JavaScript Essentials for TypeScript/Angular

1. Variables and Scope

Variable Declarations

```
// Use these
let name = "John";        // Block-scoped, can be reassigned
const age = 25;           // Block-scoped, cannot be reassigned
const user = { name };    // Object reference is constant, properties can change

// Avoid this
var oldStyle = "deprecated"; // Function-scoped, hoisted
```

Block Scope

```
if (true) {
  let blockScoped = "only available here";
  const alsoBlockScoped = "same";
}
// blockScoped is not accessible here
```

2. Functions

Arrow Functions vs Regular Functions

```
// Arrow functions (preferred for most cases)
const add = (a, b) => a + b;
const multiply = (x, y) => {
  return x * y;
};

// Regular functions (when you need 'this' context)
function regularFunction() {
```

```
  return this; // 'this' behaves differently
}

// Function expressions
const myFunc = function(param) {
  return param * 2;
};
```

Default Parameters

```
const greet = (name = "World") => `Hello, ${name}!`;
```

3. Objects and Arrays

Object Creation and Access

```
const person = {
  name: "Alice",
  age: 30,
  city: "NYC"
};

// Property access
console.log(person.name);     // Dot notation
console.log(person["age"]);   // Bracket notation

// Dynamic property names
const key = "city";
console.log(person[key]);
```

Object Methods

```
const user = { name: "Bob", age: 25, role: "dev" };

// Get all keys
Object.keys(user);           // ["name", "age", "role"]

// Get all values
Object.values(user);         // ["Bob", 25, "dev"]

// Get key-value pairs
```

```
Object.entries(user);        // [["name", "Bob"], ["age", 25], ["role", "dev"]]

// Copy/merge objects
const newUser = Object.assign({}, user, { active: true });
const anotherWay = { ...user, active: true }; // Spread operator (preferred)
```

Array Methods (CRITICAL - Used everywhere)

```
const numbers = [1, 2, 3, 4, 5];
const users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 35 }
];

// Transform arrays
const doubled = numbers.map(n => n * 2);           // [2, 4, 6, 8, 10]
const names = users.map(user => user.name);        // ["Alice", "Bob", "Charlie"]

// Filter arrays
const adults = users.filter(user => user.age >= 30); // Bob and Charlie
const evens = numbers.filter(n => n % 2 === 0);      // [2, 4]

// Reduce arrays
const sum = numbers.reduce((acc, n) => acc + n, 0);  // 15
const totalAge = users.reduce((acc, user) => acc + user.age, 0); // 90

// Find elements
const bob = users.find(user => user.name === "Bob");
const hasAdult = users.some(user => user.age >= 30);  // true
const allAdults = users.every(user => user.age >= 18); // true

// Iterate
users.forEach(user => console.log(user.name));

// Check if array includes value
numbers.includes(3); // true
```

4. Destructuring Assignment

Object Destructuring

```
const person = { name: "Alice", age: 30, city: "NYC" };
```

```
// Basic destructuring
const { name, age } = person;

// With different variable names
const { name: personName, age: personAge } = person;

// With default values
const { name, country = "USA" } = person;

// Nested destructuring
const user = {
  profile: { name: "Bob", settings: { theme: "dark" } }
};
const { profile: { name }, profile: { settings: { theme } } } = user;
```

Array Destructuring

```
const colors = ["red", "green", "blue"];

// Basic destructuring
const [first, second, third] = colors;

// Skip elements
const [primary, , tertiary] = colors;

// With rest operator
const [head, ...tail] = colors; // head = "red", tail = ["green", "blue"]
```

5. Template Literals

```
const name = "Alice";
const age = 25;

// Template literals (preferred)
const message = `Hello, my name is ${name} and I'm ${age} years old.`;

// Multi-line strings
const html = `
  <div>
    <h1>${name}</h1>
    <p>Age: ${age}</p>
  </div>
`;
```

```
// Expression evaluation
const result = `The sum is: ${10 + 5}`;
```

6. Spread Operator and Rest Parameters

Spread Operator

```
// Arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Objects
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const merged = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 3, d: 4 }

// Function arguments
const numbers = [1, 2, 3];
Math.max(...numbers); // Same as Math.max(1, 2, 3)
```

