and configurations. CodeSandbox focuses on building and sharing code demos that contain back-end components. The editor is optimized to analyze npm (Node Package Manager) dependencies, show custom error messages, and also make projects searchable by npm dependency.

CodeSandbox allows developers to go to a URL in their browser to start building. This makes it easier to get started and makes it easier to share with other developers. You can share your created work by sharing the URL. Others can then further develop on these sandboxes without downloading. You can also convert your repository (or its directories) to a CodeSandbox. Thanks to CodeSandbox, we can focus on the code without worrying about installation, build tooling, configuration.



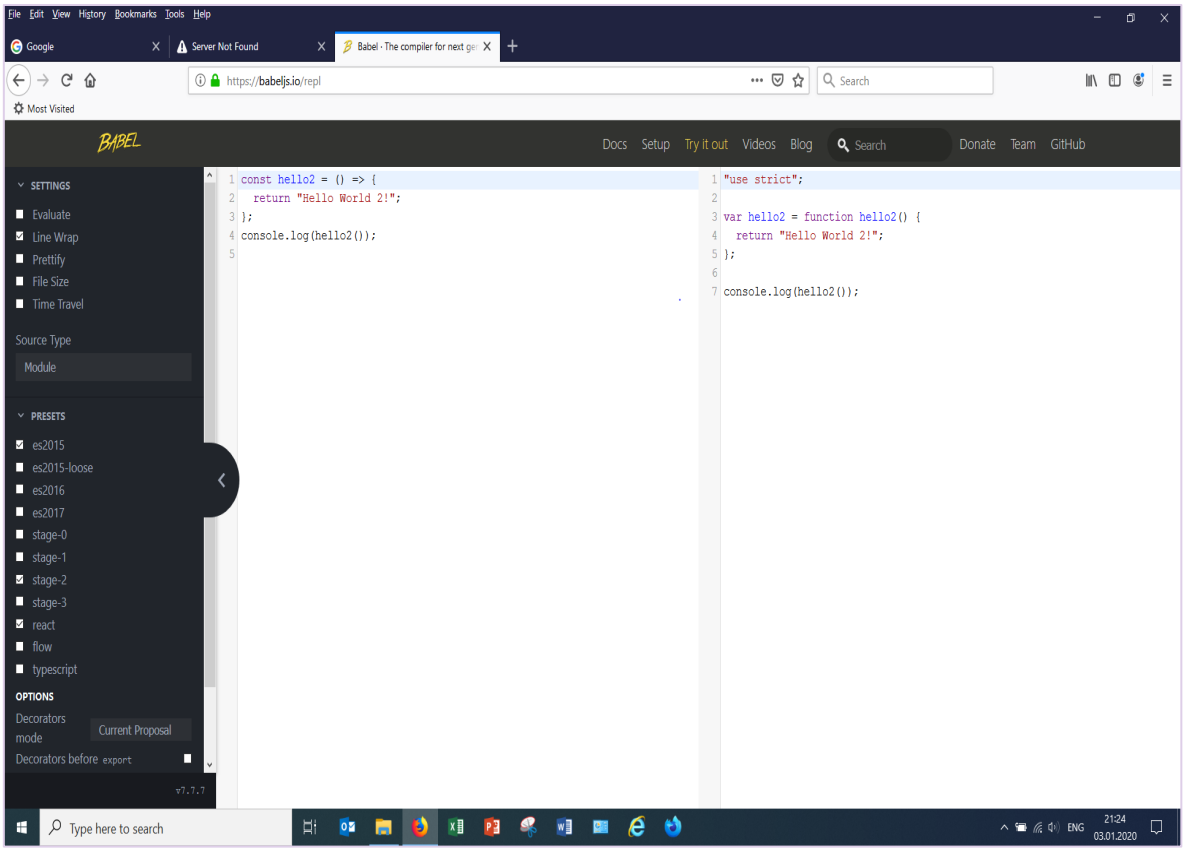
The following is an example of the “Hello CodeSandbox” with React. Please play around by modifying files - index.html, styles.css, index.js, and App.js.



Code sample: https://codesandbox.io/s/react-template-h9y7w

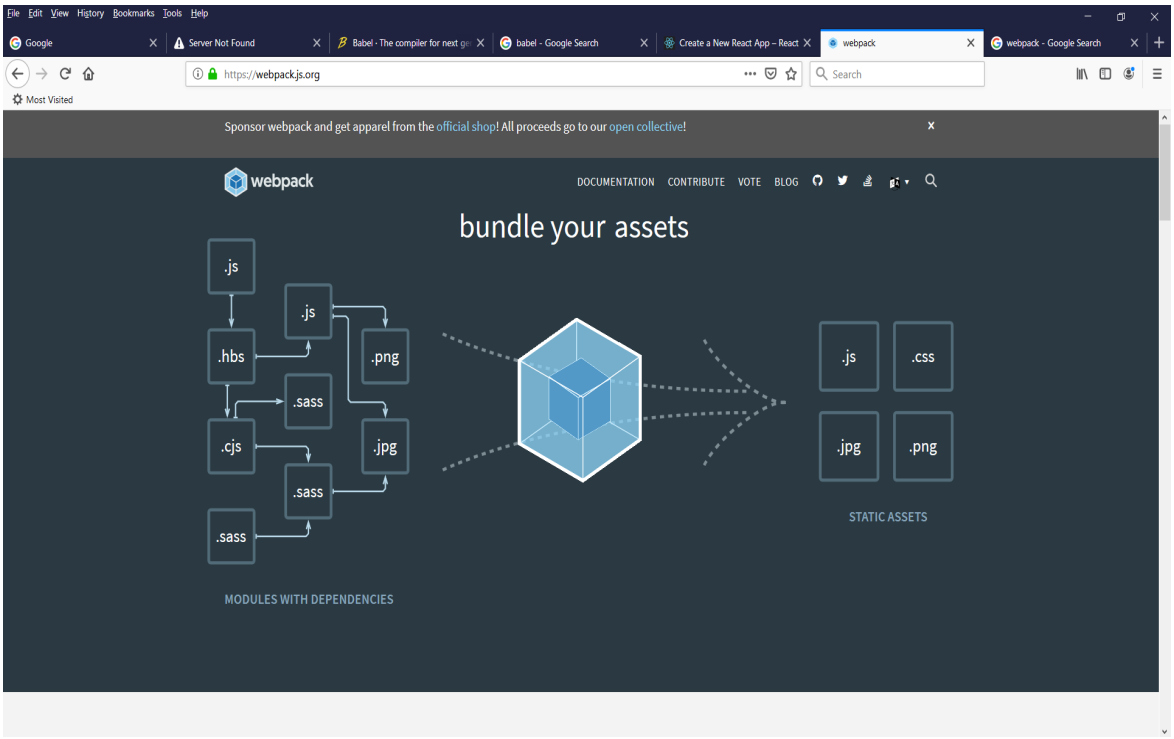
Babel

Babel is a free and open-source JavaScript transpiler, mainly used to convert ES6+, JSX, or TypeScript code into a backward-compatible version of JavaScript that can be run by older JavaScript engines. You don’t have to install Babel manually when using CodeSandbox.



Webpack

Webpack is an open-source module bundler primarily for JavaScript, but it can transform front-end assets like HTML, CSS, and images if the corresponding loaders are included. Webpack takes modules with dependencies and generates static assets representing those modules. You don't have to install Webpack manually when using CodeSandbox.



2. INTERMEDIATE JAVASCRIPT

(6 hours reading and exercise)

Common sense is not as common.

(Voltaire 1694-1778)

This chapter is not a complete tutorial on all features of the JavaScript language. Instead, it will outline a set of fundamental principal that require for understanding React ecosystems.

JavaScript is not a pleasant language by any standard. The good news is that you don't have to master the whole ECMAScript spec before learning React. You might as well focus on learning the minimum things, which are covered in this chapter.

If you don't know JavaScript at all, reading the following JavaScript tutorial is recommended.

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

We would also recommend "YOU DON´T KNOW JS series Up & Going " O'Reilly (Kyle Simpson). A free PDF version may be available depending on your area.

If you are comfortable with advanced JavaScript such as Closure, Currying, Higher-Order function, and ES6, you might as well skip this chapter.

2.1 Data Types

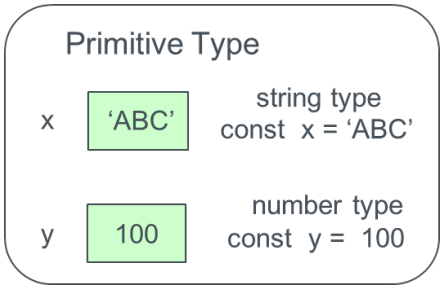
There are two kinds of JavaScript data types - primitive data type and reference data type.

Primitive data type

JavaScript has several data types passed by value: Boolean, String, Number, Null, Undefined, and Symbol. Those are referred to as **primitive data types**.

Primitive data Types	Descriptions	Examples
Boolean	true or false	var myBoolean = true
String	A character or sequence of characters placed inside quote marks (‘ ’) or (“ “)	var myString = ‘hello world’
Number	Any integer or floating point numeric value.	var myNumber = 12345
Undefined	A declared variable has no assigned value.	var myUndefined = undefined
Null	A lack of identification, indicating that a variable points to no object.	var myNull = null
Symbol (ES6)	A symbol is a unique and immutable data type that is often used to identify object properties. Every symbol value returned from Symbol( ) is unique. A symbol value may be used as an identifier for object properties.	var mySymbol1 = Symbol(sym) var mySymbol2 = Symbol(sym)

In the primitive data type, the values themselves are stored in memory directly.

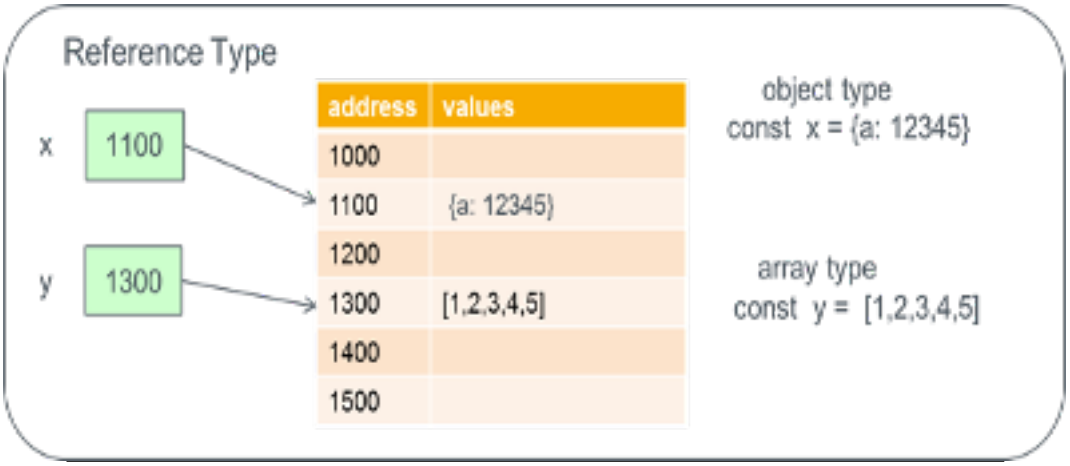


Reference data type (Complex data type or simply called Object)

Reference data types are passed by reference. The variables don't actually contain the value. **The variables store their reference values (memory locations that actually hold values) in memory.** In other words, the reference data type points to the object's location in memory. Many novice JavaScript programmers seem to misunderstand the Reference data type.

Reference data types	Descriptions	Examples
Object	An unordered list of properties consisting of a name and a value.	var myObject = {x:2, y:1, z:3}
Function	Enclosing a set of statements. Functions are the fundamental modular unit of JavaScript. They are used for code reuse, information hiding, and composition.	var myFunc = function (x, y){ return x +y}
Array	Type of structure representing block of data (numbers, strings, objects, functions, etc…) allocated in consecutive memory.	var myArray = ['USA', "India', 'UK', 'Russia']
Map (ES6)	A data collection type, in which data is stored in a form of pairs, which contains a unique key and value mapped to that key. Because of the uniqueness of each stored key, there is no duplicate pair stored.	new Map([iterable object])
Set (ES6)	An abstract data type which contains only distinct elements/objects without the need of being allocated orderly by index.	new Set([iterable object])
Error	A run time error	
Date	A date and time	var myDate = new Date()
RegExp	Regular expression	var myPattern = /pattern/i

In the reference data type, the values point to the object's location (address) in memory. In this diagram below, *x* contains address 1100, and *y* contains address 1300. These are only examples to understand the reference data type.



