TypeScript Cheatsheet

A comprehensive cheat sheet covering TypeScript syntax, types, and features, designed for quick reference.

Basic Types

Primitive Types

`boolean`	Represents `true` or `false` values. `let isDone: boolean = false;`

`number`	Represents numeric values (floating point numbers).
	`let decimal: number = 6;`
	`let hex: number = 0xf00d;`

`string`	Represents a sequence of characters.
	`let color: string = "blue";`
	`let sentence: string = `A ${ color } shirt.`;`

`array`	Represents an array of values of a specific type.
	`let list: number[] = [1, 2, 3];`
	`let list: Array<number> = [1, 2, 3];`

`tuple`	Represents an array with a fixed number of elements whose types are known.
	`let x: [string, number];`
	`x = ["hello", 10];`

`enum`	A way of giving more friendly names to sets of numeric values.
	`enum Color {Red, Green, Blue};`
	`let c: Color = Color.Green;`

Special Types

`any`	Represents a type that can be anything. Use when the type is not known.
	`let notSure: any = 4;`
	`notSure = "maybe a string instead";`

`void`	Represents the absence of a type, usually for functions that do not return a value.
	`function warnUser(): void {`
	`  console.log("This is my warning message");`
	`}`

`null` and `undefined`	Represent `null` and `undefined` values respectively. They are subtypes of all other types.
	`let u: undefined = undefined;`
	`let n: null = null;`

`never`	Represents the type of values that never occur. Used for functions that always throw an exception or never return.
	`function error(message: string): never {`
	`  throw new Error(message);`
	`}`

Interfaces and Classes

Interfaces

Interfaces define contracts for objects. They specify the properties and methods an object must have.

```
interface Person {
  firstName: string;
  lastName: string;
  age?: number; // Optional property
  greet(): string;
}

let user: Person = {
  firstName: "John",
  lastName: "Doe",
  greet: () => "Hello, world!"
};
```

Interfaces can also describe function types:

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];
```

Classes

Classes are blueprints for creating objects. They contain properties and methods.

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

let cat = new Animal("Cat");
cat.move(10);
```

Classes can implement interfaces:

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date): void;
}

class Clock implements ClockInterface {
    currentTime: Date = new Date();
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

Access Modifiers

- `public`: Accessible from anywhere (default).
- `private`: Accessible only within the class.
- `protected`: Accessible within the class and its subclasses.

Functions

Function Types

Functions in TypeScript can be defined with named parameters and types.

```
function add(x: number, y: number): number {
  return x + y;
}

let myAdd: (x: number, y: number) => number =
function(x: number, y: number): number {
  return x + y;
};
```

Optional and Default Parameters

TypeScript supports optional and default parameters in functions.

```
function buildName(firstName: string,
lastName?: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");  // works
correctly
let result2 = buildName("Bob", "Adams");  //
works correctly

function buildName2(firstName: string,
lastName = "Smith") {
  return firstName + " " + lastName;
}

let result3 = buildName2("Bob");         //
works correctly
let result4 = buildName2("Bob", undefined);
// works correctly, returns "Bob Smith"
```

Rest Parameters

Rest parameters allow you to pass a variable number of arguments to a function.

```
function buildName3(firstName: string,
...restOfName: string[]) {
  return firstName + " " + restOfName.join("
");
}

let employeeName = buildName3("Joseph",
"Samuel", "Lucas", "MacKinzie");
```

Function Overloads

Function overloads allow you to define multiple function signatures for the same function name.

```
function pickCard(x: { suit: string; card:
number; }[]): number;
function pickCard(x: number): { suit: string;
card: number; };
function pickCard(x): any {
    // Check to see if we're working with an
object/array
    // if so, they gave us the deck and we'll
pick the card
    if (typeof x == "object") {
        let pickedCard =
Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit],
card: x % 13 };
    }
}
```

Generics

Generic Functions

Generics allow you to write functions that can work with a variety of types without sacrificing type safety.

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

let output = identity<string>("myString");  //
type of output will be 'string'
let output2 = identity("myString");  // type
of output will be 'string'
```

Generic Classes

Generic classes are similar to generic interfaces.

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

let myGenericNumber = new
GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return
x + y; };
```

Generic Constraints

You can constrain the types that a generic type parameter can be.

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>
(arg: T): T {
    console.log(arg.length);  // Now we know
it has a .length property, so no more error
    return arg;
}
```

Generic Interfaces

You can also define generic interfaces.

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

let myIdentity: GenericIdentityFn<number> =
identity;
```