← Getting Started

Async JavaScript & Node.js →

QE Fundamentals & TypeScript for Test Automation

Table of Contents

1. Overview of Quality Engineering
2. Manual vs Automation Testing
3. Test Automation Pyramid
4. Development Environment Setup
5. JavaScript Fundamentals Recap
6. TypeScript Fundamentals
7. Functions in TypeScript
8. Interfaces, Types, and Classes
9. Arrays, Objects, and Destructuring
10. VS Code Debugging Basics
11. Practice Exercises

Overview of Quality Engineering

throughout the entire development lifecycle. Unlike traditional Quality Assurance (QA), which focuses on finding defects, Quality Engineering emphasizes **preventing defects** through better engineering practices.

Key Principles of Quality Engineering

1. Shift-Left Testing

- Testing activities begin early in the development cycle
- Prevents expensive late-stage bug fixes
- Involves collaboration between developers and QE engineers

2. Continuous Testing

- Testing integrated into CI/CD pipelines
- Automated tests run on every code commit
- Provides immediate feedback to developers

3. Quality Culture

- Quality is everyone's responsibility
- Developers write unit tests
- QE engineers focus on integration and E2E tests

4. Test Automation First

- Automate repetitive test cases
- Manual testing for exploratory and usability testing
- Automation enables faster releases

The Role of QE Engineers

Responsibilities:

- Design and implement test automation frameworks
- Write automated test scripts (UI, API, Mobile)

- Collaborate with developers to improve code quality
- Perform exploratory testing for critical features
- Maintain and update test suites

Skills Required:

- Programming (JavaScript, TypeScript, Python, Java)
- Test automation tools (Playwright, Cypress, Selenium)
- API testing (REST, GraphQL)
- Version control (Git, GitHub)
- CI/CD (Jenkins, GitHub Actions, GitLab CI)
- Cloud platforms (AWS, Azure, BrowserStack)
- Database and SQL knowledge

Manual vs Automation Testing

Manual Testing

Definition: Human testers execute test cases step-by-step without automation tools.

When to Use Manual Testing:

- Exploratory testing
- Usability testing
- Ad-hoc testing
- Visual verification (UI/UX)
- One-time test scenarios
- Complex scenarios requiring human judgment

Advantages:

- Flexible and adaptable

- No initial setup cost

Disadvantages:

- Time-consuming
- Prone to human error
- Not repeatable
- Expensive for regression testing
- Cannot simulate load/stress

Automation Testing

Definition: Using tools and scripts to execute tests automatically.

When to Use Automation Testing:

- Regression testing
- Smoke testing
- Data-driven testing
- Load and performance testing
- Repetitive test scenarios
- Cross-browser/device testing
- API testing

Advantages:

- Fast execution
- Reliable and repeatable
- Cost-effective for regression
- Enables CI/CD
- Can run 24/7
- Detailed reporting

Disadvantages:

- Maintenance overhead
- Cannot find all bugs
- Limited to scripted scenarios

Manual vs Automation Comparison

Aspect	Manual Testing	Automation Testing
Speed	Slow	Fast
Reliability	Human error prone	Highly reliable
Cost (Short-term)	Low	High (setup)
Cost (Long-term)	High (repetitive)	Low
Flexibility	High	Limited to scripts
Regression Testing	Inefficient	Highly efficient
Exploratory Testing	Excellent	Not suitable
CI/CD Integration	Difficult	Easy
ROI	Low for repetitive	High for repetitive

Test Automation Pyramid

Overview

The Test Automation Pyramid is a testing strategy that defines the optimal distribution of different types of automated tests.

```
        /--------\
       /   API    \        ← Moderate (30%)
      /------------\
     /     Unit     \      ← Many (60%)
    /________________\
```

Unit Tests (Base - 60%)

Purpose: Test individual functions, methods, or classes in isolation.

Characteristics:

- Very fast execution (milliseconds)
- Test smallest units of code
- No external dependencies (mocked)
- Written by developers
- Run on every code commit

Tools:

- Jest (JavaScript/TypeScript)
- Mocha/Chai
- JUnit (Java)
- pytest (Python)

Example:

```
 // Function to test
function add(a: number, b: number): number {
  return a + b;
}

// Unit test
describe('add function', () => {
  it('should return sum of two numbers', () => {
    expect(add(2, 3)).toBe(5);
    expect(add(-1, 1)).toBe(0);
```

Benefits:

- Catch bugs early
- Fast feedback
- Easy to debug
- High code coverage
- Enable refactoring

Integration/API Tests (Middle - 30%)

Purpose: Test interactions between components, modules, or services.

