React JS

anandkumar.training@gmail.com

Javascript Refresher

Writing JavaScript

The <script> Tag

- JavaScript programs can be inserted almost anywhere into an HTML document using the <script> tag. The <script> tag contains JavaScript code which is automatically executed when the browser processes the tag.

- Attributes
  - type=”text/javascript”
  - language=”javascript”
  - src=”fileName.js”

- A single <script> tag can’t have both the src attribute and code inside.

- Hiding the Script : Comment the code written inside <script> tag using HTML comments.

- If src is set, then contents written inside <script> are ignored.

JavaScript Basics

- We can write JavaScript in head section, body section and in an external file.
- JavaScript is case sensitive.
- It is good practice to end statement with semicolon.
- Single Line Comment : //this is comment
- Multiline Comment : /* this is comment */
- Nested comments are not supported.
- JavaScript is dynamically typed language (i.e. data types are available but variables are not bound to them. Type of variable is decided based on the value assigned to it)
- Semicolon at the end of statements are optional, but it is good practice to write it to indicate end of statement.
-

The Strict Mode

- ECMAScript 5 (ES5) added new features to the language and modified some of the existing ones. To keep the old code working, most such modifications are off by default. You need to explicitly enable them with a special directive: "use strict".
- "use strict" must be at the top of your scripts, otherwise strict mode may not be enabled.
- When it is located at the top of a script, the whole script works the “modern” way. If it is written inside a function, strict mode is enabled only in that function.
- For Ex. :

```
<script>
        “use strict”;
        //rest of the code
</script>
```

JavaScript Basics

Variables

- A variable is name container for value in memory.
- In modern Javascript, variables are declared using **let** keyword. In old Javascript, **var** keyword is used to declare variable.
- For Ex. : let myVar1; let myVar2 = 10;

var	let
function-scoped or global-scoped	block-scoped
redeclaration is allowed	redeclaring a variable results in an error
“var” variables can be declared below their use	“let” variables can not be declared below their use

Variable Naming Rule

- The name must contain only letters, digits, or the symbols $ and _.
- The first character must not be a digit.
- Reserved words cannot be used as variable names

The document Object

- When html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

- The document object is accessed with: **window.document** or just **document**

- **Methods**
  - getELementById(“elementId”)
  - getElementsByClassName(“nameOfClass”)
  - getElementsByName(“elementName”)
  - getElementsByTagName(“tagName”)

Functions

- Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

- Function Declaration
  - function functionName(param1, param2, ..., paramN) { function body }

- Calling the function
  - functionName(argument1, argument2, ..., argumentN);

- Variables
  - A variable declared inside a function is only visible inside that function.
  - The function has full access to the outer variable. It can modify it as well.
  - The outer variable is only used if there’s no local one. If a same-named variable is declared inside the function then it shadows the outer one.
  - Variables declared outside of any function, are called global. Global variables are visible from any function (unless shadowed by locals).

Functions

- Parameters
  - We can pass arbitrary data to functions using parameters.
  - A parameter is the variable listed inside the parentheses in the function declaration (it's a declaration time term).
  - An argument is the value that is passed to the function when it is called (it's a call time term).
  - If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

    For Ex. :

```
function sayHello(personName)
{
        alert("Hello" + personName); // Hello undefined
}
sayHello(); // no argument is passed
```

Functions

- Parameters
  - Default Value for parameter in Function Declaration

```
function sayHello(personName = “Name is not provide”)
{
        alert(“Hello” + personName);
}
sayHello(); // no argument is passed
```

  - We can provide complex expression for default parameter, which is only evaluated and assigned if the parameter is missing.

```
function sayHello(personName = anotherFunction())
{
        alert(“Hello” + personName);
}
sayHello(); // no argument is passed
```

Functions

- Parameters
  - Modern JavaScript engines support the nullish coalescing operator ??
  - ?? returns the first argument if it’s not null/undefined. Otherwise, the second one.

```
function showCount(count)
{
  // if count is undefined or null, show "unknown"
  alert(count ?? "unknown");
}
showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

- Returning a value
  - A function can return a value back into the calling code as the result. A function with an empty return or without it returns undefined.
  - Never add a newline between return and the value.

Function Expression

- Since Javascript treats function as a special kind of value, there is another way to create a function, called as Function Expression. It allows us to create a new function in the middle of any expression.

- For ex.

