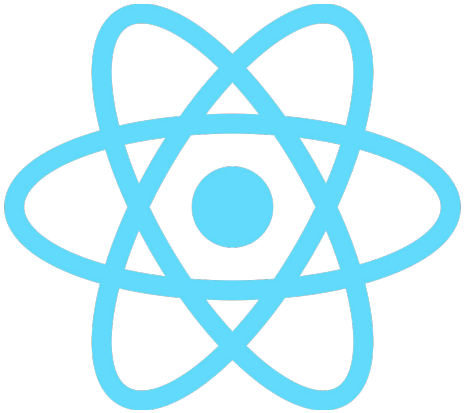
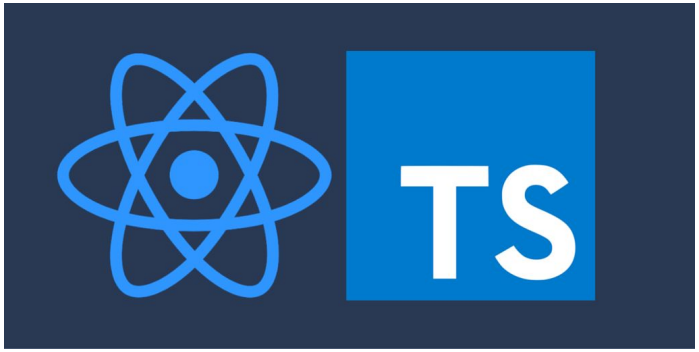


Тема занятия:

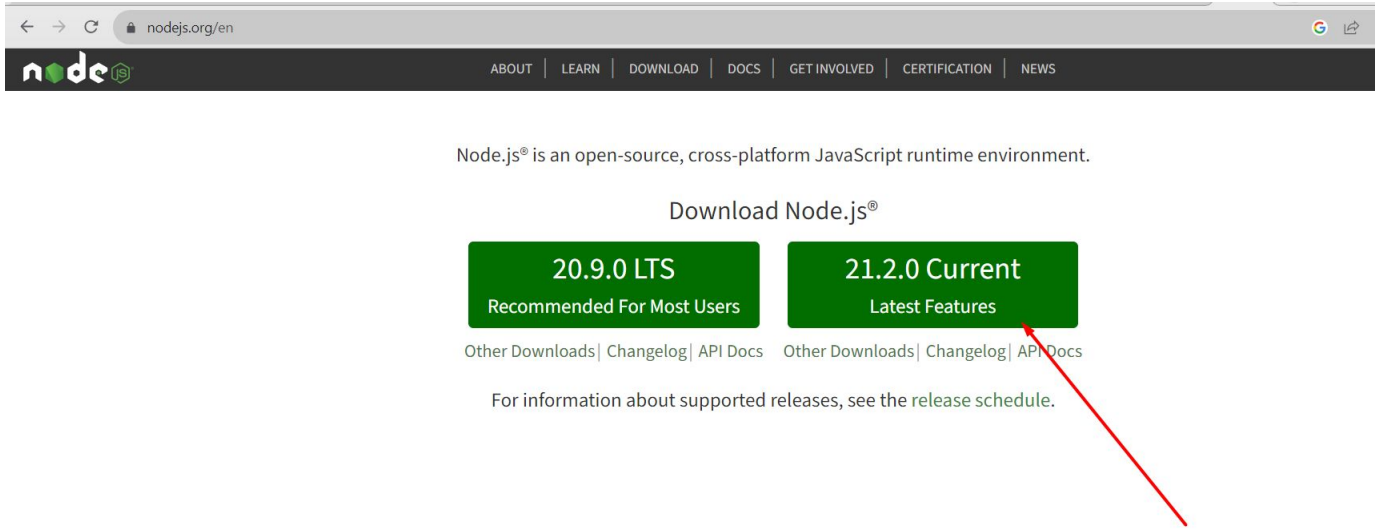
React: TypeScript



Создание React проекта с TypeScript

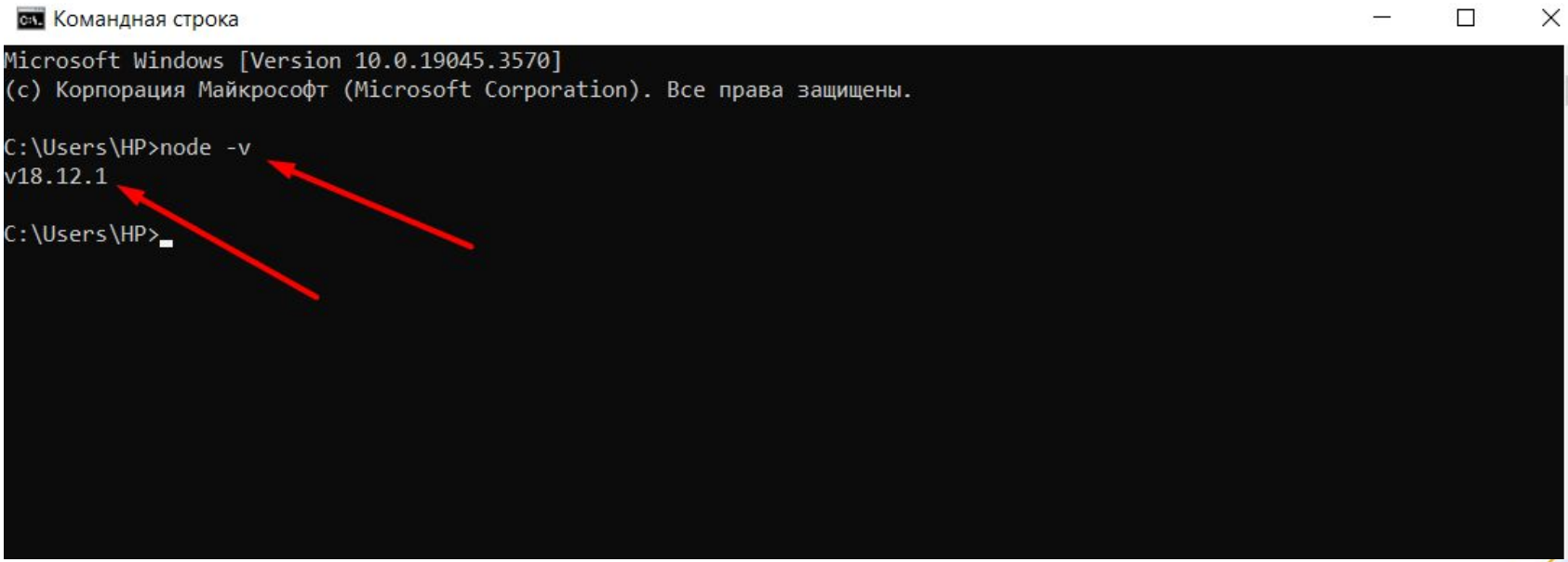
Установка NodeJS

1. Перейдите на сайт https://nodejs.org/en
2. Скачайте и установите последнюю версию **Node.js**



