# Day 3 Trainee Tasks

## Learnings Covered:

- **ES6 Numbers**:
  - Binary (`0b`) and Octal (`0o`) literals.
  - `Number.isFinite()`, `Number.isNaN()`, `Number.isInteger()`, `Number.isSafeInteger()`.
  - Importance of safe integers.
- **ES6 Math Functions**:
  - `Math.acosh()`, `Math.asinh()`, `Math.atanh()`, `Math.cosh()`, `Math.sinh()`, `Math.tanh()`, `Math.cbrt()`, `Math.clz32()`, `Math.log1p()`, `Math.log10()`, `Math.log2()`, `Math.expm1()`, `Math.hypot()`, `Math.fround()`, `Math.sign()`, `Math.trunc()`.
- **ES6 Array Functions**:
  - `Array.prototype.find()`, `findIndex()`, `fill()`, `copyWithin()`, `from()`, `of()`, `keys()`.
- **Sets in ES6**:
  - `Set` operations: `add()`, `delete()`, `has()`, `clear()`, iteration.
  - Set operations like union, intersection, difference.
- **Maps in ES6**:
  - Flexible key-value pairs,
    methods: `set()`, `get()`, `has()`, `delete()`, `clear()`, `size`, `forEach()`, `keys()`, `values()`, `entries()`.
- **WeakMap in ES6**:
  - Holds weak references, ideal for memory-efficient data management.
  - Methods: `set()`, `get()`, `has()`, `delete()`.
- **WeakSet in ES6**:
  - Stores unique objects, weak references.
  - Methods: `add()`, `has()`, `delete()`.
- **Async Programming in ES6**:
  - Callback functions and issues like callback hell.
  - Promises: `resolve()`, `reject()`, `then()`, `catch()`, `finally()`.
  - Promise chaining, `Promise.all()`, `Promise.race()`, handling timeouts.
- **Async Generators**:
  - Handle asynchronous streams of data with `async function*`.
  - Paired with promises for better control, composability, and parallelism.

## ES6 Numbers

1. **Binary (0b) and Octal (0o) Literals**:
   - In ES6, binary numbers are represented with `0b` and octal with `0o`. Hexadecimal numbers use `0x`:

     ```
     let binary = 0b101;   // 5 in decimal
     let octal = 0o10;     // 8 in decimal
     let hex = 0x1F;       // 31 in decimal
     ```

2. **Number Functions**:
   - `Number.isFinite(value)`: Checks if the number is finite.
   - `Number.isNaN(value)`: Checks if the value is `NaN`.
   - `Number.isInteger(value)`: Checks if the value is an integer.
   - `Number.isSafeInteger(value)`: Ensures that a number is a safe integer. Safe integers are within the range of `-(2^53 - 1)` and `2^53 - 1` (JavaScript can accurately represent these values).

     ```
     Number.MAX_SAFE_INTEGER;  // 9007199254740991
     Number.MIN_SAFE_INTEGER;  // -9007199254740991
     ```

3. **Why Safe Integers?**: JavaScript represents numbers with floating-point arithmetic, which means precision can degrade with larger values. `MAX_SAFE_INTEGER` and `MIN_SAFE_INTEGER` ensure operations remain precise.

---

## ES6 Math Functions

1. **`Math.acosh(x)`**: Returns the inverse hyperbolic cosine of `x`.
2. **`Math.asinh(x)`**: Returns the inverse hyperbolic sine of `x`.
3. **`Math.atanh(x)`**: Returns the inverse hyperbolic tangent of `x`.
4. **`Math.cosh(x)`**: Returns the hyperbolic cosine of `x`.
5. **`Math.sinh(x)`**: Returns the hyperbolic sine of `x`.
6. **`Math.tanh(x)`**: Returns the hyperbolic tangent of `x`.
7. **`Math.cbrt(x)`**: Returns the cube root of `x`.
8. **`Math.clz32(x)`**: Returns the number of leading zero bits in the 32-bit binary representation of `x`.
9. **`Math.log1p(x)`**: Returns the natural logarithm of `1 + x`.
10. **`Math.log10(x)`**: Returns the base-10 logarithm of `x`.
11. **`Math.log2(x)`**: Returns the base-2 logarithm of `x`.
12. **`Math.expm1(x)`**: Returns `exp(x) - 1`.
13. **`Math.hypot(...values)`**: Returns the square root of the sum of squares of the arguments.
14. **`Math.fround(x)`**: Returns the nearest 32-bit float representation of `x`.
15. **`Math.sign(x)`**: Returns the sign of `x` (-1, 0, or 1).

