Complete TypeScript Cheatsheet - Beginner to Advanced

Table of Contents

---

1. Introduction to TypeScript

TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. It adds optional static type definitions to JavaScript, enabling better tooling, error detection, and code maintainability.

Key Benefits:

- **Static Type Checking**: Catch errors at compile time
- **Enhanced IDE Support**: Better autocomplete, refactoring, and navigation
- **Self-Documenting Code**: Types serve as inline documentation
- **Better Refactoring**: Safe renaming and restructuring
- **ES6+ Features**: Access to latest JavaScript features

How TypeScript Works:

```
// TypeScript code
let message: string = "Hello, TypeScript!";
console.log(message);

// Compiles to JavaScript
var message = "Hello, TypeScript!";
console.log(message);
```

---

2. Setup and Configuration

Installation

```
# Global installation
npm install -g typescript

# Project-specific installation
npm install --save-dev typescript
npm install --save-dev @types/node

# Using with ts-node for development
npm install --save-dev ts-node
```

Basic tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "lib": ["ES2020", "DOM"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Compiler Options Explained

```
{
  "compilerOptions": {
    // Target JavaScript version
    "target": "ES2020",

    // Module system
    "module": "commonjs", // or "esnext", "amd", "umd"

    // Type checking strictness
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,

    // Output options
    "outDir": "./dist",
    "sourceMap": true,
    "declaration": true,

    // Module resolution
    "moduleResolution": "node",
    "baseUrl": "./",
    "paths": {
      "@/*": ["src/*"],
      "@utils/*": ["src/utils/*"]
    }
  }
}
```

---

3. Basic Types

Primitive Types

```
// String
let name: string = "John";
let template: string = `Hello, ${name}!`;

// Number
let age: number = 25;
let decimal: number = 6.5;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

// Boolean
let isDone: boolean = false;
let isActive: boolean = true;

// Undefined and Null
```

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

// Symbol
let sym1: symbol = Symbol("key");
let sym2: symbol = Symbol.for("key");

// BigInt
let big: bigint = 100n;
let anotherBig: bigint = BigInt(100);
```

Any Type

```
// Avoid using 'any' when possible
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

// Better alternatives
let value: unknown = 4; // Use unknown instead of any
let userInput: string | number = getUserInput(); // Use union types
```

Void and Never

```
// Void - for functions that don't return a value
function warnUser(): void {
  console.log("This is a warning message");
}

// Never - for functions that never return
function error(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {
    // This function never returns
  }
}
```

Object Type

```
// Object type (not recommended for specific shapes)
let obj: object = { name: "John", age: 30 };

// Better approach with specific type
```

```typescript
let person: { name: string; age: number } = {
  name: "John",
  age: 30
};

// Optional properties
let optionalPerson: { name: string; age?: number } = {
  name: "Jane"
};
```

---

4. Variables and Constants

Variable Declarations

```typescript
// let and const (preferred)
let mutableValue: string = "I can change";
const immutableValue: string = "I cannot change";

// var (avoid in modern TypeScript)
var oldStyle: string = "Legacy syntax";

// Type inference
let inferredString = "TypeScript infers this as string";
let inferredNumber = 42; // Inferred as number
let inferredBoolean = true; // Inferred as boolean
```

Destructuring with Types

```typescript
// Array destructuring
let input: [string, number] = ["hello", 42];
let [first, second]: [string, number] = input;

// Object destructuring
let person = { name: "John", age: 30, city: "New York" };
let { name, age }: { name: string; age: number } = person;

// With default values
let { name: personName = "Unknown", age: personAge = 0 } = person;

// Function parameter destructuring
function greet({ name, age }: { name: string; age: number }): string {
  return `Hello ${name}, you are ${age} years old`;
}
```

Readonly Properties

```
// Readonly variables
const readonlyArray: readonly number[] = [1, 2, 3];
// readonlyArray.push(4); // Error: Property 'push' does not exist

// Readonly object properties
interface ReadonlyPerson {
  readonly id: number;
  readonly name: string;
  age: number; // Can be modified
}

let person: ReadonlyPerson = { id: 1, name: "John", age: 30 };
// person.id = 2; // Error: Cannot assign to 'id'
person.age = 31; // OK
```

---

5. Functions

Function Types

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

// Function expression
let multiply = function(x: number, y: number): number {
  return x * y;
};

// Arrow function
let divide = (x: number, y: number): number => x / y;

// Function type annotation
let calculate: (x: number, y: number) => number;
calculate = add; // OK
calculate = multiply; // OK
```

Optional and Default Parameters

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

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

// Rest parameters
function concatenate(separator: string, ...strings: string[]): string {
  return strings.join(separator);
}

let result = concatenate(", ", "apple", "banana", "cherry");
```

Function Overloads

```typescript
// Function overloads
function combine(a: string, b: string): string;
function combine(a: number, b: number): number;
function combine(a: any, b: any): any {
  return a + b;
}

let stringResult = combine("Hello", "World"); // string
let numberResult = combine(1, 2); // number

// More complex overloads
function createElement(tag: "div"): HTMLDivElement;
function createElement(tag: "span"): HTMLSpanElement;
function createElement(tag: string): HTMLElement;
function createElement(tag: string): HTMLElement {
  return document.createElement(tag);
}
```

Higher-Order Functions

```typescript
// Function that returns a function
function createMultiplier(multiplier: number): (x: number) => number {
  return (x: number) => x * multiplier;
}

let double = createMultiplier(2);
let triple = createMultiplier(3);

// Function that accepts a function
function processArray<T, U>(
  array: T[],
  processor: (item: T) => U
): U[] {
  return array.map(processor);
}
```

```
let numbers = [1, 2, 3, 4, 5];
let doubled = processArray(numbers, x => x * 2);
let strings = processArray(numbers, x => x.toString());
```

Async Functions

```
// Async function with Promise return type
async function fetchData(url: string): Promise<string> {
  const response = await fetch(url);
  return response.text();
}

// Generic async function
async function fetchJson<T>(url: string): Promise<T> {
  const response = await fetch(url);
  return response.json();
}

// Error handling in async functions
async function safeAsyncOperation(): Promise<string | null> {
  try {
    const result = await fetchData("https://api.example.com/data");
    return result;
  } catch (error) {
    console.error("Failed to fetch data:", error);
    return null;
  }
}
```

---

6. Arrays and Tuples

Array Types

```
// Array type notation
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: string[] = ["apple", "banana", "cherry"];

// Generic array type
let genericNumbers: Array<number> = [1, 2, 3];
let genericStrings: Array<string> = ["a", "b", "c"];

// Multi-dimensional arrays
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

```
// Mixed type arrays (union types)
let mixed: (string | number)[] = ["hello", 42, "world", 100];
```

Array Methods with Types

```
let numbers: number[] = [1, 2, 3, 4, 5];

// Map with type inference
let doubled: number[] = numbers.map(n => n * 2);
let strings: string[] = numbers.map(n => n.toString());

// Filter with type guards
let evens: number[] = numbers.filter(n => n % 2 === 0);

// Reduce with explicit types
let sum: number = numbers.reduce((acc, curr) => acc + curr, 0);
let concatenated: string = numbers.reduce((acc, curr) => acc + curr.toString(),
"");

// Find with possible undefined
let found: number | undefined = numbers.find(n => n > 3);

// Some and every return boolean
let hasEven: boolean = numbers.some(n => n % 2 === 0);
let allPositive: boolean = numbers.every(n => n > 0);
```

Tuples

```
// Basic tuple
let person: [string, number] = ["John", 30];
let [name, age] = person; // Destructuring

// Tuple with optional elements
let coordinate: [number, number, number?] = [10, 20];
coordinate = [10, 20, 30]; // Also valid

// Readonly tuples
let readonlyTuple: readonly [string, number] = ["John", 30];
// readonlyTuple[0] = "Jane"; // Error

// Named tuple elements (TypeScript 4.0+)
let namedTuple: [name: string, age: number, isActive: boolean] =
  ["John", 30, true];

// Rest elements in tuples
let stringNumberBooleans: [string, ...number[], boolean] =
  ["hello", 1, 2, 3, true];
```

Advanced Array Patterns

```
// Array of objects with specific shape
interface User {
  id: number;
  name: string;
  email: string;
}

let users: User[] = [
  { id: 1, name: "John", email: "john@example.com" },
  { id: 2, name: "Jane", email: "jane@example.com" }
];

// Array of functions
type MathOperation = (a: number, b: number) => number;
let operations: MathOperation[] = [
  (a, b) => a + b,
  (a, b) => a - b,
  (a, b) => a * b,
  (a, b) => a / b
];

// Constant assertions for readonly arrays
const fruits = ["apple", "banana", "cherry"] as const;
type Fruit = typeof fruits[number]; // "apple" | "banana" | "cherry"
```

