**Promise**

**1. What is a Promise?**

A **Promise** in is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

* **Example of a real-world analogy**:  
  Imagine you order a book online.
  + The **Promise** is your order receipt.
  + When the book arrives, the promise is **fulfilled** (resolved).
  + If the book is lost, the promise is **rejected**.

**2. Syntax**

You create a new Promise using the constructor:

let promise = new Promise(function(resolve, reject) {

// executor function

});

**Executor Function**

The **executor function** is a function passed to the Promise constructor. It receives two arguments:

* resolve(value) → called when the task is successful.
* reject(error) → called when there’s an error.

**3. States of a Promise**

A Promise can have one of the following states:

* **Pending**: The initial state. The operation is ongoing.
* **Fulfilled**: The operation completed successfully.
* **Rejected**: The operation failed.

**4. Creating a Simple Promise**

Here’s how you create and use a Promise:

let promise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve("Task completed successfully!");

} else {

reject("Task failed!");

}

});

// Consuming the Promise

promise.then(

result => console.log(result), // Handles the resolved value

error => console.error(error) // Handles the rejected value

);

**5. Example with Asynchronous Code**

Let’s simulate a delayed task using setTimeout:

let promise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded successfully!");

}, 2000);

});

// Attach handlers

promise.then(

result => console.log(result), // "Data loaded successfully!"

error => console.error(error) // (won't run in this case)

);

**6. Chaining Promises**

You can chain multiple .then() calls to handle sequential asynchronous tasks:

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve(10), 1000); // Resolves with 10

});

promise

.then(result => {

console.log(result); // 10

return result \* 2; // Pass 20 to the next `.then`

})

.then(result => {

console.log(result); // 20

return result \* 2; // Pass 40 to the next `.then`

})

.then(result => {

console.log(result); // 40

});

**7. Error Handling with .catch()**

Use .catch() to handle errors in a Promise chain:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Something went wrong!")), 2000);

});

promise

.then(result => console.log(result)) // Won't run

.catch(error => console.error(error.message)) // "Something went wrong!"

.finally(() => console.log("Cleanup actions")); // Always runs

**8. Using finally**

The .finally() method allows you to execute cleanup code regardless of the promise outcome.

let promise = new Promise((resolve, reject) => {

let success = false;

if (success) {

resolve("Task succeeded!");

} else {

reject("Task failed!");

}

});

promise

.then(result => console.log(result)) // Task succeeded!

.catch(error => console.error(error)) // Task failed!

.finally(() => console.log("Finished!")); // Always runs

**9. Practical Example: Loading a Script**

Here’s a real-world example where you use a Promise to load a script:

function loadScript(src) {

return new Promise((resolve, reject) => {

let script = document.createElement("script");

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Failed to load script: ${src}`));

document.head.append(script);

});

}

// Usage

loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.21/lodash.min.js")

.then(script => console.log(`${script.src} loaded successfully!`))

.catch(error => console.error(error.message))

.finally(() => console.log("Loading script completed."));

**10. Promise.all and Promise.race**

**Promise.all**

Waits for all promises to resolve. If any promise rejects, it immediately rejects.

let promise1 = Promise.resolve(10);

let promise2 = Promise.resolve(20);

let promise3 = Promise.resolve(30);

Promise.all([promise1, promise2, promise3]).then(results => {

console.log(results); // [10, 20, 30]

});

**Promise.race**

Returns the result of the first settled promise.

let promise1 = new Promise(resolve => setTimeout(() => resolve("First!"), 1000));

let promise2 = new Promise(resolve => setTimeout(() => resolve("Second!"), 2000));

Promise.race([promise1, promise2]).then(result => {

console.log(result); // "First!"

});

**11. Advantages of Promises**

* **Avoids Callback Hell**: Code becomes easier to read and maintain.
* **Error Handling**: Centralized error management using .catch().
* **Composability**: Promises can be chained and combined.

**Callbacks**

**What is a Callback?**

A **callback** is a function that is passed as an argument to another function and is executed after some operation is completed. Callbacks are widely used in , especially for asynchronous programming, where tasks run in the background and you want to execute code after the task completes.

**Why Use Callbacks?**

Callbacks ensure that a function is not executed before a task is completed but will be executed right after the task is finished. For example:

* Waiting for data to load from an API.
* Waiting for a file to be read.
* Waiting for a script to load.

**Step-by-Step Explanation of Callbacks**

**1. Basic Callback Example**

A simple example where one function is called after another function completes:

function greet(name, callback) {

console.log(`Hello, ${name}`);

callback();

}

function sayGoodbye() {

console.log("Goodbye!");

}

// Passing sayGoodbye as a callback

greet("Alice", sayGoodbye);

**Output:**

Hello, Alice

Goodbye!

**Explanation:**

1. The greet function logs "Hello, Alice".
2. After greet finishes, it calls the callback() function, which logs "Goodbye!".

