Case Study 1: Setting Up a TypeScript Project for a Web Application

Objective: Learn how to set up a TypeScript environment for a real-world web application project.

Scenario: A web development company is transitioning from JavaScript to TypeScript for a new project. The team needs to set up a TypeScript environment, compile TypeScript code, and ensure that the setup works correctly.

Steps:

1. Install TypeScript globally using npm.

2. Initialize a new project using `npm init -y`.

3. Create a `tsconfig.json` file with basic configurations.

4. Create a `src` directory and add a `main.ts` file.

5. Write a simple TypeScript program and compile it.

Solution:

```sh
```

```
npm install -g typescript
npm init -y
```

Create `tsconfig.json`:

```json
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  }
}
```

Create `src/main.ts`:

```typescript
console.log("Hello, TypeScript!");
```

Compile:

```sh
tsc
```

Case Study 2: Using TypeScript for a Financial Calculation Module

Objective: Implement a financial calculation module using TypeScript to ensure type safety and reduce errors.

Scenario: A fintech startup is building a module to calculate loan payments. They want to use TypeScript to ensure that all inputs and outputs are correctly typed to avoid calculation errors.

Steps:

1. Create a new TypeScript file `loanCalculator.ts`.

2. Define a function to calculate monthly loan payments.

3. Ensure the function uses strict typing for all parameters and return values.

Solution:

```typescript
function calculateMonthlyPayment(principal: number, annualRate: number, years: number): number {
  let monthlyRate = annualRate / 12 / 100;
  let payments = years * 12;
  return principal * monthlyRate / (1 - Math.pow(1 + monthlyRate, -payments));
}

let principal = 100000;
let annualRate = 5;
let years = 30;

console.log("Monthly Payment:", calculateMonthlyPayment(principal, annualRate, years));
```

Case Study 3: TypeScript in a Content Management System (CMS)

Objective: Use TypeScript to manage data types and ensure data integrity in a CMS.

Scenario: A CMS requires strict data type management for articles, authors, and tags. Implement TypeScript interfaces to define the structure of these entities and ensure type safety.

Steps:

1. Define interfaces for Article, Author, and Tag.

2. Create a function to add new articles, ensuring it adheres to the defined interfaces.

Solution:

```typescript
interface Author {
  name: string;
  email: string;
```

```
}

interface Tag {
  name: string;
}

interface Article {
  title: string;
  content: string;
  author: Author;
  tags: Tag[];
}

function addArticle(article: Article): void {
  console.log("Article added:", article);
}

const newArticle: Article = {
  title: "TypeScript in CMS",
  content: "This article explains how to use TypeScript in a CMS.",
    author: { name: "John Doe", email: "john@example.com" },
    tags: [{ name: "TypeScript" }, { name: "CMS" }]
};

addArticle(newArticle);
```

Case Study 4: TypeScript for Frontend Validation in a Form

Objective: Use TypeScript to perform frontend validation in a user registration form.

Scenario: A user registration form requires validation for user inputs. Implement TypeScript to validate the inputs before submitting the form.

Steps:

1. Create a `register.ts` file.
2. Define an interface for the form inputs.
3. Implement a function to validate the inputs.

Solution:

```typescript
interface UserRegistration {
    username: string;
    email: string;
    password: string;
}

function validateRegistration(user: UserRegistration): boolean {
    if (!user.username || !user.email || !user.password) {
        console.log("All fields are required.");
        return false;
    }
    if (!user.email.includes("@")) {
        console.log("Invalid email.");
        return false;
    }
    if (user.password.length < 6) {
        console.log("Password must be at least 6 characters long.");
        return false;
    }
    console.log("Validation successful.");
    return true;
}

const newUser: UserRegistration = {
    username: "johndoe",
    email: "john@example.com",
    password: "password123"
};

validateRegistration(newUser);
```

Case Study 5: TypeScript in an E-commerce Application

Objective: Implement TypeScript to manage product data in an e-commerce application.