```
let sayHello = function() {
    alert( "Hello" );
}; //note the semicolon... it is required because it is expression
alert(sayHello) // Function: sayHello
alert(sayHello()) // Calls the function
```

- We can copy a function to another variable (no matter how the function is created)
  For ex. : let anotherFunctSayHello = sayHello

- A **Function Expression** is created when the execution reaches it and is usable only from that moment. A **Function Declaration** can be called earlier than it is defined.

Arrow Function

- There's another very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions", because it looks like this: let func = (arg1, arg2, ..., argN) => expression;
- This creates a function func that accepts arguments arg1..argN, then evaluates the expression on the right side with their use and returns its result. In other words, it's the shorter version of:

```
let func = function(arg1, arg2, ..., argN) {
    return expression;
};
```

- If we have only one argument, then parentheses around parameters can be omitted, making that even shorter. For example: let double = n => n * 2;
- If there are no arguments, parentheses are empty, but they must be present:

```
let sayHi = () => alert("Hello!");
```

-

Callback Functions (Callbacks)

- We can pass a function as an argument/value to other function.
- For Ex. :

```
function ask(question, yes, no) {
    if (confirm(question)) yes();
    else no();
}

function showOk() {
    alert( "You agreed." );
}

function showCancel() {
    alert( "You canceled the execution." );
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

- The arguments showOk and showCancel of ask are called **callback functions** or just **callbacks**. The idea is that we pass a function and expect it to be **“called back” later if necessary**.

