TypeScript

Статическая типизация для JavaScript

Занятие 7

Frontend Course 2025

Проблемы JavaScript

> **Вопрос:** У нас уже есть JavaScript. Зачем нужен TypeScript?

TypeScript решает `проблемы безопасности` и `масштабируемости` JavaScript.

Основные проблемы JavaScript:

- Runtime ошибки вместо compile-time
- Сложно рефакторить большие проекты
- Плохое автодополнение в IDE

Пример:

```
function getUserData(user) {
  return user.name.toUpperCase();
}
// undefined is not a function
getUserData(null); // Ошибка только в runtime!
```

Что такое TypeScript?

> **Вопрос:** Это отдельный язык или расширение JavaScript?

Надстройка над JavaScript! TypeScript компилируется в обычный JavaScript.

Что добавляет TypeScript:

- **Статическая типизация** - проверки до запуска
- **Современный синтаксис** - ES6+, классы, модули
- **Отличная поддержка IDE** - автодополнение, рефакторинг
- **Документация через типы** - самодокументирующийся код

Как работает:

```
// TypeScript → JavaScript
const user: User = { id: 1, name: "Иван" };
// Компилируется → const user = { id: 1, name: "Иван" };
```

Базовая типизация

Примитивные типы:

```
let name: string = "Иван";
let age: number = 25;
let isActive: boolean = true;
```

Массивы:

```
let numbers: number[] = [1, 2, 3];
let users: string[] = ["Анна", "Борис"];
let mixed: (string | number)[] = [1, "два", 3];
```

Объекты:

```
interface User {
  id: number;
  name: string;
  email?: string; // опциональное
}

const user: User = { id: 1, name: "Иван" };
```

Функции с типизацией

Обычные функции:

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

Стрелочные функции:

```
const multiply = (a: number, b: number): number => a * b;
```

Опциональные параметры:

```
function greet(name: string, greeting?: string): string {
  return greeting ? `${greeting}, ${name}!` : `Привет, ${name}!`;
}
```

Функции с параметрами по умолчанию:

```
function createUrl(path: string, domain: string = "example.com"): string {
  return `https://${domain}${path}`;
}
```

Union и Literal типы

> **Вопрос:** Как описать переменную, которая может быть разного типа?

Union типы! Переменная может быть одним из нескольких типов.

```
// Union - может быть один из типов
let value: string | number = "строка";
value = 123; // OK
value = true; // Ошибка
```

Literal типы - конкретные значения:

```
type Theme = "light" | "dark" | "auto";

function setTheme(theme: Theme) {
  console.log(theme);
}

setTheme("light"); // OK
setTheme("custom"); // ❌ Ошибка
```

Интерфейсы и объекты

> **Вопрос:** Как описать структуру сложных объектов?

Интерфейсы! Контракты для структуры данных.

```
interface User {
  id: number;
  name: string;
  email: string;
  age?: number; // опциональное
  readonly createdAt: Date; // только для чтения
}

interface AdminUser extends User {
  permissions: string[];
  adminLevel: number;
}
```

Интерфейсы: пример использования

```
const admin: AdminUser = {
  id: 1,
  name: "Администратор",
  email: "admin@example.com",
  createdAt: new Date(),
  permissions: ["read", "write"],
  adminLevel: 5,
};
```

Дженерики - переиспользуемость

> **Вопрос:** Как создавать функции, работающие с разными типами данных?

Дженерики! Обобщенные типы для переиспользуемого кода.

```
// Обобщенная функция
function identity<T>(arg: T): T {
  return arg;
}

const num = identity<number>(123); // number
const str = identity("строка"); // auto-inferred
```

Дженерики: интерфейсы

```
// Интерфейсы с дженериками
interface Repository<T> {
  findById(id: number): Promise<T | null>;
  save(entity: T): Promise<T>;
}

// Использование
class UserRepository implements Repository<User> {
  async findById(id: number): Promise<User | null> {
    // реализация
  }
  async save(user: User): Promise<User> {
    // реализация
  }
}
```

Классы с модификаторами

Модификаторы доступа:

Модификаторы доступа:

- `public` - доступно везде
- `private` - только внутри класса
- `protected` - в классе и наследниках
- `readonly` - только для чтения

```
class User {
  public name: string;
  private age: number;
  protected email: string;
  readonly id: number;

  constructor(name: string, age: number, email: string) {
    this.name = name;
    this.age = age;
    this.email = email;
    this.id = Math.random();
  }

  public getInfo(): string {
    return `${this.name}, ${this.age} лет`;
  }
}
```

Утилитарные типы

> **Вопрос:** Как создавать новые типы на основе существующих?

Утилитарные типы! Встроенные типы для трансформации.

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

// Утилитарные типы
type UpdateTodo = Partial<Todo>; // все поля опциональны
type TodoSummary = Pick<Todo, "title">; // только указанные поля
type TodoPublic = Omit<Todo, "desc">; // все кроме указанных
type ThemeConfig = Record<"primary", string>; // объект с типами
```

DOM типизация

> **Вопрос:** Как работать с HTML элементами в TypeScript?

Типизация DOM API! Все элементы и события имеют типы.

