#_ Getting Started with TypeScript with +100 Concepts

1- Types: TypeScript includes built-in types like number, string, and boolean.

```
let isDone: boolean = false;
let decimal: number = 6;
let color: string = "blue";
```

2- Array Types: TypeScript allows you to represent an array in multiple ways.

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3]; // another way
```

3- Tuple: Tuples are arrays where the type of a fixed number of elements is known.

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

4- Enum: Enums allow us to define a set of named constants.

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

5- Any: We may need to describe the type of variables that we don't know when we're writing an application. These values may come from dynamic content, e.g., from the user or a 3rd party library.

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

6- Void: Void is a little like the opposite of any. It means the absence of having any type at all. Typically seen as the return type of functions that do not return a value.

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

By: Waleed Mousa

7- Null and Undefined: In TypeScript, both undefined and null actually have their own types named undefined and null respectively.

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

8- Never: The never type represents the type of values that never occur.

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

9- Object: object is a type that represents the non-primitive type.

```
declare function create(o: object | null): void;
create({ prop: 0 }); // OK
```

10- Type Assertions: A way to tell the compiler "trust me, I know what I'm doing."

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

11- TypeScript Interfaces: Interfaces define the contract in your code and provide explicit names for type checking.

```
interface LabelledValue {
  label: string;
}
function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}
```

12- Optional Properties in Interface: Not all properties of an interface may be required.

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

13- Readonly properties: Some properties should only be modifiable when an object is first created.

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

14- Function Types: Interfaces are capable of describing function types.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  return source.search(subString) > -1;
}
```

15- Indexable Types: We can describe types that we can "index into", like a[10], or ageMap["daniel"].

```
interface StringArray {
  [index: number]: string;
}
let myArray: StringArray;
myArray = ["Bob", "Fred"];
```

16- Class Types: Implementing an interface using a class.

```
interface ClockInterface {
    currentTime: Date;
}
class Clock implements ClockInterface {
    currentTime: Date = new Date();
}
```

17- Extending Interfaces: Like classes, interfaces can extend each other.

```
interface Shape {
    color: string;
}
interface Square extends Shape {
    sideLength: number;
}
let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

18- Hybrid Types: Interfaces can describe a mixture of multiple types.

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}
```

19- Interfaces Extending Classes: An interface can extend a class, which causes the interface to 'inherit' the members of the class without their implementations.

```
class Control {
    private state: any;
}
interface SelectableControl extends Control {
    select(): void;
}
```

20- Classes: Traditional OOP concepts with inheritance.

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}
class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}
```

```
const dog = new Dog();
dog.bark();
dog.move(10);
```

21- Public, private, and protected modifiers: By default, all members are public in TypeScript.

```
class Animal {
    public name: string;
    private type: string;
    protected age: number;
}
```

22- Readonly modifier: You can make properties readonly with the readonly keyword.

```
class Octopus {
    readonly name: string;
    constructor (theName: string) {
        this.name = theName;
    }
}
```

23- Static Properties: We can also create static members of a class, those that are visible on the class itself rather than on the instances.

```
class Grid {
    static origin = {x: 0, y: 0};
}
```

24- Abstract Classes: Abstract classes are base classes from which other classes may be derived.

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}
```

25- TypeScript Functions: How to create a function in TypeScript.

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

26- Function Types: We can add types to each of the parameters and then to the function itself to add a return type.

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

27- Optional and Default Parameters: TypeScript has optional parameters.

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

28- Rest Parameters: TypeScript function that gets an arbitrary number of arguments.

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

29- this parameters: TypeScript lets you ensure that the this is what you expect it to be within a function with a this parameter.

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

30- Overloads: JavaScript is inherently very dynamic, and TypeScript is designed to handle common JavaScript patterns.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // ...
}
```

31- Generics: Like Java/C#, TypeScript also has generics.

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

32- Using Type Parameters: How to use type parameters in generic functions.

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);
    return arg;
}
```

33- Generic Types: The type of generic functions.

