

# Build a Strong MERN Foundation



# Advanced JavaScript



# A Day in the Life of a MERN Stack Developer

You are working as a MERN stack developer for an organization. Your company has a requirement to build a robust, secure, and performant real-time chat application, which has the following requirements:

- Enable users to see when others join or leave the chat room
- Implement user authentication for secure access to chat rooms
- Optimize the application for enhanced performance and responsiveness
- Support dynamic updates without the need for a full page reload



# A Day in the Life of a MERN Stack Developer

To complete these requirements, you must implement advanced JavaScript concepts.

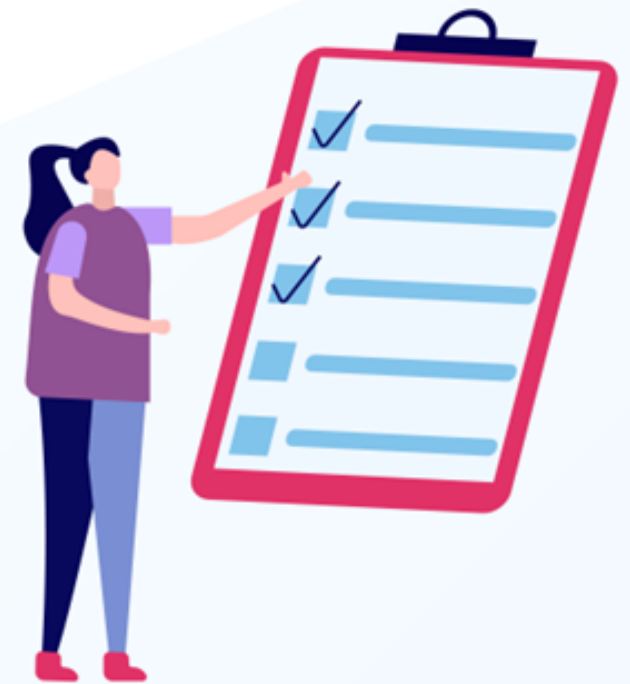
To achieve the above, you will learn a few concepts in this lesson that can help you find a solution for the scenario.



# Learning Objectives

By the end of this lesson, you will be able to:

- Grasp the concepts of IIFEs, callbacks, and closures to write clean, modular, and efficient code
- Differentiate between maps and classes to make informed design decisions in JavaScript
- Identify the use of promises and async to address the challenges faced by asynchronous programming
- Work with AJAX concepts to deliver an efficient user experience in modern web applications



# Learning Objectives

By the end of this lesson, you will be able to:

- 👁 Identify the use of Webpack to provide a streamlined way to optimize assets for web applications
- 👁 Grasp the concept of modern JavaScript to create robust and feature-rich web applications
- 👁 Explain Babel and its features, and integrate it into modern JavaScript





# **Introduction to Advanced JavaScript**

# Advanced JavaScript: Overview

Advanced JavaScript is an in-depth and comprehensive understanding of the JavaScript programming language that goes beyond the fundamentals.

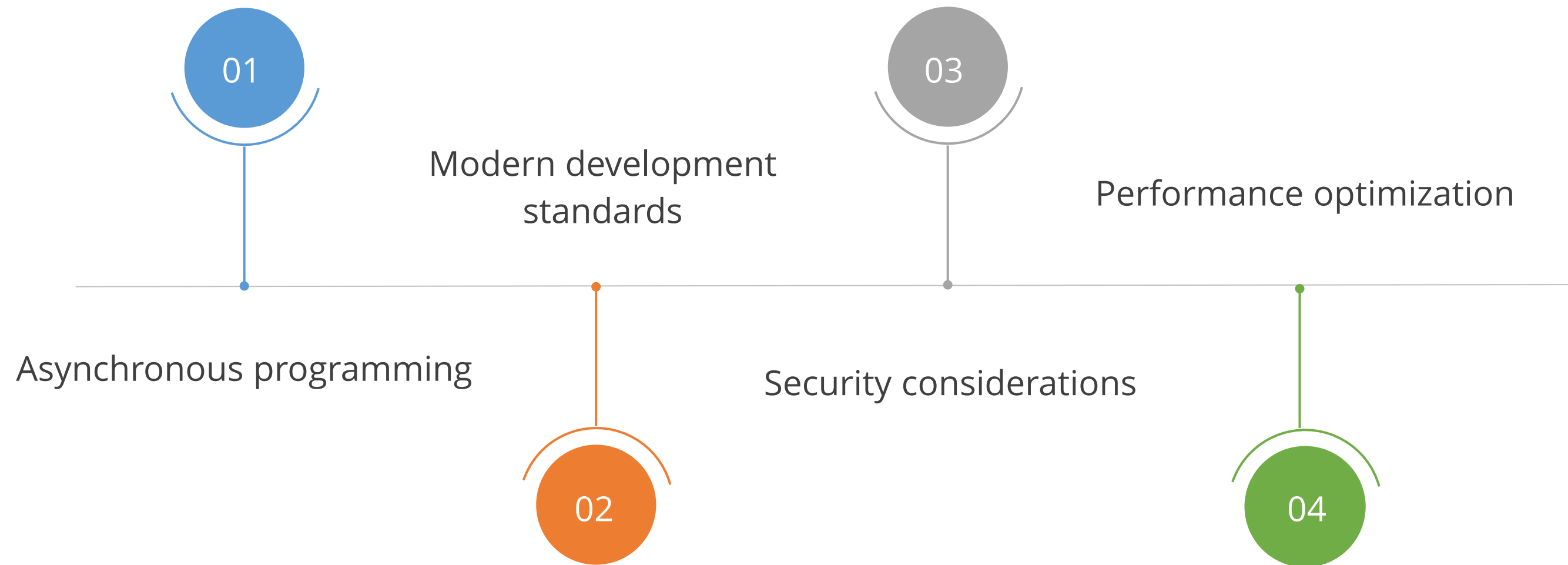


- It can insert dynamic text into HTML and CSS and make the webpage interactive.
- It can be used in front-end and back-end web development.



# Why Advanced JS?

Advanced JS is crucial for many reasons, including:



# Advanced JS: Benefits

It offers several benefits, such as:

Code efficiency

Security awareness



Rich user interface



## **IIFEs, Callbacks, and Closures**

# Immediately Invoked Function Expressions (IIFEs)

The functions that are executed as they are defined are called Immediately Invoked Function Expressions (IIFEs).

## Example:

```
(function() {  
    console.log("Welcome Simplilearns!");  
})();
```

# Immediately Invoked Function Expressions (IIFEs)

IIFE is a way to execute functions as soon as they are created.

It is also known as a regular function.

It is a simple way to isolate variable declarations and is used to achieve data privacy.

# Working with IIFEs

Below is the basic structure of an IIFE:

## Example:

```
(function () {  
    // code here  
})();
```

In this structure, the function is defined inside parentheses, and an additional pair of parentheses immediately follows, invoking the function.

# Callback Function

A function callback is to be executed after another function has finished executing, and it is used while handling an asynchronous operation.

