Super CONCEPTS TypeScript

Partial

```
type Point = { x: number; y: number };

// Same as `{x?: number, y?: number}`

class State<T> {
  constructor(public current: T) {}
  update(nex: Partial<T>) {
    this.current = { ...this.current, ...next };
  }
}

// Usage
const state = new State({ x: 0, y: 0 });

state.update({ y: 123 }); // Partial. No need to provide `x`.

console.log(state.current); // Update successfully.: {x:0, y:123}
```

Required

```
/**
 * Make all properties available in T Required!
 */
export type Required<T> = {
  //! THIS FUNCTIONALITY IMPLEMENTED in TS ==> Required<>
  [P in keyof T]-?: T[P];
};

type PartialPoint = { x?: number; y?: number };
// Same as `{x:number, y:number}`
type Point = Required<PartialPoint>;
```

```
type CircleConfig = {
  color?: string;
  radius?: number;
};

class Circle {
  // Required: Internally all members well always be present
  private config: Required<CircleConfig>;
  constructor(config: CircleConfig) {
```

```
    this.config = {
      color: config.color ?? "green",
      radius: config.radius ?? 0,
    };
  }

  draw() {
    // No null checking needed!
    console.log(
      "Drawing Circle",
      "Color: " + config.color,
      "Radius: " + config.radius
    );
  }
}
```

Readonly

```
/**
 * Make a all properties in T readonly
 */
export type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

type Point = { x: number; y: number };

// Same as `{ readonly x: number, readonly y: number}`
type ReadonlyPoint = Readonly<Point>;
```

```
function makeReadonly<T>(object: T): Readonly<T> {
  return Object.freeze({ ...object });
}

const editablePoint = { x: 0, y: 0 };
editablePoint.x = 2; // successfully allowed

const readonlyPoint = makeReadonly(editablePoint);
readonly.x = 3; // Error: readonly
```

Record <K,V> for Object

```
type Persons = Record<string, { name: string; role: string }>;
const persons: Persons = {};
persons["000"] = { name: "jon", role: "admin" };
persons["111"] = { name: "jane", role: "owner" };
```

```
persons["222"] = { name: "june" }; // ERROR: Missing property `role`

// We can achieve exactly same behavior use:
type PersonVerbose = { [key: string]: { name: string; role: string } };
```

```
type Roles = "admin" | "owner";

let peopleWithRoles: Record<Roles, string[]> = {
  owner: ["jane", "june"],
  admin: ["jane"],
};

peopleWithRoles = {
  owner: ["jane", "June"],
}; // Error: 'Admin' is Missing

const admins: string[] = peopleWithRoles["admin"]; //Safe
```

```
type PageInfo = {
  id: string;
  title: string;
};

type PageVerbose = {
  home: PageInfo;
  services: PageInfo;
  about: PageInfo;
};
/// ========  OR ======== ///

type Pages = Record<
  "home" | "services" | "about" | "contact",
  { id: string; title: string }
>;
```

AutoComplete Literals Unions with Primitives

1. If need literal string autocomplete

```
type Padding = "small" | "medium" | "large" | (string & {});

function getPadding(padding: Padding): string {
  if (padding === "small") return "12px";
  if (padding === "medium") return "16px";
  if (padding === "large") return "24px";
  return padding;
```

```
}

let padding: Padding;
(padding = "small"),
  (padding = "8px"), //Not Error because use ===> (string & {});
  (padding = "large");
```