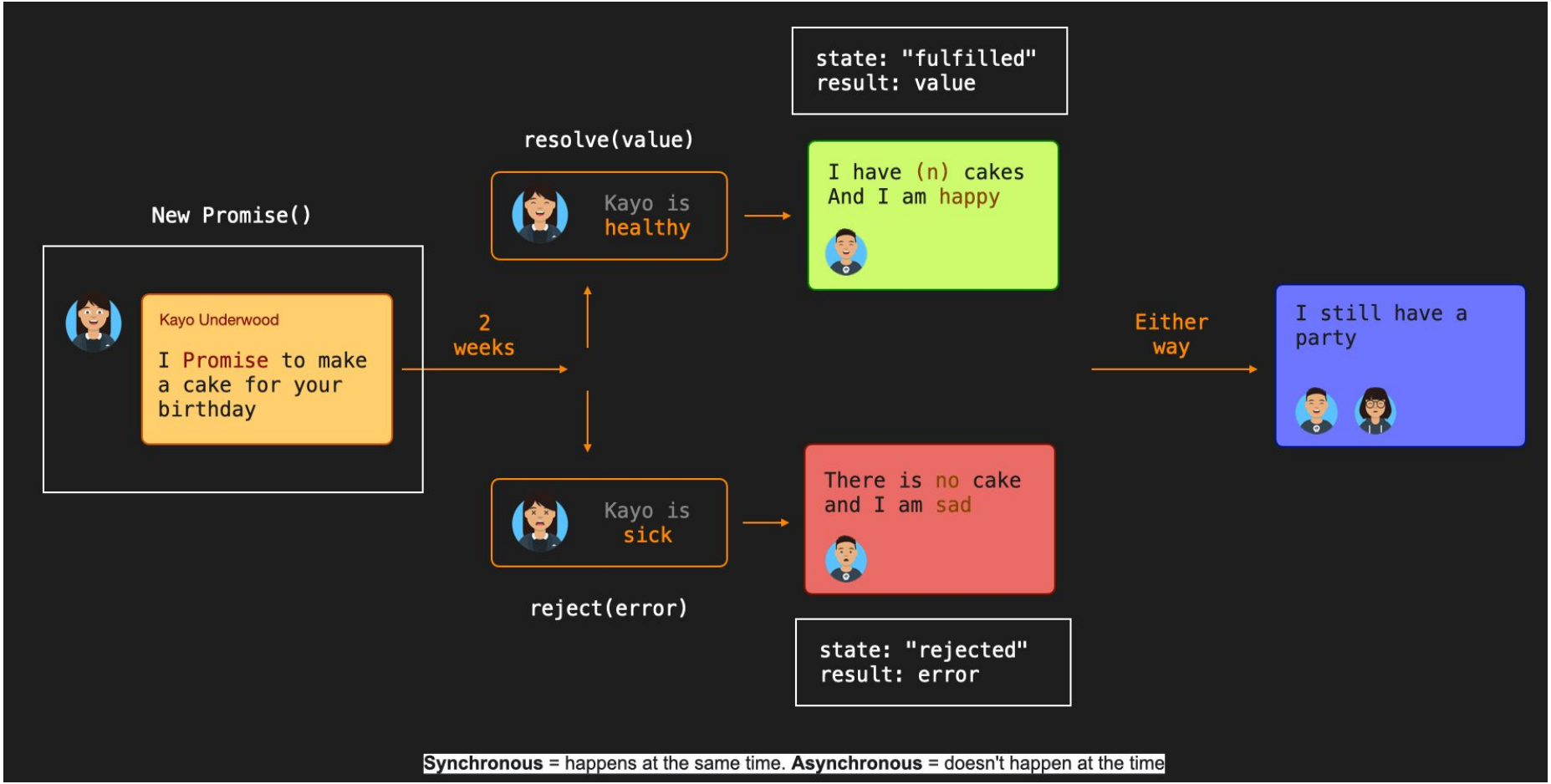
Callback Function Use Case

```
<html>
   <head>
      <title>Script Demo</title>
      <script>
         function loadScript(src)
         {
                              let     script     =
document.createElement('script');
            script.src = src;

document.head.append(script);
         }
         loadScript('./myscript.js');
         alert("Test the Flow");
         sayHello();
      </script>
   </head>
   <body></body>
</html>
```

```
<html>
   <head>
      <title>Script Demo</title>
      <script>
         function loadScript(src, callback) {
            let script = document.createElement('script');
            script.src = src;
            script.onload = () => callback(script);
            document.head.append(script);
         }
         function callSayHello(script) {
            alert(`Wow, the script ${script.src} is loaded`);
            sayHello();
         }
         loadScript('./myscript.js',callSayHello);
         /*loadScript('./myscript.js', script => {
               alert(`Cool, the script ${script.src} is loaded`);
               sayHello();
         });*/
      </script>
   </head>
   <body></body>
</html>
```

Promise - Analogy



Promise - Introduction

- A “producing code” that does something and takes time. For instance, some code that loads the data over a network.

- A “consuming code” that wants the result of the “producing code” once it’s ready. Many functions may need that result.

- A promise is a special JavaScript object that links the “producing code” and the “consuming code” together. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

