```
//============================================================
// 📘 TypeScript Complete Guide
//============================================================

/*
 * 🎯 ملخص المفاهيم الأساسية:
 *
 * 1️⃣ BASIC DATA TYPES: تحديد نوع كل متغير لمنع الأخطاء وجعل الكود أكثر أمانًا
 * 2️⃣ ARRAYS & TUPLES: تخزين مجموعات بيانات بأنواع محددة ومتعددة
 * 3️⃣ OBJECTS: تنظيم البيانات المترابطة في هيكل واحد
 * 4️⃣ ENUMS: قيم ثابتة واضحة تجعل الكود أكثر قابلية للقراءة
 * 5️⃣ TYPE ALIAS: إعطاء أسماء مختصرة للأنواع المعقدة لإعادة استخدامها
 * 6️⃣ INTERFACES: تعريف هيكل الكائنات والعقود بين الأجزاء المختلفة من الكود
 * 7️⃣ FUNCTIONS: كتابة دوال آمنة من الأخطاء مع تحديد أنواع المعاملات والقيم المُرجعة
 * 8️⃣ TYPE CASTING: إخبار TypeScript منه بنوع البيانات عندما تعرف أفضل
 * 9️⃣ CLASSES: قوالب لإنشاء كائنات متشابهة مع تحديد الخصائص والسلوك
 * 🔟 INHERITANCE: إعادة استخدام وتوسيع الكود عن طريق الوراثة من كلاسات أخرى
 * 1️⃣1️⃣ ABSTRACT CLASSES: كلاسات قالبية لا يمكن إنشاء كائنات منها مباشرة
 * 1️⃣2️⃣ GENERICS: كتابة كود يعمل مع أنواع مختلفة دون تحديد النوع مسبقًا
 * 1️⃣3️⃣ UTILITY TYPES: أدوات جاهزة من TypeScript لتحويل وتعديل الأنواع
 * 1️⃣4️⃣ TYPE GUARDS: طرق آمنة للتحقق من أنواع البيانات في وقت التشغيل
 * 1️⃣5️⃣ ADVANCED TYPES: أنواع متقدمة تعطي قوة ومرونة أكبر في الكود
 * 1️⃣6️⃣ MODULES & NAMESPACES: تنظيم الكود في ملفات منفصلة ومنع تضارب الأسماء
 * 1️⃣7️⃣ PROMISES & ASYNC/AWAIT: التعامل مع العمليات غير المتزامنة بشكل سهل وواضح
 * 1️⃣8️⃣ INDEX SIGNATURES: إنشاء كائنات بمفاتيح ديناميكية غير معروفة مسبقًا
 * 1️⃣9️⃣ READONLY ARRAYS: حماية المصفوفات من التعديل غير المقصود
 * 2️⃣0️⃣ DISCRIMINATED UNIONS: استخدام أنواع متعددة مع القدرة على التمييز بينها بأمان
 *
*/

//============================================================
// 1️⃣ BASIC DATA TYPES
//============================================================

/*
 * تساعدك على تحديد نوع البيانات المتوقعة TypeScript الأنواع الأساسية في
 * مما يمنع الأخطاء ويجعل الكود أكثر أمانًا وسهولة في الفهم
 */

// Primitive Types
const age: number = 50;                // number: للأرقام سواء صحيحة أو عشرية
const userName: string = "Mohamed";    // string: للنصوص
const isActive: boolean = true;        // boolean: للقيم المنطقية (true/false)
const nothing: null = null;            // null: قيمة فارغة معرفة
const notDefined: undefined = undefined; // undefined: قيمة غير معرفة

console.log(age); // 50
console.log(userName); // Mohamed
```

```ts
// Any & Unknown
/*
 * any: يسمح بأي نوع بيانات (غير آمن، تجنب استخدامه)
 * unknown: مشابه لـ any يتطلب فحص النوع قبل الاستخدام (لكن أكثر أمانًا)
 */
let anyValue: any = "can be anything";
anyValue = 123; // لا يوجد خطأ - any يقبل أي شيء

let unknownValue: unknown = "safer than any";
// console.log(unknownValue.length); // خطأ! يجب فحص النوع أولاً
console.log((unknownValue as string).length); // 3 - بعد التحويل للنوع الصحيح

// Never Type
/*
 * never: يستخدم للدوال التي لا ترجع أبدًا (مثل الدوال التي ترمي أخطاء)
 * مفيد للإشارة إلى حالات لا يمكن الوصول إليها
 */
function throwError(message: string): never {
    throw new Error(message);
}

//============================================================
// 2 ARRAYS & TUPLES
//============================================================

/*
 * المصفوفات تخزن مجموعة من القيم من نفس النوع
 * Tuples تخزن عدد محدد من العناصر بأنواع محددة
 */

// Array - Single Type
let numbers: number[] = [1, 2, 3, 4, 5];         // مصفوفة أرقام فقط
let names: Array<string> = ["Ali", "Mohamed", "Ahmed"]; // مصفوفة نصوص فقط

