# UTILITY TYPES
# TYPESCRIPT

# THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

# UTILITY TYPES

In TypeScript, utility types are predefined types that help with common transformations of types.

## ☆ Partial

For example, the following thing transforms the type to be optional.

```
2   interface Person {
3     name: string;
4     age: number;
5   }
6
7   const partialAri: Partial<Person> = {
8       name: "Ari"
9   }
```

We learned of Partial Utility Type which makes the type transformed into the one with optional properties.

Yes! Optional Properties!

```
2   interface Person {
3     name: string;
4     age: number;
5   }
6
7   const partialAri: Partial<Person> = {
8   }
```

## ☆ Required

It makes all the properties required. But by default, a type will enforce being required.

```
2   interface Person {
3     name: string;
4     age: number;
5   }
6
7   const partialAri: Person = {
8       name:"Ari"
9   }
```

We can transform a type with optional to be fully required.

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

const partialAri: Required<Person> = {
    name:"Ari"
}
// Property 'age' is missing in type
```

## ☆ Readonly

If the properties have to be read only, this is how...

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

const ari: Readonly<Person> = {
    name:"Ari",
    age: 22
}

ari.name = "Ari";
//Cannot assign to 'name' because it is a read-only property.
```

## ☆ Record

Record constructs an object type.

```
const scores: Record<string, number> = {
    Ari: 22,
    Arii: 21,
    Ariii: "Twenty Three"
};
//Type 'string' is not assignable to type 'number'
```

## ☆ Pick

Pick constructs an object type with properties with type picked out.

```
interface Animal {
    name: string,
    age: number,
    weight: number,
    scname: string
}

const nameAndAge : Pick<Animal, "name" | "age"> = {
    name: "Elephant",
    age: 60
}
```

## ☆ Omit

Omit constructs an object type with certain properties omitted.

```
interface Animal {
    name: string,
    age: number,
    weight: number,
    scname: string
}

const nameAndAge : Omit<Animal, "name" | "age"> = {
    weight: 4000,
    scname: "Elephas maximus",
}
```

## ☆ Exclude

Exclude constructs a type by excluding from type T all properties that are assignable to type U.

```
type myType = string | number;

let yourType: Exclude<myType, string> = 23;
yourType = "Ari";
// Type 'string' is not assignable to type 'number'.
```

## ☆ Extract

Extract constructs a type by extracting from type T all properties that are assignable to type U.

```
type myType = string | number;

let yourType: Extract<myType, string> = "Ari";
yourType = 147;
// Type 'number' is not assignable to type 'string'
```

## ☆ NonNullable

NonNullable constructs a type by removing null and undefined from type T.

```
type myType = string | number | undefined | null;

let yourType: NonNullable<myType> = "Ari";
yourType = null;
// Type 'null' is not assignable to type 'NonNullable<myType>'
```

## ☆ Parameters

Parameters extracts the parameter types of a function type as a tuple.

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

let paramType: Parameters<MyFuncType> = [2, "ari"];
paramType = [2, 34];
// Type 'number' is not assignable to type 'string'
```

## ☆ ReturnType

ReturnType extracts the return type of a function type.

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

let returnType: ReturnType<MyFuncType> = 23;
returnType = "Ari";
// Type 'string' is not assignable to type 'number'
```

## ☆ InstanceType

InstanceType extracts the type of the class.

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

type PersonInstance = InstanceType<typeof Person>;
const Ari: PersonInstance = new Person("Ari", 22);
const AriS: PersonInstance = new Person("Ari", "TwentyTwo");
// Argument of type 'string' is not assignable to parameter of type 'number'
```

## ☆ ConstructorParameters

Just like Parameters, ConstructorParameters does extracting the parameter types of a constructor function as a tuple type.

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

let Ari: ConstructorParameters<typeof Person> = ["ari", 22]; // [string, number]
Ari = [22, 23];
// Type 'number' is not assignable to type 'string'
```

## ☆ Iterable

Iterable is an interface representing any object that can be iterated over using a for...of loop.

```
function printNames(iterable: Iterable<string>) {
  for (const element of iterable) {
    console.log(element);
  }
}

const array = ["ari", "haran", "sudhan"];
printNames(array);

const arrayTwo = ["ari", "haran", 22];
printNames(arrayTwo);
```

## ☆ Iterator

It represents an object that can be manually iterated over with the .next() method.

```
const array = [1, 2, 3];
const iterator: Iterator<number> = array[Symbol.iterator]();

console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
console.log(iterator.next().done);  // true
```

## ☆ IterableIterator

When an object is both iterator and iterable, we can apt IterableIterator.

```
function* generator(): IterableIterator<number> {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generator();
console.log(gen.next().value); // 1
for (const num of gen) {
  console.log(num); // 2, 3
}
```

## ☆ Awaited

Awaited extracts the resolved type of a Promise.

```
type Result = Awaited<Promise<string>>;
async function getData(): Promise<string> { return "hello"; }
const data: Result = await getData();
```

## ☆ ReadOnlyArray

It makes the array read only.

```
4   const readonlyNumbers: ReadonlyArray<number> = [1, 2, 3];
5   readonlyNumbers[0] = 10;
6   // Index signature in type 'readonly number[]' only permits reading
```

We also have stuffs like ReadOnlyMap, ReadOnlySet.

## ☆ UpperCase

Since we have the concept of Literal Types, we do have funniest case changing Utility Types.

```
2   type Lower = 'ari';
3   type Upper = Uppercase<Lower>;
4   let myName: Upper = "ARI";
5   myName = "ARII"; // Type '"ARII"' is not assignable to type '"ARI"'.
6
```

We also have LowerCase, Capitalize, Uncapitalize just for your love!

Merci : ThankYou <Nandri>