---

7. Objects and Interfaces

Basic Interfaces

```
// Basic interface
interface Person {
  name: string;
  age: number;
  email: string;
}

let john: Person = {
  name: "John Doe",
  age: 30,
  email: "john@example.com"
};

// Optional properties
interface User {
  id: number;
  username: string;
  email?: string; // Optional
```

```
  isActive?: boolean; // Optional
}

let user: User = {
  id: 1,
  username: "johndoe"
  // email and isActive are optional
};
```

Readonly Properties

```
interface ReadonlyPerson {
  readonly id: number;
  readonly created: Date;
  name: string;
  age: number;
}

let person: ReadonlyPerson = {
  id: 1,
  created: new Date(),
  name: "John",
  age: 30
};

// person.id = 2; // Error: Cannot assign to 'id'
person.name = "Jane"; // OK
```

Index Signatures

```
// String index signature
interface StringDictionary {
  [key: string]: string;
}

let dict: StringDictionary = {
  "key1": "value1",
  "key2": "value2"
};

// Number index signature
interface NumberArray {
  [index: number]: string;
}

let arr: NumberArray = ["a", "b", "c"];

// Mixed index signatures
interface MixedDictionary {
```

```
  [key: string]: string | number;
  length: number; // OK, number is assignable to string | number
}
```

Function Properties

```
interface Calculator {
  // Method signature
  add(a: number, b: number): number;

  // Function property
  subtract: (a: number, b: number) => number;

  // Optional method
  multiply?(a: number, b: number): number;
}

let calc: Calculator = {
  add(a, b) { return a + b; },
  subtract: (a, b) => a - b
};
```

Extending Interfaces

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

interface Dog extends Animal {
  breed: string;
  bark(): void;
}

interface Cat extends Animal {
  color: string;
  meow(): void;
}

// Multiple inheritance
interface Pet extends Animal {
  owner: string;
}

interface WorkingDog extends Dog, Pet {
  job: string;
}

let workingDog: WorkingDog = {
```

```
  name: "Rex",
  age: 5,
  breed: "German Shepherd",
  owner: "John",
  job: "Police Dog",
  bark() { console.log("Woof!"); }
};
```

Interface vs Type Aliases

```
// Interface - can be extended and merged
interface User {
  name: string;
}

interface User {
  age: number; // Declaration merging
}

// Type alias - more flexible but cannot be merged
type Person = {
  name: string;
  age: number;
};

type Employee = Person & {
  employeeId: string;
  department: string;
};

// Choose interface for object shapes, type for unions/computed types
type Status = "pending" | "approved" | "rejected";
type EventHandler = (event: Event) => void;
```

Advanced Object Patterns

```
// Nested objects
interface Address {
  street: string;
  city: string;
  zipCode: string;
  country: string;
}

interface Company {
  name: string;
  address: Address;
  employees: Employee[];
}
```

```typescript
// Generic interfaces
interface Repository<T> {
  findById(id: string): Promise<T | null>;
  create(entity: T): Promise<T>;
  update(id: string, updates: Partial<T>): Promise<T>;
  delete(id: string): Promise<void>;
}

interface UserRepository extends Repository<User> {
  findByEmail(email: string): Promise<User | null>;
}

// Conditional properties
interface ApiResponse<T> {
  success: boolean;
  data: T;
  error?: string;
}

type UserResponse = ApiResponse<User>;
type UsersResponse = ApiResponse<User[]>;
```

---

8. Classes

Basic Classes

```typescript
class Person {
  // Properties
  name: string;
  age: number;
  private _id: number;
  protected email: string;

  constructor(name: string, age: number, email: string) {
    this.name = name;
    this.age = age;
    this._id = Math.random();
    this.email = email;
  }

  // Methods
  greet(): string {
    return `Hello, I'm ${this.name}`;
  }

  // Getter
  get id(): number {
    return this._id;
  }
```

```
  // Setter
  set id(value: number) {
    if (value > 0) {
      this._id = value;
    }
  }
}

let person = new Person("John", 30, "john@example.com");
console.log(person.greet());
```

Access Modifiers

```
class BankAccount {
  public accountNumber: string; // Default
  private balance: number;
  protected customerName: string;
  readonly bankName: string = "TypeScript Bank";

  constructor(accountNumber: string, initialBalance: number, customerName: string)
{
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
    this.customerName = customerName;
  }

  // Public method
  public getBalance(): number {
    return this.balance;
  }

  // Private method
  private validateAmount(amount: number): boolean {
    return amount > 0 && amount <= this.balance;
  }

  // Protected method
  protected logTransaction(type: string, amount: number): void {
    console.log(`${type}: $${amount}`);
  }

  public withdraw(amount: number): boolean {
    if (this.validateAmount(amount)) {
      this.balance -= amount;
      this.logTransaction("Withdrawal", amount);
      return true;
    }
    return false;
  }
}
```

Parameter Properties

```
// Shorthand for property declaration and initialization
class User {
  constructor(
    public name: string,
    private age: number,
    protected email: string,
    readonly id: number
  ) {
    // No need to manually assign properties
  }

  getAge(): number {
    return this.age;
  }
}

// Equivalent to:
class UserLongForm {
  public name: string;
  private age: number;
  protected email: string;
  readonly id: number;

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

Inheritance

```
class Animal {
  protected name: string;

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

  move(distance: number): void {
    console.log(`${this.name} moved ${distance} meters`);
  }
}

class Dog extends Animal {
  private breed: string;
```

```
  constructor(name: string, breed: string) {
    super(name); // Call parent constructor
    this.breed = breed;
  }

  // Override parent method
  move(distance: number): void {
    console.log("Running...");
    super.move(distance); // Call parent method
  }

  bark(): void {
    console.log(`${this.name} barks!`);
  }
}

class Bird extends Animal {
  constructor(name: string) {
    super(name);
  }

  move(distance: number): void {
    console.log("Flying...");
    super.move(distance);
  }

  fly(): void {
    console.log(`${this.name} is flying!`);
  }
}
```

Abstract Classes

```
abstract class Shape {
  protected color: string;

  constructor(color: string) {
    this.color = color;
  }

  // Abstract method - must be implemented by subclasses
  abstract calculateArea(): number;
  abstract getDescription(): string;

  // Concrete method
  getColor(): string {
    return this.color;
  }

  // Protected method for subclasses
```

```
  protected displayInfo(): void {
    console.log(`Color: ${this.color}, Area: ${this.calculateArea()}`);
  }
}

class Circle extends Shape {
  private radius: number;

  constructor(color: string, radius: number) {
    super(color);
    this.radius = radius;
  }

  calculateArea(): number {
    return Math.PI * this.radius ** 2;
  }

  getDescription(): string {
    return `A ${this.color} circle with radius ${this.radius}`;
  }
}

class Rectangle extends Shape {
  constructor(
    color: string,
    private width: number,
    private height: number
  ) {
    super(color);
  }

  calculateArea(): number {
    return this.width * this.height;
  }

  getDescription(): string {
    return `A ${this.color} rectangle ${this.width}x${this.height}`;
  }
}
```

Static Members

```
class MathUtils {
  static PI: number = 3.14159;
  private static instanceCount: number = 0;

  constructor() {
    MathUtils.instanceCount++;
  }

  static calculateCircleArea(radius: number): number {
```

```
    return MathUtils.PI * radius ** 2;
  }

  static getInstanceCount(): number {
    return MathUtils.instanceCount;
  }

  // Static block (ES2022 feature)
  static {
    console.log("MathUtils class initialized");
  }
}

// Usage
let area = MathUtils.calculateCircleArea(5);
console.log(MathUtils.PI);
```

Implementing Interfaces

```
interface Flyable {
  fly(): void;
  altitude: number;
}

interface Swimmable {
  swim(): void;
  depth: number;
}

class Duck implements Flyable, Swimmable {
  altitude: number = 0;
  depth: number = 0;

  constructor(public name: string) {}

  fly(): void {
    this.altitude = 100;
    console.log(`${this.name} is flying at ${this.altitude} feet`);
  }

  swim(): void {
    this.depth = 5;
    console.log(`${this.name} is swimming at ${this.depth} feet deep`);
  }

  quack(): void {
    console.log(`${this.name} says quack!`);
  }
}
```

