Single type

```
type Obj = {
  0: "a";
  1: "b";
  propC: "c";
  propD: "d";
};
type Result0 = Obj["propC"]; // "c"
type Result1 = Obj[0 | 1]; // "a" | "b"
type Result2 = Obj["propC" | "propD"]; // "c" | "c"
```

typeof

```
const st = "Hello";

type StType = typeof st; // string
```

keyof

```
type Object = {
  first: 1;
  second: 2;
};

type Result = keyof Object; //"first" | "second"
```

Getting values from object

```
type Obj = {
  a: "A";
  b: "B";
  c: number;
};
type Values = Obj[keyof Obj]; // number | "A" | "B"
```

index signature

```
// type key is only string | number | symbol
type Card = {
  [key: string]: number;
};

// or using Record helper
type Card = Record<string, number>;

const inCard: Card = {
  first: 20,
  second: 121,
};
```

String manipulation

```
type Res0 = Uppercase<"test">;
type Res1 = Lowercase<"TEST">;
type Res2 = Capitalize<"test">;
type Res3 = Uncapitalize<"Test">;
```

Extract

```
type Event =
  | {
      type: "click";
      event: MouseClick;
    }
  | {
      type: "mouseover";
      event: MouseOver;
    };

type MyEvent = Extract<Event, { type: "click" }>;
/*
type MyEvent = {type: "click", event: MouseClick}
*/
```

Exclude

```
type Event =
  | {
      type: "click";
      event: MouseClick;
    }
  | {
      type: "mouseover";
      event: MouseOver;
    } | {
      type "mouseout";
      event: MouseOut
    };

type MyEvent = Exclude<Event, { type: "click" }>;
/*
type MyEvent = {
  type: "mouseover";
  event: MouseOver;
} | {
  type "mouseout";
  event: MouseOut
}
*/
```

extends

```
/* Если не ограничить generic type с помощью extends,
то считается что это тип undefined
 */
T extends {} // оначает все кроме null | undeinfed

// Это потому что в JS все кроме  null, undefined является объектом
```

Pick

```
type User = {
  id: string;
  name: string;
  email: string;
};

type NameAndEmail = Pick<User, "name" | "email">;
/*
type NameAndEmail = {
  name: string;
  email: string;
}
*/
```

Omit

```
type User = {
  id: string;
  name: string;
  email: string;
};

type NameAndEmail = Omit<User, "id">;
/*
type NameAndEmail = {
name: string;
email: string;
}
*

```

Partial

```
type User = {
  id: string;
  name: string;
};

type NewUser = Partial<User>;
/*
type NewUser = {
  id?: string;
  name?: string;
}
 */
```

Readonly

```
type User = {
  id: string;
  name: string;
};

type NewUser = Readonly<User>;
/*
type NewUser = {
  readonly id: string;
  readonly name: string;
}
```

Parameters

```
// Parameters extract params from function type and return tuple
const myFunc = (url: string, opts: { id: number; name: string }) => {};

type MyFn = Parameters<typeof myFunc>;

/*
[url: string, opts: {
    id: number;
    name: string;
}] */
```

ReturnType

```
const myFunc = () => {
  return {
    id: 22,
    name: "Bill",
  };
};

type MyFunc = ReturnType<typeof myFunc>;
/*
type MyFunc = {
    id: number;
    name: string;
} */
```