**2. Asynchronous Callback Example**

Callbacks are particularly useful for asynchronous tasks, such as loading a script or fetching data.

function loadScript(src, callback) {

let script = document.createElement("script");

script.src = src;

// Call the callback once the script is loaded

script.onload = () => callback(null, script);

// Handle errors

script.onerror = () => callback(new Error(`Failed to load script: ${src}`));

document.head.append(script);

}

// Usage

loadScript("https://example.com/script.js", (error, script) => {

if (error) {

console.error(error);

} else {

console.log(`Script loaded: ${script.src}`);

}

});

**Explanation:**

1. loadScript dynamically creates a <script> tag to load a file.
2. When the script loads successfully, onload triggers the callback with null for the error and the loaded script as the second argument.
3. If an error occurs, onerror triggers the callback with an error message.

**3. Nested Callbacks**

When one asynchronous task depends on another, you can nest callbacks.

loadScript("script1.js", (error, script) => {

if (error) {

console.error(error);

} else {

console.log(`Loaded: ${script.src}`);

loadScript("script2.js", (error, script) => {

if (error) {

console.error(error);

} else {

console.log(`Loaded: ${script.src}`);

loadScript("script3.js", (error, script) => {

if (error) {

console.error(error);

} else {

console.log(`Loaded: ${script.src}`);

}

});

}

});

}

});

**Explanation:**

* script1.js is loaded first. When it finishes, script2.js starts loading.
* After script2.js loads, script3.js starts loading.
* Each loadScript depends on the previous one.

**Problem**: This creates **callback hell** or the **pyramid of doom**, which is hard to read and maintain.

**Error-First Callbacks**

The convention for handling errors in callbacks is called the **error-first callback** style. The first argument of the callback is reserved for the error object, and subsequent arguments are for the result.

**Example**

function fetchData(url, callback) {

setTimeout(() => {

if (!url) {

callback(new Error("URL is required"), null);

} else {

callback(null, { data: "Sample Data" });

}

}, 1000);

}

// Usage

fetchData("", (error, data) => {

if (error) {

console.error("Error:", error.message);

} else {

console.log("Data:", data);

}

});

**Output:**

swift

Error: URL is required

**Avoiding Callback Hell**

1. **Breaking Down into Functions**

function step1(callback) {

console.log("Step 1 complete");

callback();

}

function step2(callback) {

console.log("Step 2 complete");

callback();

}

function step3() {

console.log("Step 3 complete");

}

step1(() => {

step2(() => {

step3();

});

});

1. **Using Promises** Promises are a cleaner way to handle asynchronous operations and avoid callback hell. They are covered in the next step.

**slice()**

**What is slice()?**

The slice() method is used to extract a portion of an array into a new array without modifying the original array.

**Syntax**

arr.slice([start], [end]);

* **start** *(optional)*: The zero-based index where the slice begins. If omitted, slicing starts from the beginning (0).
* **end** *(optional)*: The zero-based index before which the slicing ends (not inclusive). If omitted, slicing goes to the end of the array.
* Both start and end can be **negative**, counting from the end of the array (-1 is the last element).

**Returns**:  
A **new array** containing the sliced portion.

**Step-by-Step Explanation**

**Example 1: Basic Usage**

let arr = ["a", "b", "c", "d", "e"];

// Slice from index 1 to 3 (end not included)

let result = arr.slice(1, 3);

console.log(result); // Output: ["b", "c"]

console.log(arr); // Original array remains unchanged: ["a", "b", "c", "d", "e"]

**Steps:**

1. **Input Array**: ["a", "b", "c", "d", "e"].
2. **start = 1**: Start slicing from index 1 ("b").
3. **end = 3**: Stop slicing before index 3 ("d").
4. **Result**: Extracted subarray is ["b", "c"].

**Example 2: Slicing with Negative Indices**

let arr = ["a", "b", "c", "d", "e"];

// Slice the last two elements

let result = arr.slice(-2);

console.log(result); // Output: ["d", "e"]

**Steps:**

1. **Negative Indices**: -2 starts slicing from the second-to-last element.
2. **No End**: Without end, slice goes to the end of the array.
3. **Result**: Extracted subarray is ["d", "e"].

**Example 3: Copying the Entire Array**

let arr = [1, 2, 3, 4, 5];

// Copy the entire array

let copy = arr.slice();

console.log(copy); // Output: [1, 2, 3, 4, 5]

console.log(copy === arr); // Output: false (different references)

**Explanation:**

* Calling slice() without arguments creates a shallow copy of the array.
* The original array remains unchanged.

**Example 4: Combining with Negative and Positive Indices**

let arr = ["apple", "banana", "cherry", "date", "elderberry"];

// Slice from index 1 to the second-to-last element

let result = arr.slice(1, -1);

console.log(result); // Output: ["banana", "cherry", "date"]

**Steps:**

1. **start = 1**: Slicing starts from "banana".
2. **end = -1**: Stops before the last element ("elderberry").
3. **Result**: Extracted subarray is ["banana", "cherry", "date"].

**Advanced Examples**

**Example 5: Working with Strings and Arrays**

The slice() method can also work with strings in a similar way:

let str = "hello";

// Slice a part of the string

let result = str.slice(1, 4);

console.log(result); // Output: "ell"

**Example 6: Combining slice() with Other Methods**

You can chain slice() with other methods to manipulate arrays:

let numbers = [1, 2, 3, 4, 5, 6];

// Get the middle three elements, then reverse them

let result = numbers.slice(2, 5).reverse();

console.log(result); // Output: [5, 4, 3]

**Key Points**

1. **Non-Mutating**: slice() does not modify the original array.
2. **Returns a New Array**: Always creates a new array containing the sliced elements.
3. **Handles Negative Indices**: Negative values count from the end of the array.
4. **Useful for Copying Arrays**: slice() without arguments (arr.slice()) creates a shallow copy.