9. Enums

Numeric Enums

```
// Basic numeric enum
enum Direction {
  Up,    // 0
  Down,  // 1
  Left,  // 2
  Right  // 3
}

let playerDirection: Direction = Direction.Up;

// Custom numeric values
enum Status {
  Pending = 1,
  InProgress = 2,
  Completed = 3,
  Cancelled = 4
}

// Auto-incrementing from custom start
enum Priority {
  Low = 10,
  Medium, // 11
  High,   // 12
  Critical // 13
}
```

String Enums

```
enum Color {
  Red = "red",
  Green = "green",
  Blue = "blue",
  Yellow = "yellow"
}

enum HttpStatus {
  OK = "200",
  NotFound = "404",
  InternalServerError = "500"
}

// Using string enum
function getColorHex(color: Color): string {
  switch (color) {
    case Color.Red:
      return "#FF0000";
```

```
    case Color.Green:
      return "#00FF00";
    case Color.Blue:
      return "#0000FF";
    default:
      return "#000000";
  }
}
```

Heterogeneous Enums

```
// Mixed string and numeric (not recommended)
enum Mixed {
  No = 0,
  Yes = "YES"
}
```

Const Enums

```
// Const enums are inlined at compile time
const enum Directions {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

// Usage gets inlined
let direction = Directions.Up; // Becomes: let direction = "UP";

// Regular enum comparison
enum RegularDirections {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}

// This generates actual JavaScript object
```

Reverse Mappings

```
enum Status {
  Active = 1,
  Inactive = 2,
  Pending = 3
```

```
}

// Numeric enums have reverse mappings
console.log(Status.Active); // 1
console.log(Status[1]); // "Active"
console.log(Status[2]); // "Inactive"

// String enums don't have reverse mappings
enum StringStatus {
  Active = "active",
  Inactive = "inactive"
}

console.log(StringStatus.Active); // "active"
console.log(StringStatus["active"]); // undefined
```

Enum as Types

```
enum UserRole {
  Admin = "admin",
  User = "user",
  Guest = "guest"
}

interface User {
  id: number;
  name: string;
  role: UserRole;
}

function hasPermission(user: User, requiredRole: UserRole): boolean {
  const roleHierarchy = {
    [UserRole.Guest]: 0,
    [UserRole.User]: 1,
    [UserRole.Admin]: 2
  };

  return roleHierarchy[user.role] >= roleHierarchy[requiredRole];
}

// Enum member as type
type AdminUser = {
  role: UserRole.Admin;
  permissions: string[];
};
```

Computed Enums

```
enum FileAccess {
  // Constant members
  None,
  Read = 1 << 1,
  Write = 1 << 2,
  ReadWrite = Read | Write,

  // Computed member
  G = "123".length
}
```

---

10. Union and Intersection Types

Union Types

```
// Basic union types
type StringOrNumber = string | number;
type Status = "loading" | "success" | "error";

function formatId(id: string | number): string {
  if (typeof id === "string") {
    return id.toUpperCase();
  } else {
    return id.toString();
  }
}

// Union with different object types
interface Bird {
  fly(): void;
  layEggs(): void;
}

interface Fish {
  swim(): void;
  layEggs(): void;
}

type Pet = Bird | Fish;

function movePet(pet: Pet): void {
  // Only methods available on both types can be called without type checking
  pet.layEggs(); // OK

  // Need type checking for specific methods
  if ("fly" in pet) {
    pet.fly(); // OK, pet is Bird
  } else {
    pet.swim(); // OK, pet is Fish
```

```
  }
}
```

Discriminated Unions

```
// Using literal types as discriminators
interface LoadingState {
  status: "loading";
}

interface SuccessState {
  status: "success";
  data: any;
}

interface ErrorState {
  status: "error";
  error: string;
}

type AsyncState = LoadingState | SuccessState | ErrorState;

function handleState(state: AsyncState): void {
  switch (state.status) {
    case "loading":
      console.log("Loading...");
      break;
    case "success":
      console.log("Data:", state.data); // TypeScript knows state has data
      break;
    case "error":
      console.log("Error:", state.error); // TypeScript knows state has error
      break;
  }
}

// More complex discriminated union
interface Circle {
  kind: "circle";
  radius: number;
}

interface Rectangle {
  kind: "rectangle";
  width: number;
  height: number;
}

interface Triangle {
  kind: "triangle";
  base: number;
```

```
  height: number;
}

type Shape = Circle | Rectangle | Triangle;

function calculateArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "rectangle":
      return shape.width * shape.height;
    case "triangle":
      return (shape.base * shape.height) / 2;
    default:
      // Exhaustiveness check
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

Intersection Types

```
// Combining types
interface Name {
  name: string;
}

interface Age {
  age: number;
}

type Person = Name & Age;

let person: Person = {
  name: "John",
  age: 30
};

// More complex intersections
interface Serializable {
  serialize(): string;
}

interface Timestamped {
  timestamp: Date;
}

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

```typescript
type DataObject = Serializable & Timestamped & Loggable;

class User implements DataObject {
  constructor(
    public name: string,
    public email: string,
    public timestamp: Date = new Date()
  ) {}

  serialize(): string {
    return JSON.stringify({
      name: this.name,
      email: this.email,
      timestamp: this.timestamp
    });
  }

  log(): void {
    console.log(`User: ${this.name} (${this.email})`);
  }
}

// Intersection with conflicting properties
interface A {
  prop: string;
}

interface B {
  prop: number;
}

type C = A & B; // prop: never (impossible type)

// Useful intersection patterns
type WithId<T> = T & { id: string };
type WithTimestamp<T> = T & { createdAt: Date; updatedAt: Date };

type UserWithId = WithId<{ name: string; email: string }>;
type TimestampedUser = WithTimestamp<UserWithId>;
```

Type Guards

```typescript
// Custom type guards
function isString(value: any): value is string {
  return typeof value === "string";
}

function isNumber(value: any): value is number {
  return typeof value === "number";
}
```

```
function processValue(value: string | number): string {
  if (isString(value)) {
    return value.toUpperCase(); // TypeScript knows value is string
  }
  if (isNumber(value)) {
    return value.toString(); // TypeScript knows value is number
  }
  return ""; // This should never happen
}

// Built-in type guards
function handleUnknown(value: unknown): string {
  if (typeof value === "string") {
    return value; // value is string
  }
  if (typeof value === "number") {
    return value.toString(); // value is number
  }
  if (value instanceof Date) {
    return value.toISOString(); // value is Date
  }
  return "Unknown type";
}

// Array type guards
function isStringArray(arr: any[]): arr is string[] {
  return arr.every(item => typeof item === "string");
}

// Object type guards with discriminated unions
function isBird(animal: Bird | Fish): animal is Bird {
  return "fly" in animal;
}

function isFish(animal: Bird | Fish): animal is Fish {
  return "swim" in animal;
}
```

---

11. Type Aliases

Basic Type Aliases

```
// Simple type aliases
type ID = string | number;
type UserID = string;
type ProductID = number;

// Object type aliases
type Point = {
  x: number;
```

```
  y: number;
};

type User = {
  id: UserID;
  name: string;
  email: string;
  isActive: boolean;
};

// Function type aliases
type EventHandler = (event: Event) => void;
type MathOperation = (a: number, b: number) => number;
type AsyncFunction<T> = () => Promise<T>;

// Array type aliases
type StringArray = string[];
type NumberMatrix = number[][];
type UserList = User[];
```

Generic Type Aliases

```
// Generic type aliases
type Result<T, E = Error> = {
  success: boolean;
  data?: T;
  error?: E;
};

type ApiResponse<T> = Result<T, string>;

// Usage
type UserResult = Result<User>;
type UsersResult = Result<User[]>;
type StringResult = ApiResponse<string>;

// More complex generic aliases
type KeyValuePair<K, V> = {
  key: K;
  value: V;
};

type Dictionary<T> = {
  [key: string]: T;
};

type Optional<T> = {
  [K in keyof T]?: T[K];
};

type Required<T> = {
```

```
  [K in keyof T]-?: T[K];
};
```

Conditional Type Aliases

```
// Conditional types
type NonNullable<T> = T extends null | undefined ? never : T;
type ArrayElement<T> = T extends (infer U)[] ? U : never;
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

// Examples
type StringOrNull = string | null;
type JustString = NonNullable<StringOrNull>; // string

type Numbers = number[];
type NumberElement = ArrayElement<Numbers>; // number

type AddFunction = (a: number, b: number) => number;
type AddReturnType = ReturnType<AddFunction>; // number

// More complex conditional types
type Flatten<T> = T extends any[] ? T[number] : T;
type FlattenDeep<T> = T extends (infer U)[]
  ? U extends any[]
    ? FlattenDeep<U>
    : U
  : T;

// Usage
type NestedArray = number[][];
type Flattened = Flatten<NestedArray>; // number[]
type DeepFlattened = FlattenDeep<NestedArray>; // number
```