## Example:

```
function greeting(learner, callback) {  
    console.log('Hi' + ' ' + learner+ ',');  
    callback();  
}  
  
function callbackfunction() {  
    console.log('This is callback function');  
}  
  
greeting('Simplilearners', callbackfunction);
```

# Exploring Callback Function

Below is the basic structure of exploring callback functions in JavaScript:

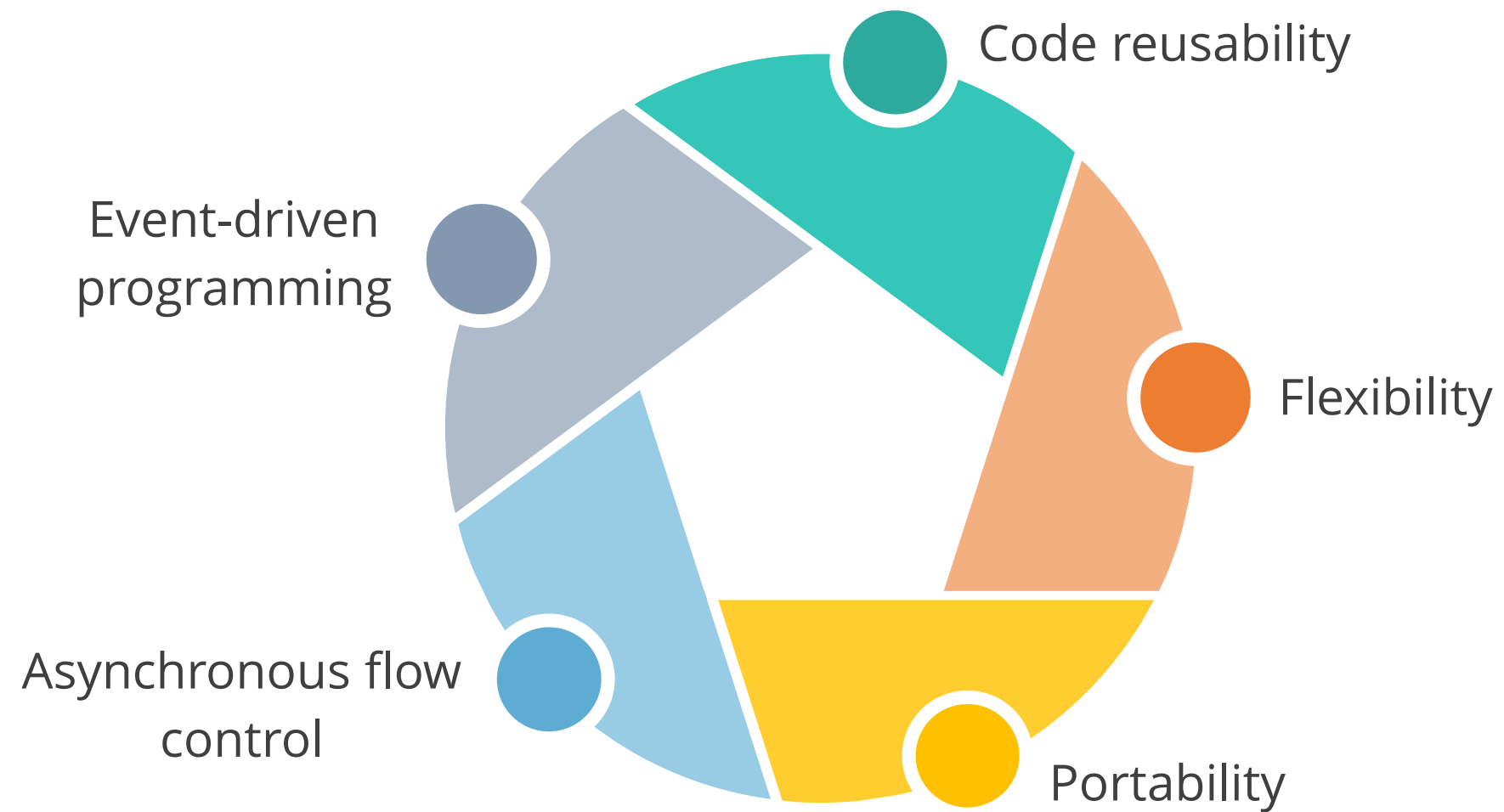
## Example:

```
function doSomethingAsync(callback) {  
    // Simulating an asynchronous task (e.g., fetching data)  
    setTimeout(function () {  
        console.log("Task completed!");  
        // Execute the callback function  
        callback();  
    }, 1000);  
}  
// Using the callback function  
doSomethingAsync(function () {  
    console.log("Callback executed!");  
});
```



# Callback Function: Benefits

Some of the benefits of callback functions are:



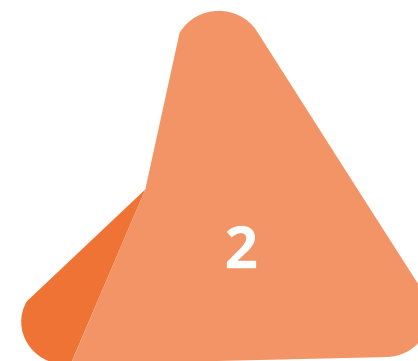
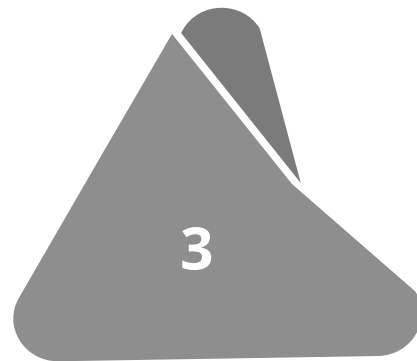
# Closures

It is the context in which a function or code block is executed which can access the variables and parameters of outer functions and global scope.

The closures carry the scope with them at the time of their invocation.




The variables and parameters can be local or global.



The global variables can be local with closures.

# Closures: Benefits



Data privacy and  
encapsulation



Partial applications



State  
maintenance

# Closures: Use Cases

## Private variables

- Closures often create private variables and encapsulate data within a function.
- This helps in avoiding variable pollution in the global scope and provides a form of data privacy.

# Closures: Use Cases

## Function factories

- Closures are employed in function factories to create and return functions with specific behaviors or configurations.
- This is useful for creating reusable and customizable functions.

# Closures: Use Cases

## Event handling

- Closures are frequently used in event handling to maintain context and state.
- The inner function in an event handler often has access to variables from the outer function.

# Assisted Practice



## Working with IIFEs, Callbacks, and Closures

Duration: 15 Min.

### Problem Statement:

You have been assigned a task to implement IIFEs, callbacks, and closures.

# Assisted Practice: Guidelines



Steps to be followed:

1. Write a JS program
2. Execute and verify the working of IIFEs, callbacks, and closures





## IIFEs and Functions

# IIFEs: Practical Use Cases

01

IIFEs are often employed to create a private scope, preventing variables from polluting the global scope.

02

They are fundamental to the module pattern, which allows developers to create modular and reusable code.

03

IIFEs are useful for creating closures to ensure data privacy by restricting access to certain variables from outside the function.

## IIFEs: Practical Use Cases

04

IIFEs help prevent variable hoisting issues by immediately executing the function and establishing a local scope for variables.

05

IIFEs can be utilized for dependency injection, where dependencies are passed as arguments, allowing for flexibility and modularity.

06

IIFEs are used to deal with browser compatibility issues, ensuring that variables and functions do not conflict with existing code.

# Functions: Overview

Functions are essential tools for implementing advanced programming concepts, supporting functional programming, and enhancing code expressiveness.

The logo features the text "JS Functions" in a bold, sans-serif font. The "JS" is white and set against a dark gray square background, while "Functions" is dark gray. The entire logo is centered within a yellow rounded rectangle.