// Array - Union Type
let mixed: (number | string)[] = [1, "Ali", 2, "Ahmed"]; // يمكن أن تحتوي على أرقام أو
    نصوص

// Tuple - Fixed length and types
/*
 * Tuple: مصفوفة محددة الطول والترتيب لكل عنصر
 * مفيدة عند الحاجة لتخزين قيم مختلفة الأنواع بترتيب معين
 */
let tuple: [number, boolean, string] = [1, true, "Ali"];
console.log(tuple); // [1, true, 'Ali']

// Tuple with Optional & Rest
let advancedTuple: [string, number, ...boolean[]] = ["test", 1, true, false];
// العنصر الأول string، الثاني number، والباقي boolean
```

```typescript
//============================================================
// 3️⃣ OBJECTS
//============================================================

/*
 * الكائنات تخزن مجموعة من الخصائص والقيم
 * يمكن تحديد نوع كل خاصية لضمان صحة البيانات
 */

// Basic Object
let user: { name: string; age: number } = {
    name: "mohamed",
    age: 22
};

// Object with Optional Properties
/*
 * علامة الاستفهام (?) تجعل الخاصية اختيارية
 * يمكن عدم كتابتها عند إنشاء الكائن
 */
let product: { id: number; name: string; price?: number } = {
    id: 1,
    name: "Laptop"
    // price اختيارية - يمكن عدم كتابتها
};

// Object with Readonly Properties
/*
 * readonly: تمنع تعديل قيمة الخاصية بعد تعريفها
 * مفيدة للقيم الثابتة التي لا يجب تغييرها
 */
let config: { readonly apiKey: string; timeout: number } = {
    apiKey: "ABC123",
    timeout: 5000
};
// config.apiKey = "XYZ"; // خطأ! لا يمكن التعديل

//============================================================
// 4️⃣ ENUMS
//============================================================

/*
 * Enum: مجموعة من القيم الثابتة المسماة
 * تجعل الكود أكثر وضوحًا وسهولة في القراءة
 * بدلاً من استخدام أرقام أو نصوص مباشرة
 */

// Numeric Enum
```

```ts
/*
 * القيم تبدأ من الرقم المحدد وتزيد تلقائيًا
 */
enum Status {
    Pending = 1,    // 1
    Approved,       // 2 (تلقائيًا)
    Rejected        // 3 (تلقائيًا)
}
console.log(Status.Approved); // 2

// String Enum
/*
 * يجب تحديد قيمة نصية لكل عنصر
 * أكثر وضوحًا عند debugging
 */
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT"
}

// Mixed Enum
/*
 * يمكن خلط الأرقام والنصوص (لكن غير منصوح به)
 */
enum Mixed {
    No = 0,
    Yes = "YES"
}
console.log(Mixed); // {0: 'No', No: 0, Yes: 'YES'}

//============================================================
// 5️⃣ TYPE ALIAS
//============================================================

/*
 * Type Alias: إعطاء اسم لنوع معين لإعادة استخدامه
 * يجعل الكود أكثر وضوحًا ويقلل التكرار
 */

// Union Type
/*
 * يسمح للمتغير بأن يكون أحد عدة أنواع
 * رمز (|) يعني "أو"
 */
type StringOrNumber = string | number;
let value: StringOrNumber = "Value";
value = 123; // صحيح أيضًا
```

```ts
// Intersection Type
/*
 * يجمع عدة أنواع في نوع واحد
 * رمز (&) يعني "و"
 * يجب أن يحتوي على كل الخصائص من جميع الأنواع
 */
type Person = { name: string };
type Employee = { employeeId: number };
type Staff = Person & Employee;

let staff: Staff = {
    name: "Mohamed",
    employeeId: 12345
};

// Literal Types
/*
 * تحدد قيم محددة بالضبط (ليس مجرد النوع)
 * مفيدة للقيم المحدودة مثل الحالات
 */
type Status2 = "pending" | "approved" | "rejected";
let orderStatus: Status2 = "pending"; // يجب أن تكون إحدى القيم الثلاث فقط

//============================================================
// 6️⃣ INTERFACES
//============================================================

/*
 * Interface: تعريف هيكل الكائن (الخصائص والدوال المتوقعة)
