- **Модуль:** Typescript. Generics and type manipulations
- **Урок:** Conditional types and infer

TypeScript Handbook
from
Иван Петриченко
udemy

ФОРМИРУЕМ ТИПЫ ЧЕРЕЗ УСЛОВИЕ

Типы можно формировать при помощи **тернарного оператора**, где мы дословно спрашиваем: *“этот тип наследуется(или совместим с) от этого? Если да - будет этот тип, нет - другой”*:

```
SomeType extends OtherType ? TrueType : FalseType;
```

```
type Example = "string" extends "Hello" ? string : number; // number, так как литеральные типы разные
```

👉 Условные типы всегда предполагают использование ограничения, то есть ключевого слова **extends.** Проверяемый тип должен чем-то ограничиваться для проверки. В целом, это и есть условие

👉 Мы работаем с типами. При использовании литерала будет **ошибка**. Конкретные значения сначала необходимо преобразовать в тип с помощью оператора **typeof**:

```
const str: string = "Hello";
type Example = "string" extends str ? string : number; // Error
type Example = "string" extends typeof str ? string : number; // Ok, string, тк тип переменной string
```

👉 Базовый синтаксис не очень полезен в работе. Поэтому условные типы вы почти всегда встретите в комбинации с дженериками. Именно там раскрывается главная суть:

```
type FromUserOrFromBase<T extends string | number> = T extends string
    ? IDataFromUser
    : IDataFromBase;

interface User<T extends "created" | Date> {
    created: T extends "created" ? "created" : Date;
}
```

- **Модуль:** Typescript. Generics and type manipulations
- **Урок:** Conditional types and infer

TypeScript Handbook
from
Иван Петриченко
udemy

РЕШЕНИЕ ОШИБКИ В ФУНКЦИЯХ

В примере ниже TS **не может определить**, что же функция в итоге должна вернуть. Эта информация появится только на этапе запуска функции, ведь она зависит **от типа приходящего аргумента.** Отсюда ошибка в том, что компилятор не сможет сопоставить **тип возвращаемого значения** с типом, **возвращаемым в теле** функции:

```ts
function calculateDailyCalories<T extends string | number>(
    numOrStr: T
): T extends string ? IDataFromUser : IDataFromBase {
    if (typeof numOrStr === "string") {
        const obj: IDataFromUser = {
            weight: numOrStr,
        };
        return obj; // Error
    } else {
        const obj: IDataFromBase = {
            calories: numOrStr,
        };
        return obj; // Error
    }
}
```

Такое поведение **соответствует дизайну TS.** Решить её можно прямым указанием того, чем является возвращаемое значение в теле функции:

```ts
return obj as T extends string ? IDataFromUser : IDataFromBase;; // Ok
```

- **Модуль:** Typescript. Generics and type manipulations
- **Урок:** Conditional types and infer

TypeScript Handbook
from
Иван Петриченко
udemy

INFER И ВЕТКИ УСЛОВИЙ

Формирование условных типов можно **разветвлять**, добиваясь нужного результата и комбинируя условия:

```ts
type GetStringType<T extends "hello" | "world" | string> = T extends "hello"
    ? "hello"
    : T extends "world"
    ? "world"
    : string;
```

Редкий гость в TS - это оператор **infer**. Он позволяет **“вытащить”** определенный тип из какой-либо сущности:

```ts
type GetFirstType<T> = T extends Array<infer First> ? First : T;

type Ex = GetFirstType<number[]>; // number
```

```ts
type UnwrappedPromise<T> = T extends Promise<infer Return> ? Return : T;
```

Реализация типа, принимающего любой тип и возвращающего массив этого типа:

```ts
type ToArray<Type> = Type extends any ? Type[] : never;
type ExArray = ToArray<number>; // number[]
type ExArray2 = ToArray<Ex | string>; // number[] | string[]
```

Прием, когда как аргумент передается юнион тип, иногда называется распределительные условные типы.