**JS Functions**

# Aspects of Functions

## Arrow functions

Provide a concise syntax for writing functions with implicit returns

## Higher order functions

Enable functional programming paradigms

## Default parameters

Allow users to provide default values for function parameters

## Rest and spread parameters

Allow a function to accept an indefinite number of arguments as an array

# Best Practices for Using Functions

Users can follow these best practices to write efficient and bug-free code while working with functions in advanced JS:

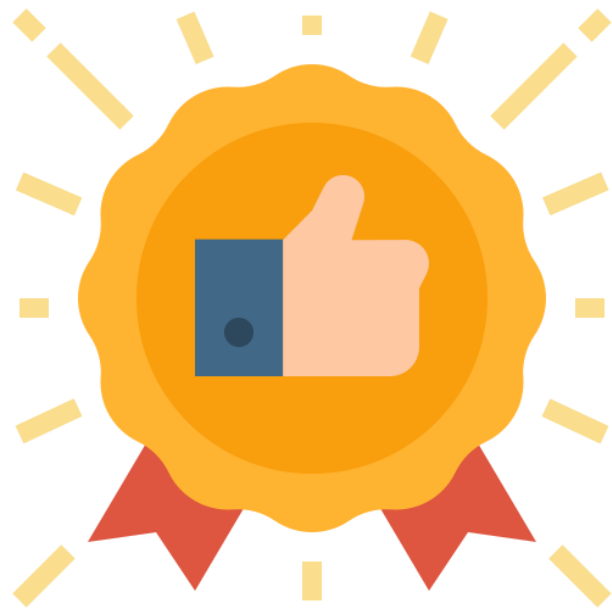
- 1 Use function declarations or expressions
- 2 Use pure functions
- 3 Avoid using global variables
- 4 Avoid callback hell
- 5 Implement error handling



## **IIFEs: Best Practices and Considerations**

# IIFEs: Best Practices

Here are a few best practices and considerations when working with IIFEs:



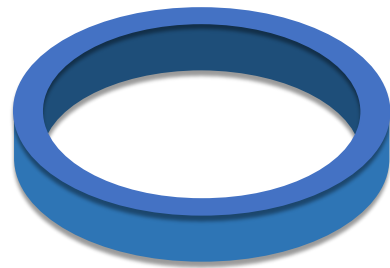
- Encapsulate variables within a local scope
- Pass parameters to IIFE to make it versatile
- Prevent the use of undeclared variables
- Avoid overusing IIFEs



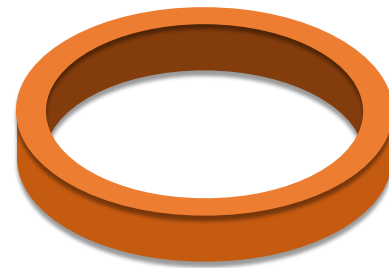
# IIFEs: Design Patterns

Immediately Invoked Function Expressions (IIFEs) are often used in various design patterns in JavaScript to achieve specific goals.

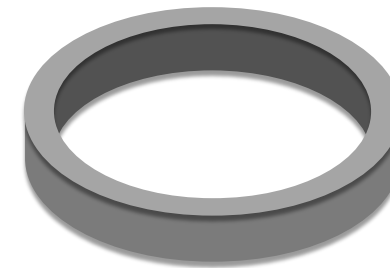
A few examples of design patterns are:



Module pattern



Singleton pattern



Augmentation pattern

# IIFEs: Design Patterns

## Module pattern

It uses IIFE to create private and public encapsulation which helps in organizing code and avoids polluting the global namespace.

## Singleton pattern

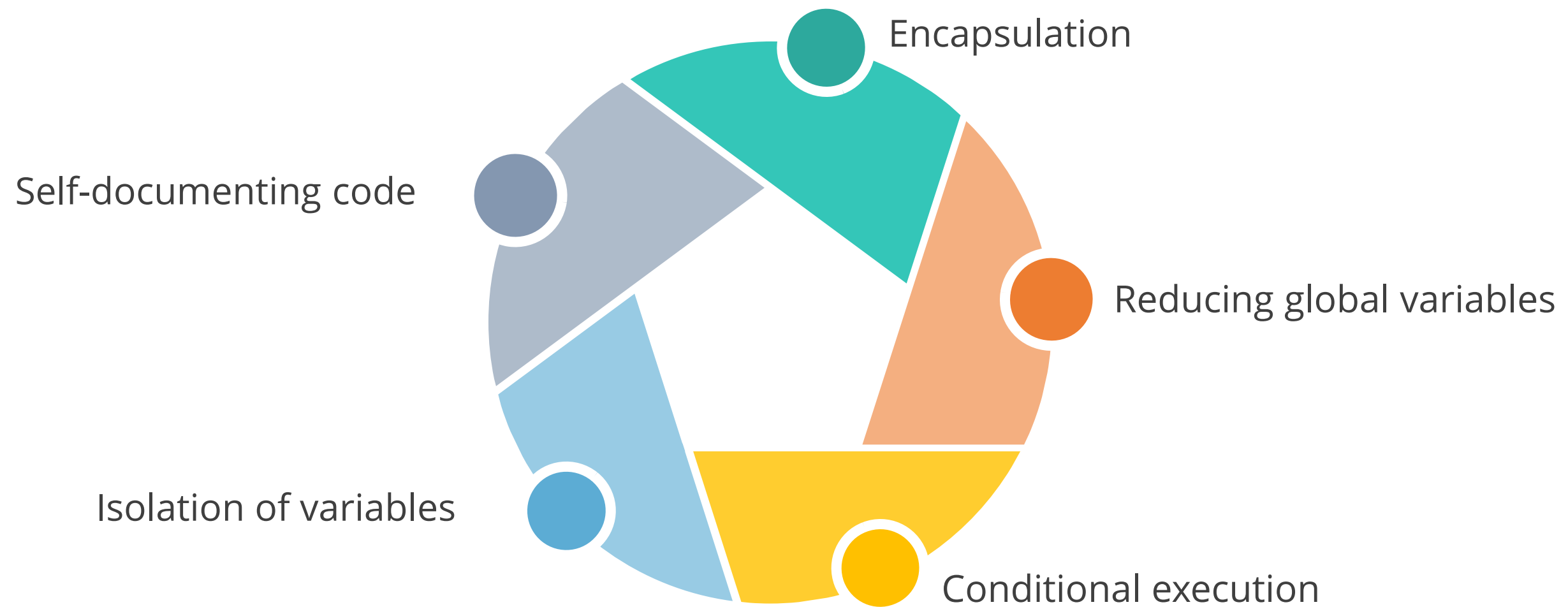
IIFE is commonly used in creating Singleton patterns, where a single instance of an object is shared across the application.

## Augmentation pattern

This pattern uses IIFE to extend an existing object with additional properties or methods without modifying its source code.

# Improving Code Readability with IIFEs

IIFEs can improve code readability in several ways. Here are some aspects where IIFE can enhance code clarity:



# Assisted Practice



## Working with IIFEs and Functions

Duration: 15 Min.

### Problem Statement:

You have been given a task to implement IIFEs and functions in JavaScript.

# Assisted Practice: Guidelines



Steps to be followed:

1. Write a JS program
2. Execute and verify the working of IIFEs and functions



# Maps

# Maps: Overview

A map is a collection of elements in which each element is stored in a key-value pair.

## Example:

```
const map_fun = new Map();  
  
map_fun.set('ab', 2);  
  
console.log(map_fun.get('ab'));
```

- A map object iterates its elements in an insertion order that returns an array of **[key, value]** for each iteration.
- A map can hold objects and primitive values as either keys or values.