Note: Reference data types are all technically "object," and we refer to them collectively as "object". In a broader sense, all reference data types are considered as "object". However, depending on the context, Function, Array, Map, Set….are not "Object". In this narrow context, an "Object" indicates a reference data type with an unordered list of properties consisting of a name and a value.

Strings & Literals

When writing strings in JavaScript, we normally use single (‘) or double (“) quotes:

var str1 = 'hello string';
var str2 = "hello string";

Template Literal (ES6)

Template literals are indicated by enclosing strings in backtick characters (`) and provide syntactic sugar for concatenating strings, multi-line strings, and line breaks. Template literals are still a string. Template literals improve readability.

```
let a = {
  title: "Dr.",
  name: "Ada Lovelace",
  amount: "100"
};

console.log(`
Dear ${a.title} ${a.name},

I am apologetic about the bad service that you have experienced at our restaurant last week.
I would like to gift you a $${a.amount} check for your continued association with us, and as a
way to make an apology.

Sincerely,
`);
```

Code sample: https://codesandbox.io/s/js-tmp-literal-jmxl2

Variables and constant

Variables are used in programming languages to refer to a value stored in memory. Variables in JavaScript do not have any specific type. Therefore, JavaScript is referred to as an untyped language or dynamic language. Once you assign a specific literal type to a variable, you can later reassign the variable to use for any other data type.

Interestingly, *var* declarations are globally scoped or function scoped. On the other hand, *let* and *const* are block-scoped.

var
var is a variable that can be updated and redeclared within its scope. Surprisingly there is no block scope. You can duplicate *var* with the same variables name. However, in ES5 strict mode, duplication is not allowed.

```
var myage = 10;
var myage = 20;    //duplication  allowed.  The last var is valid.
```

let (ES6)
let variables can be updated but not redeclared. It is block-scoped and cannot be used until declared. You cannot duplicate *let* with the same variables name.

```
let myage = 10;
let myage = 20;    //error !  duplication not allowed
```

If data is a reference data type, such as an array, an object, or a function, you can still change its contents.

```
let myarray = [1,2,3,4,5]
  myarray[2] = 100;    //[1,2,100,4,5]
```

const (ES6)
const behaves in a very similar way to *let*. It is also block-scoped and cannot be used until declared. Just Like *let*, you cannot duplicate *let* with the same variables name.

```
const myage = 10;
const myage = 20;    //error!  duplication not allowed
```

Just like *let*, if data is a reference data type, such as array, object, or function, you can still change its contents.

```
const myarray = [1,2,3,4,5]
 myarray[2] = 100;    //[1,2,100,4,5]
```

If data is a primitive type, *const* variables can neither be updated nor re-declared.
On the other hand, *let* variables can be updated but not re-declared.

```
let myage1 = 10;
myage1 = 20;        // updating primitive type variable allowed

const myage2 = 10;
myage2 = 20;        // error!  updating primitive type variable not allowed
```

Block scope of var, let, and const

Block scope controls the visibility and lifetime of variables and parameters. Variables declared using *var* do not have block scope, while variables declared using *let* and *const* have block scope. Anything within curly braces { } is a block. A variable declared in a block with *let* or *const* is only available for use within { }.

```
//====== no block scope with var ============
var myage = 1;

if (myage === 1) {
  var myage = 2;

  console.log(myage);
  // expected output: 2
}
console.log(myage); //expected value: 2

//========block scope created by let ============
var myage1 = 10;

if (myage1 === 10) {
  let myage1 = 20;

  console.log(myage1);
  // expected output: 20
}
console.log(myage1); // expected output: 10
```

2.2 Expressions and Statements

Any unit of code that can be evaluated to a value is referred to as an expression. Since they produce values, expressions can appear anywhere in a program where JavaScript expects a value, such as the arguments of a function invocation.

A statement is an instruction to perform a specific action that includes creating a variable or a function, looping through an array of elements, evaluating code based on a specific condition, etc. A statement is a standalone unit of execution. It does not return anything. For example, if / if-else / while/ do-while/for/switch/for-in are statements.

2.3 Shorthand expression

Instead of *if / if-else / while/ do-while/for* switch statements, shorthand expression can be used. The ternary expression takes three operands: a condition followed by a (?), then an expression to execute if the condition is truth followed by a (:), and finally the expression to execute if the condition is false.

```
let msg1 = undefined;
msg1 = msg1 === undefined ? "hello world 1" : msg1;
console.log(msg1);       //hello world 1
```

Logical AND operator can replace *if-else* statement. When *x is false*, right expression is not executed. When *x* is *true*, right expression is executed.

```
let x = 1;
if (x === 1) {console.log("hello world 2");}  //hello world 2
   //using && expression – the result same as above
x === 1 && console.log("hello world 3");    //hello world 3
```

Logical OR operator can be used to assign the default value. When msg2 is empty, the right value is assigned.

```
let msg2 = "";
msg2 = msg2 || "hello world 4";
console.log(msg2);       //hello world 4
```

Code sample: https://codesandbox.io/s/js-shorthand-gl10b

2.4 Iterators

Iterator is a way to loop over any collection or lists in JavaScript. It is an object with a mechanism for enumerating the contents of an object. For example, built-in objects such as Array, String, Map, and Set have an iterator by default so that the elements can be enumerated by the ***for ... of iterator***.

for…of iterator (ES6)

The ***for...of statement*** loops/iterates through the collection, providing the ability to modify specific items. It replaces the conventional way of doing a *for-loop*. It leads to a clean and readable code.

```
const cities = ['London', 'Paris', 'Rome']
for (const name of cities) {
  console.log(name)}            //Paris London Rome
```

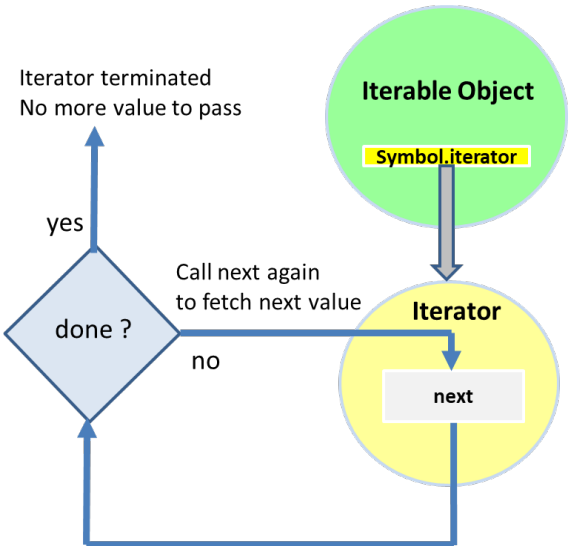
Code sample: https://codesandbox.io/s/js-iterator-zuj5n

Symbol.iterator (ES6)

Symbol.iterator will return an iterator object. The iterator has a method **next(),** which will return an object with keys value and done.

```
const iterable = ['Tokyo', ' Bangalore', 'New York'];

// generate iterator
const it = iterable[Symbol.iterator]()

while (true) {
  // get current status
  const result = it.next()

  // check condition
  if (result.done) {
    break
  }

  // aquire value
  const item = result.value
  console.log(item).       //Tokyo, Bangalore, New York
}
```

Code sample: https://codesandbox.io/s/js-iterator-zuj5n



The comparison between classic **for...in** and ES6 **for …of** is as follows:

	for…in	for…of (ES6)
Applicable to	Enumerable Properties of an Object	Objects that have a [Symbol.iterator] property
Objects	Yes	No
Arrays	Not recommended	Yes
Strings	Not recommended	Yes

2.5 Functions

If you only consider functions as a collection of reusable processes, JavaScript functions may be weird to you. A JavaScript function is a first-class object. It is not the only syntax but also values. The function itself can be set as an argument and return as a value (call back or higher-order function). Functions can be executed with the ( ) operator. There are two literal forms of functions: function declaration and function expression.

When a function is invoked, it begins execution with the first statement and ends when it hits the curly **bracket }** that closes the function body. This causes the function to return control to the part of the program that invoked the function. The return statement can be used to cause the function to return early. When the return is executed, the function returns immediately without executing the remaining statements.

When a function is invoked, the JavaScript interpreter creates an **execution context**. This record contains information about where the function was called from (the call-stack), how the function was invoked, what parameters were passed, etc. One of this record's properties is this reference, which will be used for the duration of that function’s execution.

Functions in JavaScript can be used like below:

- Store functions as variables
- Use functions in arrays
- Assign functions as object properties (methods)
- Pass functions as arguments
- Return functions from other functions

Function Declaration (Function Statement)

The function declaration starts with the function keyword and includes the name of the function following it. The contents of function are enclosed in **{ }**.

```
function foo() {//do something }
foo()              //calling above function
```

Function declaration

```
function func1() {
  console.log("dog1");
}
func1();                    // dog1

function func2(a) {
  console.log(a);
}
func2("dog2");              // dog2
```

Code sample: https://codesandbox.io/s/js-function-twvy4

Function Expression

The function expression does not require a name after the function. This is referred to as an **anonymous function**. Function expressions are referenced through variables or property. *let,* and *const* can be used as well as *var*.

```
var  foo = function () {//do something}
     foo()
```

functional expression – assignment to variable

```
//==== functional expression - assignment to variable======
let func3 = function() {
  console.log("dog3");
};
func3();                                  // dog3

let func4 = function(b) {
  console.log(b);
};
func4("dog4");                          // dog4

//====== functional expression return a value ==========
let add3 = function(x, y) {
  return x + y;
};
console.log(add3(3, 7));                 //10
```

Code sample: https://codesandbox.io/s/js-function-twvy4

The function expression can add a name after the function. This can be useful in debugging.

```
var  foo = function foo() {//do something}
foo()
```

IIFE (Immediately Invoked Function Expression)

The anonymous functions cannot invoke by themselves. The anonymous function can be an expression if we use it where JavaScript expecting a value. That means if we can tell JavaScript to expect a value with parentheses, we can pass an anonymous function as that value.

The main usage is to create a closed scope that protects internal identifiers from undesired external access. In other words, it's a method of encapsulation.

IIFE examples

```
//======= IIFE without arguments ==========

(function() {
  console.log("dog5");
})();                                    // dog5

//======= IIFE with arguments ============
(function(a, b) {
  console.log(a + b);
})("dog6", " dog7");                 // dog6 dog7
```

Code sample: https://codesandbox.io/s/js-function-twvy4

Arrow function (ES6)

Arrow functions, which are often referred to as Lambda functions, are a concise way of writing functions. Their short syntax is further enhanced by their ability to return values implicitly in one line, single return functions. Arrow functions are always considered anonymous functions.

```
const myFunction = function( ) {  //...}
```

is shortened to:

```
const myFunction = ( ) => { //...}
```

If the function body contains just a single statement, you can omit the brackets and write all on a single line:

```
const myFunction = ( ) => doSomething()
```

```
const myFunction = (param1, param2) => doProcess(param1, param2)
```

Parameters are passed in the parentheses. If you have only one parameter, you could omit the parentheses completely:

```
const myFunction = param => doSomething(param)
```

Arrow Function Return

Arrow functions have a built-in way to shorten a return syntax. **When you use curly brackets { }, you need to return the state explicitly. However, when you don't use curly brackets in the arrow function, the return is implied, and you can omit the return.**

const name = function(parameter){ return *something*}

is shortened to:

const name = (parameter) => { return *something*}

is shortened to:

const name = parameter => *something*

```
//conventional function expression
const hello1 = function() {
  return "Hello World 1!";
};
console.log(hello1());

//the syntax of an Arrow Function
const hello2 = () => {
  return "Hello World 2!";
};
console.log(hello2());

//Arrow Function without the brackets or the return keyword
//This works only if the function has only one statement.
const hello3 = () => "Hello World 3!";
console.log(hello3());

//Arrow Function with parameters
//if only one parameter, the parentheses can be omitted.
const hello4 = val => `Hello ${val}`;
console.log(hello4("World 4!"));
```