 * مشابهة لـ Type لكن أكثر مرونة في التوسع
 * تستخدم عادة للكائنات والكلاسات
 */

// Basic Interface
interface IUser {
    age: number;
    name: string;
}

let user2: IUser = {
    age: 22,
    name: "mohamed"
};

// Interface with Optional & Readonly
interface IProduct {
    readonly id: number;  // لا يمكن تعديلها
    name: string;
```

```
    price?: number;        // اختيارية
}

// Extending Interfaces
/*
 * يمكن أن ترث واجهة من واجهة أخرى (أو أكثر)
 * تحصل على جميع الخصائص من الواجهة الأم
 */
interface IEmployee extends IUser {
    employeeId: number;
    salary: number;
}

let employee: IEmployee = {
    name: "Ali",
    age: 30,
    employeeId: 101,
    salary: 5000
};

// Interface with Method
interface ICalculator {
    add(a: number, b: number): number;
    subtract(a: number, b: number): number;
}

//============================================================
// 7️⃣ FUNCTIONS
//============================================================

/*
 * في TypeScript يمكن تحديد أنواع المعاملات والقيمة المُرجعة
 * هذا يمنع الأخطاء ويوضح كيفية استخدام الدالة
 */

// Void Function
/*
 * void: تعني أن الدالة لا ترجع أي قيمة
 * تُستخدم للدوال التي تنفذ إجراء فقط
 */
function printMessage(name: string, msg: string): void {
    console.log(`${name}: ${msg}`);
}
printMessage("mohamed", "Welcome"); // mohamed: Welcome

// Function with Return Type
/*
 * تحديد نوع القيمة المُرجعة يضمن إرجاع النوع الصحيح
 */
```

```typescript
function add(a: number, b: number): number {
    return a + b;
}
console.log(add(5, 6)); // 11

// Optional Parameters
/*
 * المعامل الاختياري يأتي بعد المعاملات الإجبارية
 * يمكن عدم تمريره عند استدعاء الدالة
 */
function greet(name: string, greeting?: string): string {
    return greeting ? `${greeting}, ${name}` : `Hello, ${name}`;
}

// Default Parameters
/*
 * إذا لم يُمرر المعامل، تُستخدم القيمة الافتراضية
 */
function multiply(a: number, b: number = 1): number {
    return a * b;
}

// Rest Parameters
/*
 * (...) تسمح بتمرير عدد غير محدد من المعاملات
 * تُخزن في مصفوفة
 */
function sum(...numbers: number[]): number {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3, 4, 5)); // 15

// Anonymous Function
/*
 * دالة بدون اسم، تُخزن في متغير
 */
let divide = function (a: number, b: number): number {
    return a / b;
};

// Arrow Function
/*
 * صيغة مختصرة لكتابة الدوال
 * (=>) تُسمى السهم
 */
let subtract = (a: number, b: number): number => a - b;
console.log(subtract(6, 3)); // 3

// Function Overloading
```

```ts
/*
 * نفس الدالة يمكن أن تقبل أنواع مختلفة من المعاملات
 * وترجع أنواع مختلفة حسب المدخلات
 */
function process(value: string): string;
function process(value: number): number;
function process(value: any): any {
    return value;
}

//================================================================
// 8️⃣ TYPE CASTING
//================================================================

/*
 * Type Casting (Type Assertion): إخبار TypeScript بنوع معين
 * عندما تعرف نوع البيانات أفضل من TypeScript
 * لا يُحول البيانات فعليًا، فقط يُخبر TypeScript
 */

// Using 'as' keyword
let someValue: unknown = "Ali";
console.log((someValue as string).length); // 3

// Using angle brackets
let anotherValue: unknown = "Mohamed";
console.log((<string>anotherValue).length); // 7

// Non-null Assertion
/*
 * تخبر (!) علامة TypeScript أن القيمة ليست null أو undefined
 * استخدمها فقط عندما تكون متأكداً 100%
 */
let maybeString: string | null = "Hello";
console.log(maybeString!.length); // 5

