Advanced CONCEPTS TypeScript

Implements keyword

```ts
type Animal = {
  name: string;
  voice(): string;
};

function log(animal: Animal) {
  console.log(`Animal: ${animal.name}: ${animal.voice()}`);
}

class Cat implements Animal {
  constructor(public name: string) {}
  voice() {
    return "meow";
  }
}

class Dog implements Animal {
  constructor(public name: string) {}
  voice() {
    return "woof";
  }
}

log(new Cat("Salem"));
log(new Dog("Lassie"));
```

Use define Type Guards

```ts
type Square = {
  size: number;
};

type Rectangle = {
  width: number;
  height: number;
};

type Shape = Square | Rectangle;
// use define type guards (shape is Square)
function isSquare(shape: Shape): shape is Square {
  return "size" in shape;
}

function isRectangle(shape: Shape): shape is Rectangle {
```

```
  return "width" in shape;
}

function area(shape: Shape) {
  if (isSquare(shape)) {
    return shape.size * shape.size;
  }

  if (isRectangle(shape)) {
    return shape.width * shape.height;
  }

  const _ensure: never = shape;
  return _ensure;
}
```

Assertion type

1. if one parameter in function have true state this function will return

```
type Person = {
  name: string;
  dateOfBirth?: Date;
};

//* assertion function
function assert(condition: unknown, message: string): asserts condition {
    //! only return value if the condition asserts true
    if (!condition) {
        trow new Error(message);
    }
}

//* assertion function
                                  // asserts parameterName is type
function assertDate(value: unknown): asserts value is Data {
    if (value instanceof Date) return
    else throw new Error("value is not a Data")
}

const maybePerson = loadPerson();

assert(maybePerson != null, "Could not load person");
console.log("Name:", maybePerson.nama);

assertDate(maybePerson.dateOfBirth);

console.log("Date of Birth:", maybePerson.dateOfBirth.toISOString());
```

Function Overloading

1. this approach used only TS

2. now we now what tape function returns 2.1 First example

```
function reverse(string: string): string; // set type return string
function reverse(stringArray: string[]): string[]; // set type return array
function reverse(stringOrArray: string | string[]) {
    if (typeOf stringOrArray === 'string') {
        return stringOrArray.split("").reverse().join("");
    } else {
        return stringOrArray.slice().reverse();
    }
}

const hello = reverse("hello"): // set type return string
const h_e_l_l_0 = reverse("h_e_l_l_0"): // set type return array
```

2.2 Second example:

```
function makeDate(timesStamp: number): Date;
function makeDate(year: number, month: number, day: number): Date;
function makeDate(timesStampOrYear: number, month?: number, day?: number):
Date{
    if (month !null && day !null) {
        return new Date(timesStampOrYear, month-1, day);
    }else {
        return new Date(timesStampOrYear);
    }
}

const doomsDay = makeDate(2000, 1,1); // 1 Jan 2000;
const epoch = makeDate(0); // 1 Jun 1970;

<!-- const Invalid = makeDate(2000,1 /* Error: ignored */) -->
```

Call Signatures

1. Work with functions:

```
type Add = (a: number, b: number) => number;

interface Add3 {
  (a: number, b: number): number;
}

type Add2 = {
  (a: number, b: number): number;
```

```
  (a: number, b: number, c: number): number;
  debugName?: string;
};

add.debugName = "Addition Function";

const add: Add2 = (a: number, b: number, c?: number) => {
  return a + b + (c != null ? c : 0);
};
```

2. Work with classes

```
type PointCreator = new (x: number, y: number) => { x: number; y: number };

type PointCreator1 = {
  new (x: number, y: number): { x: number; y: number };
};

const Point: PointCreator1 = class {
  constructor(public x: number, public y: number) {}
};
```

3. Work with classes and functions

```
type Add2 = {
  new (x: number, y: number): { x: number; y: number };
  new (x: number, y: number, z: number): { x: number; y: number; z: number
};
  (x: number, y: number): { x: number; y: number };
  (x: number, b: number, z: number): { x: number; y: number; z: number };
  debugName: string;
};
```

Abstract classes

1. Use only when need creates prepare methods to real class
2. Can't create install of abstract classes

```
abstract class Command {
  abstract commandLine(): string;

  execute() {
    console.log("Executing:", this.commandLine());
  }
}

class GitResetCommand extends Command {
```

```
  commandLine() {
    return "git reset --hard";
  }
}

class GitFetchCommand extends Command {
  commandLine() {
    return "git fetch --all";
  }
}

new GitResetCommand().execute();
new GitFetchCommand().execute();

new Command(); // Error: cannot create an instance of an abstract class
```

