**TS**

**ts\_1.1 Что такое TypeScript?**

«Типы — это множества значений и то, что с ними можно сделать

«TypeScript — это надмножество JavaScript со статической типизацией» или «TypeScript — это типизированное расширение JavaScript». TS – это язык программирования.

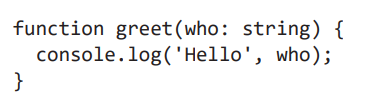
В TS добавляется система типов, и запрещаются некоторые конструкции JavaScript, например вызов функций с неправильным количеством аргументов.

TypeScript - не выполняется в интерпретаторе (как Ruby или Python) и не компилируется в низкоуровневый язык (как Java или C). Программа транспилируется в другой высокоуровневый язык — JavaScript. Поэтому выполняется именно JavaScript, а не TypeScript.

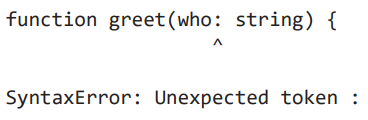
На высоком уровне tsc (TS-компилятор) делает две вещи: преобразует новейшую версию TypeScript/JavaScript в более старую версию JavaScript, которая работает во всех браузерах (транспиляция); проверяет код на наличие ошибок типов.

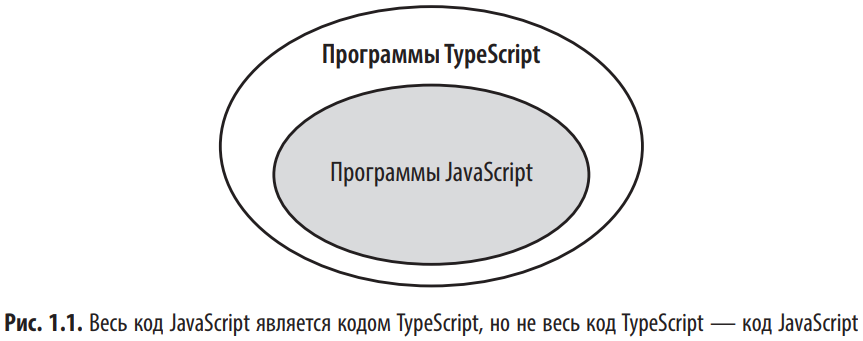
Все программы JS являются программами TS, но обратное справедливо не всегда: есть программы TS, которые не являются программами JS, из-за того, что в TypeScript добавлен дополнительный синтаксис определения типов.

Например, рабочая TS прога, не запустится через среду выполнения JavaScript:



Получится ошибка, т.к.: string является аннотацией типа, относящейся к TS, которая выходит за рамки базового JavaScript:





**ts\_1.2 Для чего он нужен и какие проблемы решает?**

- выявляет ошибок на этапе компиляции, а не на этапе выполнения.

- за счет системы типов дает такие возможности, как подсказки и переходы по коду, которые значительно ускоряют процесс разработки.

- система типов избавляет от комментирования кода, которое отнимает время.

- сокращает время на устранение ошибок и выявление багов, которые иногда сложно определить при динамической типизации JavaScript.

- позволяет писать более понятный и читаемый код, максимально описывающий предметную область.

— всё это в совокупности сокращает время разработки программы

**ts\_2 Рассказать про следующие виды типизации, их особенности, плюсы и минусы:**

**ts\_2.1 слабая/сильная**

(также иногда говорят строгая / нестрогая). Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

Сильная: Java, Python, Haskell, Lisp;

Слабая: C, JavaScript, Visual Basic, PHP.

**ts\_2.2 статическая/динамическая**

Статическая определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип, где находится. В динамической типизации все типы выясняются уже во время выполнения программы.

Главное, что отличает статическую (static) типизацию от динамической (dynamic) то, что все проверки типов выполняются на этапе компиляции, а не этапе выполнения.

Статическая: C, Java, C#;

Динамическая: Python, JavaScript, Ruby.

Преимущества статической типизации

-Проверки типов происходят только один раз — на этапе компиляции. А это значит, что не нужно постоянно выяснять, не пытаемся ли мы поделить число на строку (и либо выдать ошибку, либо осуществить преобразование).