**Comparison with Similar Methods**

|  |  |  |
| --- | --- | --- |
| **Feature** | **slice()** | **splice()** |
| **Purpose** | Extracts part of an array | Modifies the original array |
| **Modifies Original** | No | Yes |
| **Returns** | New array with sliced elements | Removed elements |
| **Use Case** | When you want a copy or subarray | When modifying the array |

**filter()**

The filter() method in is used to create a new array containing all elements of the original array that satisfy a given condition. Let's break it down step by step with examples.

**Syntax**

array.filter(function(element, index, array) {

// Return true to keep the element, false to exclude it

});

**Parameters:**

1. **element**: The current element being processed in the array.
2. **index** *(optional)*: The index of the current element.
3. **array** *(optional)*: The original array that filter was called on.

**Returns:**

* A new array containing all elements that pass the test (i.e., the function returns true).
* If no elements pass the test, it returns an empty array.

**Step-by-Step Explanation**

**Example:**

let numbers = [1, 2, 3, 4, 5, 6];

// Filter out numbers greater than 3

let filteredNumbers = numbers.filter(number => number > 3);

console.log(filteredNumbers); // Output: [4, 5, 6]

**Steps:**

1. **Input Array**: [1, 2, 3, 4, 5, 6].
2. **Condition**: number > 3.
3. **Iteration**:
   * Check 1 > 3: False (exclude it).
   * Check 2 > 3: False (exclude it).
   * Check 3 > 3: False (exclude it).
   * Check 4 > 3: True (include it).
   * Check 5 > 3: True (include it).
   * Check 6 > 3: True (include it).
4. **Output Array**: [4, 5, 6].

**Using All Parameters**

let numbers = [1, 2, 3, 4, 5, 6];

// Filter out even numbers and log their index

let evenNumbers = numbers.filter((number, index, array) => {

console.log(`Index: ${index}, Value: ${number}, Array: ${array}`);

return number % 2 === 0;

});

console.log(evenNumbers); // Output: [2, 4, 6]

**Breakdown:**

* The filter callback receives the current number, its index, and the entire array.
* It logs these details for each iteration.
* The condition number % 2 === 0 checks if the number is even.

**Filtering an Array of Objects**

let users = [

{ id: 1, name: "John", age: 25 },

{ id: 2, name: "Pete", age: 30 },

{ id: 3, name: "Mary", age: 22 },

{ id: 4, name: "Jane", age: 30 },

];

// Filter users with age 30

let filteredUsers = users.filter(user => user.age === 30);

console.log(filteredUsers);

/\* Output:

[

{ id: 2, name: "Pete", age: 30 },

{ id: 4, name: "Jane", age: 30 }

]

\*/

**Steps:**

1. **Input Array**: An array of user objects.
2. **Condition**: user.age === 30.
3. **Iteration**:
   * For each object in the array, check if age is 30.
   * If true, include the object in the new array.
4. **Output Array**: Contains only users with age: 30.

**Key Points**

1. **Non-Mutating**: filter() does not change the original array; it returns a new array.
2. **Empty Array**: If no elements satisfy the condition, it returns an empty array.
3. **Chainable**: Can be used with other array methods like map() or reduce().

**Practical Use Cases**

1. **Filtering Numbers**:

let numbers = [10, 15, 20, 25, 30];

let multiplesOfTen = numbers.filter(n => n % 10 === 0);

console.log(multiplesOfTen); // Output: [10, 20, 30]

1. **Filtering Strings**:

let fruits = ["apple", "banana", "cherry", "apricot", "blueberry"];

let fruitsWithA = fruits.filter(fruit => fruit.startsWith("a"));

console.log(fruitsWithA); // Output: ["apple", "apricot"]

1. **Removing Undefined/Null Values**:

let mixedArray = [1, undefined, 2, null, 3, 4];

let cleanedArray = mixedArray.filter(value => value != null);

console.log(cleanedArray); // Output: [1, 2, 3, 4]

1. **Filtering Objects**:

let products = [

{ id: 1, name: "Laptop", price: 1000 },

{ id: 2, name: "Mouse", price: 20 },

{ id: 3, name: "Keyboard", price: 50 },

];

let expensiveProducts = products.filter(product => product.price > 100);

console.log(expensiveProducts); // Output: [{ id: 1, name: "Laptop", price: 1000 }]

**Map Method:-**

Here’s an example that demonstrates all the operations for Map in in a simple and easy-to-understand way:

// Step 1: Create a new map

const map = new Map();

console.log("Map created:", map);

// Step 2: Add key-value pairs using set()

map.set("name", "Alice");

map.set("age", 25);

map.set("city", "New York");

console.log("Map after adding key-value pairs:", map);

// Step 3: Get a value using get()

console.log("Get 'name':", map.get("name")); // Output: Alice

console.log("Get 'country' (non-existing key):", map.get("country")); // Output: undefined

