



Лекция №1

Синтаксис языка TypeScript. Стрелочные функции. Встроенные и особые типы. Формат JSON.

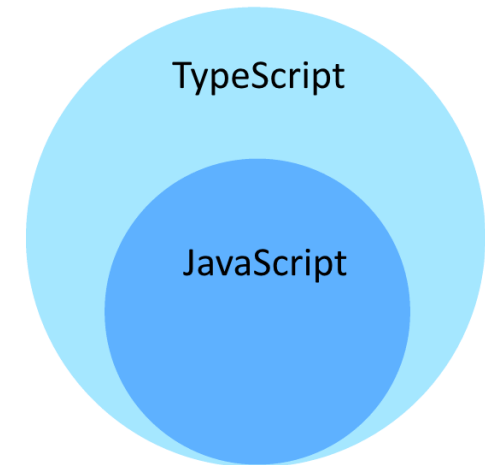


➤ Что такое TypeScript?

Это язык программирования, расширяющий возможности по типизации данных языка JavaScript, принося в него те недостающие качества, позволяющие облегчить понимание и написание кода, исключая часто возникающие ошибки при написании этого же самого кода на языке JavaScript.

➤ Что из этого следует?

TypeScript является обратно совместимым с JavaScript и компилируется в последний. После компиляции кода, его можно выполнять в любом современном браузере или использовать совместно с серверной платформой Node.js.



