Promises and Async/Await

A) Creating and Chaining Promises

Easy Examples (5)

Scenario: Simple arithmetic operations using promises.

1. **Basic Promise Resolution**

```
const promise1 = new Promise((resolve, reject) => {
  resolve("Operation Successful");
});

promise1.then(result => console.log(result));
```

Explanation:

- Promise resolves immediately.
- `.then()` handles the resolved value.

Output:

```
Operation Successful
```

2. **Promise with Delay**

```
const delay = new Promise((resolve) => {
  setTimeout(() => resolve("Done after 2 seconds"), 2000);
});

delay.then(result => console.log(result));
```

Explanation:

- Uses `setTimeout` to simulate async operation.
- `.then()` executes after promise resolves.

Output (after 2 seconds):

```
Done after 2 seconds
```

3. **Promise Reject**

```
const failPromise = new Promise((resolve, reject) => {
  reject("Something went wrong");
});
```

```
failPromise.catch(error => console.log(error));
```
Explanation:

- `.catch()` handles rejected promises.

Output:

```
Something went wrong
```

4. **Simple Chaining**

```
const add = (num) => Promise.resolve(num + 5);

add(10)
  .then(result => result * 2)
  .then(final => console.log(final));
```
Explanation:

- First promise adds 5 → returns 15
- Next `.then()` multiplies by 2 → 30

Output:

```
30
```

5. **Chaining with Multiple Steps**

```
Promise.resolve(2)
  .then(n => n + 3)
  .then(n => n * 5)
  .then(n => `Result: ${n}`)
  .then(console.log);
```
Output:

```
Result: 25
```

Medium Examples (5)

Scenario: Simulate fetching user data and orders.

1. **Fetching User Data**

```
const fetchUser = (id) => new Promise((resolve, reject) => {
  setTimeout(() => {
    if(id > 0) resolve({ id, name: "Alice" });
    else reject("Invalid User ID");
  }, 1000);
```

```
});

fetchUser(1).then(user => console.log(user));
```
Output (after 1s):

```
{ id: 1, name: 'Alice' }
```

2. **Chaining Fetch User → Fetch Orders**

```
const fetchOrders = (userId) => new Promise(resolve => {
  setTimeout(() => resolve([101, 102, 103]), 1000);
});

fetchUser(1)
  .then(user => {
    console.log("User:", user.name);
    return fetchOrders(user.id);
  })
  .then(orders => console.log("Orders:", orders));
```
Output:

```
User: Alice
Orders: [101, 102, 103]
```

3. **Error Handling in Chain**

```
fetchUser(-1)
  .then(user => fetchOrders(user.id))
  .catch(error => console.log("Error:", error));
```
Output:

```
Error: Invalid User ID
```

4. **Multiple Dependent Promises**

```
Promise.resolve(5)
  .then(n => n * 2)
  .then(n => n - 3)
  .then(n => `Final Value: ${n}`)
  .then(console.log);
```
Output:

```
Final Value: 7
```

5. **Nested Promises**

```
fetchUser(1)
  .then(user => {
```

```
    return fetchOrders(user.id).then(orders => ({ user,
orders }));
  })
  .then(result => console.log(result));
```
Output:

```
{ user: { id: 1, name: 'Alice' }, orders: [101, 102, 103] }
```

B) Error Handling in Async Code

Easy Examples (5)

Scenario: Division operation with validation.

1. **Promise Reject on Error**

```
const divide = (a, b) => new Promise((resolve, reject) => {
  if(b === 0) reject("Cannot divide by zero");
  else resolve(a / b);
});

divide(10, 0).catch(console.log);
```
Output:

```
Cannot divide by zero
```

2. **Try/Catch in Promise**

```
divide(20, 4)
  .then(result => console.log("Result:", result))
  .catch(err => console.log("Error:", err));
```
Output:

```
Result: 5
```

3. **Error Propagation**

```
Promise.resolve(10)
  .then(n => {
    if(n > 5) throw "Number too large";
    return n;
  })
  .catch(console.log);
```
Output:

```
Number too large
```

4. **Chained Errors**

```
Promise.resolve(3)
  .then(n => n * 2)
  .then(n => { throw "Something wrong!"; })
  .then(console.log)
  .catch(err => console.log("Caught:", err));
```

Output:

```
Caught: Something wrong!
```

5. **Multiple Catch Handling**

```
Promise.reject("Fail 1")
  .catch(err => {
    console.log(err);
    return Promise.reject("Fail 2");
  })
  .catch(console.log);
```

Output:

```
Fail 1
Fail 2
```

Medium Examples (5)

Scenario: Fetch user data and simulate errors.