// Step 4: Check if a key exists using has()

console.log("Has 'age':", map.has("age")); // Output: true

console.log("Has 'country':", map.has("country")); // Output: false

// Step 5: Delete a key-value pair using delete()

map.delete("city");

console.log("Map after deleting 'city':", map);

// Step 6: Get the size of the map using size

console.log("Map size:", map.size); // Output: 2

// Step 7: Clear the map using clear()

map.clear();

console.log("Map after clearing:", map); // Output: Map(0) {}

**Output:**

plaintext

Map created: Map(0) {}

Map after adding key-value pairs: Map(3) { 'name' => 'Alice', 'age' => 25, 'city' => 'New York' }

Get 'name': Alice

Get 'country' (non-existing key): undefined

Has 'age': true

Has 'country': false

Map after deleting 'city': Map(2) { 'name' => 'Alice', 'age' => 25 }

Map size: 2

Map after clearing: Map(0) {}

**Step-by-Step Explanation:**

1. **Create a Map**: Use new Map() to initialize a new map.
2. **Add Elements**: Use map.set(key, value) to add key-value pairs.
3. **Retrieve Values**: Use map.get(key) to fetch the value for a given key.
4. **Check Key Existence**: Use map.has(key) to check if a key exists in the map.
5. **Remove Elements**: Use map.delete(key) to remove a key-value pair.
6. **Get Map Size**: Access map.size to get the number of key-value pairs.
7. **Clear the Map**: Use map.clear() to remove all elements from the map.

**What is the different between for..of, for..in and foreach in**

**1. for..of**

**Purpose:**  
Iterates over the **values** of an iterable object like arrays, strings, maps, sets, or other objects with an iterator.

**Syntax:**

for (const value of iterable) {

// Code to execute

}

**Use Case:**

* Use when you need to access **values** in arrays or iterable objects.

**Example:**

const array = [10, 20, 30];

for (const value of array) {

console.log(value); // Output: 10, 20, 30

}

**Key Points:**

* Works with iterable objects.
* Does **not** work with plain objects.
* Cannot directly access the index (use array.entries() if needed).

**2. for..in**

**Purpose:**  
Iterates over the **keys** (enumerable properties) of an object or array.

**Syntax:**

for (const key in object) {

// Code to execute

}

**Use Case:**

* Use when you need to iterate over the **keys** of an object or array.

**Example:**

const object = { a: 10, b: 20, c: 30 };

for (const key in object) {

console.log(key); // Output: "a", "b", "c"

}

**Key Points:**

* Can be used with objects, arrays, and other types.
* For arrays, it gives **indices**, not values.
* Iterates over **enumerable** properties, including those inherited from the prototype chain (use hasOwnProperty to filter out inherited properties).

**3. forEach()**

**Purpose:**  
Executes a provided callback function once for each array element.

**Syntax:**

array.forEach((value, index, array) => {

// Code to execute

});

**Use Case:**

* Use when working with arrays and you need both the **value** and the **index**.

**Example:**

const array = [10, 20, 30];

array.forEach((value, index) => {

console.log(`Index: ${index}, Value: ${value}`);

});

// Output:

// Index: 0, Value: 10

// Index: 1, Value: 20

// Index: 2, Value: 30

**Key Points:**

* Works only with arrays.
* Cannot be used with break or continue (to exit early).
* Always iterates over all elements, even if the callback modifies the array.

**Comparison Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **for..of** | **for..in** | **forEach()** |
| Iterates Over | Values | Keys (enumerable properties) | Values (array elements) |
| Works With | Iterables (arrays, strings, maps, sets, etc.) | Objects and arrays | Arrays |
| Access Index | No | Yes (for arrays) | Yes (via callback) |
| Can Break/Continue | Yes | Yes | No |
| Prototype Properties | No | Yes | No |
| Callback Function | No | No | Yes |

**When to Use:**

* **for..of**: When working with iterable objects (arrays, strings, maps, etc.) and you only need values.
* **for..in**: When iterating over object keys or array indices.
* **forEach()**: When working with arrays and you want a clean callback-based iteration.

**Error handling with promises:-**

**1. States of a Promise**

A promise can be in one of these three states:

* **Pending**: Initial state, neither fulfilled nor rejected.
* **Fulfilled**: The operation completed successfully.
* **Rejected**: The operation failed.

**Example:**

let promise = new Promise((resolve, reject) => {

// Perform some asynchronous task

let success = true;

if (success) {

resolve("Operation Successful!"); // Promise is fulfilled

} else {

reject("Operation Failed!"); // Promise is rejected

}

});

**2. Handling Promises**

**a. .then() for Fulfilled Promises**

.then() is used to handle the resolved value of a promise.

promise

.then(result => {

console.log(result); // "Operation Successful!"

});

**b. .catch() for Rejected Promises**

.catch() is used to handle errors or rejected promises.

promise

.then(result => {

console.log(result);

})

.catch(error => {

console.error(error); // "Operation Failed!"