16. **`Math.trunc(x)`**: Returns the integer part of `x` by removing any fractional digits.

---

## ES6 Array Functions with Callbacks

1. **`Array.prototype.find(callback)`**: Returns the first element in an array that satisfies the provided testing function.

   ```
   [1, 2, 3, 4].find(x => x > 2);  // 3
   ```

2. **`Array.prototype.findIndex(callback)`**: Returns the index of the first element that satisfies the provided testing function.

   ```
   [1, 2, 3, 4].findIndex(x => x > 2);  // 2
   ```

3. **`Array.prototype.fill(value, start, end)`**: Fills all elements from the start to the end index with a static value.

   ```
   [1, 2, 3].fill(4);  // [4, 4, 4]
   ```

4. **`Array.prototype.copyWithin(target, start, end)`**: Shallow copies part of an array to another location in the same array.

   ```
   [1, 2, 3, 4].copyWithin(0, 2);  // [3, 4, 3, 4]
   ```

5. **`Array.from(iterable)`**: Creates a new array instance from an iterable or array-like object.

   ```
   Array.from('1234');  // ['1', '2', '3', '4']
   ```

6. **`Array.of(...elements)`**: Creates a new array with a variable number of arguments, regardless of type.

   ```
   Array.of(7);  // [7]
   ```

7. **`Array.prototype.keys()`**: Returns a new array iterator that contains the keys for each index.

   ```
   [...['a', 'b'].keys()];  // [0, 1]
   ```

---

## Array Comprehensions in ES6

Although ES6 initially considered adding array comprehensions, they were ultimately not included in the standard. However, a similar behavior can be achieved using higher-order functions such as `map`, `filter`, and `reduce`.

The syntax for array comprehensions (as seen in the image) is no longer valid. Instead, for similar functionality:

```
let ary = [1, 2, 3].map(i => i);  // [1, 2, 3]
```

---

## Sets in ES6

1. **Creating a Set**:

   ```
   let set = new Set([1, 2, 3, 4]);
   ```

2. **Set Methods**:
   - **`add(value)`**: Adds a value to the set.
   - **`delete(value)`**: Removes a value from the set.
   - **`has(value)`**: Checks if the set contains a specific value.
   - **`clear()`**: Removes all elements from the set.
   - **`size`**: Returns the number of values in the set.
3. **Manipulation & Iteration**: Sets can be iterated over using `for...of` or `forEach`:

   ```
   let set = new Set([1, 2, 3]);
   for (let value of set) {
     console.log(value);  // Logs 1, 2, 3
   }
   ```

4. **Use Cases of Sets**:
   - Sets are useful when you need a unique collection of values.
   - They are ideal for operations such as union, intersection, and difference between sets.

   ```
   // Union
   let union = new Set([...setA, ...setB]);

   // Intersection
   let intersection = new Set([...setA].filter(x => setB.has(x)));

   // Difference
   let difference = new Set([...setA].filter(x => !setB.has(x)));
   ```

These ES6 features bring significant enhancements to JavaScript, enabling cleaner and more efficient code for working with numbers, arrays, and sets.

## Maps in ES6

A **Map** in ES6 is a collection of key-value pairs, where both the keys and values can be of any data type, including objects, functions, and primitives. Unlike objects, which can only use strings or symbols as keys, Maps allow more flexibility by enabling the use of any data type as a key.

*Why Use Maps?*

1. **Flexible Key Types**: Maps allow any type (object, function, primitive) to be used as a key, while in objects, keys are limited to strings or symbols.
2. **Ordered**: Maps preserve the order of entries, which means keys are iterated in the order they were inserted.
3. **Size Property**: Maps have a `size` property that returns the number of entries in the map, unlike objects where you have to manually count keys.
4. **Efficient Key Lookups**: Maps are optimized for frequent additions and deletions of key-value pairs.

