1. Variables, Operators, Expressions

Example 1: Swap two variables

```
let a = 5;
let b = 10;

// Swapping using a temporary variable
let temp = a;
a = b;
b = temp;

console.log("a:", a, "b:", b);
```

Explanation: We use a temporary variable to hold one value while swapping.
Output:

```
a: 10 b: 5
```

Example 2: Calculate the area of a rectangle

```
let length = 7;
let width = 5;
let area = length * width;
console.log("Area:", area);
```

Explanation: Multiplication operator `*` is used to calculate area.
Output:

```
Area: 35
```

Example 3: Check if a number is even or odd

```
let num = 11;
let result = (num % 2 === 0) ? "Even" : "Odd";
console.log(num, "is", result);
```

Explanation: `%` is the modulus operator, returns remainder. Ternary operator selects the output.
Output:

```
11 is Odd
```

Example 4: Increment and Decrement operators

```
let x = 5;
console.log(++x); // Pre-increment
console.log(x--); // Post-decrement
console.log(x);
```

Explanation: Pre-increment increases value before using it; post-decrement uses the value first, then decreases.
Output:

```
6
6
5
```

Example 5: Calculate simple expression

```
let a = 10;
let b = 3;
let result = a + b * 2 - (a / b);
console.log(result);
```

Explanation: Follows operator precedence: multiplication/division before addition/subtraction.
Output:

```
15
```

2. Control Flow: if/else, switch, loops

Example 1: Check positive, negative, or zero

```
let num = -3;

if (num > 0) {
  console.log("Positive");
} else if (num < 0) {
  console.log("Negative");
} else {
  console.log("Zero");
}
```

Output:

```
Negative
```

Example 2: Day of the week using switch

```
let day = 3;
switch(day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Other day");
}
```

Output:

```
Wednesday
```

Example 3: Print numbers 1 to 5 using for loop

```
for(let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Output:

```
1
2
3
4
5
```

Example 4: Sum of first 10 numbers using while loop

```
let sum = 0, i = 1;
while(i <= 10) {
  sum += i;
  i++;
}
console.log("Sum:", sum);
```

Output:

```
Sum: 55
```

Example 5: Print odd numbers using do...while

```
let j = 1;
do {
  if(j % 2 !== 0) console.log(j);
  j++;
} while(j <= 10);
```

Output:

```
1
3
5
7
```

3. Functions and Scope

Example 1: Simple function to greet

```
function greet(name) {
  return "Hello, " + name + "!";
}
console.log(greet("Alice"));
```
Output:

```
Hello, Alice!
```

Example 2: Function with default parameter

```
function multiply(a, b = 2) {
  return a * b;
}
console.log(multiply(5));
```
Output:

```
10
```

Example 3: Function scope demonstration

```
function testScope() {
  let localVar = "I'm local";
  console.log(localVar);
}
testScope();
// console.log(localVar); // Error: localVar is not defined
```
Explanation: Variables declared inside a function are not accessible outside.
Output:

```
I'm local
```

Example 4: Arrow function

```
const square = (n) => n * n;
console.log(square(6));
```
Output:

```
36
```

Example 5: Nested function

```
function outer(a) {
  function inner(b) {
    return b * 2;
  }
  return inner(a) + 3;
}
console.log(outer(5));
```

Explanation: Inner function can access outer function parameter.
Output:

```
13
```

1. Variables, Operators, Expressions – Medium Scenarios

Example 1: Calculate total price with discount

Scenario: A shopping cart has items with a total amount. If total > 1000, apply 10% discount.

```
let totalAmount = 1200;
let discount = (totalAmount > 1000) ? totalAmount * 0.1 : 0;
let finalAmount = totalAmount - discount;

console.log("Final Amount:", finalAmount);
```

Explanation: Ternary operator checks if discount applies; subtraction calculates final amount.
Output:

```
Final Amount: 1080
```

Example 2: Determine grade from marks

Scenario: Assign grades based on marks: ≥90 A, ≥75 B, ≥60 C, else F.

```
let marks = 82;
let grade = (marks >= 90) ? "A" :
            (marks >= 75) ? "B" :
            (marks >= 60) ? "C" : "F";
console.log("Grade:", grade);
```

Output:

```
Grade: B
```

Example 3: Calculate age group

Scenario: Categorize age into “Child”, “Teen”, “Adult”, “Senior”.

```
let age = 25;
let category = (age < 13) ? "Child" :
               (age < 20) ? "Teen" :
               (age < 60) ? "Adult" : "Senior";
console.log("Category:", category);
```
Output:

```
Category: Adult
```

Example 4: Currency converter

Scenario: Convert USD to EUR, INR, or GBP based on user selection.

```
let amountUSD = 100;
let currency = "INR";
let converted = (currency === "EUR") ? amountUSD * 0.9 :
                (currency === "INR") ? amountUSD * 83 :
                (currency === "GBP") ? amountUSD * 0.78 :
amountUSD;

console.log(`${amountUSD} USD = ${converted} ${currency}`);
```
Output:

```
100 USD = 8300 INR
```

Example 5: Check eligibility for a loan

Scenario: Loan approved if age ≥21 and salary ≥25000.

```
let age = 23;
let salary = 30000;
let eligible = (age >= 21 && salary >= 25000);
console.log("Loan eligible?", eligible ? "Yes" : "No");
```
Output:

```
Loan eligible? Yes
```

2. Control Flow – Medium Scenarios

Example 1: Traffic light system

Scenario: Display action based on traffic light color.

```
let light = "yellow";
switch(light) {
    case "red":
        console.log("Stop");
        break;
    case "yellow":
        console.log("Get Ready");
        break;
    case "green":
        console.log("Go");
        break;
    default:
        console.log("Invalid color");
}
```

Output:

```
Get Ready
```

Example 2: Find the largest of three numbers

Scenario: Determine the largest number among three inputs.