Scenario: An e-commerce application needs to manage product data, including product details and pricing. Use TypeScript interfaces and classes to define and manage this data.

Steps:

1. Define interfaces for Product and Category.
2. Create a class to manage the product data.
3. Implement methods to add and retrieve product data.

Solution:

```typescript
interface Category {
   id: number;
   name: string;
}

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

class ProductManager {
  private products: Product[] = [];

  addProduct(product: Product): void {
    this.products.push(product);
  }

  getProducts(): Product[] {
    return this.products;
  }
}

const electronics: Category = { id: 1, name: "Electronics" };
const phone: Product = { id: 1, name: "iPhone", price: 999, category: electronics };

const productManager = new ProductManager();
productManager.addProduct(phone);
console.log(productManager.getProducts());
```

Case Study 6: Implementing TypeScript in a Chat Application

Objective: Use TypeScript to manage message data and ensure type safety in a chat application.

Scenario: A chat application needs to handle messages and user data. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for User and Message.
2. Create a class to handle message data.
3. Implement methods to add and retrieve messages.

Solution:

```typescript
interface User {
  id: number;
  username: string;
}

interface Message {
  id: number;
  content: string;
  sender: User;
  timestamp: Date;
}

class Chat {
  private messages: Message[] = [];

  sendMessage(message: Message): void {
    this.messages.push(message);
  }
```

```
    getMessages(): Message[] {
        return this.messages;
    }
}

const user: User = { id: 1, username: "john_doe" };
const message: Message = { id: 1, content: "Hello, world!", sender: user, timestamp: new Date() };

const chat = new Chat();
chat.sendMessage(message);
console.log(chat.getMessages());
```

Case Study 7: TypeScript for a Blogging Platform

Objective: Use TypeScript to manage posts and comments in a blogging platform.

Scenario: A blogging platform needs to manage posts and comments. Implement TypeScript interfaces and classes to handle these entities.

Steps:

1. Define interfaces for Post and Comment.

2. Create classes to manage posts and comments.

3. Implement methods to add and retrieve posts and comments.

Solution:

```typescript
interface Comment {
   id: number;
   content: string;
   author: string;
   postId: number;
}

interface Post {
   id: number;
```

```
  title: string;
  content: string;
  author: string;
  comments: Comment[];
}

class Blog {
  private posts: Post[] = [];

  addPost(post: Post): void {
    this.posts.push(post);
  }

  getPosts(): Post[] {
    return this.posts;
  }

  addComment(postId: number, comment:
Comment): void {
    const post = this.posts.find(p => p.id ===
postId);
```

```
        if (post) {
            post.comments.push(comment);
        }
    }
}

const newPost: Post = {
    id: 1,
    title: "My First Blog Post",
    content: "This is the content of my first blog post.",
    author: "Jane Doe",
    comments: []
};

const newComment: Comment = {
    id: 1,
    content: "Great post!",
    author: "John Doe",
    postId: 1
};
```

```
const blog = new Blog();
blog.addPost(newPost);
blog.addComment(1, newComment);
console.log(blog.getPosts());
```

Case Study 8: TypeScript in a Task Management System

Objective: Use TypeScript to manage tasks and projects in a task management system.

Scenario: A task management system needs to handle tasks and projects. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Task and Project.
2. Create classes to manage tasks and projects.
3. Implement methods to add and retrieve tasks and projects.