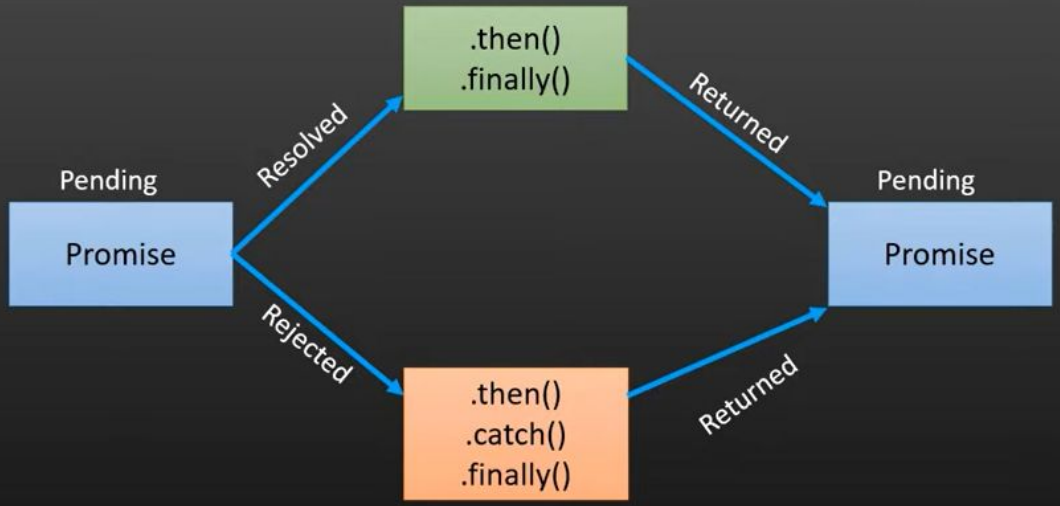
Promise - Syntax, Executor and States

- The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

- The function passed to new Promise is called the executor. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result.
- Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor.
- When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:
  - resolve(value) — if the job is finished successfully, with result value.
  - reject(error) — if an error has occurred, error is the error object.
- The promise object returned by the new Promise constructor has these internal properties:
  - state — initially "pending", then changes to either "fulfilled" when resolve is called or "rejected" when reject is called.
  - result — initially undefined, then changes to value when resolve(value) is called or error when reject(error) is called.

How Promise works

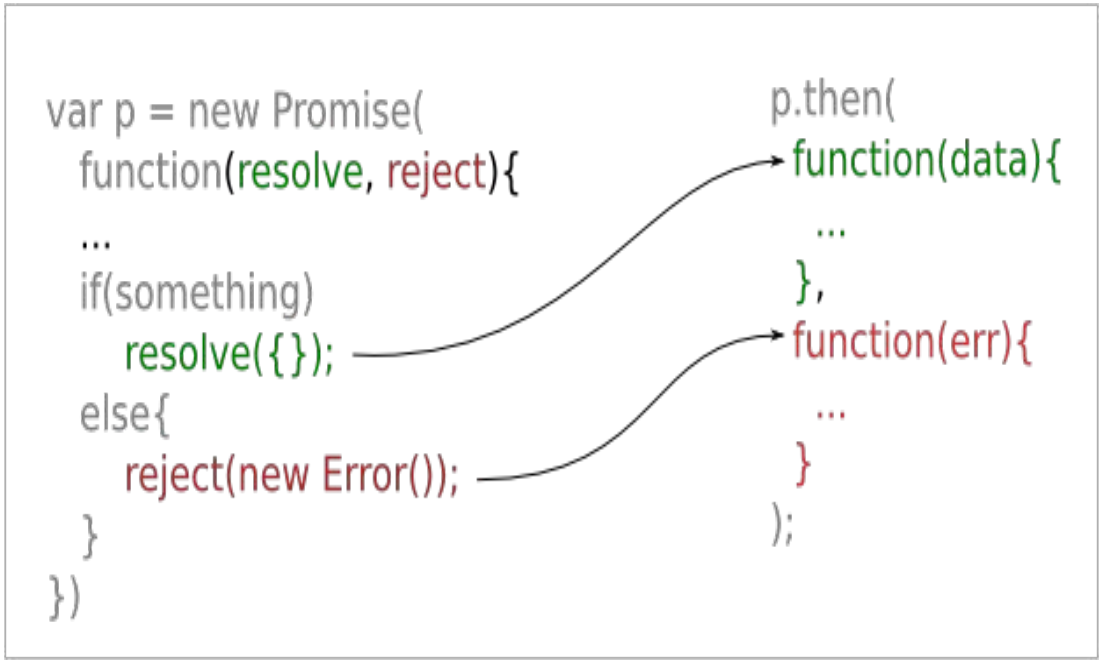


- A pending promise can either be Resolved with a value or Rejected with a reason (error).
- When either of these options happens, the associated handlers queued up by a promise's *then* method are called.
- A promise is said to be settled if it is either Resolved or Rejected, but not Pending.