```
let a = 10, b = 25, c = 15;
let largest;

if(a > b && a > c) largest = a;
else if(b > a && b > c) largest = b;
else largest = c;

console.log("Largest:", largest);
```

Output:

```
Largest: 25
```

Example 3: Print multiplication table

Scenario: Display table of 7 using a loop.

```
let num = 7;
for(let i = 1; i <= 10; i++) {
  console.log(`${num} x ${i} = ${num*i}`);
}
```

Output:

```
7 x 1 = 7
7 x 2 = 14
...
7 x 10 = 70
```

Example 4: Sum of even numbers from array

Scenario: Sum only even numbers in the given array.

```
let numbers = [5, 12, 7, 8, 20, 3];
let sumEven = 0;

for(let n of numbers){
    if(n % 2 === 0) sumEven += n;
}
console.log("Sum of even numbers:", sumEven);
```

Output:

```
Sum of even numbers: 40
```

Example 5: FizzBuzz problem

Scenario: For numbers 1–15, print “Fizz” if divisible by 3, “Buzz” if divisible by 5, “FizzBuzz” if both.

```
for(let i = 1; i <= 15; i++) {
    if(i % 3 === 0 && i % 5 === 0) console.log("FizzBuzz");
    else if(i % 3 === 0) console.log("Fizz");
    else if(i % 5 === 0) console.log("Buzz");
    else console.log(i);
}
```

Output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

3. Functions and Scope – Medium Scenarios

Example 1: Calculate factorial using function

```
function factorial(n){
    let result = 1;
    for(let i = 2; i <= n; i++){
        result *= i;
    }
    return result;
}
console.log(factorial(5));
```
Output:

```
120
```

Example 2: Check palindrome

```
function isPalindrome(str){
    let reversed = str.split('').reverse().join('');
    return str === reversed;
}
console.log(isPalindrome("racecar")); // true
console.log(isPalindrome("hello"));   // false
```

Example 3: Convert Celsius to Fahrenheit

```
function cToF(celsius){
    return (celsius * 9/5) + 32;
}
console.log(cToF(25));
```
Output:

```
77
```

Example 4: Count vowels in a string

```
function countVowels(text){
    let count = 0;
    for(let char of text.toLowerCase()){
        if("aeiou".includes(char)) count++;
    }
    return count;
}
console.log(countVowels("JavaScript"));
```
Output:

```
3
```

Example 5: Nested function – calculate final price with tax

```
function finalPrice(price){
    function addTax(p){
        return p * 1.1; // 10% tax
    }
    return addTax(price);
}
console.log(finalPrice(500));
```

Output:

```
550
```

Easy 1: Create an object and print properties

Scenario:

A user profile object contains name and age. Print both values.

Code

```
let user = {
  name: "Rahul",
  age: 25
};

console.log(user.name);
console.log(user.age);
```

Output

```
Rahul
25
```

Easy 2: Add new property to an object

Scenario:

A product object initially has name and price. Add a new property `category`.

Code

```
let product = {
  name: "Laptop",
  price: 50000
};

product.category = "Electronics";
```

```
console.log(product);
```

Output

```
{ name: 'Laptop', price: 50000, category: 'Electronics' }
```

Easy 3: Create an object method

Scenario:

A vehicle object must display its brand using a method.

Code

```
let vehicle = {
  brand: "Toyota",
  showBrand: function() {
    return this.brand;
  }
};

console.log(vehicle.showBrand());
```

Output

```
Toyota
```

Easy 4: Delete a property

Scenario:

Remove `discount` property from an item object.

Code

```
let item = {
  name: "Shoes",
  price: 2000,
  discount: 10
};

delete item.discount;

console.log(item);
```

Output

```
{ name: 'Shoes', price: 2000 }
```

Easy 5: Access nested object

Scenario:

A student object contains an address object inside it. Print the city.

Code

```
let student = {
  name: "Arun",
  address: {
    city: "Chennai",
    pincode: 600001
  }
};

console.log(student.address.city);
```

Output

```
Chennai
```

Medium 1: Create an object with method using `this` keyword

Scenario:

A bank account object contains name and balance. Write a method to display a message: **"Customer <name> has balance <balance>."**

Code

```
let account = {
  name: "Vikram",
  balance: 15000,
  showDetails: function () {
    return `Customer ${this.name} has balance $
{this.balance}.`;
  }
};

console.log(account.showDetails());
```

Output

```
Customer Vikram has balance 15000.
```

Medium 2: Update object properties based on condition

Scenario:

If a product's price is above 1000, apply a 10% discount and update price.

Code

```
let product = {
  name: "Watch",
  price: 1500
};

if(product.price > 1000) {
  product.price = product.price - (product.price * 0.1);
}

console.log(product.price);
```

Output

```
1350
```

Medium 3: Constructor function + prototype method

Scenario:

Create a constructor `Employee()` and add a prototype method `getInfo()`.

Code

```
function Employee(name, dept) {
  this.name = name;
  this.dept = dept;
}

Employee.prototype.getInfo = function() {
  return `${this.name} works in ${this.dept}`;
};

let emp1 = new Employee("Meena", "HR");
console.log(emp1.getInfo());
```

Output

```
Meena works in HR
```

Medium 4: Inheritance using prototypes

Scenario:

Create an Animal object and inherit its property inside Dog object.

Code

```
function Animal(type) {
  this.type = type;
}

Animal.prototype.sayType = function() {
  return `This is a ${this.type}`;
};

function Dog(name) {
  this.name = name;
}

Dog.prototype = Object.create(Animal.prototype);

let dog1 = new Dog("Tommy");
dog1.type = "Pet Dog";

console.log(dog1.sayType());
```

Output

```
This is a Pet Dog
```

Medium 5: Method inside object modifying another property

Scenario:

A cart object has quantity and price. Add a method to calculate total amount.

Code

