**1.Введение в TypeScript**

**1.1.Что такое TypeScript**

TypeScript представляет язык программирования на основе JavaScript.

Развитие TypeScript началось в конце 2012 года. Хотя он зародился в компании Microsoft, и его фактическим создателем является программист Андерс Хейлсберг, так же известный как создатель таких языков как Delphi, C#, но данный проект сразу стал развиваться как OpenSource. И уже с самого начала новый язык стал быстро распространяться в силу своей гибкости и производительности. Немало проектов, которые были написаны на JavaScript, стали переноситься на TypeScript. Популярность и актуальность идей нового языка привела к тому, что ряд из этих идей в последующем стали частью нового стандарта JavaScript. Преимущества TypeScript были подхвачены создателями ряда распространенных и широкоиспользуемых фреймворков. К слову, одни из наиболее популярнейших фреймворков для Web - Angular 2+ и Vue3 полностью написаны на TypeScript.

Однако, казалось бы, зачем нужен еще один язык программирования для клиентской стороны в среде Web, если со всей той же самой работой прекрасно справляется и традиционный JavaScript, который используется практически на каждом сайте, которым владеет множество разработчиков и поддержка которого в сообществе программистов довольно высока. Но TypeScript это не просто новый JavaScript.

Во-первых, следует отметить, что TypeScript - это строго типизированный и компилируемый язык, чем, возможно, будет ближе к программистам Java, C# и других строго типизированных языков. Хотя на выходе компилятор создает все тот же JavaScript, который затем исполняется браузером. Однако строгая типизация уменьшает количество потенциальных ошибок, которые могли бы возникнуть при разработке на JavaScript.

Во-вторых, TypeScript реализует многие концепции, которые свойствены объектно-ориентированным языкам, как, например, наследование, полиморфизм, инкапсуляция и модификаторы доступа и так далее.

В-третьих, потенциал TypeScript позволяет быстрее и проще писать большие сложные комплексные программы, соответственно их легче поддерживать, развивать, масштабировать и тестировать, чем на стандартном JavaScript.

В-четвертых, TypeScript развивается как opensource-проект и, как и многие проекты, хостится на гитхабе. Адрес репозитория - <https://github.com/Microsoft/TypeScript>. Кроме того, он является кроссплатформенным, а это значит, что для разработки мы можем испольвать как Windows, так и MasOS или Linux.

В то же время TypeScript является типизированным надмножеством JavaScript, а это значит, что любая программа на JS является программой на TypeScript. В TS можно использовать все те конструкции, которые применяются в JS - те же операторы, условные, циклические конструкции. Более того код на TS компилируется в javascript. В конечном счете, TS - это всего лишь инструмент, который призван облегчить разработку приложений.

Генерируемый компилятором TypeScript код JS поддерживается подавляющим большинством браузеров. Хотя в процессе разработки мы можем сами задать целевой стандарт ECMAScript.

Как использовать TypeScript? Поскольку данный язык является OpenSource, то все его инструменты доступны для всех желающих. Для работы с TypeScript мы можем использовать как Windows, так и Linux и MacOS.

Сам компилятор TS можно установить с помощью команды менеджера пакетов npm, который используется в Node.js:

> npm install -g typescript

Для написания кода на языке TypeScript можно использовать любой самый простейший текстовый редактор. Многие текстовые редакторы и среды разработки, например, Visual Code Studio, Atom, Sublime, Visual Studio, Netbeans, WebStorm и другие, имеют поддержку TypeScript на уровне плагинов, что позволяет воспользоваться рядом преимуществом, например, подцветкой кода или всплывающей подсказкой по типам и конструкциям языка.

## 1.2.Установка и компиляция из командной строки

Чтобы начать работать с TypeScript, установим необходимый инструментарий. Установить TypeScript можно двумя способами: через пакетный менеджер NPM или как плагин к Visual Studio

### Установка через NPM

Для установки через NPM вначале естественно необходимо установить Node.js (если он ранее не был установлен). После установки Node.js необходимо запустить следующую команду в командной строке (Windows) или терминале (Linux):

npm install -g typescript

При установке на MacOS также требуется ввести команду **sudo**. При вводе данной команды терминал запросит логин и пароль пользователя для установки пакета:

sudo npm install -g typescript

Вполне возможно, что ранее уже был установлен TS. В этом случае его можно обновить до последней версии с помощью команды

npm update -g typescript

Для проверки версии необходимо ввести команду

tsc -v

### Компиляция приложения

Для начала создадим каталог приложения. В моем случае это будут папка по пути **C:/typescript**. В каталог добавим файл **index.html**. Откроем этот файл в любом текстовом редакторе и определим в нем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <h2 id="header"></h2>      <script src="app.js"></script>  </body>  </html> |

Это обычный файл html, в котором определен пустой заголовок - элемент <h2> - в него мы будем выводить некоторое содержимое. И также на веб-странице подключается файл **app.js**.

Теперь в том же каталоге создадим файл **app.ts**. Причем именно **app.ts**, а не app.js, то есть файл кода typescrypt. Это также обычный текстовый файл, который имеет расширение **.ts**. И в **app.ts** определим следующее содержание:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class User{      name : string;      constructor(\_name:string){            this.name = \_name;      }  }  const tom : User = new User("Том");  const header = this.document.getElementById("header");  header.innerHTML = "Привет " + tom.name; |

Что здесь происходит? Сначала определяет класс User - шаблон данных, которые будут использоваться на веб-странице. Этот класс имеет поле name, которое представляет тип string, то есть строку. Для передачи данных этому полю определен специальный метод - constructor, который принимает через параметр \_name некоторую строку и передает ее в поле name:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class User{      name : string;      constructor(\_name:string){          this.name = \_name;      }  } |

Далее мы подробнее разберем создание и использование классов. Далее создаем константу tom, которая представляет этот класс:

|  |  |
| --- | --- |
| 1 | const tom : User = new User("Том"); |

Иначе говоря, константа tom представляет некоторого пользователя, у которого имя "Том".

Затем получаем элемент с id **header** на веб-странице в одноименную константу header:

|  |  |
| --- | --- |
| 1 | const header = this.document.getElementById("header"); |

То есть этот элемент будет представлять определенный на веб-странице index.html заголовок <h2>.

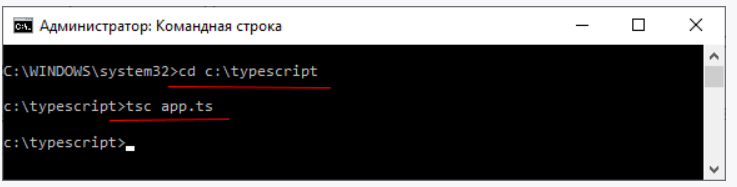
Далее с помощью свойства innerHTML меняем содержимое элемента:

|  |  |
| --- | --- |
| 1 | header.innerHTML = "Привет " + tom.name; |

В данном случае свойству innerHTML передается строку, к которой добавляется значение свойства tom.name. То есть мы ожидаем, что в заголовок на веб-странице будет выводится создаваемая здесь строка.

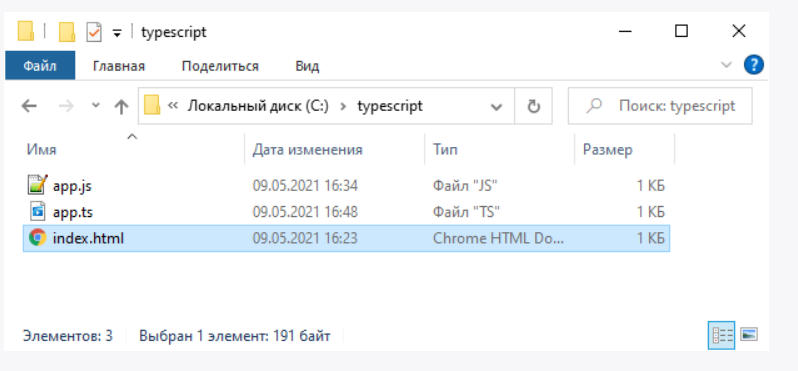
Сохраним и скомпилируем этот файл. Для этого в командной строке/терминале с помощью команды cd перейдем к каталогу, где расположен файл **app.ts** (в моем случае это C:\typescript). И для компиляции выполним следующую команду:

tsc app.ts

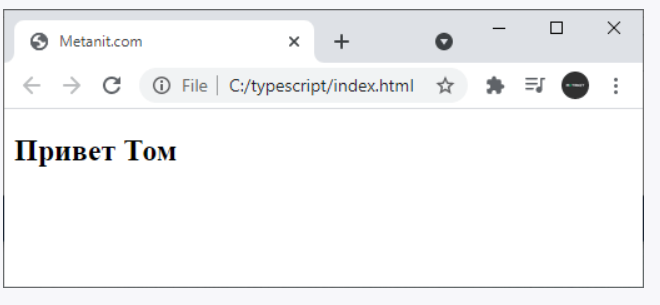


После компиляции в каталоге проекта создается файл **app.js**, который будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var User = /\*\* @class \*/ (function () {      function User(\_name) {          this.name = \_name;      }      return User;  }());  var tom = new User("Том");  var header = this.document.getElementById("header");  header.innerHTML = "Привет " + tom.name; |

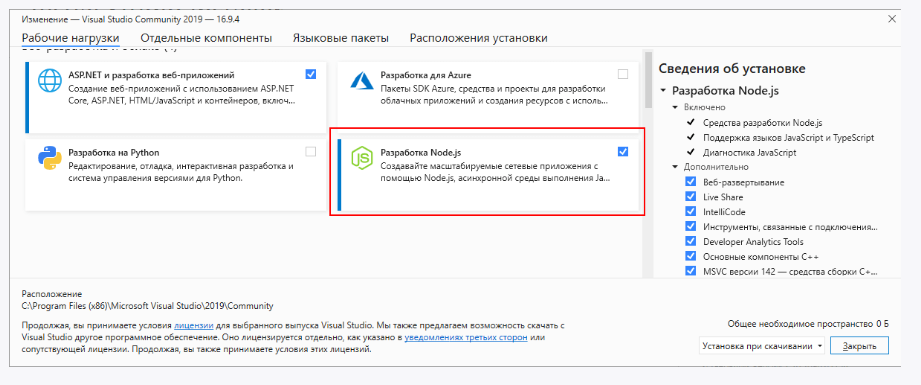


И теперь мы можем кинуть веб-страницу **index.html** в браузер и увидеть результат работы написанного на TypeScript кода:

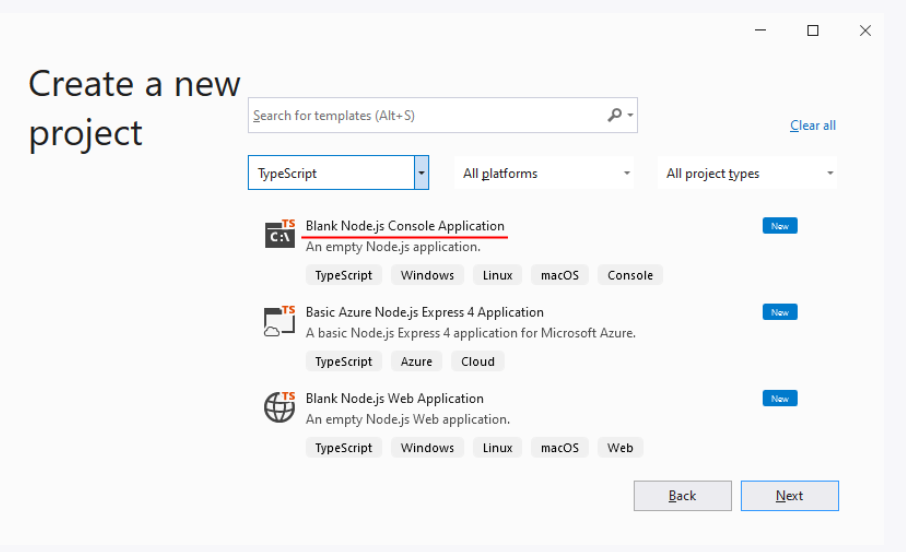


## 1.3.Первое приложение на TypeScript в Visual Studio

Среда разработки Visual Studio имеет встроенные возможности по созданию проектов с использованием TypeScript. Итак, создадим первое приложение на TypeScript в Visual Studio. Прежде всего нам необходимо с помощью установщика Visual Studio установить соответствующий компонент. Данный компонент называется **Разработка Node.js**:

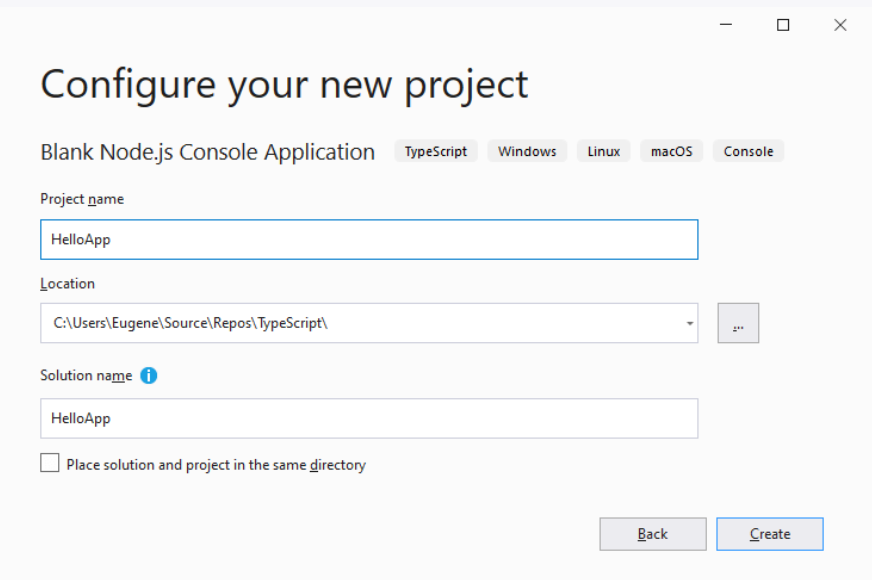


При создании нового проекта в поле языка мы можем выбрать **TypeScript** и увидеть доступные по умолчанию несколько шаблонов проектов, которые предполагают использование языка TypeScript:

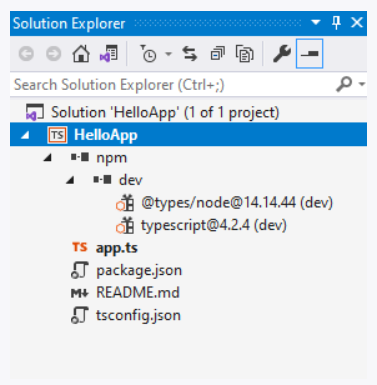


Эти проекты позволяют использовать TypeScript в различных ситуациях - в связке с Node.js, Vue и т.д. В данном случае выберем самый простой тип проекта, который можно использовать для изучения TS - **Blank Node.js Console Application**.

После выбора данного типа проекта дадим ему какое-нибудь имя, например, **HelloApp** и укажем ему расположение:



В итоге Visual Studio создаст следующий проект:



Проект по умолчанию имеет следующие файлы:

* **app.ts**: файл с кодом на TypeScript
* **package.json**: файл с определением пакетов и прочей конфигурации для Node.js
* **tsconfig.json**: файл конфигурации TypeScript

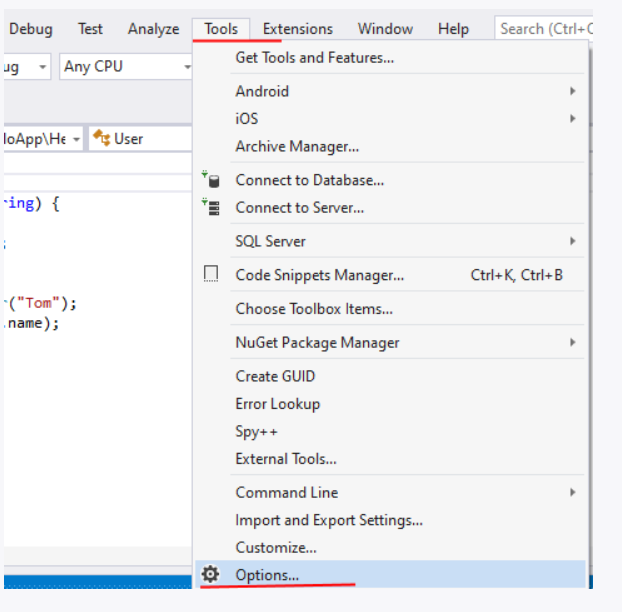
Кроме того, в узле **npm/dev** можно увидеть применяемые для разработки пакеты npm - **@types/node** и **typescript**.

Ключевым элементом здесь является файл **app.ts**, который выполняется при запуске приложения. Изменим его код следующим образом:

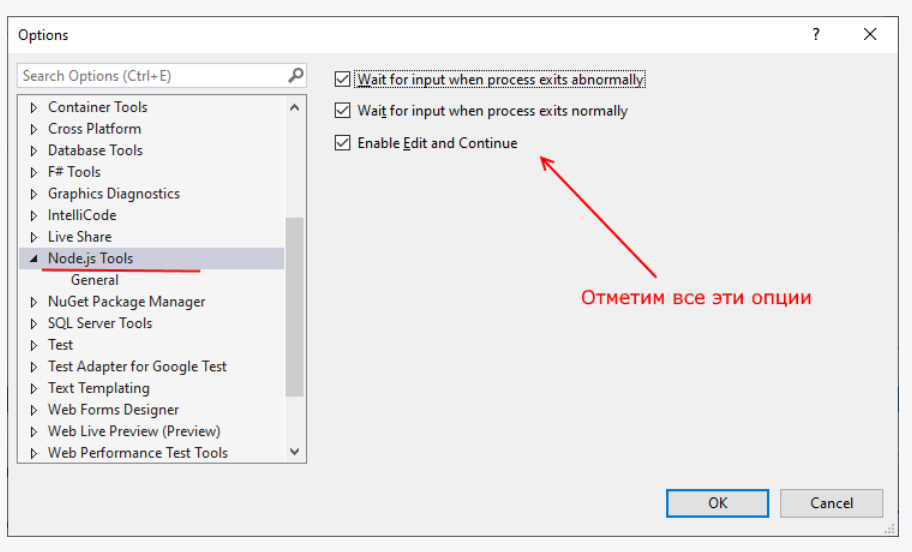
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class User{      name : string;      constructor(\_name:string){            this.name = \_name;      }  }  const tom : User = new User("Tom");  console.log("Hello ", tom.name); |

Как и в прошлой теме, здесь определяется класс User, создается объект этого класса, и данные объекта выводятся на консоль.

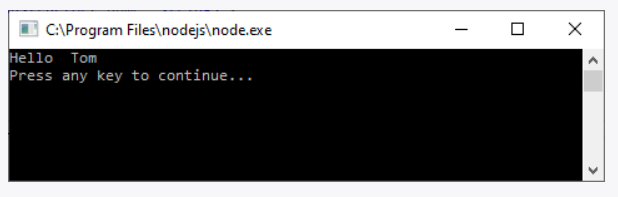
Теперь перейдем к запуску проекта. Однако разработка в Visual Studio имеет ряд тонкостей. Прежде всего при обычном запуске консоль запуститься и захлопнется быстрее, чем мы там сможем что-то увидеть. Чтобы запущенная консоль не схлопывалась, перейдем в Visual Studio к пункту меню **Tools -> Options**



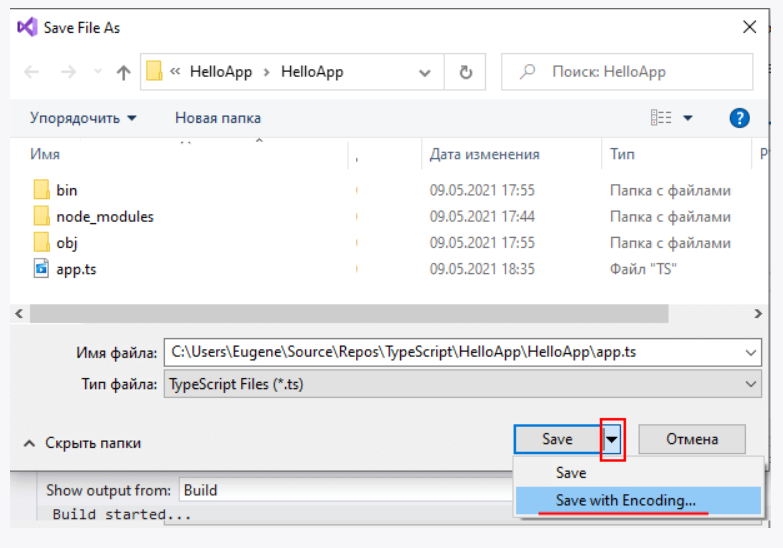
Далее в меню окна опций найдем пункт **Node.js Tools/span>. Для этого пункта предусмотрены три опции - отметим все эти опции:**



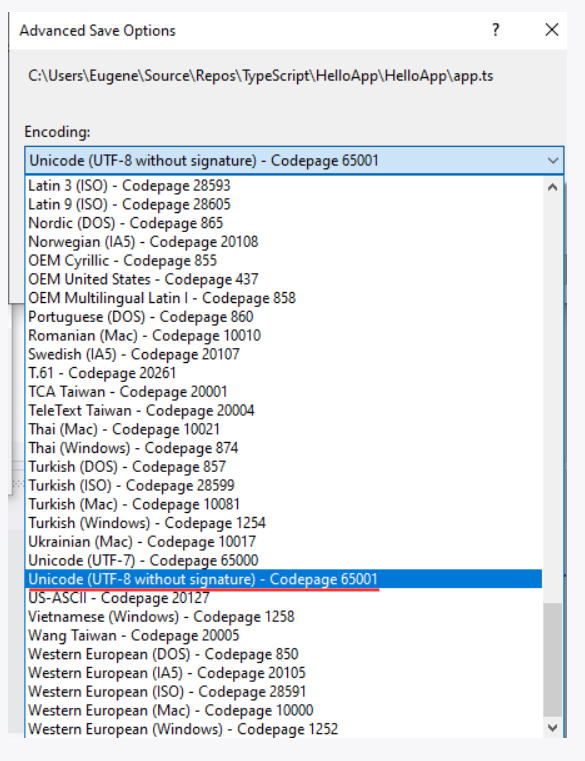
Запустим проект на выполнение с помощью комбинации клавиш **Ctrl + F5** (то есть запуск без отладки), и перед нами откроется следующая консоль:



Другая тонкость заключается в том, что по умлчанию кодировка файла app.ts может отличаться от UTF-8, из-за этого могут возникнуть проблемы с выводом кириллических символов. В этом случае мы можем в Visual Studio перейти к пункту меню **File -> Save As**. Далее в выпадающем меню у кнопки **Save** выбрать пункт **Save with Encoding...**:



Затем в поле **Encoding** укажем кодировку Unicode (UTF-8 without signature).