//================================================================
// 9️⃣ CLASSES
//================================================================

/*
 * Class: قالب لإنشاء كائنات متشابهة
 * (تحتوي على خصائص (بيانات) ودوال (سلوك)
 */

// Basic Class
/*
 * private: الخاصية يمكن الوصول إليها فقط داخل الكلاس
 * public: (الافتراضي) الخاصية يمكن الوصول إليها من أي مكان)
```

```
 * protected: الخاصية يمكن الوصول إليها داخل الكلاس والكلاسات الوارثة
 */
class Student {
    private name: string;       // خاصة - داخل الكلاس فقط
    public age: number;         // عامة - من أي مكان

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    getName(): string {
        return this.name;
    }
}

// Shorthand Constructor
/*
 * يمكن تعريف الخصائص مباشرة في معاملات الكونستركتور
 * يوفر الوقت ويقلل الكود
 */
class Teacher {
    constructor(
        private name: string,
        public subject: string,
        protected salary: number
    ) {}

    getInfo(): string {
        return `${this.name} teaches ${this.subject}`;
    }
}

// Getters & Setters
/*
 * get: دالة تُستخدم كخاصية للقراءة
 * set: دالة تُستخدم كخاصية للكتابة
 * تسمح بإضافة منطق عند القراءة أو الكتابة
 */
class BankAccount {
    private _balance: number = 0;

    get balance(): number {
        return this._balance;
    }

    set balance(amount: number) {
        if (amount < 0) {
            throw new Error("Balance cannot be negative");
```

```
        }
        this._balance = amount;
    }
}

// Static Members
/*
 * static: الخاصية أو الدالة تنتمي للكلاس نفسه وليس للكائنات
 * يمكن الوصول إليها بدون إنشاء كائن
 */
class MathHelper {
    static PI: number = 3.14159;

    static calculateCircleArea(radius: number): number {
        return this.PI * radius * radius;
    }
}
console.log(MathHelper.calculateCircleArea(5)); // 78.53975

//============================================================
// 🔟 INHERITANCE
//============================================================

/*
 * الوراثة: آلية لإنشاء كلاس جديد يرث من كلاس موجود
 * الكلاس الجديد يحصل على كل خصائص ودوال الكلاس الأب
 * ويمكنه إضافة أو تعديل السلوك
 */

// Interface Implementation
/*
 * implements: تُلزم الكلاس بتطبيق جميع خصائص ودوال الواجهة
 * الكلاس يجب أن يحتوي على كل ما في الواجهة
 */
interface ILogin {
    name: string;
    password: string;
    login(): boolean;
}

class Login implements ILogin {
    name: string;
    password: string;

    constructor(name: string, password: string) {
        this.name = name;
        this.password = password;
    }
```

```
    login(): boolean {
        return this.password.length >= 8;
    }
}

let userLogin = new Login("mohamed", "M12345678");
console.log(userLogin.login()); // true

// Class Inheritance (Extends)
/*
 * extends: ترث كل خصائص ودوال الكلاس الأب
 * يمكن إضافة خصائص ودوال جديدة
 */
class Shape {
    constructor(
        protected height: number,
        protected width: number
    ) {}

    getArea(): number {
        return 0; // تطبيق افتراضي
    }
}

class Rectangle extends Shape {
    getArea(): number {
        return this.width * this.height;
    }

    getPerimeter(): number {
        return 2 * (this.width + this.height);
    }
}

let rect = new Rectangle(5, 6);
console.log(rect.getArea()); // 30
console.log(rect.getPerimeter()); // 22

// Method Override
/*
 * override: تعيد كتابة دالة من الكلاس الأب
 * الكلاس الابن يعطي تطبيق مختلف
 */
class Square extends Shape {
    constructor(size: number) {