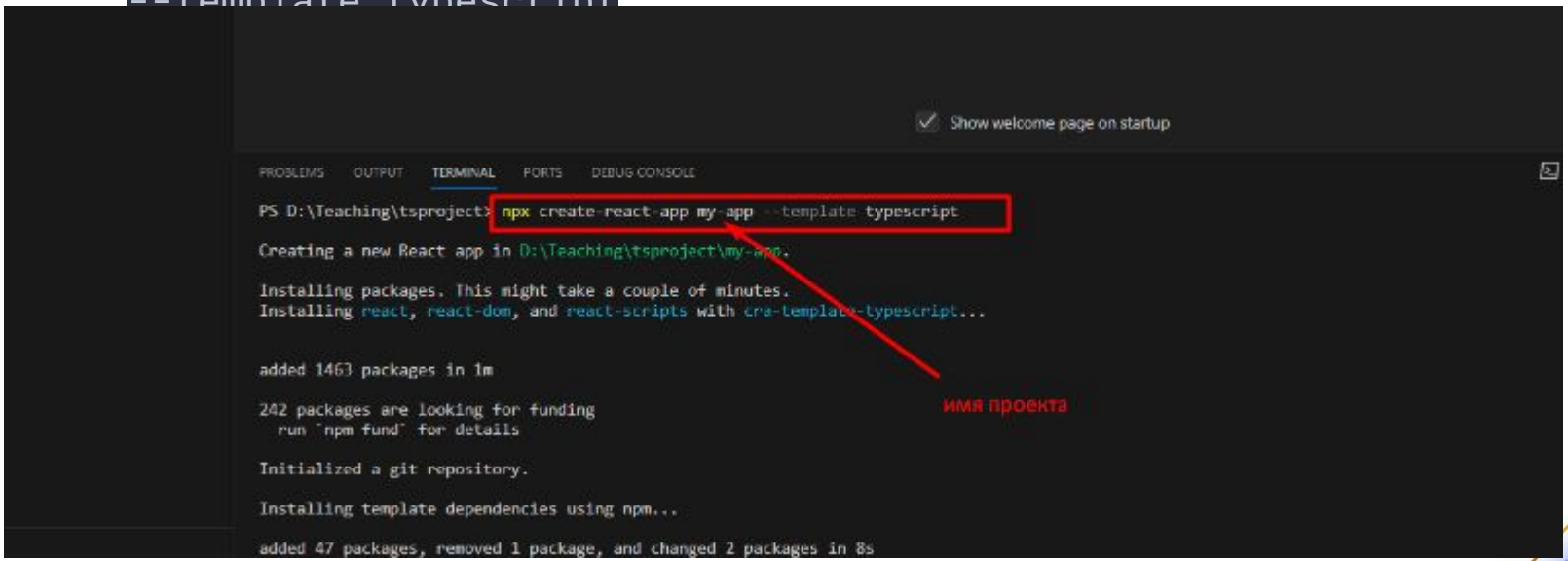
Проверка версии NodeJS

1. Перейдите в командную строку или терминал
2. Введите команду **node -v**



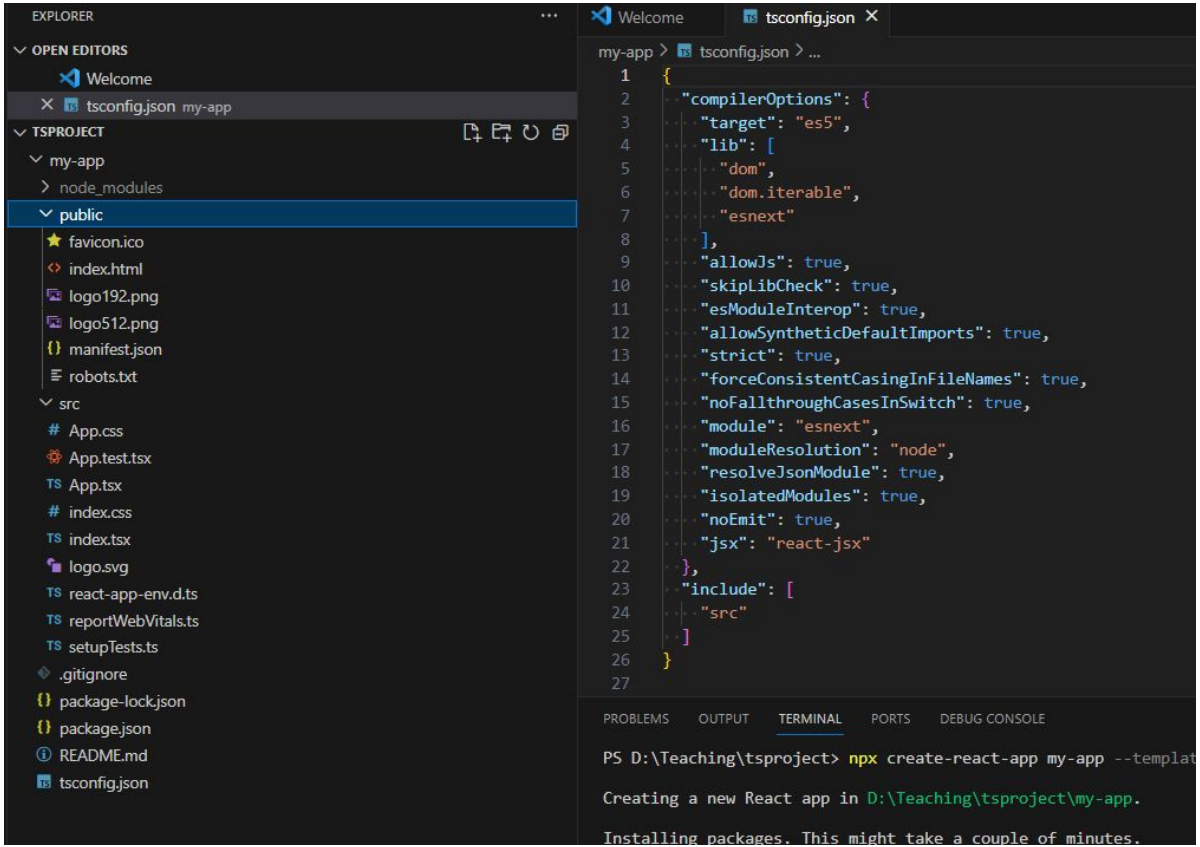
Создание проекта

1. Откройте **VSCode,** перейдите в папку, в которой будет лежать ваш проект и в терминале введите следующую команду - `npx create-react-app my-app --template typescript`



Структура проекта my-app

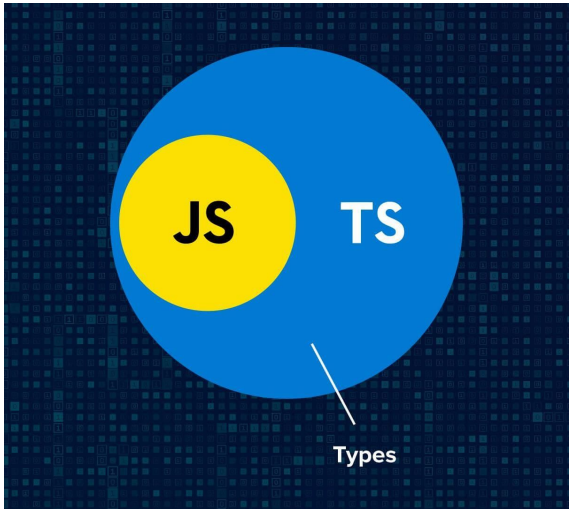
Структура проекта схожа со структурой проекта без TypeScript



TypeScript

TypeScript - это типизированная расширенная версия JavaScript со статической типизацией, которая компилируется в простой JavaScript.

Именно эти две особенности позволяют создавать масштабные приложения, сохраняя качество и упрощая разработку.



В Typescript поддерживает все типы, которые есть в Javascript, дополняя их несколькими дополнительными типами

Рассмотрим основные типы:

boolean

Тип boolean представляет логическое значение true или false:

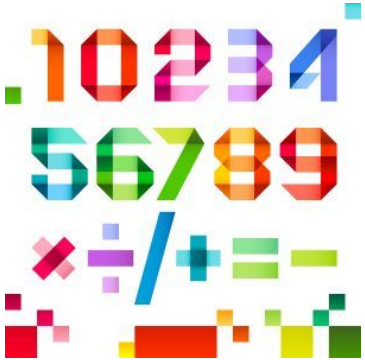
```
let isEnabled: boolean = true;
let isAlive: boolean = false;

console.log(isEnabled);
console.log(isAlive);
```

number

Тип number представляет числа, причем все числа в TypeScript, как и в JavaScript, являются числами с плавающей точкой. Поэтому с помощью данного типа можно определять как целые числа, так и числа с плавающей точкой:

```
let age: number = 36;
let height: number = 1.68;
```



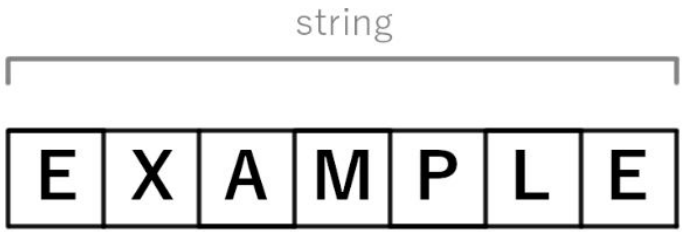
string

String представляет строки. Как и в JavaScript, в TypeScript строки можно заключать в двойные, либо в одинарные кавычки:

```
let firstName: string = "Tom";
```

TypeScript поддерживает такую функциональность, как шаблоны строк, то есть мы можем задать шаблон в косых кавычках (`), как если бы мы писали обычную строку, и затем в саму строку можно встраивать разные выражения с помощью синтаксиса ${ expr }, где expr - это выражение. Например:

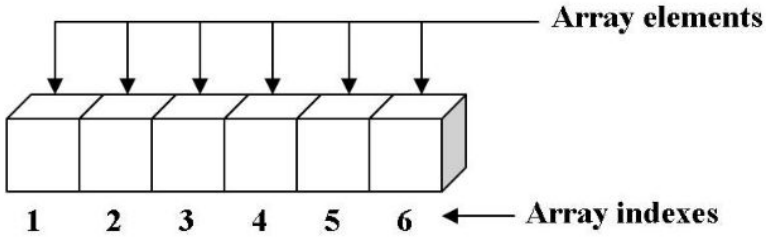
```
let firstName: string = "Tom";
let age: number = 28;
let info: string = `Имя ${firstName} Возраст: ${age}`;
console.log(info);  // Имя Tom  Возраст: 28
```



Array

Тип Array используется для указания элементов массива.

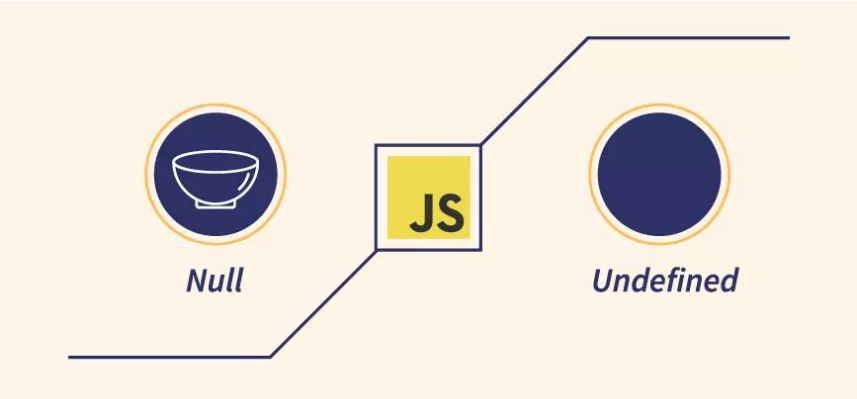
```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```



null и undefined

null и undefined сами по себе они могут быть значениями для переменных и типами

```
let a: null = null;
let b: undefined = undefined;
```



tupel

Этот тип ещё называют кортежем. Он позволяет определить массив фиксированной длины с элементами разных типов.

```
let person: [string, number] = ["John", 25];
```

void

Используется для функций, которые не возвращают значения.

```
function sayHello(): void {
    console.log("Hello!");
}
```

any

Any описывает данные, тип которых может быть неизвестен на момент написания приложения.

```
1  let someVar: any = "hello";
2  console.log(someVar);   // сейчас someVar - это string
3  someVar = 20;
4  console.log(someVar);   // сейчас someVar - это number
```

enum

Тип enum или перечисление позволяет определить набор именнованных констант, которые описывают определенные состояния.

Для определения перечисления применяется ключевое слово enum. Например, объявим простейшее перечисление:

```
enum Season { Winter, Spring, Summer, Autumn };
```

Перечисление называется **Season** и имеет четыре элемента. Теперь используем перечисление:

```
enum Season { Winter, Spring, Summer, Autumn };

let current: Season = Season.Summer;

console.log(current);       // 2
```

enum

Числовые перечисления

По умолчанию константы перечисления, как в примере выше, представляют числовые значения. То есть это так называемое числовое перечисление, в котором каждой константе сопоставляется числовое значение.

```
1  enum Season { Winter, Spring, Summer, Autumn };
```

Эквивалентно

```
1  enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };
```

enum

Числовые перечисления

Хотя мы можем явным образом переопределить эти значения. Так, мы можем задать значение одной константы, тогда значения следующих констант будет увеличиваться на единицу:

```
1  enum Season { Winter=5, Spring, Summer, Autumn };          // 5, 6, 7, 8
```

Либо можно каждой константе задать свое значение:

```
1  enum Season { Winter=4, Spring=8, Summer=16, Autumn=32 };   // 4, 8, 16, 32
```

enum

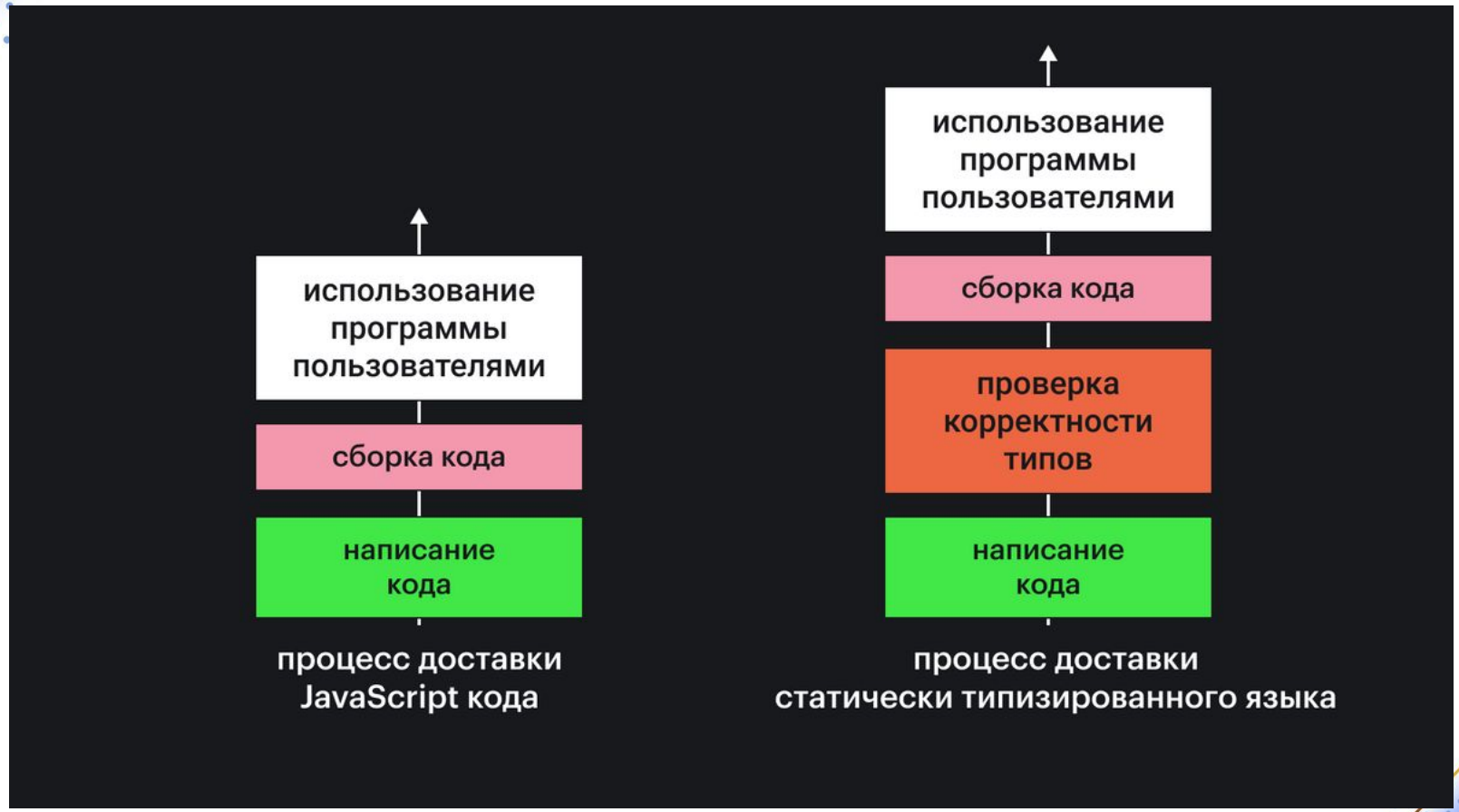
Строковые перечисления

Кроме числовых перечислений в TypeScript есть строковые перечисления, константы которых принимают строковые значения:

```
enum Season {
    Winter = "Зима",
    Spring = "Весна",
    Summer = "Лето",
    Autumn = "Осень"
};
var current: Season = Season.Summer;
console.log(current);   // Лето
```

Также можно определять смешанные перечисления, константы которых могут числа и строки.

Ошибки несовпадения типов будут заметны на стадии написания кода.



Ключевые особенности TypeScript

1. Явная аннотация типа (Explicit Type Annotation):

TypeScript позволяет явно указывать типы переменных, функций и других элементов кода. Это позволяет обнаруживать ошибки на этапе компиляции, что может сделать код более надежным и поддерживаемым.