Code sample: https://codesandbox.io/s/js-arrow-function-b1sq3

Methods

JavaScript methods are actions that can be performed on Object. It is a property containing a function definition. In ES6, the colon and function keywords can be eliminated for simplicity. This coding pattern is used in the Hooks section in this book.

```
//method in ES5 and earlier
const person1 = {
  name: "Alonzo Church",
  sayName: function(name) {
    console.log(person1.name);
  }
};
person1.sayName();

//ES6 method - the colon and function can be eliminated
const person2 = {
  name: "Haskell Curry",
  sayName(name) {
    console.log(person2.name);
  }
};
person2.sayName();
```

Code sample: https://codesandbox.io/s/js-method-6tkd1

Arguments

Surprisingly there is no strict checking of arguments. If a function is called with missing arguments (less than the argument declared), the missing values are set as: undefined. If a function is called with extra arguments, it will be ignored.

```
function func(a, b, c) {
     console.log(a, b, c)
   }
   //less arguments passed
   func(' a')                          // output:  a undefined undefined

   //more arguments passed
   func(' a', 'b', 'c', 'd')          //output:  a b c

   // no arguments
   func()                          // output: undefined undefined undefined
```

Code sample: https://codesandbox.io/s/js-arguments-qh9f6

Arguments Object

The Arguments Object is an object that manages the arguments passed by the caller and is available within the scope of the function body. Whether or not arguments are defined in the function, the Arguments Object manages all the arguments passed to the functions.

```
function func1(a, b, c) {
    for (let p of arguments) {
      console.log(p)
    }
  }
  func1('a', 'b', 'c', 'd')                    // a b c d
  func1('a')                                   // a
  func1()                                      // nothing happens

  function func2() {
   // structure similar to array object
   console.log(arguments.length, arguments[0]);
  }
  func2(' x', 'y', 'z')                              // 3  x
```

Code sample: https://codesandbox.io/s/js-arguments-qh9f6

Function Rest parameter (ES6)

The function Rest parameters (**…**), indicated by three consecutive dot characters, allow your functions to have a variable number of arguments without using the **Arguments** Object. The **Rest** parameter is an instance of **Array,** so all array methods work.

Function Rest Parameter (ES6)

```
let nameCity = (arg1, arg2, ...moreArgs) => {
  console.log(arg1); // Logs arg1
  console.log(arg2); // Logs arg2
  console.log(moreArgs);
  // Logs an array of any other arguments you pass in after arg2
};
nameCity("Seattle", "Montreal", "Berlin", "Rome", " Bangalore", " Sydney", "Prague");
```

Code sample: https://codesandbox.io/s/js-func-rest-default-param-3686r

Default parameter values and optional parameters (ES6)

Default parameters allow your functions to have optional arguments without checking **arguments.length** or checking for **undefined**.

Default parameter values and optional parameters(ES6)

```
let sayHi = (msg = "hello", name = "world") => { console.log(msg, name);};
sayHi();                                    //hello world
sayHi("goodbye");                           //goodbye world
sayHi("Hi", "Mars");                        //Hi Mars

let sum = function(a, b = 10, c) { return a + b + c;};
console.log(sum(1, 2, 3));                   // 6
console.log(sum(1, undefined, 200));         // 211
```

Code sample: https://codesandbox.io/s/js-func-rest-default-param-3686r

Functions as Values

JavaScript functions are first-class objects. It is one of the most important concepts in JavaScript. You can use functions like other objects. You can assign functions to variables, pass them to other functions as arguments, return them from functions, add them to objects, or remove them from objects.

Functions as values

```
function sayHello() {
  console.log("Hello World");
}
sayHello();                          //output: Hello World

const sayHello2 = sayHello;
sayHello2();                         //output: Hello World
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

Note: In the CodeSandbox console, instead of "Hello World Hello World", " ② Hello World" will be displayed. It is a CodeSandBox log format.

Passing Anonymous functions as values (callback)

Anonymous functions are used for passing functions as a parameter to another function. Anonymous functions can be executed later. It is referred to as a callback, which is like leaving a message in the voice message to be called later. One example of a callback is the event listener. In the event listener, the callback function registered in advance is called when the expected event occurs. You can get used to the callback concept through several exercises below:

callback example 1

```
//Passing Anonymous functions as values
// function expression that can invoke an anonymous function
const sayHi = function(myFunc) {
  myFunc();
};

//pass an anonymous function as parameter
sayHi(function() {
  console.log("Hi anonymous");
});                                    // output: Hi anonymous
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

callback example 2

```
//====Passing Anonymous functions as values
const add1 = x => x + 1;

//this is not a callback.
console.log(add1(100));                          //101

//callback function example
const setupCallback = callback => arg => {
  return callback(arg);
};
const myCallback = setupCallback(add1);
const result = myCallback(200);
console.log(result);                             //201
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

callback example 3

```
// functions declaration that can invoke an anonymous function
function myFunc(callback) {
  console.log("hello");
  callback();                    //hello
}

// pass an anonymous function as parameter
myFunc(function() {
  console.log("USA");
});                              //USA
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

callback example 4

```
//**********callback example 4 **********
//display latitude and longitude by calling Geolocation API
//which is supported by browsers.

navigator.geolocation.getCurrentPosition(success);
function success(pos) {
  console.log(`my latitude: ${pos.coords.latitude}`);
  console.log(`my longitude: ${pos.coords.longitude}`);
}
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

Use cases of callbacks

Although JavaScript runs code sequentially in top-down order, there are some cases that code runs after something else happens and also not sequentially. This is called asynchronous programming. Callbacks ensure that a function will not run before a task is completed but will run right after the task has been completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.
For details, refer to 2.8 Asynchronous Systems.

Higher-Order Functions

Functions that take and/or return other functions are called *Higher-Order Functions*. Below, we pass an anonymous function to the *myFunc* function, which we then immediately return from the *myFunc* function.

Higher Order functions

```
// functions can be sent to another function and return from the function
const myHOF = function(myFunc) {
  return myFunc;
};
const sayHi2 = myHOF(function() {
  console.log("Hello world");
});
sayHi2();                                  //  'Hello world'
```

Code sample: https://codesandbox.io/s/js-function-as-value-rjyxy

Function declaration hoisting

Function declarations in JavaScript are hoisted to the top of the execution context - enclosing function or global scope. You can use the function before you declared it. However, Function expressions are not hoisting.

```
//functional declaration hoisted to top
myFunc() // logs "function hoisted"

function myFunc() {
  console.log('function hoisted')
}

//functional expression cannot be hoisted.
myFunc2() // this does not work

var myFunc2 = function() {
  console.log('function not hoisted')
}
```

Constructors

A constructor is simply a function that is used with a *new* keyword to create an object. The advantage of constructors is that constructors contain the same properties and methods. When you create many identical objects, you can create a constructor with reference types. After the constructor is defined, you can create instances by using a *new* keyword.

Instances are runtime instances of Component Class. These are represented as JavaScript objects in memory. If the functions are intended to be used as constructors, the first letter is capitalized by convention to avoid confusion. There is no syntactic difference between constructors and normal functions.

```
// define City constructor function in order to create custom instances later
let City = function(name) {
  this.name = name;
};
// instantiate a City object and store it in the city1,city2 variable
let city1 = new City("Paris");       //create instance
let city2 = new City("Tokyo");     //create instance

console.log(city1);          //City {name:"Paris", constructor: Object}
console.log(city2);         // City{name:"Tokyo", constructor: Object}
```

Code sample: https://codesandbox.io/s/js-constructor-d9tko

Many React projects have been using ES6 Class base components to handles states. However, starting with React version 16.8.0, new projects can use Hooks, which do not require Classes any longer. Although there are no plans to remove Classes from React, Class base components have been replaced by Hooks in many projects. It is far cleaner than using Classes in React. But you still need to understand the JavaScript class to maintain and support many existing projects.

‘this’ keyword in function

A function's *this* keyword behaves differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode. JavaScript function context is defined when running the function, not while defining it. This can surprise many programmers from different fields. Such late binding can be a powerful mechanism that allows us to re-use loosely coupled functions in various contexts. On the other hand, it could lead to sloppy code.

If a function has *this* reference inside, *this* reference usually points to an object. But which object it points to is depending on how the function was called.

‘this’ location	Location where ‘this’ refers to
Outside of functions	Global object
Inside of function	Global object
Inside of function (strict mode)	undefined
Inside of ES6 arrow function	The location where the allow function declared
Function being called using call /apply	Object specified by the argument
.bind(this)	Calling bind(this) on a function will return a new (bound) function that has the value of this already defined.
Inside of constructer	Instance generated by the constructer
Inside of method	Receiver object (object calling the method)
Inside of event listener	Event origin

this behavior is very confusing in JavaScript. Fortunately, in React, *this* keyword can be avoided by using functional components like React Hooks. However, as many existing React projects still use Class, understanding *this* may still be necessary.

If you would like to practice *this* behavior, we would suggest the following link.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

‘this’ behavior in ES6 Arrow function

An arrow function does not have its own *this*, but it has *this* value of the enclosing execution context. Arrow Functions lexically bind their context, so *this* actually refers to the originating context (defined when you write the code). The following example is ES5 function with bind(this) and Arrow function without bind(this).

```
//ES5   with bind(this), this points to obj1.  Without bind(this), it is undefined.
let obj1 = {
  msg: 'obj1 timeout',
  counter: function counter() {
    setTimeout(function() {
      console.log(this.msg)              //obj1 timeout
    }.bind(this), 1000);                 //if bind(this) is removed, undefined is displayed
  }
}
obj1.counter()

// ES6 Arrow function points to obj2 without bind(this)
let obj2 = {
  msg: 'obj2 timeout',
  counter: function counter() {
    setTimeout(() => {
      console.log(this.msg)              //ob2 timeout
    }, 2000)
  }
}
obj2.counter()
```

Code example: https://codesandbox.io/s/js-arrow-bind-u0nxl

call & apply

Both *call* and a*pply* are similar in usage. They execute a function in the context or scope of the first argument that you pass. They are functions that can only be called on other functions. With *call*, subsequent arguments are passed into the function as they are, while *apply* expects the second argument to be an array that it unpacks as arguments for the called function.

```
function sum(val1, val2) {
  console.log(this.val + val1 + val2)
}

let obj1 = {val: 1}
let obj2 = {val: 2}

sum.call(obj1, 1, 1)                    // 3 （ 1 + 1 + 1 ）
sum.call(obj2, 1, 1)                    // 4 （ 2 + 1 + 1)
sum.apply(obj1, [1, 1])                 // 3 （ 1 + 1 + 1）
sum.apply(obj2, [1, 1])                 // 4 （ 2 + 1 + 1)
```

Code sample: https://codesandbox.io/s/js-call-appply-bind-isqqn

Bind

bind( ) can be used by every single function. *bind( )* returns a bound function with the correct context *this* for calling the original function when executed later. *bind( )* can be used when the function needs to be called later in certain events.

```
function sum(val1, val2) {
  console.log(this.val + val1 + val2)
}

let obj1 = { val: 100}
let obj2 = { val: 200}

let obj1Sum = sum.bind(obj1)
let obj2Sum = sum.bind(obj2, 1, 2)

obj1Sum(1, 2)                    // 103 （ 10 + 1 + 2）
obj2Sum()                        // 203 （ 200 + 1 + 2）
```

```
function sum(val1, val2, val3) {
  console.log(val1 + val2 + val3)
}

let sumA = sum.bind(null, 1, 2, 3)          // bind all arguments
let sumB = sum.bind(null, 1)                // bind partial arguments

sumA()                                      // 6 （ 1 + 2 + 3）
sumB(2, 3)                                  // 6 （ 1 + 2 + 3 ）
sumB(4, 5)                                  // 10 （ 1 + 4 + 5 ）
```

Code sample: https://codesandbox.io/s/js-call-appply-bind-isqqn

calling from arrow function

```
const obj = {
  val: 'dog',
  show: function() {
    console.log(this)                    // Object {val: 'dog'}

    let fncA = function() {
      console.log(this)                  // undefined   (no bind)
    }
    fncA()

    let fncB = () => {
      console.log(this)                  // Object {val: 'dog'}
    }
    fncB()
  }
}

obj.show()
```

Code sample: https://codesandbox.io/s/js-call-appply-bind-isqqn

Recursive function

A recursive function is a coding pattern that invokes itself until a condition is met.
In functional programming, recursion is often used instead of loop/while. The downside of the recursion is that it consumes the stack area. It is a trade-off between simplicity and higher memory usage.

In the code below, we start the *countDown* function, which then calls itself via the function name *countDown***.** This counts down from 5 to 0.