        super(size, size); // super تستدعي كونستركتور الكلاس الأب
    }

    override getArea(): number {
```

```
        return this.width * this.height;
    }
}

let square = new Square(5);
console.log(square.getArea()); // 25

//================================================================
// 1️⃣1️⃣ ABSTRACT CLASSES
//================================================================

/*
 * Abstract Class: كلاس لا يمكن إنشاء كائنات منه مباشرة
 * يُستخدم فقط كقالب للكلاسات الأخرى
 * يمكن أن يحتوي على دوال مجردة (بدون تطبيق) يجب تطبيقها في الكلاسات الوارثة
 */
abstract class Animal {
    constructor(protected name: string) {}

    // دالة مجردة - يجب تطبيقها في الكلاس الوارث
    abstract makeSound(): void;

    // دالة عادية - لها تطبيق
    move(): void {
        console.log(`${this.name} is moving`);
    }
}

class Dog extends Animal {
    makeSound(): void {
        console.log("Woof! Woof!");
    }
}

class Cat extends Animal {
    makeSound(): void {
        console.log("Meow! Meow!");
    }
}

let dog = new Dog("Buddy");
dog.makeSound(); // Woof! Woof!
dog.move(); // Buddy is moving

//================================================================
// 1️⃣2️⃣ GENERICS
//================================================================

/*
```

```
 * Generics: كتابة كود يعمل مع أنواع مختلفة دون تحديد النوع مسبقًا
 * <T> هو placeholder يُحدد عند الاستخدام للنوع،
 * يجعل الكود قابل لإعادة الاستخدام مع أنواع مختلفة
 */

// Generic Function
/*
 * T يمكن أن يكون أي نوع
 * يُحدد عند استدعاء الدالة
 */
function identity<T>(arg: T): T {
    return arg;
}
console.log(identity<string>("Hello")); // Hello
console.log(identity<number>(123)); // 123

// Generic with Multiple Types
function pair<T, U>(first: T, second: U): [T, U] {
    return [first, second];
}
console.log(pair("Age", 22)); // ['Age', 22]

// Generic Class
/*
 * الكلاس يعمل مع أي نوع بيانات
 * النوع يُحدد عند إنشاء الكائن
 */
class Box<T> {
    private content: T;

    constructor(content: T) {
        this.content = content;
    }

    getContent(): T {
        return this.content;
    }
}

let stringBox = new Box<string>("Gift");
let numberBox = new Box<number>(100);
console.log(stringBox.getContent()); // Gift

// Generic Interface
interface IPair<T, U> {
    first: T;
    second: U;
}
```

```typescript
let personAge: IPair<string, number> = {
    first: "Mohamed",
    second: 22
};

// Generic Constraints
/*
 * extends: تحدد شرط للنوع العام
 * T يجب أن يحتوي على خاصية length
 */
interface Lengthwise {
    length: number;
}

function logLength<T extends Lengthwise>(arg: T): void {
    console.log(arg.length);
}
logLength("Hello"); // 5
logLength([1, 2, 3]); // 3

//============================================================
// 1️⃣3️⃣ UTILITY TYPES
//============================================================

/*
 * Utility Types: أنواع جاهزة من TypeScript لتحويل الأنواع الموجودة
 * توفر الوقت وتجعل الكود أكثر مرونة
 */

interface IPerson {
    name: string;
    age: number;
    email: string;
}

// Partial
/*
 * يجعل جميع الخصائص اختيارية
 * مفيد عند التحديثات الجزئية
 */
let partialPerson: Partial<IPerson> = {
    name: "Ali"
    // age و email أصبحوا اختياريين
};

// Required
/*
 * يجعل جميع الخصائص إجبارية
 * حتى لو كانت اختيارية في الأصل
```

```
 */
interface IOptionalPerson {
    name?: string;
    age?: number;
}
let requiredPerson: Required<IOptionalPerson> = {
    name: "Mohamed",
    age: 22 // أصبح إجباري!