```
let cart = {
  quantity: 3,
  pricePerItem: 500,
  totalAmount: function() {
    return this.quantity * this.pricePerItem;
  }
};

console.log(cart.totalAmount());
```
Output

```
1500
```

Hard 1: Deep vs Shallow Copy Scenario

Scenario:

A company's employee object has nested properties (address).
You must clone the object, update the city, and compare results.

Code

```
let employee = {
  name: "Anitha",
  department: "Finance",
  address: {
    city: "Chennai",
    pincode: 600020
  }
};

// ❌ Shallow copy
let shallowCopy = Object.assign({}, employee);

// ✔ Deep copy
let deepCopy = JSON.parse(JSON.stringify(employee));

// Modify nested value
shallowCopy.address.city = "Bangalore";
deepCopy.address.city = "Hyderabad";

console.log("Original:", employee.address.city);
console.log("Shallow Copy:", shallowCopy.address.city);
console.log("Deep Copy:", deepCopy.address.city);
```
Explanation

- Shallow copy shares nested object → Changing it affects original.
- Deep copy creates a new independent address object.

Output

```
Original: Bangalore
Shallow Copy: Bangalore
Deep Copy: Hyderabad
```

Using Object.freeze() to prevent modification

Scenario:

A banking system freezes an account object to prevent tampering.

Code

```
let account = {
  holder: "Ravi",
  balance: 50000
};

Object.freeze(account);

// Attempt modifications
account.balance = 60000;
account.branch = "Chennai";

console.log(account);
```

Explanation

- `Object.freeze()` → No property can be added, modified, or deleted.

Output

```
{ holder: 'Ravi', balance: 50000 }
```

Class Inheritance + Method Overriding

Scenario:

A company has different employee types.
FullTimeEmployee should override calculateSalary() method.

Code

```
class Employee {
  constructor(name, baseSalary) {
    this.name = name;
    this.baseSalary = baseSalary;
  }

  calculateSalary() {
    return this.baseSalary;
  }
}

class FullTimeEmployee extends Employee {
  constructor(name, baseSalary, bonus) {
    super(name, baseSalary);
    this.bonus = bonus;
  }

  calculateSalary() {
    return this.baseSalary + this.bonus;
  }
}

let emp = new FullTimeEmployee("Karan", 40000, 10000);
console.log(emp.calculateSalary());
```

Explanation

- Child class overrides salary calculation logic using `extends` and `super()`.

Output

```
50000
```

Prototype Chaining – Multi-level Inheritance

Scenario:

Create a 3-level inheritance structure:
Vehicle → Car → ElectricCar

Code

```
function Vehicle(type) {
  this.type = type;
}

Vehicle.prototype.getType = function() {
  return this.type;
};

function Car(brand) {
  this.brand = brand;
}
Car.prototype = Object.create(Vehicle.prototype);

function ElectricCar(name, battery) {
  this.name = name;
  this.battery = battery;
}
ElectricCar.prototype = Object.create(Car.prototype);

let tesla = new ElectricCar("Model S", "100 kWh");

// Add Vehicle prototype property
tesla.type = "Electric";

console.log(tesla.getType());
```

Explanation

- `Vehicle → Car → ElectricCar` prototype chain built manually.
- ElectricCar inherits methods from Vehicle.

Output

```
Electric
```

Using this in a dynamically bound context

Scenario:

A hotel booking system reuses a function to calculate final price across different room types using `call()`.

Code

```
let deluxeRoom = {
  price: 3000,
  tax: 0.18
};

let suiteRoom = {
  price: 8000,
  tax: 0.18
};

function finalPrice() {
  return this.price + (this.price * this.tax);
}

console.log(finalPrice.call(deluxeRoom));
console.log(finalPrice.call(suiteRoom));
```

Explanation

- `call()` binds `this` explicitly.
- Same function is reused for multiple objects → optimizes memory.

Output

```
3540
9440
```

Primitive Types

(string, number, boolean, null, undefined, symbol, bigint)

Easy 1: Identify data types

Scenario

Check the datatypes of different primitive values.

```
console.log(typeof "Hello");
console.log(typeof 42);
console.log(typeof true);
console.log(typeof undefined);
console.log(typeof Symbol("id"));
```

Output

```
string
number
boolean
```

```
undefined
symbol
```

Easy 2: Create a BigInt number

Scenario

Store a large integer using BigInt.

```
let bigNumber = 9007199254740999n;
console.log(bigNumber);
console.log(typeof bigNumber);
```

Output

```
9007199254740999n
bigint
```

Easy 3: Assign null value

Scenario

Clear a variable by setting it to null.

```
let user = "Karan";
user = null;
console.log(user);
console.log(typeof user);
```

Output

```
null
object
```

Easy 4: Boolean value check

Scenario

Check if a number is greater than 100.

```
let price = 150;
let isHigh = price > 100;
console.log(isHigh);
```

Ouput

```
true
```

Easy 5: Use Symbol as unique key

Scenario

Create two different symbols with same description.

```
let s1 = Symbol("id");
let s2 = Symbol("id");

console.log(s1 === s2);
```

Output

```
false
```

Medium 1: Validate user input type

Scenario

Check if age is a valid number.

```
let age = "25";

if (typeof age === "number") {
    console.log("Valid number");
} else {
    console.log("Invalid type");
}
```

Output

```
Invalid type
```

Medium 2: Calculate total using BigInt

Scenario

Add two large BigInt values.

```
let v1 = 12345678901234567890n;
let v2 = 98765432109876543210n;

let total = v1 + v2;
console.log(total);
```

Output

```
11111111011111111100n
```

Medium 3: Null vs Undefined

Scenario

Check if a variable is intentionally empty or not initialized.