Rest Parameters

```
// Collect function arguments
const sum = (...numbers) => {
  return numbers.reduce((acc, n) => acc + n, 0);
};

sum(1, 2, 3, 4); // 10

// In destructuring
const [first, ...rest] = [1, 2, 3, 4]; // first = 1, rest = [2, 3, 4]
```

7. Modules (Import/Export)

Named Exports

```
// math.js
export const PI = 3.14159;
export const add = (a, b) => a + b;
export const multiply = (a, b) => a * b;

// main.js
import { PI, add, multiply } from './math.js';
import { add as addition } from './math.js'; // Rename import
```

Default Exports

```
// calculator.js
const Calculator = {
  add: (a, b) => a + b,
  subtract: (a, b) => a - b
};
export default Calculator;

// main.js
import Calculator from './calculator.js';
import Calc from './calculator.js'; // Can use any name
```

Mixed Exports

```
// utils.js
export const helper = () => "help";
const MainUtil = { /* ... */ };
export default MainUtil;

// main.js
import MainUtil, { helper } from './utils.js';
```

8. Promises and Async/Await

Promises

```
// Creating promises
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      const success = true;
      if (success) {
        resolve({ data: "Hello World" });
      } else {
        reject(new Error("Failed to fetch"));
      }
    }, 1000);
  });
};

// Using promises
fetchData()
  .then(result => console.log(result.data))
  .catch(error => console.error(error));

// Promise methods
Promise.all([promise1, promise2, promise3])  // All must resolve
  .then(results => console.log(results));

Promise.race([promise1, promise2])           // First to resolve/reject
  .then(result => console.log(result));
```

Async/Await

```
// Async functions always return promises
const fetchUserData = async (userId) => {
  try {
    const response = await fetch(`/api/users/${userId}`);
    const userData = await response.json();
    return userData;
  } catch (error) {
    console.error("Error fetching user:", error);
    throw error;
  }
};

// Using async functions
const loadUser = async () => {
  const user = await fetchUserData(123);
  console.log(user);
};
```

9. Classes (Foundation for TypeScript)

```
class User {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  // Method
  greet() {
    return `Hello, I'm ${this.name}`;
  }

  // Static method
  static createGuest() {
    return new User("Guest", "guest@example.com");
  }
}

// Inheritance
class Admin extends User {
  constructor(name, email, permissions) {
    super(name, email); // Call parent constructor
    this.permissions = permissions;
  }

  hasPermission(permission) {
    return this.permissions.includes(permission);
  }
}

// Usage
const user = new User("Alice", "alice@example.com");
const admin = new Admin("Bob", "bob@example.com", ["read", "write"]);
```

10. Error Handling

```
// Try-catch blocks
const divide = (a, b) => {
  try {
    if (b === 0) {
      throw new Error("Division by zero!");
    }
    return a / b;
  } catch (error) {
    console.error("Error:", error.message);
    return null;
  } finally {
    console.log("Division operation completed");
```

```
  }
};

// With async/await
const fetchData = async () => {
  try {
    const response = await fetch('/api/data');
    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }
    return await response.json();
  } catch (error) {
    console.error("Fetch failed:", error);
    return null;
  }
};
```

11. Essential Concepts for Angular

Closures and Lexical Scope

```
const createCounter = () => {
  let count = 0;
  return () => {
    count++;
    return count;
  };
};

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

This Context (Important for understanding Angular components)

```
const obj = {
  name: "Object",
  regularMethod: function() {
    console.log(this.name); // "Object"
  },
  arrowMethod: () => {
    console.log(this.name); // undefined (lexical this)
  }
};
```

Callback Functions

```
const processData = (data, callback) => {
  const processed = data.map(item => item * 2);
  callback(processed);
};

processData([1, 2, 3], (result) => {
  console.log(result); // [2, 4, 6]
});
```

Quick Reference Checklist

Before moving to TypeScript, ensure you can:

- ☐ Use `let` and `const` appropriately
- ☐ Write and understand arrow functions
- ☐ Destructure objects and arrays
- ☐ Use template literals
- ☐ Apply array methods (map, filter, reduce, find, etc.)
- ☐ Use spread operator and rest parameters
- ☐ Work with import/export statements
- ☐ Write and consume Promises
- ☐ Use async/await syntax
- ☐ Create and extend classes
- ☐ Handle errors with try-catch
- ☐ Understand scope and closures

Once you're comfortable with these concepts, you're ready for TypeScript, which will add type safety and additional features on top of these fundamentals.