};

// Readonly
/*
 * يجعل جميع الخصائص للقراءة فقط
 * لا يمكن تعديلها بعد الإنشاء
 */
let readonlyPerson: Readonly<IPerson> = {
    name: "Ahmed",
    age: 25,
    email: "ahmed@example.com"
};
// readonlyPerson.age = 30; // خطأ!

// Pick
/*
 * يختار خصائص معينة فقط من النوع
 * ينشئ نوع جديد يحتوي على الخصائص المحددة فقط
 */
type PersonNameAndAge = Pick<IPerson, "name" | "age">;
let picked: PersonNameAndAge = {
    name: "Omar",
    age: 28
};

// Omit
/*
 * يحذف خصائص معينة من النوع
 * ينشئ نوع جديد بدون الخصائص المحددة
 */
type PersonWithoutEmail = Omit<IPerson, "email">;
let omitted: PersonWithoutEmail = {
    name: "Sara",
    age: 24
};

// Record
/*
 * ينشئ نوع كائن بمفاتيح وقيم محددة
 * مفيد لإنشاء قواميس أو خرائط
 */
```

```ts
type UserRoles = Record<string, string>;
let roles: UserRoles = {
    admin: "Administrator",
    user: "Regular User",
    guest: "Guest User"
};

let scores: Record<string, number> = {
    math: 95,
    science: 88,
    english: 92
};

// Exclude
/*
 * يستبعد أنواع معينة من union type
 */
type AllTypes = string | number | boolean;
type NoBoolean = Exclude<AllTypes, boolean>; // string | number فقط

// Extract
/*
 * يستخرج أنواع معينة من union type
 */
type NumbersAndStrings = Extract<AllTypes, string | number>; // string | number

// NonNullable
/*
 * يزيل null و undefined من النوع
 */
type MaybeString = string | null | undefined;
type DefinitelyString = NonNullable<MaybeString>; // string فقط

// ReturnType
/*
 * يحصل على نوع القيمة المُرجعة من دالة
 */
function createUser() {
    return { name: "Mohamed", age: 22 };
}
type User = ReturnType<typeof createUser>; // {name: string, age: number}

// Parameters
/*
 * يحصل على أنواع معاملات الدالة كـ tuple
 */
function updateUser(id: number, name: string): void {}
type UpdateUserParams = Parameters<typeof updateUser>; // [number, string]
```

```typescript
//================================================================
// 1️⃣4️⃣ TYPE GUARDS
//================================================================

/*
 * Type Guards: طرق للتحقق من نوع البيانات في وقت التشغيل
 * تساعد TypeScript على فهم نوع البيانات في سياق معين
 */

// typeof Type Guard
/*
 * typeof: يتحقق من نوع البيانات الأساسية
 */
function printValue(value: string | number): void {
    if (typeof value === "string") {
        console.log(value.toUpperCase());
    } else {
        console.log(value.toFixed(2));
    }
}

// instanceof Type Guard
/*
 * instanceof: يتحقق من نوع الكائن (من أي كلاس)
 */
class Car {
    drive() {
        console.log("Driving...");
    }
}

class Boat {
    sail() {
        console.log("Sailing...");
    }
}

function move(vehicle: Car | Boat): void {
    if (vehicle instanceof Car) {
        vehicle.drive();
    } else {
        vehicle.sail();
    }
}

// Custom Type Guard
/*
 * دالة مخصصة للتحقق من النوع
 * يستخدم "is" للإشارة إلى نوع معين
```

```
 */
interface Fish {
    swim: () => void;
}

interface Bird {
    fly: () => void;
}

function isFish(pet: Fish | Bird): pet is Fish {
    return (pet as Fish).swim !== undefined;
}

function movePet(pet: Fish | Bird): void {
    if (isFish(pet)) {
        pet.swim();
    } else {
        pet.fly();