*Map Methods*

1. **`set(key, value)`**: Adds or updates an element with the specified key and value to the map.

```
let map = new Map();
map.set('name', 'Shafquat');
map.set(1, 'one');
```

2. **`get(key)`**: Returns the value associated with the key or `undefined` if the key is not in the map.

```
map.get('name');  // 'Shafquat'
```

3. **`has(key)`**: Returns `true` if the map contains the key, otherwise `false`.

```
map.has(1);  // true
```

4. **`delete(key)`**: Removes the element with the specified key from the map.

```
map.delete(1);  // Removes the key-value pair with key 1
```

5. **`clear()`**: Removes all elements from the map.

```
map.clear();  // Map is now empty
```

6. **`size`**: Returns the number of key-value pairs in the map.

```
map.size;  // Number of entries in the map
```

7. **`forEach(callback)`**: Iterates over the map's entries in insertion order.

```
map.forEach((value, key) => console.log(key, value));
```

8. **`keys()`**: Returns a new iterator object that contains the keys for each element.

```
for (let key of map.keys()) {
  console.log(key);  // Logs each key
}
```

9. **`values()`**: Returns a new iterator object that contains the values for each element.

```
for (let value of map.values()) {
  console.log(value);  // Logs each value
}
```

10. **`entries()`**: Returns a new iterator object that contains an array of `[key, value]` pairs for each element.

```
for (let entry of map.entries()) {
  console.log(entry);  // Logs [key, value]
}
```

*Use Cases for Maps*

- When you need key-value pairs where keys can be of any type (e.g., objects or arrays as keys).
- When you need ordered data (preserves the order of insertion).
- When frequent addition, deletion, or lookup operations are needed.

---

## WeakMap in ES6

A **WeakMap** is a special type of Map that only holds "weak" references to its keys. This means that if no other references to the keys exist, they are garbage-collected, which helps in cases where you want to manage memory efficiently.

*Why Use WeakMap?*

1. **Memory Efficiency**: WeakMaps allow garbage collection of keys when they are no longer referenced elsewhere, preventing memory leaks.
2. **Limited Use of Keys**: WeakMaps only allow objects as keys (not primitives), making them useful for associating metadata with objects in a way that doesn't prevent garbage collection.

*WeakMap Methods*

1. **`set(key, value)`**: Adds a new key-value pair where the key must be an object.

```
let wm = new WeakMap();
let obj = {};
wm.set(obj, 'some value');
```

2. **`get(key)`**: Returns the value associated with the key, or `undefined` if the key does not exist.

```
wm.get(obj);  // 'some value'
```

3. **`has(key)`**: Returns `true` if the WeakMap contains the key.

```
wm.has(obj);  // true
```

4. **`delete(key)`**: Removes the key and its associated value from the WeakMap.

```
wm.delete(obj);  // Removes the key-value pair
```

*Use Cases for WeakMaps*

- **Private Data for Objects**: Use WeakMap when you need to associate some metadata with objects without modifying the objects themselves or risking memory leaks.

```
const privateData = new WeakMap();
function User(name) {
  privateData.set(this, { name });
}
const user = new User('Shafquat');
console.log(privateData.get(user).name);  // 'Shafquat'
```

- **DOM Element Cache**: WeakMaps can be used to cache or store metadata related to DOM elements, and since DOM elements can be garbage collected when removed from the document, this helps avoid memory leaks.

---

## WeakSet in ES6

A **WeakSet** is a collection of unique objects where the objects are held weakly, meaning if no other reference to an object exists, it can be garbage collected. WeakSets, like WeakMaps, only hold objects as elements and do not prevent garbage collection of those objects.

*Why Use WeakSet?*

1. **Memory Efficiency**: Like WeakMap, WeakSet helps manage memory by allowing the garbage collector to remove objects that are no longer referenced elsewhere.
2. **No Primitives**: WeakSets only store objects (not primitives), making them useful for cases where you need to track object existence without preventing garbage collection.

*WeakSet Methods*

1. **`add(value)`**: Adds an object to the WeakSet.

```
let ws = new WeakSet();
let obj = {name: 'Shafquat'};
ws.add(obj);
```

2. **`has(value)`**: Checks if the object exists in the WeakSet.

```
ws.has(obj);  // true
```

3. **`delete(value)`**: Removes the object from the WeakSet.

```
ws.delete(obj);  // Removes the object
```

*Use Cases for WeakSets*

- **Tracking Object References**: WeakSets can be used to keep track of objects, for example, when you need to verify if an object has been processed or visited in a graph/tree traversal.
- **DOM Element Tracking**: WeakSets can track DOM elements without preventing them from being garbage collected when they are removed from the document.