Mapped Type Aliases

```
// Mapped types
type Readonly<T> = {
  readonly [K in keyof T]: T[K];
};

type Partial<T> = {
  [K in keyof T]?: T[K];
};

type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};

type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>;
```

```typescript
// Usage examples
interface User {
  id: string;
  name: string;
  email: string;
  age: number;
}

type ReadonlyUser = Readonly<User>;
type PartialUser = Partial<User>;
type UserNameAndEmail = Pick<User, "name" | "email">;
type UserWithoutId = Omit<User, "id">;

// Custom mapped types
type Nullable<T> = {
  [K in keyof T]: T[K] | null;
};

type Stringify<T> = {
  [K in keyof T]: string;
};

type GettersOnly<T> = {
  [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K];
};

// Usage
type NullableUser = Nullable<User>;
type StringUser = Stringify<User>;
type UserGetters = GettersOnly<User>;
```

---

12. Literal Types

String Literal Types

```typescript
// String literals
type Direction = "north" | "south" | "east" | "west";
type Color = "red" | "green" | "blue";
type Size = "small" | "medium" | "large";

function move(direction: Direction): void {
  console.log(`Moving ${direction}`);
}

move("north"); // OK
// move("up"); // Error: Argument of type '"up"' is not assignable

// Template literal types (TypeScript 4.1+)
type Greeting = `Hello, ${string}!`;
```

```
type CSSProperty = `--${string}`;
type EventName = `on${Capitalize<string>}`;

// More complex template literals
type HTTPMethod = "GET" | "POST" | "PUT" | "DELETE";
type Endpoint = `/api/${string}`;
type APICall = `${HTTPMethod} ${Endpoint}`;

// Examples
let validCall: APICall = "GET /api/users";
let anotherCall: APICall = "POST /api/users/create";
```

Numeric Literal Types

```
// Numeric literals
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;
type HttpStatusCode = 200 | 404 | 500;
type Port = 3000 | 8080 | 9000;

function rollDice(): DiceRoll {
  return (Math.floor(Math.random() * 6) + 1) as DiceRoll;
}

// Configuration with literal types
interface ServerConfig {
  port: Port;
  environment: "development" | "staging" | "production";
  logLevel: "debug" | "info" | "warn" | "error";
}

const config: ServerConfig = {
  port: 3000,
  environment: "development",
  logLevel: "debug"
};
```

Boolean Literal Types

```
// Boolean literals (less common but useful)
type True = true;
type False = false;

// Useful in conditional types
type IsArray<T> = T extends any[] ? true : false;
type StringIsArray = IsArray<string>; // false
type NumberArrayIsArray = IsArray<number[]>; // true
```

Literal Type Inference

```
// Type widening
let mutableString = "hello"; // Type: string
const immutableString = "hello"; // Type: "hello"

// Preventing widening with const assertion
let narrowString = "hello" as const; // Type: "hello"
const config = {
  apiUrl: "https://api.example.com",
  timeout: 5000
} as const; // All properties become readonly and literal

// Array const assertion
const fruits = ["apple", "banana", "cherry"] as const;
type Fruit = typeof fruits[number]; // "apple" | "banana" | "cherry"

// Object const assertion
const theme = {
  colors: {
    primary: "#007bff",
    secondary: "#6c757d",
    success: "#28a745"
  },
  sizes: {
    small: "8px",
    medium: "16px",
    large: "24px"
  }
} as const;

type ThemeColor = typeof theme.colors[keyof typeof theme.colors];
type ThemeSize = typeof theme.sizes[keyof typeof theme.sizes];
```

Literal Type Patterns

```
// Discriminated unions with literals
interface Button {
  type: "button";
  text: string;
  onClick: () => void;
}

interface Link {
  type: "link";
  href: string;
  text: string;
}

interface Input {
  type: "input";
  placeholder: string;
```

```
  value: string;
}

type UIElement = Button | Link | Input;

function renderElement(element: UIElement): string {
  switch (element.type) {
    case "button":
      return `<button>${element.text}</button>`;
    case "link":
      return `<a href="${element.href}">${element.text}</a>`;
    case "input":
      return `<input placeholder="${element.placeholder}"
value="${element.value}">`;
  }
}

// Brand types with literals
type Brand<T, B> = T & { __brand: B };
type UserId = Brand<string, "UserId">;
type ProductId = Brand<string, "ProductId">;

function createUserId(id: string): UserId {
  return id as UserId;
}

function getUserById(id: UserId): Promise<User> {
  // Implementation
  return Promise.resolve({} as User);
}

// Strong typing prevents mixing IDs
let userId = createUserId("user123");
let productId = "product456" as ProductId;

getUserById(userId); // OK
// getUserById(productId); // Error: different brand types
```

---

13. Type Assertions

Basic Type Assertions

```
// Angle bracket syntax (not recommended in JSX)
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

// As syntax (preferred)
let someValue2: any = "this is a string";
let strLength2: number = (someValue2 as string).length;
```

```
// Common use cases
let userInput = document.getElementById("user-input") as HTMLInputElement;
let canvas = document.querySelector("#canvas") as HTMLCanvasElement;

// API response assertions
interface ApiResponse {
  data: any;
  status: number;
}

async function fetchData(): Promise<User[]> {
  const response = await fetch("/api/users");
  const json = await response.json() as ApiResponse;
  return json.data as User[];
}
```

Non-null Assertion Operator

```
// Non-null assertion (!)
function processUser(user: User | null): void {
  // Without assertion - TypeScript error
  // console.log(user.name); // Error: Object is possibly 'null'

  // With non-null assertion
  console.log(user!.name); // Tells TypeScript user is not null

  // Better approach - type guard
  if (user) {
    console.log(user.name); // Type guard makes this safe
  }
}

// DOM manipulation
let button = document.querySelector("#submit-btn")!; // Assert not null
button.addEventListener("click", handleClick);

// Optional chaining vs non-null assertion
let user: User | undefined = getUser();

// Non-null assertion (risky)
console.log(user!.name);

// Optional chaining (safer)
console.log(user?.name);
```

Const Assertions

```
// Const assertions for immutable data
let person = {
```

```
  name: "John",
  age: 30,
  hobbies: ["reading", "gaming"]
} as const;

// person.name = "Jane"; // Error: Cannot assign to 'name'
// person.hobbies.push("cooking"); // Error: Property 'push' does not exist

// Array const assertion
let colors = ["red", "green", "blue"] as const;
type Color = typeof colors[number]; // "red" | "green" | "blue"

// Tuple const assertion
let coordinates = [10, 20] as const; // readonly [10, 20]
let mutableCoords = [10, 20]; // number[]
```

Double Assertions

```
// Double assertion (use with extreme caution)
let input: unknown = "hello world";

// This would error:
// let strLength = (input as string).length; // Error if input is not string

// Double assertion (unsafe but sometimes necessary)
let anything = input as any as string;
let strLength = anything.length;

// Better approach with type guards
function isString(value: unknown): value is string {
  return typeof value === "string";
}

if (isString(input)) {
  let strLength = input.length; // Safe
}
```

Assertion Functions

```
// Assertion functions (TypeScript 3.7+)
function assertIsNumber(value: any): asserts value is number {
  if (typeof value !== "number") {
    throw new Error("Expected number");
  }
}

function assertIsString(value: any): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Expected string");
```

```
  }
}

function processValue(value: unknown): string {
  assertIsString(value);
  // After assertion, TypeScript knows value is string
  return value.toUpperCase();
}

// Custom assertion function
function assertUser(obj: any): asserts obj is User {
  if (!obj || typeof obj !== "object") {
    throw new Error("Invalid user object");
  }
  if (typeof obj.name !== "string") {
    throw new Error("User must have a name");
  }
  if (typeof obj.email !== "string") {
    throw new Error("User must have an email");
  }
}

// Usage
function handleUserData(data: unknown): void {
  assertUser(data);
  // TypeScript now knows data is User
  console.log(data.name, data.email);
}
```

Safe Type Assertions

```
// Safe assertion patterns
function safeParseInt(value: string): number | null {
  const parsed = parseInt(value, 10);
  return isNaN(parsed) ? null : parsed;
}

function safeJsonParse<T>(json: string): T | null {
  try {
    return JSON.parse(json) as T;
  } catch {
    return null;
  }
}

// Runtime type checking with assertions
function isValidUser(obj: any): obj is User {
  return obj &&
    typeof obj === "object" &&
    typeof obj.name === "string" &&
    typeof obj.email === "string" &&
```

```
    typeof obj.age === "number";
}

function processUserData(data: unknown): User | null {
  if (isValidUser(data)) {
    return data; // TypeScript knows this is User
  }
  return null;
}

// Using libraries like zod for runtime validation
// import { z } from 'zod';
//
// const UserSchema = z.object({
//   name: z.string(),
//   email: z.string().email(),
//   age: z.number().min(0)
// });
//
// function validateUser(data: unknown): User {
//   return UserSchema.parse(data); // Throws if invalid
// }
```