## 1.4.Настройки компиляции

При компиляции файлов TypeScript из командной строки компилятор позволяет установить ряд конфигурационных настроек. Для установки нам доступна куча различных настроек, полный список которых можно найти на странице [Параметры компиляции в TypeScript](http://www.typescriptlang.org/docs/handbook/compiler-options.html). Рассмотрим лишь основные из них.

### **Автоматическая перекомпиляция**

Опция **--watch**, а также ее сокращенная версия **-w** автоматически перекомпилирует файлы typescript, если в них были внесены какие-либо изменения. Благодаря чему не надо при каждом малейшем изменении вручную вводить команду в консоль для перекомпиляции.

tsc -w app.ts

### **Версия ECMAScript**

С помощью параметра **–-target** или его сокращенной версии **–t** можно задать версию стандарта JavaScript, в которую будет компилироваться код TypeScript. Этот параметр может принимать следующие значения: "ES3" (по умолчанию), "ES5", "ES6" / "ES2015", "ES7" / "ES2016", "ES2017", "ES2018", "ES2019", "ES2020" или "ESNext":

tsc app.ts -t ES6

### **Удаление комментариев**

По умолчанию в файлы javascript переходят все комментариии, которыми снабжен код в файлах TS. Удаление комментариев при компиляции осуществляется с помощью параметра **–-removeComments**:

tsc app.ts –removeComments

### **Установка каталога**

С помощью параметра **--outDir** можно задать папку для хранения скомпилированных файлов js:

tsc --outDir D:\ts\js app.ts

В данном случае скомпилированный файл app.js окажется в папке *D:\ts\js*

### **Объединение файлов**

Если у нас несколько файлов TS, то с помощью параметра **--outFile** их можно объединить в один файл js:

tsc --outFile output.js app.ts hello.ts

Здесь файлы app.ts и hello.ts скомпилируются в один файл output.js

### **Тип модуля**

С помощью параметра **--module**, либо **-m** можно указать тип модуля, который будет использоваться для компиляции. Эта опция может принимать следующие значения: "None", "CommonJS" (значение по умолчанию, если задана версия ECMAScript "ES3" или "ES5"), "AMD", "System", "UMD", "ES2015", "ES2020" и "ESNext".

tsc -m commonjs app.ts

### **Несколько параметров**

Если надо задать несколько параметров, то они и их значения последовательно перечисляются через пробел.

tsc -t ES5 --outDir js -m commonjs app.ts

### **Вызов справки**

И чтобы посмотреть все доступные параметры и справку по ним, можно воспользоваться параметром **-h**:

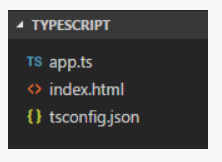
tsc -h

## 1.5.Файл конфигурации tsconfig.json

С помощью файла tsconfig.json можно настроить проект TypeScript. В частности, этот файл выполняет следующие задачи:

* устанавливает корневой каталог проекта TypeScript
* выполняет настройку параметров компиляции
* устанавливает файлы проекта

Для его использования достаточно вручную добавить новый файл с именем **tsconfig.json** в корень проекта.



### **Структура файла**

tsconfig.json представляет собой стандартный файл в формате json, который содержит ряд секций. Так, секция "compilerOptions" настраивает параметры компиляции. Здесь можно указать необходимые параметры и их значения. Параметры называются также, как и в командной строке. И соответственно значения мы им можем передать те же, что передаются в командной строке (или терминале). Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {      "compilerOptions": {          "target": "es5",          "removeComments": true,          "outDir": "js",          "sourceMap": true          "outFile": "main.js"      }  } |

Здесь используется все те же параметры, которые применяются при компиляции в командной строке. Например, параметр "target" указывает, какой стандарт JavaScript будет применяться при компиляции.

Параметр "removeComments" удаляет комментарии. Параметр "outDir" задает каталог для скомпилированных файлов. "sourceMap" указывает, что надо сгенерировать карту для сопоставления исходных и скомпилированных файлов. И параметр "outFile" задает название выходного файла.

При необходимости можно включать другие опции компиляции.

С помощью секции **files** можно установить набор включаемых в проект файлов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {      "compilerOptions": {          "target": "es5",          "removeComments": true,          "outFile": "../../built/local/tsc.js"      },      "files":[          "app.ts",          "interfaces.ts",          "classes.ts",      ]  } |

Если секция "files" не указана в файле tsconfig.json, то компилятор по умолчанию включает все файлы TypeScript (файлы с расширением *\*.ts* и *\*.tsx*), которые находятся в каталоге и подкаталогах проекта. Если же указана секция "files", то используются только файлы из этой секции.

Параметр **exclude**, наоборот, позволяет исключить при компиляции определенные файлы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {      "compilerOptions": {          "target": "es5",          "removeComments": true,          "outFile": "../../built/local/tsc.js"      },      "exclude":[          "wwwroot",          "node\_modules"      ]  } |

При компиляции компилятор не будет учитывать файлы TypeScript, которые находятся в каталогах из секции exclude.

При этом следует учитывать, что если в файле одновременно будут заданы обе секции files и exclude, то секция exclude будет игнорироваться.

Все файлы, на которые есть ссылки в файлах из секции "files", также компилируются.

Параметр **compileOnSave** при значении true указывает используемой IDE сгенерировать все файлы js при каждом сохранении файлов TypeScript:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {      "compileOnSave": true,      "compilerOptions": {          "target": "es5"      } |

### **Использование файла tsconfig.json**

Файл tsconfig.json используется при компиляции в том случае, если компилятору не передаются названия файлов, которые надо скомпилировать. В этом случае компилятор TypeScript просматривает текущий каталог, ищет в нем файл tsconfig.json и затем при компиляции использует те параметры, которые определены в этом файле.

Если же компилятору передаются названия файлов, например, tsc app.ts, то файл tsconfig.json игнорируется.

Полный список применяемых опций в tsconfig.json можно просмотреть в [документации](https://www.typescriptlang.org/tsconfig)

# 2.Основы TypeScript

## 2.1.Переменные и константы

Для хранения данных в программе в TypeScript, как и во многих языках программирования используются переменные.

Для определения переменных, как в JavaScript, можно использовать ключевое слово **var**:

|  |  |
| --- | --- |
| 1 | var z;  // переменная z |

Другой способ определения переменной применяет ключевое слово **let**, которое было добавлено в JavaScript в стандарте ES 2015:

|  |  |
| --- | --- |
| 1 | let z; |

Применение let является более предпочтительным, поскольку позволяет избежать ряд проблем, связанных с объявлением переменных. В частности, с помощью var мы можем определить два и более раз переменную с одним и тем же именем:

|  |  |
| --- | --- |
| 1  2  3  4 | var x = "hello";  console.log(x);  var x = "work";  console.log(x); |

Если программа большая, то мы можем не уследить за тем, что такая переменная уже объявлена, что является источником потенциальных ошибок. Подобную проблему позволяет решить let:

|  |  |
| --- | --- |
| 1  2  3  4 | let x = "hello";  console.log(x);  let x = "work"; // здесь будет ошибка, так как переменная x уже объявлена  console.log(x); |

То же самое относится, если переменная с одним и тем же именем определяется два раза, но с помощью обоих ключевых слов - let и var:

|  |  |
| --- | --- |
| 1  2  3  4 | let x = "hello";  console.log(x);  var x = "work"; // здесь будет ошибка, так как переменная x уже объявлена  console.log(x); |

Определив переменную, мы можем установить ее значение и в процессе работы программы поменять его на другое:

|  |  |
| --- | --- |
| 1  2 | let z = 6;  z = 8; |

Кроме переменных в TypeScript имеются константы - для них можно установить значение только один раз. И далее процессе работы программы мы уже не сможем изменить это значение. Для определение констант используется ключевое слово **const**:

|  |  |
| --- | --- |
| 1  2 | const z = 6;  z = 8;  // здесь ошибка - нельзя изменить значение константы z |

### **Область видимости переменной и констант**

Выше было сказано, что нельзя определить с помощью let определить более одного раза переменную (а с помощью const - константу) с одним и тем же именем. Данное утверждение относится к ситуации, когда переменные let и константы определяются в одной области видимости. Есть локальная область видимости, которая опреляется блоком кода, ограниченным фигурными скобками **{ //....код }**. И есть глобальная область видимости - вне какого-либо блока кода.

Переменные и константы во вложенной области видимости могут скрывать переменные и константы с тем же именем, определенные во внешней области видимости. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let x = 10;  {      let x = 25;      console.log(x); // 25  }  console.log(x); // 10 |

Здесь определены две переменных с одним и тем же именем. Но ошибки нет, поскольку они определены в разных областях видимости. Переменная let x = 10 определена в глобальной области видимости. А переменная let x = 25; определена во вложенном блоке кода - в локальной области видимости. И она скрывает внешнюю глобальную переменную. Поэтому консоль браузера выведет нам следующее:

25

10

То же самое относится к случаю, если переменные определяются в локальных областях видимости, которые вложены одна в другую:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | let x = 10;  {      let x = 25;      {          let x = 163;          console.log(x); // 163      }      console.log(x); // 25  }  console.log(x); // 10 |

### **Различия между var и let/const**

Можно следующим образом суммировать различия между var и let/const:

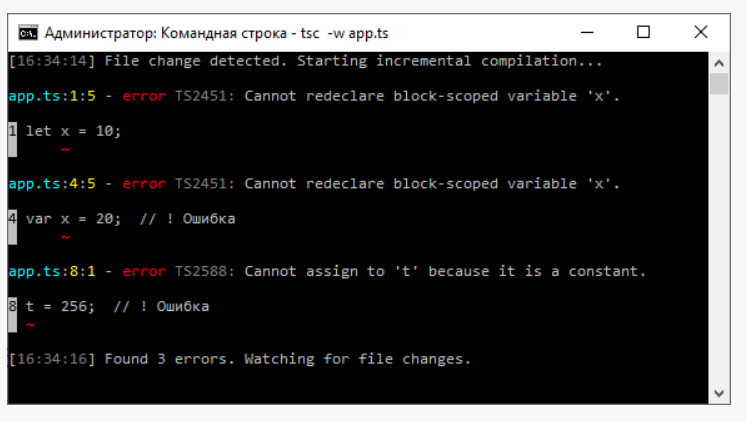
|  |  |
| --- | --- |
| **var** | **let/const** |
| Может быть доступна вне области видимости, в которой она определена.   |  |  | | --- | --- | | 1  2  3  4 | {      var x = 94;  }  console.log(x); // норм | | Доступна только в рамках области видимости, в котором она определена   |  |  | | --- | --- | | 1  2  3  4 | {      let x = 94;  }  console.log(x); // ! Ошибка | |
| Можно использовать в функции перед определением.   |  |  | | --- | --- | | 1  2 | console.log(x); // undefined, но норм  var x = 76; | | Можно использовать только после определения.   |  |  | | --- | --- | | 1  2 | console.log(x); // ! Ошибка  let x = 76; | |
| В одной и той же области видимости можно несколько раз определить переменную с одним и тем же именем.   |  |  | | --- | --- | | 1  2  3  4 | var x = 72;  console.log(x); // 72  var x = 24;     // норм  console.log(x); // 24 | | В одной и той же области видимости можно только один раз определить переменную с одним и тем же именем.   |  |  | | --- | --- | | 1  2  3  4 | let x = 72;  console.log(x); // 72  let x = 24;     // ! Ошибка  console.log(x); | |

## 2.2Параметры компиляции noEmitOnError и target

В прошлой теме было рассмотрено, что нельзя в одной области видимости несколько раз определить переменную с одним и тем же именем или что значение константы нельзя изменить.

Но давайте рассмотрим следующую небольшую программу на TypeScript:

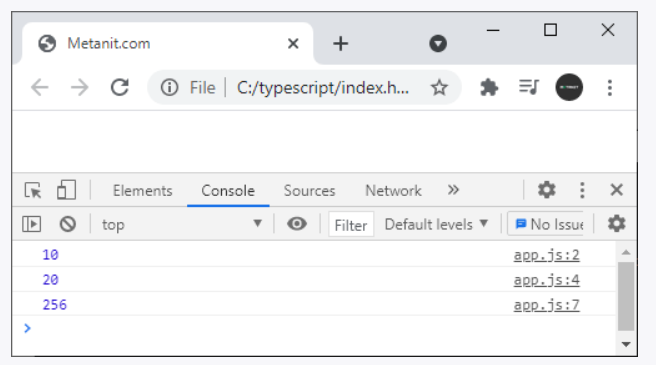
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let x = 10;  console.log(x); // 10    var x = 20;     // ! Ошибка  console.log(x); // Но консоль выведет 20    const t = 50;  t = 256;        // ! Ошибка  console.log(t); // Но консоль выведет 256 |

При компиляции ожидаемо мы увидим ряд ошибок:

Тем не менее, если применяются настройки компиляции по умолчанию, javascript-файл вполне нормально скомпилируется, в частности, он будет иметь следующее содержание:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var x = 10;  console.log(x); // 10  var x = 20; // ! Ошибка  console.log(x); // Но консоль выведет 20  var t = 50;  t = 256; // ! Ошибка  console.log(t); // Но консоль выведет 256 |

И в итоге в консоли браузера мы увидим результат работы этого скрипта:



Но для компилятора TS можно с помощью параметра **--noEmitOnError** можно указать, что не надо генерировать файл javascript, если при компиляции возникли ошибки. Этот параметр можно указать напрямую при вызове компилятора в консоли:

tsc --noEmitOnError app.ts

Либо в файле конфигурации **tsconfig.json** для настройки "noEmitOnError" указать значение true:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {      "compilerOptions": {          "target": "es5",          "noEmitOnError": true,          "outFile": "app.js"      }  } |

### **target**

Исправим предыдущий пример, чтобы он проходил компиляцию

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | let x = 10;  console.log(x); // 10    var y = 20;  console.log(y);     // 20    const t = 50;  console.log(t); // 50 |

При компиляции с настройками по умолчанию мы получим следующий файл javascript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var x = 10;  console.log(x); // 10  var y = 20;  console.log(y); // 20  var t = 50;  console.log(t); // 50 |

Мы видим, что для определения всех значений применяется ключевое слово var, несмотря на то, что даже не самые современные браузеры уже давно поддерживают операторы let и const. Но компилятор typescript по умолчанию компилирует в стандарт ES3, для которого существует только оператор var. Но var/let/const - это частные случае. Как минимум, подавляющая часть стандарта ES2015 уже давно поддерживается во многих распространенных браузерах, свой путь прорубают новые стандарты. И если мы хотим, чтобы компилятор использовал более современный стандарт, то нам надо задать при компиляции параметр **-target/-t**.

Например, опять возьмем файл, который содержит ошибки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let x = 10;  console.log(x); // 10    var x = 120;        // ! Ошибка  console.log(x); // Но консоль выведет 20    const t = 50;  t = 256;        // ! Ошибка  console.log(t); // Но консоль выведет 256 |

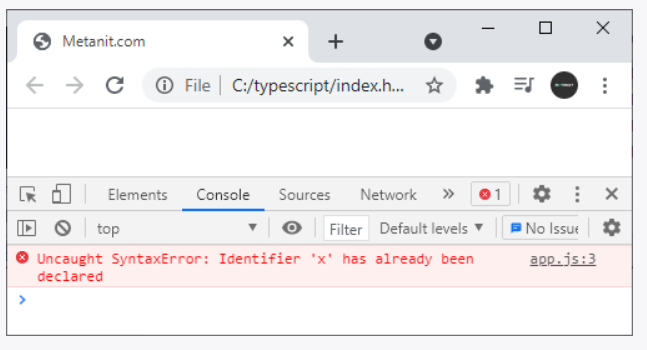
Скомпилируем в стандарт "ES2015":

tsc app.ts -t ES2015

Здесь не указан параметр "noEmitOnError", поэтому компилятор сообщит об ошибках, однако файл все равно скомпилирует. Скомпилированный файл практически не будет отличаться от кода на typescript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let x = 10;  console.log(x); // 10  var x = 120; // ! Ошибка  console.log(x); // Но консоль выведет 20  const t = 50;  t = 256; // ! Ошибка  console.log(t); // Но консоль выведет 256 |

Однако в браузере, который поддерживает let и const, все равно мы столкнемся с ошибкой:



Также можно задать эту настройку в файле **tsconfig.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {      "compilerOptions": {          "target": "es2015",          "noEmitOnError": true,          "outFile": "app.js"      }  } |

## 2.3.Типы данных

TypeScript является строго типизированным языком, и каждая переменная и константа в нем имеет определенный тип. При этом в отличие от javascript мы не можем динамически изменить ранее указанный тип переменной.

В TypeScript имеются следующие базовые типы:

* **boolean**: логическое значение true или false
* **number**: числовое значение
* **string**: строки
* **Array**: массивы
* **кортежи**
* **Enum**: перечисления
* **Any**: произвольный тип
* **Symbol**
* **null и undefined**: соответствуют значениям null и undefined в javascript
* **Never**: также представляет отсутствие значения и используется в качестве возвращаемого типа функций, которые генерируют или возвращают ошибку

Большинство из этих типов соотносятся с примитивными типами из JavaScript.

Для установки типа применяется знак двоеточия, после которого указывается название типа. Примеры создания переменных:

|  |  |
| --- | --- |
| 1  2  3 | let x: number = 10;  let hello: string = "hello world";  let isValid: boolean = true; |

То есть в данном случае выражение let hello: string = "hello world" указывает, что переменная hello будет иметь тип string и значение hello world.

При этом если в коде мы потом захотим изменить тип, например:

|  |  |
| --- | --- |
| 1  2 | let hello: string = "hello world";  hello = 23; |

То в процессе компиляции компилятор TypeScript выдаст ошибку.

Вначале рассмотрим примитивные типы данных, которые есть в TypeScript.

### **boolean**

Тип **Boolean** представляет логическое значение true или false:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let isEnabled: boolean = true;  let isAlive: boolean = false;    console.log(isEnabled);  console.log(isAlive); |

### **number**

Тип number представляет числа, причем все числа в TypeScript, как и в JavaScript, являются числами с плавающей точкой. Поэтому с помощью данного типа можно определять как целые числа, так и числа с плавающей точкой:

|  |  |
| --- | --- |
| 1  2 | let age: number = 36;  let height: number = 1.68; |

TS поддерживает двоичную, восьмеричную, десятичную и шестнадцатиричную записи чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | let decimal: number = 6;  // шестнадцатиричная система  let hex: number = 0xf00d;       // 61453 в десятичной  // двоичная система  let binary: number = 0b1010;    // 10 в десятичной  // восьмиричная система  let octal: number = 0o744;      // 484 в десятичной  console.log(decimal);  console.log(hex);  console.log(binary);  console.log(octal); |

### **string**

String представляет строки. Как и в JavaScript, в TypeScript строки можно заключать в двойные, либо в одинарные кавычки:

|  |  |
| --- | --- |
| 1  2 | let firstName: string = "Tom";  let lastName = 'Johns'; |

Кроме того, TypeScript поддерживает такую функциональность, как **шаблоны строк**, то есть мы можем задать шаблон в косых кавычках (`), как если бы мы писали обычную строку, и затем в саму строку можно встраивать разные выражения с помощью синтаксиса ${ expr }, где expr - это выражение. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | let firstName: string = "Tom";  let age: number = 28;  let info: string = `Имя ${firstName}    Возраст: ${age}`;  console.log(info);  // Имя Tom    Возраст: 28 |

Косые кавычки также можно применять для установки многострочного текста:

|  |  |
| --- | --- |
| 1  2 | let sentence: string = `Hello World!  Goob bye World!`; |

### **bigint**

Для представления очень больших чисел в TypeScript добавлен специальный тип **bigint**. Этот тип позволяет хранить числа больше, чем 253 - 1 - больше, чем позволяет тип number.

Есть два способа определения объекта этого типа. Первый способ - с помощью функции **BigInt()**:

|  |  |
| --- | --- |
| 1  2 | const num1: bigint = BigInt(100);  console.log(num1); |

Второй способ - с помощью добавления символа **n** после числового значения:

|  |  |
| --- | --- |
| 1  2 | const num2: bigint = 100n;  console.log(num2); |

Следует отметить, что этот тип - часть стандарта ES2020, поэтому при компиляции следует установить данный стандарт в качестве целевого через параметр target. Например, в файле **tsconfig.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {      "compilerOptions": {          "target": "es2020",          "outFile": "app.js"      }  } |

### **Выведение типа**

Но можно в принципе и не указывать тип переменной. Например:

|  |  |
| --- | --- |
| 1  2 | let hello = "hello world";  hello = 23; |

В этом случае TypeScript автоматически выведет тип из присваемого данной переменной значения. Так, на первой строке компилятор TS увидит, что переменной hello присваивается строка, поэтому для нее будет использоваться тип string. Однако на второй строке опять же компилятор выдаст ошибку, поскольку hello переменной уже определен тип string. А новое значение предполагает тип number.