Solution:

```typescript
interface Task {
  id: number;
  title: string;
  description: string;
  completed: boolean;
}

interface Project {
  id: number;
  name: string;
  tasks: Task[];
}

class TaskManager {
  private tasks: Task[] = [];

  addTask(task: Task): void {
    this.tasks.push(task);
  }
  getTasks(): Task[] {
    return this.tasks;
  }

  markTaskComplete(taskId: number): void {
    const task =

 this.tasks.find(t => t.id === taskId);
    if (task) {
      task.completed = true;
    }
  }
}

class ProjectManager {
  private projects: Project[] = [];

  addProject(project: Project): void {
    this.projects.push(project);
  }
```

```
    getProjects(): Project[] {
        return this.projects;
    }

    addTaskToProject(projectId: number, task: Task): void {
        const project = this.projects.find(p => p.id === projectId);
        if (project) {
            project.tasks.push(task);
        }
    }
}

const task1: Task = { id: 1, title: "Setup TypeScript", description: "Install and configure TypeScript.", completed: false };
const project: Project = { id: 1, name: "TypeScript Project", tasks: [] };

const taskManager = new TaskManager();
```

```
taskManager.addTask(task1);

const projectManager = new ProjectManager();
projectManager.addProject(project);
projectManager.addTaskToProject(1, task1);

console.log(projectManager.getProjects());
```

Case Study 9: TypeScript in a Real-time Collaboration Tool

Objective: Use TypeScript to manage users and sessions in a real-time collaboration tool.

Scenario: A real-time collaboration tool needs to handle user sessions and document sharing. Implement TypeScript interfaces and classes to manage these entities.

Steps:
1. Define interfaces for User and Session.

2. Create classes to manage users and sessions.

3. Implement methods to add and retrieve users and sessions.

Solution:

```typescript
interface User {
  id: number;
  username: string;
  email: string;
}

interface Session {
  id: number;
  documentId: number;
  participants: User[];
}

class UserManager {
  private users: User[] = [];
```

```
    addUser(user: User): void {
        this.users.push(user);
    }

    getUsers(): User[] {
        return this.users;
    }
}

class SessionManager {
    private sessions: Session[] = [];

    addSession(session: Session): void {
        this.sessions.push(session);
    }

    getSessions(): Session[] {
        return this.sessions;
    }
```

```
  addUserToSession(sessionId: number, user: User): void {
    const session = this.sessions.find(s => s.id === sessionId);
    if (session) {
      session.participants.push(user);
    }
  }
}

const user: User = { id: 1, username: "john_doe", email: "john@example.com" };
const session: Session = { id: 1, documentId: 123, participants: [] };

const userManager = new UserManager();
userManager.addUser(user);

const sessionManager = new SessionManager();
sessionManager.addSession(session);
sessionManager.addUserToSession(1, user);
```

console.log(sessionManager.getSessions());
~~~

Case Study 10: TypeScript for API Data Handling

Objective: Use TypeScript to fetch and handle data from an API.

Scenario: An application needs to fetch data from a public API and process it. Implement TypeScript to handle the fetched data and ensure type safety.

Steps:

1. Define an interface for the API response data.

2. Create a function to fetch data from the API.

3. Process and print the fetched data.

Solution:

```typescript
interface ApiResponse {
~~~

```
  userId: number;
  id: number;
  title: string;
  completed: boolean;
}

async function fetchData(): Promise<void> {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const data: ApiResponse = await response.json();
    console.log("Fetched Data:", data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

fetchData();
```

Case Study 11: TypeScript in a Weather Application

Objective: Use TypeScript to fetch and display weather data.

Scenario: A weather application needs to fetch weather data from an API and display it to the user. Implement TypeScript to ensure type safety and manage the fetched data.

Steps:

1. Define an interface for the weather data.

2. Create a function to fetch weather data from an API.

3. Display the fetched data to the user.

Solution:

```typescript
interface WeatherData {
  temperature: number;
  humidity: number;
  description: string;
}

async function getWeather(city: string): Promise<WeatherData> {
  const response = await fetch(`https://api.example.com/weather?city=${city}`);
  const data = await response.json();
  return {
    temperature: data.temp,
    humidity: data.humidity,
    description: data.weather[0].description
  };
}

getWeather("London").then(weather => {
  console.log(`Temperature: ${weather.temperature}`);
  console.log(`Humidity: ${weather.humidity}`);
  console.log(`Description: ${weather.description}`);
});
```

Case Study 12: TypeScript in a Healthcare Application

Objective: Use TypeScript to manage patient data in a healthcare application.

Scenario: A healthcare application needs to manage patient data, including personal information and medical history. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Patient and MedicalRecord.
2. Create classes to manage patient data and medical records.
3. Implement methods to add and retrieve patient data and medical records.

Solution:

```typescript
interface MedicalRecord {
  id: number;
  diagnosis: string;
  treatment: string;
  date: Date;
}

interface Patient {
  id: number;
  name: string;
  age: number;
  medicalRecords: MedicalRecord[];
}

class PatientManager {
  private patients: Patient[] = [];

  addPatient(patient: Patient): void {
    this.patients.push(patient);
  }
  getPatients(): Patient[] {
    return this.patients;
  }

  addMedicalRecord(patientId: number, record: MedicalRecord): void {
    const patient = this.patients.find(p => p.id === patientId);
    if (patient) {
      patient.medicalRecords.push(record);
    }
  }
}

const patient: Patient = { id: 1, name: "John Doe", age: 30, medicalRecords: [] };
const medicalRecord: MedicalRecord = { id: 1, diagnosis: "Flu", treatment: "Rest and hydration", date: new Date() };

const patientManager = new PatientManager();
```

```
patientManager.addPatient(patient);
patientManager.addMedicalRecord(1, medicalRecord);

console.log(patientManager.getPatients());
```

Case Study 13: TypeScript in a Social Media Application

Objective: Use TypeScript to manage posts and user interactions in a social media application.

Scenario: A social media application needs to handle user posts, likes, and comments. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for User, Post, and Comment.
2. Create classes to manage posts, likes, and comments.

3. Implement methods to add and retrieve posts, likes, and comments.

Solution:

```typescript
interface User {
  id: number;
  username: string;
}

interface Comment {
  id: number;
  content: string;
  author: User;
  postId: number;
}

interface Post {
  id: number;
  content: string;
  author: User;
```

```
    likes: number;
    comments: Comment[];
}

class SocialMedia {
    private posts: Post[] = [];

    addPost(post: Post): void {
        this.posts.push(post);
    }

    getPosts(): Post[] {
        return this.posts;
    }

    likePost(postId: number): void {
        const post = this.posts.find(p => p.id === postId);
        if (post) {
            post.likes++;
        }
```

```
  }

  addComment(postId: number, comment: Comment): void {
    const post = this.posts.find(p => p.id === postId);
    if (post) {
      post.comments.push(comment);
    }
  }
}

const user: User = { id: 1, username: "john_doe" };
const post: Post = { id: 1, content: "Hello, world!", author: user, likes: 0, comments: [] };
const comment: Comment = { id: 1, content: "Nice post!", author: user, postId: 1 };

const socialMedia = new SocialMedia();
socialMedia.addPost(post);
socialMedia.likePost(1);
socialMedia.addComment(1, comment);
```

console.log(socialMedia.getPosts());
```

Case Study 14: TypeScript in a Learning Management System (LMS)

Objective: Use TypeScript to manage courses and student enrollments in an LMS.

Scenario: A learning management system needs to handle courses and student enrollments. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Course and Student.

2. Create classes to manage courses and student enrollments.

3. Implement methods to add and retrieve courses and student enrollments.
```

**Solution:**

```typescript
interface Course {
    id: number;
    title: string;
    description: string;
}

interface Student {
    id: number;
    name: string;
    courses: Course[];
}

class CourseManager {
    private courses: Course[] = [];

    addCourse(course: Course): void {
        this.courses.push(course);
    }
    getCourses(): Course[] {
        return this.courses;
    }
}

class StudentManager {
    private students: Student[] = [];

    addStudent(student: Student): void {
        this.students.push(student);
    }

    getStudents(): Student[] {
        return this.students;
    }

    enrollStudent(courseId:

 number, studentId: number): void {
        const student = this.students.find(s => s.id ===
studentId);
```