---

14. Generics

Basic Generics

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

let stringIdentity = identity<string>("hello");
let numberIdentity = identity<number>(42);
let booleanIdentity = identity(true); // Type inference

// Generic with multiple type parameters
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

let stringNumberPair = pair("hello", 42);
let booleanStringPair = pair<boolean, string>(true, "world");

// Generic array function
function getFirstElement<T>(arr: T[]): T | undefined {
  return arr[0];
}
```

```
let firstString = getFirstElement(["a", "b", "c"]); // string | undefined
let firstNumber = getFirstElement([1, 2, 3]); // number | undefined
```

Generic Interfaces

```
// Generic interface
interface Box<T> {
  value: T;
}

let stringBox: Box<string> = { value: "hello" };
let numberBox: Box<number> = { value: 42 };

// Generic interface with multiple parameters
interface KeyValuePair<K, V> {
  key: K;
  value: V;
}

let userIdName: KeyValuePair<number, string> = {
  key: 123,
  value: "John Doe"
};

// Generic interface with methods
interface Repository<T> {
  findById(id: string): Promise<T | null>;
  findAll(): Promise<T[]>;
  create(entity: Omit<T, "id">): Promise<T>;
  update(id: string, entity: Partial<T>): Promise<T>;
  delete(id: string): Promise<void>;
}

// Implementation
class UserRepository implements Repository<User> {
  async findById(id: string): Promise<User | null> {
    // Implementation
    return null;
  }

  async findAll(): Promise<User[]> {
    // Implementation
    return [];
  }

  async create(userData: Omit<User, "id">): Promise<User> {
    // Implementation
    return { id: "123", ...userData };
  }

  async update(id: string, userData: Partial<User>): Promise<User> {
```

```
    // Implementation
    return {} as User;
  }

  async delete(id: string): Promise<void> {
    // Implementation
  }
}
```

Generic Classes

```
// Generic class
class Stack<T> {
  private items: T[] = [];

  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }

  peek(): T | undefined {
    return this.items[this.items.length - 1];
  }

  isEmpty(): boolean {
    return this.items.length === 0;
  }

  size(): number {
    return this.items.length;
  }
}

let numberStack = new Stack<number>();
numberStack.push(1);
numberStack.push(2);
let lastNumber = numberStack.pop(); // number | undefined

let stringStack = new Stack<string>();
stringStack.push("hello");
stringStack.push("world");
```

Generic Constraints

```
// Basic constraints
interface Lengthwise {
```

```
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}

logLength("hello"); // OK, string has length
logLength([1, 2, 3]); // OK, array has length
logLength({ length: 10, value: 3 }); // OK, object has length
// logLength(3); // Error: number doesn't have length

// Using keyof constraint
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

let person = { name: "John", age: 30, city: "New York" };
let name = getProperty(person, "name"); // string
let age = getProperty(person, "age"); // number
// let invalid = getProperty(person, "salary"); // Error: invalid key

// Multiple constraints
interface Serializable {
  serialize(): string;
}

interface Timestamped {
  timestamp: Date;
}

function processEntity<T extends Serializable & Timestamped>(entity: T): string {
  return `${entity.serialize()} (${entity.timestamp.toISOString()})`;
}

// Conditional constraints
type NonNullable<T> = T extends null | undefined ? never : T;
type ElementType<T> = T extends (infer U)[] ? U : T;
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Advanced Generic Patterns

```
// Generic factories
interface Constructable<T = {}> {
  new (...args: any[]): T;
}

function createInstance<T extends Constructable>(ctor: T, ...args: any[]):
InstanceType<T> {
  return new ctor(...args);
```

```
}

class User {
  constructor(public name: string) {}
}

class Product {
  constructor(public title: string, public price: number) {}
}

let user = createInstance(User, "John");
let product = createInstance(Product, "Laptop", 999);

// Generic decorators
function memoize<T extends (...args: any[]) => any>(fn: T): T {
  const cache = new Map();

  return ((...args: any[]) => {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }

    const result = fn(...args);
    cache.set(key, result);
    return result;
  }) as T;
}

// Higher-order generic functions
function compose<T, U, V>(f: (x: U) => V, g: (x: T) => U): (x: T) => V {
  return (x: T) => f(g(x));
}

const addOne = (x: number) => x + 1;
const toString = (x: number) => x.toString();
const addOneAndStringify = compose(toString, addOne);

let result = addOneAndStringify(5); // "6"

// Generic conditional types
type ApiResponse<T> = T extends string
  ? { message: T }
  : T extends number
  ? { count: T }
  : T extends boolean
  ? { success: T }
  : { data: T };

type StringResponse = ApiResponse<string>; // { message: string }
type NumberResponse = ApiResponse<number>; // { count: number }
type BooleanResponse = ApiResponse<boolean>; // { success: boolean }
type ObjectResponse = ApiResponse<User>; // { data: User }
```

Generic Utility Functions

```typescript
// Deep clone function
function deepClone<T>(obj: T): T {
  if (obj === null || typeof obj !== "object") {
    return obj;
  }

  if (obj instanceof Date) {
    return new Date(obj.getTime()) as T;
  }

  if (Array.isArray(obj)) {
    return obj.map(item => deepClone(item)) as T;
  }

  const cloned = {} as T;
  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      cloned[key] = deepClone(obj[key]);
    }
  }

  return cloned;
}

// Generic event emitter
class EventEmitter<T extends Record<string, any[]>> {
  private listeners: Partial<{
    [K in keyof T]: ((...args: T[K]) => void)[]
  }> = {};

  on<K extends keyof T>(event: K, listener: (...args: T[K]) => void): void {
    if (!this.listeners[event]) {
      this.listeners[event] = [];
    }
    this.listeners[event]!.push(listener);
  }

  emit<K extends keyof T>(event: K, ...args: T[K]): void {
    const eventListeners = this.listeners[event];
    if (eventListeners) {
      eventListeners.forEach(listener => listener(...args));
    }
  }
}

// Usage
interface Events {
  userLogin: [User];
  userLogout: [string];
```

```
  dataUpdate: [string, any];
}

const emitter = new EventEmitter<Events>();

emitter.on("userLogin", (user) => {
  console.log(`User ${user.name} logged in`);
});

emitter.on("dataUpdate", (key, value) => {
  console.log(`Data updated: ${key} = ${value}`);
});

emitter.emit("userLogin", { id: "1", name: "John", email: "john@example.com" });
emitter.emit("dataUpdate", "theme", "dark");
```

---

15. Utility Types

Built-in Utility Types

```
interface User {
  id: string;
  name: string;
  email: string;
  age: number;
  isActive: boolean;
}

// Partial<T> - makes all properties optional
type PartialUser = Partial<User>;
// { id?: string; name?: string; email?: string; age?: number; isActive?: boolean;
}

function updateUser(id: string, updates: Partial<User>): User {
  // Implementation
  return {} as User;
}

// Required<T> - makes all properties required
interface OptionalUser {
  id?: string;
  name?: string;
  email?: string;
}

type RequiredUser = Required<OptionalUser>;
// { id: string; name: string; email: string; }

// Readonly<T> - makes all properties readonly
type ReadonlyUser = Readonly<User>;
```

```
// { readonly id: string; readonly name: string; ... }

let user: ReadonlyUser = {
  id: "1",
  name: "John",
  email: "john@example.com",
  age: 30,
  isActive: true
};
// user.name = "Jane"; // Error: Cannot assign to 'name'
```

Pick and Omit

```
// Pick<T, K> - creates type with only specified properties
type UserNameAndEmail = Pick<User, "name" | "email">;
// { name: string; email: string; }

type UserIdentity = Pick<User, "id" | "name">;
// { id: string; name: string; }

// Omit<T, K> - creates type without specified properties
type UserWithoutId = Omit<User, "id">;
// { name: string; email: string; age: number; isActive: boolean; }

type PublicUser = Omit<User, "id" | "isActive">;
// { name: string; email: string; age: number; }

// Practical usage
function createUser(userData: Omit<User, "id">): User {
  return {
    id: generateId(),
    ...userData
  };
}

function getPublicUserInfo(user: User): PublicUser {
  const { id, isActive, ...publicInfo } = user;
  return publicInfo;
}
```