### **Тип any**

Any описывает данные, тип которых может быть неизвестен на момент написания приложения.

|  |  |
| --- | --- |
| 1  2  3  4 | let someVar: any = "hello";  console.log(someVar);   // сейчас someVar - это string  someVar = 20;  console.log(someVar);   // сейчас someVar - это number |

Так как здесь применяется тип any, то данный код скомпилируется без ошибок, несмотря на смену строкового значения на числовое. И также мы можем объявлять массивы данного типа:

|  |  |
| --- | --- |
| 1 | var someArray: any[] = [ 24, "Tom", false]; |

Если переменная определяется без значения и указания типа, и только впоследствии при работе программы ей присваивается значение, тогда считается, что она имеет тип **any**:

|  |  |
| --- | --- |
| 1  2  3 | let x;  // тип any  x = 10;  x = "hello"; |

С одной стороны, any может показаться удобным типом. Однако, с другой стороны, он лишает программу преимуществ языка TypeScript, в частности, статической типизации. Соответственно может привнести в программу потенциальные ошибки, связанные с типизацией данных, которые компилятор не сможет отследить на этапе компиляции.

### **Проверка типа**

С помощью оператора **typeof** мы можем проверить тип переменной. Это может быть необходимо, когда мы хотим выполнить некоторые операции с переменной, но нам неизвестен ее точный тип (например, переменная представляет тип any). Данная функциональность еще называется **type guards** или защита типа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let sum: any;  sum = 1200;  sum = "тысяча двести";  let result: number = sum / 12;  console.log(result); // NaN - строку нельзя разделить на число |

Переменная sum может хранит любое значение, однако деление может работать только с числами. Поэтому перед делением выполним проверку на тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | let sum: any;  sum = 1200;    if (typeof sum === "number") {        let result: number = sum / 12;      console.log(result);  }  else{      console.log("invalid operation");  } |

Оператор typeof возвращает тип значения. Например, выражение typeof sum в данном случае возращает "number", так как переменная sum представляет число.

Оператор typeof может возвращать следующие значения:

* "string"
* "number"
* "bigint"
* "boolean"
* "symbol"
* "undefined"
* "object"
* "function"

## 2.4.Функции

### **Определение функции**

В javascript функции определяются с помощью ключевого слова **function**, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function add(a, b) {      return a + b;  }  // использование функции  let result1 = add(1, 2); // результат 3  let result2 = add("1", "2"); // результат 12 |

TypeScript также определяет функцию с помощью ключевого слова function, но при этом добавляет дополнительные возможности по работе с функциями. В частности, теперь мы можем определить тип передаваемых параметров и тип возвращаемого значения.

### **Параметры функции**

Функция может иметь параметры, которые указываются после названия функции в скобках через запятую. Через двоеточие после имени параметра указывается его тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // определение функции  function add(a: number, b: number){      let result = a + b;      console.log(result);  }  // вызов функции  add(20, 30); // 50  add(10, 15); //25 |

Однако поскольку параметры имеют тип number, то при вызове функции

|  |  |
| --- | --- |
| 1 | add("1", "2"); |

компилятор TS выдаст ошибку, так как параметры должны иметь тип number, а не тип string.

При этом функция может не только использовать передаваемые параметры, но и глобальные переменные, определенные во вне:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let koef: number = 1.5;    function add(a: number){      let result = a \*koef;      console.log(result);  }    add(20); // 30  add(10); //15 |

### **Результат функции**

Функция может возвращать значение определенного типа, который еще называется типом функции. Возвращаемый тип функции ставится после списка параметров через двоеточие:

|  |  |
| --- | --- |
| 1  2  3  4  5 | function add(a: number, b: number): number {      return a + b;  }  let result = add(1, 2);  console.log(result); |

В данном случае функция будет возвращать значение типа number.

Если функция ничего не возвращает, то указывается тип **void**:

|  |  |
| --- | --- |
| 1  2  3  4 | function add(a: number, b: number): void {      console.log(a + b);  }  add(10, 20); |

В принципе мы можем и не указывать тип, тогда он будет выводиться неявно на основе возвращаемого значения:

|  |  |
| --- | --- |
| 1  2  3  4 | function add(a: number, b: number) {      return a + b;  }  let result = add(10, 20); |

### **Необязательные параметры**

В typescript при вызове в функцию должно передаваться ровно столько значений, сколько в ней определено параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function getName(firstName: string, lastName: string) {      return firstName + " " + lastName;  }    let name1 = getName("Иван", "Кузнецов");  let name2 = getName("Иван", "Михайлович", "Кузнецов");  //ошибка, много параметров  let name3 = getName("Иван");  //ошибка, мало параметров |

Чтобы иметь возможность передавать различное число значений в функцию, в TS некоторые параметры можно объявить как необязательные. Необязательные параметры должны быть помечены вопросительным знаком ?. Причем необязательные параметры должны идти после обязательных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function getName(firstName: string, lastName?: string) {      if (lastName)          return firstName + " " + lastName;      else          return firstName;  }    let name1 = getName("Иван", "Кузнецов");  console.log(name1); // Иван Кузнецов  let name2 = getName("Вася");  console.log(name2); // Вася |

Во втором случае, когда в функцию передается только имя, второй используемый параметр будет иметь неопределенное значение или "undefined". Поэтому с помощью условной конструкции проверяется наличие значения для этого параметра.

### **Значения параметров по умолчанию**

Параметры позволяют задать начальное значение по умолчанию. И если для такого параметра не передается значение, то он использует значение по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function getName(firstName: string, lastName: string="Иванов") {        return firstName + " " + lastName;  }    let name1 = getName("Иван", "Кузнецов");  console.log(name1); // Иван Кузнецов  let name2 = getName("Вася");  console.log(name2); // Вася Иванов |

Причем в качестве значения можно передавать результат другого выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function defaultLastName(): string{      return "Smith";  }    function getName(firstName: string, lastName: string=defaultLastName()) {        return firstName + " " + lastName;  }    let name1 = getName("Tom");  console.log(name1); // Tom Smith |

## 2.5.Тип функции и стрелочные функции

### **Тип функции**

Каждая функция имеет тип, как и обычные переменные. Тип функции фактически представляет комбинацию типов параметров и типа возвращаемого значения. В общем виде определение типа функции выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | (параметр1: тип, параметр2: тип,...параметрN: тип) => тип\_результата; |

В скобках идет перечисление через запятую параметров и их типов. После списка параметров через оператор **=>** указывается тип возвращаемого функцией результата.

Например, возьмем следующую функцию:

|  |  |
| --- | --- |
| 1  2  3 | function hello (){      console.log("Hello TypeScript");  }; |

Эта функция не имеет параметров и ничего не возвращает. Если функция ничего не возвращает, то фактически тип ее возвращаемого значения - **void**. Таким образом, функция hello имеет тип

|  |  |
| --- | --- |
| 1 | ()=>void; |

Используя тип функции, мы можем определить переменные, константы и параметры этого типа. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function hello (){      console.log("Hello TypeScript");  };    let message: ()=>void = hello;  message(); |

В данном случае переменная message представляет любую функцию, которая не принимает параметров и ничего не возвращает.

Другой пример - функция, которая принимает параметры и возвращает некоторый результат:

|  |  |
| --- | --- |
| 1  2  3 | function sum (x: number, y: number): number {      return x + y;  }; |

Она имеет тип (x:number, y:number) => number;, то есть принимает два параметра number и возвращает значение типа number.

Также мы можем определять значения этого типа функции:

|  |  |
| --- | --- |
| 1 | let op: (x:number, y:number) => number; |

То есть переменная op представляет любую функцию, которая принимает два числа и которая возвращает число. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function sum (x: number, y: number): number {      return x + y;  };  function subtract (a: number, b: number): number {      return a - b;  };    let op: (x:number, y:number) => number;    op = sum;  console.log(op(2, 4));  // 6    op = subtract;  console.log(op(6, 4));  // 2 |

Здесь вначале переменная op указывает на функцию sum. И соответственно вызов op(2, 4) фактически будет представлять вызов sum(2, 4). А затем op указывает на функцию subtract.

### **Функции как параметры других функций**

Тип функции можно использовать как тип переменной, но он также может применяться для определения типа параметра другой функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function sum (x: number, y: number): number {      return x + y;  };  function multiply (a: number, b: number): number {      return a \* b;  };    function mathOp(x: number, y: number, op: (a: number, b: number) => number): number{        return op(x, y);  }  console.log(mathOp(10, 20, sum)); // 30  console.log(mathOp(10, 20, multiply)); // 200 |

Здесь в функции mathOp() третий параметр представляет функцию, которая принимает два параметра типа number и возвращает значение типа number. Соответственно при вызове функции mathOp() мы можем передать в нее, например, определенные здесь функции sum() или multiply(), которые соответствуют типу (a: number, b: number) => number

Если определенный тип функции предстоит очень часто использовать, то для него оптимальнее определить псевдоним и обращаться к типу по этому псевдониму:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | type Operation = (a: number, b: number) => number;    function mathOp(x: number, y: number, op: Operation): number{        return op(x, y);  }  let sum: Operation = function(x: number, y: number): number {      return x + y;  };    console.log(mathOp(10, 20, sum)); // 30 |

В данном случае тип (a: number, b: number) => number проецируется на псевдоним Operation, который может использоваться для определения переменных и параметров.

### **Стрелочные функции**

Для определения функций в TypeScript можно использовать стрелочные функции или arrow functions. Стрелочные функции представляют выражения типа (параметры) => тело функции. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | let sum = (x: number, y: number) => x + y;    let result = sum(15, 35); // 50  console.log(result); |

Тип параметров можно опускать:

|  |  |
| --- | --- |
| 1  2  3  4 | let sum = (x, y) => x + y;    let result = sum(15, 35); // 50  console.log(result); |

Если стрелочная функция не требует параметров, то используются пустые круглые скобки. Если передается только один параметр, то скобки можно опустить:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let square = x => x \* x;  let hello = () => "hello world"    console.log(square(5)); // 25  console.log(hello());   // hello world |

Если тело функции представляет множество выражений, а не просто одно выражение, как в примере выше, тогда можно опять же заключить все выражения в фигурные скобки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let sum = (x: number, y: number) => {      x \*= 2;      return x + y;  };    let result = sum(15, 35); // 65  console.log(result); |

Стрелочные функции можно передавать в функцию вместо параметра, который представляет собой функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function mathOp(x: number, y: number, operation: (a: number, b: number) => number): number{        let result = operation(x, y);      return result;  }  console.log(mathOp(10, 20, (x,y)=>x+y)); // 30  console.log(mathOp(10, 20, (x, y) => x \* y)); // 200 |

**2.6.Объединения union**

Объединения или union не являются собственно типом данных, но они позволяют комбинировать или объединить другие типы. Так, с помощью объединений можно определить переменную, которая может хранить значение двух или более типов:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let id : number | string;  id = "1345dgg5";  console.log(id); // 1345dgg5  id = 234;  console.log(id);  // 234 |

Чтобы определить все типы, которые должно представлять переменная, все эти типы разделяются прямой чертой: number | string. В данном случае переменная id может представлять как тип string, то есть строку, так и число.

Подобным образом можно использовать union для определения параметров функции:

|  |  |
| --- | --- |
| 1  2  3 | function printId(id: number|string){      console.log(`Id: ${id}`);  } |

Здесь функция printId() в качестве параметра id может принимать как значения типа number, так и значения типа string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function printId(id: number|string){      console.log(`Id: ${id}`);  }    let id: string|number = "ruy74";    printId("1h2e3l4o5");  printId(9876);  printId(id); |

Тип union подходит для тех ситуаций, когда логика работы со всеми объединенными типами однообразна, например, как в случае выше, где значение извне встраивается в строку и выводится на консоли. Вне зависимости от типа действия одинаковы.

Однако иногда логика может различаться в зависимости от переданного типа. В этом случае можно использовать проверку на тип, чтобы разграничить логику для различных типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function printSentence(words: string[]|string){        // если words - строка        if (typeof words === "string") {          console.log(words);        } else {          // Если words - массив string[]          console.log(words.join(" "));        }  }  printSentence(["Язык", "программирования", "TypeScript"]);  printSentence("Язык программирования JavaScript"); |

В данном случае функция printSentence() может принимать как строку, так и массив строк, чтобы затем вывести строки на консоль. И в зависимости от типа данных действия отличаются. Так, для массива применяется метод words.join(" "), который объединяет все элементы массива в одну строку, разделенные пробелом.

## 2.7.Null и undefined

Как и в JavaScript, в TypeScript есть специальные типы **undefined** и **null**, которые принимают соответствующие значения undefined и null.

Использование undefined и null зависит от параметра компиляции **strictNullChecks**. По умолчанию он имеет значение false и не используется, а это значит, что по умолчанию мы можем применять эти типы, как и в javascript:

|  |  |
| --- | --- |
| 1  2 | let a: undefined = undefined;  let b: null = null; |

Но фактически мы можем присваивать значения undefined и null переменным других типов, например, number:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let x: number = undefined;  console.log(x);  x = null;  console.log(x);  x = 5;  console.log(x); |

В этом плане null и undefined выступают как подтипы других типов и полезны преимущественно в каких-то операциях, где неизвестен результат - то ли это будет число или строка, то ли это будет null. В этом случае, чтобы избежать возможной ошибки, мы можем проверить значение на undefined или null, собственно как и в javascript.

В то же время отсутствие проверки для этих типов со стороны компилятора является потенциальным источников багов, поэтому нередко при компиляции применяется параметр **strictNullChecks**. Либо при установке флага при компиляции в консоли:

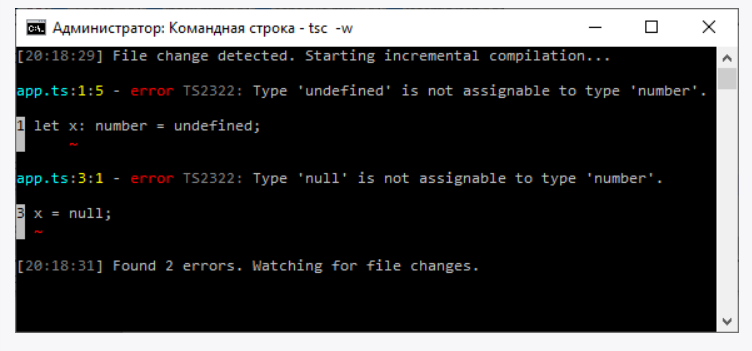
--strictNullChecks

Либо с помощью установки значения true в файле конфигурации **tsconfig.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {      "compilerOptions": {          "target": "es2015",          "noImplicitAny": true,          "noEmitOnError": true,            "strictNullChecks": true,            "outFile": "app.js"      }  } |

В этом случае при попытке присвоить значение типа undefined или null переменной другого типа компилятор при компиляции выбросит ошибку. Например, при компиляции предыдущего примера кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let x: number = undefined;  console.log(x);  x = null;  console.log(x);  x = 5;  console.log(x); |

Мы получим следующие ошибки:

Также мы получим ошибку, если значение undefined присвоим переменной типа null или значение null переменной типа undefined.

Тем не менее нередки ситуации, когда мы точно не знаем, имеет ли какая-то переменная или параметр функции или конкретное значение или оно отсутствует. Особенно в тех случааях, когда мы получаем значение из вне, например, с помощью запроса к какому-нибудь сетевому ресурсу. В этом случае может потребоваться, чтобы переменная или параметр могли принимать значение null. И в этом случае можно использовать объединения. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let userId: number|null = null;    function printId(id: number|null){      if (id === null) {          console.log("пользователь отсутствует");      } else {          console.log(`id пользователя: ${id}`);      }  }  printId(userId)     // пользователь отсутствует  userId = 45;  printId(userId);    // id пользователя: 45 |

В данном случае переменная userId представляет объединение number|null, поэтому этой переменной можно присвоить как конкретное числовое значение, так и значение null.

Аналогично функция printId через параметр id может принимать как значение null, так и числовое значение.

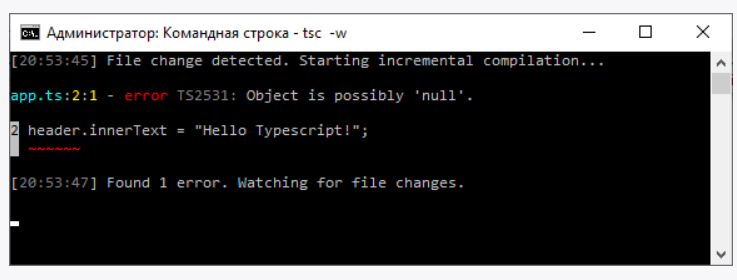
### **Оператор !**

Оператор **!** (non-null assertion operator) позволяет указать, что объект не представляет значение null и undefined. Так, возьмем следующий пример:

|  |  |
| --- | --- |
| 1  2 | const header: HTMLElement|null = document.getElementById("header");  header.innerText = "Hello Typescript!"; |

Встроенная функция document.getElementById() возвращает элемент веб-страницы по id, который представляет тип **HTMLElement|null**. То есть он может иметь значение null, если html-элемента с подобным id нет на веб-странице. Получив элемент, мы пытаемся с помощью его свойства innerText изменить его текстовое содержимое.

Вроде все нормально, однако умолчанию с включенной опцией strictNullChecks при компиляции мы получим ошибку:



Чтобы избежать ошибки, используем оператор !:

|  |  |
| --- | --- |
| 1  2 | const header: HTMLElement|null = document.getElementById("header");  header!.innerText = "Hello Typescript!"; |

Оператор **!** ставится после объекта, который теоретически может принимать значение null перед обращением к его свойствам и методам:

|  |  |
| --- | --- |
| 1  2 | object!.property  object!.method() |

В то же время надо учитывать, что этот оператор не меняет значения объекта. Например, если объект имеет значение null или undefined, то данный оператор не поможет. Программа скомпилируется, но при выполнении скрипта программа все равно сгенерирует ошибку:

|  |  |
| --- | --- |
| 1  2 | const header: HTMLElement|null = null;  header!.innerText = "Hello Typescript!"; |

Поэтому рекомендуется применять данный оператор, когда мы знаем, что объект не равен null или undefined.

## 2.8.Объекты

Кроме простых типов данных, как и в javascript, можно создавать сложные комплексные объекты, которые состоят из других объектов и примитивных данных. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | let person = {name:"Tom", age:23};  console.log(person.name);  // альтернативный вариант получения свойства  console.log(person["name"]); |

Но несмотря на то, что это фактически тот же самый объект, что мы могли бы использовать в JavaScript, в силу строготипизированности TS мы имеем в данном случае ограничения. В частности, если у нас будет следующий код:

|  |  |
| --- | --- |
| 1  2 | let person = { name: "Tom", age: 23 };  person = { name: "Bob" };           // ! Ошибка |

То на второй строке мы получим ошибку, поскольку компилятор после первой строки предполагает, что объект person будет иметь два свойства name и age, которые имеют тип string и number соответственно. То есть в данном случае переменная person представляет тип { name: string; age: number }. И мы могли написать так:

|  |  |
| --- | --- |
| 1  2 | let person: { name: string; age: number } = { name: "Tom", age: 23 };  console.log(person.name); |

Поэтому данной переменной мы можем присвоить другой объект, который соответствует типу { name: string; age: number } по названиям, количеству и типу свойств, как в следующем случае

|  |  |
| --- | --- |
| 1  2 | let person = { name: "Tom", age: 23 };  person = { name: "Bob", age: 35 };      // Норм |

**Необязательные свойства**

TypeScript позволяет сделать свойства необязательными. Для этого после названия свойства указывается знак вопроса **?**

|  |  |
| --- | --- |
| 1 | let person: { name: string; age?: number }; // Свойство age - необязательное |

В данном случае свойство age является необязательным, поэтому мы можем не предоставлять для него значение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let person: { name: string; age?: number }; // Свойство age - необязательное    person = { name: "Tom", age: 23 };  console.log(person.name);   // Tom  person = { name: "Bob"};    // Норм, свойство age - необязательное  console.log(person.name);   // Bob |

При обращении к неустановленному свойству мы получим **undefined**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let person: { name: string; age?: number } = { name: "Tom", age: 23 };    console.log(person.age);    // 23  person = { name: "Bob"};  console.log(person.age);    // undefined |

Поэтому при операциях с таким свойством мы можем проверять его на значение undefined:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let person: { name: string; age?: number } = { name: "Tom", age: 36};  if (person.age !== undefined) {        console.log(person.age);  } |

### **Объекты в функциях**

Функция может принимать объекты в качестве параметров и могут возвращать объект. В этих случаях необходимо указать тип объекта для параметров и результата функции. Например, объект как параметр функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function printUser(user: { name: string; age: number}) {    console.log(`name: ${user.name}  age: ${user.age}`);  }  let tom = {age: 36, name: "Tom"};    printUser(tom); |

В данном случае параметр функции printUser представляет тип { name: string; age: number} - то есть объект, который имеет два свойства типов string и number.

При этом объект, передаваемый в качестве параметра, может быть более широким - содержать больше свойств, но главное, чтобы он содержал те свойства, которые предусмотрены для параметра функции:

|  |  |
| --- | --- |
| 1  2  3  4  5 | function printUser(user: { name: string; age: number}) {    console.log(`name: ${user.name}  age: ${user.age}`);  }  let bob = {name: "Bob", age: 44, isMarried: true};  printUser(bob); |

Здесь переменная bob представляет тип {name: string, age: number, isMarried: boolean}, тем не менее он также соответствует определению типа { name: string; age: number}.