Characteristics:

- Test business logic
- Validate API contracts
- Test database interactions
- Moderate execution speed
- More complex than unit tests

Tools:

- Playwright (API Testing)
- Postman/Newman
- REST Assured
- Supertest

Example:

```
 // API Integration Test
test('GET /users should return user list', async ({ request }) => {
  const response = await request.get('/api/users');
  expect(response.status()).toBe(200);
  const users = await response.json();
  expect(users.length).toBeGreaterThan(0);
```

Benefits:

- Test business logic
- Faster than E2E tests
- Catch integration issues
- Independent of UI

End-to-End Tests (Top - 10%)

Purpose: Test complete user workflows from start to finish.

Characteristics:

- Simulate real user scenarios
- Test entire application stack
- Slow execution (minutes)
- Fragile (UI changes break tests)
- High maintenance cost

Tools:

- Playwright
- Cypress
- Selenium WebDriver
- TestCafe

Example:

```
 // E2E Test
test('User can complete checkout process', async ({ page }) => {
  await page.goto('/products');
  await page.getByRole('button', { name: 'Add to Cart' }).first().click();
  await page.getByRole('link', { name: 'Checkout' }).click();
  await page.getByLabel('Email').fill('user@example.com');
  await page.getByLabel('Card Number').fill('4111111111111111');
```

Benefits:

- Test real user scenarios
- Validate entire system
- Catch system-level issues
- Build confidence in releases

Why the Pyramid Shape?

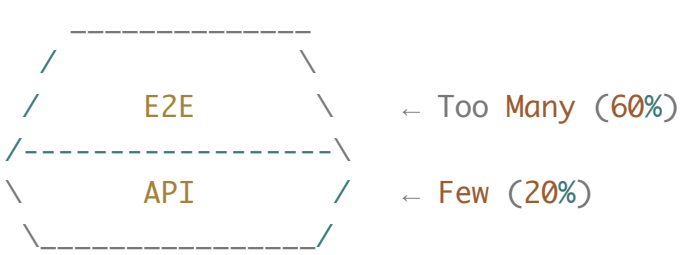
Reasons for More Unit Tests:

1. **Speed**: Unit tests execute in milliseconds
2. **Stability**: Less flaky than UI tests
3. **Cost**: Cheaper to maintain
4. **Debugging**: Easy to pinpoint failures
5. **Coverage**: Can test edge cases easily

Reasons for Fewer E2E Tests:

1. **Slow**: Take minutes to execute
2. **Fragile**: UI changes break tests
3. **Expensive**: High maintenance cost
4. **Debugging**: Hard to identify root cause
5. **Environment**: Require full stack setup

Anti-Pattern: Ice Cream Cone



Problems:

- Slow test suite execution
- Flaky tests
- High maintenance cost
- Late feedback
- Difficult debugging

Development Environment Setup

Prerequisites

Before starting test automation, set up your development environment with essential tools.

Node.js and npm

Node.js is a JavaScript runtime that executes JavaScript outside the browser. **npm** (Node Package Manager) manages JavaScript packages.

Installation:

Windows:

1. Download Node.js from https://nodejs.org/
2. Run the installer (LTS version recommended)
3. Verify installation:

```
node --version
npm --version
```

macOS:

```
# Verify
node --version
npm --version
```

Linux (Ubuntu/Debian):

```
 # Using apt
curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt-get install -y nodejs

# Verify
node --version
npm --version
```

Visual Studio Code (VS Code)

Why VS Code?

- Free and open-source
- Excellent TypeScript support
- Rich extension ecosystem
- Integrated terminal
- Built-in debugger
- Git integration

Installation:

1. Download from https://code.visualstudio.com/
2. Install for your operating system

Essential Extensions:

- ESLint (Code linting)
- Prettier (Code formatting)
- Playwright Test for VSCode

- Thunder Client (API testing)
- Path Intellisense
- Auto Rename Tag
- Bracket Pair Colorizer

VS Code Configuration:

Create `.vscode/settings.json` :

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "typescript.preferences.importModuleSpecifier": "relative",
  "files.autoSave": "onFocusChange"
}
```

Git

Why Git?

- Version control for code
- Collaboration with team
- Track changes and history
- Integration with CI/CD

Installation:

Windows:

- Download from https://git-scm.com/
- Run installer with default options

macOS:

Linux:

```
sudo apt-get install git
```

Configuration:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
git config --global init.defaultBranch main
```

Create Project Structure

Initialize a Test Automation Project:

```
# Create project directory
mkdir test-automation-project
cd test-automation-project

# Initialize npm project
npm init -y

# Initialize git repository
git init

# Create .gitignore
cat > .gitignore << EOF
node_modules/
test-results/
playwright-report/
.env
*.log
EOF

# Create folder structure
mkdir -p tests/web tests/api tests/mobile
mkdir -p utils
mkdir -p pages
mkdir -p fixtures
```

```
test-automation-project/
├── node_modules/
├── tests/
|   ├── web/
|   ├── api/
|   └── mobile/
├── pages/
├── utils/
├── fixtures/
├── .gitignore
├── package.json
├── tsconfig.json
└── playwright.config.ts
```