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

34- Generic Classes: A generic class has a similar shape to a generic interface.

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

35- Generic Constraints: Sometimes we'd like to work with part of a type coming from the generics.

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

36- Using Type Parameters in Generic Constraints: Using type parameters in generic constraints.

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

37- Using Class Types in Generics: Using class types in Generics.

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

38- Optional Chaining: The optional chaining ?. stops the evaluation if the value before ?. is undefined or null and returns undefined.

```
let x = foo?.bar.baz();
```

39- **Nullish Coalescing**: The nullish coalescing operator (??) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

```
let x = foo ?? bar();
```

40- User-Defined Type Guards: Sometimes, a variable could be one of several different types. User-defined type guards are expressions that perform a runtime check that guarantees the type in some scope.

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

41- instanceof type guards: The instanceof type guards are useful when working with classes.

```
if (pet instanceof Fish) {
    pet.swim();
} else {
    pet.fly();
}
```

42- typeof type guards: typeof type guards are handy when dealing with primitive types.

```
if (typeof x === "string") {
    console.log(x.substr(1));
}
```

43- Literal Types: Literal types allow you to specify the exact value a variable must have.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

44- Discriminated Unions: A common technique for working with unions is to have a single field which uses literal types which you can use to let TypeScript narrow down the possible current type.

```
interface Bird {
    type: 'bird';
    flyingSpeed: number;
}

interface Horse {
    type: 'horse';
    runningSpeed: number;
}
```

```
type Animal = Bird | Horse;

function moveAnimal(animal: Animal) {
    let speed;
    switch (animal.type) {
        case 'bird':
            speed = animal.flyingSpeed;
            break;
        case 'horse':
            speed = animal.runningSpeed;
            break;
    }
    console.log('Moving at speed: ', speed);
}
```

45- Intersection Types: An intersection type is a way of combining multiple types into one.

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (result as any)[id] = (first as any)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (result as any)[id] = (second as any)[id];
        }
    }
    return result;
}
```

46- Mixins: TypeScript does not support multiple inheritance, but mixins can model the same pattern.

```
class Disposable {
    isDisposed: boolean;
    dispose() {
        // ...
    }
}
```

```
class Activatable {
    isActive: boolean;
    activate() {
        // ...
    }
    deactivate() {
        // ...
    }
}

class SmartObject implements Disposable, Activatable {
    // ...
}

applyMixins(SmartObject, [Disposable, Activatable]);
```

47- Conditional Types: A conditional type selects one of two possible types based on a condition expressed as a type relationship test.

```
type TypeName<T> =
    T extends string ? "string" :
    T extends number ? "number" :
    T extends boolean ? "boolean" :
    T extends undefined ? "undefined" :
    T extends Function ? "function" :
    "object";
```

48- Mapped Types: A mapped type is a generic type which uses a union created via a keyof to iterate through the keys of one type to create another.

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

49- TypeScript with JSX: TypeScript supports embedding, type checking, and compiling JavaScript directly to JavaScript.

```
const element = <h1>Hello, world!</h1>;
```

50- Module Resolution: TypeScript module resolution logic mimics the Node.js runtime resolution strategy.

```
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator = new ZipCodeValidator();
```

51- TypeScript Declaration Files: If you're using an external JavaScript library, you'll need to use a declaration file (.d.ts) to describe the shape of that library.

```
declare function require(moduleName: string): any;
```

52- Decorators: Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members.

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

53- Mixins via Decorators: TypeScript Decorators to take care of mixin pattern.

```
@DisposableMixin
@ActivatableMixin
class SmartObject {
    // ...
}
```

54- Type Compatibility: TypeScript is a structural type system which means that it relates types based solely on their members.

```
interface Named {
    name: string;
}
let x: Named;
let y = { name: "Alice", location: "Seattle" };
x = y;
```

55- Advanced Types: Intersection and Union types.

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

56- Symbols: Symbols are new primitive type introduced in ECMAScript 2015.