```
        const course = this.courses.find(c => c.id === courseId);
        if (student && course) {
            student.courses.push(course);
        }
    }
}

const course: Course = { id: 1, title: "TypeScript Basics", description: "Learn the basics of TypeScript." };
const student: Student = { id: 1, name: "John Doe", courses: [] };

const courseManager = new CourseManager();
courseManager.addCourse(course);

const studentManager = new StudentManager();
studentManager.addStudent(student);
studentManager.enrollStudent(1, 1);

console.log(studentManager.getStudents());
```

~~~

Case Study 15: TypeScript in an Inventory Management System

Objective: Use TypeScript to manage inventory and track stock levels.

Scenario: An inventory management system needs to handle products and track stock levels. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Product and Inventory.

2. Create classes to manage inventory and track stock levels.

3. Implement methods to add and retrieve products and update stock levels.

Solution:

```typescript
~~~

```
interface Product {
    id: number;
    name: string;
    quantity: number;
}

class Inventory {
    private products: Product[] = [];

    addProduct(product: Product): void {
        this.products.push(product);
    }

    getProducts(): Product[] {
        return this.products;
    }

    updateStock(productId: number, quantity: number): void {
        const product = this.products.find(p => p.id === productId);
        if (product) {
            product.quantity += quantity;
        }
    }
}

const product: Product = { id: 1, name: "Laptop", quantity: 10 };

const inventory = new Inventory();
inventory.addProduct(product);
inventory.updateStock(1, 5);

console.log(inventory.getProducts());
```

Case Study 16: TypeScript in a Travel Booking System

Objective: Use TypeScript to manage bookings and customer data in a travel booking system.

Scenario: A travel booking system needs to handle bookings and customer data. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Booking and Customer.

2. Create classes to manage bookings and customer data.

3. Implement methods to add and retrieve bookings and customer data.

Solution:

```typescript
interface Customer {
  id: number;
  name: string;
  email: string;
}

interface Booking {
  id: number;
  customer: Customer;
  destination: string;
  date: Date;
}

class BookingManager {
  private bookings: Booking[] = [];

  addBooking(booking: Booking): void {
    this.bookings.push(booking);
  }

  getBookings(): Booking[] {
    return this.bookings;
  }
}

const customer: Customer = { id: 1, name: "Jane Doe", email: "jane@example.com" };
const booking: Booking = { id: 1, customer: customer, destination: "Paris", date: new Date() };

const bookingManager = new BookingManager();
bookingManager.addBooking(booking);

console.log(bookingManager.getBookings());
```

Case Study 17: TypeScript in a Restaurant Management System

Objective: Use TypeScript to manage menu items and orders in a restaurant management system.

Scenario: A restaurant management system needs to handle menu items and orders. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for MenuItem and Order.

2. Create classes to manage menu items and orders.

3. Implement methods to add and retrieve menu items and orders.

Solution:

```typescript
interface MenuItem {
  id: number;
  name: string;
  price: number;
}

interface Order {
  id: number;
  items: MenuItem[];
  total: number;
}

class Menu {
  private items: MenuItem[] = [];
  addItem(item: MenuItem): void {
    this.items.push(item);
  }

  getItems(): MenuItem[] {
    return this.items;
  }
}

class OrderManager {
  private orders: Order[] = [];

  addOrder(order: Order): void {
    this.orders.push(order);
  }

  getOrders(): Order[] {
    return this.orders;
  }
}
```

```
const item: MenuItem = { id: 1, name: "Burger", price: 5.99 };
const order: Order = { id: 1, items: [item], total: 5.99 };

const menu = new Menu();
menu.addItem(item);

const orderManager = new OrderManager();
orderManager.addOrder(order);

console.log(orderManager.getOrders());
```

Case Study 18: TypeScript in a Real Estate Application

Objective: Use TypeScript to manage property listings and customer inquiries in a real estate application.

Scenario: A real estate application needs to handle property listings and customer inquiries. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Property and Inquiry.