```
// Типизированные элементы DOM
const button = document.querySelector("#myButton") as HTMLButtonElement;
const input = document.querySelector("#myInput") as HTMLInputElement;
const form = document.querySelector("form") as HTMLFormElement;
```

DOM: типизированные события

```
// Типизированные обработчики событий
button.addEventListener("click", (event: MouseEvent) => {
  console.log("Клик в координатах:", event.clientX, event.clientY);
});

input.addEventListener("input", (event: Event) => {
  const target = event.target as HTMLInputElement;
  console.log("Значение:", target.value);
});

form.addEventListener("submit", (event: SubmitEvent) => {
  event.preventDefault();
  console.log("Форма отправлена");
});
```

Асинхронность с типизацией

Promise с типизацией:

```
const fetchUsers = (): Promise<User[]> => {
  return fetch("/api/users").then((res) => res.json());
};
```

Async/Await:

```
const loadUser = async (id: number): Promise<User | null> => {
  try {
    const response = await fetch(`/api/users/${id}`);
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  } catch (error) {
    console.error("Ошибка загрузки пользователя:", error);
    return null;
  }
};
```

Использование:

```
const user = await loadUser(1);
if (user) {
  console.log(user.name); // TypeScript знает, что user не null
}
```

Type Guards

> **Вопрос:** Как TypeScript понимает типы внутри условий?

Type Guards! Функции для проверки типов в runtime.

```
interface Cat {
  type: "cat";
  meow: () => void;
}

interface Dog {
  type: "dog";
  bark: () => void;
}

type Animal = Cat | Dog;

function isCat(animal: Animal): animal is Cat {
  return animal.type === "cat";
}

function makeSound(animal: Animal) {
  if (isCat(animal)) {
    animal.meow(); // TypeScript знает, что это Cat
  } else {
    animal.bark(); // TypeScript знает, что это Dog
  }
}
```

Конфигурация TypeScript

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true,
    "outDir": "./dist",
    "sourceMap": true,
    "noUnusedLocals": true,
    "noImplicitReturns": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Интеграция с проектом:

```
{
  "scripts": {
    "build": "tsc",
    "dev": "tsc --watch",
    "type-check": "tsc --noEmit"
  }
}
```

Работа с внешними библиотеками

> **Вопрос:** Как добавить типы для JavaScript библиотек?

DefinitelyTyped! Централизованный репозиторий типов.

```
# Установка типов для популярных библиотек
npm install --save-dev @types/node @types/express @types/react
```

Преимущества в разработке

Рефакторинг:

- Безопасное переименование переменных
- Изменение сигнатуры функции с обновлением всех вызовов
- Автоматическое исправление импортов

Автодополнение:

- IntelliSense в VS Code и других IDE
- Подсказки параметров функций
- Автоматическое импортирование типов

Отладка:

- Ошибки видны в IDE, а не только в браузере
- Статический анализ кода
- Лучшая навигация по коду

Пример: Преимущества в реальном проекте

Без TypeScript:

```
function calculatePrice(items) {
  return items.reduce((sum, item) => sum + item.price, 0);
}
calculatePrice([{ name: "Товар" }]); // NaN! - ошибка в проде
```

С TypeScript:

```
interface CartItem {
  name: string;
  price: number;
}

function calculatePrice(items: CartItem[]): number {
  return items.reduce((sum, item) => sum + item.price, 0);
}

// Ошибка видна сразу в IDE!
calculatePrice([{ name: "Товар" }]); // ❌ Property 'price' is missing
```

Практические советы

Правильные практики:

- Используйте `interface` для объектов
- Предпочитайте композицию наследованию
- Создавайте маленькие, специфичные типы
- Используйте `readonly` для immutable данных
- Не используйте `any` (лучше `unknown`)

Чего избегать:

- Слишком сложные nested типы
- Игнорирование ошибок компиляции
- Типизация ради типизации
- Избыточные аннотации типов

Результаты перехода на TypeScript

Статистика:

- На 85% меньше runtime ошибок (Airbnb, 2017)
- На 15-30% выше производительность команды (Microsoft)
- Легче onboarding новых разработчиков
- Быстрее рефакторинг без регрессий

Успешные кейсы:

- **Slack** - переписали 70% кода на TypeScript
- **Airbnb** - TypeScript стал обязательным
- **Angular** - полностью на TypeScript
- **VS Code** - TypeScript в основе IDE

Заключение

TypeScript = Надежность + Продуктивность

Преимущества:

- Меньше багов в продакшене
- Быстрее разработка благодаря автодополнению
- Легче поддержка больших проектов
- Лучше рефакторинг без страха сломать

TypeScript - это не замена JavaScript, а его улучшение!

Спасибо!

Вопросы?

Frontend Course 2025