```
let sym1 = Symbol();
let sym2 = Symbol("key");
```

57- Iterators and Generators: Objects that have a next method which returns the result object {done, value}.

```
let someArray = [1, "string", false];
for (let entry of someArray) {
    console.log(entry);
}
```

58- Module Augmentation: Module augmentation is used to add additional members to existing modules.

```
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
```

59- Dynamic import() Expressions: Dynamic import() expressions are a new feature and part of ECMAScript that allows users to asynchronously request a module at any arbitrary point in your program.

```
async function getComponent() {
    const element = document.createElement('div');
    const { default: _ } = await import('lodash');
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
    return element;
}
```

60- TypeScript Configuration Options: tsconfig.json file to specify the root level files and the compiler options.

```
{
    "compilerOptions": {
        "module": "system",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```

61- TypeScript with Babel: TypeScript can work with Babel using the @babel/preset-typescript.

```
{
    "presets": [
        "@babel/preset-env",
        "@babel/preset-typescript"
    ]
}
```

62- TypeScript with Webpack: TypeScript can work with Webpack using ts-loader.

```
{
    test: /\.tsx?$/,
    use: 'ts-loader',
    exclude: /node_modules/,
}
```

63- TypeScript with ESLint: TypeScript can work with ESLint using @typescript-eslint/parser.

```
{
    "parser": "@typescript-eslint/parser",
    "plugins": ["@typescript-eslint"],
    "extends": [
        "eslint:recommended",
        "plugin:@typescript-eslint/eslint-recommended",
        "plugin:@typescript-eslint/recommended"
    ]
}
```

64- Using tsc command line: The tsc command line utility to compile TypeScript code.

```
tsc index.ts
```

65- TypeScript Compiler API: TypeScript compiler API allows you to manipulate and navigate the abstract syntax tree.

```
import * as ts from "typescript";

function delint(sourceFile: ts.SourceFile) {
    delintNode(sourceFile);

    function delintNode(node: ts.Node) {
        // ...
        ts.forEachChild(node, delintNode);
    }
}

let program = ts.createProgram(["./file.ts"], { allowJs: true });
let sourceFile = program.getSourceFile("./file.ts");
delint(sourceFile);
```

66- Tagged Template Strings: A more advanced form of template strings are tagged template strings.

```
let str = tag`Hello ${ a + b } world ${ a * b }`;
```

67- Type Inference: TypeScript tries to infer the types when there are no explicit types specified.

```
let x = 3;  // `x` has the type of `number`
```

68- Type Assertions: Sometimes you will know more about a value than TypeScript does and you would need to assert the type.

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

69- Compile-Time vs Runtime Typing: TypeScript types do not exist at runtime, they are only used by the compiler for type checking.

```
let x: number = 1;
console.log(typeof x);  // 'number', not 'Number'
```

70- Type Predicates: A special kind of type that can be returned from a function's signature.

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

71- Optional Properties in Classes: You can use the optional modifier ? to indicate properties that might not exist.

```
class User {
    username: string;
    password?: string;
}
```

72- TypeScript with React: How to use TypeScript with React.

```
interface AppProps {
    title: string;
}

class App extends React.Component<AppProps, {}> {
    render() {
        return <h1>{this.props.title}</h1>;
    }
}
```

73- Enums: Enums are a feature added to JavaScript by TypeScript which makes it easier to handle named sets of constants.

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

74- Never Type: The never type represents the type of values that never occur.

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

75- Namespace: Namespaces are used to organize code and avoid naming collisions.

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

76- Module: Modules are executed within their own scope, not in the global scope.

