**TS**

**ts\_1.1 Что такое TypeScript?**

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

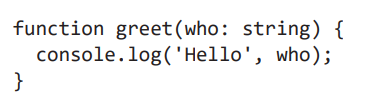
В TS добавляется система типов, и запрещаются некоторые сомнительные конструкции JavaScript, например вызов функций с неправильным количеством аргументов.

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.

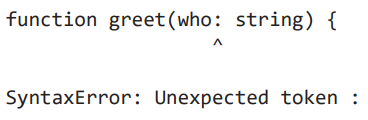
На высоком уровне tsc (TS-компилятор) делает две вещи: преобразует новейшую версию TypeScript/JavaScript в более старую версию JavaScript, которая работает во всех браузерах (транспиляция); проверяет код на наличие ошибок типов.

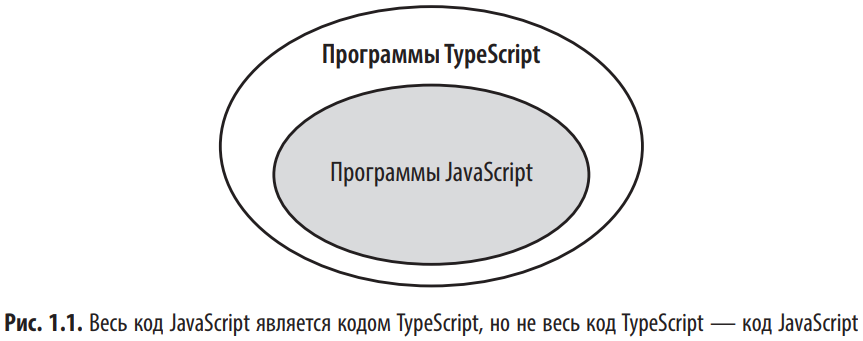
Все программы JS являются программами TS, но обратное справедливо не всегда: есть программы TS, которые не являются программами JS, из-за того, что в TypeScript добавлен дополнительный синтаксис определения типов.

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.

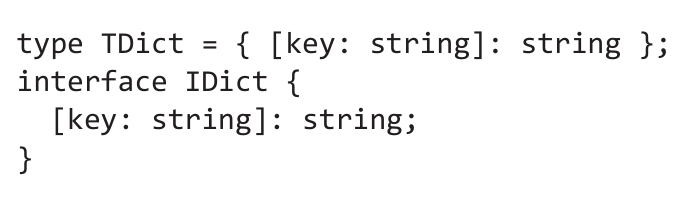
«Типы — это множества значений и то, что с ними можно сделать

**Что такое Type Aliases? Зачем нужны? Чем отличаются от интерфейсов?**

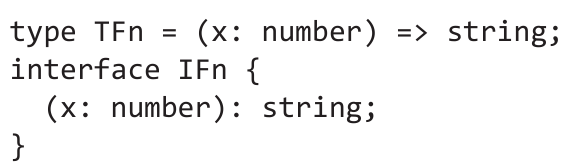
Что общего?

- проверка лишних свойств в литералах объектов

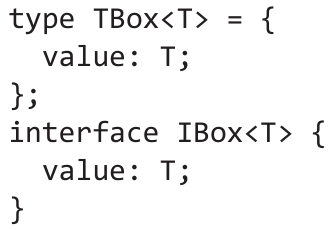
- может использоваться сигнатура индекса



- можно определять типы функций



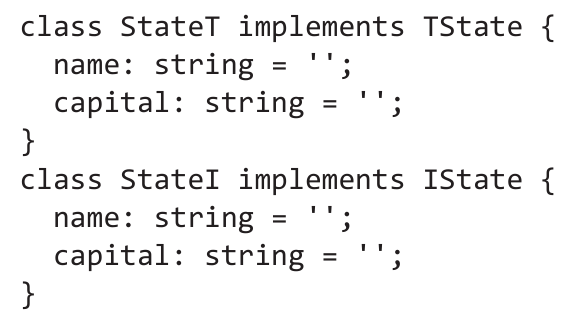
- могут быть обобщенными



- 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

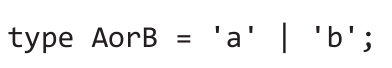


- могут быть рекурсивными

Отличия?