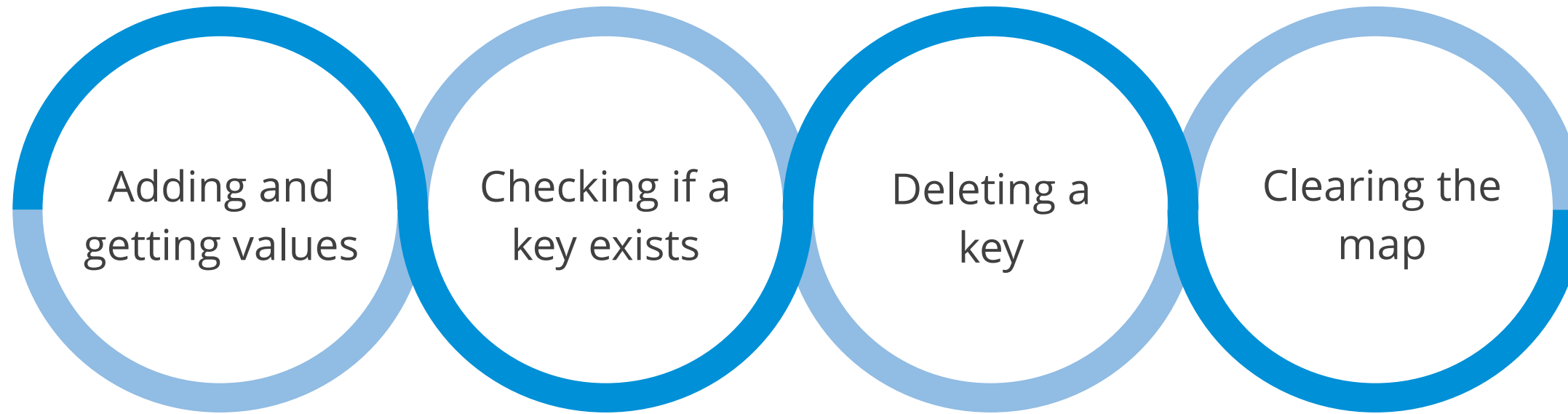
# Maps: Methods

Methods	Description
Map.prototype.set()	Adds and updates key and value to a map object
Map.prototype.has()	Returns a Boolean value depending on the element with the specified key is present
Map.prototype.get()	Returns the element from a map object
Map.prototype.delete()	Deletes both the key and the value from the map object
Map.prototype.clear()	Removes all elements from the map object
Map.prototype.entries()	Returns an iterator object that contains a key-value pair for each element present in the map object in intersection order



# Maps: Basic Operations

Maps perform some basic operations, including:



# Maps: Advanced Techniques

Below are a few advanced maps techniques in JS:

01

Basic usage of the **map** method

02

Arrow functions with **map**

03

Chaining **map** with other array methods

# Maps: Advanced Techniques

Below are a few advanced maps techniques in JS:

04

Mapping over object properties

05

Mapping with index

06

Mapping over nested arrays



# Classes

# Classes: Overview

JavaScript classes are different from Java classes. Classes are special functions, just like function expressions and function declarations.

## Example:

```
class Rectangle {  
  constructor(height, width) {  
  
    this.height = height;  
    this.width = width;  
  
  }  
}
```

- Classes do not allow property value assignments like constructor functions or object literals.
- Class syntax has two components: class expressions and class declarations.

# Classes: Features

## Subclassing

Users can implement inheritance in JavaScript with it.

## Getter and Setter

These are used to get and set the property value.

## Constructor

It is a special function in the class declaration and defines a function that represents that class.

## Static methods

These are functions of classes and not of their prototypes.



# Class Methods

Class methods, also known as static methods, are methods that are associated with the class rather than its instances. They are defined using the static keyword.

## Example:

```
class MathOperations {  
    static add(x, y) {  
        return x + y;  
    }  
  
    static subtract(x, y) {  
        return x - y;  
    }  
}  
console.log(MathOperations.add(5, 3));  
// Outputs: 8  
console.log(MathOperations.subtract(10, 4));  
// Outputs: 6
```

- In this example, add and subtract are class methods.
- They do not operate on specific instances of the MathOperations class and are called directly on the class.

# Assisted Practice



## Implementing Maps and Classes

Duration: 15 Min.

### Problem Statement:

You have been given a task to implement maps and classes in JS.

ASSISTED PRACTICE



# Assisted Practice: Guidelines



Steps to be followed:

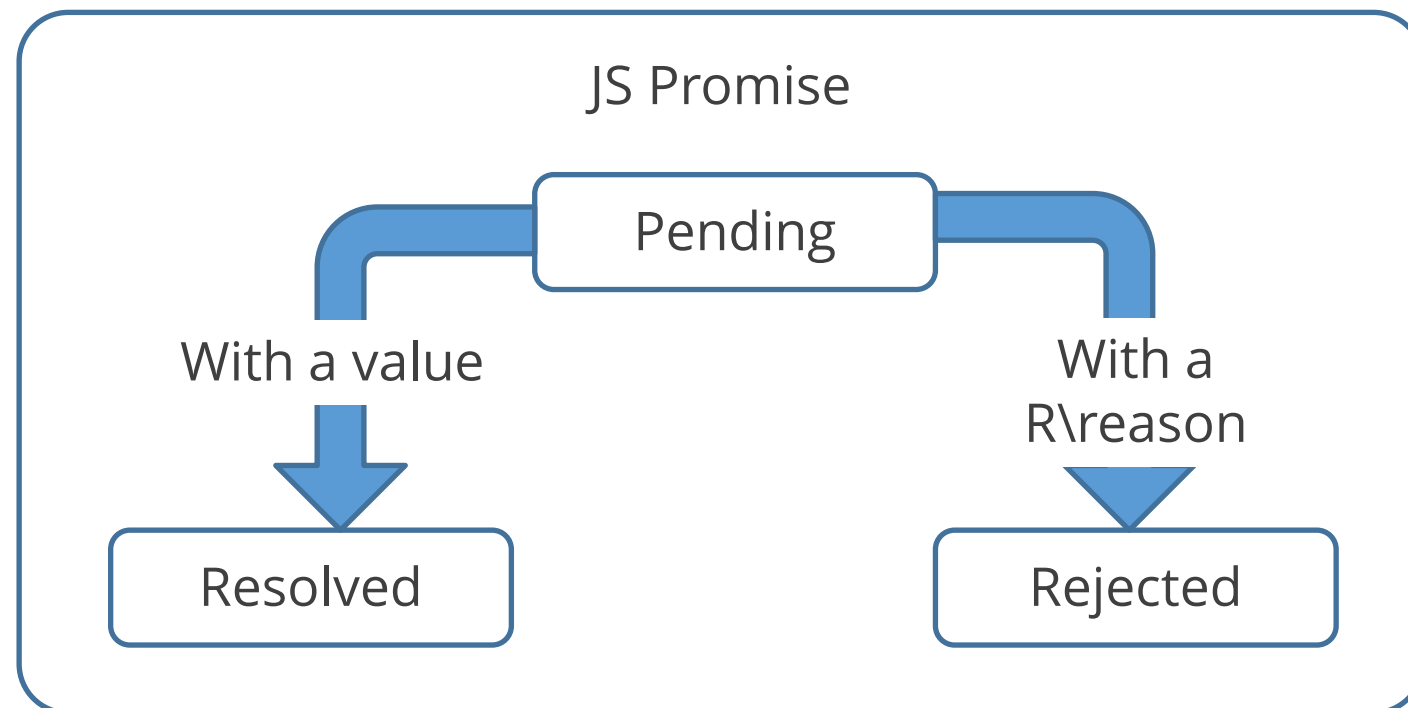
1. Write a JS program
2. Execute and verify the working of maps and classes



# Promises

# Promises: Overview

A promise is an object that represents the completion of an event in an asynchronous operation and its result.



