35. TypeScript
Продолжаем изучение

Цель:

Познакомиться со следующими возможностями typescript:

- generics
- классы
- Utility Types

Generics:

Generic позволяет резервировать место для типа, который будет заменен после на конкретный переданный, при вызове функции или метода, а также при работе с классами.

Для чего существуют обобщенные типы

- Безопасность типов
- Более простой и понятный код

Generics. Базовые ограничения обобщенных типов:

Можно ограничивать не только принимаемые аргументы по типу, но и сами параметр-типы.

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Now we know it has a .length property, so no more error
  return arg;
}
```

Использование параметров типа в ограничениях:

Можно определить параметр типа, который будет ограничен типом другого параметра типа.

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
```

```
Argument of type '"m"' is not assignable to parameter of type '"a" | "b" | "c" | "d"'.
```

Обобщения и интерфейсы:

Некоторые встроенные типы для обобщений:

- Partial<T> – указывает, что все свойства некоторого типа T являются необязательными
- Readonly<T> – указывает, что все свойства некоторого типа T доступны только для чтения
- Record<K extends string, T> – конструирует объект, у которого значения свойств имеют некоторый тип T
- Pick<T, K extends keyof T> – выделяет из некоторого типа T некоторый набор свойств K

Классы:

Класс — это шаблон, используя который мы можем создавать экземпляры объектов, у которых будет точно такая же конфигурация, как и у шаблона — те же свойства и методы. Интерфейс — это группа взаимосвязанных свойств и методов, которые описывают объект, но не обеспечивают реализацию или инициализацию этих свойств и методов в объектах.

В TS есть полноценная поддержка классов.

Поля и методы класса:

```
class Point {
  x: number;
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

```
class Point {
  x = 10;
  y = 10;

  scale(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

Модификаторы доступа:

Модификаторы доступа позволяют скрыть состояние объекта от внешнего доступа и управлять доступом к этому состоянию.

В TypeScript три модификатора: **public, protected и private.**

Если к свойствам и функциям классов не применяется модификатор, то такие свойства и функции расцениваются как будто они определены с модификатором **public**.

Модификаторы доступа:

- **public:** свойства публичными, то следует использовать модификатор
- **private**:
  Если же к свойствам и методам применяется модификатор private, то к ним нельзя будет обратиться извне при создании объекта данного класса.
- **protected:**
  Модификатор protected определяет поля и методы, которые извне класса видны только в классах-наследниках
- **readonly:**
  поле для чтения, то к модификатору доступа добавляется модификатор

Реализация интерфейса:

```
interface Pingable {
  ping(): void;
}

class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}

class Ball implements Pingable {
```

```
Class 'Ball' incorrectly implements interface
'Pingable'.
  Property 'ping' is missing in type 'Ball'
but required in type 'Pingable'.
```

```
  pong() {
    console.log("pong!");
  }
}
```

Статические поля и методы:

Статические поля и методы относятся не к отдельным объектам, а в целом к классу. И для обращения к статическим полям и методам применяется имя класса.

В статических методах мы можем обращаться к статическим полям или другим статическим методам класса, **но мы не можем обращаться к не статическим полям и методам и использовать ключевое слово this**.

Статические поля и методы:

Статические поля и методы относятся не к отдельным объектам, а в целом к классу. И для обращения к статическим полям и методам применяется имя класса.

В статических методах мы можем обращаться к статическим полям или другим статическим методам класса, **но мы не можем обращаться к не статическим полям и методам и использовать ключевое слово this**.

Абстрактные классы, методы и поля:

Главное отличие абстрактного класса от обычного класса заключается в отсутствии возможности создания его экземпляров.

```
abstract class Base {
  abstract getName(): string;

  printName() {
    console.log("Hello, " + this.getName());
  }
}

const b = new Base();
```

```
Cannot create an instance of an abstract
class.
```

Utility Types. Pick (отфильтровать объектный тип):

```ts
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};

todo;
```

```
const todo: TodoPreview
```

Exclude (исключает из T признаки присущие U):

```ts
type T0 = Exclude<"a" | "b" | "c", "a">;
```

```ts
type T0 = "b" | "c"
```

```ts
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;
```

```ts
type T1 = "c"
```

```ts
type T2 = Exclude<string | number | (() => void), Function>;
```

```ts
type T2 = string | number
```

Extract (общие для двух типов признаки):

```ts
type T0 = Extract<"a" | "b" | "c", "a" | "f">;
```

```ts
type T0 = "a"
```

```ts
type T1 = Extract<string | number | (() => void), Function>;
```

```ts
type T1 = () => void
```

Omit:

Исключить из T признаки ассоциированными с ключами перечисленных множеством K.

Omit<T, K> может быть полезен тогда, когда нам необходимо определить тип, который представляет собой некоторую часть уже существующего типа.

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
  createdAt: number;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
  createdAt: 1615544252770,
};

todo;

  const todo: TodoPreview


type TodoInfo = Omit<Todo, "completed" | "createdAt">;

const todoInfo: TodoInfo = {
  title: "Pick up kids",
  description: "Kindergarten closes at 5pm",
};

todoInfo;

    const todoInfo: TodoInfo
```