-Скорость выполнения. Из предыдущего пункта ясно, что статически типизированные языки практически всегда быстрее динамически типизированных.

-При некоторых дополнительных условиях, позволяет обнаруживать потенциальные ошибки уже на этапе компиляции.

-Ускорение разработки при поддержке IDE (отсеивание вариантов, заведомо не подходящих по типу).

Преимущества динамической типизации

- Простота создания универсальных коллекций — куч всего и вся (редко возникает такая необходимость, но когда возникает динамическая типизация выручит).

- Удобство описания обобщенных алгоритмов (например сортировка массива, которая будет работать не только на списке целых чисел, но и на списке вещественных и даже на списке строк).

- Легкость в освоении — языки с динамической типизацией обычно очень хороши для того, чтобы начать программировать.

**ts\_2.3 явная/неявная**

Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

Явная: C++, D, C#

Неявная: PHP, Lua, JavaScript

**ts\_2.4 структурная/номинативная;**

**ts\_2.5 бестиповая типизация**

В бестиповых языках программирования — все сущности считаются просто последовательностями бит, различной длины.

Бестиповая типизация обычно присуща низкоуровневым (язык ассемблера, Forth) и эзотерическим (Brainfuck, HQ9, Piet) языкам. Однако и у нее, наряду с недостатками, есть некоторые преимущества.

Преимущества

-Позволяет писать на предельно низком уровне, причем компилятор / интерпретатор не будет мешать какими-либо проверками типов. Вы вольны производить любые операции над любыми видами данных.

-Получаемый код обычно более эффективен.

-Прозрачность инструкций. При знании языка обычно нет сомнений, что из себя представляет тот или иной код.

Недостатки

-Сложность. Часто возникает необходимость в представлении комплексных значений, таких как списки, строки или структуры. С этим могут возникнуть неудобства.

-Отсутствие проверок. Любые бессмысленные действия, например вычитание указателя на массив из символа будут считаться совершенно нормальными, что чревато трудноуловимыми ошибками.

-Низкий уровень абстракции. Работа с любым сложным типом данных ничем не отличается от работы с числами, что, конечно, будет создавать много трудностей.

**ts\_3. Какие виды типизации используются в JavaScript и TypeScript?**

**ts\_4. Как и для каких целей используются типы данных, приведенные ниже?**

**ts\_4.1 Boolean**

**ts\_4.2 Number**

**ts\_4.3 String**

**ts\_4.4 Null and Undefined**

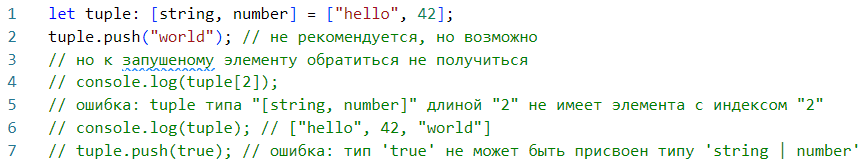
**ts\_4.5 Object and object**

**ts\_4.6 Array**

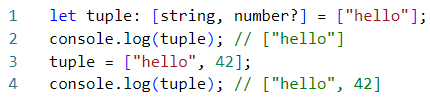
**ts\_4.7 Tuple**

Кортежи (tuples) представляют собой особый вид массивов, где порядок элементов зафиксирован, и каждый элемент имеет заранее определенный тип.

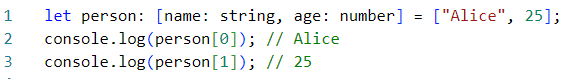


Кортежи, как и массивы, поддерживают методы push, pop, shift, unshift и другие и они работают, но так делать не рекомендуется, потому что это ломает типизацию.  


Кортежи могут содержать необязательные элементы, которые указываются с помощью знака вопроса (?).



Можно использовать именованные элементы в кортежах для улучшения читаемости кода.



Используя tuple и spread оператор можно типизировать массивы произвольной длины.