Record and Extract

```
// Record<K, T> - creates object type with specified keys and values
type UserRoles = "admin" | "user" | "guest";
type RolePermissions = Record<UserRoles, string[]>;
// { admin: string[]; user: string[]; guest: string[]; }

const permissions: RolePermissions = {
  admin: ["read", "write", "delete"],
```

```
  user: ["read", "write"],
  guest: ["read"]
};

type HttpStatus = Record<number, string>;
const statusMessages: HttpStatus = {
  200: "OK",
  404: "Not Found",
  500: "Internal Server Error"
};

// Extract<T, U> - extracts types from union that are assignable to U
type StringOrNumber = string | number | boolean;
type OnlyStringOrNumber = Extract<StringOrNumber, string | number>;
// string | number

// Exclude<T, U> - excludes types from union that are assignable to U
type WithoutBoolean = Exclude<StringOrNumber, boolean>;
// string | number

// NonNullable<T> - excludes null and undefined
type MaybeString = string | null | undefined;
type DefinitelyString = NonNullable<MaybeString>;
// string
```

Function Utility Types

```
// ReturnType<T> - extracts return type of function
function getUser(): User {
  return {} as User;
}

function getUsers(): User[] {
  return [];
}

type GetUserReturn = ReturnType<typeof getUser>; // User
type GetUsersReturn = ReturnType<typeof getUsers>; // User[]

// Parameters<T> - extracts parameter types as tuple
function createUser(name: string, email: string, age: number): User {
  return {} as User;
}

type CreateUserParams = Parameters<typeof createUser>;
// [string, string, number]

// ConstructorParameters<T> - extracts constructor parameter types
class Database {
  constructor(host: string, port: number, username: string) {}
}
```

```
type DatabaseConstructorParams = ConstructorParameters<typeof Database>;
// [string, number, string]

// InstanceType<T> - extracts instance type of constructor
type DatabaseInstance = InstanceType<typeof Database>;
// Database

// ThisParameterType<T> and OmitThisParameter<T>
function greetUser(this: User, greeting: string): string {
  return `${greeting}, ${this.name}!`;
}

type GreetThisType = ThisParameterType<typeof greetUser>; // User
type GreetWithoutThis = OmitThisParameter<typeof greetUser>; // (greeting: string)
=> string
```

Advanced Utility Patterns

```
// Custom utility types
type DeepPartial<T> = {
  [P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];
};

type DeepReadonly<T> = {
  readonly [P in keyof T]: T[P] extends object ? DeepReadonly<T[P]> : T[P];
};

// Nullable and Optional utilities
type Nullable<T> = T | null;
type Optional<T, K extends keyof T> = Omit<T, K> & Partial<Pick<T, K>>;

type OptionalEmailUser = Optional<User, "email">;
// { id: string; name: string; age: number; isActive: boolean; email?: string; }

// Key transformation utilities
type PrefixKeys<T, P extends string> = {
  [K in keyof T as `${P}${string & K}`]: T[K];
};

type SuffixKeys<T, S extends string> = {
  [K in keyof T as `${string & K}${S}`]: T[K];
};

type UserWithPrefix = PrefixKeys<User, "user_">;
// { user_id: string; user_name: string; user_email: string; ... }

// Conditional utility types
type NonFunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];
```

```typescript
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

class UserService {
  id: string = "";
  name: string = "";
  getName(): string { return this.name; }
  setName(name: string): void { this.name = name; }
}

type UserServiceData = NonFunctionProperties<UserService>;
// { id: string; name: string; }
```

---

16. Advanced Types

Conditional Types

```typescript
// Basic conditional types
type IsString<T> = T extends string ? true : false;

type Test1 = IsString<string>; // true
type Test2 = IsString<number>; // false

// Conditional types with infer
type GetArrayType<T> = T extends (infer U)[] ? U : never;

type StringArray = GetArrayType<string[]>; // string
type NumberArray = GetArrayType<number[]>; // number

// Nested conditional types
type Flatten<T> = T extends any[]
  ? T[number] extends any[]
    ? Flatten<T[number]>
    : T[number]
  : T;

type NestedArray = number[][][];
type Flattened = Flatten<NestedArray>; // number

// Distributive conditional types
type ToArray<T> = T extends any ? T[] : never;
type StringOrNumberArray = ToArray<string | number>; // string[] | number[]
```

Mapped Types

```typescript
// Basic mapped types
type ReadonlyVersion<T> = {
  readonly [P in keyof T]: T[P];
```

```
};

type OptionalVersion<T> = {
  [P in keyof T]?: T[P];
};

// Mapped types with key remapping
type Getters<T> = {
  [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K];
};

type Setters<T> = {
  [K in keyof T as `set${Capitalize<string & K>}`]: (value: T[K]) => void;
};

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

type PersonGetters = Getters<Person>;
// { getName: () => string; getAge: () => number; }

type PersonSetters = Setters<Person>;
// { setName: (value: string) => void; setAge: (value: number) => void; }

// Filtering properties with mapped types
type PickByType<T, U> = {
  [P in keyof T as T[P] extends U ? P : never]: T[P];
};

type StringProperties = PickByType<Person, string>; // { name: string; }
type NumberProperties = PickByType<Person, number>; // { age: number; }
```

Template Literal Types

```
// Basic template literal types
type Greeting = `Hello, ${string}!`;
type WelcomeMessage = `Welcome, ${string}`;

let greeting: Greeting = "Hello, John!"; // ✓
// let invalid: Greeting = "Hi, John!"; // ✗

// Template literals with unions
type EventName = "click" | "focus" | "blur";
type HandlerName = `on${Capitalize<EventName>}`;
// "onClick" | "onFocus" | "onBlur"

// Pattern matching with template literals
type ExtractRouteParams<T extends string> =
  T extends `${string}:${infer Param}/${infer Rest}`
```

```
    ? { [K in Param | keyof ExtractRouteParams<Rest>]: string }
    : T extends `${string}:${infer Param}`
    ? { [K in Param]: string }
    : {};

type Route = "/users/:id/posts/:postId";
type RouteParams = ExtractRouteParams<Route>;
// { id: string; postId: string; }

// SQL-like template literals
type SQLSelectClause<T> = `SELECT ${keyof T & string} FROM ${string}`;
type UserSelect = SQLSelectClause<User>;
// "SELECT id FROM users" | "SELECT name FROM users" | ...
```

---

17. Modules and Namespaces

ES6 Modules

```
// user.ts - Named exports
export interface User {
  id: string;
  name: string;
  email: string;
}

export class UserService {
  getUser(id: string): User {
    // Implementation
    return {} as User;
  }
}

export const DEFAULT_USER: User = {
  id: "0",
  name: "Guest",
  email: "guest@example.com"
};

// Default export
export default class Database {
  connect(): void {
    console.log("Connected to database");
  }
}

// main.ts - Importing
import Database from './user'; // Default import
import { User, UserService, DEFAULT_USER } from './user'; // Named imports
import * as UserModule from './user'; // Namespace import
import { User as UserType } from './user'; // Aliased import
```

```
// Re-exporting
export { User, UserService } from './user';
export { default as Database } from './user';
```

Module Resolution

```
// tsconfig.json module resolution
{
  "compilerOptions": {
    "moduleResolution": "node",
    "baseUrl": "./src",
    "paths": {
      "@/*": ["*"],
      "@/components/*": ["components/*"],
      "@/utils/*": ["utils/*"]
    }
  }
}

// Using path mapping
import { Button } from '@/components/Button';
import { formatDate } from '@/utils/date';

// Ambient modules for third-party libraries
declare module "my-library" {
  export function doSomething(): void;
  export interface Config {
    apiKey: string;
  }
}

// Global augmentation
declare global {
  interface Window {
    myCustomProperty: string;
  }
}

window.myCustomProperty = "Hello";
```

Namespaces

```
// Internal namespaces
namespace Geometry {
  export interface Point {
    x: number;
    y: number;
  }
```

```
  export class Circle {
    constructor(public center: Point, public radius: number) {}

    area(): number {
      return Math.PI * this.radius ** 2;
    }
  }

  export namespace Utils {
    export function distance(p1: Point, p2: Point): number {
      return Math.sqrt((p2.x - p1.x) ** 2 + (p2.y - p1.y) ** 2);
    }
  }
}

// Using namespaces
const point: Geometry.Point = { x: 0, y: 0 };
const circle = new Geometry.Circle(point, 5);
const dist = Geometry.Utils.distance(point, { x: 3, y: 4 });

// Namespace merging
namespace Animals {
  export class Dog {
    breed: string = "";
  }
}

namespace Animals {
  export class Cat extends Dog {
    meow(): void {
      console.log("Meow!");
    }
  }
}

// Both Dog and Cat are available
const dog = new Animals.Dog();
const cat = new Animals.Cat();
```