```
let x;
let y = null;

console.log(x === undefined);
console.log(y === null);
```

Output

```
true
true
```

Medium 4: Boolean conversion in condition

Scenario

Check if discount is applied.

```
let discount = 0;
if (!discount) {
    console.log("No discount applied");
}
```

Output

```
No discount applied
```

Medium 5: Symbol as object key

Scenario

Use symbol to create hidden property.

```
let secretKey = Symbol("secret");

let user = {
  name: "Arun",
  [secretKey]: "hiddenValue"
};
```

```
console.log(user.name);
console.log(user[secretKey]);
```
Output

```
Arun
hiddenValue
```

Reference Types

(objects, arrays, functions, arrow functions)

Easy 1: Create an object

```
let person = { name: "Meena", age: 22 };
console.log(person.name);
```
Output

```
Meena
```

Easy 2: Create an array and access element

```
let fruits = ["Apple", "Banana", "Mango"];
console.log(fruits[1]);
```
Output

```
Banana
```

Easy 3: Simple function

```
function greet() {
  return "Hello!";
}
console.log(greet());
```
Output

```
Hello!
```

Easy 4: Simple arrow function

```
const add = (a, b) => a + b;
console.log(add(5, 3));
```
Output

Easy 5: Array push

```
let nums = [1, 2, 3];
nums.push(4);
console.log(nums);
```

Output

```
[1, 2, 3, 4]
```

Medium 1: Add method to object

Scenario

Customer object must return full details.

```
let customer = {
  name: "Ravi",
  age: 30,
  details() {
    return `${this.name}, Age: ${this.age}`;
  }
};

console.log(customer.details());
```

Output

```
Ravi, Age: 30
```

Medium 2: Array of objects

Scenario

Print employee names from array.

```
let employees = [
  { name: "John", id: 1 },
  { name: "Sara", id: 2 }
];

for (let emp of employees) {
  console.log(emp.name);
}
```
Output

```
John
Sara
```

Medium 3: Function returning another function

Scenario

Create tax calculator generator.

```
function taxCalculator(rate) {
  return function(amount) {
    return amount * rate;
  };
}

let gst = taxCalculator(0.18);
console.log(gst(1000));
```
Output

```
180
```

Medium 4: Arrow function inside object (lexical this)

Scenario

Check `this` behavior.

```
let product = {
  price: 500,
  showPrice: () => console.log(this.price)
};

product.showPrice();
```
Output

```
undefined
```
Explanation → arrow functions do NOT have their own `this`.

Medium 5: Mutate array inside function

Scenario

Add new value using function.

```
let data = [10, 20];

function append(arr, value) {
  arr.push(value);
}

append(data, 30);

console.log(data);
```

Output

```
[10, 20, 30]
```

Type Conversion & Coercion

Easy 1: Convert string to number

```
let x = "25";
console.log(Number(x));
```

Output

```
25
```

Easy 2: Convert number to string

```
let n = 100;
console.log(String(n));
```

Output

```
100
```

Easy 3: Boolean conversion

```
console.log(Boolean(""));
console.log(Boolean("Hello"));
```

Output

```
false
true
```

Easy 4: Implicit conversion (coercion)

```
console.log("10" * 2);
console.log("10" + 2);
```

Output

```
20
102
```

Easy 5: Parse float

```
let value = "45.67";
console.log(parseFloat(value));
```

Output

```
45.67
```

Medium 1: Input validation

Scenario

Convert input to number and verify.

```
let input = "250";
let amount = Number(input);

if(!isNaN(amount)) {
  console.log("Valid:", amount);
} else {
  console.log("Invalid input");
}
```

Output

```
Valid: 250
```

Medium 2: Addition vs concatenation

Scenario

Show difference between + and *.

```
let a = "5";
let b = 10;
```

```
console.log(a + b);
console.log(a * b);
```
Output

```
510
50
```

Medium 3: Convert to boolean in real-world scenario

Scenario

Check if user has uploaded a file.

```
let uploadedFile = "";

if (!uploadedFile) {
  console.log("No file uploaded");
}
```
Output

```
No file uploaded
```

Medium 4: Calculate bill using coerced values

Scenario

Prices come as strings from form input.

```
let price = "300";
let quantity = "2";

let total = price * quantity;
console.log(total);
```
Output

```
600
```

Medium 5: Date conversion to number

Scenario

Compare two timestamps.

```
let d1 = new Date("2024-01-01");
let d2 = new Date("2024-02-01");
```

```
console.log(Number(d2) > Number(d1));
```
Output

```
true
```

Closures & Lexical Scope

Easy 1 — Basic Closure

Question: What will the following code print?

```
function outer() {
  const x = 10;
  return function inner() {
    console.log(x);
  }
}

const f = outer();
f();
```
Output:

```
10
```
Explanation:
`inner()` closes over `x` from `outer()`. Even after `outer()` finishes, `x` remains accessible.

Easy 2 — Updating Closed Variable

```
function counter() {
  let c = 1;
  return function() {
    c++;
    return c;
  }
}

const count = counter();
console.log(count());
console.log(count());
```
Output:

```
2
3
```

Easy 3 — Closure Returning Multiple Times

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

const g = greet("Alex");
console.log(g("Hello"));
```

Output:

```
Hello, Alex!
```

Easy 4 — Lexical Scope

```
let a = 5;

function one() {
  let b = 10;
  function two() {
    console.log(a + b);
  }
  two();
}

one();
```

Output:

```
15
```

Easy 5 — Closure with Counter Reset

```
function makeCounter() {
  let c = 0;
  return () => ++c;
}

const x = makeCounter();
console.log(x());
console.log(x());
console.log(x());
```