Promise - Consumer

- A Promise uses an executor function to complete a task (mostly asynchronously). A consumer function (that uses an outcome of the promise) should get notified when the executor function is done with either resolving (success) or rejecting (error).

- The handler methods, .then(), .catch() and .finally(), help to create the link between the executor and the consumer functions so that they can be in sync when a promise resolves or rejects.



Promise - Example

Producer

```
function loadScript(src)
 {
   return new Promise(function(resolve, reject)
   {
     let script = document.createElement('script');
     script.src = src;
     script.onload = () => resolve(script);
     script.onerror = () => reject(new Error(`Script
load error for ${src}`));

   document.head.append(script);
  });
}
```

Consumer

```
let promise = loadScript("./myscript.js");

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another
handler...'));
```

async await

- We use the async keyword with a function to represent that the function is an asynchronous function. The async function returns a promise. The syntax of async function is:
  async function name(parameter1, parameter2, ...paramaterN) {
  // statements
  }

- The await keyword is used inside the async function to wait for the asynchronous operation. The syntax to use await is:
  let result = await promise;

- The use of await pauses the async function until the promise returns a result (resolve or reject) value.

```
<script>
    async function example() {

        let promise = new Promise((resolve, reject) => {
            setTimeout(() => resolve("done!"), 2000)
        });

        let result = await promise; // wait until the promise resolves (*)

        alert(result); // "done!"
    }
    example();
</script>
```

React Introduction

Introduction

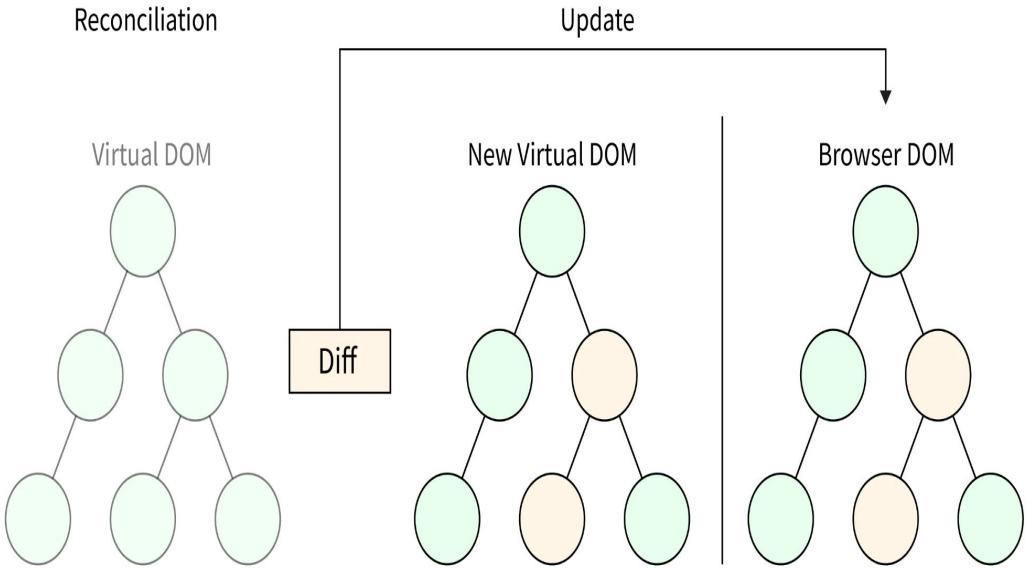
What is React?

- ReactJS is a declarative, open source JavaScript library for building reusable UI components.
- Created by Jordan Walke at Facebook in 2011
- Available for public from May 2013
- Used by Facebook for WhatsApp and Instagram
- Latest version is 18.2
- With React developers can create reusable UI components. These components can be used any number of times anywhere in the application. This reduces the time for debugging and rewriting.
- ReactJS uses virtual DOM based mechanism to fill data in HTML DOM.

