TypeScript CheatSheet Basic to Advance

A comprehensive TypeScript cheat sheet covering basic to advanced concepts, including types, functions, classes, generics, utility types, and best practices.

TypeScript Basics

Type Annotations & Variables

Type Annotations
Explicitly define the type of a variable.

```
let myString: string = "Hello";
let myNumber: number = 42;
let myBoolean: boolean = true;
```

Variable Declarations
`let` and `const` are preferred over `var`.

```
let changeableValue = 10;
const constantValue = "Fixed";
```

Basic Types
`string`, `number`, `boolean`, `any`, `null`, `undefined`, `void`.

```
let str: string = "text";
let num: number = 123;
let bool: boolean = true;
let anything: any = 'can be anything';
let nothing: void = undefined;
```

Any Type
Opt-out of type checking. Use sparingly.

```
let flexible: any = "string";
flexible = 123; // No error
```

Null and Undefined
Represent the absence of a value.

```
let nothingHere: null = null;
let notAssigned: undefined = undefined;
```

Functions

Function Types
Specifying parameter types and return type.

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

Optional Parameters
Mark parameters as optional using `?`.

```
function greet(name: string, greeting?: string): string {
  return `Hello, ${name}! ${greeting || 'Welcome'}`;
}
```

Default Parameters
Assign default values to parameters.

```
function multiply(num: number, factor: number = 2): number {
  return num * factor;
}
```

Function Expressions
Assigning functions to variables.

```
const subtract = function(x: number, y: number): number {
  return x - y;
};
```

Arrow Functions
A concise syntax for writing functions.

```
const divide = (x: number, y: number): number => x / y;
```

Arrays & Tuples

Arrays
Collections of values of the same type.

```
let numbers: number[] = [1, 2, 3];
let strings: Array<string> = ['a', 'b', 'c'];
```

Tuples
Arrays with a fixed number of elements and known types.

```
let person: [string, number] = ['Alice', 30];
```

Readonly Arrays
Arrays that cannot be modified after creation.

```
let readonlyArray: readonly number[] = [1, 2, 3];
// readonlyArray[0] = 5; // Error
```

Void
Typically used for functions that do not return a value.

```
function logMessage(message: string): void {

console.log(message);
}
```

Objects, Interfaces, and Types

Objects & Interfaces

Objects
Collections of key-value pairs.

```
let car: { make: string; model: string; year: number } = {
  make: 'Toyota',
  model: 'Camry',
  year: 2020,
};
```

Interfaces
Define a contract for objects.

```
interface Person {
  name: string;
  age: number;
}

let person: Person = { name: 'Bob', age: 25 };
```

Optional Properties
Properties in an interface can be optional.

```
interface Config {
  apiUrl: string;
  timeout?: number;
}
```

Readonly Properties
Properties that cannot be changed after initialization.

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

Union & Intersection Types

Union Types
A variable can have one of several types.

```
let statusCode: string | number = 404;
statusCode = 'Not Found';
```

Intersection Types
Combines multiple types into one.

```
interface ErrorHandling {
  success: boolean;
  message?: string;
}

interface Data {
  data: any;
}

type ApiResponse = ErrorHandling & Data;

const response: ApiResponse = {
success: true, data: { name: 'John' } };
```

Type Aliases
Create a name for a type.

```
type StringOrNumber = string | number;

let value: StringOrNumber = 'text';
value = 123;
```

Literal Types & Enums

Literal Types
Specify exact values a variable can have.

```
type Direction = 'North' | 'East' | 'South' | 'West';
let myDirection: Direction = 'North';
```

Enums
Define a set of named constants.

```
enum Color { Red, Green, Blue }

let myColor: Color = Color.Red;
```

String Enums
Enums where members have string values.

```
enum Status {
Success = 'OK',
Failure = 'Error' }
let requestStatus: Status = Status.Success;
```

Classes & Object-Oriented Programming

Classes

Basic Class
Define properties and methods.

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

console.log('Moving...');
  }
}

const animal = new
Animal('Dog');
```

Inheritance
Create new classes from existing classes.

```
class Dog extends Animal {
  bark() {
    console.log('Woof!');
  }
}

const dog = new
Dog('Fido');
dog.move(); // Inherited
from Animal
dog.bark();
```

Access Modifiers
Control the visibility of class members (`public`, `private`, `protected`).

```
class Person {
  public name: string; //
Accessible from anywhere
  private age: number; //
Only accessible within the
class
  protected
constructor(name: string,
age: number) {
    this.name = name;
    this.age = age;
  }
}
```

Abstract Classes
Cannot be instantiated directly and may contain abstract methods.

```
abstract class Shape {
  abstract getArea():
number;
}

class Circle extends Shape
{
  radius: number;
  constructor(radius:
number) {
    super();
    this.radius = radius;
  }
  getArea() {
    return Math.PI *
this.radius * this.radius;
  }
}
```

Interfaces with Classes

Implementing Interfaces
A class can implement one or more interfaces.

```
interface Loggable {
  log(): void;
}

class Logger implements
Loggable {
  log() {

console.log('Logging...');
  }
}
```

Advanced Types & Generics

Generics

Generic Functions
Write functions that work with a variety of types.

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

let myString: string = identity<string>('hello');
let myNumber: number = identity<number>(123);
```

Generic Interfaces
Create reusable interfaces for different types.

```
interface Result<T> {
  success: boolean;
  data?: T;
  error?: string;
}

let successResult: Result<number> = { success: true, data: 42 };
```

Generic Classes
Create reusable classes for different types.

```
class DataHolder<T> {
  data: T;
  constructor(data: T) {
    this.data = data;
  }
}

let numberHolder = new DataHolder<number>(10);
```

Generic Constraints
Limit the types that a generic type can be.

```
interface Lengthwise {
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): void {

console.log(arg.length);
}
```

Utility Types

Partial
Makes all properties in T optional.

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}
```

Readonly
Makes all properties in T readonly.

```
const todo: Readonly<Todo> = {
  title: 'Clean room',
  description: 'Clear mess',
};

// todo.title = 'Do something else'; // Error: cannot reassign
```

Pick<T, K>
Selects a set of properties K from T.

```
interface Person {
  name: string;
  age: number;
  location: string;
}

type JustNameAndAge = Pick<Person, 'name' | 'age'>;
```

Omit<T, K>
Removes a set of properties K from T.

```
type LocationOmitted = Omit<Person, 'location'>;
```

Mapped & Conditional Types

Mapped Types
Transform existing types by mapping over their properties.

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

Conditional Types
Define types that act like if statements in the type system.

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

type StringOrNumber = NonNullable<string | number | undefined>; // string | number
```