JavaScript Fundamentals Recap

Variables

Three Ways to Declare Variables:

```
// var (old way - avoid in modern code)
var name = "John";

// let (block-scoped, can be reassigned)
let age = 25;
age = 26; // OK

// const (block-scoped, cannot be reassigned)
const PI = 3.14159;
// PI = 3.14; // Error: Cannot reassign const
```

Best Practices:

- Use `const` by default
- Use `let` only if you need to reassign
- Avoid `var` completely

```
// String
const name = "Alice";
const greeting = 'Hello';
const template = `Hello, ${name}!`;

// Number
const age = 30;
const price = 19.99;
const negative = -42;

// Boolean
const isActive = true;
const hasError = false;

// Undefined
let value; // undefined
console.log(value); // undefined

// Null
const empty = null;

// Symbol (unique identifier)
const id = Symbol('id');
```

Reference Types:

```
// Object
const user = {
  name: "Bob",
  age: 28,
  email: "bob@example.com"
};

// Array
const numbers = [1, 2, 3, 4, 5];
const mixed = [1, "two", true, null];

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

Operators

Arithmetic Operators:

```
 const a = 10;
const b = 3;

console.log(a + b);  // 13 (Addition)
console.log(a - b);  // 7 (Subtraction)
console.log(a * b);  // 30 (Multiplication)
console.log(a / b);  // 3.333... (Division)
console.log(a % b);  // 1 (Modulus/Remainder)
console.log(a ** b); // 1000 (Exponentiation)
```

Comparison Operators:

```
 const x = 5;
const y = "5";

console.log(x == y);   // true (loose equality - converts types)
console.log(x === y);  // false (strict equality - no type conversion)
console.log(x != y);   // false
console.log(x !== y);  // true
console.log(x > 3);    // true
console.log(x >= 5);   // true
console.log(x < 10);   // true
```

Logical Operators:

```
 const isLoggedIn = true;
const hasPermission = false;

console.log(isLoggedIn && hasPermission);  // false (AND)
console.log(isLoggedIn || hasPermission);  // true (OR)
console.log(!isLoggedIn);                  // false (NOT)
```

Control Flow

```
const age = 20;

if (age >= 18) {
  console.log("Adult");
} else if (age >= 13) {
  console.log("Teenager");
} else {
  console.log("Child");
}

// Ternary operator
const status = age >= 18 ? "Adult" : "Minor";
```

Switch Statement:

```
const day = "Monday";

switch (day) {
  case "Monday":
  case "Tuesday":
  case "Wednesday":
  case "Thursday":
  case "Friday":
    console.log("Weekday");
    break;
  case "Saturday":
  case "Sunday":
    console.log("Weekend");
    break;
  default:
    console.log("Invalid day");
}
```

Loops:

```
// For loop
for (let i = 0; i < 5; i++) {
  console.log(i);
}

// While loop
let count = 0;
```

```
}

// Do-while loop
let num = 0;
do {
  console.log(num);
  num++;
} while (num < 5);

// For...of loop (arrays)
const fruits = ["apple", "banana", "orange"];
for (const fruit of fruits) {
  console.log(fruit);
}

// For...in loop (objects)
const person = { name: "John", age: 30 };
for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

TypeScript Fundamentals

What is TypeScript?

TypeScript is a **superset of JavaScript** that adds static typing. It compiles to plain JavaScript and helps catch errors during development.

Benefits:

- Type safety (catch errors at compile time)
- Better IDE support (autocomplete, refactoring)
- Self-documenting code
- Enhanced code maintainability
- Modern JavaScript features

```
 # Install TypeScript globally
npm install -g typescript

# Check version
tsc --version

# Install TypeScript in project
npm install --save-dev typescript @types/node

# Initialize TypeScript configuration
npx tsc --init
```

tsconfig.json Configuration

```
 {
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "lib": ["ES2022"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "moduleResolution": "node",
    "types": ["node"]
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Basic Type Annotations

Primitive Types:

```
 // String
let name: string = "Alice";
let greeting: string = 'Hello';
```

```
let price: number = 19.99;

// Boolean
let isActive: boolean = true;
let hasError: boolean = false;

// Any (avoid if possible)
let anything: any = "can be anything";
anything = 42;
anything = true;

// Unknown (safer than any)
let userInput: unknown;
userInput = "test";
if (typeof userInput === "string") {
  console.log(userInput.toUpperCase()); // Type narrowing
}

// Void (no return value)
function log(message: string): void {
  console.log(message);
}

// Never (never returns)
function throwError(message: string): never {
  throw new Error(message);
}

// Null and Undefined
let nullable: null = null;
let undefinedValue: undefined = undefined;
```

Type Inference

TypeScript can automatically infer types:

```
 // Type inferred as string
let message = "Hello"; // message: string

// Type inferred as number
let count = 10; // count: number

// Type inferred from return value
```

```
const result = add(5, 3); // result: number
```

Union Types

Variables can have multiple types:

```
 // Union type
let id: number | string;
id = 123;      // OK
id = "ABC123"; // OK
// id = true;  // Error

// Function with union type
function printId(id: number | string): void {
  if (typeof id === "string") {
    console.log(id.toUpperCase());
  } else {
    console.log(id.toFixed(2));
  }
}

// Array of union types
let mixed: (number | string)[] = [1, "two", 3, "four"];
```

Type Aliases

Create custom type names:

```
 // Simple alias
type ID = number | string;
type Status = "pending" | "approved" | "rejected";

let userId: ID = 123;
let orderStatus: Status = "pending";

// Object type alias
type User = {
  id: number;
  name: string;
```

```typescript
const user: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com"
};

// Function type alias
type CalculateFunction = (a: number, b: number) => number;

const add: CalculateFunction = (a, b) => a + b;
const multiply: CalculateFunction = (a, b) => a * b;
```

Literal Types

Exact values as types:

```typescript
 // String literal type
let direction: "north" | "south" | "east" | "west";
direction = "north"; // OK
// direction = "up"; // Error

// Numeric literal type
let diceRoll: 1 | 2 | 3 | 4 | 5 | 6;
diceRoll = 4; // OK
// diceRoll = 7; // Error

// Combined literal types
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";
type StatusCode = 200 | 201 | 400 | 401 | 404 | 500;

function makeRequest(method: HttpMethod, url: string): StatusCode {
  // Implementation
  return 200;
}
```

Functions in TypeScript

```typescript
// Basic function
function greet(name: string): string {
  return `Hello, ${name}!`;
}

// Function with optional parameter
function greetUser(name: string, greeting?: string): string {
  return `${greeting || "Hello"}, ${name}!`;
}

// Function with default parameter
function calculatePrice(price: number, tax: number = 0.1): number {
  return price + (price * tax);
}

// Function with rest parameters
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15
```

Arrow Functions

```typescript
// Basic arrow function
const add = (a: number, b: number): number => {
  return a + b;
};

// Concise arrow function
const multiply = (a: number, b: number): number => a * b;

// Arrow function with one parameter (parentheses optional)
const square = (x: number): number => x * x;

// Arrow function with no parameters
const getRandomNumber = (): number => Math.random();

// Arrow function with complex logic
const calculateDiscount = (price: number, discountPercent: number): number
  if (discountPercent < 0 || discountPercent > 100) {
    throw new Error("Invalid discount percentage");
```

Function Types

```
 // Function type
type MathOperation = (a: number, b: number) => number;

const add: MathOperation = (a, b) => a + b;
const subtract: MathOperation = (a, b) => a - b;

// Function as parameter
function calculate(
  a: number,
  b: number,
  operation: MathOperation
): number {
  return operation(a, b);
}

console.log(calculate(10, 5, add));      // 15
console.log(calculate(10, 5, subtract)); // 5

// Function returning function
function createMultiplier(multiplier: number): (value: number) => number {
  return (value: number) => value * multiplier;
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Async Functions

```
 // Async function returning Promise
async function fetchUser(id: number): Promise<User> {
  const response = await fetch(`/api/users/${id}`);
  const user = await response.json();
  return user;
}
```

```
  const user = await fetchUser(userId);
  return user;
};

// Error handling in async functions
async function fetchUserSafe(id: number): Promise<User | null> {
  try {
    const response = await fetch(`/api/users/${id}`);
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const user = await response.json();
    return user;
  } catch (error) {
    console.error("Failed to fetch user:", error);
    return null;
  }
}
```

Interfaces, Types, and Classes

Interfaces

Interfaces define the structure of objects:

```
 // Basic interface
interface User {
  id: number;
  name: string;
  email: string;
  age?: number;        // Optional property
  readonly role: string; // Read-only property
}

const user: User = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  role: "admin"
```

```
// Interface with methods
interface Product {
  id: number;
  name: string;
  price: number;
  calculateDiscount(percent: number): number;
}

const product: Product = {
  id: 1,
  name: "Laptop",
  price: 1000,
  calculateDiscount(percent: number): number {
    return this.price - (this.price * percent / 100);
  }
};

// Interface extending another interface
interface Employee extends User {
  employeeId: string;
  department: string;
  salary: number;
}

const employee: Employee = {
  id: 1,
  name: "Bob",
  email: "bob@example.com",
  role: "employee",
  employeeId: "EMP001",
  department: "Engineering",
  salary: 75000
};
```

Type Aliases vs Interfaces

```
 // Type alias
type Point = {
  x: number;
  y: number;
};
```

```
  x: number;
  y: number;
}

// Both work similarly for objects
const p1: Point = { x: 10, y: 20 };
const p2: IPoint = { x: 10, y: 20 };

// Type aliases can represent union types
type ID = number | string;

// Interfaces can be extended
interface Shape {
  color: string;
}

interface Circle extends Shape {
  radius: number;
}

// Type aliases can use intersection
type Colorful = { color: string };
type CircleType = Colorful & { radius: number };
```

