TypeScript Fundamentals

What Is TypeScript?

TypeScript is a **strongly typed programming language** that builds on JavaScript by adding static type definitions. It was developed and is maintained by Microsoft.

Key Characteristics:

- **Superset of JavaScript:** All JavaScript code is valid TypeScript code
- **Compiles to JavaScript:** TypeScript can't run directly in browsers or Node.js
- **Static Typing:** Types are checked at compile-time, not runtime
- **Object-Oriented Features:** Supports interfaces, generics, decorators
- **Modern JavaScript:** Supports latest ECMAScript features and compiles down to older versions

```
// This is TypeScript - notice the type annotation : string
let message: string = "Hello, TypeScript!";
// message = 42; // Error: Type 'number' is not assignable to type 'string'

// Regular JavaScript is valid TypeScript
let javaScriptCode = "This works too!";
```

JavaScript Vs TypeScript

JavaScript

```js
// JavaScript - Dynamic typing
function add(a, b) {
    return a + b;
}

console.log(add(5, 10));      // 15
console.log(add("5", 10));    // "510" - Unexpected behavior!
console.log(add(true, false)); // 1 - What's happening?
```

TypeScript

```
// TypeScript - Static typing
function add(a: number, b: number): number {
    return a + b;
}

console.log(add(5, 10));      // 15
// console.log(add("5", 10)); // Error: Argument of type 'string' is not assignable to parameter of type 'number'
// console.log(add(true, false)); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number'
```

Comparison Table

Feature	JavaScript	TypeScript
Type System	Dynamic (runtime)	Static (compile-time)
Learning Curve	Easier to start	Steeper but safer
IDE Support	Limited intellisense	Excellent autocomplete
Error Detection	Runtime	Compile-time

Feature	JavaScript	TypeScript
Configuration	Minimal	`tsconfig.json` required
Compilation	Interpreted	Needs compilation
Niche	Quick scripts, small apps	Large-scale applications

How & Why Should We Use TypeScript?

Why Use TypeScript?

1. **Catch Errors Early**

```
// Without TypeScript - error appears at runtime
function getUser(id) {
    return api.fetchUser(id);
}
user.name.toUpperCase(); // If user is null → Runtime crash!

// With TypeScript - caught during development
function getUser(id: number): User {
    return api.fetchUser(id);
}
user.name.toUpperCase(); // TypeScript ensures user is never null/undefined
```

2. **Better Code Documentation**

```
interface User {
    id: number;
    name: string;
    email: string;
    age?: number; // Optional property
}

// The interface serves as documentation
function processUser(user: User): void {
    console.log(`Processing user: ${user.name}`);
    // IDE knows exactly what properties user has
}
```

3. **Refactoring Confidence**

```
// Changing property name? TypeScript tells you everywhere it's used
interface Product {
    id: number;
    // price: number; // Rename this to 'cost'
    cost: number; // TypeScript will flag all usages of 'price'
}

function calculateTotal(products: Product[]): number {
    return products.reduce((total, p) => total + p.cost, 0);
}
```

4. **Team Collaboration**

```
// Types serve as contracts between team members
type APIResponse = {
    status: 'success' | 'error';
    data: User[];
    message?: string;
};

// Backend team: "We'll return exactly this structure"
// Frontend team: "We know exactly what to expect"
```

How to Use TypeScript?

1. Incrementally Add Types

```
// Start with any, gradually add proper types
let data: any = fetchData();
// Later, define proper type
interface ApiData { /* ... */ }
let typedData: ApiData = fetchData();
```

2. Use Type Inference When Possible

```
// TypeScript infers the type - no need to explicitly type
let count = 42; // TypeScript knows it's a number
let name = "John"; // TypeScript knows it's a string

// Explicit typing for complex cases
let prices: number[] = [10, 20, 30];
```

TypeScript Compiler and Install TypeScript

Installation

Global Installation

SHELL

```shell
# Using npm
npm install -g typescript

# Using yarn
yarn global add typescript

# Verify installation
tsc --version
```

Local Project Installation

SHELL

```shell
# Create project directory
mkdir my-ts-project
cd my-ts-project

# Initialize npm project
npm init -y

# Install TypeScript as dev dependency
npm install --save-dev typescript

# Check version
npx tsc --version
```

TypeScript Compiler (tsc)

The TypeScript compiler (`tsc`) is the core tool that converts TypeScript to JavaScript.

Basic Compilation

SHELL

```shell
# Compile a single file
tsc app.ts

# Watch mode - recompile on changes
tsc --watch app.ts

# Compile with specific target
tsc --target ES2020 app.ts
```

Compiler Configuration (tsconfig.json)

SHELL

```shell
# Generate tsconfig.json
tsc --init
```

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "ES2020",                       // JavaScript version to compile to
    "module": "commonjs",                      // Module system
    "lib": ["ES2020", "DOM"],                  // Library files to include
    "outDir": "./dist",                         // Output folder
    "rootDir": "./src",                         // Source folder
    "strict": true,                             // Enable all strict type checks

    /* Module Resolution */
    "esModuleInterop": true,                    // Better CommonJS/ES module compatibility
    "skipLibCheck": true,                        // Skip type checking of declaration files