```
const countDown = function countDown(value) {
   console.log(value)
   value--;      // decrement
   return (value >= 0) ? countDown(value) : false
}

countDown(5)                              // output: 5,4,3,2,1,0
```

Code sample: https://codesandbox.io/s/js-memoize-ktwo5

Memoization

Memoization is an optimization technique that increases a function's performance by remembering its previously computed results. Because JavaScript objects behave like associative arrays, they can act as caches. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned without executing the entire function. However, if the data is not cached, then the function is executed, and the result is added to the cache. This method of optimization is not unique to JavaScript. It is useful in recursive functions as calls are more likely to call with the same arguments.

In the below example, *memoizedAdd* returns a function that is invoked later. This is possible because, in JavaScript, functions are first-class objects which let us use them as higher-order functions and return another function. A cache can remember its values since the returned function has a closure over it.

The memoized function must be pure. A pure function will return the same output for a particular input, no matter how many times being called, which makes the cache work as expected.

```
// a simple memoized function to add something
const memoizedAdd = () => {
  let cache = {};
  return n => {
    if (n in cache) {
      console.log("Fetching from cache");
      return cache[n];
    } else {
      console.log("Calculating result");
      let result = n * (n - 1);
      cache[n] = result;
      return result;
    }
  };
};

// returned function from memoizedAdd
const newAdd = memoizedAdd();
console.log(newAdd(6)); // calculated result 30
console.log(newAdd(6)); // cached  result 30
```

Code sample: https://codesandbox.io/s/js-memoize-ktwo5

Enhanced Object Properties

Property shorthand

ES6 syntax for creating an object literal is simple and clean.

Enhanced Object Properties

```
let first = "Ada";
let last = "Lovelace";

//ES5 and earlier
let obj1 = {
    first: first,
    last: last
};
console.log(obj1);      // Object {first: "Ada", last: "Lovelace"}

//ES6 Shorthand
let obj2 = {
   first,
   last
};
console.log(obj2);     // Object {first: "Ada", last: "Lovelace"}
```

Code sample: https://codesandbox.io/s/js-enhanced-obj-props-vfq24

ES6 Shorthand for Function Declarations

In ES5, we must state the property name, then define it as a function. ES6 added a shorthand for declaring function methods inside objects. We can omit the property name and the function keyword. ES6 syntax for methods is used throughout this book.

ES6 method syntax example

```
//===============ES5 and earlier method============================
let mathES5 = {
  add: function(a, b) {
    return a + b;
  },
  subtract: function(a, b) {
    return a - b;
  }
};
console.log(mathES5.add(1, 2));               //3
console.log(mathES5.subtract(9, 7));        //2

//===============ES6 method=============================
let mathES6 = {
  add(a, b) {
    return a + b;
  },
  subtract(a, b) {
    return a - b;
  }
};
console.log(mathES6.subtract(10, 2));         //8
console.log(mathES6.add(4, 5));               //9
```

Code sample: https://codesandbox.io/s/js-enhanced-obj-props-vfq24

Object.assign (ES6)

The **Object.assign( )** method is used to copy property values from one or more source objects to a given target object. It will return the target object.

In the following example, Object.assign( ) method is used to create the new object by specifying { } as the first parameter:

ES6 Object.assign(example 1)

```
let obj = {
  firstname: "Ada",
  lastname: "Lovelace"
};
let copy = Object.assign({}, obj);
console.log(copy);

// output:    Object {firstname="Ada",   lastname="Lovelace"}
```

Code sample: https://codesandbox.io/s/js-obj-assign-r9evy

The next example is to provide multiple sources as well as a target with values. The target will receive the values from the sources and return with the values merged. The original target variable is also changed as well, *o1* being the target with *o2* and *o3* the two sources:

ES6 Object.assign (example 2)

```
let o1 = { a: 1 };
let o2 = { b: 2 };
let o3 = { c: 3 };

let obj1 = Object.assign(o1, o2, o3);

console.log(obj1);                  //output:  Object {a=1, b=2, c=3}
console.log(o1);                    //output:  Object {a=1, b=2, c=3}
```

Code sample: https://codesandbox.io/s/js-obj-assign-r9evy

Destructuring Assignment (ES6)

Destructuring assignment syntax is an ES6 expression that unpacks values from arrays, or properties from objects, into named variables. It is a widely used syntax in React/Redux.

Array destructuring examples

```
let numbers = ['one', 'two'] //Basic array destructuring
let [one, two] = numbers
console.log(one)                  // "one"
console.log(two)                  // "two"

let a, b                          //Assignment separate from declaration
[a, b] = [10, 20]
console.log(a)                    // expected output: 10
console.log(b)                    // expected output: 20

let [c, d, ...rest] = [1, 2, 3, 4, 5]                  // Assigning the rest of an array to a variable
console.log(c)                                         //expected output: 1
console.log(d)                                         //expected output: 2
console.log(rest)                                      // expected output: [3,4,5]

let [x, , y] = [10, 20, 30]                            //Ignoring some returned values
console.log(x)                                         //expected output: 10
console.log(y)                                         //expected output: 30

let [e = 100, f = 200] = [10]                          //default values
console.log(e)                                         // 10
console.log(f)                                         // 200

let g = 1;
let h = 2;                                             //swapping variables
[g, h] = [h, g]
console.log(g)                                         // 2
console.log(h)                                         // 1
```

Code sample: https://codesandbox.io/s/js-destructure-assign-ychh4

Object destructuring examples

```
let o = { p: 10, q: 20} //Basic Object destructuring
let {p, q} = o
console.log(p)                          // 10
console.log(q)                          // 20

let a, b   // Assignment without declaration
({a, b} = { a: 30, b: 40})
console.log(a)                          // 30
console.log(b)                          // 40

let x = {c: 50,  d: 60}    //Assigning to new variable names
let {c: foo, d: bar} = x
console.log(foo)                        // 50
console.log(bar)                        // 60

let {e = 70, f = 80} = {e: 1}           //default values
console.log(e)                          //1
console.log(f)                          // 80
```

Code sample: https://codesandbox.io/s/js-destructure-assign-ychh4

Functions can return multiple values in array.

```
let [i,j] =(function f(){return [50,60]})()
console.log(i)                          // 50
console.log(j)                          // 60
```

Code sample: https://codesandbox.io/s/js-destructure-assign-ychh4

Spread Operator (ES6)

The Spread Operator expands an array into multiple parameters. It is like a reverse operation of function rest parameters. It can also be used for object literals. It is a simple and intuitive way to pass arguments. The Spread Operator operator makes copying and merging arrays a lot simpler. Rather than calling the *concat( )* or *slice( )* method, you could use the Spread Operator. Spread Operator is a syntax commonly used in React/ Redux.

```
// showing elements of array without creating a loop function.
const city = ["Paris", "London", "Rome"]
console.log(...city)                                    //Paris London Rome

//  if you do not use the Spread Operator,  it is like this
// or create loop if there are many element in the array.
console.log(city[0], city[1], city[2])                  //Paris London Rome

//combing two arrays.
let myArray1 = ['two', 'three', 'four']
let myArray2 = ['zero', 'one', ...myArray1, 'five']

console.log(myArray2)          // output: ['zero', "one", "two", "three", "four", "five"]
```

Code sample: https://codesandbox.io/s/js-spread-operator-k12ov

Using Spread Operator, functions can be called without using apply

```
function sum(x, y, z) {
  return x + y + z
}
var myArray = [1, 2, 3]

// Call the function with apply
console.log(sum.apply(null, myArray))              //output:  6

// Call the function using Spread operator
console.log(sum(...myArray))                       //output:  6
```

Code sample: https://codesandbox.io/s/js-spread-operator-k12ov

2.6 Arrays Methods

In JavaScript, arrays are basically a data structure that holds a list of values. These values can be strings, integers, objects, or even functions. Arrays don't have fixed types or restricted lengths.

Includes( ) method

The **includes( )** method is used to check if a specific string exists in an array, returning true or false. It is case-sensitive.

array includes examples

```
const cities1 = ["London", "Paris", "Rome"];
const findcity1 = cities1.includes("Paris");
console.log(findcity1);                          //true
const findcity2 = cities1.includes("NYC");
console.log(findcity2);                          //false
```

Code sample: https://codesandbox.io/s/js-array-method1-50lhi

Some( ) method

The *some( )* method checks if some elements exist in an array, returning true or false. It is similar to the concept of the *includes( )* method, except the argument is a function and not a string. (logical OR)

array some( ) examples

```
const cities2 = ["London", "Paris", "Rome"];
let result1 = cities2.some(name => name === "Paris");
console.log(result1);                                  //true
let result2 = cities2.some(name => name === "Berlin");
console.log(result2);                                  //false
```

Code sample: https://codesandbox.io/s/js-array-method1-50lhi

every( ) method

The *every( )* method loops through the array, checks every item, returning true or false. Same concept as *some( ).* Except every item must satisfy the conditional statement. Otherwise, it will return false. (logical AND)

array every( ) examples

```
const age = [85, 25, 36];
result1 = age.every(person => person >= 20);
console.log(result1);                          // true
result2 = age.every(person => person >= 30);
console.log(result2);                         // false
```

Code sample: https://codesandbox.io/s/js-array-method1-50lhi

forEach( ) method

The *forEach( )* method executes a provided function once for each array element. The *forEach( )* method doesn't actually return anything (undefined). It simply calls a provided function on each element in your array. The *forEach()* method allows a callback function to mutate the current array.

array forEach( ) examples

```
const array1 = ["1", "2", "3"];
array1.forEach(function(element) {
  console.log(element);                    //  1    2    3
});
console.log(array1);                      //["1", "2", "3"];
```

Code sample: https://codesandbox.io/s/js-array-method1-50lhi

map( ) method

The *map( )* method creates a new array by calling a provided function on every element in the calling array. The *map( )* method is used to modify each element. The original array is unchanged.

Array Map( )

```
const myArray = [50, 10, 3, 1, 2];

//using Anonymous function
const myArray1 = myArray.map(function(x) {
  return x * 2;
});
console.log(myArray1);                          //[100,20,6,2,4];

//using ES6 arrow function
const myArray2 = myArray.map(x => x * 2);
console.log(myArray2);                          //[100,20,6,2,4];


console.log(myArray);                          //[50,10,3,1,2]
```

Code sample: https://codesandbox.io/s/js-array-method2-wzp3b

Filter( ) method

The *filter( )* method creates a new array with all elements that pass the test.
It lets you filter out items you do not need. The original array remains the same.

Array Filter( )

```
const age = [85, 25, 6, 100, 17];

//using Anonymous function
const result1 = age.filter(function(legalAge) {
  return legalAge >= 18;
});
console.log(result1);                                    //[85, 25, 100]

//using ES6 arrow function
const result2 = age.filter(legalAge => legalAge >= 18);
console.log(result2);                                     //[85, 25, 100]
```

Code sample: https://codesandbox.io/s/js-array-method2-wzp3b

reduce( ) method

The *reduce( )* method is used to apply a function to each element in the array to reduce the array to a single value. The *reduce( )* transforms an array into something else, which could be an integer, and another array, an object, a chain of promises, etc. In short, it reduces the whole array into one value. It is used for calculating value, grouping objects by the property, executing promises sequentially. The original array remains the same.

Array Reduce( ) example 1

```
const array1 = [5, 4, 3, 2, 1];
const myReducer = (accumulator, currentValue) => accumulator + currentValue;
    // 5+4+3+2+1
const array2 = array1.reduce(myReducer);
console.log(array2);                                          // output: 15     //accumulated value

  // 100+5+4+3+2+1
const initialValue = 100;         //set initial value
const array3 = array1.reduce(myReducer, initialValue);
console.log(array3);                                           // output: 115
```

Code sample: https://codesandbox.io/s/js-array-method2-wzp3b

Array Reduce( ) example 2