---

## Differences between Map, WeakMap, Set, and WeakSet

- **Map**: Stores key-value pairs, where keys can be of any type. Keys are strongly referenced.
- **WeakMap**: Stores key-value pairs, but keys must be objects and are weakly referenced.
- **Set**: Stores unique values of any type, strongly referenced.
- **WeakSet**: Stores unique objects, weakly referenced.

In summary, **WeakMap** and **WeakSet** are ideal for cases where you want to associate data with objects but don't want to prevent those objects from being garbage collected. **Map** and **Set** provide flexible and efficient ways to handle collections of key-value pairs or unique values.

## Asynchronous Coding in ES6

Asynchronous programming is a key concept in JavaScript, especially because operations like fetching data from an API or reading files from a disk take time. Instead of blocking the code execution while waiting for these operations to complete, JavaScript provides tools to handle them asynchronously, meaning they run in the background while the rest of your code continues to execute.

*Callback Functions*

Before ES6 introduced Promises, asynchronous operations were primarily handled using callbacks. A **callback** is a function that is passed as an argument to another function and is executed after the completion of some operation.

For example, without promises:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { name: "John" };
    callback(null, data); // first argument is error (null), second is the
result
  }, 2000);
}

fetchData((error, result) => {
  if (error) {
    console.error(error);
  } else {
    console.log(result); // Output: { name: "John" }
  }
});
```

Problems with Callbacks

- **Callback Hell**: When you have multiple asynchronous operations, you might end up nesting callbacks, leading to hard-to-read code, also known as "callback hell."

  ```
  asyncOperation1(function() {
    asyncOperation2(function() {
      asyncOperation3(function() {
        asyncOperation4(function() {
          // deeply nested callbacks
        });
      });
    });
  });
  ```

- **Inversion of Control**: With callbacks, control is handed over to the callback function, which can lead to unpredictability or errors.

*Promises*

**Promises** were introduced in ES6 to provide a cleaner way to handle asynchronous code and avoid callback hell.

A Promise is an object that represents a value that might not be available yet, but will be resolved or rejected in the future.

Promise Lifecycle

A promise can have one of three states:

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation was completed successfully.
3. **Rejected**: The operation failed.

```
const promise = new Promise((resolve, reject) => {
  const success = true;

  setTimeout(() => {
    if (success) {
      resolve('Data fetched successfully!');
    } else {
      reject('Data fetching failed');
    }
  }, 2000);
});

promise
  .then(result => console.log(result))   // If the promise is resolved
  .catch(error => console.error(error)); // If the promise is rejected
```

## Promise Basics

1. **`resolve()`**: Marks the promise as fulfilled (successful).
    - When `resolve` is called, it passes the result to the `.then()` handler.
2. **`reject()`**: Marks the promise as rejected (failed).
    - When `reject` is called, it passes the error to the `.catch()` handler.
3. **`then()`**: Used to handle the resolved result of a promise. You can chain multiple `.then()` calls for more asynchronous operations.
4. **`catch()`**: Used to handle errors or the rejected state of a promise.
5. **`finally()`**: Used to run code after the promise has either resolved or rejected. This is often used for cleanup operations.

Example:

```
const promise = new Promise((resolve, reject) => {
  // Simulating an async operation
  setTimeout(() => {
```

```
    resolve('Operation successful!');
  }, 1000);
});

promise
  .then(result => console.log(result))  // Will log "Operation successful!"
  .catch(error => console.error(error)) // Will not trigger since resolve was
called
  .finally(() => console.log('Operation completed!'));
```

## Advanced Concepts

*1. Promise Chaining*

You can chain promises to run multiple asynchronous operations in sequence. The return value of each `.then()` block becomes the input of the next one.

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Data fetched'), 1000);
});

fetchData
  .then(result => {
    console.log(result); // Data fetched
    return 'Processing data';
  })
  .then(processed => {
    console.log(processed); // Processing data
    return 'Data processed';
  })
  .then(final => console.log(final)) // Data processed
  .catch(error => console.error(error)); // If any step fails, this will be
called
```

*2. Composability of Promises*

Promises can be composed to run multiple operations either in sequence or parallel.