Что нужно, чтобы начать работать с TypeScript

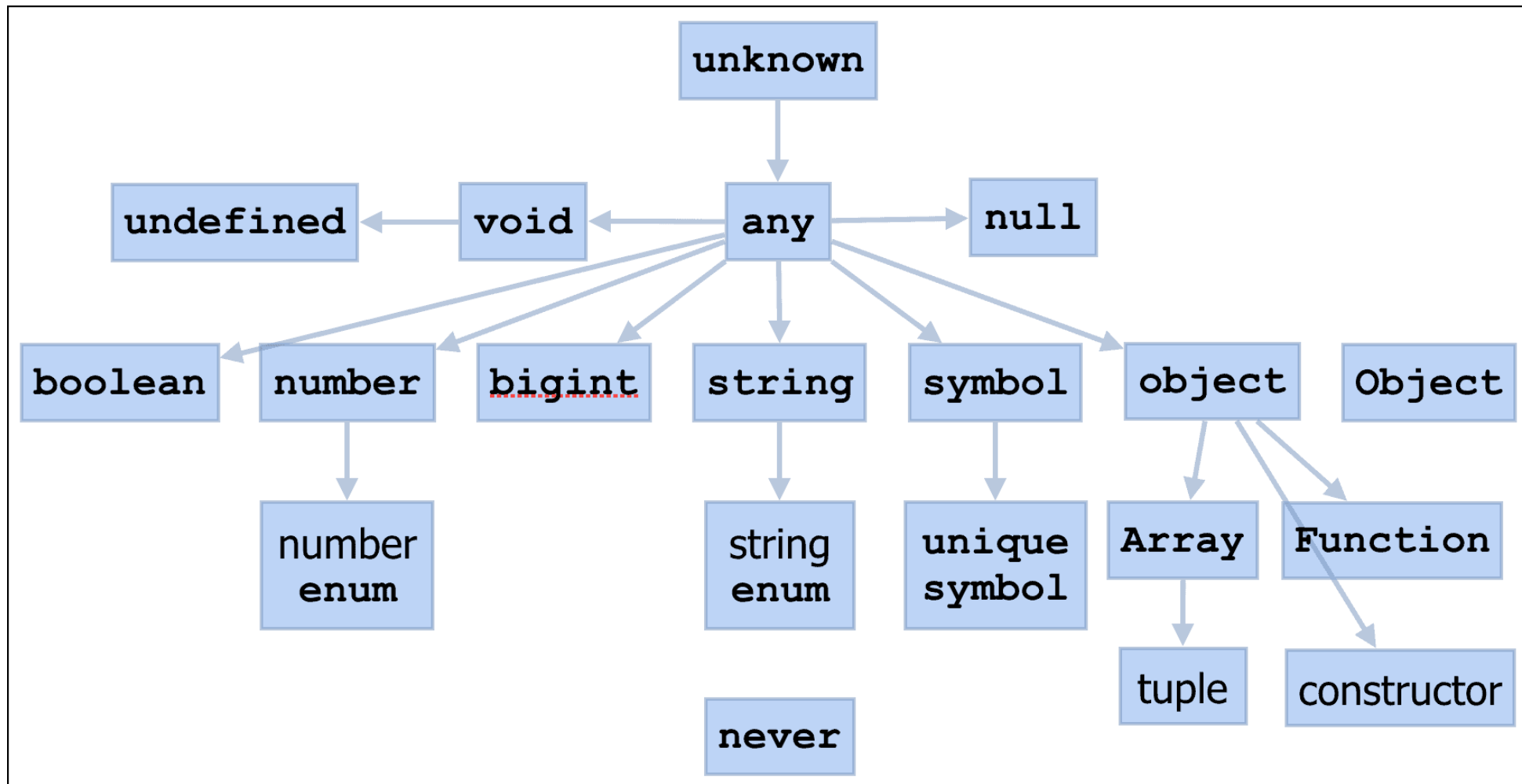


➤ Недостатки JavaScript?

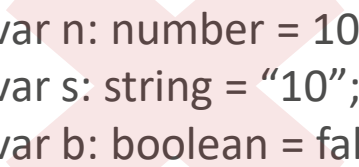
- Отсутствие модульности — могут возникать проблемы из-за конфликта больших файлов.
- Нелогичное поведение.
- Динамическая типизация — нет IntelliSense, из-за чего мы не видим, какие ошибки будут возникать во время написания, а видим их только во время исполнения программы

➤ Достоинства TypeScript?

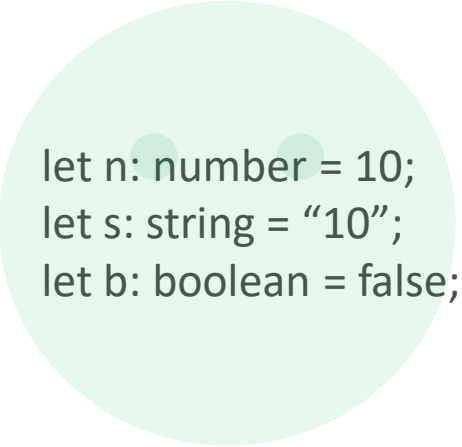
- Статическая типизация.
- Компилируется в «хороший» JavaScript.
- Легко читаемый код, который понятен с первого взгляда.
- Делает язык приближенным к языкам общего назначения



```
const n: number = 10;  
const s: string = "10";  
const b: boolean = false;
```



```
var n: number = 10;  
var s: string = "10";  
var b: boolean = false;
```



```
let n: number = 10;  
let s: string = "10";  
let b: boolean = false;
```

```
interface Person {  
  id: number;  
  name: string;  
  surname: string;  
  move: (number) => void;  
  justFunction(string)? : number;  
}
```



```
let person: Person = {  
  id: 1;  
  name: "Анастасия",  
  surname: "Петрова-Водкина",  
  move: (move: number) => {  
    //I Like To Move It;  
    return; //Можно не писать так!  
  }  
}
```

Интерфейсы могут быть:

- Преобразованы с помощью Утилитарных типов
- Расширены (extends)
- Быть вложены один в другой

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

```
enum Season {  
  Winter = "Зима",  
  Spring = "Весна",  
  Summer = "Лето",  
  Autumn = "Осень "  
};
```

```
enum Season {  
  Winter = 1,  
  Spring = "Весна",  
  Summer = 3,  
  Autumn = "Осень"  
};
```



```
enum RolePart1 { CallManager, Admin, Director};
```

```
enum RolePart2 { Nurse, Practitioner };
```

Без псевдонима

```
interface Employee {  
  id: number;  
  name: string;  
  surname: string;  
  role: RolePart1 & RolePart2;  
}
```

type Role: RolePart1 | RolePart2;

```
interface Employee {  
  id: number;  
  name: string;  
  surname: string;  
  role: Role;  
}
```

Пересечение типов производится с помощью оператора &
Объединение типов производится с помощью оператора |

Синтаксис языка TypeScript :: Утилитарные вспомогательные типы

`Awaited<T>` - это специальный тип, который может быть использован для обозначения типа, который будет возвращен из асинхронной функции

`Partial<T>` - делает все свойства объекта типа `T` необязательными.