1. **Invalid User ID**

```
fetchUser(-10)
  .then(user => fetchOrders(user.id))
  .catch(err => console.log("Error:", err));
```

Output:

```
Error: Invalid User ID
```

2. **Chain with Multiple Errors**

```
fetchUser(1)
  .then(user => fetchOrders(-1)) // invalid call
  .catch(err => console.log("Error in chain:", err));
```

Output:

```
Error in chain: Invalid Order ID (assuming validation in
fetchOrders)
```

3. **Promise.all with Error**

```
const p1 = Promise.resolve("Success 1");
const p2 = Promise.reject("Failed 2");

Promise.all([p1, p2])
  .then(console.log)
  .catch(err => console.log("Error:", err));
```

Output:

```
Error: Failed 2
```

4. **Error Logging in Nested Promises**

```
fetchUser(1)
  .then(user => {
    return fetchOrders(user.id)
      .then(orders => orders[5].toString()) // undefined
index
      .catch(e => "Handled missing order");
  })
  .then(console.log);
```

Output:

```
Handled missing orderChained Async Operations with Error

Promise.resolve(5)
  .then(n => n * 2)
  .then(n => { if(n > 8) throw "Too big"; return n; })
  .catch(console.log);
```

Output:

```
Too big
```

C) Using `async` / `await`

Easy Examples (5)

Scenario: Basic async operations.

1. **Simple Async Function**

```
async function greet() {
  return "Hello World";
}

greet().then(console.log);
```

Output:

```
Hello World
```

2. **Awaiting Promise**

```
async function fetchData() {
  const data = await Promise.resolve("Data loaded");
  console.log(data);
}

fetchData();
```

Output:

```
Data loaded
```

3. **Async Function with Error Handling**

```
async function riskyOperation() {
  try {
    const result = await Promise.reject("Failed operation");
    console.log(result);
  } catch(err) {
    console.log("Caught Error:", err);
  }
}

riskyOperation();
```

Output:

```
Caught Error: Failed operation
```

4. **Sequential Async Calls**

```
async function calc() {
  const a = await Promise.resolve(5);
  const b = await Promise.resolve(a * 2);
  console.log(b);
}

calc();
```

Output:

```
10
```

5. **Returning Values**

```
async function multiply() {
  return 6 * 7;
}
```

```
multiply().then(console.log);
```
Output:

```
42
```

Medium Examples (5)

Scenario: Fetching users and orders using async/await.

1. **Fetch User**

```
async function getUser(id) {
  if(id <= 0) throw "Invalid ID";
  return { id, name: "Alice" };
}

getUser(1).then(console.log);
```
Output:

```
{ id: 1, name: 'Alice' }
```

2. **Await Chained Async Operations**

```
async function getOrders() {
  const user = await fetchUser(1);
  const orders = await fetchOrders(user.id);
  console.log("User:", user.name, "Orders:", orders);
}
```
Output:

```
User: Alice Orders: [101,102,103]
```

3. **Error Handling in Async/Await**

```
async function testError() {
  try {
    const user = await fetchUser(-1);
    console.log(user);
  } catch(err) {
    console.log("Error:", err);
  }
}
```
Output:

```
Error: Invalid User ID
```

4. **Sequential Async Calculations**

```
async function calcAsync() {
  let n = await Promise.resolve(10);
  n = await Promise.resolve(n * 3);
  console.log(n);
}

calcAsync();
```
Output:

```
30
```

5. **Async Function Returning Object**

```
async function getSummary() {
  const user = await fetchUser(1);
  const orders = await fetchOrders(user.id);
  return { user, orders };
}

getSummary().then(console.log);
```
Output:

```
{ user: { id:1, name:'Alice' }, orders: [101,102,103] }
```

Modules (ES Modules)

A) Import/Export Syntax (ES Modules)

Easy Examples (5)

Scenario: Exporting and importing simple functions or variables.

1. **Exporting a Variable**

- **File:** `constants.js`
- **Content:**

```
export const PI = 3.14;
```

- **File:** `main.js`

```
import { PI } from './constants.js';
console.log("Value of PI:", PI);
```

Output:

```
Value of PI: 3.14
```

Explanation: Named export and import using `{}`.