Объект как результат функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function defaultUser(): { name: string; age: number} {      return {name: "Tom", age: 37};  }    let user = defaultUser();  console.log(`name: ${user.name}  age: ${user.age}`); |

### **Оператор in**

Оператор **in** позволяет проверить наличие определенного свойства в объекта. Он возвращает true, если свойство есть в объекте, и false, если свойство отсутствует. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | let tom: { name: string; age?: number } = { name: "Tom", age: 23 };  let bob: { name: string; age?: number } = { name: "Bob"};      function printUser(user: { name: string; age?: number }){        if("age" in user){          console.log(`Name: ${user.name} Age: ${user.age}`);      }      else{          console.log(`Name: ${user.name}`);      }  }  printUser(tom);  printUser(bob); |

Здесь функция printUser() принимает объект типа { name: string; age?: number }, то есть свойство age в таком объкте может присутствовать, а может отсутствовать. Поэтому применяем оператор in для проверки наличия свойства:

|  |  |
| --- | --- |
| 1 | if("age" in user){ |

Название свойства передается как строка.

### **Декомпозиция объектов-параметров**

Если функция принимает объект в качестве параметра, то TypeScript позволяет автоматически разложить его на свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function printUser({name, age}: { name: string; age: number}) {    console.log(`name: ${name}  age: ${age}`);  }    let tom = {name: "Tom", age: 36};  printUser(tom); |

Здесь функция printUser() в качестве параметра принимает объект из двух свойств. Запись {name, age} указывает, что свойства объекта автоматически будут разложены на два параметра: name и age. Главное, чтобы названия этих параметров соответствовали названиям свойств объекта.

Некоторые параметры могут необязательными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function printUser({name, age}: { name: string; age?: number}) {    if(age!==undefined){ console.log(`name: ${name}  age: ${age}`);}    else {console.log(`name: ${name}`);}  }  let tom = {name: "Tom"};  printUser(tom);     // name: Tom    let bob = {name: "Bob", age: 44};  printUser(bob);     // name: Bob  age: 44 |

В данном случае параметр age является необязательным. Кроме того, имеет значение по умолчанию. Поэтому функция может принимать объект без свойства age.

Также они могут принимать значения по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function printUser({name, age = 25}: { name: string; age?: number}) {    console.log(`name: ${name}  age: ${age}`);  }    let tom = {name: "Tom"};  printUser(tom);     // name: Tom  age: 25    let bob = {name: "Bob", age: 44};  printUser(bob);     // name: Bob  age: 44 |

## 2.9.Псевдонимы типов

TypeScript позволяет определять псевдонимы типов с помощью ключевого слова **type**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | type id = number | string;    let userId : id = 2;  console.log(`Id: ${userId}`);  userId = "qwerty";  console.log(`Id: ${userId}`); |

Здесь для объединения number|string определяется псевдоним id. Далее мы можем использовать этот псевдоним для определения переменных.

В том числе псевдонимы могут применяться для определения типа параметров и результата функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // определяем псевдоним  type id = number | string;    // параметр функции представляет псевдоним  function printId(inputId: id){      console.log(`Id: ${inputId}`);  }  // тип результата - псевдоним  function getId(isNumber:boolean): id{      if(isNumber)          return 1;      else          return "1";  }  // применение функций  printId(12345);  printId("qwerty");  console.log(getId(true)); |

Особенно полезны могут псевдонимы, когда мы имеем дело со сложными объектами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | type Person = {name: string; age: number};    let tom: Person = {name: "Tom", age: 36};  let bob: Person = {name: "Bob", age: 41};    function printPerson(user: Person){      console.log(`Name: ${user.name}  Age: ${user.age}`);  }    printPerson(tom);  printPerson(bob); |

В данном случае определяется псевдоним Person для типа {name: string; age: number}, то есть типа, который представляет сложный объект. Затем этот псевдоним применяется для определения переменных и параметров функции.

### **Расширение псевдонимов**

Одни псевдонимы могут заимствовать или расширять код других. Для этого применяется операция **&**. Например:

|  |  |
| --- | --- |
| 1  2 | type Person = {name: string; age: number};  type Employee = Person & {company: string}; |

В данном случае псевдоним Employee расширяет псевдоним Person, добавляя к нему свойство company, которое представляет тип string. То есть фактически мы имеем дело с типом:

|  |  |
| --- | --- |
| 1 | type Employee = {name: string; age: number; company: string}; |

Применение будет аналогично применению обычных псевдонимов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | type Person = {name: string; age: number};  // расширяем псевдоним Person  type Employee = Person & {company: string};    let tom: Person = {name: "Tom", age: 36};  let bob: Employee = {name: "Bob", age: 41, company: "Microsoft"};    function printPerson(user: Person){      console.log(`Name: ${user.name}  Age: ${user.age}`);  }    printPerson(tom);  printPerson(bob);  // bob представляет Employee, но он также соответствует псевдониму Person |

**2.10.Type assertion. Преобразование к типу**

Type assertion представляет модель преобразования значения переменной к определенному типу. Обычно в некоторых ситуациях одна переменная может представлять какой-то широкий тип, например, any или union, которые по факту допускают значения различных типов. Однако при этом нам надо использовать переменную как значение строго определенного типа. И в этом случае мы можем привести к этому типу.

В качестве примера возьмем простейшую задачу - на веб-странице есть html-элемент с id = header, и мы хотим получить этот элемент, например, чтобы изменить его текст. Для получения элемента по id мы могли бы использовать встроенную js-функцию document.getElementById():

|  |  |
| --- | --- |
| 1  2 | const header = document.getElementById("header");  header.innerText = "Hello Typescript!"; |

Но в TypeScript эта функция возвращает объединение HTMLElement|null. То есть возвращаемое значение может представлять null, если соответствующий html-элемент отсутствует на веб-странице. Вследствие этого обращение к свойству header.innerText - к свойству объекта, который может быть null, содержит потенциальную ошибку. Поэтому компилятор при компиляции сгенерирует ошибку.

Однако ситуация может быть такова, что мы точно знаем, что у нас на странице есть такой элемент. Есть разные способы для решения этой проблемы, чтобы указать компилятору, что все нормально. И одним из ним является приведение типов с помощью type assertion.

Есть две формы приведения. Первая форма заключается в использовании угловых скобок:

|  |  |
| --- | --- |
| 1  2 | const header = <HTMLElement>document.getElementById("header");  header.innerText = "Hello Typescript!"; |

Перед значением в угловых скобках указывается тип, к которому надо выполнить приведение. Так, в данном случае мы получаем объект типа HTMLElement

Вторая форма заключается в применении оператора **as**, после которого указывается тип, к которому надо выполнить преобразование:

|  |  |
| --- | --- |
| 1  2 | const header = document.getElementById("header") as HTMLElement;  header.innerText = "Hello Typescript!"; |

Однако такие преобразования будут иметь силу, если мы точно знаем, что значение может быть преобразовано к целевому типу. Например, на странице есть элемент с id=header, поэтому мы можем преобразовать значение к типу HTMLElement. Однако если такого элемента нет, то во время выполнения мы опять же получим ошибку.

## 2.11.Массивы

Массивы определяются с помощью выражения [] и также являются строго типизированными. То есть если изначально массив содержит строки, то в будущем он сможет работать только со строками.

|  |  |
| --- | --- |
| 1  2  3  4 | let list: number[] = [10, 20, 30];  let colors: string[] = ["red", "green", "blue"];  console.log(list[0]);  console.log(colors[1]); |

Как и в JavaScript, с помощью индексов можно обращаться к элементам массива.

Альтернативный способ определения массивов представляет применение типа Array<>, где в фигурных скобках указывается тип элементов массива:

|  |  |
| --- | --- |
| 1  2 | let names: Array<string> = ["Tom", "Bob", "Alice"];  console.log(names[1]);  // Bob |

Но фактически такие формы массивов, как number[] или string[] являются сокращением соответственно типов Array<number> или Array<string>

В остальном массивы в TypeScript поддерживают все те же операции, что и [массивы в JavaScript](https://metanit.com/web/javascript/5.3.php).

### **ReadonlyArray**

Массивы позволяют изменять значения своих элементов:

|  |  |
| --- | --- |
| 1  2  3 | const people = ["Tom", "Bob", "Sam"];  people[1] = "Kate";  console.log(people[1]); // Kate |

Но TypeScript также позволяет определять массивы, элементы которых нельзя изменять. Для этого применяется тип **ReadonlyArray<>**, для которого в угловых скобках указывается тип элементов массива.

В отличие от типа Array для типа ReadonlyArray мы не можем принимать конструктор, как в следующем случае:

|  |  |
| --- | --- |
| 1 | const people: ReadonlyArray<string> = new ReadonlyArray("Tom", "Bob", "Sam"); |

Вместо этого необходимо передать значения в виде обычного массива:

|  |  |
| --- | --- |
| 1 | const people: ReadonlyArray<string> = ["Tom", "Bob", "Sam"]; |

Для определения подобных массивов также можно использовать сокращение типа - readonly Тип\_данных[]:

|  |  |
| --- | --- |
| 1 | const people: readonly string[]= ["Tom", "Bob", "Sam"]; |

Массив ReadonlyArray поддерживаtт большинство тех же операций, что и обычные массивы, за тем исключением операций, которые изменяют массив и его элементы. Так, мы не можем менять отдельные значения:

|  |  |
| --- | --- |
| 1  2 | const people: ReadonlyArray<string> = ["Tom", "Bob", "Sam"];  people[1] = "Kate";     // ! Ошибка элементы массива ReadonlyArray нельзя изменить |

Также мы не можем добавлять новые или удалять уже имеющиеся элементы:

|  |  |
| --- | --- |
| 1  2  3  4 | const people: ReadonlyArray<string> = ["Tom", "Bob", "Sam"];    people.push("Kate");    // ! Ошибка -  нельзя добавить новые элементы  people.pop();           // ! Ошибка -  нельзя удалить существующие элементы |

Более того при компиляции компилятор сообщит нам, что для типа ReadonlyArray не определены методы push() и pop(). Это относится ко всем операциям, которые изменяют массив.

Все остальные операции, которые предусматривают чтение массива, мы по прежнему можем использовать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function printUsers(users: readonly string[]) {      for(const user of users){          console.log(user);      }  }    function usersToString(users: ReadonlyArray<string>): String{        return users.join(", ");  }    const people: readonly string[]= ["Tom", "Bob", "Sam"];    printUsers(people);  console.log(usersToString(people)); |

### **Декомпозиция массивов**

Как и в javascript, массивы поддерживают декомпозицию на константы и переменные. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const people: string[]= ["Tom", "Bob", "Sam"];    const [first, second, third] = people;  console.log(first);     // Tom  console.log(second);    // Bob  console.log(third);     // Sam |

Здесь массив people раскладывается на три константы - first, second, third. Значения массива передаются константам/переменным по позиции, то есть первой константе - первый элемент массива, второй константе - второй элемент и так далее. Причем констант/переменных не больше, чем элементов массива.

С помощью оператора **...** можно указать массив, в который будут помещаться все оставшиеся элементы раскладываемого массива, которые не вошли в предыдущие переменные/константы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const people: string[]= ["Tom", "Bob", "Sam"];    const [first, ...rest] = people;  console.log(first);     // Tom  console.log(rest[0]);   // Bob  console.log(rest[1]);       // Sam |

Здесь константа first принимает первый элемент массива - "Tom". Все оставшиеся элементы кортежа будут помещаться в массив rest.

Также можно оставить пустое место вместо переменной/константы, если мы хотим пропустить соответствующий элемент массива:

|  |  |
| --- | --- |
| 1  2  3  4  5 | const people: string[]= ["Tom", "Bob", "Sam", "Kate"];    const [, second, , forth] = people; // пропускаем первый и третий элементы массива  console.log(second);        // Bob  console.log(forth);         // Kate |

## 2.12.Кортежи

Кортежи (Tuples) также, как и массивы, представляют набор элементов, для которых уже заранее известен тип. В отличие от массивов кортежи могут хранить значения разных типов. Для определения кортежа применяется синтаксис массива:

|  |  |
| --- | --- |
| 1  2 | // определение кортежа - кортеж состоит из двух элементов - строки и числа  let user: [string, number]; |

В данном случае кортеж user представляет тип [string, number], то есть такой кортеж, который состоит из двух элементов, при чем первый элемент представляет тип string, а второй элемент - тип number.

Для присвоения значения применяется массив:

|  |  |
| --- | --- |
| 1 | user = ["Tom", 28]; |

Причем передаваемые кортежу данные должны соответствовать элементам по типу. Например, ниже неправильный пример инициализации кортежа;

|  |  |
| --- | --- |
| 1  2 | // Неправильная инициализация - переданные значения не соответствуют типам по позиции  //userInfo = [28, "Tom"]; // Ошибка |

Для обращения к элементам кортежа, так же как и для массива, применяются индексы:

|  |  |
| --- | --- |
| 1  2  3  4 | let user: [string, number] = ["Tom", 36];  console.log(user[1]); // 36  user[1] = 37;  console.log(user[1]); // 37 |

С помощью цикла for можно перебрать элементы кортежа:

|  |  |
| --- | --- |
| 1  2  3  4 | let user: [string, number] = ["Tom", 36];  for(const prop of user){      console.log(prop);  } |

### **Кортежи в функциях**

Кортежи как параметры функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function printUser(user: [string, number]) {      console.log(user[0]);      console.log(user[1]);  }  let tom: [string, number] = ["Tom", 36];  printUser(tom); |

Кортеж как результат функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function createUser(name: string, age: number) : [string, number]{        return [name, age];  }  let user = createUser("Bob", 41);  console.log(user[0]);  console.log(user[1]); |

### **Необязательные элементы кортежей**

Кортежи могут иметь необязательные элементы, для которых можно не предоставлять значение. Чтобы указать, что элемент является необязательным, после типа элемента ставится вопросительный знак ?:

|  |  |
| --- | --- |
| 1  2 | let bob: [string, number, boolean?] = ["Bob", 41, true];  let tom: [string, number, boolean?] = ["Tom", 36]; |

В данном случае последний элемент, который представляет тип boolean, необязательный. Причем необязательные элементы должны идти в самом конце - после обязательных элементов.

Кортеж с необязательными элементами как параметр функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function printUser(user: [string, number, boolean?]) {        if(user[2]!==undefined){          console.log(`name: ${user[0]}  age: ${user[1]}  isMarried: ${user[2]}`);      }      else{          console.log(`name: ${user[0]}  age: ${user[1]}`);      }  }    let bob: [string, number, boolean] = ["Bob", 41, true];  let tom: [string, number] = ["Tom", 36];    printUser(bob);  printUser(tom); |

Стоит отметить, что здесь кортеж tom представляет тип [string, number] и также соответствует типу [string, number, boolean?], поскольку последний элемент является необязательным.

### **Заполнение кортежа**

С помощью оператора **...** внутри определения типа кортежа можно определить набор элементов, количество которых неопределено. Например:

|  |  |
| --- | --- |
| 1  2 | let math: [string, ...number[]] = ["Math", 5, 4, 5, 4, 4];  let physics: [string, ...number[]] = ["Physics", 5, 5, 5]; |

В данном случае оба кортежа представляют тип [string, ...number[]]. То есть первый элемент кортежа должен представлять тип string. А остальные элементы кортежа должны представлять тип number, причем таких элементов может быть неопределенное количество.

Кортеж с неопределенным количеством элементов в качестве параметра функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function printMarks(marks: [string, ...number[]]){        for(const mark of marks){          console.log(mark);      }  }  let math: [string, ...number[]] = ["Math", 5, 4, 5, 4, 4];  let physics: [string, ...number[]] = ["Physics", 5, 5, 5];    printMarks(math);  printMarks(physics); |

При этом неопределенное количество элементов можно определять как в конце, так и в середине и в начале кортежа:

|  |  |
| --- | --- |
| 1  2  3 | let math: [string, ...number[]] = ["Math", 5, 4, 5, 4, 4];  let physics: [...number[], string] = [5, 5, 5, "Physics"];  let chemistry: [string, ...number[], boolean] = ["Chemistry", 3, 3, 4, 5, false]; |

В случае с кортежем [...number[], string] он должен оканчиваться на элемент типа string, перед которым может быть неопределенное количество элементов типа number.

А в кортеже типа [string, ...number[], boolean] первый элемент должен представлять тип string, а последний - тип boolean. Между ними может быть неопределенное количество элементов типа number.

### **Кортеж для чтения**

Стандартные кортежи позволяют изменять значения их элементов:

|  |  |
| --- | --- |
| 1  2  3 | const tom: [string, number] = ["Tom", 36];  tom[1] = 37;  console.log(tom[1]);    // 37 |

Однако TypeScript также позволяет создавать кортежи только для чтения, элементы которого нельзя изменить. Для этого перед типом кортежа указывается ключевое слово **readonly**:

|  |  |
| --- | --- |
| 1  2 | const tom: readonly [string, number] = ["Tom", 36];  tom[1] = 37; // ! Ошибка - элементы кортежа для чтения нельзя изменять |

Кортеж для чтения в качестве параметра функции:

|  |  |
| --- | --- |
| 1  2  3  4 | function printUser(user: readonly [string, number]) {        console.log(`name: ${user[0]}  age: ${user[1]}`);  } |

Кортеж для чтения в качестве результата функции:

|  |  |
| --- | --- |
| 1  2  3  4 | function generateUser(): readonly [string, number]{        return ["Sam", 18];  } |

### **Декомпозиция кортежа**

Кортеж, как и массив, можно раскладывать на переменные и константы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let tom: [string, number, boolean] = ["Tom", 36, false];  let [username, userage, isMarried] = tom;   // декомпозиция  console.log(username);      // Tom  console.log(userage);       // 36  console.log(isMarried);     // false |

Здесь кортеж tom раскладывается на три переменных: username, userage, isMarried. При этом количество переменных должно соответствовать количеству элементов кортежа.

С помощью оператора **...** можно указать кортеж, в который будут помещаться все оставшиеся элементы раскладываемого кортежа, которые не вошли в предыдущие переменные или константы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let tom: [string, number, boolean] = ["Tom", 36, false];  const [username, ...rest] = tom;  console.log(username);      // Tom  console.log(rest[0]);       // 36  console.log(rest[1]);       // false |

Здесь константа username также принимает первый элемент кортежа - "Tom". Все оставшиеся элементы кортежа будут помещаться в кортеж rest.

Мы можем определить меньше переменных/констант, чем элементов в кортеже, тогда оставшиеся элементы кортежа будут игнорироваться:

|  |  |
| --- | --- |
| 1  2  3  4 | let tom: [string, number, boolean] = ["Tom", 36, false];  const [username, userage] = tom;  console.log(username);      // Tom  console.log(userage);       // 36 |

Также можно оставить пустое место вместо переменной/константы, если мы хотим пропустить соответствующий элемент кортежа:

|  |  |
| --- | --- |
| 1  2  3  4 | let tom: [string, number, boolean, number] = ["Tom", 36, false, 170];  const [, age, , height] = tom;      // пропускаем первый и третий элементы  console.log(age);           // 36  console.log(height);        // 170 |

## 2.13.Неопределенный набор и наполнение параметров

### **Неопределенный набор параметров**

Если необходимо, чтобы функция принимала набор однотипных параметров, то используется знак многоточия, после которого идет массив:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function addNumbers(firstNumber: number, ...numberArray: number[]): number {        let result = firstNumber;      for (let i = 0; i < numberArray.length; i++) {          result+= numberArray[i];      }      return result;  }    let num1 = addNumbers(3, 7, 8);  console.log(num1); // 18    let num2 = addNumbers(3, 7, 8, 9, 4);  console.log(num2); // 31 |

В данном случае функция addNumbers фиксированно имеет только один параметр - firstNumber, остальные параметры необязательны и передаются через массив numberArray. Причем поскольку этот массив представляет тип number[] - массив чисел, то при вызове функции для этого параметра мы можем передать нефиксированный набор значений, которые должны представлять тип number.

### **Наполнение параметров**

TypeScript позволяет использовать массивы для передачи данных сразу нескольким параметрам. Рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function sum(...args: number[]): number {        let result = 0;      for (let i = 0; i < args.length; i++) {          result+= args[i];      }      return result;  }  const numbers = [1, 3, 5, 7, 9] as const;  let num = sum(...numbers);  console.log(num); // 25 |

Здесь функция sum() через массив args принимает набор необязательных параметров. Этих параметров может быть неопределенное количество - ноль, один, два, три...

