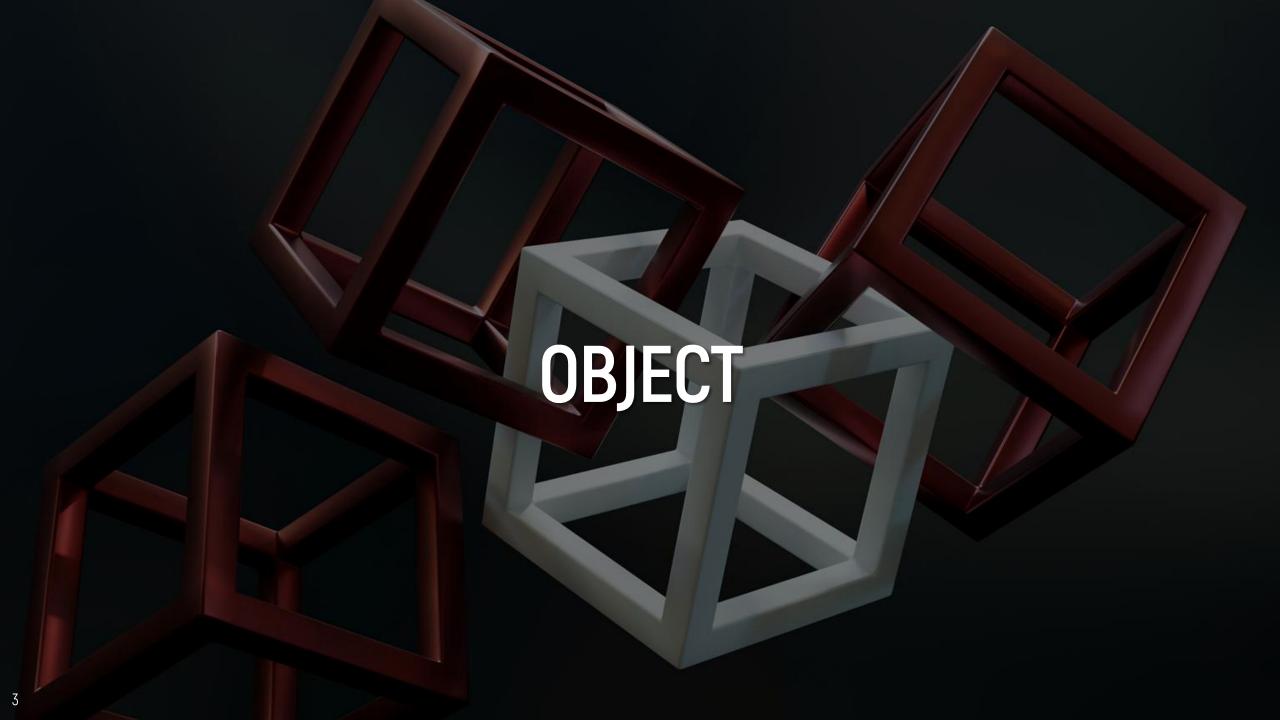
!{OBEZZZA.LAB

Объектные типы

TYPESCRIPT IN TSC

В ПРОШЛЫЙ РАЗ МЫ ПОЗНАКОМИЛИСЬ

- * Со встроенными примитивными типами TS
- ***** Константными типами и массивами
- Объединениями типов и типами функций
- * Сегодня мы будем учиться работать с объектными типами



ТИП ОВЈЕСТ

- 🗱 Означает любой объект
- 🤲 Под это подходят как функции, так и любые другие объекты
- 🗱 null и undefined (но это можно отключить опцией strictNullChecks)
- Данный тип слишком абстрактный
- * Для функций и массивов мы уже умеем описывать «конкретные» типы
- ***** Теперь осталось научится это делать для любых объектных типов