2. **Exporting a Function**

- **File:** `utils.js`

```
export function greet(name) {
  return `Hello, ${name}!`;
}
```

- **File:** `main.js`

```
import { greet } from './utils.js';
console.log(greet("Alice"));
```

Output:

```
Hello, Alice!
```

Explanation: Functions can be exported and imported like variables.

3. **Default Export**

- **File:** `math.js`

```
export default function add(a, b) {
  return a + b;
}
```

- **File:** `main.js`

```
import add from './math.js';
console.log(add(5, 10));
```

Output:

```
15
```

Explanation: Default export can be imported without `{}` and any name.

4. **Export Multiple Items**

- **File:** `data.js`

```
export const name = "John";
export const age = 25;
```

- **File:** `main.js`

```
import { name, age } from './data.js';
console.log(name, age);
```

Output:

```
John 25
```

Explanation: Named exports allow multiple variables from the same module.

5. **Combining Default and Named Exports**

- **File:** `person.js`

```
export default function info() { return "Person Info"; }
export const city = "London";
```

- **File:** `main.js`

```
import info, { city } from './person.js';
console.log(info());
console.log(city);
```

Output:

```
Person Info
London
```

Explanation: Can mix default and named exports in one module.

Medium Examples (5)

Scenario: Calculator module with multiple functions.

1. **Module with Multiple Functions**

- **File:** `calculator.js`

```
export function add(a,b){ return a+b; }
export function subtract(a,b){ return a-b; }
export function multiply(a,b){ return a*b; }
```

- **File:** `main.js`

```
import { add, multiply } from './calculator.js';
console.log(add(10,5)); // 15
console.log(multiply(10,5)); // 50
```
Explanation: Only import the functions you need.

2. **Renaming Imports**

```
import { add as sum } from './calculator.js';
console.log(sum(8,2)); // 10
```
Explanation: Avoid name conflicts using `as`.

3. **Import All**

```
import * as calc from './calculator.js';
console.log(calc.subtract(10,3)); // 7
console.log(calc.multiply(4,5)); // 20
```
Explanation: `* as` imports everything as an object.

4. **Default Export Function**

```
// defaultExport.js
export default function greet(name){ return `Hi ${name}`; }

// main.js
import greetUser from './defaultExport.js';
console.log(greetUser("Alice"));
```
Output:

```
Hi Alice
```
Explanation: Default export allows flexible import naming.

5. **Re-exporting Modules**

```
// mathOps.js
export { add, subtract } from './calculator.js';

// main.js
import { add } from './mathOps.js';
console.log(add(3,4)); // 7
```

Explanation: Modules can re-export items from other modules for better organization.

Organizing Code into Modules - Complete Implementation

Project Structure

```
project/
  ├─ modules/
  │    ├─ constants.js
  │    ├─ utils.js
  │    ├─ math.js
  │    └─ person.js
  └─ main.js
```

1. constants.js

```
// constants.js
export const TAX = 5;
export const DISCOUNT = 10;
```

Explanation:

- `TAX` and `DISCOUNT` are **constants** used throughout the project.
- **Named exports** allow us to import only what we need.

2. utils.js

```
// utils.js
export function formatName(name) {
  return name.toUpperCase();
}

export function greetUser(name) {
  return `Hello, ${name}! Welcome to the system.`;
}
```

Explanation:

- Utility functions handle **reusable logic** like formatting or greetings.

- Named exports allow selective import in `main.js`.

3. math.js

```
// math.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

export function calculatePrice(price, tax) {
  return price + (price * tax) / 100;
}
```

Explanation:

- Math operations are grouped in a separate module.
- `calculatePrice` applies tax to a base price.

4. person.js

```
// person.js
export default function getInfo(name, age) {
  return `Name: ${name}, Age: ${age}`;
}

export const city = "New York";
```

Explanation:

- Default export is a function `getInfo`.
- Named export `city` can be imported separately.
- Mix of default + named exports demonstrates **flexibility**.

