- **Модуль:** Typescript. Generics and type manipulations
- **Урок:** Generics types and interfaces, constraints

TypeScript Handbook
from
Иван Петриченко
udemy

ОБОБЩЕННЫЕ TYPES И ИНТЕРФЕЙСЫ

Создание **обобщенного type** так же позволяет подставлять нужный тип уже во время использования. Синтаксис использует все те же идентификаторы в угловых скобках:

```ts
type User<T> = {
    login: T;
    age: number;
};
```

Синтаксис

```ts
const user: User<string> = {
    login: 'str',
    age: 54
}
```

Использование

```ts
const user: User<'str'> = {
    login: 'str',
    age: 54
}
```

Использование с литералом

Type так же позволяет создавать **generic helper types** за счет поддержки литеральных значений:

```ts
type OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];

const data: OneOrMany<number[]> = [5];
```

В целом, обобщенные type в практике вы будете встречать **редко**. Куда чаще используются интерфейсы, которые позволяют помещать в объекты разные данные:

```ts
interface User<T> {
    login: T;
    age: number;
};
```

```ts
interface User<ParentsData> {
    login: string;
    age: number;
    parents: ParentsData
};
```

GENERIC CONSTRAINS

То, что мы можем передать в дженерик **любые данные** - это и преимущество и недостаток. В обобщенных функциях мы использовали сужение типов для работы с разными данными, но для ограничения входящих данных есть механизм **ограничений, generic constrains**

Проблема следующего кода в том, что при создании объекта мы можем поместить в свойство parents все что угодно. А по задумке это должен быть объект со свойствами **mother и father**:

```ts
interface User<ParentsData> {
    login: string;
    age: number;
    parents: ParentsData
};

const user: User<{mother: string, father: string}> = {
    login: 'str',
    age: 54,
    parents: {mother: 'Anna', father: 'no data'}
}
```

Эту проблему можно решить, если создать отдельный интерфейс и типизировать свойство parents. **Но**, проблема в том, что тогда в этот объект **не сможет** попасть никакое другое свойство. А мы бы хотели сделать его **расширяемым**, но с **двумя обязательными** свойствами:

```ts
interface ParentsOfUser {
    mother: string; father: string
}

interface User {
    login: string;
    age: number;
    parents: ParentsOfUser;
}

const user: User = {
    login: "str",
    age: 54,
    parents: { mother: "Anna", father: "no data" } // Никаких других свойств
};
```

Для решения этой задачи и создан механизм ограничения, который позволит “ограничить” идентификатор в дженерике. В данном случае мы можем сказать, что он будет **только объектом любого размера с двумя обязательными свойствами.** Для этого используем **extends:**

```ts
interface ParentsOfUser {
    mother: string;
    father: string;
}

interface User<ParentsData extends ParentsOfUser> {
    login: string;
    age: number;
    parents: ParentsData;
}

const user: User<{ mother: string; father: string; married: boolean }> = {
    login: "str",
    age: 54,
    parents: { mother: "Anna", father: "no data", married: true },
};
```

Для ограничений можно использовать и **примитивные типы,** в том числе и union. Например функция, которая принимает только строку или число:

```ts
const depositMoney = <T extends number | string>(amount: T): T => {
    console.log(`req to server with amount: ${amount}`)
    return amount;
}

depositMoney(500);
depositMoney('500');
depositMoney(true) // Error
```

Альтернативный вариант с использованием обычного union типа так же можно использовать. Но тут будет **повторение** кода:

```ts
const depositMoney = (amount: number | string): number | string => {
    console.log(`req to server with amount: ${amount}`);
    return amount;
};

depositMoney(500);
depositMoney("500");
```