});

**c. .finally() for Cleanup**

.finally() is executed regardless of the promise's outcome.

promise

.then(result => console.log(result))

.catch(error => console.error(error))

.finally(() => console.log("Operation Complete"));

**3. Chaining Promises**

Promises can be chained to perform sequential asynchronous operations.

fetch('https://api.github.com/users/octocat')

.then(response => response.json())

.then(data => {

console.log(data); // Logs user data

return fetch(data.repos\_url);

})

.then(reposResponse => reposResponse.json())

.then(repos => console.log(repos)) // Logs repositories

.catch(error => console.error("Error occurred:", error));

**4. Error Handling in Promises**

**a. Catching Errors in Chains**

If an error occurs in any part of the chain, it propagates down to the nearest .catch().

new Promise((resolve, reject) => {

throw new Error("Something went wrong!");

})

.then(() => console.log("This will not run"))

.catch(error => console.error("Caught error:", error.message)); // "Caught error: Something went wrong!"

**b. Rethrowing Errors**

Errors can be rethrown for additional handling.

new Promise((resolve, reject) => {

reject(new Error("Critical error!"));

})

.catch(error => {

console.error("Handled error:", error.message);

throw error; // Rethrow for further handling

})

.catch(error => console.error("Caught again:", error.message)); // "Caught again: Critical error!"

**5. Creating Promises**

**a. Using the Constructor**

The Promise constructor takes an executor function with resolve and reject.

let myPromise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Task Completed"), 2000);

});

myPromise.then(result => console.log(result)); // "Task Completed" after 2 seconds

**b. Static Methods**

* **Promise.resolve()**: Returns a promise resolved with a given value.

Promise.resolve("Success").then(console.log); // "Success"

* **Promise.reject()**: Returns a promise rejected with a given reason.

Promise.reject("Error").catch(console.error); // "Error"

**6. Parallel Execution with Promise.all**

Promise.all() runs multiple promises in parallel and resolves when all of them are resolved.

let p1 = Promise.resolve(10);

let p2 = new Promise((resolve) => setTimeout(() => resolve(20), 2000));

let p3 = Promise.resolve(30);

Promise.all([p1, p2, p3]).then(results => console.log(results)); // [10, 20, 30]

**7. Handling the First Settled Promise with Promise.race**

Promise.race() resolves or rejects as soon as one of the promises is settled.

let slow = new Promise(resolve => setTimeout(() => resolve("Slow"), 3000));

let fast = new Promise(resolve => setTimeout(() => resolve("Fast"), 1000));

Promise.race([slow, fast]).then(result => console.log(result)); // "Fast"

**8. Handling All Settled Promises with Promise.allSettled**

Promise.allSettled() waits for all promises to settle (fulfilled or rejected).

let p1 = Promise.resolve("Success");

let p2 = Promise.reject("Failure");

let p3 = Promise.resolve("Another Success");

Promise.allSettled([p1, p2, p3]).then(results => console.log(results));

/\*

[

{ status: 'fulfilled', value: 'Success' },

{ status: 'rejected', reason: 'Failure' },

{ status: 'fulfilled', value: 'Another Success' }

]

\*/

**9. Unhandled Rejections**

Unhandled promise rejections are tracked and can trigger a global handler.

window.addEventListener('unhandledrejection', event => {

console.error("Unhandled rejection:", event.reason);

});

new Promise((\_, reject) => reject("Oops!")); // Triggers the handler

**10. Summary**

* Promises simplify async code, making it readable and maintainable.
* Use .then() for handling success and .catch() for errors.
* Use static methods like Promise.all, Promise.race, and Promise.allSettled for advanced use cases.
* Always handle potential errors to avoid unhandled rejections.

**Async/Await in**

async/await is a modern and cleaner syntax for working with promises in . It allows you to write asynchronous code that is easy to read and reason about, as it resembles synchronous code.

**1. Async Functions**

An async function always returns a **promise**. The async keyword makes a function asynchronous, which means it wraps the return value in a promise, even if it's not explicitly a promise.

**Example:**

async function exampleAsync() {

return "Hello, Async!";

}

exampleAsync().then(result => console.log(result)); // Output: Hello, Async!

This is equivalent to:

function exampleAsync() {

return Promise.resolve("Hello, Async!");

}

**Key Points:**

* If a non-promise value is returned, async wraps it in a resolved promise.
* If a promise is returned, it’s used as-is.

**2. Await Keyword**

The await keyword pauses the execution of an async function until a promise is settled (resolved or rejected). The result of the promise is returned once it’s resolved.

**Syntax:**

let result = await promise;

**Example:**

async function fetchData() {

let promise = new Promise((resolve) => {

setTimeout(() => resolve("Data received!"), 2000);

});

let result = await promise; // Pauses here until the promise resolves

console.log(result); // Output: Data received!

}

fetchData();

**3. Error Handling**

When a promise is rejected, await throws the error. To handle errors, use a try...catch block inside the async function.

**Example:**

async function fetchData() {

try {

let response = await fetch("http://invalid-url");

let data = await response.json();

console.log(data);

} catch (error) {

console.error("Error occurred:", error.message);

}

}

fetchData();

If no try...catch is provided, the promise returned by the async function will be rejected, and you can handle it using .catch().

**4. Using Async/Await with Promises**

When dealing with multiple promises, you can still use Promise.all with await.