undefined vs optional

```
type ExampleOptional = {
  name?: string;
};

let optional: ExampleOptional;

optional = { name: undefined };

optional = {};

type ExampleUnion = {
  name: string | undefined;
};

let union: ExampleUnion;

union = { name: undefined };
union = {}; //! ERROR: NAME is missing
```

```
function logOptional(message?: string) {
  console.log(message);
}
function logUnion(message: string | undefined) {
  console.log(massage);
}

logOptional(undefined);
logOptional();

logUnion(undefined);
logUnion(); //! ERROR: Expected 1 argument. `message` was not provided.
```

```
function logOptional(error?: Error, message: string) {
  if (error != undefined) {
    console.log(error, message); //! ERROR: Expected 1 argument. `message` was not
provided
  } else {
    console.log(message);
```

```
  }
}

function logUnion(error: Error | undefined, message: string) {
  if (error != undefined) {
    console.log(error, message);
  } else {
    console.log(message);
  }
}
```

satisfies operator

```
type Color = ColorString | ColorRGB;
type ColorString = "red" | "green" | "blue";
type ColorRGB = [red: number, green: number, blue: number];

type Theme = Record<string, Color>;

const theme = {
  primary: "red",
  secondary: [0, 255, 0],
  tertiary: "ekiw", // <== ERROR because use satisfies Theme
} satisfies Theme;

const [r, g, b] = theme.secondary; //Not error because use satisfies Theme
```

PropertyKey

```
const str: string = "key";
const num: number = 1;
const sym: symbol = Symbol();
const valid = {
  [str]: "valid",
  [nym]: "valid",
  [symm]: "valid",
};
const obj = {};
const invalid = {
  [obj]: "invalid",
};
let example: PropertyKey; //===>>>>> string | number | symbol
example = str;
example = num;
example = sym;
example = obj; // !Error ===key value only  string | number | symbol
```

ThisTypeUtility

```
type Math = {
    double(): void,
    half(): void,
}

export const math: Math&ThisType<{value: number}> = {
    double()// (this: {value: number}){ <<<== not use because we use
=>>ThisType<{value: number}>
        this.value *= 2;
    }
    half()// (this: {value: number}){  <<<== not use because we use
=>>ThisType<{value: number}>
        this.value /= 2;
    }
}

const object = {
    value: 1,
    ...math,
}

obj.double();
console.log(obj.value);//2

obj.half();
console.log(obj.value);//1
```

AwaitedUtility

1. Basic example use Type Promise

```
main();
async single: Promise<string> = new Promise(res => res("Hello, world"));

const triple: Promise<Promise<Promise<string>>> =
    new Promise<Promise<Promise<string>>>(res =>
        res(
            new Promise<Promise<string>>((res) => {
                res (
                    new Promise<string>((res =>{
                        res("Vin Diesel")
                    }
                )
            })
        )
    )

const singleResult = await single;
```

```
console.log(singleResult); // "Hello, world"

const tripleResult = await triple;
console.log(tripleResult); // "Vin Diesel"
```

2. Example when we use Awaited

```
type WrappedInDeep = Promise<Promise<Promise<string>>>;

// AwaitedResult type `string`
type AwaitedResult = Awaited<WrappedInDeep>;
```

```
async function example<T>(input: T) {
  const output: Awaited<T> = await input;
}
```

String Manipulation

1. Basic example

```
type abba = Uppercase<"abba">;

type Loud = "HELLO WORLD";

type Quiet = Lowercase<Loud>;

type Hello = "fef fi fo fum";
type Better = Capitalize<Hello>; // ==> Fef fi fo fum

type UncomfortableGreeting = Uncapitalized<Loud>; // hELLO WORLD

type Scream = Uppercase<"Hello!">; //HELLO!
```

2. Real example

```
type Getter<T extends string> = `get${Capitalize<T>}`;
type Setter<T extends string> = `set${Capitalize<T>}`;

type Name = "name";

type GetName = Getter<Name>; // getName
type SetName = Setter<Name>; // getName
```

Mapped types

```ts
type State = {
  name: string;
  age: number;
};

/**
 * {
 *  setName: (value: string) => void;
 *  setAge: (value: number) => void;
 * }
 */

type Setters = {
  [k in keyof State as `set${Capitalized<K>}`]: (value: State[K]) => void;
};
```

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