Introduction

Virtual DOM

- Virtual DOM is a lightweight copy of the Real DOM stored in the memory. React uses the virtual DOM represented as a tree.



Setup and Project Creation

- Download and install Node.js from nodejs.org (Download LTS version)

- **Open Command prompt and type**
  - **node --version** to check if node is installed successfully

  - **npx create-react-app prjName** : this command will create a new subfolder with a basic React project setup (i.e., with various files and folders) in the place where you ran it

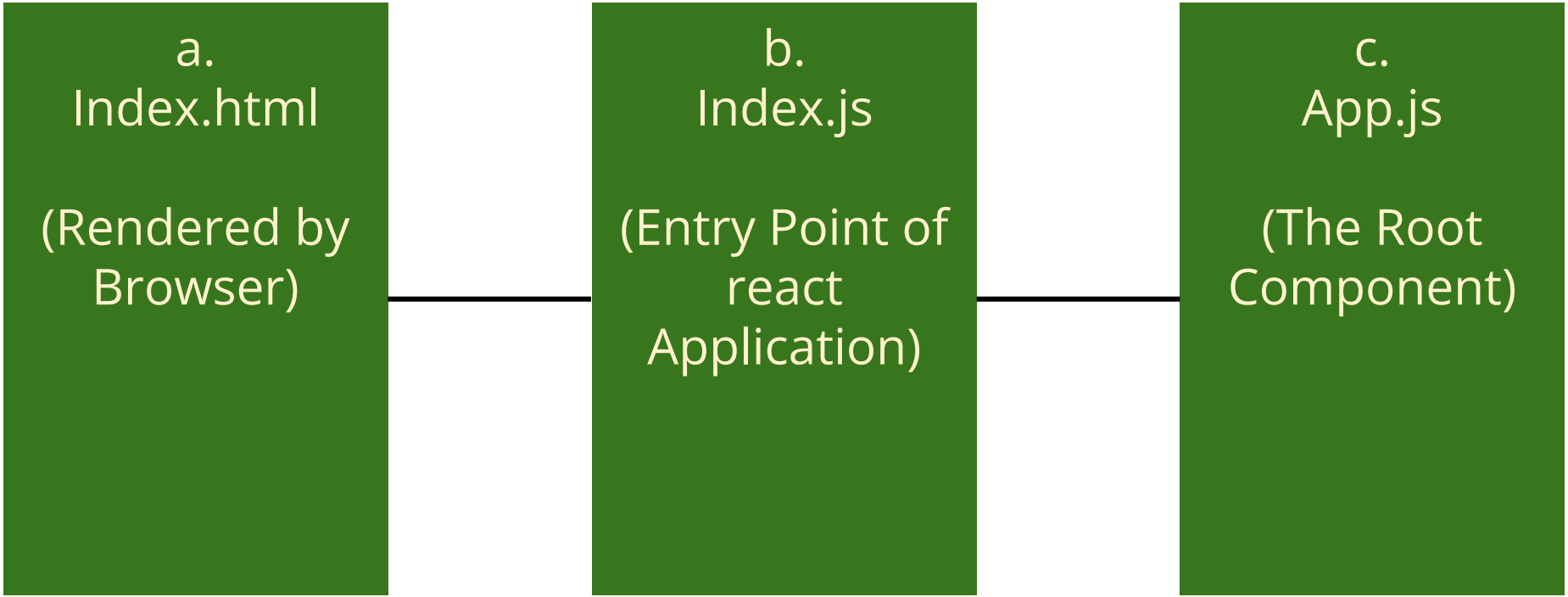
  - **npm start** : This will start a built-in development server and open the preview page. If that doesn't happen, you can manually open a new tab and navigate to **localhost:3000**