A promise:

- Improves code readability
- Handles asynchronous operations
- Handles errors

# Promises: Example

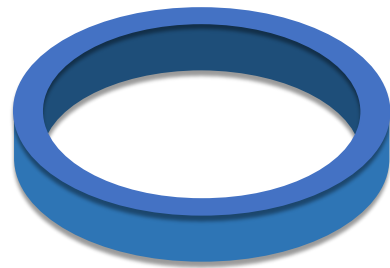
## Example:

```
var promise = new  
  
Promise(function(resolve, reject) {  
  
  Resolve('JavaScript Promises');  });  
  
promise.then(function(successMessage) {  
  
  console.log(successMessage);  
  
}, function(errorMessage) {  
  
  console.log(errorMessage); })
```

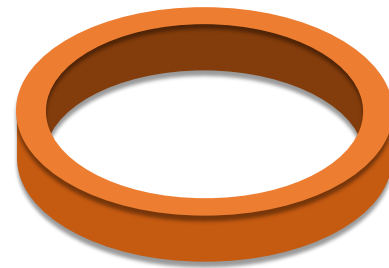
# Promises: States

These states represent the current stage of the asynchronous operation that the promise is handling.

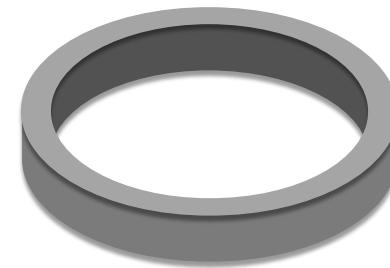
Promises have three states:



Pending



Fulfilled



Rejected

# Promises: States

## Pending

This means that the asynchronous operation represented by the promise is ongoing, and the outcome (either success or failure) has not been determined.

## Fulfilled

This means that the operation produced a result, and the promise now holds that result.

## Rejected

If an error occurs during the asynchronous operation, the promise transitions to the rejected state. In this state, the promise holds the reason for the failure.

# Promise Chaining

It is a technique that allows you to perform a sequence of asynchronous operations one after another.



# Promise Chaining

Below is the example of promise chaining:

```
const firstAsyncOperation = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('First operation completed');
      resolve('Result of the first operation');
    }, 1000);
  });
};

const secondAsyncOperation = (result) => {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Second operation completed');
      resolve(`Result of the second operation
using ${result}`);
    }, 1000);
  });
};
```

```
firstAsyncOperation()
  .then((result) => {
    // Result of the first operation
    console.log(result);
    // Return a new Promise for the next
    operation
    return secondAsyncOperation(result);
  })
  .then((finalResult) => {
    // Result of the second operation
    console.log(finalResult);
  })
  .catch((error) => {
    console.error('Error:', error);
  });
```



# Assisted Practice



## Working with Promises

Duration: 15 Min.

### Problem Statement:

You have been assigned a task to implement promises.

ASSISTED PRACTICE

# Assisted Practice: Guidelines



Steps to be followed:

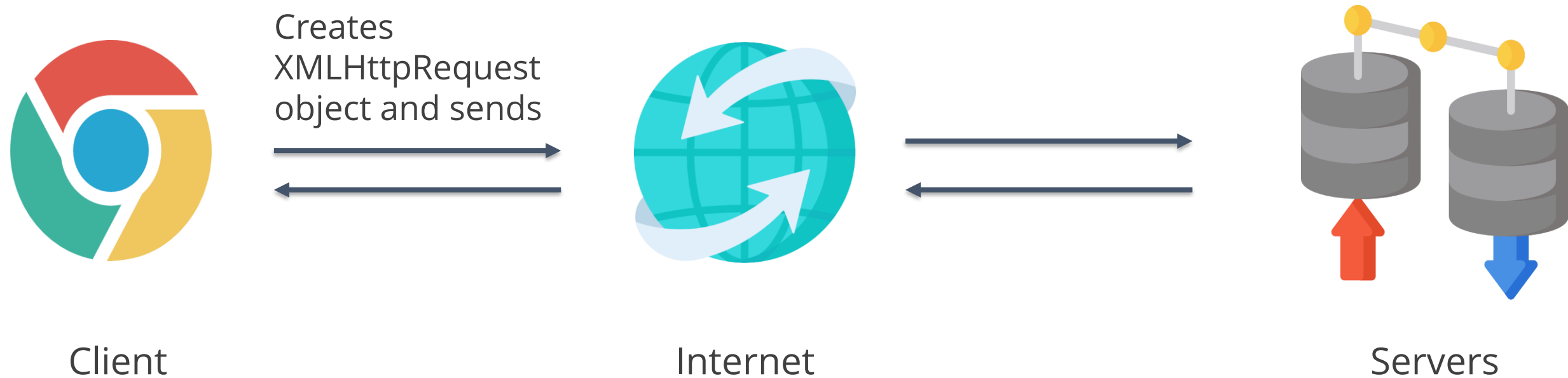
1. Write a JS program
2. Execute and verify the working of promises



## **Asynchronous JavaScript and XML (AJAX) Calls**

# AJAX

AJAX stands for **Asynchronous JavaScript and XML**. It helps in developing better, faster, and more interactive web applications with XML, HTML, CSS, and JavaScript.