Для передачи этим параметрам применяется оператор **...**, после которого указывается массив значений:

|  |  |
| --- | --- |
| 1 | let num = sum(...numbers); |

То есть это аналогично тому, если бы мы написали

|  |  |
| --- | --- |
| 1 | let num = sum(1, 3, 5, 7, 9); |

Однако не всякий массив можно так передать. Обратите внимания, что для массива используется выражение **as const**:

|  |  |
| --- | --- |
| 1 | const numbers = [1, 3, 5, 7, 9] as const; |

Теперь возьмем функцию с фиксированным набором параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function sum(a: number, b: number): number {        return a + b;  }  const numbers = [1, 3, 5, 7, 9] as const;  let num = sum(...numbers);  // ! Ошибка - функция sum принимает только два параметра  console.log(num); // 25 |

При компиляции данного кода мы получим ошибку. Поскольку функция sum() теперь четко определяет два параметра. Однако при вызове ей передается пять значений - [1, 3, 5, 7, 9]. И в данном случае нам надо передать массив, который имеет столько элементов, сколько функция определяет обязательных параметров:

|  |  |
| --- | --- |
| 1  2 | const numbers = [1, 3] as const;  let num = sum(...numbers); |

Если функция имеет необязательные параметры, то передаваемый массив значений может содержать для них значения, а может и не содержать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function sum(a: number, b: number, c?: number): number {        let result = a + b;      if(c!==undefined) { result +=c;}      return result;  }  const numbers1 = [1, 3] as const;  let num1 = sum(...numbers1);    // a = 1  b = 3  console.log(num1); // 4    const numbers2 = [1, 3, 7] as const;  let num2 = sum(...numbers2);    // a = 1  b = 3  c = 7  console.log(num2); // 11 |

При этом мы можем таким образом передавать значения разных типов в виде кортежа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function printValues(name: string, age: number){        console.log(name);      console.log(age);  }  const values = ["Tom", 36] as const;  printValues(...values); // name = "Tom", age = 36 |

В данном случае функция printValues() в качестве первого значения принимает значение string, а в качестве второго - значение типа number. Набор values как представляет набор значений подобных типов - ["Tom", 36]. В итоге первое значение из кортежа values сопоставляется с первым параметром функции, а второе значение - со вторым параметрам. Важно, что передаваемые значения соответствовали по типам параметрам по позиции. Например, в следующем случае мы получим ошибку:

|  |  |
| --- | --- |
| 1  2 | const values = [36, "Tom"] as const;  printValues(...values); |

Так как первое значение из набора values представляет тип number, а первый параметр функции - тип string. Соответственно получаем несоответствие по типу данных.

## 2.14.Перечисление enum

Тип **enum** или перечисление позволяет определить набор именнованных констант, которые описывают определенные состояния.

Для определения перечисления применяется ключевое слово **enum**. Например, объявим следующее перечисление:

|  |  |
| --- | --- |
| 1 | enum Season { Winter, Spring, Summer, Autumn }; |

Перечисление называется Season и имеет четыре элемента. Теперь используем перечисление:

|  |  |
| --- | --- |
| 1  2  3  4 | enum Season { Winter, Spring, Summer, Autumn };  let current: Season = Season.Summer;  console.log(current);       // 2  current = Season.Autumn;    // изменение значения |

Здесь создается переменная current, которая имеет тип Season. При этом консоль браузера выведет нам число 2 - значение константы Season.Summer.

### **Числовые перечисления**

По умолчанию константы перечисления, как в примере выше, представляют числовые значения. То есть это так называемое числовое перечисление, в котором каждой константе сопоставляется числовое значение.

Так, созданное выше в примере перечисление

|  |  |
| --- | --- |
| 1 | enum Season { Winter, Spring, Summer, Autumn }; |

фактически эквивалентно следующему:

|  |  |
| --- | --- |
| 1 | enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 }; |

Хотя мы можем явным образом переопределить эти значения. Так, мы можем задать значение одной константы, тогда значения следующих констант будет увеличиваться на единицу:

|  |  |
| --- | --- |
| 1 | enum Season { Winter=5, Spring, Summer, Autumn };           // 5, 6, 7, 8 |

Либо можно каждой константе задать свое значение:

|  |  |
| --- | --- |
| 1 | enum Season { Winter=4, Spring=8, Summer=16, Autumn=32 }; // 4, 8, 16, 32 |

Также мы можем получить непосредственно текстовое значение:

|  |  |
| --- | --- |
| 1  2  3 | enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };  var current: string = Season[2];    // 2 - числовое значение Summer  console.log(current);   // Summer |

### **Строковые перечисления**

Кроме числовых перечислений в TypeScript есть строковые перечисления, константы которых принимают строковые значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | enum Season {      Winter = "Зима",      Spring = "Весна",      Summer = "Лето",      Autumn = "Осень"  };  var current: Season = Season.Summer;  console.log(current);   // Лето |

### **Смешанные гетерогенные перечисления**

Также можно определять смешанные перечисления, константы которых могут числа и строки.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | enum Season {      Winter = 1,      Spring = "Весна",      Summer = 3,      Autumn = "Осень"  };  var current: Season = Season.Summer;  console.log(current);           // 3  console.log(Season.Autumn);     // Осень |

### **Перечисления в функциях**

Перечисление может выступать в качестве параметра функции.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | enum DayTime {      Morning,      Evening  };  function welcome(dayTime: DayTime){        if(dayTime === DayTime.Morning){          console.log("Доброе утро");      }      else{          console.log("Добрый вечер");      }  }  let current: DayTime = DayTime.Morning;  welcome(current);           // Доброе утро  welcome(DayTime.Evening);   // Добрый вечер |

Каждая константа перечисления описывает некоторое состояние. И функция welcome() в виде параметра dayTime принимает это состояние и в зависимости от полученного значения выводит на консоль определенное значение.

Однако стоит отметить, что поскольку здесь перечисление DayTime представляет числовое перечисление, то в реальности в функцию welcome() мы можем передать числовые значения:

|  |  |
| --- | --- |
| 1 | welcome(1); // Добрый вечер |

Либо даже определить параметр функции как числовой и передавать константы числового перечисления:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | enum DayTime {      Morning,      Evening  };  function welcome(dayTime: number){        if(dayTime === DayTime.Morning){          console.log("Доброе утро");      }      else{          console.log("Добрый вечер");      }  }  let current: DayTime = DayTime.Morning;  welcome(current);           // Доброе утро  welcome(DayTime.Evening);   // Добрый вечер |

Пример параметра-строкового перечисления:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | enum DayTimeMessage {      Morning = "Доброе утро",      Evening = "Добрый вечер"  };  function welcome(message: DayTimeMessage){        console.log(message);  }  let mes: DayTimeMessage = DayTimeMessage.Morning;  welcome(mes);           // Доброе утро  welcome(DayTimeMessage.Evening);    // Добрый вечер |

При использовании строковых перечислений в отличие от числовых мы не можем передать переметру произвольную строку:

|  |  |
| --- | --- |
| 1 | welcome("Привет, ты спишь?") |

В этом случе компилятор выкатит нам ошибку при компиляции.

В то же время если параметр представляет тип string, то такому параметру можно передавать как строки, так и константы строкового перечисления:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | enum DayTimeMessage {      Morning = "Доброе утро",      Evening = "Добрый вечер"  };  function welcome(message: string){        console.log(message);  }  let mes: DayTimeMessage = DayTimeMessage.Morning;  welcome(mes);           // Доброе утро  welcome(DayTimeMessage.Evening);    // Добрый вечер |

# 3.Объектно-ориентированное программирование

## 3.1.Классы

TypeScript реализует объектно-ориентированный подход, в нем есть полноценная поддержка классов. Класс представляет шаблон для создания объектов и инкапсулирует функциональность, которую должен иметь объект. Класс определяет состояние и поведение, которыми обладает объект.

Для определения нового класса применяется ключевое слово class. Например, определим класс User:

|  |  |
| --- | --- |
| 1 | class User {} |

После определения класса мы можем создавать его объекты:

|  |  |
| --- | --- |
| 1  2 | let tom: User = new User();  let alice = new User(); |

Здесь определено два объекта класса User - tom и alice.

### **Поля класса**

Для хранения состояния объекта в классе определяются поля:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class User {        name: string;      age: number;  } |

Здесь определены два поля - name и age, которые имеют типы string и number соответственно. Фактически поля представляют переменные уровня класса, только для их объявления не применяются var и let.

По имени объекта мы можем обращаться к этим полям:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class User {        name: string;      age: number;  }    let tom = new User();  tom.name = "Tom";  tom.age = 36;  console.log(`name: ${tom.name}  age: ${tom.age}`);  // name: Tom  age: 36 |

При определении полей им можно задать некоторые начальные значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class User {        name: string = "Tom Smith";      age: number = 18;  }    let user = new User();  console.log(`name: ${user.name}  age: ${user.age}`);    // name: Tom Smith  age: 18 |

### **Методы**

Классы также могут определять поведение - некоторые действия, которые должны выполнять объекты этого класса. Для этого внутри класса определяются функции, которые называются методами.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class User {        name: string;      age: number;      print(){          console.log(`name: ${this.name}  age: ${this.age}`);      }      toString(): string{          return `${this.name}: ${this.age}`;      }  } |

Здесь в классе User определены два метода. Метод print() выводит информацию об объекте на консоль, а метод toString() возвращает некоторое представление объекта в виде строки.

В отличие от обычных функций для определения методов не указывается ключевое слово function.

Для обращения внутри методов к полям и другим методам класса применяется ключевое слово **this, которое указывает на текущий объект этого класса.**

Применение методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class User {        name: string;      age: number;      print(){          console.log(`name: ${this.name}  age: ${this.age}`);      }      toString(): string{          return `${this.name}: ${this.age}`;      }  }    let tom = new User();  tom.name = "Tom";  tom.age = 36;  tom.print();                    // name: Tom  age: 36    console.log(tom.toString());    // Tom: 36 |

### **Конструкторы**

Кроме обычных методов классы имеют специальные функции - конструкторы, которые определяются с помощью ключевого слова **constructor**. Конструкторы выполняют начальную инициализацию объекта. Например, добавим в класс User конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class User {        name: string;      age: number;      constructor(userName: string, userAge: number) {            this.name = userName;          this.age = userAge;      }      print(){          console.log(`name: ${this.name}  age: ${this.age}`);      }  }    let tom = new User("Tom", 36);  tom.print();        // name: Tom  age: 36 |

Здесь конструктор применимает два параметра и использует их значения для установки значения полей name и age:

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(userName: string, userAge: number) {      this.name = userName;      this.age = userAge;  } |

Затем при создании объекта в конструктор передается два значения для его параметров:

|  |  |
| --- | --- |
| 1 | let tom = new User("Tom", 36); |

### **Поля для чтения**

Полям класса в процессе работы программы можно присваивать различные значения, которые соответствуют типу полей. Однако TypeScript также позволяет определять поля для чтения, значения которых нельзя изменить (кроме как в конструкторе). Для определения таких полей применяется ключевое слово **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class User {        readonly name: string = "Default user";      age: number;      constructor(userName: string, userAge: number) {            this.name = userName;          this.age = userAge;      }      print(){          console.log(`name: ${this.name}  age: ${this.age}`);      }  } |

Значение полей для чтения можно установить либо при объявлении:

|  |  |
| --- | --- |
| 1 | readonly name: string = "Default user"; |

Либо в конструкторе:

|  |  |
| --- | --- |
| 1  2  3  4  5 | constructor(userName: string, userAge: number) {        this.name = userName;      this.age = userAge;  } |

В остальных местах программы значение этого поля нельзя изменить. Например, в следующем случае мы получим при компиляции ошибку, потому что пытаемся установить поле для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class User {        readonly name: string = "Default user";      age: number;      constructor(userName: string, userAge: number) {            this.name = userName;          this.age = userAge;      }      print(){          console.log(`name: ${this.name}  age: ${this.age}`);      }  }    let tom = new User("Tom", 36);  tom.name = "Bob";       // ! Ошибка - поле name - только для чтения  tom.print();                    // name: Tom  age: 36 |

## 3.2.Наследование

Одним из ключевых моментов объектно-ориентированной парадигмы является наследование. В TypeScript наследование реализуется с помощью ключевого слова **extends** (как в Java):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }      print(): void {          console.log(`Имя: ${this.name}`);      }  }    class Employee extends Person {        company: string;      work(): void {          console.log(`${this.name} работает в компании ${this.company}`);      }  } |

Класс Employee, который представляет работника, является подклассом или наследуется от класса Person. А класс Person называется родительским или базовым классом. При наследовании класс Employee перенимает весь функционал класса Person - все его свойства и функции и может их использовать. И также можно определить в подклассе новые свойства и методы, которых нет в классе Person.

|  |  |
| --- | --- |
| 1  2 | let bob: Employee = new Employee("Bob");  bob.print(); |

Также мы можем расширить функциональность класса следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }      print(): void {          console.log(`Имя: ${this.name}`);      }  }    class Employee extends Person {        company: string;      work(): void {          console.log(`${this.name} работает в компании ${this.company}`);      }  }    let bob: Employee = new Employee("Bob");  bob.print();  bob.company = "Microsoft";  bob.work(); |

### **Переопределение конструктора**

Если подкласс определяет свой конструктор, то в нем должен быть вызван конструктор базового класса с помощью ключевого слова **super**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }      print(): void {          console.log(`Имя: ${this.name}`);      }  }    class Employee extends Person {        company: string;      constructor(name: string, company: string) {            super(name);          this.company = company;      }      work(): void {          console.log(`${this.name} работает в компании ${this.company}`);      }  }    let bob: Employee = new Employee("Bob", "Microsoft");  bob.work();     // Bob работает в компании Microsoft |

С помощью ключевого слова **super** подкласс может обратиться к функционалу базового класса. В данном случае идет обращение к конструктору класса Person, который устанавливает значение свойства name: super(name)

Причем даже если базовый класс не определяет явным образом никакого конструктора, в производном классе при определении конструктора все равно надо вызывать конструктор базового класса - в этом случае это будет вызов конструктора по умолчанию с помощью super().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class Person {      name: string;  }    class Employee extends Person {        company: string;      constructor(name: string, company: string) {            super();    // вызов конструктора базового класса          this.name = name;          this.company = company;      }      work(): void {          console.log(`${this.name} работает в компании ${this.company}`);      }  }    let bob: Employee = new Employee("Bob", "Microsoft");  bob.work();     // Bob работает в компании Microsoft |

### **Переопределение методов**

Также производные классы могут переопределять методы базовых классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class Person {        name: string;      constructor(name: string) {            this.name = name;      }      print(): void {          console.log(`Имя: ${this.name}`);      }  }    class Employee extends Person {        company: string;      constructor(name: string, company: string) {            super(name);          this.company = company;      }      print(): void {          console.log(`Имя: ${this.name}`);          console.log(`Работает в компании: ${this.company}`);      }  }    let bob: Employee = new Employee("Bob", "Microsoft");  bob.print(); |

В данном случае переопределяется метод print(), который кроме имени выводит также компанию сотрудника. Однако в данном случае реализация метода print() из базового класса повторяется в производном классе. И вместо того, чтобы дублировть код, мы можем с помощью ключевого слова **super** вызвать реализацию этого метода из базового класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class Person {        name: string;      constructor(name: string) {            this.name = name;      }      print(): void {          console.log(`Имя: ${this.name}`);      }  }    class Employee extends Person {        company: string;      constructor(name: string, company: string) {            super(name);          this.company = company;      }      print(): void {          super.print();          console.log(`Работает в компании: ${this.company}`);      }  }    let bob: Employee = new Employee("Bob", "Microsoft");  bob.print(); |

## 3.3.Абстрактные классы, методы и поля

### **Абстрактные классы**

Абстрактные классы представляют классы, определенные с ключевым словом **abstract**. Они во многом похожи на обычные классы за тем исключением, что мы не можем создать напрямую объект абстрактного класса, используя его конструктор.

|  |  |
| --- | --- |
| 1  2  3  4 | abstract class Figure {  }    // let someFigure = new Figure()    // Ошибка! |

Как правило, абстрактные классы описывают сущности, которые в реальности не имеют конкретного воплощения. Например, геометрическая фигура может представлять круг, квадрат, треугольник, но как таковой геометрической фигуры самой по себе не существует. Есть конкретные фигуры, с которыми мы и работаем. В то же время все фигуры могут иметь какой-то общий функционал. В этом случае мы можем определить абстрактный класс фигуры, поместить в него общий функционал, и от него унаследовать классы конкретных геометрических фигур:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | abstract class Figure {      getArea(): void{          console.log("Not Implemented")      }  }  class Rectangle extends Figure{        constructor(public width: number, public height: number){          super();      }        getArea(): void{          let square = this.width \* this.height;          console.log("area =", square);      }  }    let someFigure: Figure = new Rectangle(20, 30)  someFigure.getArea();   // area = 600 |

В данном случае абстрактный класс определяет метод getArea(), который вычисляет площадь фигуры. Класс прямоугольника определяет свою реализацию для этого метода.

### **Абстрактные методы**

Однако в данном случае метод getArea в базовом классе не выполняет никакой полезной работы, так как у абстрактной фигуры не может быть площади. И в этом случае подобный метод лучше определить как **абстрактный**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | abstract class Figure {      abstract getArea(): void;  }  class Rectangle extends Figure{        constructor(public width: number, public height: number){          super();      }        getArea(): void{          let square = this.width \* this.height;          console.log("area =", square);      }  }    let someFigure: Figure = new Rectangle(20, 30)  someFigure.getArea(); |

Абстрактный метод не определяет никакой реализации. Если класс содержит абстрактные методы, то такой класс должен быть абстрактным. Кроме того, при наследовании производные классы обязаны реализовать все абстрактные методы.

### **Абстрактные поля**

Также абстрактный класс может иметь абстрактные поля, то есть поля определенные с модификатором **abstract**. При наследовании класс-наследник также обязан предоставить для них реализацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | abstract class Figure {      abstract x: number;      abstract y: number;      abstract getArea(): void;  }  class Rectangle extends Figure{      //x: number;      //y: number;        constructor(public x: number, public y: number, public width: number, public height: number){          super();      }        getArea(): void{          let square = this.width \* this.height;          console.log("area =", square);      }  }    let someFigure: Figure = new Rectangle(10, 10, 20, 25)  someFigure.getArea(); |

В данном случае класс Figure определяет два абстрактных поля x и y, которые условно представляют начальную точку фигуры:

|  |  |
| --- | --- |
| 1  2 | abstract x: number;  abstract y: number; |

Класс Rectangle предоставляет для них реализацию с помощью определения полей через параметры конструктора:

|  |  |
| --- | --- |
| 1 | constructor(public x: number, public y: number, public width: number, public height: number) |

По большому счету в данном случае большого смысла в абстрактных полях нет, тем не менее TypeScript позволяет их применять.

## 3.4.Модификаторы доступа

Модификаторы доступа позволяют сокрыть состояние объекта от внешнего доступа и управлять доступом к этому состоянию. В TypeScript три модификатора: **public**, **protected** и **private**.

Если к свойствам и функциям классов не применяется модификатор, то такие свойства и функции расцениваются как будто они определены с модификатором public. То есть следующее определение класса:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person {        name: string;      year: number;  } |

Будет эквивалентно:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person {        public name: string;      public year: number;  } |

### **private**

Если же к свойствам и методам применяется модификатор private, то к ним нельзя будет обратиться извне при создании объекта данного класса.

Например, создадим класс с приватными свойствами и методами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Person {        private \_name: string;      private \_year: number;        constructor(name: string, age: number) {            this.\_name = name;          this.\_year = this.setYear(age);      }      public print(): void {          console.log(`Имя: ${this.\_name}  Год рождения: ${this.\_year}`);      }        private setYear(age: number): number {            return new Date().getFullYear() - age;      }  }    let tom = new Person("Tom", 24);  tom.print();  // console.log(tom.\_name); // нельзя обратиться, так как \_name - private  // tom.setYear(45); // нельзя обратиться, так как функция - private |

Два свойства \_name и \_year используются с модификатором private, поэтому мы не можем их использовать вне класса, например, console.log(tom.\_name). То же самое относится к функции setYear(). Остальные функции доступны.

### **protected**

Модификатор **protected** определяет поля и методы, которые из вне класса видны только в классах-наследниках:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class Person {      protected name: string;      private year: number;      constructor(name: string, age: number) {            this.name = name;          this.year = this.setYear(age);      }      protected printPerson(): void {            console.log(`Имя: ${this.name}  Год рождения: ${this.year}`);      }      private setYear(age: number): number {            return new Date().getFullYear() - age;      }  }  class Employee extends Person {        protected company: string;      constructor(name: string, age: number, company: string) {          super(name, age);          this.company = company;      }      public printEmployee(): void {          //console.log("Year: " + this.year);    // поле year недоступно, так как private          // setYear(25);                         // метод setYear недоступен, так как private          this.printPerson();                     // метод printPerson доступен, так как protected          console.log(`Компания: ${this.company}`);      }  }    let sam = new Employee("Sam", 31, "Microsoft");  sam.printEmployee(); |

В классе Employee будет доступно поле name, а также метод printPerson(), так как они имеют модификатор protected. А вот приватное свойство year и приватный метод setYear() будут недоступны.

### **Определение полей через конструктор**

Использование модификаторов в параметрах конструктора позволяет сократить написание кода. Например, пусть у нас есть следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Person {        private name: string;      private age: number;        constructor(name: string, age: number) {            this.name = name;          this.age = age;      }      printPerson(): void {            console.log(`Имя: ${this.name}  Возраст: ${this.age}`);      }  } |

Этот класс будет аналогичен следующему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person {        constructor(private name: string, private age: number) {  }        printPerson(): void {            console.log(`Имя: ${this.name}  Возраст: ${this.age}`);      }  } |

Используя модификаторы в параметрах конструктора, нам больше не надо явно создать свойства для этих параметров. Свойства создаются автоматически, называются они по имени параметров и имеют те же модификаторы, что и параметры.

Подобным образом, если мы хотим сделать свойства публичными, то следует использовать модификатор **public**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person {        constructor(public name: string, public age: number) {  }        printPerson(): void {            console.log(`Имя: ${this.name}  Возраст: ${this.age}`);      }  } |

Если необходимо сделать поле для чтения, то к модификатору доступа добавляется модификатор **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4 | class Person {        constructor(private readonly name: string, private age: number) {  }  } |

**3.5.Методы доступа get и set**

В стандарте JavaScript ECMAScript 5 была предложена концепция методов доступа: для доступа к свойству определяется пара методов - get-метод для получения значения свойства и set-метод для установки значения. Это довольно распространенная концепция, которая нашла свое применение, например, в Java, где для управления доступом к приватным переменным создается пара методов - get/set, или в C#, где для доступа к приватным переменным создается свойство с двумя методами get/set.