    }
}

//============================================================
// 1️⃣5️⃣ ADVANCED TYPES
//============================================================

/*
 * أنواع متقدمة تعطي قوة ومرونة أكبر في TypeScript
 */

// Mapped Types
/*
 * تحول نوع إلى نوع آخر عن طريق تطبيق تحويل على كل خاصية
 * [K in keyof T]: تمر على كل خاصية في النوع
 */
type Flags = {
    option1: boolean;
    option2: boolean;
};

type NullableFlags = {
    [K in keyof Flags]: Flags[K] | null;
};
// كل خاصية الآن يمكن أن تكون boolean أو null

// Conditional Types
/*
 * نوع يعتمد على شرط
 * مثل if-else للأنواع
 */
```

```ts
type IsString<T> = T extends string ? "yes" : "no";
type Test1 = IsString<string>; // "yes"
type Test2 = IsString<number>; // "no"

// Template Literal Types
/*
 * إنشاء أنواع نصية بناءً على قوالب
 * مفيد لتوليد أسماء تلقائية
 */
type EventName = "click" | "scroll" | "mousemove";
type EventHandler = `on${Capitalize<EventName>}`;
// "onClick" | "onScroll" | "onMousemove"

//============================================================
// 1️⃣6️⃣ MODULES & NAMESPACES
//============================================================

/*
 * تنظيم الكود في ملفات منفصلة
 * تمنع تضارب الأسماء وتجعل الكود أكثر تنظيماً
 */

// Module Export
/*
 * export: يجعل الكلاس أو المتغير متاح للاستيراد في ملفات أخرى
 * import: لاستيراد ما تم تصديره من ملفات أخرى
 */

// مثال على التصدير:
// export class UserService {
//     getUser(id: number): string {
//         return `User ${id}`;
//     }
// }
// export const API_KEY = "ABC123";

// مثال على الاستيراد:
// import { UserService, API_KEY } from './user-service';

// Namespace
/*
 * طريقة قديمة لتنظيم الكود (الوحدات أفضل الآن)
 * تجمع الكود المترابط تحت اسم واحد
 */
namespace Validation {
    export interface StringValidator {
        isValid(s: string): boolean;
    }
```

```
    export class EmailValidator implements StringValidator {
        isValid(s: string): boolean {
            return s.includes("@");
        }
    }
}

// استخدام Namespace
let emailValidator = new Validation.EmailValidator();
console.log(emailValidator.isValid("test@example.com")); // true

//============================================================
// 1️⃣7️⃣ PROMISES & ASYNC/AWAIT
//============================================================

/*
 * للتعامل مع العمليات التي تأخذ وقت (مثل طلبات API)
 * بدون تجميد البرنامج
 */

// Promise
/*
 * Promise: كائن يمثل نتيجة عملية غير متزامنة
 * resolve: عند النجاح
 * reject: عند الفشل
 */
function fetchData(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data loaded");
        }, 1000);
    });
}

// Async/Await
/*
 * async: تجعل الدالة غير متزامنة
 * await: تنتظر انتهاء Promise
 * (تجعل الكود غير المتزامن يبدو متزامنًا (أسهل في القراءة
 */
async function loadData(): Promise<void> {
    try {
        const data = await fetchData();
        console.log(data);
    } catch (error) {
        console.error("Error:", error);
    }
}
```

```typescript
// Generic Promise
/*
 * Promise يمكن أن يرجع أي نوع بيانات
 */
async function getUserData<T>(id: number): Promise<T> {
    // محاكاة استدعاء API
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({ id, name: "Mohamed" } as T);