# AJAX

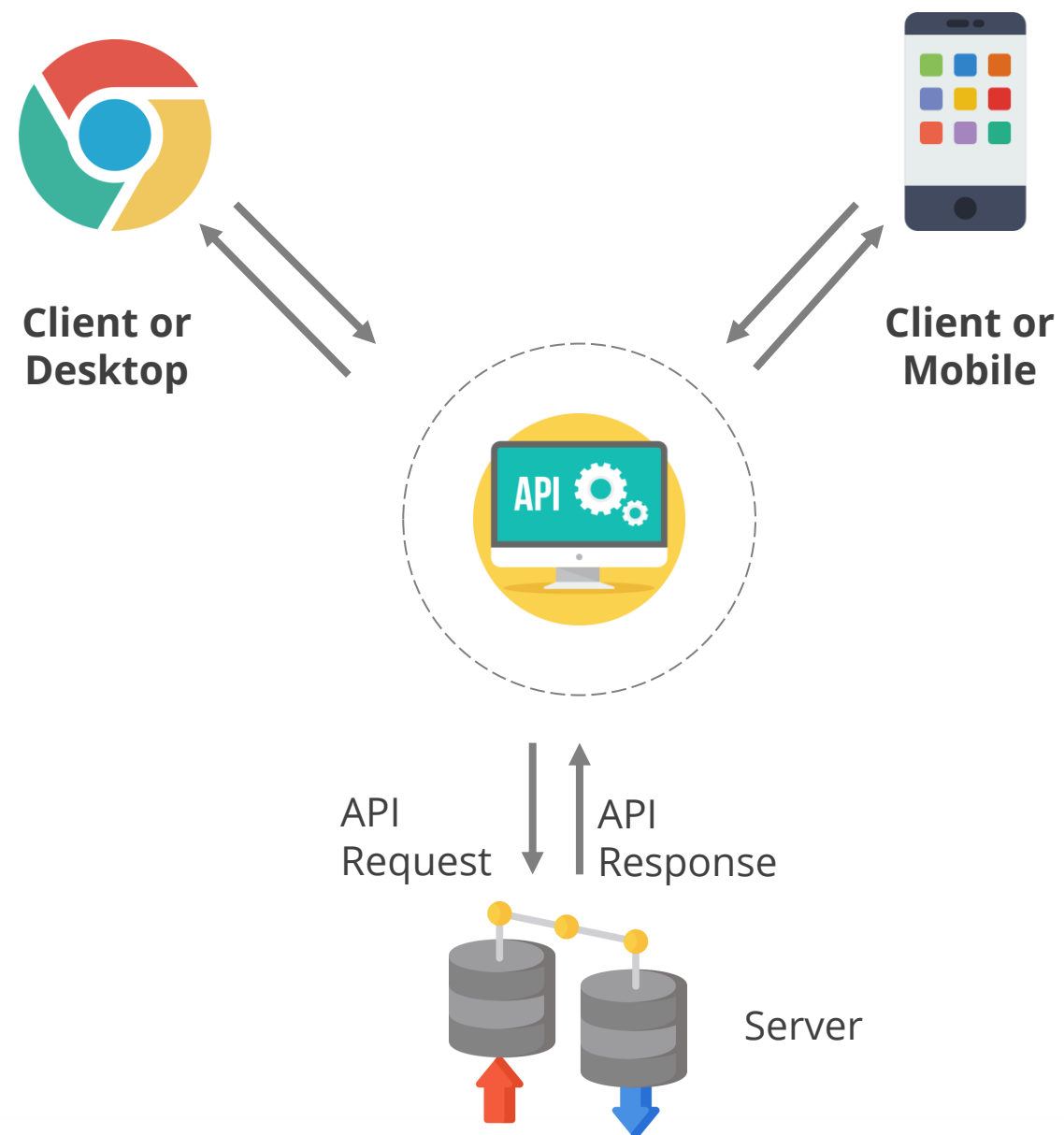
AJAX is based on the following standards:



- Browser-based presentation
- Data fetched from servers and stored in XML format
- Data fetched using **XMLHttpRequest** objects

# APIs

API stands for Application Programming Interface. APIs are techniques that allow two software components to communicate with each other.



## Third-party APIs:

- Google maps
- YouTube videos
- Weather data
- Movies data

# AJAX with Fetch API

The Fetch API provides an interface to fetch resources across networks. It helps in defining HTTP-related concepts such as extensions to HTTP.

## Example:

```
fetch('<URL>', {method: 'GET'})  
  
  .then(response=>response.json())  
  
  .then(json=>console.log(json))  
  
  .catch(error=>console.log('error:', error));
```

It is widely used by progressive web app service workers.

# Fetch API: Features

## Cookie less by default

The application's authentication could fail as all implementations of Fetch API may not send cookies.

## Unaccepted Errors

Rejections only occur if a request cannot be completed; therefore, error trapping is complicated to implement.

## Unsupported Timeouts

Browsers will continue to run until they are stopped.

## Fetch Aborting

Fetch can be aborted by calling **`controller.abort();`**.



# AJAX with Promise

There are two functions, **welcome()** and **userProfile()**.  
The **userProfile()** function will not work as it depends on the **welcome()** function.

## Ajax without Promise

### Example:

```
function welcome() {  
  $.ajax({  
    url:<some URL>,  
    type:'POST',  
    data:{ //some data },  
    success: userProfile()  
  })  
}
```

## Ajax with Promise

### Example:

```
function welcome() {  
  return new Promise((resolve,  
    reject)=>{  
    $.ajax({  
      url:<some URL>,  
      type:'POST',  
      data:{ //some data },  
      success: userProfile() }) }) }  
}
```



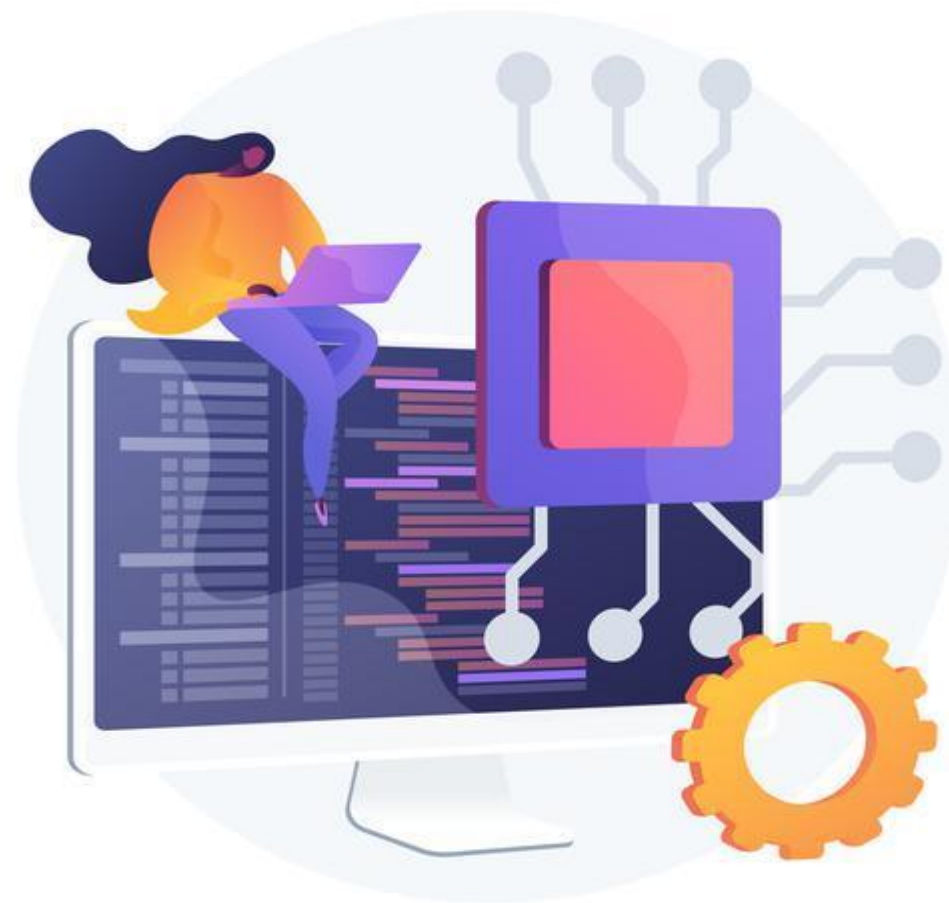
## **Working with Advanced AJAX Concepts**

# Advanced AJAX Concepts

AJAX concepts in JavaScript involve using more sophisticated techniques to handle asynchronous requests, interact with servers, and manage data.

Here are two types of advanced AJAX concepts:

Authentication



Authorization

# Authentication in AJAX Requests

## Token-Based Authentication

Many web applications use this authentication, where a token is obtained during the login process and subsequently included in the headers of AJAX requests.

## Authentication Tokens

When making an AJAX request, include the authentication token in the request headers.

## Token Expiry and Refresh

If a token expires, the user may need to refresh it using a refresh token (if available) or by reauthenticating.

# Authorization in AJAX Requests

## Role-Based Access Control

When making AJAX requests, ensure that the user making the request has the necessary permissions based on their role.