СТРУКТУРА ОБЪЕКТНОГО ТИПА

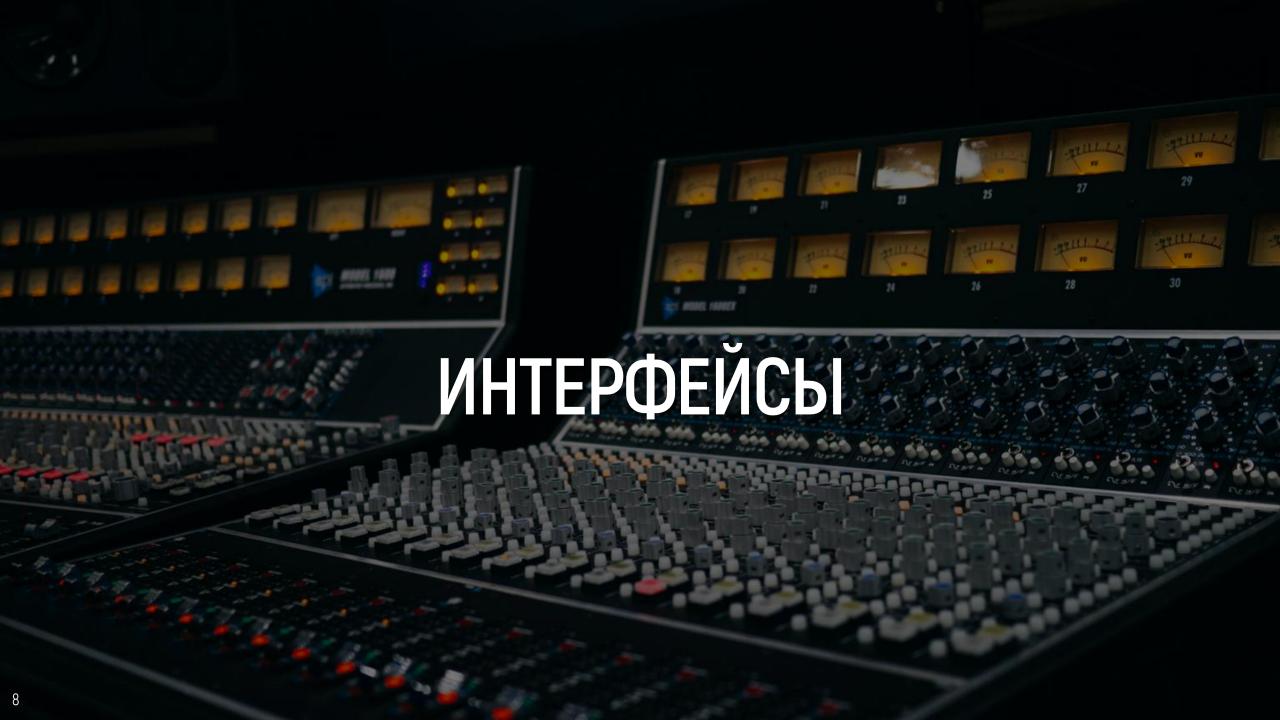
- Тип объекта описывается внутри {}
- Мы можем явно перечислить все ключи и их типы через ;
- * Мы можем добавить модификатор readonly перед именем ключа
- Или ? после имени, что означает опциональность
- **Ж** Для методов есть более короткая нотация для записи типа аргументов и возвращаемого значения

АССОЦИАТИВНЫЙ ДОСТУП К ЧАСТЯМ ТИП ОБЪЕКТА

- Мы можем сохранить объектный тип с помощью type псевдонима
- * А потом получить тип части объектного типа
- ***** Для этого используется оператор []
- **Ж** Массивы, функции это тоже объекты, поэтому к ним такое тоже применимо

СТРУКТУРА ФУНКЦИОНАЛЬНОГО ОБЪЕКТА

- ***** Функция это тоже объект в JS
- Мы можем описать функцию используя нотацию объекта
- ***** Это удобно для описания перегрузок



ИНТЕРФЕЙСЫ

- # Для описания типа объекта можно также использовать конструкцию interface
- Интерфейсы могут наследоваться от других интерфейсов
- Интерфейсы с одинаковым именем «сливаются» в один
- * Внутри интерфейса появляется новый тип this
- * Есть поддержка ассоциативного доступа (в том числе и для this)

НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ

- При описании интерфейса мы можем наследовать интерфейс одного или нескольких других интерфейсов или объектных типов
- ***** При этом такие типы находятся в отношении
- Родитель называется надтипом
- ***** Ребенок подтипом

ПОЛИМОРФИЗМ ПОДТИПОВ

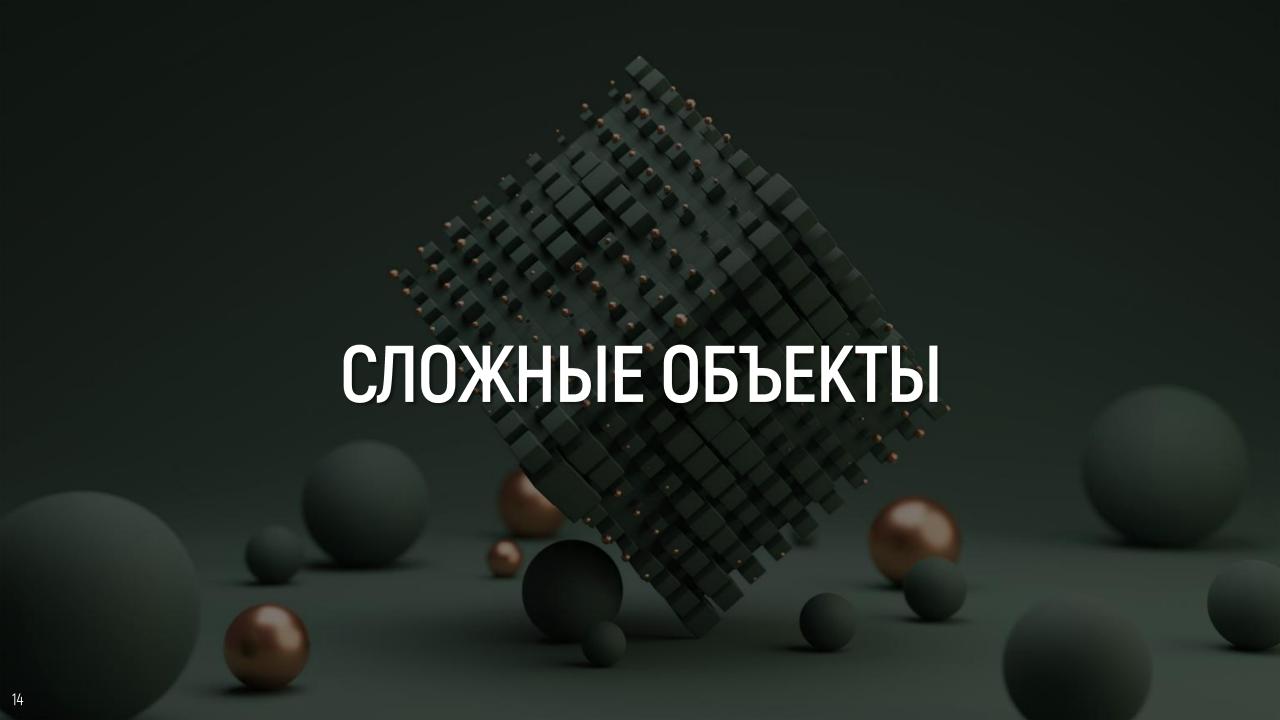
- * Интуитивно понятно, что везде где ожидается string мы можем передать тип шаблона или константы
- Наследование интерфейсов позволяет нам самостоятельно вводить такие отношения
- 🗱 В любом месте, где ожидается конкретный тип можно передать его подтип
- ***** Это называется полиморфизмом подтипов

СТРУКТУРНАЯ ТИПИЗАЦИЯ

- * В TS структурная типизация это означает, что если два типа одинаковы или могут быть приведены друг к другу, то они считаются эквивалентными
- 🗱 Мы можем использовать полиморфизм подтипов, даже если между ними нет явного отношения

ИНТЕРФЕЙСЫ VS ПСЕВДОНИМЫ

- * Может показаться, что type A = {} и interface A {} делают одно и тоже
- * Но на самом деле, их возможности пересекаются лишь частично
- * Только Interface поддерживают this, «сливание», наследование и описывают только один объект
- * С другой стороны, type поддерживает куда более могущественные вещи
- Например, объединения типов
- ☀ По мере курса мы будем видеть эту разницу



СТРУКТУРА СЛОВАРЯ

- ₩ Мы можем описать словарь задав его как {[key: тип]: тип}
- ***** Словарь может содержать и явно описанные ключи
- * Так и модификаторы readonly и ?

