JavaScript Promises

In Detail | ES6

Mallikarjun | @CodeBustler

JS Promises

JavaScript is a **single threaded,** two bits of script cannot run at the same time; they have to run **one after another**.

Promises are **used to handle asynchronous** operations in Javasript. They are **easy to manage when dealing with multiple async operations** where callback can create **callback hell** leading to unmanageable code

A **Promise is object** that keep track about whether a certain event has happened already or not **Determines what happens after** the events has happend

@CodeBustler

JS Promises | States

A JavaScript Promise object can be :

- Pending
- Fulfilled
- Rejected

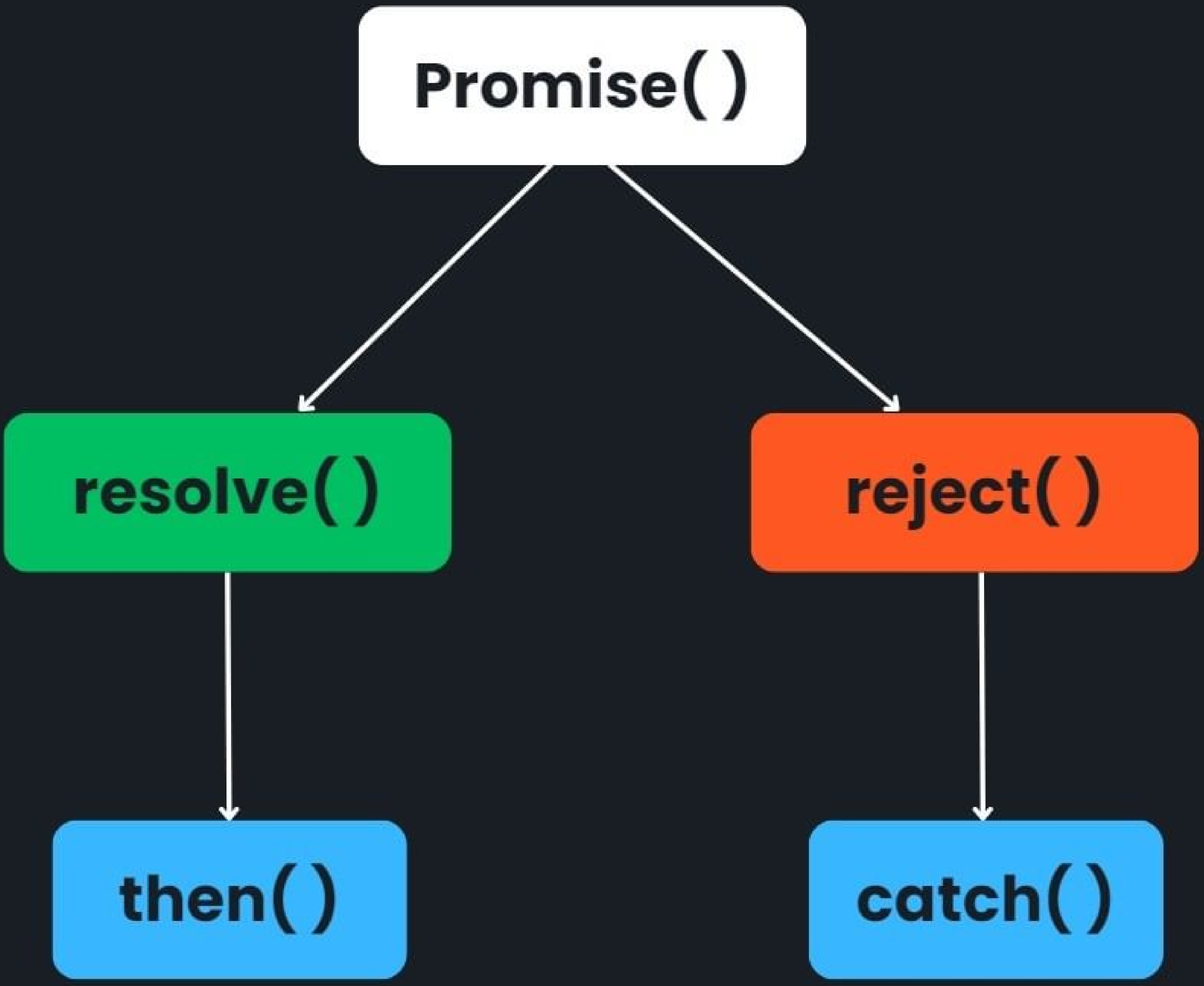
Pending : initial state, neither fulfilled nor rejected. | working, the result is **undefined**

Fulfilled : Operation completed successfully. | The result is a **value**

Rejected : Operation failed. | The result is an **error** object.

@CodeBustler

JS Promises



The .then() and .catch() methods are **inbuilt callback functions** that allow you to handle the **resolved & rejected states** of a promise, respectively.

@CodeBustler

Promise Syntax

```
let promise = new Promise((resolve, reject) ⇒ {
  //Executor
});
```

resolve and **reject** are two callbacks provided by javascript itself

resolve(value) : If the job is finished successfully

reject(error) : If the job fails

JS Promises | Example

```
let prom = new Promise((resolve, reject) ⇒ {
  console.log("Please Wait..."); // Pending State

  setTimeout(() ⇒ {
    if (2 > 1) { // Condition
      resolve("Success"); // onFulfilment
    } else {
      reject("Failed"); // OnRejection
    }
  }, 3000);
});

prom
  .then((result) ⇒ {
    console.log(result);
  })
  .catch((result) ⇒ {
    console.log(result);
  });

// Success | (Condition Fulfilment)
```

Result value from **resolve()**

Error message from **reject()**

@CodeBustler

JS Promise Chaining

Promise chaining in JavaScript allows you to **execute a sequence of asynchronous operations in a specific order,** one after another.

It provides a **clean and organized** way to handle the results of each operation and pass them to the next one.

Example

@CodeBustler

JS Promise Chaining

Example

```javascript
// 2 Different Promises

function asyncOperation1() {
  return new Promise((resolve) ⇒ {
    setTimeout(() ⇒ {
      resolve("Operation 1 completed");
    }, 2000);
  });
}

function asyncOperation2() {
  return new Promise((resolve) ⇒ {
    setTimeout(() ⇒ {
      resolve("Operation 2 completed");
    }, 3000);
  });
}
```

```javascript
asyncOperation1()
  .then((result) ⇒ {
    console.log(result);
    // Output: Operation 1 completed

    return asyncOperation2();
    // Return a new promise for chaining
  })
  .then((result) ⇒ {
    console.log(result);
    // Output: Operation 2 completed

    /* You can continue chaining with
       more .then() if needed */
  })
  .catch((error) ⇒ {
    console.log("Error:", error);
  });
```

@CodeBustler

Attaching Multiple Handler

You can add multiple handlers to a promise using the **.then()** method.

Each .then() method can have its **own** success callback to handle the resolved state of the promise.

```
// let p is a promise

p.then(handler1);

p.then(handler2);

p.then(handler3);
```

Runs Independently

@CodeBustler

@CodeBustler

Mallikarjun | Web Developer

Follow For More !

And i need dopamine ⚡