## Authorization Information

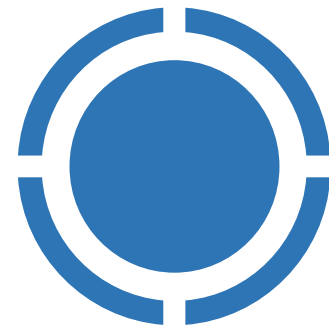
This includes user roles or specific permissions needed for the requested resource or action.

## Unauthorized Responses

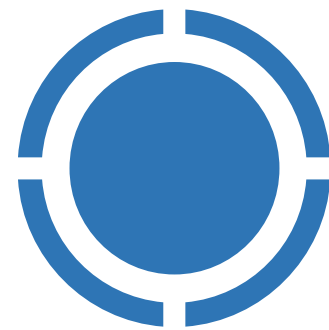
This involves redirecting the user to a login page or displaying an error message.

# Uploading Files Using AJAX

Below are the different approaches to upload files using AJAX:



Using FormData



Using FileReader

# Downloading Files Using AJAX

Below are the different approaches to download files using AJAX:



# Assisted Practice



## Implementing AJAX Calls

Duration: 15 Min.

### Problem Statement:

You have been assigned a task to implement AJAX calls.

ASSISTED PRACTICE



# Assisted Practice: Guidelines



Steps to be followed:

1. Write a JS program
2. Execute and verify the working of AJAX calls



# **Webpack in JavaScript**

# Webpack

It is a static module bundler for modern JavaScript applications that helps in mapping every module of the project requirements by building a dependency graph.

Webpack module dependencies can be implemented in any one of the following ways:



- An ES6 **import** statement
- A commonJS **require()** statement
- An **@import** statement inside of a CSS/SASS file
- An image URL in a stylesheet or an HTML file

# Webpack: Features

Some of the features of Webpack are:

Module bundling





Asset optimization

Code splitting







# Setting Up Webpack in JS

Webpack setup in a JavaScript project involves several steps to configure and integrate Webpack into your development workflow, which are:

-  Initialize your project
-  Install webpack CLI
-  Create a webpack configuration file
-  Create a simple JS file

# Setting Up Webpack in JS

Webpack setup in a JavaScript project involves several steps to configure and integrate Webpack into your development workflow, which are:

-  Add a build script
-  Run the build script
-  Create an HTML file
-  Open the HTML file in a browser



# Modern JavaScript

# Modern JavaScript: Overview

It is a safe, secure, and reliable programming language. It can execute in browsers as well as on servers. The browsers have an embedded engine to execute the scripts.



## The workflow of the Engine:

- The engine reads the script.
- It converts the JavaScript to machine code.
- The machine code is then executed.

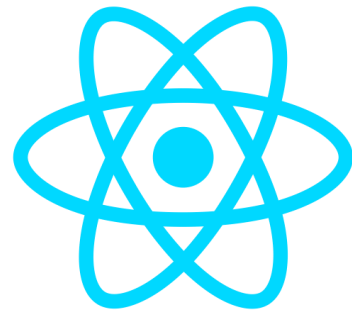


# Modern JavaScript: Overview

Modern JavaScript supports the below frameworks:



Angular



React



Vue.js



Node.js



Next.js



Express.js



BACKBONE.JS

Backbone.js



Meteor.js

# Modern JavaScript: Compatibility

New languages transpired in JavaScript before execution:

- **CoffeeScript** is a language that compiles JavaScript.
- **Flow** is a static type checker for JavaScript.
- **TypeScript** is a strongly typed programming language that builds on JavaScript.
- **Dart** is a classical, object-oriented language where everything is an object.



CoffeeScript



Flow by  
Facebook



TypeScript by  
Microsoft



Dart

Dart by  
Google

# Assisted Practice



## Working with Webpack and Modern JavaScript

Duration: 15 Min.

### Problem Statement:

You have been given a task to work with Webpack and modern JS.

ASSISTED PRACTICE

# Assisted Practice: Guidelines



Steps to be followed:

1. Write a JS program
2. Execute and verify the working of Webpack



# **Introduction to Babel**

# Babel: Overview

Babel is an open-source JavaScript compiler used to convert ES6+ code into a backwards-compatible version of JavaScript.



## Popular uses of Babel:

- Transforming syntax
- Adding polyfill features
- Transforming source code

# Babel: Overview

The below example demonstrates the arrow function:

## Arrow function as input

### Example:

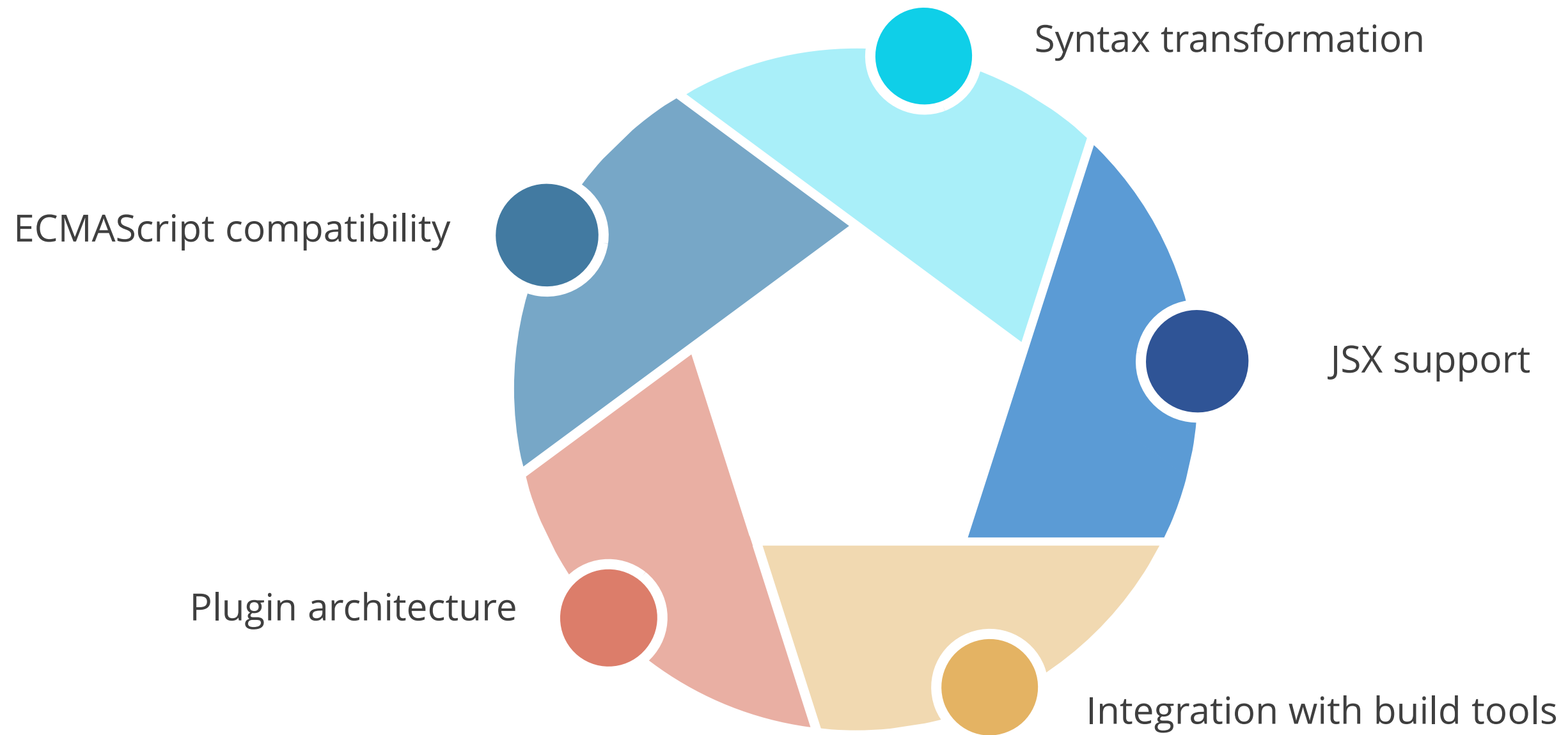
```
[10, 30, 21].map((n) => n + 1);
```

## Arrow function as output

### Example:

```
[10, 30, 21].map(function(n) {  
  return n + 1;  
});
```

# Babel: Features





# Babel: Benefits

Babel offers several benefits, including:



Cross-browser compatibility



Future proofing code



Improved developer productivity



Community support and ecosystem



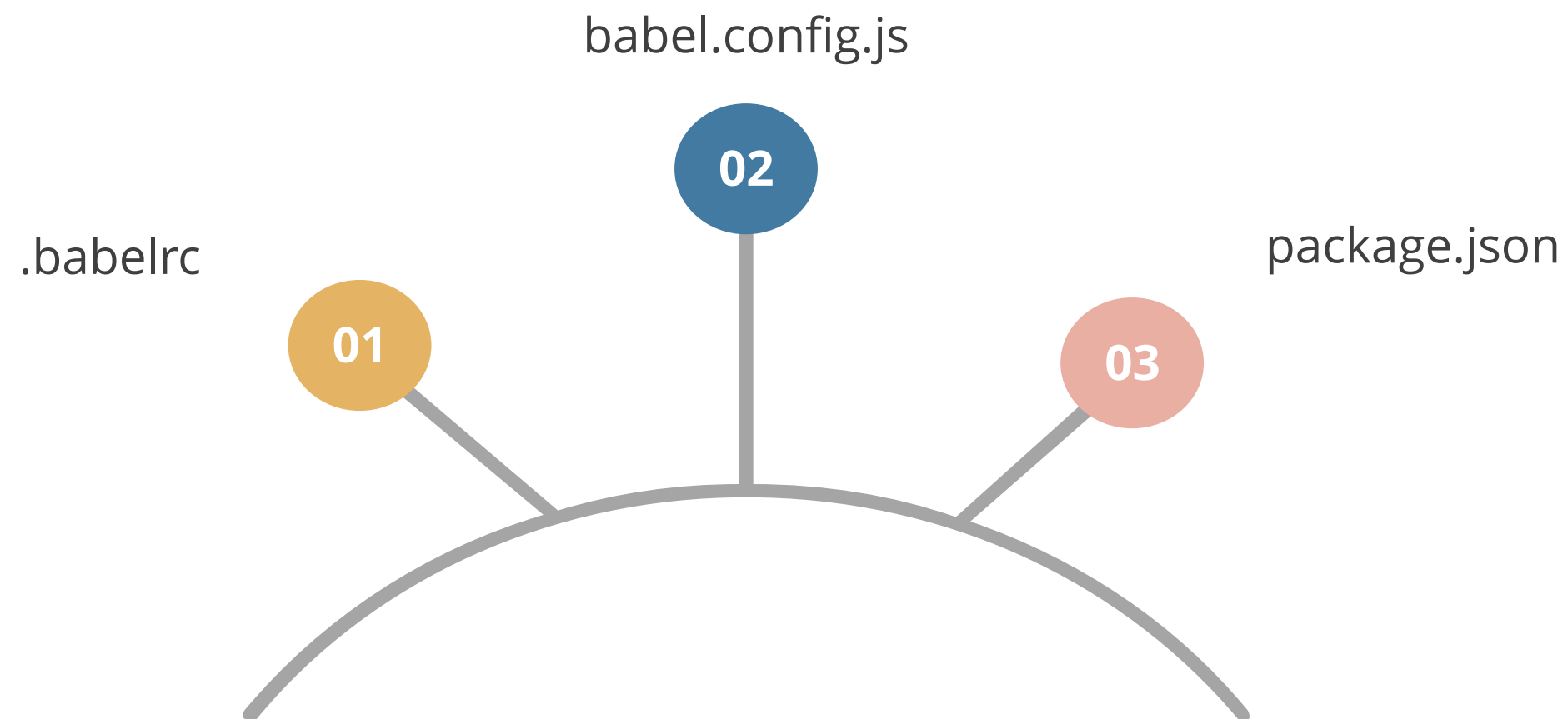
Adaptability to project needs



## Configuring Babel

# Babel Configuration Files

These files are used to specify how Babel should transform your JavaScript code. Below are the commonly used files:



# Security Best Practices for Babel Configuration

Users can follow these best practices to optimize and secure their code while configuring Babel for their JavaScript project:

1

Update dependencies regularly

2

Limit plugin usage

3

Avoid untrusted plugins

4

Restrict code execution

5

Review and audit plugins

6

Minimize global installation

7

Consider using a lockfile

8

Review external configurations

9

Secure development environment

10

Enable source maps in development

# Assisted Practice



## Working with Babel

**Duration: 15 Min.**

### Problem Statement:

You have been assigned a task to compile your JS program in Babel.

ASSISTED PRACTICE

# Assisted Practice: Guidelines

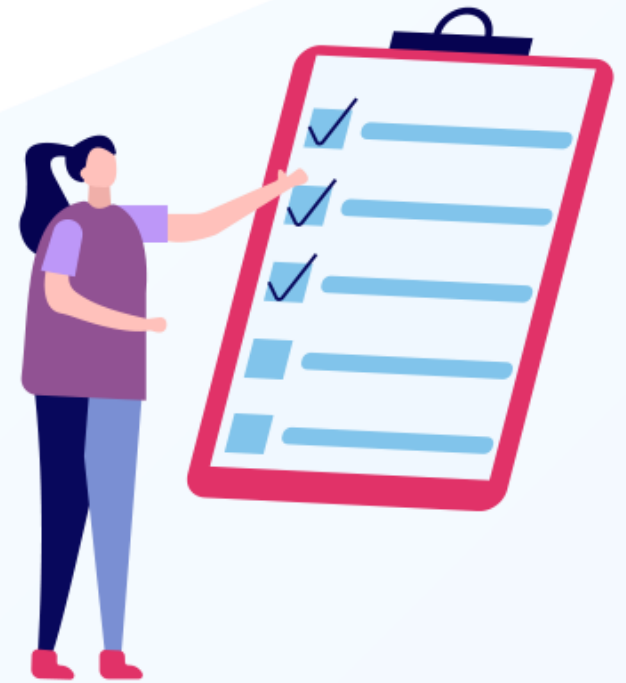


Steps to be followed:

1. Write a JS program for Babel
2. Execute and verify the implementation of Babel

# Key Takeaways

- Advanced JavaScript is an in-depth and comprehensive understanding of the JavaScript programming language that goes beyond the fundamentals.
- A function callback is to be executed after another function has finished executing, and it is used while handling an asynchronous operation.
- A promise is an object that represents the completion of an event in an asynchronous operation and its result.
- The Fetch API provides an interface to fetch resources across networks. It helps in defining HTTP-related concepts such as extensions to HTTP.
- Webpack is a static module bundler for modern JavaScript applications that helps in mapping every module of the project requirements by building a dependency graph.





**Thank You**