- **`Promise.all()`**: Takes an array of promises and returns a single promise that resolves when all promises in the array are resolved. If any promise is rejected, `Promise.all()` is rejected.

  ```
  const promise1 = new Promise((resolve, reject) => setTimeout(resolve,
  1000, 'Result 1'));
  const promise2 = new Promise((resolve, reject) => setTimeout(resolve,
  2000, 'Result 2'));

  Promise.all([promise1, promise2]).then(results => {
    console.log(results); // [ 'Result 1', 'Result 2' ]
  });
  ```

- **`Promise.race()`**: Takes an array of promises and returns a single promise that resolves or rejects as soon as any of the promises resolves or rejects.

```
const promise1 = new Promise((resolve, reject) => setTimeout(resolve,
1000, 'First one done'));
const promise2 = new Promise((resolve, reject) => setTimeout(resolve,
2000, 'Second one done'));

Promise.race([promise1, promise2]).then(result => {
  console.log(result); // First one done
});
```

*3. Handling Timeout in Promises*

Promises by themselves don't inherently have a timeout. However, you can implement a timeout mechanism by creating a custom promise that rejects after a certain amount of time.

```
const fetchWithTimeout = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(new Error('Request timed out'));
  }, 3000); // 3-second timeout
});

Promise.race([
  fetchData, // Some asynchronous operation
  fetchWithTimeout
]).then(result => {
  console.log(result);
}).catch(error => {
  console.error(error); // Will print "Request timed out" if fetchData takes
longer than 3 seconds
});
```

- **Callback**: A function passed as an argument to another function to be called after an asynchronous operation completes.
- **Promise**: A more modern, cleaner way to handle asynchronous code, avoiding "callback hell." It has three states: `pending`, `resolved` (fulfilled), and `rejected`.
- **Chaining**: Promises can be chained using `.then()` for sequential async operations.
- **`Promise.all()`**: Runs multiple promises in parallel and waits for all to resolve.
- **`Promise.race()`**: Resolves or rejects as soon as the fastest promise resolves or rejects.
- **Timeout**: Can be implemented manually using `Promise.race()` to reject after a certain duration.

## Understanding ES6 `async` Functions and `async Generators`

JavaScript introduced `async` functions in ES6 to simplify asynchronous programming and make code easier to understand and maintain. However, as development needs have grown, there's a more advanced tool called `async generators`, which allows for more granular control over asynchronous operations, especially when working with streams of data. Let's dive into each concept and see how they work individually and in combination.

---

## 1. Async Functions

**`async`** functions are an abstraction over Promises, making it easier to work with asynchronous code. They allow us to use `await` to pause execution until a Promise is resolved or rejected.

### Why use `async` functions?

- **Simplicity**: Async functions allow you to write asynchronous code that looks more like synchronous code, without deeply nested `.then()` and `.catch()` chains.
- **Error handling**: You can use `try...catch` blocks inside `async` functions to handle errors, which makes for a more consistent error-handling experience.

```
async function fetchData() {
  try {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data); // Response data
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}

fetchData(); // Returns a promise
```

### Key features of `async` functions:

- An `async` function always returns a Promise, implicitly.
- You can use `await` inside the function to wait for a Promise to resolve.
- Error handling is simplified with `try...catch` around `await` calls.

---

## 2. Async Generators

**`async generators`** combine the power of `async` functions with **generators**. A generator is a function that can "yield" multiple values over time instead of returning a single result, making them ideal for **streams** of data.

**Why use `async generators`?**

- **Handling streams of data**: They are great when dealing with large data streams or event streams where you want to process data bit by bit as it arrives, such as when reading from a file or streaming data from a server.
- **Fine-grained control**: You can yield data at each step of an asynchronous process, rather than waiting for the whole operation to complete.
- **Memory efficiency**: Instead of waiting for the entire result set to be available, you process one chunk at a time, which reduces memory usage when dealing with large data sets.

**How `async generators` work**: An `async generator` is defined using both `async` and `function*`. Instead of `return`, it uses `yield` to provide values one at a time. With `await`, it waits for asynchronous operations.

```
async function* fetchChunks() {
  for (let i = 1; i <= 5; i++) {
    await new Promise(resolve => setTimeout(resolve, 1000)); // Simulate
async operation
    yield `Chunk ${i}`;
  }
}

(async () => {
  for await (let chunk of fetchChunks()) {
    console.log(chunk); // Logs chunks one by one over time
  }
})();
```

**Key features of `async generators`:**

- An `async generator` produces a stream of asynchronous values over time using `yield`.
- You can consume `async generator` values with a `for await...of` loop.
- Async generators pause at each `yield` and wait until the next value is requested.