5. main.js

```
// main.js
```

```
// Import constants
import { TAX, DISCOUNT } from './modules/constants.js';

// Import utility functions
import { formatName, greetUser } from './modules/utils.js';

// Import math functions
import { add, calculatePrice } from './modules/math.js';

// Import default and named from person.js
import getInfo, { city } from './modules/person.js';

// Usage
const name = "Alice";
const age = 25;
const basePrice = 1000;

// Use utils
console.log(greetUser(formatName(name)));

// Use math functions
const totalPrice = calculatePrice(basePrice, TAX);
console.log(`Base Price: ${basePrice}, Price after TAX: $
{totalPrice}`);

// Use constants
console.log(`Discount Applied: ${DISCOUNT}%`);

// Use default and named export
console.log(getInfo(name, age));
console.log("City:", city);

// Use add function
console.log("10 + 20 =", add(10, 20));
```

Explanation of main.js

1. **Importing Constants**
    - `TAX` and `DISCOUNT` are imported for price calculation.
2. **Utility Functions**
    - `formatName()` converts name to uppercase.
    - `greetUser()` uses formatted name to print a greeting.

3. **Math Functions**
    - `calculatePrice()` computes price including tax.
    - `add()` demonstrates general math operations.
4. **Person Module**
    - `getInfo()` (default) prints user info.
    - `city` (named) is imported separately.
5. **Combining Everything**
    - Demonstrates how **modules organize related code**.
    - Keeps `main.js` clean and readable.

Sample Output

```
Hello, ALICE! Welcome to the system.
Base Price: 1000, Price after TAX: 1050
Discount Applied: 10%
Name: Alice, Age: 25
City: New York
10 + 20 = 30
```

Medium Examples (5)

E-Commerce System Modules - Complete Implementation

Project Structure

```
ecommerce/
  ├─ modules/
  │    ├─ products.js
  │    ├─ cart.js
  │    ├─ config.js
  │    └─ index.js
  └─ main.js
```

1. products.js

```
// modules/products.js

export const products = [
  { id: 1, name: "Laptop", price: 1000 },
  { id: 2, name: "Smartphone", price: 500 },
  { id: 3, name: "Headphones", price: 100 }
];
```

Explanation:

- Exports an array of products.
- Named export allows selective import.

2. cart.js

```
// modules/cart.js

let cartItems = [];

export function addToCart(product) {
  cartItems.push(product);
  console.log(`${product.name} added to cart.`);
}

export function removeFromCart(productId) {
  cartItems = cartItems.filter(p => p.id !== productId);
  console.log(`Product with ID ${productId} removed from
cart.`);
}

export function viewCart() {
  console.log("Current Cart:", cartItems);
}
```

Explanation:

- Functions for **adding, removing, and viewing cart items**.
- Only `addToCart` and `viewCart` may be imported in `main.js`.

3. config.js

```
// modules/config.js

export const API_ENDPOINT = "https://api.ecommerce.com/v1";
export const ENV = "development"; // could be "production"
```

Explanation:

- Centralized configuration module.
- Can be imported dynamically based on environment.

4. index.js (Re-export Modules)

```
// modules/index.js

export { products } from './products.js';
export { addToCart, removeFromCart, viewCart } from './
cart.js';
export { API_ENDPOINT, ENV } from './config.js';
```

Explanation:

- **Re-exporting** modules allows importing everything from a single file.
- Makes code cleaner and easier to manage.

5. main.js

```
// main.js

// Import everything from index.js
import { products, addToCart, viewCart, API_ENDPOINT, ENV }
from './modules/index.js';

// Display all products
console.log("Available Products:");
products.forEach(p => console.log(`${p.name} - $${p.price}
`));

// Add products to cart
addToCart(products[0]); // Laptop
```

```
addToCart(products[2]); // Headphones

// View cart
viewCart();

// Conditional/Dynamic import based on environment
if (ENV === "development") {
  import('./modules/config.js').then(config => {
    console.log("Using API Endpoint (Dev):",
config.API_ENDPOINT);
  });
} else {
  import('./modules/config.js').then(config => {
    console.log("Using API Endpoint (Prod):",
config.API_ENDPOINT);
  });
}
```

Explanation of main.js

1. **Import from index.js**
   - Products, cart functions, and config values imported **from a single module**.
2. **Display Products**
   - Iterates over the `products` array to display them in the console.
3. **Cart Operations**
   - `addToCart()` adds selected products.
   - `viewCart()` displays current cart items.
4. **Dynamic/Conditional Import**
   - Loads API endpoint dynamically based on environment (`development` or `production`).
   - Demonstrates **lazy loading** modules.

Sample Output

```
Available Products:
Laptop - $1000
Smartphone - $500
```

```
Headphones - $100
Laptop added to cart.
Headphones added to cart.
Current Cart: [ { id: 1, name: 'Laptop', price: 1000 },
  { id: 3, name: 'Headphones', price: 100 } ]
Using API Endpoint (Dev): https://api.ecommerce.com/v1
```

C) Dynamic Imports

Easy Examples (5)

Scenario: Load modules only when needed.