---

18. Decorators

Class Decorators

```
// Enable decorators in tsconfig.json
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
```

```
}

// Class decorator
function Entity(tableName: string) {
  return function <T extends { new (...args: any[]): {} }>(constructor: T) {
    return class extends constructor {
      tableName = tableName;
    };
  };
}

@Entity("users")
class User {
  id: number = 0;
  name: string = "";
}

const user = new User();
console.log((user as any).tableName); // "users"

// Multiple decorators
function Timestamped<T extends { new (...args: any[]): {} }>(constructor: T) {
  return class extends constructor {
    createdAt = new Date();
    updatedAt = new Date();
  };
}

@Entity("posts")
@Timestamped
class Post {
  title: string = "";
  content: string = "";
}
```

Method Decorators

```
// Method decorator for logging
function Log(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyKey} with arguments:`, args);
    const result = originalMethod.apply(this, args);
    console.log(`${propertyKey} returned:`, result);
    return result;
  };
}

// Async method decorator
function Timeout(ms: number) {
```

```
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    const originalMethod = descriptor.value;

    descriptor.value = async function (...args: any[]) {
      const timeoutPromise = new Promise((_, reject) =>
        setTimeout(() => reject(new Error('Timeout')), ms)
      );

      return Promise.race([
        originalMethod.apply(this, args),
        timeoutPromise
      ]);
    };
  };
}

class ApiService {
  @Log
  getData(id: string): string {
    return `Data for ${id}`;
  }

  @Timeout(5000)
  async fetchUser(id: string): Promise<User> {
    // Simulated API call
    return new Promise(resolve => {
      setTimeout(() => resolve({ id, name: "John", email: "john@example.com" }),
1000);
    });
  }
}
```

Property and Parameter Decorators

```
// Property decorator
function MinLength(length: number) {
  return function (target: any, propertyKey: string) {
    let value: string;

    const getter = () => value;
    const setter = (newVal: string) => {
      if (newVal.length < length) {
        throw new Error(`${propertyKey} must be at least ${length} characters
long`);
      }
      value = newVal;
    };

    Object.defineProperty(target, propertyKey, {
      get: getter,
```

```
      set: setter,
      enumerable: true,
      configurable: true
    });
  };
}

// Parameter decorator
function Required(target: any, propertyKey: string, parameterIndex: number) {
  const existingRequiredParameters: number[] =
    Reflect.getOwnMetadata('required', target, propertyKey) || [];
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata('required', existingRequiredParameters, target,
propertyKey);
}

class UserValidator {
  @MinLength(3)
  username: string = "";

  validateUser(@Required id: string, @Required name: string): boolean {
    // Validation logic
    return true;
  }
}
```

---

19. Declaration Files

Creating Declaration Files

```
// math-utils.d.ts - Ambient declarations
declare module "math-utils" {
  export function add(a: number, b: number): number;
  export function multiply(a: number, b: number): number;
  export const PI: number;

  export interface Calculator {
    add(a: number, b: number): number;
    subtract(a: number, b: number): number;
  }

  export class ScientificCalculator implements Calculator {
    add(a: number, b: number): number;
    subtract(a: number, b: number): number;
    sin(x: number): number;
    cos(x: number): number;
  }
}

// Global declarations
```

```
declare global {
  interface Window {
    gtag: (command: string, ...args: any[]) => void;
  }

  var MY_GLOBAL_VAR: string;

  namespace NodeJS {
    interface ProcessEnv {
      NODE_ENV: 'development' | 'production' | 'test';
      API_KEY: string;
    }
  }
}

// Utility type declarations
declare type JSONValue =
  | string
  | number
  | boolean
  | null
  | JSONValue[]
  | {[key: string]: JSONValue};

declare interface APIResponse<T = any> {
  data: T;
  status: number;
  message: string;
}
```

Module Augmentation

```
// Extending existing modules
declare module "express" {
  interface Request {
    user?: {
      id: string;
      email: string;
    };
  }
}

// Now you can use the extended interface
import { Request, Response } from 'express';

function authMiddleware(req: Request, res: Response, next: Function) {
  req.user = { id: "123", email: "user@example.com" };
  next();
}

// Augmenting global types
```

```
declare module "*.vue" {
  import Vue from 'vue';
  export default Vue;
}

declare module "*.json" {
  const value: any;
  export default value;
}
```

---

20. Error Handling

Basic Error Handling

```
// Custom Error classes
class ValidationError extends Error {
  constructor(
    message: string,
    public field: string,
    public value: any
  ) {
    super(message);
    this.name = 'ValidationError';
  }
}

class NetworkError extends Error {
  constructor(
    message: string,
    public statusCode: number,
    public url: string
  ) {
    super(message);
    this.name = 'NetworkError';
  }
}

// Error handling with Result pattern
type Result<T, E = Error> = {
  success: true;
  data: T;
} | {
  success: false;
  error: E;
};

function safeDivide(a: number, b: number): Result<number> {
  if (b === 0) {
    return { success: false, error: new Error("Division by zero") };
  }
```

```
  return { success: true, data: a / b };
}

// Using Result pattern
const result = safeDivide(10, 2);
if (result.success) {
  console.log(result.data); // TypeScript knows this is number
} else {
  console.error(result.error.message); // TypeScript knows this is Error
}
```

Async Error Handling

```
// Promise-based error handling
async function fetchUser(id: string): Promise<User> {
  try {
    const response = await fetch(`/api/users/${id}`);

    if (!response.ok) {
      throw new NetworkError(
        `Failed to fetch user: ${response.statusText}`,
        response.status,
        response.url
      );
    }

    const userData = await response.json();
    return userData as User;
  } catch (error) {
    if (error instanceof NetworkError) {
      console.error('Network issue:', error.statusCode);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
}

// Error boundaries for React components
interface ErrorBoundaryState {
  hasError: boolean;
  error?: Error;
}

class ErrorBoundary extends React.Component<
  React.PropsWithChildren<{}>,
  ErrorBoundaryState
> {
  constructor(props: React.PropsWithChildren<{}>) {
    super(props);
    this.state = { hasError: false };
```

```
  }

  static getDerivedStateFromError(error: Error): ErrorBoundaryState {
    return { hasError: true, error };
  }

  componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {
    console.error('Error caught by boundary:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong: {this.state.error?.message}</h1>;
    }
    return this.props.children;
  }
}
```

Validation and Error Types

```
// Schema validation with detailed errors
interface ValidationResult<T> {
  isValid: boolean;
  data?: T;
  errors: ValidationError[];
}

function validateUser(input: unknown): ValidationResult<User> {
  const errors: ValidationError[] = [];

  if (typeof input !== 'object' || input === null) {
    return { isValid: false, errors: [new ValidationError('Input must be an
object', 'root', input)] };
  }

  const obj = input as Record<string, unknown>;

  if (typeof obj.id !== 'string' || obj.id.length === 0) {
    errors.push(new ValidationError('ID is required and must be a non-empty
string', 'id', obj.id));
  }

  if (typeof obj.name !== 'string' || obj.name.length < 2) {
    errors.push(new ValidationError('Name must be at least 2 characters long',
'name', obj.name));
  }

  if (typeof obj.email !== 'string' || !obj.email.includes('@')) {
    errors.push(new ValidationError('Email must be a valid email address',
'email', obj.email));
  }
```

```
  if (errors.length > 0) {
    return { isValid: false, errors };
  }

  return {
    isValid: true,
    data: obj as User,
    errors: []
  };
}

// Type-safe error handling utility
type ErrorMap = {
  validation: ValidationError;
  network: NetworkError;
  generic: Error;
};

function handleError<K extends keyof ErrorMap>(
  error: unknown,
  type: K
): error is ErrorMap[K] {
  switch (type) {
    case 'validation':
      return error instanceof ValidationError;
    case 'network':
      return error instanceof NetworkError;
    case 'generic':
      return error instanceof Error;
    default:
      return false;
  }
}
```

---

21. Best Practices

Code Organization

```
// Use barrel exports for clean imports
// index.ts
export { User, UserService } from './user';
export { Product, ProductService } from './product';
export { Order, OrderService } from './order';

// Import from barrel
import { User, Product, Order } from './models';

// Prefer interfaces over type aliases for object shapes
interface ApiResponse<T> {
```

```
  data: T;
  status: number;
  message: string;
}

// Use type aliases for unions, primitives, and computed types
type Status = 'loading' | 'success' | 'error';
type EventHandler<T> = (event: T) => void;

// Consistent naming conventions
interface UserPreferences {        // PascalCase for interfaces
  theme: 'light' | 'dark';        // camelCase for properties
  language: string;
}

const API_BASE_URL = 'https://api.example.com';  // UPPER_SNAKE_CASE for constants
const userService = new UserService();           // camelCase for variables

// Use readonly for immutable data
interface ReadonlyConfig {
  readonly apiKey: string;
  readonly endpoints: readonly string[];
  readonly retryAttempts: number;
}
```