**ts\_4.8 Enum**

**Я понимаю, что 3 года уже прошло, но всё равно оставлю это здесь.**

**Из данного ролика не понял, где в enum ключ, где значение, а где индекс. Полез в доки по TS. Выяснил, что ключи (key), они же константы - это значения СЛЕВА от знака присвоения, а сопряжённые с ключами значения (value) - СПРАВА от знака присвоения.**

**Enum по умолчанию задаётся числовым, где значение (value) первого элемента (ключа) - нуль, а далее, итерируя по ключам, их значения увеличиваются на один от ПРЕДЫДУЩЕГО значения (value), если ключу не присвоено иное.**

**Т.е. тут НЕТ никаких индексов, а есть пары ключ - значение. И по КЛЮЧУ Up автор тут находил ЗНАЧЕНИЕ (value) - 0, по Down - 1 и т.д. И наоборот.**

**Как пример можно загуглить "Get an Enum Key by Value in TypeScript". Там всё чёрным по белому объяснено:**

**enum Sizes {**

**Small = 'S',**

**Medium = 'M',**

**Large = 'L',**

**}**

**console.log(Object.keys(Sizes)); // ['Small', 'Medium', 'Large'] - вот ключи.**

**console.log(Object.values(Sizes)); // ['S', 'M', 'L'] - а вот значения, НЕ ИНДЕКСЫ.**

**в Enum нет индексов. Ни в одном примере не обращаются по индексу. К Enum обращаются по ключу или значению.**

**Плюсую. В enum нет индексов. Заблуждение идёт от того что поведение enum похоже на массив. Когда мы вызываем console.log(enum[0]), выглядит как будто мы вызываем массив по индексу, хотя 0 - это значение (value). Просто enum по умолчанию задаёт ключу числовое значение в порядке от 0. А в массиве наоборот нет понятия "ключ". Ключ в массиве - это и есть индекс, просто его не принято называть ключом.**

**ts\_4.9 Any**

**ts\_4.10 Void**

**ts\_4.1 Never**

**ts\_4.12 Unknown**

**ts\_4.13 Union**

**Как и для каких целей используются типы данных**

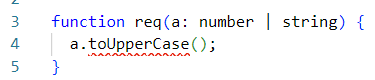
Тип данных, который позволяет объединять несколько типов в один. Обозначается символом «|». Это позволяет переменным или параметрам функций принимать значения разных типов.

Удобно использовать с данными, получаемыми из внешних источников, таких как API, базы данных или пользовательский ввод.

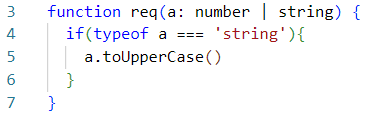
let myVar: number | string;

**Narrowing**Сужение типа (Narrowing) - это процесс уточнения типов переменных.

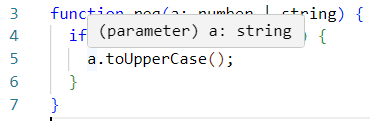
При использовании union методы того или иного типа недоступны в смешанном типе, так методы строки например не будут существовать в типе number | string. Будет ошибка Property 'toUpperCase' does not exist on type 'string | number'.Property 'toUpperCase' does not exist on type 'number'.



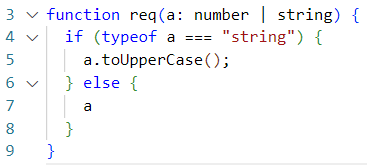
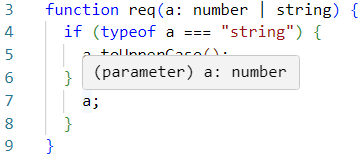
Чтобы использовать методы необходимо сделать проверку, и установить ограничение типа:



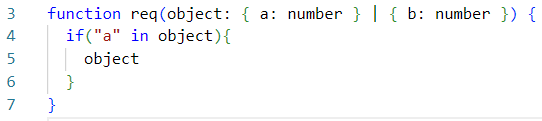
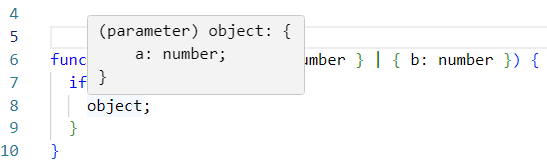
Тогда тс врубается, что в переменной может быть только строка и даже подсвечивает ее тип:



Более того, т.с. поймет какой тип будет лежать в переменной при последующих проверках:

Объекты можно проверять по наличию ключа оператором in:

**ts\_5. Что такое вывод типов (относительно любого языка)?**

Вывод типов (type inference) в программировании — это процесс, при котором компилятор или интерпретатор автоматически определяет тип данных переменной, функции или выражения, основываясь на контексте использования, без явного указания типа программистом.

В TypeScript, при объявлении переменной let x = "hello";, компилятор выводит, что x имеет тип string.

**ts\_6. В каких случаях TS автоматически выводит переменную типа как any?**

**ts\_7. Для чего используются type assertions? Почему стоит их избегать?**

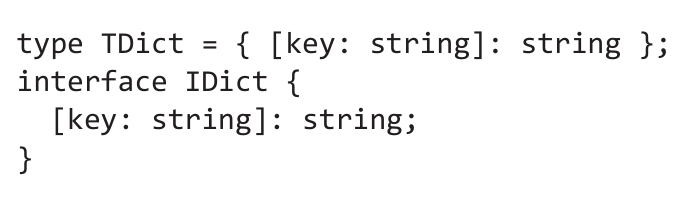
**ts\_8. Для чего предназначены интерфейсы?**

**ts\_9. Что такое Type Aliases? Зачем нужны? Чем отличаются от интерфейсов?**

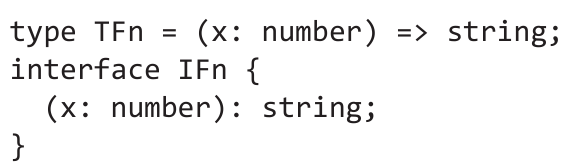
Что общего?

- проверка лишних свойств в литералах объектов

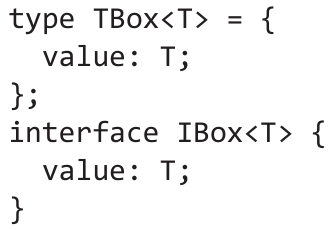
- может использоваться сигнатура индекса



- можно определять типы функций



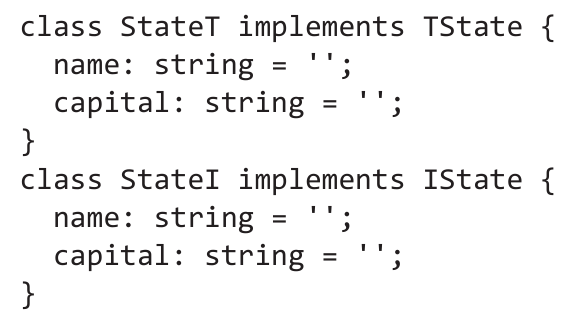
- могут быть обобщенными



- могут расширять друг друга, с некоторыми оговорками

interface может расширять только объектные типы, которые могли бы быть определены с interface. Например, не допускается расширение типов объединений. Если вам нужно именно это, используйте type и &.

- могут реализовывать классы

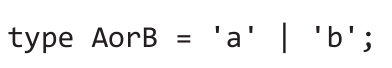


- могут быть рекурсивными

Отличия?

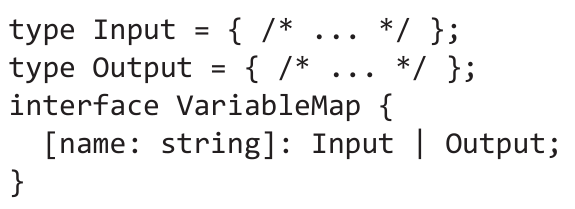
В целом type предоставляет больше возможностей, чем interface. Он может выступать в качестве объедине ния, а также пользоваться более продвинутыми возможностями вроде отобра женных типов (правило 15) или условных типов (правило 52).

- в типах могут содержаться объединения, в интерфейсах нет



- интерфейс не может расширять типы объединений. (Некоторые типы интерфейс может расширить).