ПЕРЕЧИСЛЕНИЕ

- * Иногда нам нужно объявить набор констант как часть некоторого единого типа
- ₩ Именно это и позволяет сделать нам TS перечисление enum
- 🗱 Значение констант (дискриминант) в данном случае может быть либо числом, либо строкой
- 🗱 В общем случае их можно даже опустить
- * Перечисление при трансляции преобразуется в специальный объект JS (размеченное объединение)
- * Это первая возможность TS влияющая в Runtime

КОНСТАНТНОЕ ПЕРЕЧИСЛЕНИЕ

- * Если при декларации к enum добавить слово const, то такое перечисление будет существовать только на этапе написания кода и компиляции
- * B Runtime любой обращение к enum будет раскрываться в значение к константе
- ***** Это можно использовать для оптимизации размера и эффективности кода

ОБЪЕДИНЕНИЕ НЕСКОЛЬКО ENUM

- * enum из коробки не поддерживают наследование или примеси
- Однако мы можем «распечатать» несколько enum в один объект и велеть TS трактовать его как «константный» объект
- * Константный объект выводит все значения типов как константные значения
- 🗱 Чтобы сделать объект константным надо после его литерала добавить as const



МАССИВЫ И КОРТЕЖИ

- * Мы знаем как из любого типа в TS сделать массив
- * Но мы можем описать тип массива используя литеральную запись
- * В таком случае мы можем сразу задать конечное число элементов
- * И тип каждого элемента явно
- * Изменить длину такого массива или тип конкретного элемента потом нельзя
- * Такие массивы в TS называются кортежи

ИМЕНОВАННЫЕ КОРТЕЖИ

- ***** Элементам кортежа можно давать явные имена
- Это полезно для читаемости

ОПЦИОНАЛЬНЫЕ ЭЛЕМЕНТЫ

- * С помощью оператора ? можно помечать часть элементов кортежа как опциональные
- * В таком случае длина кортежа может меняться

ПЕРЕМЕННОЕ ЧИСЛО ЭЛЕМЕНТОВ

- * Если мы не знаем сколько элементов в кортеже, то мы можем задать переменное число с помощью оператора ...
- * Тип таких элементов всегда является массивом
- * Такую запись можно смешивать с обычным перечислением в любых комбинациях
- ☀ Однако, после переменного числа элементов не могут идти опциональные элементы

МОДИФИКАТОР READONLY

- # Для любого массива или кортежа можно задать модификатор readonly
- Данные модификатор делает массив неизменяемым
- * Однако рекурсивно менять элементы в глубину по прежнему можно