1. **Basic Dynamic Import**

```
import('./utils.js').then(module => {
  console.log(module.formatName("Alice"));
});
```

Output:

```
ALICE
```

Explanation: Load module at runtime instead of startup.

2. **Inside Function**

```
async function loadUtils(){
  const utils = await import('./utils.js');
  console.log(utils.formatName("Bob"));
}
loadUtils();
```

3. **Conditional Import**

```
if(condition){
  import('./moduleA.js').then(mod => console.log(mod.info));
}
```

Explanation: Only load module if condition is true.

4. **Error Handling with Dynamic Import**

```
import('./nonExistent.js')
  .then(console.log)
  .catch(err => console.log("Module not found:", err));
```

Output:

```
Module not found: ...
```

5. **Dynamic Import in Event**

```
document.getElementById('btn').addEventListener('click',
async () => {
  const module = await import('./utils.js');
  console.log(module.formatName("Charlie"));
});
```

Explanation: Module loaded only when button clicked.

Medium Examples (5)

Scenario: Large application loading modules on demand.

1. **Load Payment Module**

```
async function processPayment() {
  const payment = await import('./payment.js');
  payment.pay(100);
}
```

2. **Feature Flags**

```
if(isAdmin){
  const adminModule = await import('./admin.js');
  adminModule.init();
}
```

3. **Lazy Load Charts**

```
document.getElementById('chartBtn').addEventListener('click',
async () => {
  const charts = await import('./charts.js');
  charts.renderChart();
});
```

4. **Dynamic Config Based on User Role**

```
const role = "manager";
const module = await import(`./modules/${role}.js`);
module.init();
```

5. **Fallback Module**

```
try {
  const mod = await import('./optionalModule.js');
  mod.run();
} catch {
  console.log("Module not available, using default");
}
```

1️⃣ Node.js Architecture: Event Loop, Non-Blocking I/O, Process Model

Easy Examples (5)

Scenario: Understanding asynchronous behavior with setTimeout and process.nextTick.

```
// eventLoop.js
console.log("Start");

setTimeout(() => {
  console.log("Inside setTimeout");
}, 0);

process.nextTick(() => {
  console.log("Inside nextTick");
});

console.log("End");
```

Explanation:

- Node.js is **single-threaded**, uses **event loop** for async operations.
- `process.nextTick` executes **before the next event loop iteration**.

Output:

```
Start
End
Inside nextTick
Inside setTimeout
```

Scenario: Non-blocking I/O

```
const fs = require('fs');

console.log("Start reading file");
```

```
fs.readFile('sample.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log("File content:", data);
});

console.log("After readFile call");
```
Explanation:

- File read is **non-blocking**, Node.js continues execution without waiting for I/O.

Output (if sample.txt contains "Hello Node"):

```
Start reading file
After readFile call
File content: Hello Node
```

Medium Examples (5)

Scenario: Event loop with CPU-intensive task

```
console.log("Start");

setTimeout(() => console.log("Timeout 1"), 0);
setTimeout(() => console.log("Timeout 2"), 0);

for(let i=0; i<1e9; i++) {} // CPU-intensive

console.log("End of script");
```
Explanation:

- Long-running synchronous loop **blocks event loop**, delaying setTimeout callbacks.
- Demonstrates importance of **non-blocking I/O**.

Output:

```
Start
End of script
Timeout 1
Timeout 2
```

Scenario: Process model - using `process` object

```
console.log("Process ID:", process.pid);
console.log("Node version:", process.version);
console.log("Platform:", process.platform);
```
Explanation:

- `process` object gives info about **current Node.js process**.

Output Example:

```
Process ID: 12345
Node version: v20.5.0
Platform: win32
```

Scenario: Multiple async callbacks

```
setTimeout(()=> console.log("Timeout"), 0);
setImmediate(()=> console.log("Immediate"));
process.nextTick(()=> console.log("NextTick"));
```

Explanation:

- `nextTick` → `setImmediate` → `setTimeout` execution order in **event loop**.