- The react project typically contains following files and folders:
  - A **src** folder that contains the main source code files for the project
  - An **index.js** file which is the main entry script file that will be executed first
  - An **App.js** file which contains the root component of the application
  - Various styling (*.css) files
  - Other files, like code files for automated tests
  - A **public** folder which contains static files that will be part of the final website. This folder may contain static images like favicons. The folder also contains an **index.html** file which is the single HTML page of this website
  - **package.json** and **package-lock.json** are files that manage third-party dependencies of your project

Test Your Knowledge

- 1. What is React?
- 2. Which advantage does React offer over vanilla JavaScript projects?
- 3. What's the difference between imperative and declarative code?
- 4. How can you create new React projects and why do you need such a more complex project setup?

How React Works?



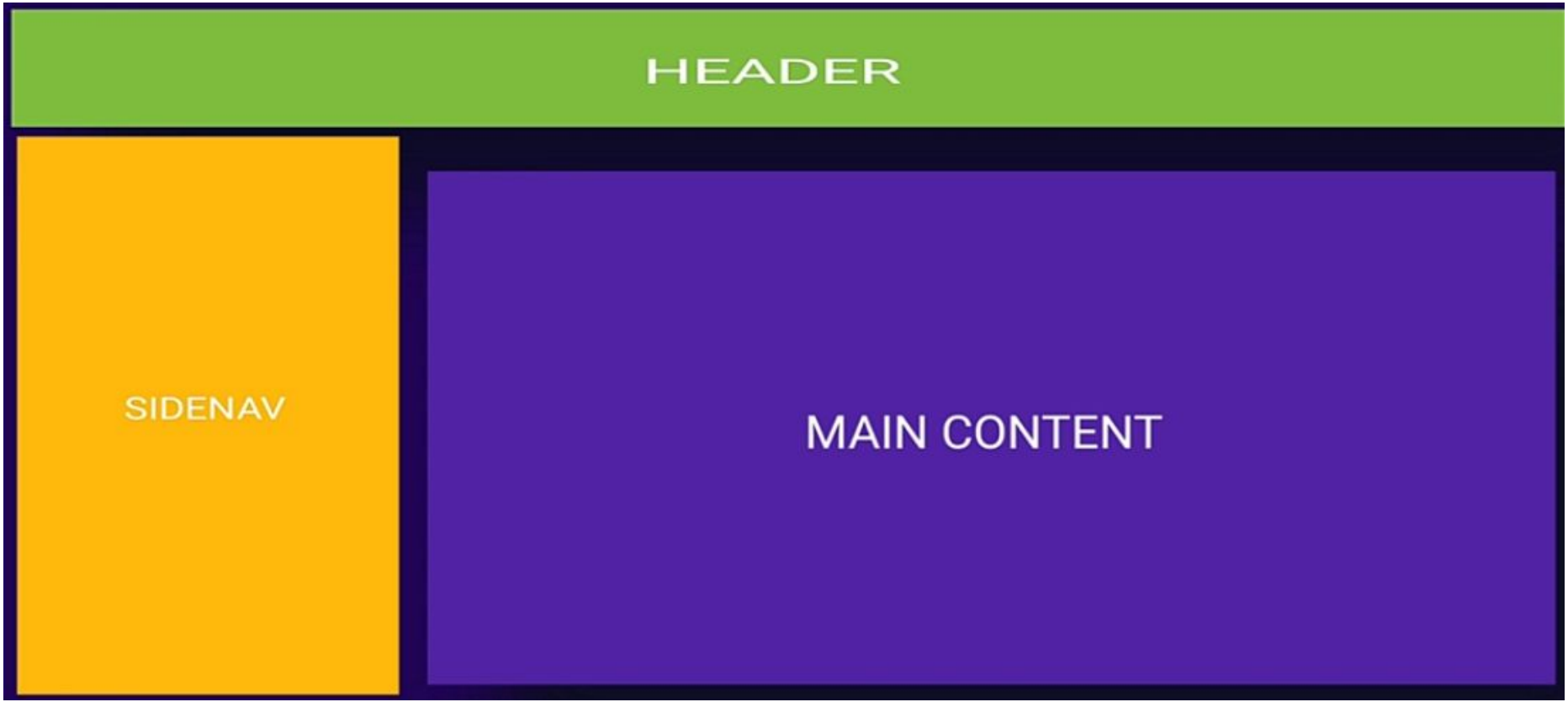
Introduction to Components

- A Component is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components. Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

- Every React component have their own structure, methods as well as APIs. They can be reusable as per your need.