```
const data = [
  { area: "Paris", population: 11 },
  { area: "London", population: 13 },
  { area: "Rome", population: 4 },
  { area: "Berlin", population: 6 }
];

let sum = data.reduce((acc, val) => { return val.area === "London" ? acc : acc + val.population;}, 0);
console.log(sum);                                  // 21  (Million) metropolitan population
```

Code sample: https://codesandbox.io/s/js-array-method2-wzp3b

Array Reduce( ) example 3

```
const array4 = [["A", "B", "C"], ["x", "y"], ["1", "2"]];

const array4x = array4.reduce((acc, val) => {return acc.concat(val); }, []);
console.log(array4);
console.log(array4x);
```

Code sample: https://codesandbox.io/s/js-array-method2-wzp3b

map.filter.reduce chaining example

map method, *filter* method, and *reduce* method can be combined together. This pattern is often used in functional programming.

array map filter

```
let a = [1, 2, 3, 4, 5];
const result = a
  .map(function(i) {
    return i + 1;
  })
  .filter(function(i) {
    return i % 2 === 0;
  })
  .reduce(function(total, current) {
    return (total += current);
  });
console.log(result);                    //12
console.log(a);                         //1, 2, 3, 4, 5];
```

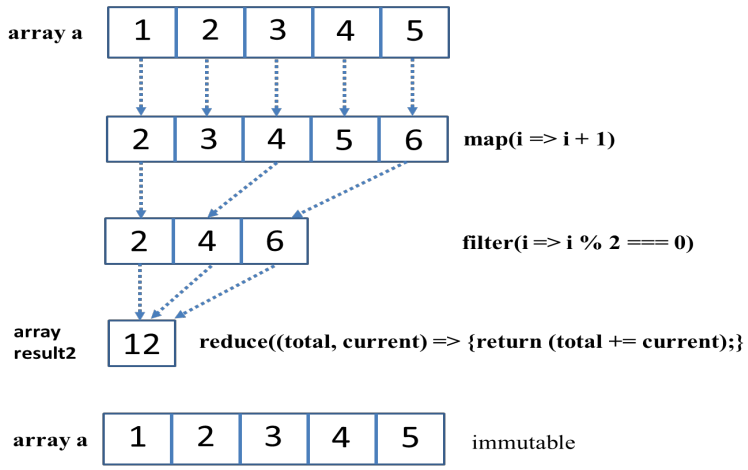
Code sample: https://codesandbox.io/s/js-array-map-filter-reduce-fxyey

array map filter using Arrow function

```
let b = [1, 2, 3, 4, 5];
const result2 = b
  .map(i => i + 1)
  .filter(i => i % 2 === 0)
  .reduce((total, current) => {
    return (total += current);
  });

console.log(result2);                   //12
console.log(b);                         //1, 2, 3, 4, 5];
```

Code sample: https://codesandbox.io/s/js-array-map-filter-reduce-fxyey

If we visualize this code, it is like this.



In React, it is recommended to use immutable array methods. Mutable array methods should be avoided.

Immutable array methods (recommended to use with React/Redux)

Length	isArray(obj)	toString( )
toLocaleString( )	indexOf(elm [,index])	lastIndexOf(elm [,index])
entries( )	keys( )	values( )
concat(ary)	join(del)	slice(start [,end])
from(alike [,map [,othis]]	of(el ,…)	
map(fnc [,that])	every(fnc [,that])	some(fnc [,that])
filter(fnc [,that])	find(fnc [,that])	findIndex(fnc [,that])
reduce(fnc [,init])	reduceRight(fnc [,init])	

Destructive array methods (not recommended to use with React/Redux)

splice(,start, cnt [,rep [,…]])	copyWithin(target ,start [,end])	fill(val [,start [,end]])
pop( )	push(data)	shift( )
Unshift(data1 [,data2 ,…])	reverse( )	sort([fnc])
forEach(fnc [,that])		

If you really prefer to use methods mutating original arrays, you can use Immer library. We will learn it later.

2.7 Function Scope & Closure

A closure is an important concept for React/Redux. To comprehend closure, understanding the concept of lexical scope and functional scope is mandatory.

A lexical scope or scope chain in JavaScript refers to the accessibility of the variables, functions, and objects based on their physical location in the source code. **The lexical scope is defined when you write the code.** No matter where the function is invoked from or how it is invoked, its lexical scope is defined by where the function was declared.

The function scope is created for a function call execution, not for the function itself. Every function call creates a new function scope dynamically at run time. When the function has completed the execution, the scope is usually destroyed. The function scope is garbage collected. (JavaScript has memory management)

In the following example, there are two variables: outside is accessible throughout the program and inside is only accessible inside the function:

```
const outside = 'I live in the global scope'

function myFunc() {
  const inside = 'I live in the function scope'
}
console.log(outside)                    //I live in the global scope
console.log(inside)                     //(cannot access)
```

Execution Context

An execution context is an abstract environment where the JavaScript code is evaluated and executed. When the global code is executed, it is executed inside the global execution context, and the function code is executed inside the function execution context.

JavaScript is a single-threaded language. There is only one currently running execution context managed by a stack data structure known as Execution Stack or Call Stack.

An execution stack uses LIFO (Last-In, First-Out) buffer structure in which items can only be added or removed from the top of the stack only. The current running execution context will always be on the top of the stack. When the currently running function completes the execution, its execution context is popped off from the stack, and the control reaches the execution context below it in the stack.

Lexical Environment

In JavaScript, scopes are implemented via lexical environments. Every time the JavaScript engine creates an execution context, it also creates a new lexical environment to store the variable defined in that function during the execution of that function.

A lexical environment is a data structure that holds an identifier-variable mapping. Identifier refers to the name of variables/functions, and the variable is the reference to the actual complex type object or primitive value.

A Lexical Environment has two components:

1. An **environment record**: the actual place where the variable and function declarations are stored.
2. A **reference to the outer environment**: access to its outer (parent) lexical environment. This concept is the most important to understand how closures work.

Closure

Most JavaScript Developers use closure consciously or unconsciously. Even if you use it unconsciously, it may work fine in many cases. But understanding closure will provide you better control over the code. After React Hooks had been released in 2019, React gradually moved towards functional programming. That means understanding the closure is one of the most important knowledge for studying React.

In JavaScript, every running function, code block, and script has an associated object known as the Lexical Environment. A new Lexical Environment is created each time a function is executed.

If the function has more than two-level nested inner functions, the innermost Lexical Environment is searched first, then the more outer one, and so on until the end of the chain.

For example, we write a two-level nested function like a Closure example 2b. When this function is called multiple times, each invocation will have its own Lexical Environment, with local variables and parameters specific for that very run. When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one.

In plain English, **a closure is a function with access to its outer function scope even after the outer function has returned. This means a closure can remember and access variables and arguments of its outer function even after the function has been completed.**

Typically, a function has local variables in its code body, including any parameters, and it also might have valuables that are not defined locally nor global valuables. These valuables are called free valuables. Closures are functions that refer to free variables. In other words**, the function defined in the closure remembers the environment in which it was created.**

Learning Closure through code examples

One of the key principles in creating closures is that an “inner” function, which is declared inside another function, has full access to all of the variables declared inside the scope of the function in which it’s declared (the “outer” function). In other words, the inner function preserves the scope chain of the enclosing function at the time the enclosing function was executed. Therefore, it can access the enclosing function’s variables.

Closure examples 1 – Function scope

The following example shows the scope of the inner function and the outer function. The *outer( )* function has access to the variable outside only, which is declared in its scope. On the other hand, the *inner( )* function has access to the variable inside and the variable outside.

```
function outer() {
  const outside = 'I live outside!';

  function inner() {
    const inside = 'I live inside!';
    console.log(outside);        //I live outside!
    console.log(inside);         //I live inside!
  }
  return inner;
}
const myclosure = outer();
myclosure();
```

Code sample: https://codesandbox.io/s/js-closure1-oo30c

Closure examples 2a – counter without closure

The following example is a counter without closure. The problem is that each time *xcounter* is called, the global variable *xcount* is incremented. Global variables could clash with the names of other variables.

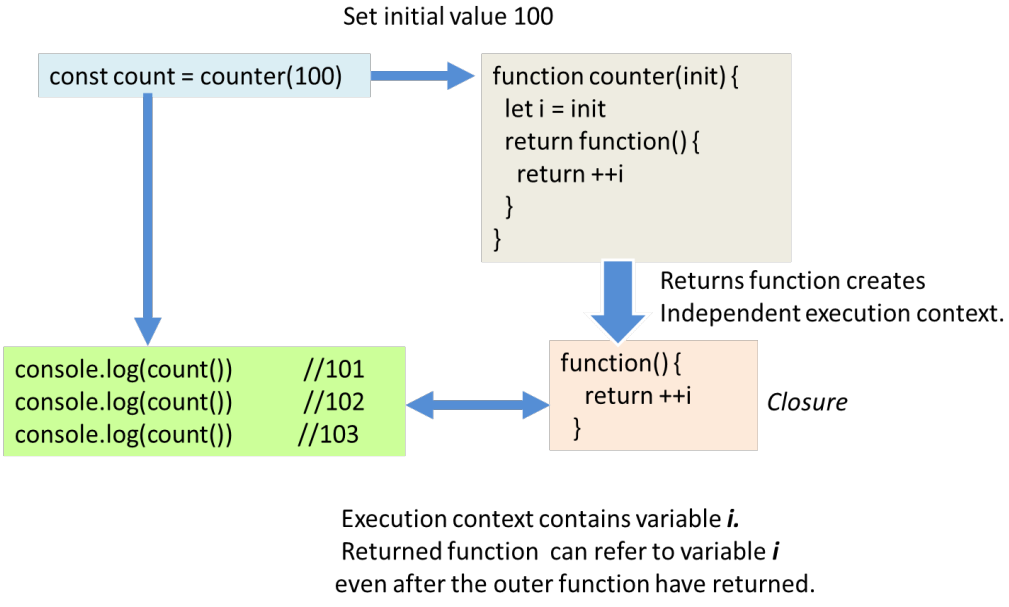
```
let xcount = 0;       //global variable !

function xcounter() {
  return ++xcount;
}

console.log(xcounter()); //1
console.log(xcounter()); //2
console.log(xcounter()); //3
```

Closure examples 2b –closure remains even after the function returned

The following example is a counter with local and protected variable *i*. The inner function creates an independent execution context containing variable *i*. The returned function can refer to the variable *i*, even after the outer function is terminated. Therefore each time *count* is invoked, the variable *i* is incremented from the previous value. The closure keeps the execution context.

```
function counter(init) {
  let i = init
  return function() {
    return ++i
  }
}
const count = counter(100)
console.log(count())          //101
console.log(count())          //102
console.log(count())          //103
```

Code sample: https://codesandbox.io/s/js-closure1-oo30c

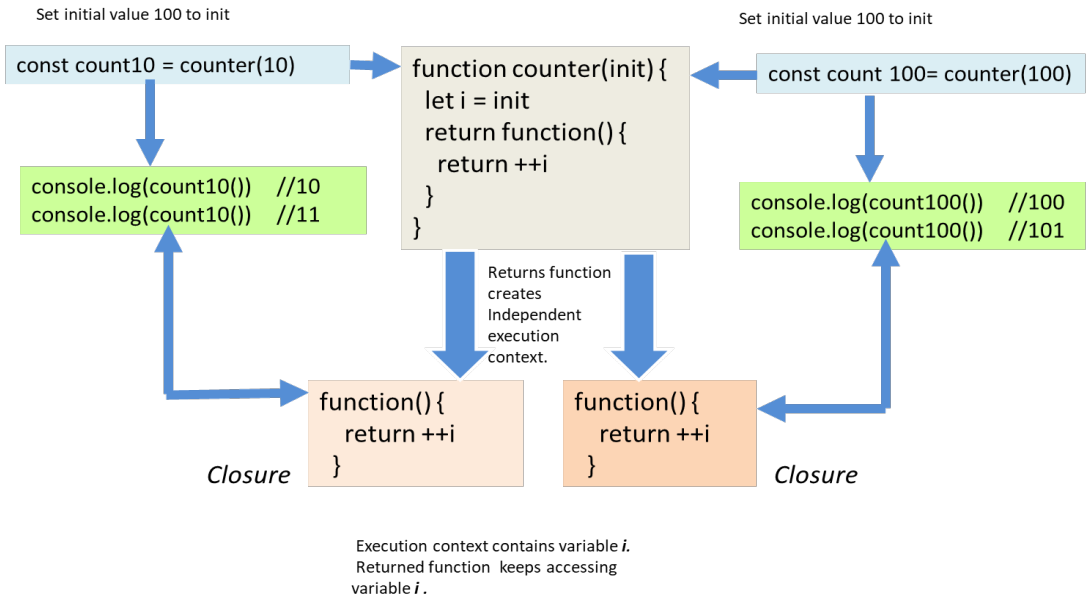


Closure examples 3 – two independent closures created

In the following example, *count10* and *count100* maintain their independence from the other. Each closure references a different version of the private counter variable through its own closure. Each time one of the counters is called, its lexical environment is updated by changing the value of this variable. However, changes to the variable value in one closure do not affect the value in the other closure. We have access to variables defined in the enclosing function(s) even after the enclosing function, which defines these variables, has returned.

