Expert CONCEPTS TypeScript

typeof type operator

1. Created type use dynamically schema

```ts
const center = {
  x: 0,
  y: 0,
  z: 0,
};

type Point = typeof center;

const unit: typeof center = {
  // :typeof center = {  Shorthand annotation inline
  x: center.x + 1,
  y: center.y + 1,
  z: center.z + 1,
};
```

2. Use in function

```ts
const personResponse = {
  name: "John",
  email: "john@example.com",
  firstName: "John",
  lastName: "John",
};

type PersonResponse = typeof personResponse;

function processResponse(person: PersonResponse) {
  console.log("Full name: " + person.firstName + " " + person.lastName);
}
```

Lookup types

1. For example we working with Response data and need get dynamic types

```ts
export type SubmitRequest = {
  transactionId: string;
  personal: {
    title: string;
    email: string;
```

```
    previousAliases: {
      firstName: string;
      middleName: string;
      lastName: string;
    }[];
  };

  payment: {
    creditCard: string;
  };
};

type PaymentRequest = SubmitRequest["payment"]; //<<<>>> Lookup types
type PreviousAliasesRequest = SubmitRequest["personal"]["previousAliases"][0];
//<<<>>> Lookup types

export function getPayment(): PaymentRequest {
  return {
    creditCard: "adasa123assfafaf",
  };
}
```

keyof type operator

1. get all keys in Object

```
type Person = {
  name: string;
  age: number;
  location: string;
};

const john: Person = {
  nama: "John",
  age: 36,
  location: "Melbourne",
};

function logGet(obj: any, key: keyof Person) {
  // keyof Person =>> name | age | location
  const value = obj[key];
  console.log("Getting:", key, value);
  return value;
}
//========= OR ==========
function logGet1<Obj, Key extends keyof Obj>(obj: Obj, key: Key) {
  // <Obj, Key extends keyof Obj>
  // 1. Generic type Obj and Generic type Key
  // 2. Key will be something that is in the Key of Obj
  // 3. function will be return Obj[Key]
```

```
  const value = obj[key];
  console.log("Getting:", key, value);
  return value;
}

const age = logGet(jon, "age"); // 36
console.log(logGet(jon, "email")); //! Error
//              Generic, Generic Constrains
function logSet<Obj, Key extends keyof Obj>(
  obj: Obj,
  key: Key,
  value: Obj[key] // Obj[key] -> Lookup type
) {
  console.log("Setting:", key, value);
  obj[key] = value;
}

logSet(jon, "age", 36);
```

Conditional types

```
type IsNumber<T>=
    ?"number"
    : "other"

type WithNumber = IsNumber<number>
type WithOther = IsNumber<string>

const IsNumber = (value: unknown) =>
    typeOf value === "number"
    ? "number"
    : "other"

const withNumber = isNumber(123);
const withOther = isNumber("hello");
```

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

    function TypeName<T>(t:T): TypeName<T> {
        if (t === null) return 'null' as TypeName<T>;
```

```
        return typeOf t as TypeName<T>;
    }

    const str = TypeName("h_e_l_l");
    const num = TypeName(123)
    const bool = TypeName(undefined)
    const undefined = TypeName(undefined)
    const sym = TypeName(Symbol("star"))
    const big = TypeName(24n);
    const func = typeName(function(){})
    const obj = typeName(null);

    console.log(typeof null) // object;
```

ReturnType (Function)

1. If need created type form function return data

```
export function createPerson(firstName: string, lastName: string) {
  return {
    firstName,
    lastName,
  };
}

function logPerson(person: ReturnType<typeof createPerson>) {
  console.log("Person: " + person.firstName + " " + person.lastName);
}
```

Mapped types

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

//! Impotent this feature implemented in TypeScript
// type ReadonlyPoint<T> = {
//   // <<<<<<<Mapped types>>>>>>>
//   // 1. this using loop
//   // 2. keyof T => get all keys from
//   // 3. Item => this is a variable

//   readonly [Item in keyof T]: T[Item];
// };
//                  [Item in keyof T]: T[Item]
const center: ReadonlyPoint<Point> = {
  x: 0,
```

```
  y: 0,
  z: 0,
};
```

```
export type Point = {
  readonly x: number;
  y?: number;
};

export type Mapped<T> = {
  // 1. use (-readonly)  for removed readonly type
  // 2. use (-?) for removed question type
  -readonly [P in keyof T]-?: T[P];
};
```

Template Literal Types

1. Use Template `My name: ${string}`

```
type CSSValue =
  // implies px
  | number
  // number + px|em|rem
  | `${number}px`
  | `${number}em`
  | `${number}rem`;

function size(input: CSSValue) {
  return typeof input === "number" ? input + "px" : input;
}

size(123);
size("123px");
size("123em");
size("123ex"); // ERROR
```

```
type Size = "small" | "medium" | "large";
type Color = "primary" | "secondary";
type Style = `${Size}-${Color}`;

/**
 * @param style is a combination of
 * Size: "small" | "medium" | "large
 * Color: "primary" | "secondary
 * e.g "small-secondary"
 */
function applyStyle(style: string) {
```

```
  // ...
}

applyStyle("small-primary");
applyStyle("large-secondary");
applyStyle("asdsad-secondary"); //ERROR!!!!
```