К примеру, у нас есть следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person {      name: string;      age: number;  }    let tom = new Person();  tom.name = "Tom";  tom.age = -1243;  console.log(tom.age); |

Здесь полю age, которое представляет условный возраст человека, присваивается некорретное значение.

Использование аксессоров или методов доступа позволяет управлять тем, как значение устанавливается и как оно возвращается. В частности, мы можем переписать предыдущих класс с использованием акссессоров следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Person {      name: string;      private \_age: number;      private \_name: string;        public get age(): number {          return this.\_age;      }        public set age(n: number) {          if(n < 0 || n > 110){              console.log("Недопустимый возраст!");          }          else{              this.\_age = n;          }      }  }    let tom = new Person();  tom.name = "Tom";  tom.age = 36;  console.log(tom.age);   // 36  tom.age = -1243;        // Недопустимый возраст!  console.log(tom.age);   // 36 |

Методы доступа определяются как обычные методы, только перед ними ставятся ключевые слова **get/set**. Set-метод контроллирует установку значения, а get-метод возвращает значение.

В данном случае мы опосредовали доступ к переменной \_age, сделав ее приватной. Теперь к ней можно обратиться только через методы доступа. Метод get просто возвращает значение переменной:

|  |  |
| --- | --- |
| 1  2  3 | public get age(): number {          return this.\_age;      } |

А метод set устанавливает значение \_age, но только в том случае, если оно укладывается в корректный диапазон.

Когда мы присваиваем значение:

|  |  |
| --- | --- |
| 1 | tom.age = 36; |

срабатывает set-метод

Когда мы получаем значение:

|  |  |
| --- | --- |
| 1 | console.log(tom.age); |

срабатывает get-метод

**3.6.Статические поля и методы**

Кроме обычных полей и методов класс может иметь статические. Статические поля и методы относятся не к отдельным объектам, а в целом к классу. И для обащения к статическим полям и методам применяется имя класса.

Статические поля и методы определяются с помощью ключевого слова **static**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Person {        age: number;      name: string;        static retirementAge: number = 65;      static calculateYears(age: number): number{            return Person.retirementAge - age;      }        constructor(name: string, age: number) {          this.name = name;          this.age = age;      }  }    let tom = new Person("Tom", 36);  let years = Person.calculateYears(36);  console.log(Person.retirementAge);  console.log(`До пенсии осталось: ${years} лет`); |

Здесь в классе Person определено статическое поле retirementAge, которое условно хранит возраст выхода на пенсию для объектов Person. Поскольку это свойство будет общим для всех объектов Person, то имеет смысл сделать его статическим, то есть общим для всего класса, как в данном случае.

Для обращения к этому полю применяется имя класса:

|  |  |
| --- | --- |
| 1 | console.log(Person.retirementAge); |

Также определен статический метод calculateYears(), который условно вычисляет оставшее до пенсии врем (для упрощения задачи условимся, что передаваемый возраст меньше возраста выхода на пенсию). И также для обращения к этому методу применяется имя класса, а не объекта:

|  |  |
| --- | --- |
| 1 | let years = Person.calculateYears(36); |

Стоит отметить, что в статических методах мы можем обращаться к статическим полям или другим статическим методам класса, но мы не можем обращаться к нестатическим полям и методам и использовать ключевое слово this. Например, в следующем случае мы столкнемся с ошибкой при компиляции:

|  |  |
| --- | --- |
| 1  2  3  4 | static calculateYears(): number{        return Person.retirementAge - this.age; // ! Ошибка - this.age - обращение к нестатическому полю  } |

Как и обычные поля и методы статические могут иметь модификаторы доступа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person {        age: number;      name: string;        private static retirementAge: number = 65;      static calculateYears(age: number): number{            return Person.retirementAge - age;      }        constructor(name: string, age: number) {          this.name = name;          this.age = age;      }  } |

Также статические поля и методы могут наследоваться, что позволяет обращаться к ним через имя производного класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class Person {        age: number;      name: string;        static retirementAge: number = 65;      static calculateYears(age: number): number{            return Person.retirementAge - age;      }        constructor(name: string, age: number) {          this.name = name;          this.age = age;      }  }  class Employee extends Person {}    let years = Employee.calculateYears(36);  console.log(Employee.retirementAge); |

## 3.7.Интерфейсы

### **Интерфейсы объектов**

Интерфейс определяет свойства и методы, которые объект должен реализовать. Другими словами, интерфейс - это определение кастомного типа данных, но без реализации. В данном случае интерфейсы в TS похожи на интерфейсы в языках Java и C#. Интерфейсы определяются с помощью ключевого слова **interface**. Для начала определим простенький интерфейс:

|  |  |
| --- | --- |
| 1  2  3  4 | interface IUser {      id: number;      name: string;  } |

Интерфейс в фигурных скобках определяет два свойства: id, которое имеет тип number, и name, которая представляет строку. Теперь используем его в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let employee: IUser = {        id: 1,      name: "Tom"  }  console.log("id: ", employee.id);  console.log("name: ", employee.name); |

По сути employee - обычный объект за тем исключением, что он имеет тип IUser. Если правильнее говорить, то employee реализует интерфейс IUser. Причем эта реализация накладывает на employee некоторые ограничения. Так, employee должен реализовать все свойства и методы интерфейса IUser, поэтому при определении employee данный объект обязательно должен включать в себя свойства id и name.

Параметры методов и функций также могут представлять интерфейсы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | interface IUser {      id: number;      name: string;  }  let employee: IUser = {        id: 1,      name: "Alice"  }    function printUser(user: IUser): void {        console.log("id: ", user.id);      console.log("name: ",  user.name)  }    printUser(employee); |

В этом случае аргумент, который передается в функцию, должен представлять объект или класс, который реализует соответствующий интерфейс.

И также можно возвращать объекты интерфейса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface IUser {      id: number;      name: string;  }  function buildUser(userId: number, userName: string): IUser {        return { id: userId, name: userName };  }    let newUser = buildUser(2, "Bill");  console.log("id: ", newUser.id);  console.log("name: ", newUser.name); |

### **Необязательные свойства и свойства только для чтения**

При определении интерфейса мы можем задать некоторые свойства как необязательные с помощью знака вопроса. Подобные свойства реализовать необязательно:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface IUser {      id: number;      name: string;      age?: number;  }  let employee: IUser = {        id: 1,      name: "Alice",      age: 23  }  let manager: IUser = {        id: 2,      name: "Tom"  } |

Свойство age помечено как необязательное, поэтому его можно не определять в объектах.

Также интерфейс может содержать свойства только для чтения, значение которых нельзя изменять. Такие свойства определяются с помощью ключевого слова **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface Point {      readonly x: number;      readonly y: number;  }  let p: Point = { x: 10, y: 20 };  console.log(p);  // p.x = 5; // Ошибка - свойство доступно только для чтения |

### **Определение методов**

Кроме свойств интерфейсы могут определять функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface IUser {      id: number;      name: string;      sayWords(words: string): void;  }  let employee: IUser = {        id: 1,      name: "Alice",      sayWords: function(words: string): void{          console.log(`${name} говорит "${words}"`);      }  }    employee.sayWords("Привет, как дела?"); |

Опять же объект, который реализует интерфейс, также обязан реализовать определенную в интерфейсе функцию с тем же набором параметров и тем типом выходного результата. В данном случае функция sayWords() в качестве параметра принимает строку и ничего возвращает, выводя на консоль некоторое сообщение.

### **Интерфейсы классов**

Интерфейсы могут быть реализованы не только объектами, но и классами. Для этого используется ключевое слово **implements**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | interface IUser {      id: number;      name: string;      getFullName(surname: string): string;  }    class User implements IUser{        id: number;      name: string;      age: number;      constructor(userId: number, userName: string, userAge: number) {            this.id = userId;          this.name = userName;          this.age = userAge;      }      getFullName(surname: string): string {            return this.name + " " + surname;      }  }    let tom = new User(1, "Tom", 23);  console.log(tom.getFullName("Simpson")); |

Класс User реализует интерфейс IUser. В этом случае класс User обязан определить все те же свойства и функции, которые есть в IUser.

При этом объект tom является как объектом User, так и объектом IUser:

|  |  |
| --- | --- |
| 1  2  3 | let tom :IUser = new User(1, "Tom", 23);  //или  let tom :User = new User(1, "Tom", 23); |

### **Расширение интерфейса**

TypeScript позволяет добавлять в интерфейс новые поля и методы, просто объявив интерфейс с тем же именем и определив в нем необходимые поля и методы. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | interface IUser {      id: number;      name: string;  }  interface IUser{      age: number;  }  let employee: IUser = {        id: 1,      name: "Alice",      age: 31  }    function printUser(user: IUser): void {        console.log(`id: ${user.id}  name: ${user.name}  age: ${user.age}`);  }    printUser(employee); |

В данном случае первое определение интерфейса IUser содержит поля id и name. Второе определение интерфейса содержит объявление поля age. В итоге объект или класс, который реализует этот интерфейс, должен определить все три поля - id, name и age.

### **Наследование интерфейсов**

Интерфейсы, как и классы, могут наследоваться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | interface IMovable {        speed: number;      move(): void;  }  interface ICar extends IMovable {        fill(): void;  }  class Car implements ICar {        speed: number;      move(): void {            console.log("Машина едет со скоростью " + this.speed + " км/ч");      }        fill(): void {            console.log("Заправляем машину топливом");      }  }    let auto = new Car();  auto.speed = 60;  auto.fill();  auto.move(); |

После наследования интерфейс ICar будет также иметь все те свойства и функции, которые определены в IMovable. И тогда, класс Car, реализующий интерфейс ICar, должен будет реализовать также и свойства и методы интерфейса IMovable.

### **Интерфейсы функций**

Интерфейсы функций содержат определение типа функции. Затем они должны быть реализованы объектом, который представляет функцию данного типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | interface FullNameBuilder {      (name: string, surname: string): string;  }    let simpleBuilder: FullNameBuilder = function (name:string, surname: string): string {          return "Mr. " + name + " " + surname;  }    let fullName = simpleBuilder("Bob", "Simpson");  console.log(fullName); // Mr. Bob Simpson |

Здесь определен интерфейс FullNameBuilder, который лишь содержит сигнатуру функции. Далее определяется переменная simpleBuilder, которая имеет тип FullNameBuilder и поэтому должна представлять функцию с данной сигнатурой.

### **Интерфейсы массивов**

Интерфейсы массивов описывают объекты, к которым можно обращаться по индексу, как, например, к массивам

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface StringArray {      [index: number]: string;  }    let phones: StringArray;  phones = ["iPhone 7", "HTC 10", "HP Elite x3"];    let myPhone: string = phones[0];  console.log(myPhone); |

Здесь определен интерфейс StringArray, который содержит сигнатуру массива. Эта сигнатура указывает, что объект, который реализует StringArray, может индексироваться с помощью чисел (объекта типа number). И, кроме того, данный объект должен хранить объекты типа string, то есть строки.

Выше индекс представлял тип number. Но мы можем использовать для индексации и тип string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | interface Dictionary {      [index: string]: string;  }    var colors: Dictionary = {};  colors["red"] = "#ff0000";  colors["green"] = "#00ff00";  colors["blue"] = "#0000ff";    console.log(colors["red"]); |

### **Гибридные интерфейсы**

Интерфейсы могут сочетать различные стили, могут применяться сразу как к определению объекта, так и к определению функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | interface PersonInfo {      (name: string, surname: string):void;      fullName: string;      password: string;      authenticate(): void;  }    function personBuilder(): PersonInfo {        let person = <PersonInfo>function (name: string, surname: string): void{          person.fullName = name + " " + surname;      };      person.authenticate = function () {          console.log(person.fullName + " входит в систему с паролем " + person.password);      };      return person;  }    let tom = personBuilder();  tom("Tom", "Simpson");  tom.password = "qwerty";  tom.authenticate(); |

Тип функции, определяемый в таком гибридном интерфейсе, как правило, выступает в роли конструктора объекта. В данном случае такой конструктор имеет тип (name: string, surname: string):void;.

А функция, которая представляет данный интерфейс (в данном случае - функция personBuilder), реализует эту функцию конструктора, и также может использовать другие свойства и методы, которые были определены в интерфейсе.

## 3.8.Преобразование типов

Ранее рассматривалась тема [type assertion](https://metanit.com/web/typescript/2.4.php), где говорилось про преобразования типов. Те же способы преобразаний мы можем применять и при работе с классами и интерфейсами.

Так, рассмотрим следующую иерархию типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }  }    class Employee extends Person {        company: string;      constructor(userName: string, company: string) {            super(userName);          this.company = company;      }  } |

Здесь класс Employee унаследован от класса Person. Поскольку объекты Employee в то же время являются и объектами Person, то при определении объектов мы можем написать так:

|  |  |
| --- | --- |
| 1 | let tom : Person = new Employee("Tom", "Microsoft"); |

Соответственно везде, где в функцию в качестве параметра передается объект Person или возвращается из функции объект Person, мы вместо объекта Person можем передавать объект Employee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }  }    class Employee extends Person {        company: string;      constructor(userName: string, company: string) {            super(userName);          this.company = company;      }  }    function printPerson(user: Person): void{      console.log(`Person ${user.name}`);  }    function personFactory(userName: string): Person {      return new Employee(userName, "не установлено");  }    let tom : Person = new Employee("Tom", "Microsoft");  printPerson(tom);    let bob = personFactory("Bob");  printPerson(bob); |

Здесь продемонстрированы восходящиие преобразования, то есть преобразования от более конкретного типа к более общему - от призводного типа Employee к базовому типу Person. Они производятся неявно, и нам не надо писать какой-то дополнительный код.

Но есть и другой тип преобразований - нисходящие или от более общего типа к более конкретному. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }  }    class Employee extends Person {        company: string;      constructor(userName: string, company: string) {            super(userName);          this.company = company;      }  }    let tom : Person = new Employee("Tom", "Microsoft");  console.log(tom.company);   // ошибка - в классе Person нет свойства company |

Здесь переменная tom имеет тип Person, однако в реальности эта переменная указывает на объект типа Employee, так как для ее инициализации мы использовали конструктор типа Employee, который устанавливает свойство company. Однако попытка вывести значение свойства company у объекта tom завершится ошибкой, так как tom - это все таки переменная типа Person, в котором нет свойства company.

Чтобы решить эту ситуацию, нам надо явно преобразовать объект tom к типу Employee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let tom : Person = new Employee("Tom", "Microsoft");    let tomEmployee: Employee = <Employee>tom; // преобразование к типу Employee  console.log(tomEmployee.company);    // или так  console.log((<Employee>tom).company); |

Выражение <Тип> переменная позволяет преобразовать переменную к типу, который идет в угловых скобках.

Другой способ осуществить явное преобразование типов представляет применение оператора **as**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let tom : Person = new Employee("Tom", "Microsoft");    let tomEmployee: Employee = tom as Employee; // преобразование к типу Employee  console.log(tomEmployee.company);    // или так  console.log((tom as Employee).company); |

### **Интерфейсы в преобразаниях типов**

Все сказанное в отношении преобразования классов будет справедливо и для преобразования интерфейсов. В то же время есть некоторые особенности. Пусть у нас будет интерфейс IPerson, никак не связанный с классами Person и Employee и ими не реализуемый:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | interface IPerson {      name: string;  }  class Person {        name: string;      constructor(userName: string) {            this.name = userName;      }  }    class Employee extends Person {        company: string;      constructor(userName: string, company: string) {            super(userName);          this.company = company;      }  }  function printPerson(user: IPerson): void {      console.log(`IPerson ${user.name}`);  } |

Функция printPerson в качестве параметра принимает объект интерфейса IPerson:

|  |  |
| --- | --- |
| 1  2  3  4  5 | let tom: Person = new Employee("Tom", "Microsoft");  printPerson(tom);    printPerson({ name: "Sam" });  //printPerson({ name: "Bob", company:"Microsoft" }); // ошибка |

Ни класс Person, ни класс Employee не применяют интерфейс IPerson, однако мы можем их использовать, так как они имеют все те же свойства и методы, что интерфейс IPerson (в данном случае только свойство name).

Объект { name: "Sam" } также является объектом интерфейса, так как он имеет свойство name. В то же время при передаче объекта { name: "Bob", company:"Microsoft" } мы получим ошибку, так как он уже расширяет возможности IPerson, добавляя свойство company и напрямую интерфейсу IPerson не соответствует. Но даже в этом случае мы его можем вполне использовать, применив преобразование типов:

|  |  |
| --- | --- |
| 1 | printPerson({ name: "Bob", company:"Microsoft" } as IPerson); |

### **Оператор instanceOf**

С помощью оператора **instanceOf** можно проверить, принадлежит ли объект определенному классу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let tom = new Employee("Tom", "Microsoft");  if (tom instanceof Person) {      console.log("Tom is a Person");  }  else {      console.log("Tom is not a Person");  } |

## 3.9.Обобщения

TypeScript является строго типизированным языком, однако иногда надо построить функционал так, чтобы он мог использовать данные любых типов. В некоторых случаях мы могли бы использовать тип any:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function getId(id: any): any {        return id;  }  let result = getId(5);  console.log(result); |

Однако в этом случае мы не можем использовать результат функции как объект того типа, который передан в функцию. Для нас это тип any. Если бы вместо числа 5 в функцию передавался бы объект какого-нибудь класса, и нам потом надо было бы использовать этот объект, например, вызывать у него функции, то это было бы проблематично. И чтобы конкретизировать возвращаемый тип, мы можем использовать обобщения:

|  |  |
| --- | --- |
| 1  2  3  4 | function getId<T>(id: T): T {        return id;  } |

С помощью выражения <T> мы указываем, что функция getId типизирована неким типом T. Причем на момент написания функции мы можем не знать, что именно это будет за тип. А при выполнении функции вместо Т будет подставляться конкретный тип. Причем на этапе компиляции конкретный тип не известен. И возвращать функция будет объект этого типа. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function getId<T>(id: T): T {        return id;  }  let result1 = getId<number>(5);  console.log(result1);  let result2 = getId<string>("abc");  console.log(result2); |

В первом случае вместо параметра T будет испльзоваться тип number, поэтому в функцию мы можем передать число. Во втором случае вместо T используется тип string, поэтому во втором случае можно передать строку. Таким образом, мы можем передать в функцию объекты различных типов, но при этом сохраняется строгая типизация, каждый вариант обобщенной функции может принимать объекты только определенного типа.

Подобным образом еще можно использовать обобщенные массивы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function getString<T>(args: Array<T>): string {        return args.join(", ");  }    let result = getString<number>( [1, 2, 34, 5]);  console.log(result); |

В данном случае вне зависимости от типа данных, переданных в массиве, все его элементы соединятся в одну общую строку.

### **Обобщенные классы и интерфейсы**

Кроме обобщенных функций и массивов также бывают обобщенные классы и интерфейсы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class User<T> {        private \_id: T;      constructor(id:T) {            this.\_id=id;      }      getId(): T {            return this.\_id;      }  }    let tom = new User<number>(3);  console.log(tom.getId()); // возвращает number    let alice = new User<string>("vsf");  console.log(alice.getId()); // возвращает string |

Только в данном случае надо учитывать, что если мы типизировали объект определенным типом, то сменить данный тип уже не получится. То есть в следующем случае второе создание объекта не будет работать, так как объект tom уже типизирован типом number:

|  |  |
| --- | --- |
| 1  2  3 | let tom = new User<number>(3);  console.log(tom.getId());  tom = new User<string>("vsf"); // ошибка |

Все то же самое и с интерфейсами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | interface IUser<T> {        getId(): T;  }    class User<T> implements IUser<T> {        private \_id: T;      constructor(id:T) {            this.\_id=id;      }      getId(): T {            return this.\_id;      }  } |

### **Ограничения обобщений**

Обобщения позволяют работать с любым типом данных. Однако иногда возникает необходимость использовать не любой тип, а только некоторый набор типов, который соответствует некоторым критериям. Например, возьмем следующую функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function compareName<T>(obj1: T, obj2: T): void{        if(obj1.name === obj2.name){          console.log("Имена совпадают");      }      else{          console.log("Имена различаются");      }    } |

Функция принимает два объекта, тип которых неизвестен. В коде функции сравниваются значения полей name этих объектов.

Попробуем с помощью этой функции сравнить два объекта, которые имеют свойство name:

|  |  |
| --- | --- |
| 1  2  3 | let tom: {name:string} = {name: "Tom"};  let sam: {name: string} = {name: "Sam"};  compareName<{name:string}>(tom, sam); |

Здесь сравниваются два объекта: tom и sam, которые имеют одини и тот же тип {name:string}. То есть оба объекта имеют свойство name.

При вызове функция compareName() типизируется этим типом - {name:string}. Казалось бы, никаких проблем не должно возникнуть. Тем не менее при компиляции мы получим ошибку:

Property 'name' does not exist on type 'T'

Использование обобщений, которые подходят под любой тип, расширяют набор используемых типов, однако ограничивает их применение.

Ограничения или **constraints** позволяют ограничить набор типов, которые могут использоваться в обобщениях. Ограничения задаются в форме

|  |  |
| --- | --- |
| 1 | <T extends критерий\_типов> |

После названия параметра (в данном случае "T") идет ключевое слово **extends**, после которого указывается критерий, которому должны соответствовать типы данных, передаваемые вместо параметра "T".

Например, в случае с функцией compareName() в примере выше типы должны иметь свойство name. Поэтому перепишим функцию следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function compareName<T extends {name:string}>(obj1: T, obj2: T): void{        if(obj1.name === obj2.name){          console.log("Имена совпадают");      }      else{          console.log("Имена различаются");      }    }    let tom: {name:string} = {name: "Tom"};  let sam: {name: string} = {name: "Sam"};  compareName<{name:string}>(tom, sam); |

Запись <T extends {name:string}> означает, что параметр T должен представлять такой тип, который содержит свойство name, как в случае выше два объекта tom и sam.