Output:

```
1
```

```
2
3
```

MEDIUM

Scenario 1 — API Rate Limiter

A web app must prevent a button from being clicked more than once every 2 seconds.

```
function rateLimiter() {
  let lastClick = 0;

  return function() {
    const now = Date.now();
    if (now - lastClick > 2000) {
      lastClick = now;
      return "Action allowed";
    }
    return "Too fast!";
  }
}

const click = rateLimiter();

console.log(click());
setTimeout(() => console.log(click()), 1000);
setTimeout(() => console.log(click()), 2500);
```

Output:

```
Action allowed
Too fast!
Action allowed
```

Scenario 2 — Private Account Balance

```
function bankAccount(initial) {
  let balance = initial;

  return {
    deposit(amount) { balance += amount; return balance; },
    withdraw(amount) { balance -= amount; return balance; }
  };
}
```

```
const acc = bankAccount(1000);
console.log(acc.deposit(500));
console.log(acc.withdraw(700));
```

Output:

```
1500
800
```

Scenario 3 — Remember Last Search Query

```
function searchCache() {
  let lastQuery = null;
  return function(query) {
    if (query === lastQuery) return "Using cached result";
    lastQuery = query;
    return `Fetching for ${query}`;
  };
}

const search = searchCache();
console.log(search("laptop"));
console.log(search("laptop"));
```

Output:

```
Fetching for laptop
Using cached result
```

Scenario 4 — Counter for API Calls

```
function apiTracker() {
  let calls = 0;
  return function() {
    calls++;
    return `API calls so far: ${calls}`;
  }
}

const track = apiTracker();
console.log(track());
console.log(track());
console.log(track());
```

Output:

```
API calls so far: 1
API calls so far: 2
API calls so far: 3
```

Scenario 5 — Generate Unique IDs

```
function idGenerator() {
  let id = 100;
  return () => ++id;
}

const gen = idGenerator();
console.log(gen());
console.log(gen());
```

Output:

```
101
102
```

2. Rest & Spread Syntax

Easy 1 — Rest Function

```
function sum(...nums) {
  return nums.reduce((a,b) => a + b);
}

console.log(sum(1,2,3));
```

Output: 6

Easy 2 — Spread Array into Function

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

console.log(add(...[1,2,3]));
```

Output: 6

Easy 3 — Merge Objects

```
const a = {x:1};
const b = {y:2};

console.log({...a, ...b});
```

Output: `{x:1, y:2}`

Easy 4 — Copy Array

```
const arr = [1,2,3];
const copy = [...arr];

console.log(copy);
```
Output: `[1,2,3]`

Easy 5 — Rest with Named Params

```
function show(a, ...rest) {
  console.log(a, rest);
}

show(1,2,3,4);
```
Output:

```
1 [2,3,4]
```

MEDIUM

Scenario 1 — Dynamic Discount Function

```
function applyDiscount(discount, ...prices) {
  return prices.map(p => p - p * discount);
}

console.log(applyDiscount(0.1, 100, 200, 300));
```
Output:
```
[90, 180, 270]
```

Scenario 2 — Merge Multiple Configs

```
const base = {theme:"light"};
const user = {font:"large"};
const system = {cache:true};

console.log({...base, ...user, ...system});
```
Output:
```
{theme:"light", font:"large", cache:true}
```

Scenario 3 — Spread to Clone

```
const product = {name:"Laptop", price:45000};
const clone = {...product};
```

```
console.log(clone);
```

Scenario 4 — Filtering Unknown Inputs

```
function filterNumbers(...values) {
  return values.filter(v => typeof v === "number");
}

console.log(filterNumbers(1, "a", 3, true, 9));
```

Output: `[1,3,9]`

Scenario 5 — Combine Multiple Arrays

```
const a = [1,2];
const b = [3,4];
const c = [5,6];

console.log([...a, ...b, ...c]);
```

Output: `[1,2,3,4,5,6]`

3. Arrow Functions

Easy 1 — Basic Arrow

```
const add = (a,b) => a+b;
console.log(add(3,4));
```

Output: `7`

Easy 2 — Single Parameter

```
const square = x => x*x;
console.log(square(5));
```

Output: `25`

Easy 3 — No Parameter

```
const greet = () => "Hello";
console.log(greet());
```

Easy 4 — Arrow Returning Object

```
const getUser = () => ({name:"John"});
console.log(getUser());
```

Easy 5 — Arrow Inside Map

```
console.log([1,2,3].map(n => n * 2));
```
Output: `[2,4,6]`

MEDIUM

Scenario 1 — Billing Calculation

```
const bill = items => items.reduce((t,i) => t+i.price, 0);

console.log(bill([{price:100},{price:200}]));
```
Output: `300`

Scenario 2 — Sorting by Age

```
const users = [
  {name:"A", age:25},
  {name:"B", age:20}
];

console.log(users.sort((a,b) => a.age - b.age));
```

Scenario 3 — Filter Employees

```
const employees = [
  {name:"A", salary:25000},
  {name:"B", salary:50000}
];

console.log(employees.filter(e => e.salary > 30000));
```

Scenario 4 — Cart Total

```
const cart = [
  {item:"Shirt", qty:2, price:500},
  {item:"Pant", qty:1, price:800},
];

const total = cart.reduce((sum,i) => sum + i.qty*i.price, 0);
```

```
console.log(total);
```
Output: `1800`

Scenario 5 — Arrow & Closures

```
function counter() {
  let c = 0;
  return () => ++c;
}

const x = counter();
console.log(x());
console.log(x());
```

4. Timers (`setTimeout`, `setInterval`)

Easy 1 — Basic Timeout