- When working with React, it's especially important to keep your code manageable and work with small, reusable components because React components are not just collections of HTML code. Instead, a React component also encapsulates JavaScript logic and often also CSS styling.

Introduction to Components



Function Components

- In React, function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input and returns what should be rendered.

- The functional component is also known as a stateless component because they do not hold or manage state.

- A valid functional component can be shown in the below example.

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

Class Components

- Class components are more complex than functional components. It requires you to extend from React. Component and create a render function which returns a React element. You can pass data from one class to other class components. You can create a class by defining a class that extends Component and has a render function.

- The class component is also known as a stateful component because they can hold or manage local state.

- Valid class component is shown in the below example.

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>This is main component.</div>
    );
  }
}
```

Introduction to JSX

- JSX (JavaScript XML), is a React extension which allows writing JavaScript code that looks like HTML. In other words, JSX is an HTML-like syntax used by React that extends ECMAScript so that HTML-like syntax can co-exist with JavaScript/React code. The syntax is used by preprocessors (i.e., transpilers like babel) to transform HTML-like syntax into standard JavaScript objects that a JavaScript engine will parse.

- JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then preprocessor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children.

Why JSX?

Create <h1> using JSX

```
function Hello() {  return <h1>Hello, World!</h1>}
```

Create <h1> using Javascript

```
function Hello() {  return React.createElement("h1", {}, "Hello, World!")}
```

Why JSX?

Another Example

Create Nested Elements using JSX

```
NavList() {
return (
        <ul>
                <li>Home</li>
                <li>About</li>
                <li>Portfolio</li>
                <li>Contact</li>
        </ul>
        );
}
```

Create Nested Elements using Javascript

```
NavList() {
return (
    React.createElement
      ("ul",     {},
         React.createElement("li", null, "Home"),
         React.createElement("li", null, "About"),
         React.createElement("li", null, "Portfolio"),
         React.createElement("li", null, "Contact")
       )
   );
}
```

Which syntax is easy and convenient? JSX or JS?

JSX Rules

- A React component name must be capitalized (Pascal Case). Component names that do not begin with a capital letter are treated like built-in components.
- JSX allows you to return only one element from a given component. This is known as a parent element. If you want to return multiple HTML elements, simply wrap all of them in a single , <React.fragments></React.fragments>, <></> or any semnatic tag.
- In JSX, every tag, including self closing tags, must be closed. In case of self closing tags you have to add a slash at the end (for example <img/>, <hr/>, and so on).
- Since JSX is closer to JavaScript than to HTML, the React DOM uses the camelCase naming convention for HTML attribute names. For example: tabIndex, onChnage, and so on.
- "class" and "for" are reserved keywords in JavaScript, so use "className" and "forHTML" instead, respectively.
- To include JavaScript expressions in JSX, you need to wrap them in curly braces. Content between the opening and closing curly braces will be evaluated as JavaScript.
- Comment : {/* This is a JSX comment */}

Props

- props stands for properties. Props are arguments passed into React components.

- Props are passed to components via HTML attributes.

- React Props are like function arguments in JavaScript and attributes in HTML. To send props into a component, use the same syntax as HTML attributes:
  - Ex. : const myElement = <Car brand="Ford" />;

- The component receives the argument as a props object. Use the brand attribute in the component.
  - function Car(props) {
    return <h2>I am a { props.brand }!</h2>;
    }