```
function add(a: number, b: number): number {
    return a + b;
}

let result: number = add(5, 10);
```

Ключевые особенности TypeScript

2. Вывод типа (Type Inference):

TypeScript автоматически определяет тип на основе контекста, если явная аннотация типа не указана.

```
let age = 25; // TypeScript автоматически определит тип number

function greet(person: string) {
    console.log("Hello, " + person + "!");
}
```

Ключевые особенности TypeScript

3. Union и Intersection типы:

Позволяют объединять или пересекать несколько типов.

```
type StringOrNumber = string | number;

let value: StringOrNumber = "hello";
```

Ключевые особенности TypeScript

4. Интерфейсы (Interfaces):

TypeScript позволяет создавать пользовательские типы данных с помощью интерфейсов.

```
interface Person {
    name: string;
    age: number;
}

let person: Person = { name: "John", age: 25 };
```

Ключевые особенности TypeScript

5. Типы (Type Aliases):

Позволяют создавать именованные типы

```
type Point = { x: number; y: number };

function move(point: Point): Point {
    // some logic
    return { x: point.x + 1, y: point.y + 1 };
}
```

Разница между interface и type

Interface

- Интерфейсы могут объединяться (мержиться), если они имеют одинаковые имена.

```
// Исходный интерфейс
interface Car {
    brand: string;
    speed: number;
}

// Объединение интерфейсов (добавление нового свойства)
interface Car {
    color: string;
}

// Теперь интерфейс Car объединен с новым свойством color
let myCar: Car = {
    brand: "Toyota",
    speed: 120,
    color: "blue",
};
```

- Интерфейсы могут расширяться (extends), что позволяет использовать один интерфейс как базовый для другого.

```
interface Employee extends Person {
    role: string;
}
```

Разница между interface и type

Type

- Типы не объединяются автоматически. Если вы объявляете тип с тем же именем, это приведет к ошибке.

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

// Ошибка: Duplicate identifier 'Person'.
type Person = {
    gender: string;
};
```

- Типы могут быть использованы с оператором & для создания объединенного (intersection) типа.

```
type Employee = Person & { role: string };
```

Задание необязательных параметров в interface и type

В TypeScript можно делать параметры необязательными, добавляя к ним знак вопроса **?**. Это касается как интерфейсов (interfaces), так и типов (type aliases). Необязательные параметры означают, что вы можете не предоставлять значение для данного параметра при создании объекта, и TypeScript не будет считать это ошибкой.

```typescript
interface Person {
    name: string;
    age?: number; // Возраст необязателен
}

let person1: Person = { name: "John" };
let person2: Person = { name: "Jane", age: 25 };
```

Ключевые особенности TypeScript

6. Дженерики (Generic Types):

Дженерики (generics) предоставляют средства для создания обобщенных типов и функций. Это позволяет писать более универсальный и безопасный код, который может работать с различными типами данных, не теряя статической типизации

Пример

```
// Обобщенный тип для массива любого типа
type MyArray<T> = T[];

// Пример использования
const stringArray: MyArray<string> = ['a', 'b', 'c'];
const numberArray: MyArray<number> = [1, 2, 3];
```

Ключевые особенности TypeScript

6. Дженерики (Generic Types):

В дженериках можно задавать Default значения

Пример

```
// Обобщенный тип с default значением
type MyDefaultArray<T = number> = T[];

// Пример использования
const numberArray: MyDefaultArray = [1, 2, 3];
const stringArray: MyDefaultArray<string> = ['a', 'b', 'c'];
```

Ключевые особенности TypeScript

6. Дженерики (Generic Types):

Дженерики можно использовать в интерфейсах

Пример

```typescript
// Обобщенный интерфейс для объекта с одним свойством
interface KeyValue<T> {
  key: string;
  value: T;
}

// Пример использования
const stringKeyValue: KeyValue<string> = { key: 'name', value: 'John' };
const numberKeyValue: KeyValue<number> = { key: 'age', value: 30 };
```