Output:

```
NextTick
Immediate
Timeout
```

CommonJS vs ES Modules, Module Resolution, Custom Modules

Easy Examples (5)

CommonJS Example (require):

```
// math.js
module.exports.add = (a, b) => a + b;

// main.js
const math = require('./math.js');
console.log(math.add(5, 10));
```

Output:

```
15
```

ES Module Example (import/export):

```
// utils.mjs
export const greet = (name) => `Hello ${name}`;

// main.mjs
import { greet } from './utils.mjs';
console.log(greet("Alice"));
```

Output:

```
Hello Alice
```
Scenario: Custom module

```
// logger.js
exports.log = (msg) => console.log(`[LOG]: ${msg}`);

// main.js
const logger = require('./logger');
logger.log("Server started");
```
Output:

```
[LOG]: Server started
```

Medium Examples (5)

Scenario: Module resolution & nested modules

```
// modules/math/add.js
module.exports = (a,b) => a+b;

// modules/math/index.js
const add = require('./add');
module.exports = { add };

// main.js
const math = require('./modules/math');
console.log(math.add(10,20));
```
Output:

```
30
```
Scenario: Using ES modules dynamically

```
const moduleName = './utils.mjs';
import(moduleName).then((mod) => {
  console.log(mod.greet("Bob"));
});
```

Standard Library: fs, http, path, os, crypto

Easy Examples (5)

1. fs - Read file async

```
const fs = require('fs');
fs.readFile('sample.txt','utf8',(err,data)=>{
  if(err) console.error(err);
```

```
  else console.log(data);
});
```
Output:

```
Hello Node
```
2. http - Simple server

```
const http = require('http');
const server = http.createServer((req,res)=>{
  res.end("Hello from Node server");
});
server.listen(3000, ()=> console.log("Server running"));
```
Output:

```
Server running
```
Visit `http://localhost:3000` → shows `Hello from Node server`.

3. path - Join paths

```
const path = require('path');
console.log(path.join('folder','file.txt'));
```
Output:

```
folder/file.txt
```
4. os - System info

```
const os = require('os');
console.log("CPU Cores:", os.cpus().length);
console.log("Free Memory:", os.freemem());
```
Output:

```
CPU Cores: 8
Free Memory: 2147483648
```
5. crypto - Hash a string

```
const crypto = require('crypto');
const hash =
crypto.createHash('sha256').update('Hello').digest('hex');
console.log(hash);
```
Output:

```
185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381
969
```

Medium Examples (5)

1. fs - Stream file

```
const fs = require('fs');
const readStream = fs.createReadStream('sample.txt','utf8');
readStream.on('data', chunk => console.log("Chunk:", chunk));
```
Explanation: Read file in chunks → **efficient for large files**.

2. http - JSON API

```
const http = require('http');
const server = http.createServer((req,res)=>{
  res.writeHead(200,{'Content-Type':'application/json'});
  res.end(JSON.stringify({msg:"Hello JSON"}));
});
server.listen(3001);
```
Visit `http://localhost:3001` → returns JSON: `{"msg":"Hello JSON"}`

3. path - Resolve absolute path

```
const path = require('path');
console.log(path.resolve('folder','file.txt'));
```
Output:

```
C:\Users\You\project\folder\file.txt
```

4. os - Network interfaces

```
const os = require('os');
console.log(os.networkInterfaces());
```
Output: Shows all network interface details.

5. crypto - Encrypt/Decrypt

```
const crypto = require('crypto');
const algorithm = 'aes-256-cbc';
const key = crypto.randomBytes(32);
const iv = crypto.randomBytes(16);

const cipher = crypto.createCipheriv(algorithm,key,iv);
let encrypted = cipher.update('Hello Node','utf8','hex');
encrypted += cipher.final('hex');
console.log("Encrypted:", encrypted);
```

4️⃣ Debugging and Profiling Node.js Applications

Easy Examples (5)

Scenario: Use `console.log` for basic debugging

```
const num = 10;
console.log("Number is", num);
```
Output:

```
Number is 10
```

Scenario: Using `node --inspect`

```
node --inspect main.js
```
- Opens Chrome DevTools → allows breakpoints and step debugging.

Scenario: Using `process` object

```
console.log("Memory Usage:", process.memoryUsage());
console.log("Uptime:", process.uptime());
```

Medium Examples (5)

Scenario: Profiling CPU usage

```
console.time("loop");
for(let i=0;i<1e7;i++){}
console.timeEnd("loop");
```
Output:

```
loop: 45.678ms
```
Scenario: Monitoring event loop lag

```
setInterval(()=>{
  console.log("Event loop running", Date.now());
},1000);
```