Операторы

- `&&=`

```ts
let a: number | undefined = 5;
let b: number | undefined;

// присваиваем значение, только если текущее значение является истинным
a &&= 10;
b &&= 10;

console.log(a); // 10
console.log(b); // undefined
```

- `||=`

```ts
let str: string | undefined;

// присваиваем значение, только если текущее значение является ложным
str ||= "default value";

console.log(str); // выведет "default value"

// str уже имеет значение, поэтому оператор ||= не будет его изменять
str ||= "new default value";

console.log(str); // выведет "default value"
```

Операторы

- `??=`

```
let str: string | undefined;

// присваиваем значение,
// только если текущим значением является `null` или `undefined`
str ??= "default value";

console.log(str); // выводит "default value"
```

- `??`

```
// определяем переменную, которая может быть null или undefined
let str: string | null | undefined;

// используем оператор ?? для установки значения по умолчанию
const result = str ?? "значение по умолчанию";

// выведет "значение по умолчанию", так как myVariable равна null
console.log(result);
```

Объектные типы

- *Объект с произвольным количеством свойств*

```
{ [key: string]: Type }
{ [key: number]: Type }
{ [key: symbol]: Type }
{ [key: `data-${string}`]: Type }
```

- или

```
type NumericObject = Record<string, number>;

const myObj: NumericObject = {
  a: 1,
  b: 2,
  c: 3,
  // ...
};
```

Литеральные типы

- Строковый

```
let direction: 'left' | 'right'
```

- Числовой

```
let roll: 1 | 2 | 3 | 4 | 5 | 6
```

- Определение переменной с литеральным типом:

```
let name: "John";
name = "John"; // OK
name = "Mary"; // Ошибка: Type '"Mary"' is not assignable to type '"John"'
```

- Определение параметра функции с литеральным типом:

```
function printColor(color: "red" | "green" | "blue") {
  console.log(`The color is ${color}`);
}

printColor("red"); // OK
printColor("green"); // OK
 // Ошибка: Type '"yellow"' is not assignable to type '"red" | "green" | "blue"
printColor("yellow");
```

Массивы и кортежи

- Массив строк

```
string[]
// или
Array<string>
```

- Массив функций, возвращающих строки

```
(() => string)[]
// или
{ (): string }[]
// или
Array<() => string>
```

- Кортеж

```
let myTuple: [string, number, boolean?]

myTuple = ['test', 42]
```

Объединение и пересечения

- **Объединение**

```
// Объединение типов string и number
let value: string | number;

value = "hello"; // корректно
value = 123; // корректно
value = true; // ошибка, значение типа boolean не является string или number
```

- **Пересечение**

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

type Employee = {
  companyId: string;
  jobTitle: string;
};

type PersonAndEmployee = Person & Employee;

const personAndEmployee: PersonAndEmployee = {
  name: 'John Doe',
  age: 30,
  companyId: '1234',
  jobTitle: 'Developer'
};
```

Дженерики

- Пример интерфейса, который определяет общие свойства для различных типов данных

```
interface Box<T> {
  contents: T;
}

const numberBox: Box<number> = { contents: 42 };
const stringBox: Box<string> = { contents: "Hello, world!" };
```

- Пример класса, который принимает дженерик типа и имеет метод, который возвращает значение этого типа:

```
class ValueHolder<T> {
  private value: T;

  constructor(value: T) {
    this.value = value;
  }

  getValue(): T {
    return this.value;
  }
}

const stringValueHolder = new ValueHolder<string>("Hello, world!");
console.log(stringValueHolder.getValue()); // Output: "Hello, world!"

const numberValueHolder = new ValueHolder<number>(42);
console.log(numberValueHolder.getValue()); // Output: 42
```

Вспомогательные типы

- `Partial`

```
Partial<{ x: number; y: number; z: number }>
// ===
{ x?: number; y?: number; z?: number }
```

- `Readonly`

```
Readonly<{ x: number; y: number; z: number }>
// ===
{
  readonly x: number
  readonly y: number
  readonly z: number
}
```

- `Pick`

```
Pick<{ x: number; y: number; z: number }, 'x' | 'y'>
// ===
{ x: number; y: number }
```

- `Record`

```
Record<'x' | 'y' | 'z', number>
// ===
{ x: number; y: number; z: number }
```

- `Exclude`

```
type Excluded = Exclude<string | number, string>
// ===
number
```

Вспомогательные типы

- *Extract*

```ts
type Extracted = Extract<string | number, string>
// ===
string
```

- *NonNullable*

```ts
type NotNull = NonNullable<string | number | void>
// ===
string | number
```

- *ReturnType*

```ts
type ReturnType = ReturnType<() => string>
// ===
string
```

- *InstanceType*

```ts
class Renderer {}
type Instance = InstanceType<typeof Renderer>
// ===
Renderer
```