When to use Interface vs Type:

- Use **Interface** for object shapes that might be extended
- Use **Type** for unions, intersections, or primitive aliases
- Both work well for most cases

Classes

```
 // Basic class
class Person {
  // Properties
  name: string;
  age: number;

  // Constructor
  constructor(name: string, age: number) {
    this.name = name;
```

```
  // Method
  greet(): string {
    return `Hello, my name is ${this.name}`;
  }
}

const person = new Person("Alice", 30);
console.log(person.greet()); // "Hello, my name is Alice"

// Class with access modifiers
class BankAccount {
  public accountNumber: string;    // Accessible everywhere
  private balance: number;          // Accessible only within class
  protected accountType: string;    // Accessible in class and subclasses

  constructor(accountNumber: string, initialBalance: number) {
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
    this.accountType = "Savings";
  }

  // Public method
  public deposit(amount: number): void {
    if (amount > 0) {
      this.balance += amount;
    }
  }

  // Public method
  public getBalance(): number {
    return this.balance;
  }

  // Private method
  private calculateInterest(): number {
    return this.balance * 0.05;
  }
}

// Shorthand constructor
class User {
  constructor(
    public id: number,
    public name: string,
    private email: string
```

```
    return this.email;
  }
}

const user2 = new User(1, "Bob", "bob@example.com");

// Class inheritance
class Animal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log("Some generic sound");
  }
}

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name); // Call parent constructor
  }

  makeSound(): void {
    console.log("Woof! Woof!");
  }

  fetch(): void {
    console.log(`${this.name} is fetching the ball`);
  }
}

const dog = new Dog("Buddy", "Golden Retriever");
dog.makeSound(); // "Woof! Woof!"
dog.fetch();     // "Buddy is fetching the ball"

// Abstract class
abstract class Shape {
  constructor(public color: string) {}

  abstract calculateArea(): number;

  describe(): string {
    return `This is a ${this.color} shape with area ${this.calculateArea()}
  }
}

class Rectangle extends Shape {
```

```
  calculateArea(): number {
    return this.width * this.height;
  }
}

const rectangle = new Rectangle("blue", 10, 5);
console.log(rectangle.describe()); // "This is a blue shape with area 50"

// Implementing interfaces
interface Drivable {
  speed: number;
  accelerate(amount: number): void;
  brake(): void;
}

class Car implements Drivable {
  speed: number = 0;

  accelerate(amount: number): void {
    this.speed += amount;
  }

  brake(): void {
    this.speed = 0;
  }
}
```

Arrays, Objects, and Destructuring

Arrays

```
 // Array declaration
const numbers: number[] = [1, 2, 3, 4, 5];
const names: Array<string> = ["Alice", "Bob", "Charlie"];

// Array methods
const fruits = ["apple", "banana", "orange"];
```

```
// Pop (remove from end)
const lastFruit = fruits.pop();

// Shift (remove from start)
const firstFruit = fruits.shift();

// Unshift (add to start)
fruits.unshift("mango");

// Slice (extract portion)
const sliced = fruits.slice(1, 3);

// Splice (remove/add items)
fruits.splice(1, 1, "kiwi"); // Remove 1 item at index 1, add "kiwi"

// Array iteration methods
const nums = [1, 2, 3, 4, 5];

// forEach
nums.forEach((num) => console.log(num));

// map (transform array)
const doubled = nums.map((num) => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

// filter (filter array)
const evenNumbers = nums.filter((num) => num % 2 === 0);
console.log(evenNumbers); // [2, 4]

// reduce (reduce to single value)
const sum = nums.reduce((acc, num) => acc + num, 0);
console.log(sum); // 15

// find (find first matching element)
const found = nums.find((num) => num > 3);
console.log(found); // 4

// some (check if any element matches)
const hasEven = nums.some((num) => num % 2 === 0);
console.log(hasEven); // true

// every (check if all elements match)
const allPositive = nums.every((num) => num > 0);
console.log(allPositive); // true
```

```
  id: number;
  name: string;
  price: number;
}

const products: Product[] = [
  { id: 1, name: "Laptop", price: 1000 },
  { id: 2, name: "Mouse", price: 25 },
  { id: 3, name: "Keyboard", price: 75 }
];

// Filter expensive products
const expensiveProducts = products.filter((p) => p.price > 50);

// Get product names
const productNames = products.map((p) => p.name);

// Calculate total price
const totalPrice = products.reduce((total, p) => total + p.price, 0);
```