        }, 1000);
    });
}

//============================================================
// 1️⃣8️⃣ INDEX SIGNATURES
//============================================================

/*
 * Index Signatures: تسمح بإنشاء كائنات بمفاتيح ديناميكية
 * مفيدة عندما لا تعرف أسماء الخصائص مسبقًا
 */

// Basic Index Signature
interface StringDictionary {
    [key: string]: string;
}

let dictionary: StringDictionary = {
    hello: "مرحبًا",
    goodbye: "وداعًا",
    thanks: "شكرًا"
};

// Mixed Index Signature
interface MixedData {
    name: string;  // خاصية محددة
    [key: string]: string | number;  // خصائص ديناميكية
}

let userData: MixedData = {
    name: "Mohamed",
    age: 22,
    city: "Cairo"
};

//============================================================
// 1️⃣9️⃣ READONLY ARRAYS & TUPLES
//============================================================
```

```ts
/*
 * ReadonlyArray: مصفوفة لا يمكن تعديلها
 * مفيدة لحماية البيانات من التغيير غير المقصود
 */

// Readonly Array
const readonlyNumbers: ReadonlyArray<number> = [1, 2, 3, 4, 5];
// readonlyNumbers.push(6); // خطأ! لا يمكن إضافة عناصر
// readonlyNumbers[0] = 10; // خطأ! لا يمكن تعديل العناصر

// Readonly Tuple
const readonlyTuple: readonly [string, number] = ["Mohamed", 22];
// readonlyTuple[0] = "Ali"; // خطأ! لا يمكن التعديل

//============================================================
// 2️⃣0️⃣ DISCRIMINATED UNIONS
//============================================================

/*
 * استخدام خاصية مشتركة للتمييز بين الأنواع
 * يجعل الكود أكثر أمانًا وسهولة في الفهم
 */

interface Circle {
    kind: "circle";
    radius: number;
}

interface Square2 {
    kind: "square";
    size: number;
}

interface Triangle {
    kind: "triangle";
    base: number;
    height: number;
}

type Shape2 = Circle | Square2 | Triangle;

function calculateArea(shape: Shape2): number {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.size ** 2;
        case "triangle":
            return (shape.base * shape.height) / 2;
```

```
    }
}

//============================================================
// 📌 END OF TYPESCRIPT COMPLETE GUIDE
//============================================================


/*
  *
 * 💡 نصائح مهمة للمبتدئين:
 * ✅ ابدأ بتحديد الأنواع البسيطة (string, number, boolean) قبل الانتقال للمعقدة
 * ✅ استخدم أنواع محددة بدلاً من any لأن any يلغي فوائد TypeScript
 * ✅ استخدم interfaces عند تعريف شكل الكائنات والبيانات
 * ✅ استخدم type alias عند العمل مع Union أو Intersection Types
 * ✅ اجعل الخصائص readonly عندما لا تريد تعديلها بعد الإنشاء
 * ✅ استخدم Generics عندما تريد كتابة كود يعمل مع أنواع مختلفة
 * ✅ استخدم Utility Types (مثل Partial, Required, Pick) لتوفير الوقت
 * ✅ استخدم Type Guards للتحقق من الأنواع بشكل آمن قبل الاستخدام
 * ✅ تجنب Type Casting إلا عند الضرورة القصوى
 * ✅ استخدم async/await بدلاً من .then() للتعامل مع Promises
 * ✅ نظم الكود في modules منفصلة لسهولة الصيانة
 *
 * 📚 موارد إضافية للتعلم:
 * 🌐 الموقع الرسمي: https://www.typescriptlang.org
 * 📖 الوثائق الرسمية: https://www.typescriptlang.org/docs
 * 🎮 ملعب TypeScript: https://www.typescriptlang.org/play
 * 📝 GitHub Repository: https://github.com/microsoft/TypeScript
 */
```