Причем параметр T необязательно должен представлять именно тип {name:string}. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function compareName<T extends {name:string}>(obj1: T, obj2: T): void{        if(obj1.name === obj2.name){          console.log("Имена совпадают");      }      else{          console.log("Имена различаются");      }    }    class User{      constructor(public name: string, public age: number){}  }  let bob = new User("Bob", 38);  let bobic = new User("Bob", 24);  compareName<User>(bob, bobic);    type Person = {id:number; name:string};  let tom: Person = {id:1, name: "Tom"};  let sam: Person = {id: 2, name: "Sam"};  compareName<Person>(tom, sam); |

Здесь в первом вызове функция compareName() типизируется классом User, то есть передаваемые в нее объекты должны представлять класс User. При втором вызове функция типизируется типом Person, который представляет объект {id:number; name:string}. Но и User и Person объединяет то, что они имеют свойство name и поэтому соответствуют ограничению {name: string}.

Причем в качестве типа могут использоваться любые типы, например, интерфейсы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface Named{      name: string;  }  function compareName<T extends Named>(obj1: T, obj2: T): void{        if(obj1.name === obj2.name){          console.log("Имена совпадают");      }      else{          console.log("Имена различаются");      }  } |

Подобным образом ограничения обобщений можно применять в интерфейсах и классах:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | interface Named{      name: string;  }  class NameInfo<T extends Named>{        printName(obj: T): void{            console.log(`Name: ${obj.name}`);      }  }    class User{      constructor(public name: string, public age: number){}  }  let bob = new User("Bob", 38);  let nameInfo1 = new NameInfo<User>();  nameInfo1.printName(bob);    type Person = {id:number; name:string}  let tom: Person = {id:1, name: "Tom"};  let nameInfo2 = new NameInfo<Person>();  nameInfo2.printName(tom); |

В данном случае класс NameInfo применяет параметр типа T, который ограничивается интерфейсом Named.

И в этом случае мы можем типизировать объекты класса NameInfo любым типом, который имеет свойство name, как в данном случае классом User или типом Person.

### **Ключевое слово new**

Чтобы создать новый объект в коде обобщений, нам надо указать, что обобщенный тип T имеет конструктор. Это означает, что вместо параметра type:T нам надо указать type: {new(): T;}. Например, следующий обобщенный интерфейс работать не будет:

|  |  |
| --- | --- |
| 1  2  3 | function UserFactory<T>(): T {      return new T(); // ошибка компиляции  } |

Чтобы интерфейс начал работать, используем слово new:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function userFactory<T>(type: { new (): T; }): T {        return new type();  }      class User {        constructor() {          console.log("создан объект User");      }  }    let user : User = userFactory(User); |

**3.10Миксины**

TypeScript, как и многие объектно-ориентированные языки, как, например, Java или C#, не позволяет использовать напрямую множественное наследование. Мы можем реализовать множество интерфейсов в классе, но унаследовать его можем только от одного класса. Однако функциональность миксинов (**mixins**) частично позволяют унаследовать свойства и методы сразу двух и более классов.

Рассмотрим на примере. Пусть, у нас есть класс Animal, который представляет животное, и класс Movable, который представляет транспортное средство. Оба эти класса имеют свой уникальный функционал, который позволяет выполнять заложенные в них задачи. И также пусть у нас будет класс, который представляет лошадь - с одной стороны, лошадь является животным и наследует все черты, присущие животному, а с другой стороны, лошадь также можно использовать в качестве транспортного средства. То есть для создания подобного класса было бы неплохо унаследовать его сразу и от класса Animal, и от класса Movable. Решим эту задачу на языке TypeScript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Animal {        feed():void {          console.log("Кормим животное");      }  }    class Movable {        speed: number=0;      move(): void {          console.log("Перемещаемся");      }  }    class Horse {}    interface Horse extends Animal, Movable {}    function applyMixins(derivedCtor: any, baseCtors: any[]) {      baseCtors.forEach(baseCtor => {          Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {              derivedCtor.prototype[name] = baseCtor.prototype[name];          });      });  }    applyMixins(Horse, [Animal, Movable]);    let pony: Horse = new Horse();  pony.feed();  pony.move(); |

Чтобы класс Horse мог унаследовать методы классов Animal и Movable, определяется одноименный интерфейс Horse, который расширяет интерфейсы Animal и Movable:

|  |  |
| --- | --- |
| 1 | interface Horse extends Animal, Movable {} |

Но чтобы миксин мог унаследовать функционал, этого недостаточно. Нам еще надо использовать специальную функцию, которая перекопирует функционал из родительских классов в миксин:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function applyMixins(derivedCtor: any, baseCtors: any[]) {      baseCtors.forEach(baseCtor => {          Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {              derivedCtor.prototype[name] = baseCtor.prototype[name];          });      });  } |

Затем применяем эту функцию:

|  |  |
| --- | --- |
| 1 | applyMixins(Horse, [Animal, Movable]); |

Первым параметром идет класс-миксин, а второй параметр - массив применяемых классов.

# 4.Модули и пространства имен

## 4.1.Пространства имен

Для организации больших программ предназначены пространства имен. Пространства имен содержат группу классов, интерфейсов, функций, других пространств имен, которые могут использоваться в некотором общем контексте.

Для определения пространств имен используется ключевое слово **namespace**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | namespace Personnel {      export class Employee {            constructor(public name: string){          }      }  } |

В данном случае пространство имен называется Personnel, и оно содержит класс Employee. Чтобы типы и объекты, определенные в пространстве имен, были видны извне, они определяются с ключевым словом **export**. В этом случае во вне мы сможем использовать класс Employee:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | namespace Personnel {      export class Employee {            constructor(public name: string){          }      }  }    let alice = new Personnel.Employee("Alice");  console.log(alice.name);    // Alice |

При этом пространства имен могут содержать и интерфейсы, и объекты, и функции, и даже обычные переменные и константы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | namespace Personnel {        export interface IUser{          displayInfo(): void;      }        export class Employee {          constructor(public name: string){          }      }        export function work(emp: Employee) : void{          console.log(emp.name, "is working");      }        export let defaultUser= { name: "Kate" }        export let value = "Hello";  }    let tom = new Personnel.Employee("Tom")  Personnel.work(tom);                    // Tom is working    console.log(Personnel.defaultUser.name);    // Kate  console.log(Personnel.value);    // Hello |

### **Пространство имен в отдельном файле**

Нередко пространства имен определяются в отдельных файлах. Например, определим файл **personnel.ts** со следующим кодом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | namespace Personnel {      export class Employee {            constructor(public name: string){          }      }      export class Manager {            constructor(public name: string){          }      }  } |

И в той же папке определим главный файл приложения **app.ts**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | /// <reference path="personnel.ts" />    let tom = new Personnel.Employee("Tom")  console.log(tom.name);    let sam = new Personnel.Manager("Sam");  console.log(sam.name); |

С помощью директивы /// <reference path="personnel.ts" /> подключается файл personnel.ts.

Далее нам надо объединить оба файла в один файл, который затем можно подключать на веб-страницу. Для этого при компиляции указывается опция:

|  |  |
| --- | --- |
| 1 | --outFile target.js sourse1.ts source2.ts source3.ts ... |

Опции outFile в качестве первого параметра передается название файла, который будет генерироваться. А последующие параметры - файлы с кодом TypeScript, которые будут компилироваться.

То есть в данном случае нам надо выполнить в консоли команду

tsc --outFile app.js app.ts personnel.ts

В итоге будет создан один файл app.js.

### **Вложенные пространства имен**

Пространства имен могут быть вложенными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | namespace Data{      export namespace Personnel {          export class Employee {                constructor(public name: string){              }          }      }      export namespace Clients {          export class VipClient {                constructor(public name: string){              }          }      }  }    let tom = new Data.Personnel.Employee("Tom")  console.log(tom.name);    let sam = new Data.Clients.VipClient("Sam");  console.log(sam.name); |

Причем вложенные пространства имен определяются со словом **export**. Соответственно при обращении к типам надо использовать все пространства имен.

### **Псевдонимы**

Возможно, нам приходится создавать множество объектов Data.Personnel.Employee, но каждый раз набирать полное имя класса с учетом пространств имен, вероятно, не всем понравиться, особенно когда модули имеют глубокую вложенность по принципу матрешки. Чтобы сократить объем кода, мы можем использовать псевдонимы, задаваемые с помощью ключевого слова **import**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | namespace Data{      export namespace Personnel {          export class Employee {                constructor(public name: string){              }          }      }  }    import employee = Data.Personnel.Employee;  let tom = new employee("Tom")  console.log(tom.name); |

После объявления псевдонима employee будет рассматриваться как краткое имя для Data.Personnel.Employee.

## 4.2.Создание и подключение модулей

TypeScript поддерживает работу с модулями. Модули являются концепцией, привнесенной стандартом ES2015. Модули в некотором смысле похожи на пространства имен: они могут заключать различные классы, интерфейсы, функции, объекты. Модули выделяются в отдельные файлы, что позволяет сделать код приложения более ясным и чистым, и в то же время позволяет использовать модули в других приложения.

В TypeScript, как и в стандарте ECMAScript 2015, любой файл, который содержит выражения import или export верхнего уровня, рассматривается как модуль. Модули выполняются не в глобальном контексте, а в своей собственной области видимости. То есть переменные, функции, классы. интерфейсы и т.д., определенные внутри модуля, не доступны извне, пока они не будут явным образом экспортированы. А чтобы другой модуль мог их использовать, он должен их импортировать.

Если файл не содержит выражений import или export верхнего уровня, он рассматривается как обычный скрипт, содержимое которого доступно в глобальной области видимости. Такие файлы можно склеить в один с помощью параметра компиляции **--outFile**, которому передается название выходного файла.

Чтобы сделать из простого скрипта модуль, достаточно добавить в файл

|  |  |
| --- | --- |
| 1 | export {}; |

Этот модуль ничего не экспортирует, но тем не менее это модуль.

### **Типы модулей**

Все модули имеют определенный формат и относятся к определенном типу. Всего мы можем использовать следующие типы модулей:

* AMD (Asynchronys Module Definition)
* CommonJS (используется по умолчанию, если параметр --target равен "ES3" или "ES5")
* UMD (Universal Module Definition)
* System
* ES 2015
* ES 2020
* ESNext

Модули типов ES2015 и ES2020 практически одинаковы за тем исключение, что ES2020 имеет поддержку для динамического импорта и выражения импорта import.meta.

Рассмотрим модули на примере модулей типа ES2015/ES2020, поскольку это один из наиболее популярных типов (наряду с CommonJS), кроме того, этот тип модулей современные браузеры поддерживают по умолчанию.

### **Определение модуля**

Определим простейший модуль. Для этого создадим файл **message.ts**, в котором определим следующий код:

|  |  |
| --- | --- |
| 1  2  3 | export default function hello() {    console.log("Hello Typescript");  } |

Здесь определена обычная функция hello(), которая выводит некоторое сообщение на консоль. Но она определена с ключевым словом **export**, а это значит, что данный файл представляет модуль, а функцию hello() можно импортировать в другие модули.

Кроме того, здесь применяется ключевое слово **default**, которое устанавливает тип, экспортируемый по умолчанию.

### **Подключение модуля. Импорт**

Теперь подключим эту функцию в другой файл. Для этого возьмем файл **main.ts**:

|  |  |
| --- | --- |
| 1  2 | import hello from "./message.js";  hello(); |

Для подключения функционала из другого модуля применяется ключевое слово **import**, после которого идут названия подключаемых компонентов - в данном случае функция hello. Затем после оператора **from** указывается модуль, из которого идет импорт. В данном случае указываем "./message.js". В данном случае предполагается что оба скомпилированных модуля - main.js и message.js будут находиться в одной папке.

### **Компиляция модуля**

При компиляции из командной строки или терминала для установки модуля необходимо передать соответствующее значение параметру --module:

tsc --module commonjs main.ts // для CommonJS

tsc --module amd main.ts // для AMD

tsc --module umd main.ts // для UMD

tsc --module system main.ts // для SytemJS

tsc --module esnext main.ts // для ESNext

В данном случае у нас оба модуля - main.ts и message.ts являются модулями ES, то есть в качестве типа модуля необходимо выбрать "ES2015", "ES2020" или "ESNext". Для данного конкретного случае не столь важно, какой именно из этих трех типов выбирать, поэтому выберем последний вариант.

Сначала перейдем в командной строке/консоли с помощью команды **cd** к папке, где расположены файлы main.ts и message.ts. Так, в моем случае они расположены в папке **c:\typescript**

Далее для компиляции модулей введем команду:

tsc main.ts --module esnext

После этого в папке появятся файлы main.js и message.js.

**Загрузка модулей**

Для загрузки модулей определим в папке со скомпилированными файлами веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <h2>Модули в TypeScript</h2>      <script type="module" src="main.js"></script>  </body>  </html> |

Для загрузки главного модуля приложения - main.js определяется элемент <script>, у которого устанавливается атрибут type="module".

Загрузка модулей производится через AJAX, поэтому скомпилированные модули должны быть размещены на веб-сервере. То есть у нас не получится просто кинуть страницу в веб-браузер, как, например, это было в первых темах. Поэтому прежде всего надо определиться с веб-сервером. Веб-сервер может быть любым. В данном случае воспользуемся самым демократичным вариантом - Node.js. Но опять же вместо node.js это может быть любая другая технология сервера - php, asp.net, python и т.д. либо какой-то определенный веб-сервер типа Apache или IIS.

Итак, создадим в папке с файлами модулей файл сервера. Пусть он будет называться **server.js** и будет иметь следующий код:

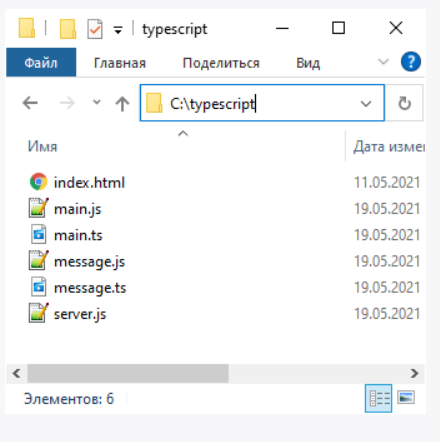
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | const http = require("http");  const fs = require("fs");    http.createServer(function(request, response){        // получаем путь после слеша      let filePath = request.url.substr(1);      if(filePath == "") filePath = "index.html";      fs.readFile(filePath, function(error, data){            if(error){                response.statusCode = 404;              response.end("Resourse not found!");          }          else{              if(filePath.endsWith(".js")) response.setHeader("Content-Type", "text/javascript");              response.end(data);          }      });  }).listen(3000, function(){      console.log("Server started at 3000");  }); |

Это самый примитивный сервер, который отдает пользователю статические файлы. Для создания сервера применяется функция http.createServer(), а для считывания и отправки файлов - функция fs.readFile(). Если имя файла не указано, то отправляется файл index.html. Сервер будет запускаться по адресу http://localhost:3000/

Стоит отметить, что при отправке модулей js нам надо устанавливать mime-тип отправляемого контента в "text/javascript":

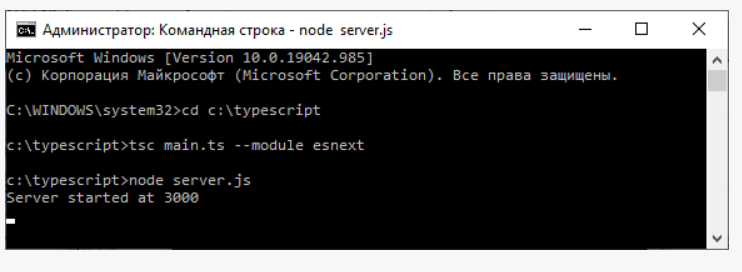
|  |  |
| --- | --- |
| 1 | if(filePath.endsWith(".js")) response.setHeader("Content-Type", "text/javascript"); |

Структура финального проекта:

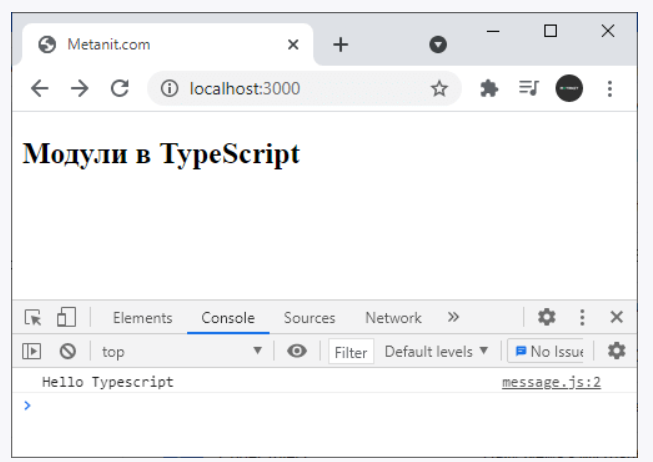


Теперь запустим сервер с помощью команды

node server.js



После запуска сервера мы можем перейти в браузере по адресу http://localhost:3000, нам отобразится страница, а в консоли браузера мы сможем увидеть результат работы нашего кода на typescript:



**Конфигурация typescript**

Вместо того, чтобы указывать тип модуля в консоли при компиляции, можно использовать соответствующие параметры в файле конфигурации **tsconfig.json**. Так, параметр **module** задает тип модуля:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {      "compilerOptions": {          "noImplicitAny": true,          "noEmitOnError": true,          "strictNullChecks": true,          "outFile": "main.js"            "target": "es2015",          "module": "esnext"      }  } |

## 4.3.Работа с модулями

### **Экспорт компонентов модуля**

Пусть у нас будет в проекте файл **devices.ts**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | export interface Device{      name: string;  }    export class Phone implements Device {      name: string;      constructor(n:string){          this.name = n;      }  }    export function call(phone: Phone) : void{      console.log("Make a call by", phone.name);  } |

Чтобы классы, интерфейсы, функции были видны извне, они определяются с ключевым словом export.

Но мы могли бы и по другому экспортировать все сущности:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface Device{      name: string;  }    class Phone implements Device {      name: string;      constructor(n:string){          this.name = n;      }  }    function call(phone: Phone) : void{      console.log("Make a call by", phone.name);  }  export {Device, Phone, call}; |

### **Импорт**

Чтобы задействовать модуль в приложении, его надо импортировать с помощью оператора **import**. Например, импортируем класс Phone и функцию Call из выше определенного модуля devices.ts:

|  |  |
| --- | --- |
| 1  2  3 | import {Phone, call} from "./devices";  let iphone: Phone = new Phone("iPhone X");  call(iphone); |

После слова **import** определяется набор импортируемых типов - класов, интерфейсов, функций, объектов. А после слова **from** указывается путь к модулю. В данном случае модуль располагается в файле devices.js, который находится в той же папке, поэтому в начале пути ставится точка и далее указывается название файла без расширения. Если бы модуль располагался бы в папке lib, находящеся в текущем каталоге, то название папки также бы включалось в путь к модулю: "./lib/devices".

### **Псевдонимы**

При экспорте и импорте для компонента модуля можно указать псевдоним с помощью оператора **as**:

Например, установка псевдонима для компонента при импорте:

|  |  |
| --- | --- |
| 1  2  3 | import {Phone, call as makeCall} from "./devices.js";  let iphone: Phone = new Phone("iPhone X");  makeCall(iphone); |

Так, в данном случае для функции call() установлен псевдоним makeCall(), и далее мы обращаемся к функции call() через ее псевдоним makeCall().

Также псевдоним можно установить при экспорте компонента:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface Device{      name: string;  }    class Phone implements Device {      name: string;      constructor(n:string){          this.name = n;      }  }    function call(phone: Phone) : void{      console.log("Make a call by", phone.name);  }  export {Device, Phone, call as makeCall}; |

Затем компонент импортируется через его псевдоним:

|  |  |
| --- | --- |
| 1  2  3 | import {Phone, makeCall} from "./devices.js";  let iphone: Phone = new Phone("iPhone X");  makeCall(iphone); |

### **Импорт всего модуля**

Можно импортировать сразу весь модуль:

|  |  |
| --- | --- |
| 1  2  3 | import \* as dev from "./devices.js";  let iphone: dev.Phone = new dev.Phone("iPhone X");  dev.makeCall(iphone); |

В данном случае модуль импортируется через псевдоним "dev". И, используя этот псевдоним, мы можем обращаться к расположенным в этом модуле типам.

### **Экспорт по умолчанию**

Параметры экспорта по умолчанию позволяют определить тип, который будет импортироваться из модуля по умолчанию. К примеру, добавим новый модуль **smartwatch.ts**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export default class SmartWatch{        constructor(private model:string){}        printModel(){          console.log(`Model: ${this.model}`);      }  } |

Ключевое слово **default** позволяет установить класс SmartWatch в качестве типа по умолчанию. И затем мы можем импортировать его следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | import SmartWatch from "./smartwatch.js";  let watch: SmartWatch = new SmartWatch("Apple Watch");  watch.printModel(); |

При этом мы можем установить для экспортируемого по умолчанию компонента другое имя:

|  |  |
| --- | --- |
| 1  2  3 | import Watch from "./smartwatch.js";  let watch: Watch = new Watch("Apple Watch 2");  watch.printModel(); |

# 5.Заголовочные файлы

## 5.1.Работа с заголовочными файлами

Для установки связи с внешними файлами скриптов javascript в TS служат декларативные или заголовочные файлы. Это файлы с расширением *.d.ts*, они описывают синтаксис и структуру функций и свойств, которые могут использоваться в программе, не предоставляя при этом конкретной реализации. Их действие во многом похоже на работу файлов с расширением *.h* в языках C/C++. Они выполняют своего рода роль оберток над библиотеками JavaScript.