```
function counter(init) {
  let i = init
  return function() {
    return i++
  }
}
const count10 = counter(10)
const count100 = counter(100)
console.log(count10())                //10
console.log(count100())               //100
console.log(count10())                //11
console.log(count100())               //101
```

Code sample: https://codesandbox.io/s/js-closure2-r564h



Closure examples 4 – Adder using closures

In the following example, we define a function *adder(x)*, which takes a single argument *x,* and returns another function. The returning function **takes a single argument *y* and returns the sum of *x* and *y***. *adder(x)* creates functions that can add a specific value to their argument.

We use *adder(x)* to create two new functions — *add10 for* adding 10 and *add100* for adding 100. *add10* and *add100* share the same function body definition but store different lexical scope environments. In the lexical scope environment for *add10*, *x* is 10, while in the lexical scope environment for *add100*, *x* is 100.

Using a closure, a function can remember and access its lexical scope even when that function is executing outside its lexical scope.

```
function adder(x) {
  return function(y) {
    return x + y
  }
}

const add10 = adder(10)
const add100 = adder(100)

console.log(add10(5))              // 15
console.log(add100(5))             // 105
```

Code sample: https://codesandbox.io/s/js-closure2-r564h

Closure examples 5 – Methods create closures

JavaScript Methods can be used as well to create closures. This example produces an object containing two closures: inc and reset properties. Each closure shares access to the *n* variable. The *inc* closure updates the value of *n*, and *reset* closure clears the values of *n*.

```
function xcounter() {
  var n = 0;

  return {
    inc: function() {
      return ++n;
    },
    reset: function() {
      n = 0;
    }
  };
}

let xcounter1 = xcounter(); //create xcounter1
let xcounter2 = xcounter(); //create xcounter2

//two counters count independently
console.log(xcounter1.inc());                    //1
console.log(xcounter2.inc());                    //1
xcounter2.reset();                               //reset xcounter2
console.log(xcounter1.inc());                    //2
console.log(xcounter2.inc());                   //1
```

Code sample: https://codesandbox.io/s/js-closure2-r564h

Curry

Named after Haskell Curry, Curry is the process of breaking down **a function into a series of functions that take a single argument**. A curried function is a function that takes multiple arguments one at a time. A curried function will be split into many chained functions, all taking exactly one argument. A curried function works by creating a closure that holds the original function and the arguments. With Curry, functions become more predictable and reusable. For example, Redux middleware uses Curry. The below examples un-curried and curried functions.

```
//==============un-curried divider ====================
//calling function with two arguments x,y.

const div = (x, y) => x / y;
console.log(div(10, 2));                    //5

//============curried divider ======================
//invoking function with the first argument x.
//The result is a function which will be called
//with the second argument y to produce the result.

const divCurry1 = function(x) {
  return function(y) {
    return x / y; // first x=20,   second y=2
  };
};
console.log(divCurry1(20)(2));              //10

//============curried divider using arrow function======
//syntax is simpler
const divCurry2 = x => y => x / y;   // first x=30,   second y=2
console.log(divCurry2(30)(2));        // 15
```

Code sample: https://codesandbox.io/s/js-curry-partialapplication-08zi5

Partial Application

Partial application is a technique used to transform functions with multiple arguments into multiple functions that take fewer arguments. A function can be called with fewer arguments than it expects. And the function returns a function that takes the remaining arguments. And this is called the Partial Application of functions. A curried function will be split into many chained functions, all taking exactly one argument, but a partially applied function does not have that limitation.

Partial Application

```
//call a function with fewer arguments than it expects.
//And the function returns a function that takes the remaining arguments.

const add = (a, b) => a + b;
let add1 = add.bind(null, 1);
let add2 = add.bind(null, 2);

console.log(add1(10));          //11
console.log(add2(10));          //12
```

Code sample: https://codesandbox.io/s/js-curry-partialapplication-08zi5

Prototypes and ES6 Classes

A prototype is a mechanism provided by JavaScript for inheritance implementation. It is the basis for code reuse. While most Object-Oriented languages use class-based inheritance, JavaScript utilizes prototypal inheritance. Every function has a prototype property that defines any properties shared by objects created with a particular constructor.

Prototypes with Constructors

```
// define City constructor function in order to create custom instances later
let City = function(name) {
  this.name = name;
};
// sayName() is a prototype property instead of an own property.
City.prototype.sayName = function() {
  console.log(this.name);
};

// instantiate a City object and store it in the city1,city2 variable
let city1 = new City("Kyoto");
let city2 = new City("Tokyo");

city1.sayName();
city2.sayName();
```

Code sample: https://codesandbox.io/s/js-prototype-95x48

ES6 Classes

JavaScript does not have Classes. JavaScript uses Prototypes, singleton objects from which other objects inherit. In fact, all objects in JavaScript have a prototype from which they inherit. This means that JavaScript Classes do not behave exactly like conventional Classes. You can actually add a new method to an array, and suddenly all arrays can use it. This can be done in runtime, affecting an already instanced object.

ES6 Class is merely syntax sugar of the prototypes with constructers. JavaScript developers who want to implement classical inheritance can use ES6 Classes and avoid explicit use of prototypes:

ES6 class

```
class Point1 {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return this.x + ", " + this.y;
  }
}

let p = new Point1(25, 8);
console.log(p.toString());              // 25, 8
```

Code sample: https://codesandbox.io/s/js-prototype-95x48

Subclassing

The *extends* clause lets you create a subclass of an existing constructor (which may or may not have been defined via a class):

ES6 subclass

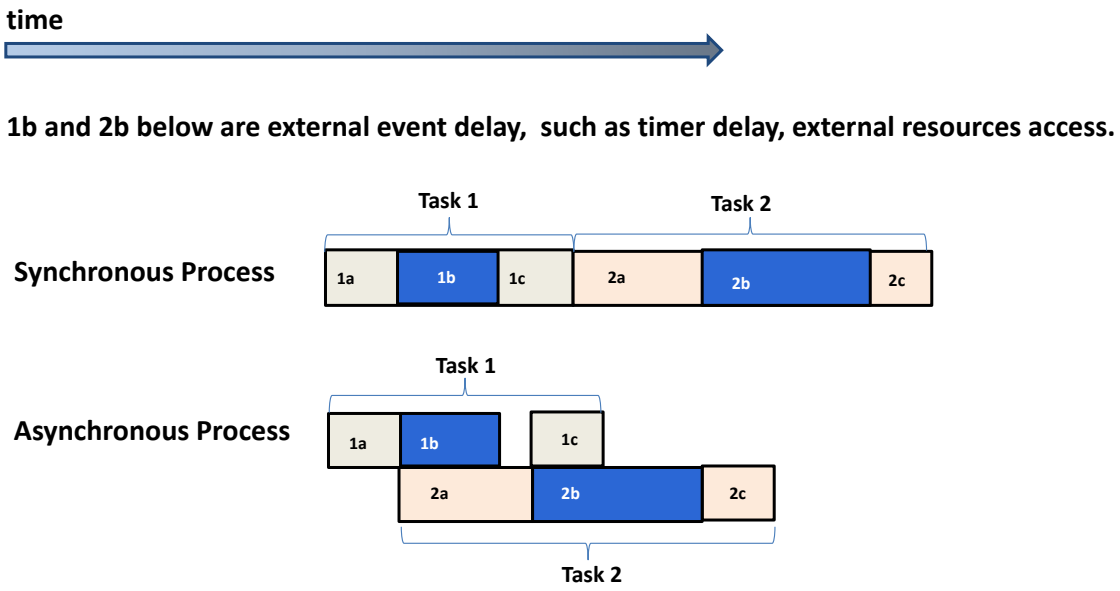
```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return this.x + ", " + this.y;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);                                    // (A)
    this.color = color;
  }
  toString() {
    return super.toString() + " in " + this.color;          // (B)
  }
}
let cp = new ColorPoint(25, 8, "green");
console.log(cp.toString());                         //'(25, 8) in green'
```

Code sample: https://codesandbox.io/s/js-prototype-95x48

2.8 Asynchronous systems

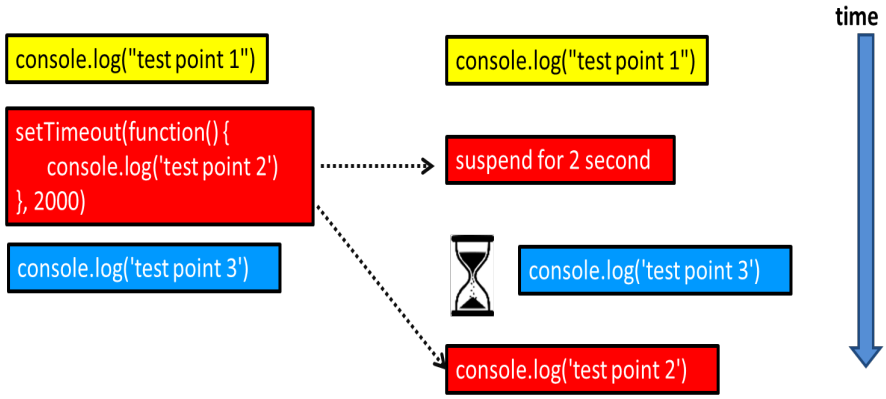
Computers are synchronous systems from digital circuits point of view. Generally, all internal logic gates are synchronized with the system clock. However, computers are asynchronous from a high-level systems point of view. Normally, programming languages are synchronous, and some provide a way to manage the asynchronous operation, using the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, Python are all synchronous by default. Some of them handle async by using threads, spawning a new process. Unlike most languages, JavaScript is asynchronous by default and is single-threaded. This means that code cannot create new threads and run in parallel. However, commands which can take any amount of time do not halt the execution. That includes operations such as requesting a URL, reading a file, or updating a database. The asynchronous event handling of JavaScript is similar to the event driven RTOS (Real Time Operation System) in low-level software.

The below picture is a comparison between the synchronous process and the asynchronous process. The synchronous process executes task 1a →1b → 1c → 2a →2b →2c sequentially. If one of the tasks blocks the execution, it would add the total system delay. On the other hand, the asynchronous process does not block the process. Even if one of the tasks blocks the execution, it would not add to the total system delay.



The following example and diagram could help you understand the asynchronous process with a timer event. The setTimeout() method calls a function that evaluates an expression after a specified number of milliseconds. As the setTimeout() method does not block the next processing, "test point 3" is displayed before displaying "test point 2".

Asynchronous (non-blocking) example




```
console.log("test point 1");

setTimeout(function() {
  //setTimeout triggers an asynchronous operation.
  console.log("test point 2");
}, 2000);

console.log("test point 3");      //this runs while waiting for the 2 sec. timeout.