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

77- Ambient Declarations: If you want to use JavaScript libraries not written in TypeScript, we need to declare the types of variables that we will use.

```
declare var jQuery: (selector: string) => any;
```

78- TypeScript with Express: How to use TypeScript with Express.

```
import * as express from "express";

const app: express.Application = express();

app.get('/', function (req, res) {
    res.send('Hello World!');
});
```

79- Triple-Slash Directives: Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

```
/// <reference types="node" />
```

80- Decorators and Metadata Reflection: Decorators to annotate and modify classes and properties at design time.

```
@Reflect.metadata(metadataKey, metadataValue)
class MyClass {
}
```

81- BigInt: BigInt is a built-in object that provides a way to represent whole numbers larger than 2^53.

```
let foo: bigint = BigInt(100); // the BigInt function
```

82- Utility Types: Utility types provide a set of type transformations used to manipulate types.

```
type T1 = Partial<User>;  // Make all properties in User optional
type T2 = Readonly<User>;  // Make all properties in User read-only
```

83- Mapped Types: Mapped types allow you to create new types based on old types.

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

84- Declaration Merging: Declaration merging is the process of joining two separate declarations declared with the same name into a single definition.

```
interface Cloner {
    clone(animal: Animal): Animal;
}
interface Cloner {
    clone(animal: Sheep): Sheep;
}
```

85- Using `tsc --init` to generate a tsconfig.json file: `tsc --init` creates a new tsconfig.json file in the current directory with default options.

```
tsc --init
```

86- TypeScript with Vue.js: How to use TypeScript with Vue.js.

```
import Vue from 'vue'

const Component = Vue.extend({
  // type inference enabled
})
```

88- TypeScript with Angular: How to use TypeScript with Angular.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<h1>Hello {{name}}</h1>',
})
export class AppComponent  { name = 'Angular'; }
```

89- Importing other files: TypeScript allows you to import other files using the import keyword.

```
import { MyClass } from './my-class';
```

90- Exporting a declaration: Any declaration can be exported by using the export keyword.

```
export interface MyInterface { /* ... */ }
export class MyClass { /* ... */ }
```

91- Ambient Modules: In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own .d.ts file with top-level export declarations, but it's more convenient to write them as one larger .d.ts file.

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?,
slashesDenoteHost?): Url;
}
```

92- Working with Type Guards: A type guard is some expression that performs a runtime check that guarantees the type in some scope.

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

93- Optional Parameters in Callbacks: When writing functions that will be used by others, it's a good idea to make sure the functions can be called with fewer parameters than declared.

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void)
{
    for(let i = 0; i < arr.length; i++) {
        callback(arr[i], i);
    }
}
```

94- Rest Parameters and Spread Syntax: TypeScript supports rest parameters and spread syntax, both of which can be used with tuples.

```
function f(...args: [number, string, boolean]) {
    let [a, b, c] = args;
}
```

95- Type Aliases: Type aliases create a new name for a type.

```
type MyBool = true | false;
```

96- String Literal Types: String literal types allow you to specify the exact value a string must have.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

97- Numeric Literal Types: TypeScript also has numeric literal types.

```
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;
```

98- Readonly Array and Tuple Types: TypeScript provides readonly versions of array and tuple types.

```
let a: readonly number[] = [1, 2, 3, 4];
let b: readonly [number, string] = [1, "hello"];
```

99- Global Augmentation: TypeScript allows declaration merging to build up definitions through extension.

```
declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}
```

100- Triple Equals vs Double Equals: `===` is the equality operator, and `==` is the loose equality operator.

```
let x: any = 0;
console.log(x == false);  // True, loose equality
console.log(x === false);  // False, strict equality
```

101- Project References: Project references are a new feature in TypeScript 3.0 that allow projects to depend on other projects.

```
{
    "compilerOptions": {
        "outDir": "../out",
        "rootDir": "../",
        "composite": true
    },
    "references": [
        { "path": "../core" },
        { "path": "../types" }
    ]
}
```

102- TypeScript Versioning: TypeScript follows semantic versioning.

```
npm install typescript@3.1
```