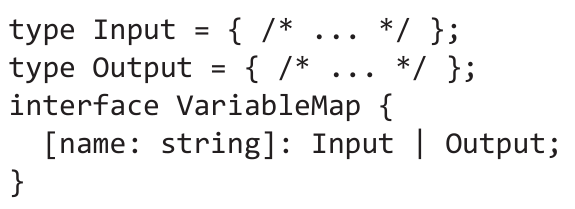
В целом type предоставляет больше возможностей, чем interface. Он может выступать в качестве объедине ния, а также пользоваться более продвинутыми возможностями вроде отобра женных типов (правило 15) или условных типов (правило 52).

- 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:

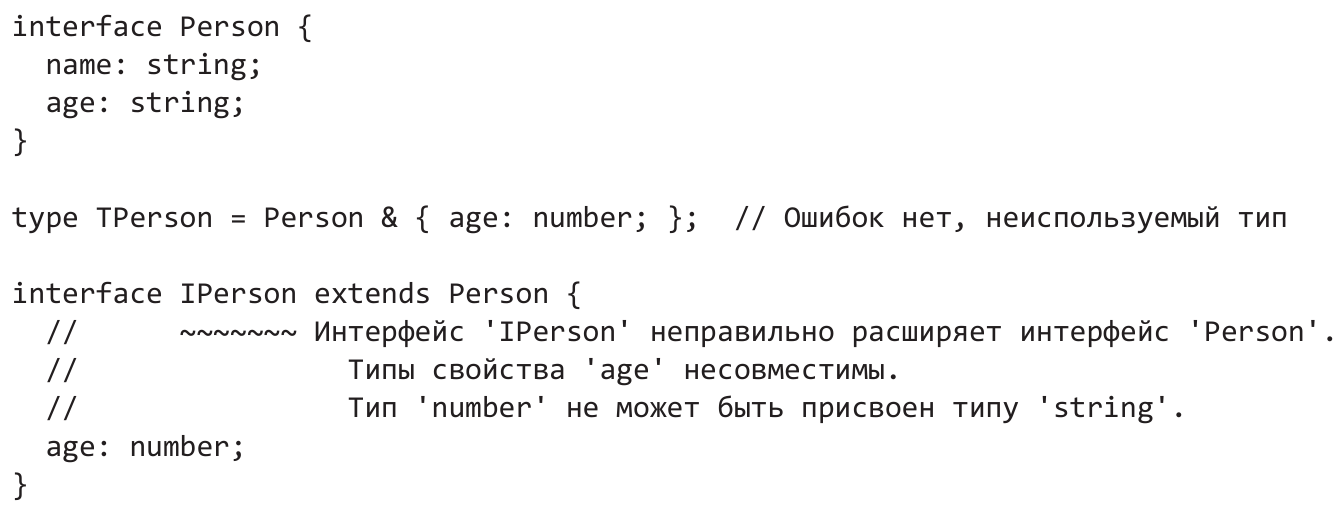


может понадобиться тип, который связывает имя с переменной. Например:



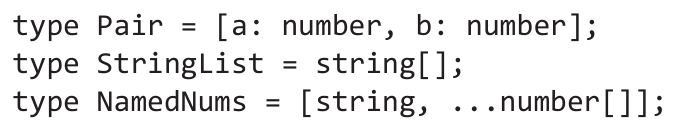
Этот тип не может быть выражен через interface.

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

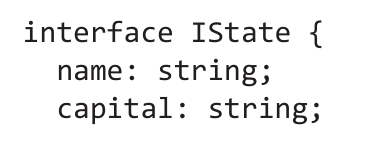


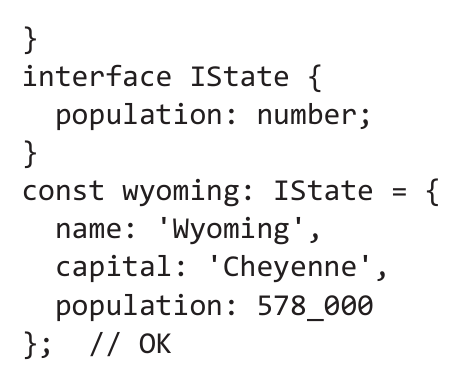
Изменение типа свойства в подтипе допустимо при условии, что он совместим с базовым типом (см. правило 7). Желательно иметь больше проверок безопас ности, так что это веская причина для использования extends c interface.

- 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.





Как и для каких целей используются типы данных, приведенные ниже?

* Any

Используйте any, если действительно хотите отключить проверку типа,

* Unknown

Используйте unknown, если вам нужно соблюдать осторожность.