//expect result
// test point 1
// test point 3
// test point 2
```

Code sample: https://codesandbox.io/s/js-async-n19gm

Nested callbacks - Callback hell

A callback can lead to callback hell when a series of nested asynchronous functions are executed in order. While the concept of callbacks is great in theory, it can lead to some really confusing and difficult-to-read code. For example:

```
function doSomething() {
  doSomething1((response1) => {
    doSomething2(response1, (response2) => {
      doSomething3(response2, (response3) => {
        // etc...
      };
    });
  });
}
```

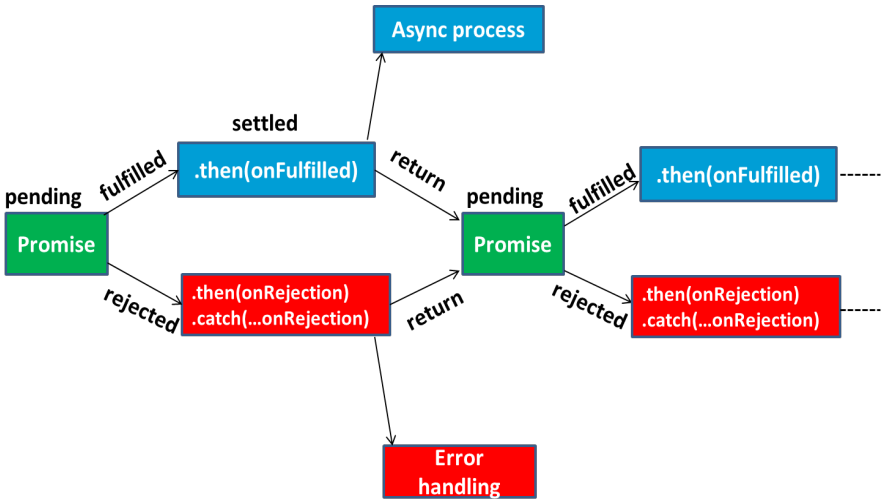
Promise

ES6 introduced Promises, which provided a cleaner way to express the same functionality. A promise represents a value that we can handle at some point in the future. A promise is an object which can be returned synchronously from an asynchronous function. Promises give us advantages over callbacks:

- No other registered handlers of that value can change it. A promise contract is immutable.
- We are guaranteed to receive the value, regardless of when we register a handler for it, even if it's already resolved. This contrasts with conventional events - once an event is fired, you can't access its value at a later time.
- The promise can handle multiple asynchronous operations easily and provide better error handling than callbacks and events.

A Promise has four states:

1. **pending:** Promise is pending. i.e., not fulfilled or rejected yet
2. **fulfilled:** Action related to the promise succeeded
3. **rejected:** Action related to the promise failed
4. **settled:** Promise has fulfilled or rejected



Creating Promise

new promise(function (resolve, reject) {…..}) return promise

Promise constructor takes only one argument, which is a callback function.
The callback function has two arguments, resolve and reject. It performs operations inside the callback function and if everything went well, then call a resolve. If the expected operations fail, then call a reject.

Promise Consumer - Promise.then

Promise.then ([onFulfilled], [OnRejected]) return promise

```
//======immediately resolved promise================
let myPromise1 = Promise.resolve("my promise 1");
myPromise1.then(res => console.log(res));                    //my promise 1

//======Promise constructor and then resolve============
/*After 1 sec, a Promise will be resolved.*/

let myPromise2 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("my promise 2"), 1000);
});

myPromise2.then(res => {
  console.log(res);                              //my promise 2
});
```

Code sample: https://codesandbox.io/s/js-promise-xekte

In the below example we are calling *reject* after 2 sec delay. *then( )* can also take in a second handler for errors.

Promise constructor and then reject

```
/*reject with time out error */
let myPromise3 = new Promise(function(resolve, reject) {
  setTimeout(() => reject("timedout 1"), 2000);
});
myPromise3.then(res => console.log(res), err => console.log(err));
                                                                                    //time out 1
```

Code sample: https://codesandbox.io/s/js-promise-xekte

For example, we can integrate resolve and reject like this.

A function returns a Promise object and then reject

```
function asyncProcess(value) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (value) {
        resolve(`input value: ${value}`);
      } else {
        reject("no input");
      }
    }, 4000);
  });
}
asyncProcess().then(
  response => {
    console.log(response);
  },
  error => {
    console.log(`error: ${error}`);                    //error: no input
  }
);
```

Code sample: https://codesandbox.io/s/js-promise-xekte

Promise consumer - Promise.catch

Promise.catch (OnRejected) return promise.

The Promise.catch method returns a Promise and deals with rejected cases only. So you have to provide an onRejected function even if you want to fall back to an undefined result value.

Promise constructor and then Promise.catch

```
let myPromise4 = new Promise(function(resolve, reject) {
  setTimeout(() => reject("Timeout 2"), 3000);
});
myPromise4
  .then(res => console.log("Response:", res))
  .catch(err => console.log("error:", err));                //error: Timeout2
```

Code sample: https://codesandbox.io/s/js-promise-xekte

Promise consumer - Promise.all

Promise.all(iterable) return promise.

Promise.all will return a promise once all the promises are resolved. If one of the promises is failed, it will reject immediately with the error value of the promise that is rejected regardless of whether the remaining promises are resolved or not.

Promise.all

```
function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}
Promise.all([
  delay(2000).then(() => "second."),
  delay(1000).then(() => "first.")
])
  .then(function(txt) {
    console.log(txt);                        //["second", "first"]
  })
  .catch(function(err) {
    console.log("error:", err);
  });
```

Code sample: https://codesandbox.io/s/js-promiseall-promiserace-e3lx0

Promise consumer - Promise.race

Promise.race(iterable) return promise.

Like *Promise.all,* it takes an iterable of promises, but instead of waiting for all of them to finish, it waits for the first result (or error) and returns a promise.

Promise.race

```
function delay1(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}
Promise.race([
  delay1(4000).then(() => "second."),
  delay1(3000).then(() => "first.")
])
  .then(function(txt) {
    console.log(txt);                    //first
  })
  .catch(function(err) {
    console.log("error:", err);
  });
```

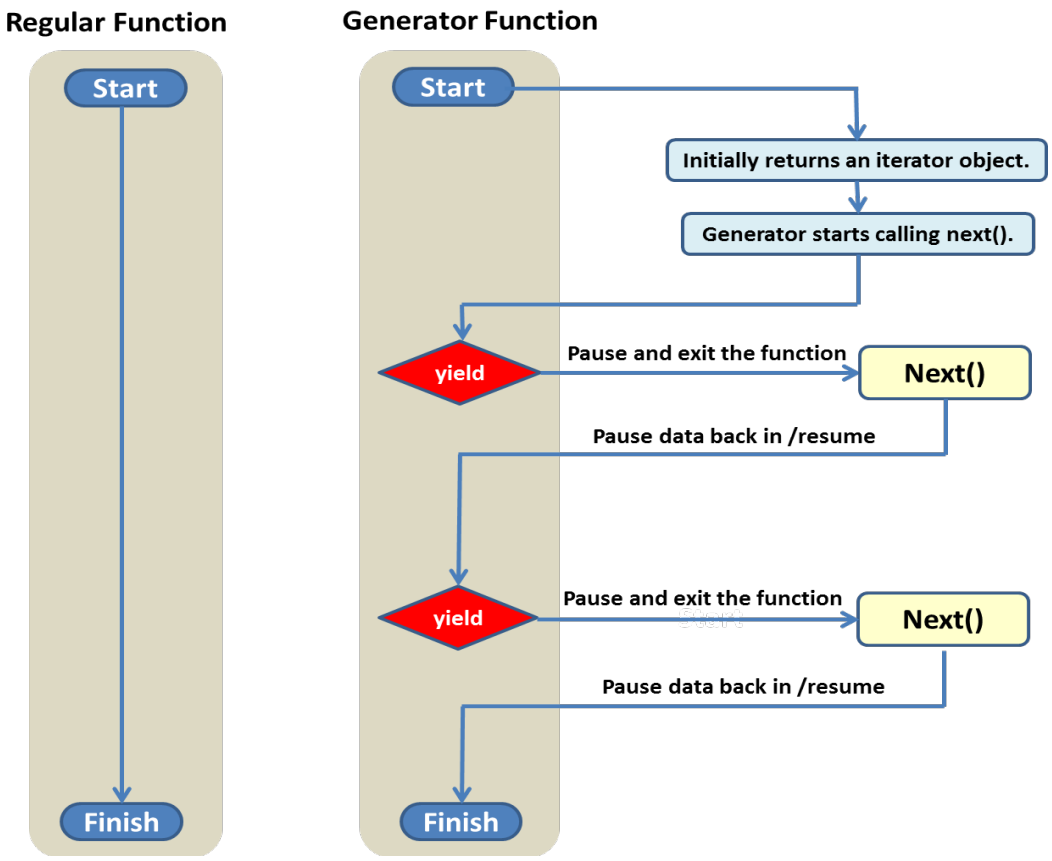
Code sample: https://codesandbox.io/s/js-promiseall-promiserace-e3lx0

Function	Syntax	Description
Promise	new Promise(function (resolve, reject) {…} return promise	The promise global creates promise objects that have the two methods - then and catch.
promise.then	promise.then ([onFulfilled], [onRejected]) returns promise	The promise.then() method accepts an onFulfilled and onRejected callback. The function returns a promise that is resolved by the return value of the onFulfilled or onRejected callback. Any errors thrown inside the callback rejects the new promise with that error.
promise.catch	promise.catch (onRejected) returns promise	The catch() method returns a Promise and deals with rejected cases only if the callback throws an error. You must explicitly re-throw an error inside a catch callback.
Promise.resolve	Promise.resolve ([value|promise]) returns promise	The Promise.resolve() function is a for creating a promise with a given value. Passed a promise as the argument to Promise.resolve(), the new promise is bound to the promise and will be fulfilled or rejected accordingly.
Promise.reject	Promise.reject([reason]) returns promise	The Promise.reject() function is a function for creating a rejected promise with a given reason.
Promise.all	Promise.all(iterable) returns promise	The Promise.all() function accepts a single argument, which is an iterable (such as array) and returns a promise that is resolved only when every promise in the iterable is resolved. The returned promise is fulfilled when every promise in the iterable is fulfilled.
Promise.race	Promise.race(iterable) returns promise	This method also accepts an iterable of promises to monitor and returns a promise, but the returned promise is settled as soon as the first promise is settled.

Generators (ES6)

Generators are functions that can be paused and resumed instead of executing all the statements of the function in one shot. They simplify iterator-authoring using the *function** and *yield* keywords. A function declared as *function** returns a *Generator* object.

The Generator function can be implemented in ES3/ES5 through the closures, returning an iterator object that will remember its own state. However, it is more complicated to implement to be adapted to each use case. We recommend using ES6 generator syntax.



In the following example, we will create a generator named genFunc. We are pausing in the middle of the function using the *yield* keyword. When we call the function, it won't be executed until we iterate over the function, and so here, we iterate over it using the *next* function. The *next* function will run until it reaches a *yield* statement. Here it will pause until another *next* function is called. This, in turn, will resume and continue executing until the end of the function.

```
function* genFunc() {
  console.log("First");              // First
  yield;
  console.log("Second");          //Second
  yield;
  console.log("Third");          //Third
}
let gen1 = genFunc();
console.log(gen1.next());           //{ value=undefined   done=false}
console.log(gen1.next());           //{ value=undefined   done=false}
console.log(gen1.next());           //{ value=undefined   done=true }
```

Code sample: https://codesandbox.io/s/js-generator-nq18f

generator example 2

```
function* foodGenerator() {
  console.log("First"); //output:  First
  yield "Sushi"; //return "Sushi" to caller

  console.log("Second"); //output:  Second
  yield "Hamburger"; //return  "Hamburger" to caller

  console.log("Third"); //output: Third
  return "Pizza"; //return  "Pizza" to caller
}

let gen2 = foodGenerator();
console.log(gen2.next()); // {value="Sushi", done=false}
console.log(gen2.next()); // {value="Hamburger", done=false}
console.log(gen2.next()); // {value="Pizza", done=true}
```

Code sample: https://codesandbox.io/s/js-generator-nq18f

Redux-Saga is based on the ES6 generator. It is mandatory to understand the generator concept to use the Redux-Saga library.

async/await (ES8)

async/await functions expand on Promises+Generators to make asynchronous calls even cleaner. The *async* function declaration defines an asynchronous function, which returns an asynchronous function object. An asynchronous function operates asynchronously via the event loop, using an implicit Promise to return its result. But the syntax and structure of your code using *async* functions are still like using conventional synchronous functions:

```
async function doSomething() {
  const   response1 = await doSomething1(),
  const   response2 = await doSomething2(response1),
  const   response3 = await doSomething3(response2);
}
```

await effectively makes each call appear as if it's a synchronous operation while not holding up JavaScript's single processing thread.

async

We will start with the *async* keyword. It is placed before function, like below:

```
async function f(){  return "promise 1 done";}        f().then(alert);
                                                                  // "promise 1 done!"
```

Code sample: https://codesandbox.io/s/js-async-await-z26x3

The keyword *async* before a function means that a function always returns a Promise. If the code has return <non-promise> in it, then JavaScript automatically wraps it into a resolved promise with that value. For instance, the code above returns a resolved Promise with the result "promise 1 done".

We could explicitly return a Promise that would result in the same:

```
async function f() { return Promise.resolve("promise 2 done");}     f().then(alert);
                                                                                    // "promise 2 done!"
```

Code sample: https://codesandbox.io/s/js-async-await-z26x3

async ensures that the function returns a *Promise*, wraps non-promises in it. And there is another keyword *await* that works only inside *async* functions.

await

The *await* syntax works only inside *async* functions:
let value = await promise;

The keyword *await* makes JavaScript wait until that Promise settles and returns its result. Below is an example with a Promise that resolves in 2 seconds:

```
async function myFunc() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("promise 3 done!"), 2000)
  })
  let result = await promise; // wait till the promise resolves (*)
  console.log(result)                                          // "promise 3 done!"
}

myFunc()
console.log("Please wait")                                  //while waiting, this is executed
```

Code sample: https://codesandbox.io/s/js-async-await-z26x3

The function execution “pauses” at the line (*) and resumes when the Promise settles. So the code above shows “promise 3 done!” in 2 seconds. *await* literally makes JavaScript wait until the Promise settles and then go on with the result. However, that doesn’t cost any CPU resources because the JavaScript engine can execute other tasks while waiting.

Solution	Description	Pros	Cons
Native Javascipt with callback	Uses native Javascript callback and clousure to handle async events.	Highest performance	Difficult to track in case of nested callbacks.
Promise (ES6)	The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.	It can handle multiple async operations easily and provide better error handling than callbacks and events. Being able to wait for multiple promises at the same time.	A bit difficult API syntax.
Generators (ES6)	halting function execution and yielding values . Generator function are executed yield by yield.	Easy to test. A generator function is similar to a state machine.	Generators are one-time access only. To generate the values again, a new generator object has to be made.
Async/Await (ES8)	built on top of promises + generatos. Functions with the async keyword will always return a promise.	It allows applications to retain a logical structure that resembles synchronous but handles asynchronous. It doesn't use callbacks	It makes asynchronous code less obvious. The await keyword can only wait for promises one at a time, not several at once.

Asynchronous process summary

2.9 Making HTTP requests

Fetch and Axios are promise-based JavaScript APIs that lets us use Ajax requests without using complicated XMLHttpRequest API.

Fetch API

The Fetch API provides an interface for accessing and manipulating parts of the HTTP sequence, such as requests and responses. It also provides a global fetch( ) method that provides a simple, logical way to fetch resources asynchronously across the network.

The fetch( ) method takes one mandatory argument that indicates the path to the resource you want to fetch, and it returns a Promise that resolves with an object of the built-in Response class as soon as the server responds with headers.

The code below shows a basic fetch request in which we are fetching a JSON file across the network.

```
fetch("https://examples/example.json")      // make the request
  .then(response => response.json())        // json method on the response
  .then(data => {console.log(data)}).       // handle success
  .catch(error => console.error(error)).    //handle error
```

When receiving data without errors and assuming that we expect a JSON response, we first need to call the json( ) method to transform the Response object into an object that we can interact with.

There are a number of methods available to define the body content in various formats: *response.json(), response.text(), response.formData(), response.blob(), response.arrayBuffer()*

For more details Fetch API information:
https://developers.google.com/web/updates/2015/03/introduction-to-fetch
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Axios library

Axios is a JavaScript library that helps us make HTTP requests to external resources and supports the Promise API.

Unlike Fetch API, Axios does not come as a native JavaScript API, but there are many advantages over fetch API.

- It automatically transforms request and response data.
- It has built-in support for download progress.
- It has the ability to cancel requests.
- It can be used to intercept HTTP requests and responses.
- It enables client-side protection against CSRF(Cross-Site Request Forgery).

On the other hand, Fetch API supports lower-level operations and more flexible than Axios.

The code below shows a basic fetch request in which we are fetching JSON data across the network. With Fetch API, we need to deal with two promises. With Axios, we can directly access the JSON result inside of the response object data property.

```
axios.get('https://examples/example.json')          //make the request expecting JSON
  .then((response) => {console.log(response);})     //handle success
  .catch((error) => {console.log(error);}).         // handle error
```

Axios also provides functions to make other network requests as well, matching the HTTP verbs that you wish to execute:

axios.request, axios.get, axios.delete, axios.head, axios.options, axios.post, axios.put, axios.patch

The response from a request contains the following information:

response.data, response.status, response.statusText, response.headers, response.config, response.request

For details, refer to the official information:
https://github.com/axios/axios

2.10 Modules

ES6 supports modules using *import/export* syntax. You can use the *export* keyword to export parts of code to other modules. And when you have a module with *exports*, you can access the functionality in another module by using the *import* keyword.

Named exports

Named exports are useful to export several values. During the import, you can use the same name to refer to the corresponding value. Exported functions can be used by other modules by using import syntax. It is also possible to export all the variables in one area by using this syntax for listing. You can import all of the module's functionality by using import * as <Namespace> syntax.

(File name: test.js)

```
// export an array
export let months = ["Jan","Feb","Mar","Apr","Aug","Sep","Oct","Nov","Dec"];

// export a constant
export const year = 2050;

// export function
export const hello1 = function() {console.log("Hello World 1");};

// export arrow function
export const hello2 = () => console.log("Hello World 2");
```

(File name: index.js)

```
import { months, year } from "./test.js";
import { hello1, hello2 } from "./test.js";

console.log(months);                  //{"Jan","Feb","Mar".....}
console.log(year);                    //2050

console.log(hello1);                  //function hello1() {}
hello1();                             //hello World 1

console.log(hello2);                  //function hello2() {}
hello2();                             //hello World 2
```

Code sample: https://codesandbox.io/s/js-module1-l34x3

Default export

The other type of export is a default export. A module can contain only one default export. A default export can be a function, a class, an object, or anything else. This value is considered as the “main” exported value since it will be the simplest to import.
This is a common pattern for libraries because you can import the library without specifying each library.

(File name: test.js)

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

(File name: index.js)

```
import sum from "./test.js";

console.log("10 + 20 = " + sum(10, 20));
```

It is common for a module using mixing of both named exports and default exports.

```
import ReactDOM from ‘react-dom’;

ReactDOM .render(
  // …
);
```

In case of using only *render( )* function , the named render( ) function can be imported like this.

```
import {render} from ‘react-dom’;

render(
  // …
);
```

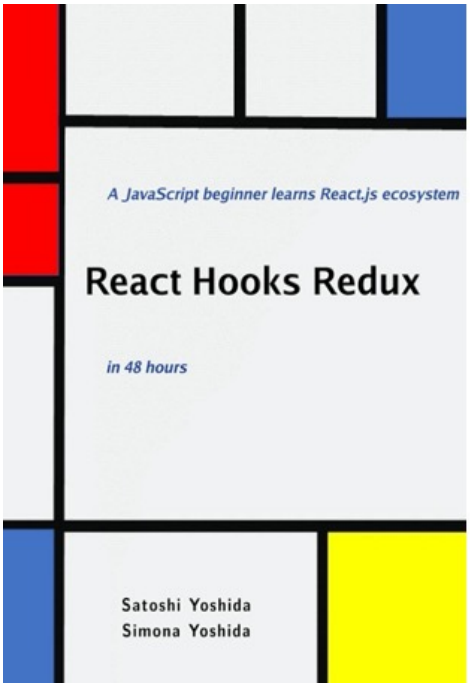
The export implementation to achieve above is like this.

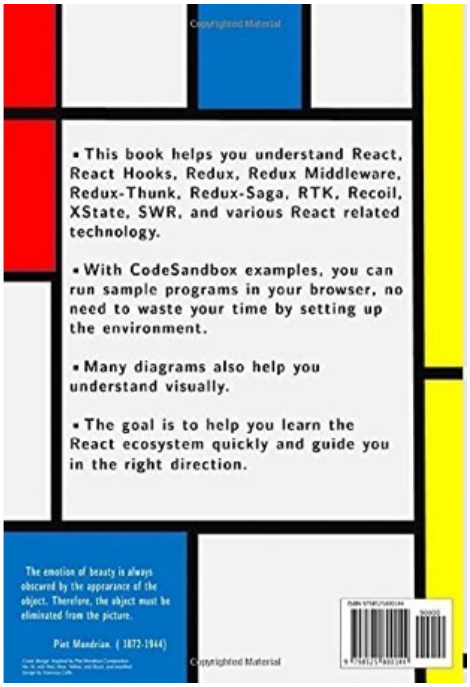
```
export const render = (component ,target) => {
  // …
};

const ReactDom = {
render,
 // … other functions
};

export default ReactDOM;
```

If this book helps you understand JavaScript, we would recommend the following book for you. You will learn React, Hooks, and Redux in a structured learning method.

React Hooks Redux in 48 hours:
A JavaScript beginner learns React.js ecosystem

by Satoshi Yoshida, Simona Yoshida





ISBN-13 : 979-8525800144

Learning React with CodeSandbox

This book helps you learn ReactJS, React Hooks, Redux, Redux Middleware, Redux-Thunk, Redux-Saga, RTK, Recoil, XState, SWR, React Router, Jest, React Test Library, and various React related technology.

With 99 CodeSandbox code examples we have created, you can run sample programs in your browser, no need to waste your time by setting up the environment. The goal is to help you learn the React ecosystem quickly and guide you in the right direction.

Who this book is for:

- JavaScript beginners: some experience in Web design using HTML, CSS, and JavaScript.
- JavaScript programmers who are not familiar with functional programming and ES6.
- Busy developers who do not have time to go through React/Redux official documentations.
- Non-Web software engineers, managers, or system architects: at least know one of the high-level languages.

This book is not for:

- No experience in any programming.
- React experts: Please refer to the React and Redux official docs.

Learning schedule:

- Overview (one hour reading)
- Intermediate JavaScript (6 hours reading and exercise)
- React Fundamentals (10 hours reading and exercise)
- React Hooks (10 hours reading and exercise)
- React Router (2 hours reading and exercise)
- Test-Driven Development (2 hours reading and exercise)
- Redux, Middleware, Thunks, Saga, and RTK (12 hours reading and exercise)
- Recoil (2 hours reading and exercise)
- Various State Management Libraries (one hour reading and exercise)
- APPENDIX - various libraries/frameworks (2 hours reading and exercise)

CodeSandbox environment:

You will need a Windows PC, Mac, Chromebook, Android device, iOS device, or Linux-based device. We have tested all code on Chrome, Firefox, and Edge browser.

About an author

Satoshi Yoshida started his professional carrier in the 1980s, developing CPU circuits, mobile phone modem LSIs and firmware. He also had management and architect experiences in various multinational companies in Japan, Canada, France, and Germany. He has 16 patents in wireless communication and has been working as a freelance consultant (wireless hardware, IoT, smart metering, UI software) since 2018.

About a co-author

Simona Yoshida has been working on web-based UI software and e-commerce platform since 2015. Sample codes in chapter 2, chapter 3, and chapter 4 were written and tested by her. In her spare time, she takes care of dogs and cats from animal shelters.

Contact information: react3001@gmail.com

Recommendation by BookAuthority.org:

- **14 Best New React Hooks Books To Read In 2021**

As featured on CNN, Forbes and Inc – BookAuthority identifies and rates the best books in the world, based on recommendations by thought leaders and experts.

https://bookauthority.org/books/new-react-hooks-books

How to order the "React Hooks Redux in 48 hours"

You can order it from Amazon in your country/region.

Paperback ASIN : B097SSBP2V
eBook ASIN : B0987SZHW4

Amazon.com	(USA)
Amazon.co.uk	(UK)
Amazon.de	(Germany)
Amazon.fr	(France)
Amazon.es	(Spain)
Amazon.it	(Italy)
Amazon.nl	(Netherlands)
Amazon.co.jp	(Japan)
Amazon.in	(India)
Amazon.ca	(Canada)
Amazon.com.br	(Brasil)
Amazon.com.mx	(Mexico)
Amazon.com.au	(Australia)