Type Safety Best Practices

```
// Prefer unknown over any
function processApiResponse(response: unknown): User {
  // Type guard to safely handle unknown
  if (isUser(response)) {
    return response;
  }
  throw new Error('Invalid user data');
}

function isUser(obj: unknown): obj is User {
  return (
    typeof obj === 'object' &&
    obj !== null &&
    typeof (obj as User).id === 'string' &&
    typeof (obj as User).name === 'string' &&
    typeof (obj as User).email === 'string'
  );
}

// Use discriminated unions for state management
type LoadingState = { status: 'loading' };
type SuccessState = { status: 'success'; data: User[] };
type ErrorState = { status: 'error'; error: string };
```

```
type AsyncState = LoadingState | SuccessState | ErrorState;

function handleState(state: AsyncState) {
  switch (state.status) {
    case 'loading':
      return <div>Loading...</div>;
    case 'success':
      return <UserList users={state.data} />;
    case 'error':
      return <div>Error: {state.error}</div>;
    default:
      // TypeScript ensures exhaustive checking
      const exhaustiveCheck: never = state;
      return exhaustiveCheck;
  }
}

// Use assertion signatures for runtime checks
function assertIsNumber(value: unknown): asserts value is number {
  if (typeof value !== 'number') {
    throw new Error('Expected number');
  }
}

function processNumber(input: unknown) {
  assertIsNumber(input);
  // TypeScript now knows input is number
  return input * 2;
}
```

Performance and Optimization

```
// Use const assertions for literal types
const themes = ['light', 'dark', 'auto'] as const;
type Theme = typeof themes[number]; // 'light' | 'dark' | 'auto'

// Prefer type-only imports/exports
import type { User } from './types';
export type { ApiResponse } from './api';

// Use satisfies operator for type checking without widening
const config = {
  development: {
    apiUrl: 'http://localhost:3000',
    debug: true
  },
  production: {
    apiUrl: 'https://api.example.com',
    debug: false
  }
} satisfies Record<string, { apiUrl: string; debug: boolean }>;
```

```
// TypeScript knows the exact structure
config.development.debug; // boolean, not just any

// Lazy loading with dynamic imports
async function loadUserModule() {
  const { UserService } = await import('./services/UserService');
  return new UserService();
}

// Generic constraints for better type inference
interface Identifiable {
  id: string;
}

function updateEntity<T extends Identifiable>(
  entities: T[],
  id: string,
  updates: Partial<Omit<T, 'id'>>
): T[] {
  return entities.map(entity =>
    entity.id === id ? { ...entity, ...updates } : entity
  );
}
```

Testing Best Practices

```
// Type-safe mocking
interface DatabaseConnection {
  query<T>(sql: string, params: unknown[]): Promise<T[]>;
  close(): Promise<void>;
}

// Mock implementation
const mockDb: DatabaseConnection = {
  query: jest.fn().mockResolvedValue([]),
  close: jest.fn().mockResolvedValue(undefined)
};

// Type-safe test utilities
function createMockUser(overrides: Partial<User> = {}): User {
  return {
    id: '1',
    name: 'Test User',
    email: 'test@example.com',
    age: 25,
    isActive: true,
    ...overrides
  };
}
```

```
// Property-based testing types
type UserGenerator = () => User;

const generateRandomUser: UserGenerator = () => ({
  id: Math.random().toString(36),
  name: `User ${Math.random()}`,
  email: `user${Math.random()}@example.com`,
  age: Math.floor(Math.random() * 100),
  isActive: Math.random() > 0.5
});
```

---

22. What to Do Next

Learning Path

```
// 1. Master Advanced TypeScript Patterns
// Study these advanced concepts:

// Branded types for type safety
type UserId = string & { readonly brand: unique symbol };
type Email = string & { readonly brand: unique symbol };

function createUserId(id: string): UserId {
  return id as UserId;
}

function createEmail(email: string): Email {
  if (!email.includes('@')) throw new Error('Invalid email');
  return email as Email;
}

// Builder pattern with TypeScript
class QueryBuilder<T> {
  private query: string = '';

  select<K extends keyof T>(fields: K[]): QueryBuilder<Pick<T, K>> {
    this.query += `SELECT ${fields.join(', ')} `;
    return this as any;
  }

  from(table: string): this {
    this.query += `FROM ${table} `;
    return this;
  }

  where(condition: string): this {
    this.query += `WHERE ${condition} `;
    return this;
  }
```

```
  build(): string {
    return this.query.trim();
  }
}

const query = new QueryBuilder<User>()
  .select(['id', 'name'])
  .from('users')
  .where('age > 18')
  .build();
```

Real-World Projects

```
// 2. Build Type-Safe APIs
// Example: Express.js with TypeScript

interface ApiEndpoint<TRequest, TResponse> {
  method: 'GET' | 'POST' | 'PUT' | 'DELETE';
  path: string;
  handler: (req: TRequest) => Promise<TResponse> | TResponse;
}

type GetUsersRequest = { query: { page?: string; limit?: string } };
type GetUsersResponse = { users: User[]; total: number };

const getUsersEndpoint: ApiEndpoint<GetUsersRequest, GetUsersResponse> = {
  method: 'GET',
  path: '/users',
  handler: async (req) => {
    const page = parseInt(req.query.page || '1');
    const limit = parseInt(req.query.limit || '10');

    // Implementation
    return { users: [], total: 0 };
  }
};

// 3. Frontend Framework Integration
// React with TypeScript

interface ComponentProps<T> {
  data: T;
  onUpdate: (data: T) => void;
  loading?: boolean;
}

function DataComponent<T>({ data, onUpdate, loading = false }: ComponentProps<T>)
{
  // Generic component implementation
  return null;
}
```

```
// Usage with type inference
<DataComponent
  data={user}
  onUpdate={(updatedUser) => {
    // TypeScript knows updatedUser is User type
  }}
/>
```

Tools and Ecosystem

```
// 3. Essential Tools to Learn

// Zod for runtime validation
import { z } from 'zod';

const UserSchema = z.object({
  id: z.string(),
  name: z.string().min(2),
  email: z.string().email(),
  age: z.number().positive()
});

type User = z.infer<typeof UserSchema>; // Automatically generates TypeScript type

// tRPC for type-safe APIs
import { router, publicProcedure } from './trpc';

const appRouter = router({
  getUser: publicProcedure
    .input(z.object({ id: z.string() }))
    .query(async ({ input }) => {
      // Return type automatically inferred
      return getUserById(input.id);
    }),
});

export type AppRouter = typeof appRouter;

// Prisma for type-safe database access
// Generates TypeScript types from database schema
const user = await prisma.user.findUnique({
  where: { id: '1' },
  include: { posts: true } // Type-safe includes
});
// user type is automatically inferred with posts included
```

Next Steps Checklist

```
// 4. Progressive Learning Goals

// ☑ Beginner Level
// - Master basic types and interfaces
// - Understand generic basics
// - Use utility types effectively
// - Write type-safe functions

// ☑ Intermediate Level
// - Create custom utility types
// - Use conditional types
// - Implement design patterns with TypeScript
// - Build type-safe APIs

// ☑ Advanced Level
// - Master template literal types
// - Create complex mapped types
// - Use assertion signatures effectively
// - Contribute to TypeScript ecosystem

// Practice Projects:
const practiceProjects = [
  'Build a type-safe REST API with Express',
  'Create a React component library with TypeScript',
  'Implement a type-safe state management solution',
  'Build a CLI tool with TypeScript and Node.js',
  'Create type definitions for a JavaScript library',
  'Build a GraphQL API with type generation'
] as const;

// Community Resources:
const resources = {
  documentation: 'https://www.typescriptlang.org/docs/',
  playground: 'https://www.typescriptlang.org/play',
  handbook: 'https://www.typescriptlang.org/docs/handbook/intro.html',
  github: 'https://github.com/Microsoft/TypeScript',
  stackoverflow: 'Search for "typescript" tag',
  reddit: 'r/typescript community'
} as const;

// Keep Learning!
console.log('🚀 Continue your TypeScript journey!');
```

This completes your comprehensive TypeScript cheatsheet. Practice these concepts in real projects, contribute to open source, and keep exploring the evolving TypeScript ecosystem!