cheat sheet, concentrating only on the core Promise API.

JavaScript Promises Cheat Sheet (Core Promises Only)

Promises are a powerful mechanism in JavaScript for handling **asynchronous operations**. They provide a structured way to deal with tasks that take time to complete, like fetching data from a server or setting a timer.

---

1. Core Concepts: The Life Cycle of a Promise

A Promise object represents an operation that hasn't completed yet but is expected to in the future.

* **States of a Promise:** A Promise is always in one of these three mutually exclusive states:

    * **Pending:** The initial state. The asynchronous operation is still running.
    * **Fulfilled (Resolved):** The operation completed successfully. The promise now holds a `value`.
    * **Rejected:** The operation failed. The promise now holds a `reason` (typically an `Error` object).

* **Immutability:** Once a Promise moves from `pending` to either `fulfilled` or `rejected` (it becomes "settled"), its state and its value/reason cannot change.

---

2. Creating a Promise

You create a new Promise instance using the `Promise` constructor. It takes a single argument: an "executor" function.

```javascript
new Promise((resolve, reject) => {
  // --- This is the 'executor' function ---
  // It contains the asynchronous code.

  // Simulate an asynchronous operation (e.g., a network request)
  const isOperationSuccessful = Math.random() > 0.5; // Random success/failure

  setTimeout(() => { // Simulate a delay
    if (isOperationSuccessful) {
      // Call resolve() when the operation succeeds.
      // Pass the successful result as an argument.
      resolve("   Data successfully fetched!");
    } else {
      // Call reject() when the operation fails.
      // Pass an Error object or a reason for the failure.
      reject(new Error("   Failed to fetch data. Network error."));
    }
  }, 1500); // 1.5 seconds delay
});
```

* **`executor` function:** This function is executed immediately by the `Promise` constructor. It receives two arguments:

    * **`resolve(value)`:** A function you call when the asynchronous operation completes successfully. `value` is the result you want to pass on.
    * **`reject(reason)`:** A function you call when the asynchronous operation fails. `reason` is typically an `Error` object explaining why it failed.

---

3. Consuming a Promise (Handling Outcomes)

Once you have a Promise, you attach "handlers" to it to react to its eventual state (`fulfilled` or `rejected`).

`.`then()` - Handling Success and Chaining

The primary way to consume a Promise.

```javascript
myPromise // Assume 'myPromise' is an existing Promise instance
  .then(value => {
    // This callback function runs ONLY if the promise is RESOLVED (fulfilled).
    // 'value' is the data passed to resolve().
    console.log("Success handler 1:", value);

    // YOU CAN RETURN A VALUE HERE:
    return "This value will be passed to the next .then()";

    // OR YOU CAN RETURN ANOTHER PROMISE FOR CHAINING:
    // return anotherAsyncTask();
  })
  .then(nextValue => {
    // This callback runs if the previous .then() returned a value or resolved a promise.
    // 'nextValue' is the value returned by the previous .then() handler.
    console.log("Success handler 2 (chained):", nextValue);
  });
```

`.`catch()` - Handling Errors (Rejections)

Specifically designed for handling Promise rejections.

```javascript
myPromise
  .then(value => {
    console.log("Operation successful:", value);
    // If an error occurs here (e.g., throw new Error()),
    // it will be caught by the .catch() below.
  })
  .catch(reason => {
    // This callback function runs ONLY if the promise is REJECTED.
    // 'reason' is the error/reason passed to reject().
    // It also catches errors thrown in any preceding .then() handlers in the chain.
    console.error("Error occurred:", reason.message || reason);
  });
```

* **Best Practice:** Always include a `.catch()` block at the end of your promise chains to handle potential errors gracefully. Unhandled promise rejections can lead to unexpected behavior or silent failures.

`.`finally()` - Cleanup (Always Runs)

A handler that runs regardless of whether the Promise was fulfilled or rejected.

```javascript
myPromise
  .then(value => {
    console.log("Promise resolved:", value);
  })
  .catch(reason => {
    console.error("Promise rejected:", reason);
  })
  .finally(() => {
    // This callback runs when the promise is "settled" (either fulfilled or rejected).
    // It doesn't receive any arguments (no value or reason).
    // Ideal for cleanup tasks: hiding loading indicators, closing connections, etc.
    console.log("Promise settled. Cleanup operations complete.");
  });
```

---

4. Promise Chaining: Sequential Asynchronous Operations

Chaining allows you to execute asynchronous operations in sequence, where each subsequent operation depends on the successful completion of the previous one.

```javascript
// Function that returns a promise after a delay
function downloadData(url) {
  console.log(`Downloading from: ${url}...`);
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (url === "https://api.example.com/data") {
        resolve({
          id: 1,
          content: "Downloaded data"
        });
      } else {
        reject(new Error("Invalid URL"));
      }
    }, 1000);
  });
}

// Function that returns a promise to process the downloaded data
function processData(data) {
  console.log(`Processing data ID: ${data.id}...`);
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (data.content) {
        resolve({
          processedContent: data.content.toUpperCase() + " (PROCESSED)"
```

```
      });
    } else {
      reject(new Error("No content to process"));
    }
  }, 800);
 });
}

// Chaining them together:
downloadData("https://api.example.com/data")
 .then(rawData => {
  console.log("Raw data received:", rawData);
  // RETURN a new promise. The next .then() will wait for THIS promise to resolve.
  return processData(rawData);
 })
 .then(processedResult => {
  console.log("Final processed result:", processedResult);
 })
 .catch(error => {
  // A single .catch() at the end handles errors from ANYWHERE in the chain.
  console.error("An error occurred during the process:", error.message);
 })
 .finally(() => {
  console.log("Download and process flow complete.");
 });

// Example of an error in the chain:
downloadData("invalid-url")
 .then(rawData => {
  console.log("This will not run:", rawData);
  return processData(rawData);
 })
 .then(processedResult => {
  console.log("Nor this:", processedResult);
 })
 .catch(error => {
  console.error("Caught error in chain (invalid URL):", error.message); // "Invalid URL"
 });
```

* **Key to Chaining:** When a `.then()` handler returns a **new Promise**, the subsequent `.then()` in the chain will wait for *that new Promise* to settle before executing. If it returns a non-Promise value, that value is passed directly to the next `.then()`.