---

## 3. Async Generators Paired with Promises

Now, let's explore the pairing of `async generators` and **promises**, which allows for better composition and handling of asynchronous streams.

**Why pair async generators with promises?**

- **Promise-driven control**: Async generators can naturally pair with promises because promises help control when and how the next piece of data in the stream is fetched.

- **Pipelining data**: Often, when you consume streams of data, you want to handle them asynchronously using promises, so pairing async generators with promises allows you to process each piece of data (or "chunk") as it arrives asynchronously.
- **Chaining and Composability**: By pairing with promises, you can easily chain multiple asynchronous operations or even parallelize them for better performance.

Consider this example, where we fetch data asynchronously in chunks and then handle them one by one:

```
async function* fetchDataInChunks() {
  for (let i = 0; i < 3; i++) {
    const data = await new Promise(resolve => setTimeout(() => resolve(`Data
chunk ${i + 1}`), 1000));
    yield data;
  }
}

async function processChunks() {
  for await (let chunk of fetchDataInChunks()) {
    console.log('Processing:', chunk);
  }
}

processChunks();
```

In this example:

- We fetch data in chunks asynchronously using an `async generator`.
- The function waits for each chunk to be fetched (using promises) and processes them one by one.

This pairing of **async generators** and **promises** allows for:

- **Efficient memory usage**: As chunks are processed as soon as they arrive, rather than waiting for all data to be ready.
- **Flexibility**: You can decide what happens after each chunk is fetched, giving you control over the flow of your application.

---

## 4. Why Async Generators with Promises Are Better Than Async Generators Alone

While `async generators` on their own are useful for yielding asynchronous values, when you pair them with **promises**, you gain several advantages:

1. **Promise Composability**: You can compose asynchronous streams with other promises. For instance, you could await multiple promises inside an async generator and yield when all are resolved.

```
async function* asyncGenerator() {
  const promise1 = new Promise(resolve => setTimeout(resolve, 1000,
'First Promise Resolved'));
  const promise2 = new Promise(resolve => setTimeout(resolve, 2000,
'Second Promise Resolved'));
  yield await promise1;
  yield await promise2;
}

(async () => {
  for await (const value of asyncGenerator()) {
    console.log(value); // Logs promises in sequence
  }
})();
```

2. **Advanced Flow Control**: With promises, you have more control over the flow of asynchronous data, like retrying failed promises or handling them in parallel.

```
async function* asyncGenWithRetries() {
  let success = false;
  while (!success) {
    try {
      const data = await fetchData(); // Fetching data (can fail)
      success = true;
      yield data;
    } catch (error) {
      console.error('Retrying due to error:', error);
      await new Promise(resolve => setTimeout(resolve, 1000)); // Retry
after 1 second
    }
  }
}

(async () => {
  for await (const data of asyncGenWithRetries()) {
    console.log(data);
  }
})();
```

3. **Parallelism with `Promise.all`**: You can yield multiple asynchronous tasks at once and process them in parallel, then yield the result back to the generator.

```
async function* parallelDataFetcher() {
  const results = await Promise.all([
    fetch('https://api.example.com/data1'),
    fetch('https://api.example.com/data2')
  ]);

  for (const result of results) {
    yield result.json();
  }
}

(async () => {
  for await (const data of parallelDataFetcher()) {
```

```
    console.log(data); // Process fetched data in parallel
  }
})();
```

- **`async` functions**: Simplify asynchronous programming by allowing you to write code that looks synchronous while managing asynchronous operations behind the scenes with promises.
- **`async generators`**: Provide a way to yield values from asynchronous operations one at a time, useful for working with data streams or processing data in chunks over time.
- **Pairing async generators with promises**: Unlocks the full potential of both tools, allowing for better composability, control over async operations, parallelism, and memory-efficient processing of large data streams.

In short, while `async generators` by themselves allow you to control the flow of data over time, pairing them with promises adds composability, parallelism, and advanced error handling, making them far more powerful for handling complex async workflows.