`Required<T>` - делает все свойства объекта типа `T` обязательными.

`Readonly<T>` - делает все свойства объекта типа `T` доступными только для чтения

`Record<Keys, Type>` - создает тип, который является записью с ключами, определенными в первом параметре, и значениями типа, определенного во втором параметре

`Pick<T, K extends keyof T>` - выбирает свойства объекта типа `T` с ключами, указанными в `K`.

`Exclude<UnionType, ExcludedMembers>` - исключает определенные типы из объединенного типа

```
interface Todo {  
  title: string;  
}
```

```
const todo: Readonly<Todo> = {  
  title: "Delete inactive users",  
};
```

Если мы захотим изменить значение `title` после воздействия утилитарного типа `Readonly`, то получим ошибку

```
todo.title = "Hello"; //Произойдет ошибка при компиляции
```

Cannot assign to 'title' because it is a read-only property.

```
function getCarName(manufacturerName: string, model?: string): string {  
  if (model) {  
    return manufacturerName + " " + model;  
  }  
  return manufacturerName;  
}
```

```
const getCarName = (manufacturerName: string, model?: string) => {  
  if (model) {  
    return manufacturerName + " " + model;  
  }  
  return manufacturerName;  
}
```

const carName: string =
getCarName("Toyota", "Land Cruiser
Prado"); // => Toyota Land Cruiser Prado

ОПРЕДЕЛЯЕМ СТРЕЛОЧНУЮ ФУНКЦИЮ

```
const getCarName = (manufacturerName: string, model?: string) => {  
  if (model) {  
    return manufacturerName + " " + model;  
  }  
  return manufacturerName;  
}
```

ВЫЗЫВАЕМ ЭТУ СТРЕЛОЧНУЮ ФУНКЦИЮ ИЗ ОБЫЧНОЙ

```
function getCarNameSimple(carFunction: function, manufacturerName: string, model?: string):  
string {  
  return carFunction(manufacturerName, model);  
}
```

```
getCarNameSimple(getCarName, "Toyota", "Land Cruiser Prado"); // => Toyota Land Cruiser Prado  
getCarNameSimple(getCarName, "Toyota"); // => Toyota
```

```
getCarName("Toyota"); // => Toyota  
getCarName(getCarName, "Toyota", "Land Cruiser Prado"); // => Toyota Land Cruiser Prad
```

JSON (англ. JavaScript Object Notation) – текстовый формат представления данных, основанный на JavaScript, но является независимым форматом, используемым повсеместно в программировании

В общем представлении формат может быть записан в следующем виде:

node = { key: (node | [node]) | value }

где

key – ключ (строка, уникальная для каждого узла node),

value – строка, число или булево значение (конкретное значение).

Пример использования JSON при сериализации данных в TypeScript

JSON (англ. JavaScript Object Notation) – текстовый формат представления данных, основанный на JavaScript, но независимый от него

```
interface Person {  
  id: number;  
  name: string;  
  surname: string;  
}
```

```
let person: Person = {  
  id: 1;  
  name: "Анастасия",  
  surname: "Петрова-Водкина",  
}
```



```
{  
  "id": 1;  
  "name": "Анастасия",  
  "surname": "Петрова-Водкина"  
}
```

```
← const jsonPerson: string = JSON.strignify(person);
```

Пример использования JSON при десериализации данных в TypeScript

JSON (англ. JavaScript Object Notation) – текстовый формат представления данных, основанный на JavaScript, но независимый от него

```
interface Person {  
  id: number;  
  name: string;  
  surname: string;  
}
```

```
let person: Person = {  
  id: 1;  
  name: "Анастасия",  
  surname: "Петрова-Водкина",  
}
```

```
const jsonPerson: string = "{  
  \"id\": 1;  
  \"name\": \"Анастасия\",  
  \"surname\": \"Петрова-Водкина\"  
}"
```

```
const person: Person = JSON.parse(jsonPerson);
```





РУБРИКА: «ВОПРОСЫ СЛУШАТЕЛЕЙ»





КОНЕЦ ЛЕКЦИИ № 1
СПАСИБО ЗА ВНИМАНИЕ