Index Signatures

```
type Dictionary = {
  [key: string]: boolean;
};
```

```
type Person = {
  displayName: string;
  email: string;
};

type PersonDictionary = {
  [userName: string]: Person | undefined;
};

const persons: PersonDictionary = {
  john: { displayName: "John", email: "jon@gmail.com" },
};

person["john"] = { displayName: "John", email: "jon@gmail.com" };

console.log(persons["john"]);

delete persons["john"];
const result = persons["missing"];
console.log(result, result.email); // undefined
```

Readonly array and tuples

1. If need block modified array

```
function reverseSorted(input: readonly number[]): number[] {
  return input
    .slice() // use this method because we don't want to modify the array
passed
    .sort() // modifier current array
    .reverse(); // modifier current array
}

const start = [1, 2, 3, 4, 5, 6, 7, 8, 9];
const result = reverseSorted(start);

console.log(result); // [9,8,7,6,...]
console.log(start); // [1,2,3,4,5,6,7,8,9]
```

```
type Neat = readonly number[];
type Long = ReadonlyArray<number>;
```

2. Readonly Tuples array

```
type Point = readonly [number, number];

function move(point: Point, x: number, y: number): Point {
  return [point[0] + x, point[1] + y];
}

const point: Point = [0, 0];
const moved = move(point, 10, 10);

console.log(moved); // [10, 10],
console.log(point); // [10, 10],
```

Double Assertion

```
type Point2D = { x: number; y: number };
type Point3D = { x: number; y: number; z: number };
type Person = { name: string; age: number };

let point2: Point2D = { x: 10, y: 10 };
let point3: Point3D = { x: 10, y: 10, z: 10 };
let person = (Person = { name: "Jon", age: 12 });

point2 = point3;
point3 = point2; //!ERROR

point3 = point2 as Point3D; // Ok: I trust you
```

```
person = point3; //!ERROR
point3 = person; //!ERROR

point3 = person as Point3D; // Error: I don't trust you enough
point3 = person as unknown as Point3D; // Ok: I Double trust you
```

const Assertion

1. If need block mutable object use =>>> as const

```
const king = "elvis";
king = "john"; //!ERROR
const upperCased = king.toUpperCase(); // king === "elvis;

const dave = {
    nama: "dave",
    role: "drummer",
    skills: ['drumming', 'headbanging],
} as const

dave = { //!ERROR as const!!
    nama: "geol",
    role: "singer",
    skills: ['singer', 'drumming']
};

dave.name = "max" //!ERROR as const
```

```
function layout(settings: {
  align: "center" | "left" | "right";
  number: number;
}) {
  console.log("Performing layout", settings);
}

const example = {
  align: "center" as const,
  padding: 0,
} as const; // not needed if align as const

layout(example); //ERROR type of property "center" are incompatible.
```

this parameter

```
function double(this: { value: number }) {
  // only for TS ==>> this: {value: number}
```

```
  this.value = this.value * 2;
}

const valid = {
  value: 10,
  double,
};

valid.double();
console.log(valid.value); // 20

const invalid = {
  valuessss: 10,
  double,
};

invalid.double(); //!ERROR
```

Generic Constrains

```
type NameFields = { firstName: string; lastName: string };

function addFullName<T extends NameFields>(obj: T): T & { fullName: string } {
  return {
    ...obj,
    fullName: `${obj.firstName}${obj.lastName}`,
  };

  const john = addFullName({
    email: "john@example.com",
    firstName: "John",
    lastName: "Doe",
  });

  console.log(john.email); // john@example.com
  console.log(john.fullName); // Jon Doe

  const jane = addFullName({ firstName: "Jane", lastName: "Austen" });in
```

Dealing with Temporal Uncertainty

1. Need use local variables if TS will not now what variable will be.

```
let suffix: string | null = getSuffix();

if (suffix != null) {
    const suffixLocal = suffix; // this local variable
    let exampleOne: string = "jane" + suffixLocal.toUpperCase();
    ["jane", "jon"].forEach(name => {
```

```
        let exampleTwo: string =name + suffixLocal.toUpperCase();
    });
}

let example: string | null = forExample();

let (example != null) {
    const exampleLocal = example; // this local variable
    setTimeout(() => {
        console.log(exampleLocal.toUpperCase());
    });
}

example = null;
```