Objects

```
 // Object literal
const user = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  age: 30
};

// Access properties
console.log(user.name);     // Dot notation
console.log(user["email"]); // Bracket notation

// Modify properties
user.age = 31;
user["email"] = "alice.new@example.com";

// Add new properties
(user as any).phone = "123-456-7890";

// Delete properties
```

```
const person = {
  name: "Bob",
  age: 25,
  greet() {
    return `Hello, I'm ${this.name}`;
  },
  celebrateBirthday() {
    this.age++;
  }
};

console.log(person.greet()); // "Hello, I'm Bob"
person.celebrateBirthday();
console.log(person.age); // 26

// Object.keys, Object.values, Object.entries
const config = {
  host: "localhost",
  port: 3000,
  debug: true
};

console.log(Object.keys(config));   // ["host", "port", "debug"]
console.log(Object.values(config)); // ["localhost", 3000, true]
console.log(Object.entries(config)); // [["host", "localhost"], ["port", 3

// Object spread operator
const defaults = { theme: "light", fontSize: 14 };
const userPrefs = { fontSize: 16, language: "en" };
const finalConfig = { ...defaults, ...userPrefs };
console.log(finalConfig);
// { theme: "light", fontSize: 16, language: "en" }
```

Destructuring

Array Destructuring:

```
 // Basic array destructuring
const colors = ["red", "green", "blue"];
const [first, second, third] = colors;
console.log(first);  // "red"
console.log(second); // "green"
```

```
console.log(tertiary); // "blue"

// Rest operator
const numbers = [1, 2, 3, 4, 5];
const [one, two, ...rest] = numbers;
console.log(one);  // 1
console.log(two);  // 2
console.log(rest); // [3, 4, 5]

// Default values
const [a, b, c = 0] = [1, 2];
console.log(c); // 0

// Swapping variables
let x = 1;
let y = 2;
[x, y] = [y, x];
console.log(x, y); // 2, 1
```

Object Destructuring:

```
 // Basic object destructuring
const user = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  age: 30
};

const { name, email } = user;
console.log(name);  // "Alice"
console.log(email); // "alice@example.com"

// Rename variables
const { name: userName, age: userAge } = user;
console.log(userName); // "Alice"
console.log(userAge);  // 30

// Default values
const { name, phone = "N/A" } = user;
console.log(phone); // "N/A"

// Nested destructuring
```

```
  address: {
    street: "123 Main St",
    city: "New York",
    country: "USA"
  }
};

const {
  name: empName,
  address: { city, country }
} = employee;

console.log(empName); // "Bob"
console.log(city);    // "New York"
console.log(country); // "USA"

// Rest in objects
const settings = {
  theme: "dark",
  fontSize: 16,
  language: "en",
  notifications: true
};

const { theme, ...otherSettings } = settings;
console.log(theme);         // "dark"
console.log(otherSettings);
// { fontSize: 16, language: "en", notifications: true }

// Function parameter destructuring
interface Config {
  host: string;
  port: number;
  debug?: boolean;
}

function connectDatabase({ host, port, debug = false }: Config) {
  console.log(`Connecting to ${host}:${port}`);
  if (debug) {
    console.log("Debug mode enabled");
  }
}

connectDatabase({ host: "localhost", port: 5432, debug: true });
```

VS Code Debugging Basics

Setting Up Debug Configuration

1. Create launch.json

Create `.vscode/launch.json` :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug TypeScript",
      "program": "${workspaceFolder}/src/index.ts",
      "preLaunchTask": "tsc: build - tsconfig.json",
      "outFiles": ["${workspaceFolder}/dist/**/*.js"],
      "sourceMaps": true
    },
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Current File",
      "program": "${file}",
      "skipFiles": ["<node_internals>/**"],
      "console": "integratedTerminal"
    }
  ]
}
```

Breakpoints

Setting Breakpoints:

1. Click in the gutter (left margin) next to line number
2. Or press F9 on the line
3. Red dot appears indicating breakpoint

- **Conditional Breakpoint**: Right-click → Add Conditional Breakpoint

```
Example: count > 5
```

- **Logpoint**: Right-click → Add Logpoint (logs without stopping)

```
Example: Count is {count}
```

Debug Controls

Toolbar Buttons:

- **Continue (F5)**: Resume execution
- **Step Over (F10)**: Execute current line, don't go into functions
- **Step Into (F11)**: Go into function calls
- **Step Out (Shift+F11)**: Exit current function
- **Restart (Ctrl+Shift+F5)**: Restart debugging
- **Stop (Shift+F5)**: Stop debugging

Watch Variables

Debug Sidebar Panels:

1. Variables Panel:

- Shows local variables
- Shows function arguments
- Expand objects to inspect properties

2. Watch Panel:

- Add expressions to watch
- Example: `user.email` , `count > 5`

- Click to navigate to different stack frames

4. Breakpoints Panel:

- List of all breakpoints
- Enable/disable breakpoints

Debug Console

Access with `Ctrl+Shift+Y` or Debug → Debug Console

Evaluate Expressions:

```
 // In debug console while paused
> count
10
> count * 2
20
> user.name
"Alice"
> JSON.stringify(user, null, 2)
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com"
}
```

Debugging Example

Example Code:

```
 // src/calculator.ts
export function calculateTotal(items: number[]): number {
  let total = 0;

  for (let i = 0; i < items.length; i++) {
    const item = items[i];
    total += item;
```

```
  return total;
}

// src/index.ts
import { calculateTotal } from './calculator';

const prices = [10, 20, 30, 40];
const total = calculateTotal(prices);
console.log(`Final total: ${total}`);
```

Debugging Steps:

1. Set breakpoint on line with `total += item`
2. Press F5 to start debugging
3. When breakpoint hits, inspect:
   - `item` variable (current value)
   - `total` variable (running sum)
   - `i` variable (loop index)
   - `items` array (all values)
4. Press F10 to step through loop
5. Watch how `total` changes with each iteration

Common Debugging Scenarios

1. Inspect Function Arguments:

```
function processUser(user: User) {
  // Set breakpoint here
  console.log(user.name);
  // Inspect user object in Variables panel
}
```

2. Debug Async Functions:

```
  // Set breakpoint after await
  const data = await response.json();
  return data;
}
```

3. Conditional Breakpoint:

```
for (let i = 0; i < 100; i++) {
  // Right-click → Add Conditional Breakpoint
  // Condition: i === 50
  console.log(i);
}
```

4. Debug Test Code:

```
// Add this to launch.json for debugging tests
{
  "type": "node",
  "request": "launch",
  "name": "Debug Tests",
  "program": "${workspaceFolder}/node_modules/.bin/jest",
  "args": ["--runInBand"],
  "console": "integratedTerminal"
}
```

Practice Exercises

Exercise 1: TypeScript Basics

Task: Create a User Management System

```
// Define interfaces
interface User {
```

```
  role: "admin" | "user" | "guest";
  createdAt: Date;
}

// Create user array
const users: User[] = [];

// Add functions
function addUser(name: string, email: string, role: User["role"]): User {
  // Implementation
}

function getUserById(id: number): User | undefined {
  // Implementation
}

function updateUser(id: number, updates: Partial<User>): boolean {
  // Implementation
}

function deleteUser(id: number): boolean {
  // Implementation
}

function getUsersByRole(role: User["role"]): User[] {
  // Implementation
}

// Test your implementation
const user1 = addUser("Alice", "alice@example.com", "admin");
const user2 = addUser("Bob", "bob@example.com", "user");
console.log(getUsersByRole("admin"));
```

Exercise 2: Array and Object Manipulation

Task: Create a Product Inventory System

```
interface Product {
  id: number;
  name: string;
  category: string;
  price: number;
  inStock: boolean;
```

```
const inventory: Product[] = [
  { id: 1, name: "Laptop", category: "Electronics", price: 1000, inStock:
  { id: 2, name: "Mouse", category: "Electronics", price: 25, inStock: tru
  { id: 3, name: "Desk", category: "Furniture", price: 300, inStock: false
  { id: 4, name: "Chair", category: "Furniture", price: 150, inStock: true
];

// Tasks:
// 1. Get all products in stock
const inStockProducts = // Your code

// 2. Get products by category
function getProductsByCategory(category: string): Product[] {
  // Your code
}

// 3. Calculate total inventory value
const totalValue = // Your code

// 4. Get most expensive product
const mostExpensive = // Your code

// 5. Update product quantity
function updateQuantity(id: number, quantity: number): boolean {
  // Your code
}

// 6. Get low stock products (quantity < 5)
const lowStockProducts = // Your code
```

Exercise 3: Function Types and Callbacks

Task: Create a Calculator with Higher-Order Functions

```
type Operation = (a: number, b: number) => number;

const add: Operation = (a, b) => a + b;
const subtract: Operation = (a, b) => a - b;
const multiply: Operation = (a, b) => a * b;
const divide: Operation = (a, b) => {
  if (b === 0) throw new Error("Division by zero");
  return a / b;
};
```

```
}

// Create a function that returns a function
function createCalculator(operation: Operation) {
  return (a: number, b: number) => {
    console.log(`Calculating: ${a} and ${b}`);
    return operation(a, b);
  };
}

// Test
const adder = createCalculator(add);
console.log(adder(5, 3)); // 8

// Advanced: Chain operations
function chainOperations(
  initial: number,
  operations: Array<{ op: Operation; value: number }>
): number {
  // Your code
}

// Test chaining
const result = chainOperations(10, [
  { op: add, value: 5 },
  { op: multiply, value: 2 },
  { op: subtract, value: 10 }
]);
console.log(result); // Should be 20
```

Exercise 4: Classes and OOP

Task: Create a Library Management System

```
 // Book class
class Book {
  constructor(
    public id: number,
    public title: string,
    public author: string,
    public isbn: string,
    private available: boolean = true
  ) {}
```