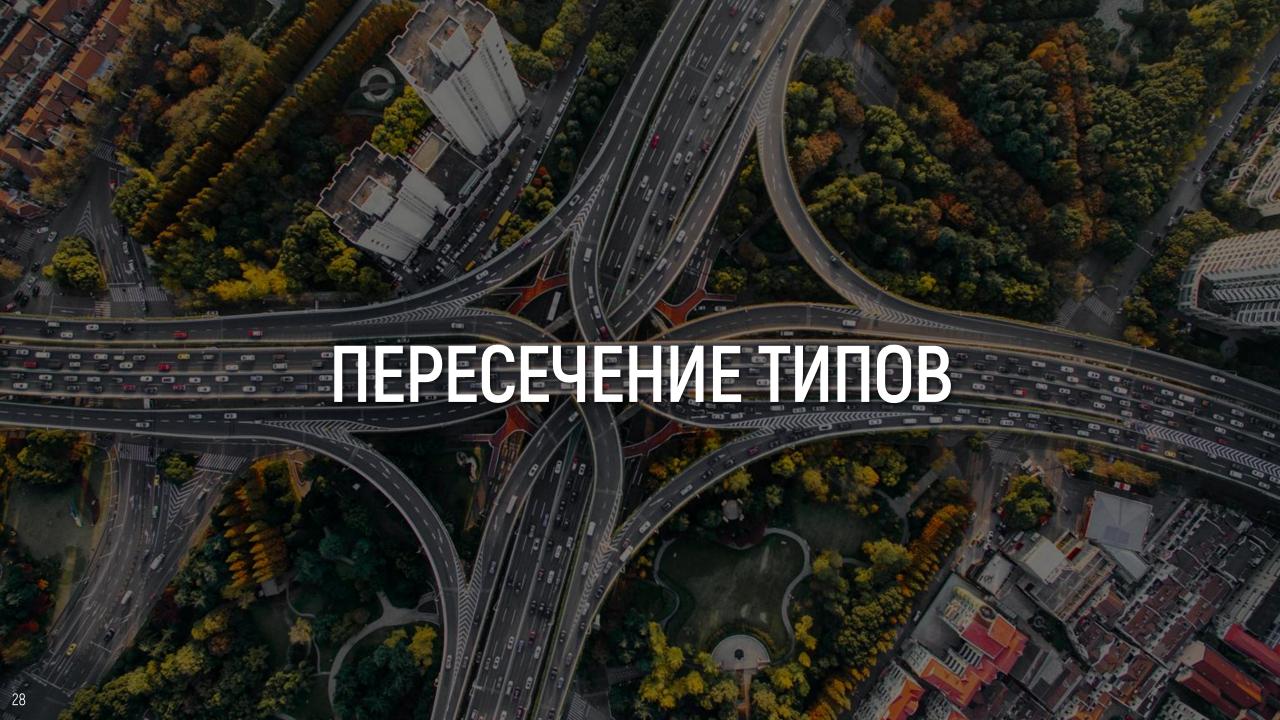


КЛАСС ОБЪЕКТА

- Классы в TS определяют и тип объекта
- При этом данный тип привязан к конструктору класса
- 🧩 Указывая имя конструктора мы ссылаемся на тип объекта, который он производит
- ☀ Допускается явно типизировать конструктор класса
- **Ж** Для этого можно использовать тип функции с модификатором new
- **Ж** Или функциональный объект с модификатором new

КЛАССЫ ВСТРОЕННЫХ ОБЪЕКТОВ

- * TS типизирует все встроенные классы JavaScript
- Для использования достаточно указать имя конструктора
- * Если некоторый класс появился в определенном окружении, то его может понадобиться подключить в .tsconfig
- * Конструктор Function можно трактовать как «любая функция»
- * Object использовать не надо используйте object



ПЕРЕСЕЧЕНИЕ ТИПОВ

- * Любые два и более типов или интерфейсов можно объединить в один
- Для этого следует использовать оператор &
- ***** Это не тоже самое, что и наследование интерфейсов
- 🗱 Все дело в том, как обрабатываются конфликты

NEVER

- * Специальный тип never означает пустое множество
- 🗱 Ни один тип не соответствует ему
- Пересечение разных примитивных типов даёт never
- ***** Его можно использовать явно для удаление свойств объекта
- 🗱 Или чтобы показать, что функция никогда не возвращает результат

ПОДВЕДЕМ ИТОГИ



ПОДВЕДЕМ ИТОГИ

- * Кроме типа object TS вводит множество других способов описать любой объект более конкретно
- Объектная нотация позволяет описать любую структуру или словарь
- ***** Поддерживается ассоциативный доступ к свойству любого объектного типа
- * TS поддерживает специальный тип перечислений, которые существуют в Runtime
- **Ж** Для фиксированных гомогенных массивов есть тип кортежей
- * Классы в TS описывают тип экземпляра
- * С помощью пересечения типов мы можем объединять два и более типов в один
- ***** Встроенный тип never означает пустое множество