Если у вас есть раздельные типы для переменных Input и Output и отображения из имени в переменную:

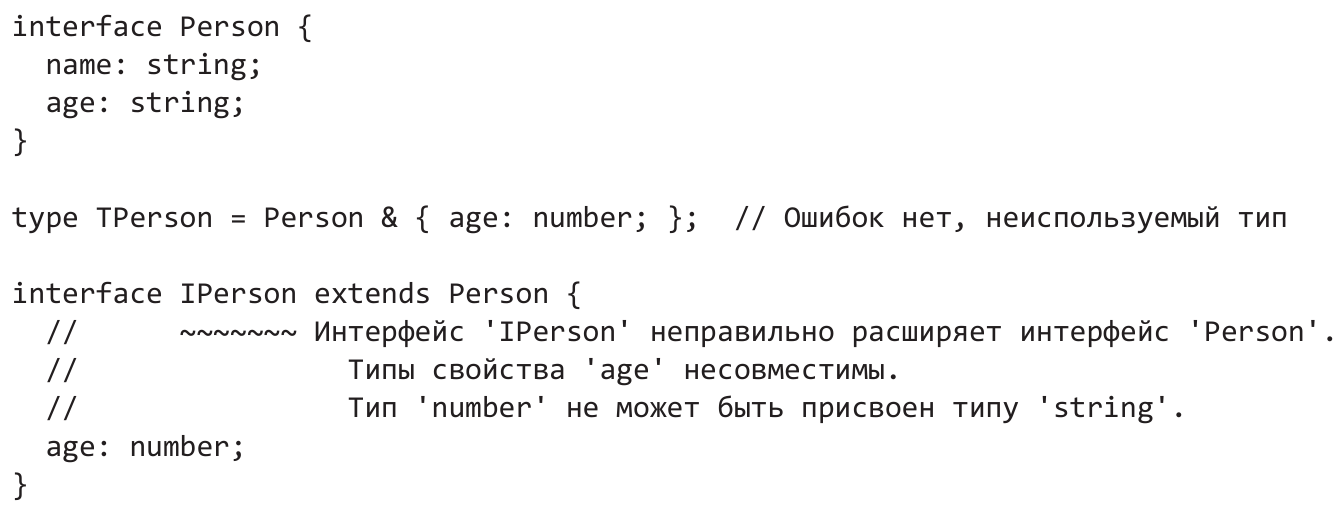


может понадобиться тип, который связывает имя с переменной. Например:



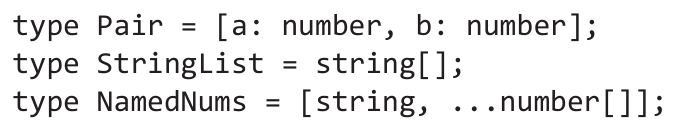
Этот тип не может быть выражен через interface.

- interface и extends обеспечивают чуть более высокий уровень проверки ошибок, чем type и &:

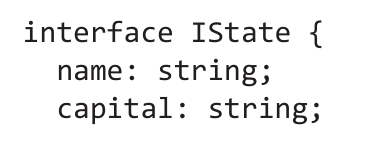


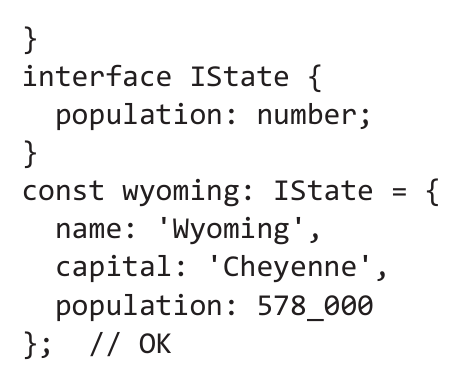
Изменение типа свойства в подтипе допустимо при условии, что он совместим с базовым типом (см. правило 7). Желательно иметь больше проверок безопас ности, так что это веская причина для использования extends c interface.