**Example:**

async function fetchAllData() {

let urls = ["https://api.github.com/users/user1", "https://api.github.com/users/user2"];

try {

let results = await Promise.all(urls.map(url => fetch(url).then(res => res.json())));

console.log(results);

} catch (error) {

console.error("Error fetching data:", error.message);

}

}

fetchAllData();

**5. Practical Use Case**

**Fetching and Displaying Data:**

async function showUserAvatar() {

try {

// Fetch user data

let response = await fetch("https://api.github.com/users/octocat");

let user = await response.json();

// Create and display avatar

let img = document.createElement("img");

img.src = user.avatar\_url;

img.alt = "User Avatar";

document.body.appendChild(img);

// Wait 3 seconds

await new Promise(resolve => setTimeout(resolve, 3000));

// Remove avatar

img.remove();

console.log("Avatar removed.");

} catch (error) {

console.error("Error:", error.message);

}

}

showUserAvatar();

**6. Async Class Methods**

You can use async in class methods just like in regular functions.

**Example:**

class User {

async getName() {

return "John Doe";

}

}

let user = new User();

user.getName().then(name => console.log(name)); // Output: John Doe

**7. Awaiting Thenables**

A thenable is an object with a then method. await works with promises and thenables.

**Example:**

class Thenable {

constructor(value) {

this.value = value;

}

then(resolve) {

setTimeout(() => resolve(this.value \* 2), 1000);

}

}

async function processThenable() {

let result = await new Thenable(5);

console.log(result); // Output: 10 (after 1 second)

}

processThenable();

**8. Top-Level Await**

Modern environments (like ES modules) support top-level await. This means you can use await outside of async functions, but only in module scripts.

**Example:**

let response = await fetch("https://api.github.com/users/octocat");

let user = await response.json();

console.log(user);

**9. Advantages of Async/Await**

* **Improved Readability:** It eliminates the need for .then() and .catch(), making code easier to follow.
* **Error Handling:** Works seamlessly with try...catch.
* **Sequential Execution:** Ensures steps are executed in the correct order.

**10. Limitations**

* **Can Only Be Used Inside Async Functions:** await cannot be used in regular functions.
* **Doesn't Cancel Promises:** If a promise is no longer needed, you must handle its cancellation manually.

**Objects :-**

**1. What are Objects?**

Objects are a collection of **key-value pairs**. Each key (also known as a "property name") maps to a value, which can be of any data type: string, number, boolean, array, function, or even another object.

Think of an object as a **real-world entity** with properties and behaviors. For example:

let car = {

brand: "Toyota",

model: "Corolla",

year: 2020,

start: function() {

console.log("Car started");

}

};

**2. How to Create Objects**

You can create objects using:

**a. Object Literal Syntax (Recommended)**

let user = {

name: "John",

age: 30

};

console.log(user.name); // "John"

**b. Object Constructor Syntax**

let user = new Object();

user.name = "John";

user.age = 30;

console.log(user.name); // "John"

**3. Accessing Object Properties**

You can access properties using:

**a. Dot Notation**

console.log(user.name); // "John"

**b. Square Bracket Notation**

Use square brackets when:

* Property names contain spaces or special characters.
* Property names are stored in variables.

Example:

let user = { "likes birds": true };

console.log(user["likes birds"]); // true

**4. Adding, Modifying, and Deleting Properties**

**a. Add/Modify**

user.isAdmin = true; // Adding

user.name = "Jane"; // Modifying

console.log(user); // { name: "Jane", age: 30, isAdmin: true }

**b. Delete**

delete user.age;

console.log(user); // { name: "Jane", isAdmin: true }

**5. Computed Property Names**

You can use a variable or expression as a property name by wrapping it in square brackets [].

Example:

let fruit = "apple";

let bag = {

[fruit]: 5

};

console.log(bag.apple); // 5

**6. Property Shorthand**

If the property name and variable name are the same, you can use shorthand:

function createUser(name, age) {

return { name, age };

}

let user = createUser("John", 30);

console.log(user); // { name: "John", age: 30 }

**7. Property Existence Check**

To check if a property exists:

**a. Compare with undefined**

console.log(user.age !== undefined); // true if age exists

**b. Use the in operator**

console.log("age" in user); // true

**8. Iterating Over Object Properties**

Use the for...in loop to iterate over all properties of an object:

for (let key in user) {

console.log(`${key}: ${user[key]}`);

}

// Output:

// name: John

// age: 30

**9. Property Order**

Object property keys are ordered as:

1. **Integer keys** (sorted numerically).
2. **String keys** (in the order they were added).

Example:

let obj = { "2": "two", "1": "one", "a": "A" };

for (let key in obj) {

console.log(key); // 1, 2, a

}

**10. Nested Objects**

Objects can contain other objects:

let user = {

name: "John",

address: {

city: "New York",

zip: "10001"

}

};

console.log(user.address.city); // "New York"

**11. Methods in Objects**

A method is a function stored in an object:

let user = {

name: "John",

greet() {

console.log(`Hello, ${this.name}`);

}

};

user.greet(); // "Hello, John"

**12. Dynamic Properties**

You can dynamically add or remove properties at runtime:

let user = {};

user["newProp"] = "dynamic value";

console.log(user.newProp); // "dynamic value"

**13. Object Cloning**

To clone an object:

**a. Using Object.assign**

let user = { name: "John" };

let clone = Object.assign({}, user);

console.log(clone); // { name: "John" }

**b. Using Spread Operator (ES6+)**

let clone = { ...user };

console.log(clone); // { name: "John" }

**14. Object Destructuring**

Extract properties into variables:

let user = { name: "John", age: 30 };

let { name, age } = user;

console.log(name); // "John"

console.log(age); // 30

**Type Guards:-**

A **type guard function** in is a special function used to determine the type of a value at runtime. Type guards narrow down the type of a variable within a specific code block, ensuring type safety and reducing errors.