```
setTimeout(() => console.log("Hello"), 1000);
```

Easy 2 — Interval

```
let i = 1;
const id = setInterval(() => {
  console.log(i++);
  if (i > 3) clearInterval(id);
}, 500);
```

Easy 3 — Timeout with Param

```
setTimeout(name => console.log("Hi " + name), 500, "Alex");
```

Easy 4 — Cancel Timeout

```
const id = setTimeout(() => console.log("Not printed"),
1000);
clearTimeout(id);
```

Easy 5 — Timer in Loop

```
for(let i=1;i<=3;i++){
  setTimeout(()=> console.log(i), i*200);
}
```

MEDIUM

Scenario 1 — Auto Logout

```
function autoLogout() {
  console.log("User active...");
  setTimeout(() => console.log("Logged out due to
inactivity"), 3000);
}

autoLogout();
```

Scenario 2 — Live Clock

```
setInterval(() => {
  console.log(new Date().toLocaleTimeString());
}, 1000);
```

Scenario 3 — Animated Counter

```
let n = 1;
const countId = setInterval(() => {
  console.log(n++);
  if (n > 5) clearInterval(countId);
}, 500);
```

Scenario 4 — Retry API

```
let attempts = 0;

function fakeAPI() {
  attempts++;
  console.log(`Attempt ${attempts}`);
  if (attempts < 3) {
    setTimeout(fakeAPI, 1000);
  } else {
    console.log("Success");
  }
}

fakeAPI();
```

Scenario 5 — Debounced Search

```
let timer;

function search(text) {
  clearTimeout(timer);
  timer = setTimeout(() => {
    console.log("Searching for:", text);
  }, 800);
}

search("lap");
search("laptop");
search("laptop bag");
```
Output:
Only last search executes → `"laptop bag"`.

ES6 CLASS SYNTAX

Easy 1 — Create a Basic Class

```
class Person {}
console.log(typeof Person);
```
Output:

```
function
```
Explanation:
Classes are *special functions* in JavaScript.

Easy 2 — Create a Class with a Method

```
class Car {
  start() {
    return "Engine started";
  }
}

const c = new Car();
console.log(c.start());
```
Output:

```
Engine started
```

Easy 3 — Class Expression

```
const Animal = class {
  sound() { return "makes sound"; }
}
```

```
const a = new Animal();
console.log(a.sound());
```

Easy 4 — Class with Property

```
class Box {
  size = "Large";
}

const b = new Box();
console.log(b.size);
```

Output:

```
Large
```

Easy 5 — Class vs Function

```
class A {}
function B(){}

console.log(typeof A, typeof B);
```

Output:

```
function function
```

MEDIUM

Scenario 1 — Represent a Product

```
class Product {
  name = "Laptop";
  price = 45000;

  details() {
    return `${this.name} costs ₹${this.price}`;
  }
}

const p = new Product();
console.log(p.details());
```

Scenario 2 — Represent an Employee

```
class Employee {
  constructor(id, name) {
    this.id = id;
    this.name = name;
  }

  show() {
    return `${this.id} - ${this.name}`;
  }
}

const e = new Employee(101, "Ravi");
console.log(e.show());
```

Scenario 3 — Utility Class for Logging

```
class Logger {
  log(msg) {
    console.log("LOG:", msg);
  }
}

new Logger().log("System start");
```

Scenario 4 — Represent a Bank Account

```
class Account {
  balance = 500;

  view() {
    return `Balance: ₹${this.balance}`;
  }
}

console.log(new Account().view());
```

Scenario 5 — Online Course Class

```
class Course {
  title = "JavaScript Basics";
  duration = "5 hours";

  summary() {
    return `${this.title} - ${this.duration}`;
  }
}

console.log(new Course().summary());
```

CONSTRUCTORS & METHODS

Easy 1 — Basic Constructor

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const p = new Person("Alex");
console.log(p.name);
```

Output:

```
Alex
```

Easy 2 — Method in Class

```
class MathOps {
  double(n) {
    return n * 2;
  }
}

console.log(new MathOps().double(5));
```

Easy 3 — Multiple Properties

```
class Item {
  constructor(id, price) {
    this.id = id;
    this.price = price;
  }
}

const i = new Item(10, 250);
console.log(i);
```

Easy 4 — Default Constructor Value

```
class User {
  constructor(name = "Guest") {
    this.name = name;
  }
}
```

```
console.log(new User().name);
```
Output:
```
Guest
```

Easy 5 — Calling Method

```
class Greet {
  sayHello() { return "Hello!"; }
}

console.log(new Greet().sayHello());
```

MEDIUM

Scenario 1 — Student Record

```
class Student {
  constructor(name, marks) {
    this.name = name;
    this.marks = marks;
  }

  result() {
    return this.marks >= 40 ? "Pass" : "Fail";
  }
}

const s = new Student("Anu", 55);
console.log(s.result());
```

Scenario 2 — Cart Item Cost

```
class CartItem {
  constructor(qty, price) {
    this.qty = qty;
    this.price = price;
  }

  total() {
    return this.qty * this.price;
  }
}

console.log(new CartItem(3, 150).total());
```

Scenario 3 — Billing System

```
class Bill {
  constructor(amount, tax) {
    this.amount = amount;
    this.tax = tax;
  }

  finalAmount() {
    return this.amount + (this.amount * this.tax);
  }
}

console.log(new Bill(1000, 0.18).finalAmount());
```

Scenario 4 — Package Weight System

```
class Package {
  constructor(w1, w2) {
    this.total = w1 + w2;
  }

  show() { return this.total; }
}

console.log(new Package(5, 7).show());
```

Scenario 5 — Login Validation

```
class Login {
  constructor(username, password) {
    this.username = username;
    this.password = password;
  }

  validate() {
    return this.username === "admin" && this.password ===
"1234";
  }
}

console.log(new Login("admin", "1234").validate());
```