- Псевдонимы типов позволяют естественно выражать типы кортежей и массивов(а интерфейсы нет чтоли?):



- interface может быть дополнен, а type нет – этот синтаксис называется «слиянием объявлений» (declaration merging). Это может происходить в пользовательском коде, но только если два интерфейса определяются в одном модуле (то есть в одном файле .ts). Тем самым предотвращаются случайные конфликты с глобальными интерфейсами, имеющими такие общие имена, как Location и FormData.



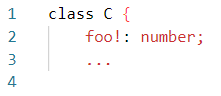


**ts\_10. Что делает ? после имени свойства интерфейса или аргумента функции?**

**ts\_10.1 В чем различие между x?: number и x: number | undefined?**

**ts\_11. Зачем нужен постфикс !? Почему его стоит избегать?**

**ts\_11.1 Для чего используется с полями классов? Например:**



**ts\_12. От чего могут наследоваться интерфейсы? Какие накладывает ограничения наследование от класса?**

**ts\_13. Какие есть способы типизации static-side класса и instance-side класса?**

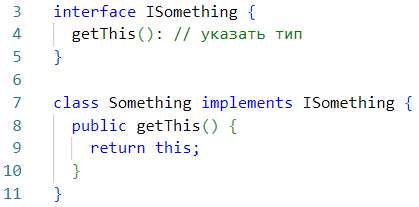
**ts\_14. Может ли класс одновременно реализовывать набор интерфейсов и наследоваться от другого класса?**

**ts\_15. Каковы назначения модификаторов свойств классов?**

**ts\_15.1. В чем отличие private от #?**

**ts\_16. В чем отличие абстрактных классов от обычных классов и интерфейсов?**

**ts\_17. Можно ли объявить в интерфейсе тип метода таким образом, чтобы он возвращал контекст своего вызова?**



**ts\_18. Для чего используются Hybrid Types?**

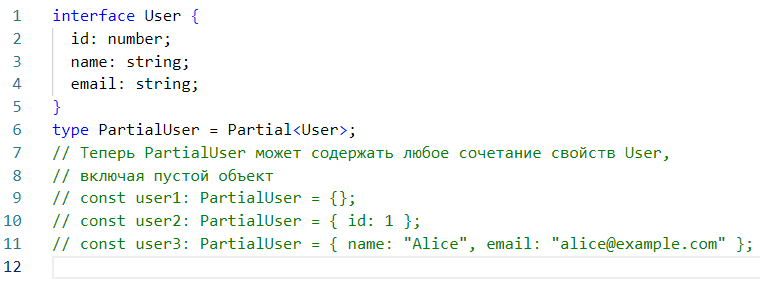
**ts\_19. Для чего используются Index types?**

**ts\_19.1 Для чего нужен оператор keyof?**

**ts\_n.** Для чего нужны generics?

**ts\_n.1 Какие есть встроенные обобщенные интерфейсы (built-in generic types) или по другому называются типы утилиты?**

- Partial<T> - используется для создания нового типа на основе существующего, при этом делая все его свойства необязательными.

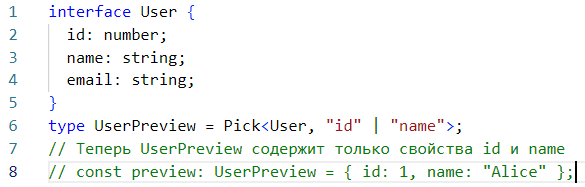


Применение «Partial»

- Создание черновых объектов: часто используется при работе с формами или данными, поступающими из разных источников, где часть данных может быть неизвестной или необязательной.

- Обновление объектов: дает легко описывать объекты для обновления, когда только некоторые поля могут быть изменены.

- Pick<Type, Keys> - создает новый тип, в котором перечисляются только те свойства из Type, которые указаны в Keys (строковый литерал или объединение строковых литералов).



Применение «Pick»

- Создание облегчённых типов: используется для создания типов, содержащих только необходимые свойства, что помогает снизить избыточность данных и уменьшить объем передаваемой информации.

- Типизация ответов API: полезен для типизации данных, возвращаемых API, когда только часть данных объекта нужна для конкретного запроса.