Here’s a **step-by-step explanation** of type guard functions:

**Step 1: Understanding the Need for Type Guards**

In , when you work with **union types** (e.g., string | number), you may want to perform operations that are specific to one of the types. Without a type guard, doesn’t know which type you're dealing with, leading to type errors.

**Example:**

function printLength(input: string | number) {

console.log(input.length); // Error: Property 'length' does not exist on type 'number'.

}

Type guards help safely narrow down input to string or number at runtime.

**Step 2: Syntax and Purpose**

A type guard is a function or construct that checks the type of a value and informs about it.

**Basic Syntax:**

function isString(value: unknown): value is string {

return typeof value === "string";

}

Here, value is string tells that if the function returns true, the value is of type string.

**Step 3: Writing a Type Guard Function**

Let's implement a type guard function with the isString example:

1. **Function Declaration**:
   * The function takes an argument of type unknown or a union type (e.g., string | number).
   * Use the return type value is string to indicate it’s a type guard.

function isString(value: unknown): value is string {

// Check if the value is a string

return typeof value === "string";

}

1. **Return a Boolean**:
   * The function returns true if the type matches (typeof value === "string").
   * Otherwise, it returns false.

**Step 4: Using the Type Guard in Code**

Now, use the type guard in a function to handle different types safely.

function printLength(input: string | number) {

if (isString(input)) {

// now knows `input` is a string

console.log("String length:", input.length);

} else {

// now knows `input` is a number

console.log("Number value:", input);

}

}

Here’s what happens:

* When isString(input) returns true, narrows the type to string.
* When it returns false, narrows the type to number.

**Step 5: Built-in Type Guards**

provides several built-in ways to narrow types without custom functions:

1. **typeof**:

function printType(value: string | number) {

if (typeof value === "string") {

console.log("It's a string:", value);

} else {

console.log("It's a number:", value);

}

}

1. **instanceof**:

class Dog {

bark() {

console.log("Woof!");

}

}

class Cat {

meow() {

console.log("Meow!");

}

}

function makeSound(animal: Dog | Cat) {

if (animal instanceof Dog) {

animal.bark(); // Safe: knows `animal` is Dog

} else {

animal.meow(); // Safe: knows `animal` is Cat

}

}

1. **in Operator**:
   * Use in to check if a property exists in an object.

function processAnimal(animal: { bark?: () => void; meow?: () => void }) {

if ("bark" in animal) {

animal.bark?.();

} else {

animal.meow?.();

}

}

**Step 6: Advanced Type Guards**

**Example 1: Type Guard for Arrays**

function isArray<T>(value: unknown): value is T[] {

return Array.isArray(value);

}

const data: unknown = [1, 2, 3];

if (isArray<number>(data)) {

console.log("Array sum:", data.reduce((sum, num) => sum + num, 0)); // Safe

}

**Example 2: Type Guard for Objects**

function isUser(obj: unknown): obj is { name: string; age: number } {

return typeof obj === "object" && obj !== null && "name" in obj && "age" in obj;

}

const person = { name: "John", age: 30 };

if (isUser(person)) {

console.log(`User Name: ${person.name}, Age: ${person.age}`);

}

**Step 7: Best Practices**

1. **Use Narrow Checks**:
   * Type guards should be specific and concise to avoid unnecessary complexity.
2. **Leverage Built-in Guards**:
   * Prefer typeof, instanceof, or in for common checks before writing custom guards.
3. **Combine Guards**:
   * Use multiple guards when working with complex data structures or nested types.
4. **Test Edge Cases**:
   * Ensure your type guards handle null, undefined, and unexpected types gracefully.

**Step 8: Common Errors to Avoid**

1. **Using Non-Boolean Returns**:
   * Type guard functions must return true or false. Any other return value will fail.
2. **Skipping Type Assertions**:
   * Failing to use value is Type in the return type of the function leads to unsafe code.
3. **Not Handling Complex Structures**:
   * Type guards should handle nested or recursive structures carefully, as they can cause runtime issues if not handled properly.

**Final Example: Advanced Type Guard**

Here’s an example combining multiple concepts:

type Animal = { name: string; type: "dog" | "cat" };

type Dog = Animal & { type: "dog"; bark: () => void };

type Cat = Animal & { type: "cat"; meow: () => void };

function isDog(animal: Animal): animal is Dog {

return animal.type === "dog" && "bark" in animal;

}

function isCat(animal: Animal): animal is Cat {

return animal.type === "cat" && "meow" in animal;

}

const pet: Animal = { name: "Rex", type: "dog", bark: () => console.log("Woof!") };

if (isDog(pet)) {

pet.bark(); // knows pet is Dog

} else if (isCat(pet)) {

pet.meow(); // knows pet is Cat

}

**Utility Types Explanation :-**

**TypeScript Utility Types Explanation: Pick<Type, Keys>, Omit<Type, Keys>, and NonNullable<Type>**

These utility types help manipulate and transform existing types in TypeScript. They’re especially useful when working with complex types or objects, allowing you to create new, specialized types without rewriting code.