```
      this.available = false;
      return true;
    }
    return false;
  }

  return(): void {
    this.available = true;
  }

  isAvailable(): boolean {
    return this.available;
  }
}

// Library class
class Library {
  private books: Book[] = [];

  addBook(book: Book): void {
    // Your code
  }

  findBookByTitle(title: string): Book | undefined {
    // Your code
  }

  findBooksByAuthor(author: string): Book[] {
    // Your code
  }

  checkoutBook(id: number): boolean {
    // Your code
  }

  returnBook(id: number): boolean {
    // Your code
  }

  getAvailableBooks(): Book[] {
    // Your code
  }
}

// Test
```

```
console.log(library.checkoutBook(1)); // true
console.log(library.getAvailableBooks().length); // 1
console.log(library.returnBook(1)); // true
```

Exercise 5: Async Programming Practice

Task: Create an Async Data Fetcher with Error Handling

```
 interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
}

interface User {
  id: number;
  name: string;
  email: string;
}

// Simulate API call
function mockApiCall<T>(data: T, delay: number = 1000): Promise<ApiRespons
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({
        data,
        status: 200,
        message: "Success"
      });
    }, delay);
  });
}

// Simulate API error
function mockApiError(delay: number = 1000): Promise<never> {
  return new Promise((_, reject) => {
    setTimeout(() => {
      reject(new Error("API Error"));
    }, delay);
  });
}
```

```
  // Your code with error handling
}

// 2. Create async function to fetch multiple users
async function fetchMultipleUsers(ids: number[]): Promise<User[]> {
  // Your code - use Promise.all
}

// 3. Create async function with retry logic
async function fetchWithRetry<T>(
  fetchFn: () => Promise<T>,
  maxRetries: number = 3
): Promise<T> {
  // Your code
}

// 4. Create async function with timeout
async function fetchWithTimeout<T>(
  fetchFn: () => Promise<T>,
  timeout: number = 5000
): Promise<T> {
  // Your code - use Promise.race
}

// Test your implementations
async function runTests() {
  try {
    const user = await fetchUser(1);
    console.log("User:", user);

    const users = await fetchMultipleUsers([1, 2, 3]);
    console.log("Users:", users);

    const result = await fetchWithRetry(() => fetchUser(1));
    console.log("Result:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

runTests();
```

Exercise 6: Debugging Challenge

```
// This code has several bugs - find and fix them using VS Code debugger

interface Product {
  id: number;
  name: string;
  price: number;
}

function calculateTotal(products: Product[]): number {
  let total = 0;

  for (let i = 0; i <= products.length; i++) {
    total += products[i].price;
  }

  return total;
}

function applyDiscount(total: number, discountPercent: number): number {
  return total - (total * discountPercent);
}

function formatPrice(price: number): string {
  return "$" + price.toFixed(2);
}

const products: Product[] = [
  { id: 1, name: "Laptop", price: 1000 },
  { id: 2, name: "Mouse", price: 25 },
  { id: 3, name: "Keyboard", price: 75 }
];

const total = calculateTotal(products);
const discounted = applyDiscount(total, 10);
const formatted = formatPrice(discounted);

console.log("Total:", formatted);

// Expected output: Total: $990.00
// Use debugger to find and fix bugs
```

Debugging Steps:

1. Set breakpoints in calculateTotal

4. Fix the bug in loop condition
5. Set breakpoint in applyDiscount
6. Check discount calculation
7. Fix discount formula

Summary

In Day 1, you learned:

âœ… **Quality Engineering Fundamentals**

- QE vs QA differences
- Shift-left testing approach
- Role and responsibilities of QE engineers

âœ… **Manual vs Automation Testing**

- When to use each approach
- Advantages and disadvantages
- Cost-benefit analysis

âœ… **Test Automation Pyramid**

- Unit tests (60%)
- Integration tests (30%)
- E2E tests (10%)
- Anti-patterns to avoid

âœ… **Development Environment**

- Node.js and npm setup
- VS Code configuration
- Git basics

- Variables and data types
- Operators
- Control flow
- Arrays and objects

âœ… **TypeScript Fundamentals**

- Type annotations
- Type inference
- Union and literal types
- Type aliases

âœ… **TypeScript Functions**

- Function declarations
- Arrow functions
- Function types
- Async functions

âœ… **OOP in TypeScript**

- Interfaces
- Classes
- Inheritance
- Access modifiers

âœ… **Data Manipulation**

- Array methods (map, filter, reduce)
- Object operations
- Destructuring

âœ… **Debugging**

- Debug console
- Call stack

Next Steps

- Practice TypeScript coding exercises
- Set up your development environment
- Familiarize yourself with VS Code debugging
- Complete all practice exercises
- Review any concepts you found challenging

Resources

- TypeScript Official Documentation
- MDN Web Docs - JavaScript
- VS Code Debugging Guide
- Node.js Documentation

End of Day 1 Documentation

← Getting Started

Async JavaScript & Node.js →