---

5. Static Promise Methods (Combining Promises)

These methods operate on multiple Promises.

`Promise.all(iterable)`

* **Purpose:** Waits for *all* promises in the input iterable (e.g., an array) to resolve.
* **Result (on success):** Resolves with an array of the resolved values, in the same order as the input promises.
* **Result (on failure):** Rejects immediately with the `reason` of the *first* promise that rejects.

```javascript
const promiseA = new Promise(res => setTimeout(() => res("Result A"), 1000));
const promiseB = new Promise(res => setTimeout(() => res("Result B"), 500));
const promiseC = Promise.resolve("Result C (immediate)");
const promiseFail = new Promise((_, rej) => setTimeout(() => rej(new Error("Failure!")), 200));

console.log("\n--- Promise.all Examples ---");

Promise.all([promiseA, promiseB, promiseC])
  .then(allResults => {
    console.log("All successful:", allResults); // ["Result A", "Result B", "Result C (immediate)"]
  })
  .catch(error => {
    console.error("One of them failed:", error.message);
  });

Promise.all([promiseA, promiseFail, promiseB]) // promiseFail will cause immediate rejection
  .then(allResults => {
    console.log("This won't run:", allResults);
  })
  .catch(error => {
    console.error("Caught error from Promise.all (first failure):", error.message); // "Failure!"
  });
```

`Promise.race(iterable)`

* **Purpose:** Waits for the *first* promise in the input iterable to settle (either fulfill or reject).
* **Result:** Settles with the value or reason of that first settled promise.

```javascript
const pSlow = new Promise(res => setTimeout(() => res("Slow operation"), 1000));
const pFast = new Promise(res => setTimeout(() => res("Fast operation"), 300));
const pImmediateReject = new Promise((_, rej) => rej("Immediate rejection"));

console.log("\n--- Promise.race Examples ---");

Promise.race([pSlow, pFast])
  .then(winner => {
    console.log("Race winner:", winner); // "Fast operation"
  })
  .catch(error => {
    console.error("Race error:", error);
  });

Promise.race([pSlow, pImmediateReject, pFast]) // pImmediateReject settles first
  .then(winner => {
    console.log("This will not run:", winner);
  })
  .catch(error => {
    console.error("Race error (first to settle was a rejection):", error); // "Immediate rejection"
  });
```

`Promise.allSettled(iterable)` (ES2020+)

* **Purpose:** Waits for *all* promises in the input iterable to settle (either fulfill or reject). It never rejects itself.
* **Result:** Always resolves with an array of objects. Each object describes the outcome of an individual promise (`status: 'fulfilled' | 'rejected'`, plus `value` or `reason`).

```javascript
const p1 = Promise.resolve("Value 1");
const p2 = new Promise((_, rej) => setTimeout(() => rej(new Error("Error 2")), 50));
const p3 = Promise.resolve("Value 3");

console.log("\n--- Promise.allSettled Example ---");

Promise.allSettled([p1, p2, p3])
  .then(results => {
    console.log("All promises settled:", results);
    /* Output will be an array like:
    [
      { status: 'fulfilled', value: 'Value 1' },
      { status: 'rejected', reason: Error: Error 2 },
      { status: 'fulfilled', value: 'Value 3' }
    ]
    */
  });
```

`Promise.any(iterable)` (ES2021+)

* **Purpose:** Waits for the *first* promise in the input iterable to *fulfill*.
* **Result (on success):** Resolves with the `value` of the first promise that fulfills.
* **Result (on failure):** Only rejects if *all* of the promises in the iterable reject. In that case, it rejects with an `AggregateError` containing all rejection reasons.

```javascript
const pFailA = Promise.reject("Reason A");
const pSuccessEarly = new Promise(res => setTimeout(() => res("Early Success!"), 50));
const pFailB = Promise.reject("Reason B");
const pSuccessLate = new Promise(res => setTimeout(() => res("Late Success!"), 200));

console.log("\n--- Promise.any Examples ---");

Promise.any([pFailA, pSuccessEarly, pFailB, pSuccessLate])
  .then(winner => {
    console.log("Any winner:", winner); // "Early Success!"
  })
  .catch(error => {
    console.error("Any error:", error.errors); // This won't run in this case
  });

Promise.any([pFailA, pFailB]) // All promises reject
  .then(winner => {
    console.log("This won't run:", winner);
  })
  .catch(error => {
    // error will be an AggregateError with error.errors containing ["Reason A", "Reason B"]
    console.error("Any failed (all rejected):", error.errors);
  });
```

---

6. Important Reminders

* **Asynchronous Nature:** The code immediately following a Promise creation or a `.then()`/`.catch()` call will execute *before* the Promise settles. Don't expect immediate results; expect them in the callbacks.
* **Error Handling is Crucial:** Always have a `.catch()` in your chain. Unhandled promise rejections are a common source of bugs.
* **Return from `.then()`:** For proper chaining, ensure your `.then()` callbacks either return a value (which becomes the input for the next `.then()`) or another Promise (which the chain will wait for).

This cheat sheet provides a solid foundation for mastering Promises. Practice building and consuming them in your code!