INHERITANCE

Easy 1 — Simple Inheritance

```
class A {}
class B extends A {}

console.log(new B instanceof A);
```

Output:

```
true
```

Easy 2 — Call Parent Method

```
class Parent {
  greet() { return "Hello"; }
}

class Child extends Parent {}

console.log(new Child().greet());
```

Easy 3 — Override Method

```
class A {
  show(){ return "A"; }
}

class B extends A {
  show(){ return "B"; }
}

console.log(new B().show());
```

Easy 4 — Access Parent Using super()

```
class A {
  msg() { return "Parent"; }
}

class B extends A {
  msg() { return super.msg() + " + Child"; }
}

console.log(new B().msg());
```

Easy 5 — Constructor Inheritance

```
class A {
  constructor(n) { this.n = n; }
}

class B extends A {}

console.log(new B(10).n);
```

MEDIUM

Scenario 1 — Vehicle → Car

```
class Vehicle {
  move() { return "Moving"; }
}

class Car extends Vehicle {
  wheels = 4;
}

console.log(new Car().move(), new Car().wheels);
```

Scenario 2 — Employee → Manager

```
class Employee {
  constructor(name) { this.name = name; }
}

class Manager extends Employee {
  role = "Manager";
}

console.log(new Manager("Meera"));
```

Scenario 3 — Shape → Square

```
class Shape {
  area() { return 0; }
}

class Square extends Shape {
  constructor(side) {
    super();
    this.side = side;
  }
  area() {
    return this.side * this.side;
  }
}

console.log(new Square(5).area());
```

Scenario 4 — Account → SavingsAccount

```
class Account {
  constructor(balance){ this.balance = balance; }
}

class SavingsAccount extends Account {
  addInterest() { return this.balance * 1.05; }
}

console.log(new SavingsAccount(1000).addInterest());
```

Scenario 5 — Electronics → Mobile

```
class Electronics {
  warranty() { return "1 year"; }
}

class Mobile extends Electronics {
  os = "Android";
}

console.log(new Mobile().os, new Mobile().warranty());
```

4. STATIC METHODS & PROPERTIES

Easy 1 — Static Method

```
class MathUtil {
  static add(a,b){ return a+b; }
}

console.log(MathUtil.add(2,3));
```

Easy 2 — Static Property

```
class Counter {
  static count = 0;
}

console.log(Counter.count);
```

Easy 3 — Access Static Inside Class

```
class A {
  static x = 10;
  static show(){ return A.x; }
}

console.log(A.show());
```

Easy 4 — Instance Cannot Access Static

```
class A {
  static greet(){ return "Hello"; }
}

const a = new A();
console.log(typeof a.greet);
```

Output:

```
undefined
```

Easy 5 — Static Factory Method

```
class User {
  constructor(name){ this.name = name; }
  static createGuest(){ return new User("Guest"); }
}

console.log(User.createGuest());
```

🔷 MEDIUM — SCENARIO BASED (5)

Scenario 1 — ID Generator

```
class IDGen {
  static id = 100;

  static next() { return ++this.id; }
}

console.log(IDGen.next());
console.log(IDGen.next());
```

Scenario 2 — Validate Email

```
class Validator {
  static isEmail(str){
    return str.includes("@");
  }
}

console.log(Validator.isEmail("test@mail.com"));
```

Scenario 3 — Create Default Config

```
class Config {
  static default() {
    return { theme:"light", lang:"en" };
  }
}

console.log(Config.default());
```

Scenario 4 — Calculate Tax

```
class Tax {
  static rate = 0.18;

  static calc(amount) {
    return amount * this.rate;
  }
}

console.log(Tax.calc(1000));
```

Scenario 5 — Static Counter in Child Class

```
class Order {
  static count = 0;
  constructor(){ Order.count++; }
}

new Order();
new Order();
console.log(Order.count);
```

TRY / CATCH / FINALLY

Easy 1 — Basic try/catch

```
try {
  let a = b;  // b is not defined
} catch (err) {
  console.log("Error caught!");
}
```

Output:

```
Error caught!
```

Explanation:
Undefined variable `b` triggers a runtime error → caught in `catch`.

Easy 2 — Access error message

```
try {
  JSON.parse("{name: 'John'}");
} catch (err) {
  console.log(err.message);
}
```

Output:

```
Unexpected token n in JSON at position 1
```

Easy 3 — finally block always runs

```
try {
  throw new Error("Failed");
} catch (err) {
  console.log("Caught");
} finally {
  console.log("Cleanup");
}
```

Output:

```
Caught
Cleanup
```

Easy 4 — No error case

```
try {
  console.log("OK");
} catch {
  console.log("Error");
} finally {
  console.log("Done");
}
```

Output:

```
OK
Done
```

Easy 5 — Try inside function

```
function test() {
  try {
    return "Success";
  } finally {
    console.log("Still running cleanup");
  }
}
console.log(test());
```

Output:

```
Still running cleanup
Success
```

MEDIUM

Scenario 1 — Validate user input in form

```
function getAge(age) {
  try {
    if (age < 0) throw new Error("Age cannot be negative");
    return age;
  } catch (err) {
    return err.message;
  }
}

console.log(getAge(-5));
```

Output:

```
Age cannot be negative
```

Scenario 2 — File parsing

```
function parseData(data) {
  try {
    return JSON.parse(data);
  } catch {
    return "Invalid JSON";
  }
}

console.log(parseData("{bad json}"));
```
Output:

```
Invalid JSON
```

Scenario 3 — Database simulation

```
function connectDB(isConnected) {
  try {
    if (!isConnected) throw new Error("DB connection
failed");
    return "Connected";
  } catch (err) {
    return err.message;
  } finally {
    console.log("Attempt complete");
  }
}

console.log(connectDB(false));
```
Output:

```
Attempt complete
DB connection failed
```