    /* Advanced */
    "forceConsistentCasingInFileNames": true,    // Disallow inconsistent file casing
    "resolveJsonModule": true,                    // Allow importing JSON
    "sourceMap": true                             // Generate source maps for debugging
  },
  "include": ["src/**/*"],                         // Files to include
  "exclude": ["node_modules", "dist"]               // Files to exclude
}
```

Project Structure

```
my-ts-project/
├── src/
│   ├── index.ts
│   └── utils/
│       └── helpers.ts
├── dist/
│   ├── index.js
│   └── utils/
│       └── helpers.js
├── node_modules/
├── package.json
├── tsconfig.json
└── .gitignore
```

Package.json

Basic package.json

```json
{
  "name": "my-ts-project",
  "version": "1.0.0",
  "description": "A TypeScript project",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "build:watch": "tsc --watch",
    "start": "node dist/index.js",
    "dev": "ts-node src/index.ts",
    "clean": "rm -rf dist",
    "type-check": "tsc --noEmit"
  },
  "keywords": ["typescript", "node"],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.0.0",
    "typescript": "^5.0.0"
  },
  "dependencies": {
    "express": "^4.18.0"
  }
}
```

Important Scripts Explained

- **build:** Compiles TypeScript to JavaScript
- **build:watch:** Automatically recompiles on file changes
- **start:** Runs the compiled JavaScript
- **dev:** Runs TypeScript directly using ts-node
- **type-check:** Checks types without emitting JavaScript

Node as JS/TS Runtime

Running JavaScript with Node

```shell
# Run JavaScript file
node app.js

# Run with experimental features
node --experimental-specifier-resolution=node app.mjs
```

Running TypeScript with Node

Option 1: Compile then Run

```shell
# Step 1: Compile TypeScript
npm run build

# Step 2: Run JavaScript
npm start

# Or in one line
tsc && node dist/index.js
```

Option 2: ts-node (Development)

```shell
# Install ts-node
npm install --save-dev ts-node

# Run TypeScript directly
npx ts-node src/index.ts

# With nodemon for auto-restart
npm install --save-dev nodemon
npx nodemon --exec ts-node src/index.ts
```

Option 3: tsx (Fast, Modern)

```shell
# Install tsx
npm install --save-dev tsx

# Run TypeScript directly
npx tsx src/index.ts
```

Example: Simple Express Server

```typescript
// src/index.ts
import express, { Request, Response } from 'express';

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

const app = express();
const port = 3000;

app.use(express.json());

const users: User[] = [
    { id: 1, name: 'John Doe', email: 'john@example.com' },
    { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];

// GET endpoint with type safety
app.get('/api/users', (req: Request, res: Response) => {
    res.json(users);
});

// POST endpoint with typed request body
app.post('/api/users', (req: Request<{}, {}, Omit<User, 'id'>>, res: Response) => {
    const newUser: User = {
        id: users.length + 1,
        ...req.body
    };
    users.push(newUser);
    res.status(201).json(newUser);
});

// Type-safe route parameters
app.get('/api/users/:id', (req: Request<{ id: string }>, res: Response) => {
    const userId = parseInt(req.params.id);
    const user = users.find(u => u.id === userId);

    if (!user) {
        return res.status(404).json({ message: 'User not found' });
    }

    res.json(user);
});

app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}`);
});
```

package.json for the Express App

```json
{
  "name": "ts-express-app",
  "version": "1.0.0",
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
    "dev": "tsx watch src/index.ts"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "@types/express": "^4.17.17",
    "@types/node": "^20.0.0",
    "tsx": "^4.0.0",
    "typescript": "^5.0.0"
  }
}
```

Summary

TypeScript Benefits Checklist

- ✅ **Type Safety**: Catch errors during development
- ✅ **Better IDE Support**: Autocomplete, refactoring, navigation
- ✅ **Self-Documenting**: Types serve as documentation
- ✅ **Safer Refactoring**: Compiler catches breaking changes
- ✅ **Modern JavaScript**: Use latest features, compile for older environments
- ✅ **Team Collaboration**: Clear contracts between developers
- ✅ **Scalability**: Essential for large codebases

When to Use TypeScript

- **Large applications** with multiple developers
- **Libraries** that will be used by others
- **Critical systems** where reliability is important
- **Legacy codebases** being modernized
- **Any project** where you want better tooling and safety

When JavaScript Might Be Enough

- Quick prototypes
- Small scripts
- Learning projects
- When working with teams unfamiliar with TypeScript

Practice Exercise

Create a simple TypeScript program that demonstrates:

1. Type annotations and inference
2. Interface definition
3. Function with typed parameters and return
4. Array manipulation with type safety
5. Compilation and execution

```typescript
// exercise.ts
interface Book {
    title: string;
    author: string;
    year: number;
    inStock: boolean;
}

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

    addBook(book: Book): void {
        this.books.push(book);
    }

    findBooksByAuthor(author: string): Book[] {
        return this.books.filter(book => book.author === author);
    }

    getTotalBooks(): number {
        return this.books.length;
    }

    getAvailableBooks(): Book[] {
        return this.books.filter(book => book.inStock);
    }
}

// Usage
const myLibrary = new Library();

myLibrary.addBook({
    title: "The TypeScript Handbook",
    author: "Microsoft",
    year: 2023,
    inStock: true
});

console.log(myLibrary.getAvailableBooks());
```

Try it yourself:

1. Create the file
2. Compile with `tsc exercise.ts`
3. Run with `node exercise.js`
4. Experiment with adding incorrect types