TS CHEATSHEET

📌 1. Installation & Setup

Install TypeScript globally via npm:

```
npm install -g typescript
```

Initialize a TypeScript project with a default configuration:

```
tsc --init
```

This creates a `tsconfig.json` file where you can adjust compiler options.

📌 2. Basic Types

Type	Example
Boolean	`ts let isActive: boolean = true; `
Number	`ts let count: number = 42; `
String	`ts let message: string = "Hello, TS!"; `
Array	`ts let scores: number[] = [10, 20, 30]; `
Tuple	`ts let person: [string, number] = ["Alice", 25]; `
Enum	`ts enum Color { Red, Green, Blue } let c: Color = Color.Green; `
Any	`ts let random: any = "anything"; `
Unknown	`ts let value: unknown = 123; `
Void	`ts function log(msg: string): void { console.log(msg); } `
Null / Undefined	`ts let n: null = null; let u: undefined = undefined; `

📌 3. Functions & Arrow Functions

- **Named Function:**

```
function add(a: number, b: number): number {
  return a + b;
}
```

- **Optional & Default Parameters:**

```
function greet(name: string, greeting: string = "Hello"): string {
  return `${greeting}, ${name}!`;
```

- **Arrow Function:**

```
const multiply = (a: number, b: number): number ⇒ a * b;
```

📌 4. Interfaces & Type Aliases

- **Interface:**

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

- **Type Alias:**

```
type Point = {
  x: number;
  y: number;
};
```

Interfaces are best for defining object shapes that can be extended, while type aliases work well for unions and primitives.

📌 5. Classes & Inheritance

- **Basic Class:**

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  move(distance: number = 0): void {
    console.log(`${this.name} moved ${distance}m.`);
  }
}
```

- **Inheritance:**

```
class Dog extends Animal {
  breed: string;
  constructor(name: string, breed: string) {
    super(name);
    this.breed = breed;
  }
  bark(): void {
    console.log("Woof! Woof!");
  }
}
```

- **Access Modifiers:**
  - `public` (default)
  - `private`
  - `protected`

📌 6. Generics

- **Generic Function:**

```
function identity<T>(arg: T): T {
  return arg;
}
```

- **Generic Class:**

```
class Box<T> {
  content: T;
  constructor(content: T) {
    this.content = content;
  }
}
```

- **Generic Constraints:**

```
interface Lengthwise {
  length: number;
}
function logLength<T extends Lengthwise>(arg: T): void {
  console.log(arg.length);
}
```

📌 7. Advanced Types

- **Union & Intersection Types:**

```

type Success = { response: string };
type Failure = { error: string };
type Result = Success | Failure;
```

- **Type Guards:**

```
function isString(value: any): value is string {
  return typeof value === "string";
}
```

- **Conditional Types:**

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

- **Mapped & Utility Types:**

```
interface User {
  id: number;
  name: string;
}
// Partial makes all properties optional
const updateUser = (user: Partial<User>) => { /* ... */ };
```

📌 8. Modules & Namespaces

- **Modules (ES Modules):**

```
// math.ts
export const PI = 3.14;
export function square(x: number): number {
  return x * x;
}

// app.ts
import { PI, square } from "./math";
```

- **Namespaces:**

```
namespace Utils {
  export function log(msg: string): void {
    console.log(msg);
  }
}
Utils.log("Hello from a namespace!");
```

Modules are preferred for modern codebases; namespaces can help organize code in legacy setups.

📌 9. Compiler Options & Configuration

- **Basic tsconfig.json Example:**

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true},
  "include": ["src/**/*"]
}
```

- **Common Flags:**
  - `-strict`: Enables strict type-checking.
  - `-noImplicitAny`: Warns when a variable is implicitly of type `any`.
  - `-outDir`: Specifies the output directory for compiled files.

📌 10. Utility Types & Advanced Concepts

- **Built-in Utility Types:**

Utility	Description	Example

Partial	Makes all properties optional	`Partial<User>`
Required	Makes all properties required	`Required<User>`
Readonly	Makes all properties readonly	`Readonly<User>`
Record	Constructs a type with a set of properties of a given type	`Record<string, number>`
Pick	Picks a set of properties from a type	`Pick<User, "id"
Omit	Removes a set of properties from a type	`Omit<User, "email">`

- **Declaration Merging:**

  Interfaces with the same name can be merged to extend functionality.

- **Decorators:**

  Experimental metadata for classes and methods (requires enabling in `tsconfig.json`).

📌 11. Additional Resources

- TypeScript Official Documentation
- DefinitelyTyped (Community Type Definitions)
- Awesome TypeScript – a curated list of TypeScript resources