- **Модуль:** Typescript. Необходимый уровень
- **Урок:** Про “внутренние” типы и приведение типов

TypeScript Handbook
from
Иван Петриченко
udemy

СОЗДАНИЕ ЛИТЕРАЛОВ ЧЕРЕЗ КОНСТРУКТОР

В JS есть возможность создавать сущности через **конструкторы**. Это **не самый лучший подход**, так как создание экземпляра занимает больше времени, чем простое объявление литерала. Код усложняется. И можно получать довольно странные ошибки:

```
> const x = 500;
  const y = new Number(500);

  console.log(x == y);

  console.log(x === y)
  true
  false
```

В TS тоже есть такая возможность. При этом тип переменной будет указан с большой буквы.
Так что в TypeScript существует типы, **представляющие конструкторы одноименных типов из JavaScript** (*Number, String, Boolean, Symbol, BigInt*), и также существуют типы, представляющий примитивные **значения литералов** (*number, string, boolean, symbol, bigInt*)

```
1  const num = 5; // type: 5
2  const num: number = 5; // type: number
3  const num = new Number(5); // type: Number
4  const num = Number(5); // type: number
```

В работе это тоже может дать свои особенности, так как тип number неявно преобразуется в тип Number, но **не наоборот:**

```
1  let n: number = 5;
2  let N: Number = new Number(5);
3
4  N = n; // Ok
5  n = N; // Error
```

Отсюда сделаем **небольшие выводы:**

- 👉 Не используйте конструкторы для создания значений. Это может привести к непонятным ошибкам
- 👉 На собеседованиях бывают вопросы с подвохом, где спрашивают сколько примитивных типов есть в TC. Сюда можно и относить эти типы с большой буквы, которые являются аналогами конструкторов в JS
- 👉 В продвинутой литературе вы можете встретить эти типы. Будьте внимательны и обращайте внимание на регистр первой буквы

ПРЕОБРАЗОВАНИЕ ТИПОВ

Самые банальные преобразования работают точно так же, как и в нативном JS. Они валидны за счет того, что TS четко понимает последовательность действий.

```ts
const num = 5;
const strNum: string = num.toString();

const str = "5";
const numStr: number = +str;
// и другие методы
```

С объектами все немного сложнее. Если мы говорим про трансформацию одного объекта в другой, то чаще всего мы используем трансформирующие функции. Они позволяют гибко настраивать преобразование и легко его изменять в будущем:

```ts
interface Department {
    name: string;
    budget: number;
}

const department = {
    name: "web-dev",
    budget: 50000,
};

interface Project {
    name: string;
    projectBudget: number;
}

function transformDepartment(department: Department, amount: number): Project {
    return {
        name: department.name,
        projectBudget: amount,
    };
}

const mainProject: Project = transformDepartment(department, 4000);
```

А интерфейсы позволяют нам контролировать содержание этих объектов, типизировать их содержимое