Scenario 4 — Payment gateway

```
function processPayment(balance, amount) {
  try {
    if (amount > balance) throw new Error("Insufficient
balance");
    return "Payment Successful";
  } catch (e) {
    return e.message;
  }
}

console.log(processPayment(1000, 2000));
```

Scenario 5 — API call mock

```
function fetchAPI(success) {
  try {
    if (!success) throw new Error("API request failed");
    return "Data received";
  } catch (e) {
    return e.message;
  }
}

console.log(fetchAPI(false));
```

2. CUSTOM ERRORS

Easy 1 — Create simple custom error

```
class MyError extends Error {}

try {
  throw new MyError("Something wrong");
} catch (e) {
  console.log(e.message);
}
```

Output:

```
Something wrong
```

Easy 2 — Custom error with name

```
class ValidationError extends Error {
  constructor(msg) {
    super(msg);
    this.name = "ValidationError";
  }
}

try {
  throw new ValidationError("Invalid Input");
} catch (e) {
  console.log(e.name);
}
```

Output:

```
ValidationError
```

Easy 3 — Throw inside function

```
class AgeError extends Error {}

function check(age) {
  if (age < 18) throw new AgeError("Underage");
  return "Allowed";
}

try {
  console.log(check(10));
} catch (e) {
  console.log(e.message);
}
```

Easy 4 — Custom Range Error

```
class RangeErrorCustom extends RangeError {}

try {
  throw new RangeErrorCustom("Out of range");
} catch (e) {
  console.log(e.message);
}
```

Easy 5 — Validate username

```
class UserError extends Error {}

try {
  throw new UserError("Username required");
} catch (e) {
  console.log(e.message);
}
```

MEDIUM

Scenario 1 — Login system

```
class AuthError extends Error {}

function login(user) {
  if (!user) throw new AuthError("User not found");
  return "Login success";
}

try {
  console.log(login(null));
} catch (e) {
  console.log(e.message);
}
```
Output:

```
User not found
```

Scenario 2 — Banking limit

```
class LimitError extends Error {}

function withdraw(amount) {
  if (amount > 10000) throw new LimitError("Limit exceeded");
  return "Withdrawn";
}

try {
  console.log(withdraw(15000));
} catch (e) {
  console.log(e.message);
}
```

Scenario 3 — File upload validation

```
class FileTypeError extends Error {}

function upload(type) {
  if (type !== "jpg" && type !== "png")
    throw new FileTypeError("Invalid file type");
  return "Uploaded";
}

try {
  console.log(upload("pdf"));
} catch (e) {
  console.log(e.message);
}
```

Scenario 4 — Product availability

```
class StockError extends Error {}

function buy(item, stock) {
  if (stock === 0) throw new StockError(item + " out of
stock");
  return "Purchased";
}

try {
  console.log(buy("Laptop", 0));
} catch (e) {
  console.log(e.message);
}
```

Scenario 5 — Custom Password Error

```
class PasswordError extends Error {}

function validate(pwd) {
  if (pwd.length < 6) throw new PasswordError("Too short");
  return "Valid";
}

try {
  console.log(validate("123"));
} catch (e) {
  console.log(e.message);
}
```

3. ERROR PROPAGATION

Easy 1 — Rethrow error

```
function a() {
  try {
    throw new Error("Oops");
  } catch (e) {
    throw e;
  }
}

try {
  a();
} catch (e) {
  console.log(e.message);
}
```

Easy 2 — Error thrown inside nested function

```
function inner() {
  throw new Error("Inner error");
}

function outer() {
  inner();
}

try {
  outer();
} catch (e) {
  console.log(e.message);
}
```

Easy 3 — Function chain

```
function f1() { throw new Error("Err"); }
function f2() { f1(); }

try { f2(); }
catch (e) { console.log(e.message); }
```

Easy 4 — Try inside outer

```
function run() {
  try {
    throw new Error("Break");
  } catch (e) {
    console.log("Handled");
  }
}

run();
```

Easy 5 — Propagation with return

```
function test() {
  try {
    throw new Error("Problem");
  } finally {
    console.log("Running");
  }
}

try {
  test();
} catch (e) {
  console.log(e.message);
}
```

MEDIUM

Scenario 1 — API → Service → UI Layer

```
function apiCall() {
  throw new Error("API failed");
}

function service() {
  apiCall(); // propagates
}

function ui() {
  try {
    service();
  } catch (e) {
    console.log("Handled in UI:", e.message);
  }
}

ui();
```

Output:

```
Handled in UI: API failed
```

Scenario 2 — Billing system

```
function calculate(price) {
  if (price <= 0) throw new Error("Invalid price");
  return price * 1.18;
}

function checkout(price) {
  return calculate(price); // passes error
}
```

```
try {
  console.log(checkout(-200));
} catch (err) {
  console.log("Checkout error:", err.message);
}
```

Scenario 3 — Registration workflow

```
function validateEmail(email) {
  if (!email.includes("@")) throw new Error("Invalid email");
}

function register(email) {
  validateEmail(email);
}

try {
  register("wrongmail.com");
} catch (e) {
  console.log("Registration failed:", e.message);
}
```

Scenario 4 — Payment gateway layers

```
function debit(amount) {
  if (amount > 5000) throw new Error("Over limit");
}

function process(amount) { debit(amount); }

try {
  process(7000);
} catch (e) {
  console.log("Payment failed:", e.message);
}
```

Scenario 5 — Inventory system

```
function checkStock(stock) {
  if (stock === 0) throw new Error("Out of stock");
}

function order(stock) {
  checkStock(stock);
}
```

```
try {
  order(0);
} catch (e) {
  console.log("Order failed:", e.message);
}
```