2. Create classes to manage property listings and customer inquiries.

3. Implement methods to add and retrieve property listings and customer inquiries.

Solution:

```typescript
interface Property {
  id: number;
  address: string;
  price: number;
  description: string;
}

interface Inquiry {
    id: number;
    customerName: string;
    propertyId: number;
    message: string;
}

class PropertyManager {
    private properties: Property[] = [];

    addProperty(property: Property): void {
        this.properties.push(property);
    }

    getProperties(): Property[] {
        return this.properties;
    }
}

class InquiryManager {
    private inquiries: Inquiry[] = [];
```

```
  addInquiry(inquiry: Inquiry): void {
    this.inquiries.push(inquiry);
  }

  getInquiries(): Inquiry[] {
    return this.inquiries;
  }
}

const property: Property = { id: 1, address: "123 Main St", price: 250000, description: "Beautiful 3-bedroom house" };
const inquiry: Inquiry = { id: 1, customerName: "John Doe", propertyId: 1, message: "Interested in this property" };

const propertyManager = new PropertyManager();
propertyManager.addProperty(property);

const inquiryManager = new InquiryManager();
inquiryManager.addInquiry(inquiry);
```

```
console.log(propertyManager.getProperties());
console.log(inquiryManager.getInquiries());
```

Case Study 19: TypeScript in a Logistics Management System

Objective: Use TypeScript to manage shipments and track delivery status in a logistics management system.

Scenario: A logistics management system needs to handle shipments and track their delivery status. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Shipment and DeliveryStatus.

2. Create classes to manage shipments and track delivery status.

3. Implement methods to add and retrieve shipments and update delivery status.

Solution:

```typescript
interface DeliveryStatus {
  id: number;
  status: string;
  updatedAt: Date;
}

interface Shipment {
  id: number;
  description: string;
  status: DeliveryStatus[];
}

class ShipmentManager {
  private shipments: Shipment[] = [];

  addShipment(shipment: Shipment): void {
    this.shipments.push(shipment);
  }
  getShipments(): Shipment[] {
    return this.shipments;
  }

  updateStatus(shipmentId: number, status:
DeliveryStatus): void {
    const shipment = this.shipments.find(s => s.id
=== shipmentId);
    if (shipment) {
      shipment.status.push(status);
    }
  }
}

const shipment: Shipment = { id: 1, description:
"Electronics", status: [] };
const status: DeliveryStatus = { id: 1, status:
"Shipped", updatedAt: new Date() };

const shipmentManager = new ShipmentManager();
shipmentManager.addShipment(shipment);
shipmentManager.updateStatus(1, status);
```

```
console.log(shipmentManager.getShipments());
```

Case Study 20: TypeScript in an Online Examination System

Objective: Use TypeScript to manage exams and student results in an online examination system.

Scenario: An online examination system needs to handle exams and track student results. Implement TypeScript interfaces and classes to manage these entities.

Steps:

1. Define interfaces for Exam and Result.
2. Create classes to manage exams and student results.
3. Implement methods to add and retrieve exams and student results.

Solution:

```typescript
interface Result {
  studentId: number;
  score: number;
  date: Date;
}

interface Exam {
  id: number;
  title: string;
  date: Date;
  results: Result[];
}

class ExamManager {
  private exams: Exam[] = [];

  addExam(exam: Exam): void {
    this.exams.push(exam);
  }
  getExams(): Exam[] {
    return this.exams;
  }

  addResult(examId: number, result: Result): void {
    const exam = this.exams.find(e => e.id === examId);
    if (exam) {
      exam.results.push(result);
    }
  }
}

const exam: Exam = { id: 1, title: "TypeScript Basics", date: new Date(), results: [] };
const result: Result = { studentId: 1, score: 90, date: new Date() };

const examManager = new ExamManager();
examManager.addExam(exam);
```

~~~
examManager.addResult(1, result);


console.log(examManager.getExams());
```
~~~