



TypeScript

CHEAT SHEET



Before start

TypeScript is an open-source programming language developed by Microsoft that extends JavaScript by adding static types, providing developers with powerful tools to write cleaner, safer, and more predictable code, especially in large-scale projects.

Basic Configuration

npm installation

```
npm install -g typescript
```

Compilation

```
tsc hello.ts
```

Configuration

Configures TypeScript compiler options.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true
  }
}
```

Basic Typing



In TypeScript, typing is a system that allows developers to define the types of variables, function parameters, and object properties. It provides a way to describe the shape and behavior of an object, ensuring that the code behaves as expected during runtime and significantly reducing the chance of runtime errors.

Boolean

```
let isDone: boolean = false;
```

Number

```
let decimal: number = 6;
```

String

```
let color: string = "blue";
```

Array

```
let list: number[] = [1, 2, 3];
```

Tuple

```
let x: [string, number] = ["hello", 10];
```

Enum

```
enum Color {Red, Green, Blue}  
let c: Color = Color.Green;
```





Any

```
let notSure: any = 4;
```

Void

```
function warnUser(): void {  
    console.log("This is a warning message");  
}
```

Null and Undefined

```
let u: undefined = undefined;  
let n: null = null;
```

Never

Used for functions that never return (e.g. a function that throws an exception).

```
function error(message: string): never {  
    throw new Error(message);  
}
```

Unknown

```
let notKnown: unknown = 4;
```



Interfaces

- Simple definition: Used to define the structure of an object, ensuring that the object has certain properties.
- Optional properties: Used to indicate that certain interface properties are not required.
- Read-only properties: Prevent property reassignment after initial assignment.

Simple Definition :

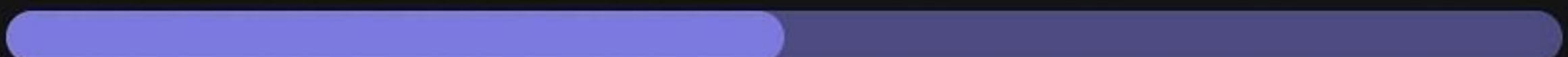
```
interface LabeledValue {  
    label: string;  
}  
  
function printLabel(labeledObj: LabeledValue) {  
    console.log(labeledObj.label);  
}
```

Optional Properties

```
interface SquareConfig {  
    color?: string;  
    width?: number;  
}
```

Readonly Properties

```
interface Point {  
    readonly x: number;  
    readonly y: number;  
}
```





Advanced Types

Union Types

Allow a value to be of one of the specified types.

```
function padLeft(value: string, padding: string | number) {  
  // ...  
}
```

Type Guards

Mechanism to influence the type of verification by providing more precise type information.

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

Intersection Types

Combine several types into one, which means that an object of this type will have all the properties of the combined types.

```
type Combined = { a: number } & { b: string };
```

Type Aliases

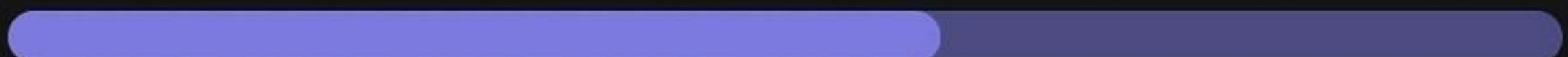
Create a name for an existing type, simplifying complex types or unions.

```
type StringOrNumber = string | number;
```

Mapped Types

Create new types by transforming all types of another type.

```
type Readonly<T> = { readonly [P in keyof T]: T[P]; }
```



Classes

Basic definition

```
class Greeter {  
  greeting: string;  
  constructor(message: string) {  
    this.greeting = message;  
  }  
  greet() {  
    return "Hello, " + this.greeting;  
  }  
}
```

Inheritance

```
class Animal {  
  move() {  
    console.log("Moving along!");  
  }  
}  
  
class Dog extends Animal {  
  bark() {  
    console.log("Woof! Woof!");  
  }  
}
```

Access Modifiers

```
class Animal {  
  private name: string;  
  constructor(theName: string) { this.name = theName; }  
}
```


Functions

Optional and Default Parameters

```
function buildName(firstName: string, lastName?: string) {  
    // ...  
}  
  
function buildName(firstName: string, lastName = "Smith") {  
    // ...  
}
```

Rest Parameters

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}
```

Generics

General Use

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

Generic Interface

```
interface GenericIdentityFn<T> {  
    (arg: T): T;  
}
```

Generic Class

```
class GenericNumber<T> {  
    zeroValue: T;  
    add: (x: T, y: T) => T;  
}
```


Enumerations

Simple Enum

```
enum Direction {  
  Up = 1,  
  Down,  
  Left,  
  Right,  
}
```

String-valued Enums

```
enum Response {  
  No = 0,  
  Yes = "YES",  
}
```

Namespaces

Namespaces in TypeScript are used to organize code into named groups, allowing developers to group related functionalities under a named scope to prevent naming conflicts and improve modularity.

```
namespace Validation {  
  export interface StringValidator {  
    isAcceptable(s: string): boolean;  
  }  
}
```

Decorators

```
function sealed(constructor: Function) {  
  Object.seal(constructor);  
  Object.seal(constructor.prototype);  
}  
  
@sealed  
class Greeter {  
  // ...  
}
```