**1. Pick<Type, Keys>**

**Definition:**

Pick<Type, Keys> creates a new type by **selecting specific properties** (Keys) from an existing type (Type).

* **Type**: The original type (an interface or object type).
* **Keys**: A string literal or a union of string literals representing the keys to pick from the Type.

**Step-by-Step Explanation:**

1. You start with a type (Type) that has several properties.
2. Use Pick to create a new type by selecting only the properties you need.
3. The result is a new type that includes only the specified keys.

**Example:**

interface Todo {

title: string;

description: string;

completed: boolean;

}

// Create a new type with only the "title" and "completed" properties.

type TodoPreview = Pick<Todo, "title" | "completed">;

// Using the new type

const todo: TodoPreview = {

title: "Clean room", // Valid: "title" is included.

completed: false, // Valid: "completed" is included.

};

console.log(todo); // Output: { title: "Clean room", completed: false }

**Key Concepts:**

* **Use Case**: You want to extract only some properties from a larger type, reducing unnecessary data.
* **Error Handling**: If you specify a key that doesn’t exist in the original type, TypeScript throws an error.

**2. Omit<Type, Keys>**

**Definition:**

Omit<Type, Keys> creates a new type by **excluding specific properties** (Keys) from an existing type (Type). It’s the **opposite of Pick**.

* **Type**: The original type (an interface or object type).
* **Keys**: A string literal or a union of string literals representing the keys to exclude from the Type.

**Step-by-Step Explanation:**

1. You start with a type (Type) that has several properties.
2. Use Omit to create a new type by excluding unwanted properties.
3. The result is a new type with all the properties of the original type, except for the ones you excluded.

**Example:**

interface Todo {

title: string;

description: string;

completed: boolean;

createdAt: number;

}

// Create a new type without the "description" property.

type TodoPreview = Omit<Todo, "description">;

// Using the new type

const todo: TodoPreview = {

title: "Clean room", // Valid: "title" is included.

completed: false, // Valid: "completed" is included.

createdAt: 1615544252770, // Valid: "createdAt" is included.

};

console.log(todo); // Output: { title: "Clean room", completed: false, createdAt: 1615544252770 }

**Key Concepts:**

* **Use Case**: You want to work with an object type but need to remove some properties to simplify or refine the structure.
* **Error Handling**: If you try to exclude a property that doesn’t exist, TypeScript throws an error.

**Comparison: Pick vs Omit**

* **Pick** selects specific properties.
* **Omit** excludes specific properties.

// Given the Todo type:

interface Todo {

title: string;

description: string;

completed: boolean;

createdAt: number;

}

// Pick example: Create a type with only "title" and "completed".

type PickedTodo = Pick<Todo, "title" | "completed">;

// Omit example: Create a type excluding "description".

type OmittedTodo = Omit<Todo, "description">;

**3. NonNullable<Type>**

**Definition:**

NonNullable<Type> constructs a type by **removing null and undefined** from Type.

* **Type**: The original type, which may include null or undefined.

**Step-by-Step Explanation:**

1. Use NonNullable to remove the null and undefined values from a type.
2. The resulting type is cleaner and more restricted.

**Example:**

typescript

CopyEdit

// Example with union types

type Name = string | null | undefined;

// Create a new type that removes null and undefined

type ValidName = NonNullable<Name>;

const validName: ValidName = "John"; // Valid: string is allowed.

// const invalidName: ValidName = null; // Error: null is not assignable.

// const invalidName2: ValidName = undefined; // Error: undefined is not assignable.

**Key Concepts:**

* **Use Case**: When you want to ensure a value is neither null nor undefined.
* **Error Handling**: If you mistakenly try to assign null or undefined, TypeScript helps catch the error.

**Real-World Use Case of NonNullable<Type>**

You might encounter null or undefined in APIs or optional fields. NonNullable ensures you only work with valid values.

typescript

CopyEdit

interface User {

id: number | null;

name: string | undefined;

}

type ValidUser = NonNullable<User["id"]>; // Ensures `id` is not null or undefined.

const userId: ValidUser = 42; // Valid

// const invalidUserId: ValidUser = null; // Error

**Summary**

**1. Pick<Type, Keys>**

* **Purpose**: Extract specific properties from a type.
* **Usage**: When you need a subset of an object type.
* **Example**: Pick<Todo, "title" | "completed">.

**2. Omit<Type, Keys>**

* **Purpose**: Exclude specific properties from a type.
* **Usage**: When you want to remove unwanted properties.
* **Example**: Omit<Todo, "description">.

**3. NonNullable<Type>**

* **Purpose**: Remove null and undefined from a type.
* **Usage**: When working with values that should never be null or undefined.
* **Example**: NonNullable<string | null | undefined> results in string.

**When to Use Each**

* Use **Pick** to create a type with only the properties you need.
* Use **Omit** to create a type without certain properties.
* Use **NonNullable** to ensure your values are always valid (not null or undefined).