Рассмотрим, как мы можем использовать заголовочные файлы. Иногда в программах на javascript используются глобальные переменные, которые должны быть видны для всех функций приложения. Например, пусть на веб-странице (или во внешнем подключаемом файле javascript) в коде js определена переменная:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <h2>Приложение на TypeScript</h2>      <script>          let message = "Hello TypeScript!";      </script>      <script src="app.js"></script>  </body>  </html> |

В данном случае для простоты переменная определена веб-странице, хотя она также могла быть определена во внешнем подключенном js-файле.

И, допустим, мы хотим использовать эту переменную message в коде TypeScript в файле app.ts:

|  |  |
| --- | --- |
| 1 | console.log(message); |

При запуске приложения компилятор TS не сможет скомпилировать программу, так как для кода TS глобальная переменная пока не существует. В этом случае нам надо подключать определение глобальной переменной с помощью декларативных файлов. Для этого добавим в проект новый файл, который назовем **globals.d.ts** и который будет иметь следующее содержимое:

|  |  |
| --- | --- |
| 1 | declare let message: string; |

С помощью ключевого слова **declare** в программу на TS подключается определение глобальной переменной.

То есть у нас получится следующая структура проекта:

* app.ts
* globals.d.ts
* index.html

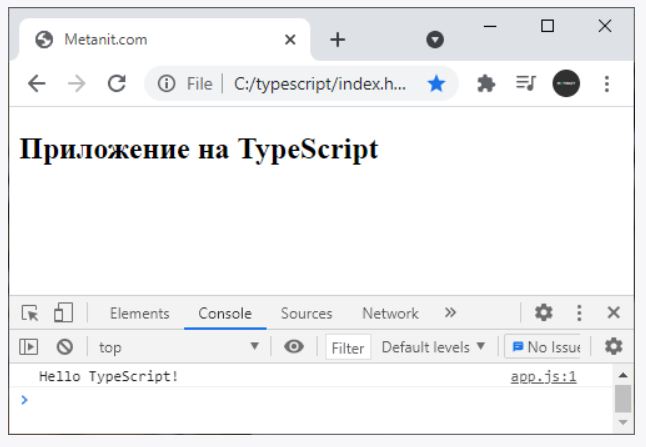
### **Компиляция**

Если мы компилируем, передавая компилятору в консоли название файла:

tsc app.ts

То в этом случае компилятор не найдет автоматически файл globals.d.ts. В этом случае нам надо в файле **app.ts** явно указать расположение файла globals.d.ts с помощью директивы **reference**

|  |  |
| --- | --- |
| 1  2 | /// <reference path="globals.d.ts" />  console.log(message); |



Если же мы полагаемся на файл конфигурации tsconfig.json, просто выполняя команду

tsc

то директиву /// <reference path="globals.d.ts" /> можно не указывать.

Подобным образом мы можем подключать другие компоненты кода JavaScript - функции, объекты, классы. Рассмотрим их подключение.

### **Функции**

Пусть на веб-странице в коде js объявлены две следующие функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | let message = "Hello TypeScript!";  function hello(){      console.log(message);  }    function sum(a, b){      return a + b;  } |

Функция hello() выводит значение переменной message на консоль, а функция sum() возвращает сумму двух чисел.

И, допустим, в коде TS мы хотим вызывать эти функции:

|  |  |
| --- | --- |
| 1  2  3  4 | hello();    let result = sum(2, 5);  console.log(result); |

В этом случае подключение в файле globals.d.ts выглядело бы так:

|  |  |
| --- | --- |
| 1  2 | declare function hello(): void;  declare function sum(a: number, b: number): number; |

### **Подключение объектов**

Пусть в коде JavaScript есть следующий объект:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | const tom = {      name: "Tom",      age: 37,      print(){          console.log(`Name: ${this.name}  Age: ${this.age}`);      }  } |

Используем этот объект в коде typescript:

|  |  |
| --- | --- |
| 1 | tom.print(); |

В этом случае определение объекта в файле **globals.d.ts** выглядело бы так:

|  |  |
| --- | --- |
| 1 | declare const tom: {name: string, age: number, print: ()=> void}; |

### **Подключени сложных объектов**

Однако может возникнуть сложность с подключением более сложных объектов. Например, пусть есть такой объект javascript:

|  |  |
| --- | --- |
| 1  2  3 | var points = [{ X: 10, Y: 34 },                { X: 24, Y: 65 },                 { X: 89, Y: 12 }]; |

Для данного массива объектов в файле **globals.d.ts** мы можем определить соответствующий отдельному объекту интерфейс и подключить массив объектов некоторого интерфейса, который содержит два свойства X и Y:

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface IPoint {      X: number;      Y: number;  }  declare var points: IPoint[]; |

И в TS мы сможем использовать этот массив:

|  |  |
| --- | --- |
| 1  2  3 | for (let point of points){      console.log(`Точка с координатами X = ${point.X}  Y = ${point.Y}`);  } |

Консольный вывод браузера:

Точка с координатами X = 10 Y = 34 Точка с координатами X = 24 Y = 65 Точка с координатами X = 89 Y = 12

### **Подключение классов**

Рассмотрим последний пример - подключение в typescript классов, определенных в javascript. Пусть в коде JavaScript определен следующий класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person{      constructor(name, age){          this.name = name;          this.age = age;      }      display(){          console.log(this.name, this.age);      }  } |

Для этого класса в файле **globals.d.ts** определим следующее объявление класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | declare class Person{        name: string;      age: number;      constructor(name: string, age: number);      display(): void;  } |

Для класса прописываем все его поля и методы, при этом методы (в том числе конструктор) не имеют реализации, для них только определяются параметры и их типы и тип возвращаемого значения.

И в коде TypeScript используем этот класс:

|  |  |
| --- | --- |
| 1  2  3 | let tom = new Person("Tom", 37);  tom.display();          // Tom 37  console.log(tom.name);  // Tom |

**5.2.Заголовочные файлы для популярных библиотек**

В прошлой теме было рассмотрено использование декларативных заголовочных файлов для внешнего кода на javascript. Однако, как правило, нам приходится взаимодействовать с крупными библиотеками, например, jquery, angularjs, extjs и так далее. Если мы хотим использовать функционал этих библиотек в коде на TS, то для них надо создать свои файлы определений. И это может быть довольно утомительно в виду сложности библиотек и больших объемов кода. Но в сообществе TypeScript возникла идея создать общий репозиторий для подобных файлов, чтобы не надо было заново определять свои файлы, а можно бы было взять уже готовые. Этот репозиторий расположен на гитхабе: <https://github.com/DefinitelyTyped/DefinitelyTyped/>

Как использовать js-библиотеки и заголовочные файлы для них? Рассмотрим на примере библиотеки [jquery](https://jquery.com/download/).

Вначале загрузим определения типов для библиотеки jquery с помощью следующей команды:

npm install --save-dev @types/jquery

В итоге в проекте будет создана папка **node\_modules/@types**, в которой каталог **jquery** будет хранить заголовочные файлы для библиотеки jquery.

Кроме того, в проекте будет создан файл **package.json** со следующим содержимым:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    "devDependencies": {      "@types/jquery": "^3.5.5"    }  } |

Здесь мы видим, что текущие определения для jquery работают с версией 3.5.5 и выше.

Теперь нам надо указать компилятору TypeScript с помощью параметра **typeRoots**, где находятся заголовочные файлы - а это каталог **node\_modules/@types**. Для этого в проекте определим следующий файл конфигурации **tsconfig.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | {      "compilerOptions": {          "noImplicitAny": true,          "noEmitOnError": true,          "strictNullChecks": true,            "outFile": "app.js",            "typeRoots": [            "node\_modules/@types"          ],      }  } |

Далее в корневой папке проекта определим следующую веб-страницу **index.html**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Metanit.com</title>  </head>  <body>      <div id="content"></div>      <script src="<https://code.jquery.com/jquery-3.6.0.min.js>"></script>      <script src="app.js"></script>  </body>  </html> |

Здесь только подключается библиотека jquery из cdn, а также файл нашего приложения - app.js.

Далее в корневой папке проекта определим следующий файла **app.ts**:

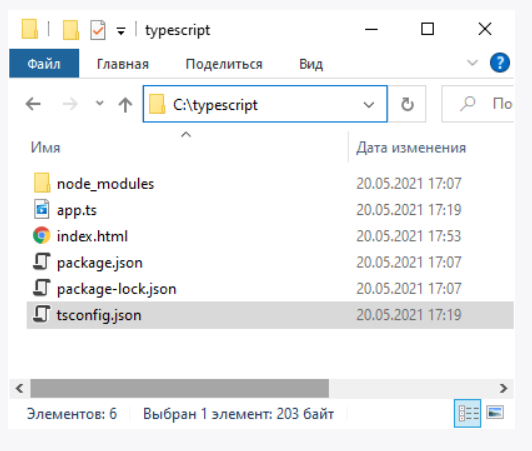
|  |  |
| --- | --- |
| 1  2  3 | $(document).ready(() => {      $("#content").html("<h1>Привет TypeScript</h1>");  }); |

В данном случае мы задействуем событие document.ready, которое определено в jquery и которое срабатывает при загрузке документа. И далее с помощью лямбда-выражения, которое определяет функцию обратного вызова, с помощью знакомого многим синтаксиса jquery на веб-страницу добавляется новый элемент.

Также мы могли бы использовать сокращенное определение функции:

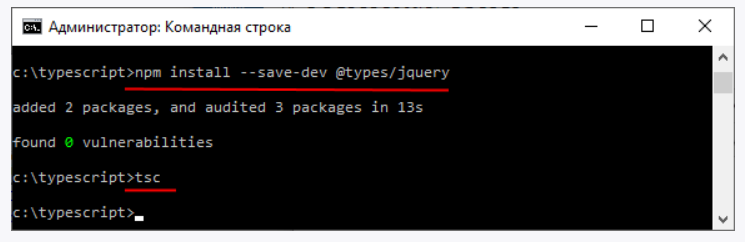
|  |  |
| --- | --- |
| 1  2  3 | $(() => {      $("#content").html("<h1>Привет TypeScript</h1>");  }); |

То есть в итоге проект будет выглядеть следующим образом:

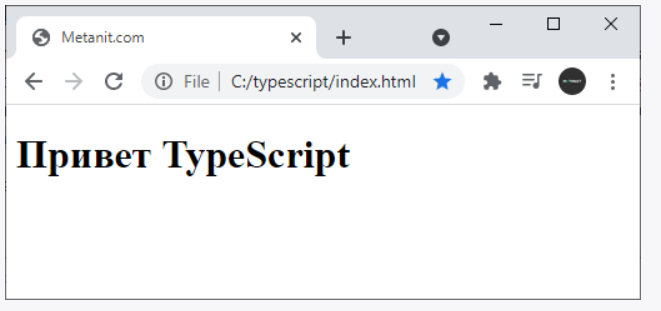


Теперь скомпилируем файл приложения с помощью команды

tsc



В итоге при запуске веб-страницы index.html сработает код из файла app.ts:



Для примера еще обработаем событие нажатия кнопки. Допустим, в html-коде страницы есть такая кнопка:

|  |  |
| --- | --- |
| 1 | <button id="alertBtn">Жми</button> |

В коде на TS мы могли бы так обработать нажатия на эту кнопку:

|  |  |
| --- | --- |
| 1  2  3 | $(() => {      $("#alertBtn").click((e) => { $("#content").html("<h2>Привет мир</h2>"); });  }); |

Обработчики событий, например, click в качестве параметра также принимают лямбда-выражение, которое определяет набор инструкции, выполняемых при нажатии.

**6.Декораторы**

Декораторы являются инструментом декларативного программирования, они позволяют добавить к классам и их членам метаданные и тем самым изменить их поведение без изменения их кода.

Декораторы представляют функции, которые могут применяться к классам, методам, методом доступа (геттерам и сеттерам), свойствам, параметрам.

На текущий момент декораторы являются экпериментальной функциональностью языка TypeScript, поэтому при компиляции следует указывать параметр **experimentalDecorators**. Например, через файл **tsconfig.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {      "compilerOptions": {          "target": "ES5",          "experimentalDecorators": true      }  } |

Либо через параметры в командной строке:

tsc app.ts -t ES5 --experimentalDecorators

**6.1.Декораторы классов**

Декоратор класса применяется к конструктору класса и позволяет изменять или заменять определение класса.

Декоратор класса представляет функцию, которая принимает один параметр:

|  |  |
| --- | --- |
| 1 | function classDecoratorFn(constructor: Function){ } |

В качестве параметра выступает конструктор класса. Например, определим простейший декоратор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function sealed(constructor: Function) {      console.log("sealed decorator");      Object.seal(constructor);      Object.seal(constructor.prototype);  }    @sealed  class User {      name: string;      constructor(name: string){          this.name = name;      }      print():void{          console.log(this.name);      }  } |

Декоратор sealed с помощью функции Object.seal запрещает расширение прототипа класса User.

Для применения декоратора используется знак @. Сам декоратор ставится перед названием класса. То есть из-за применения декоратора мы, к примеру, не сможем добавить в класс User новое свойство следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | Object.defineProperty(User, 'age', {      value: 17  }); |

Также декораторы могут изменять результат работы конструктора. В этом случае определение функции декоратора немного меняется, но она также в качестве параметра принимает конструктор класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | function logger<TFunction extends Function>(target: TFunction): TFunction{        let newConstructor: Function = function(name:string){          console.log("Creating new instance");          this.name = name;          this.age = 23;          this.print = function():void{              console.log(this.name, this.age);          }      }      return <TFunction>newConstructor;  }    @logger  class User {      name: string;      constructor(name: string){          this.name = name;      }      print():void{          console.log(this.name);      }  }  let tom = new User("Tom");  let bob = new User("Bob");  tom.print();  bob.print(); |

В данном случае декоратор logger типизирован типом TFunction, который является расширением типа Function, то есть функции. По сути это тип функции конструктора.

В самом декораторе передаваемый конструктор target никак не используется. Но создается новый конструктор. Мы предполагаем, что в конструктор будет передаваться некоторый параметр, который будет называться name. Значение этого параметра передается свойству this.name = name;. Также в конструкторе устанавливается новое свойство this.age и метод this.print(), который выводит на консоль значения обоих свойств.

Далее декоратор применяется к классу User. У этого класса определен конструктор, который устанавливает свойство name. Однако поскольку мы переопределили конструктор, то в реальности при создании объекта User будет устанавливаться как свойство name, так и свойство age. И, кроме того, будет переопределяться метод print.

Вывод консоли браузера

Creating new instance Creating new instance Tom 23 Bob 23

Следует учитывать, что замена конструктора приводит к полной замене всех свойств и методов класса.

## 6.2.Декораторы методов и их параметров

### **Декоратор метода**

Декоратор метода также представляет функцию, которая принимает три параметра:

|  |  |
| --- | --- |
| 1  2  3 | function deprecated(target: any, propertyName: string, descriptor: PropertyDescriptor){      console.log("Method is deprecated");  } |

Декоратор принимает следующие параметры:

1. Функция конструктора класса для статического метода, либо прототип класса для обычного метода.
2. Название метода.
3. Объект интерфейса PropertyDescriptor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface PropertyDescriptor{      configurable?: boolean;      enumerable?: boolean;      value?: any;      writable?: boolean;      get? (): any;      set? (v: any): void;  } |

1. Этот объект описывает изменение декорируемого метода. Применяется при компиляции в ES5 и выше, при ES3 имеет значение undefined.
2. Его свойство value содержит определение функции. Свойство writable указывает, является ли функция модифицируемой (если значение true, то является).

Определим простейший декоратор для метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function readable (target: Object, propertyKey: string, descriptor: PropertyDescriptor) {      descriptor.writable = false;  };    class User {        name: string;      constructor(name: string){          this.name = name;      }        @readable      print():void{          console.log(this.name);      }  }  let tom = new User("Tom");  tom.print = function(){console.log("print has been changed");}  tom.print();  // Tom |

Декоратор readable с помощью выражения descriptor.writable = false; устанавливает, что метод, к которому применяется данный декоратор, не может быть изменен.

В итоге после применения данного декоратора следующая инструкция

|  |  |
| --- | --- |
| 1 | tom.print = function(){console.log("print has been changed");} |

не имеет смысла и не будет работать. Однако если бы декоратор не применялся, то инструкция сработала бы.

### **Параметры декоратора**

Декоратор может принимать параметры, которые позволяют настроить из вне поведение декоратора. Например, немного изменим предыдущий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function readable(onlyRead : boolean){        return function (target: Object, propertyKey: string, descriptor: PropertyDescriptor) {          descriptor.writable = !onlyRead;      };  }    class User {        name: string;      constructor(name: string){          this.name = name;      }        @readable(false)      print():void{          console.log(this.name);      }  }  let tom = new User("Tom");  tom.print = function(){console.log("print has been changed");}  tom.print();  // Tom |

Теперь параметр readable принимает параметр типа boolean:

|  |  |
| --- | --- |
| 1 | function readable(onlyRead : boolean){ |

Значение этого параметра используется для установки свойства descriptor.writable:

|  |  |
| --- | --- |
| 1 | descriptor.writable = !onlyRead; |

При этом сам декоратор должен возратить функцию, которая соответствует выше рассмотренному определению функции декоратора.

При применении декоратора в скобках ему можно передать значения для параметров:

|  |  |
| --- | --- |
| 1  2 | @readable(false)  print():void{ |

### **Параметры и выходной результат метода**

Декоратор метода позволяет нам манипулировать параметрами и возвращаемым результатом метода. Например, определим следующий декоратор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | function log(target: Object, method: string, descriptor: PropertyDescriptor){      let originalMethod = descriptor.value;      descriptor.value = function(...args: number[]){          console.log(JSON.stringify(args));          let returnValue = originalMethod.apply(this, args);          console.log(`${JSON.stringify(args)} => ${returnValue}`)          return returnValue;      }  }    class Calculator{        @log      add(x: number, y: number): number{          return x + y;      }  }    let calc = new Calculator();  let z = calc.add(4, 5);  z = calc.add(6, 7); |

Декоратор log логгирует (выводит на консоль) значения параметров и возвращаемый результат метода. Свойство descriptor.value позволяет получить начальное значение метода - то есть ту функцию, которую представляет метод.

|  |  |
| --- | --- |
| 1 | let originalMethod = descriptor.value; |

Затем происходит переустановка значения descriptor.value.

|  |  |
| --- | --- |
| 1  2 | descriptor.value = function(...args: number[]){  } |

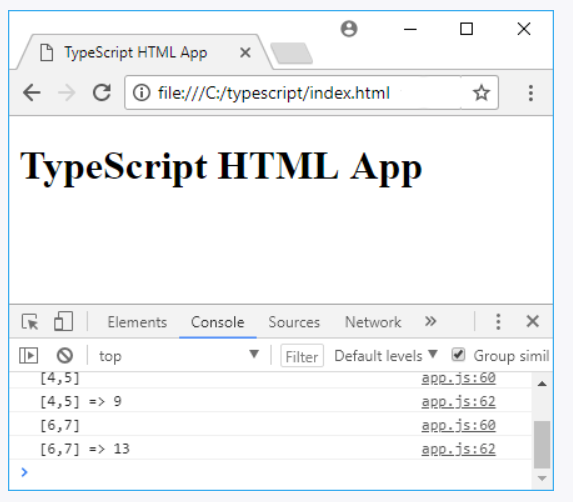
Параметр ...args - это все те параметры, которые будут передаваться в функцию. И мы можем логгировать эти параметры. Далее выывается оригинальная функция, которой передаются параметры args:

|  |  |
| --- | --- |
| 1 | let returnValue = originalMethod.apply(this, args); |

И таким образом мы можем получить результат вызова оригинальной функции и возвратить его из функции. То есть фактически получается, что мы берем оригинальную функцию, обертываем ее в какую-ту другую функцию, в которой опять же вызываем оригинальную функцию и возвращаем ее результат.

Таким образом, новое значение descriptor.value принимает те же параметры, возвращает тот же результат, что и оригинальная функция, но при этом добавляет некоторое дополнительное поведение.

Далее мы можем применить декоратор, например, к методу add класса Calculator и вызвать этот метод.



### **Декораторы параметров методов**

Декоратор параметра метода представляет функцию, которая принимает три параметра:

|  |  |
| --- | --- |
| 1  2  3 | function MyParameterDecorator(target: Object, propertyKey: string, parameterIndex: number){      // код декоратора  } |

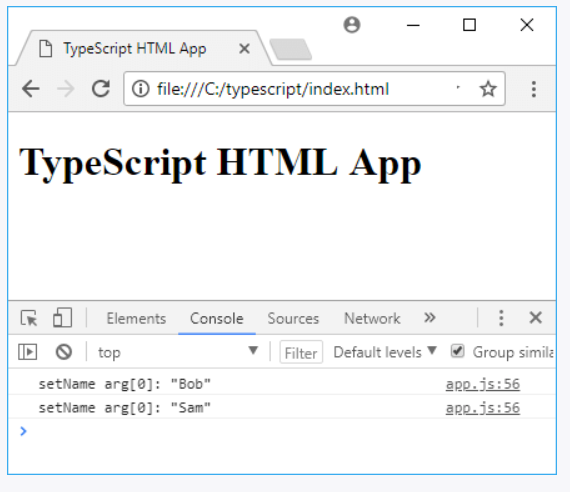
Где первый параметр представляет конструктор класса, если метод статический, либо прототип класса, если метод нестатический. А второй параметр представляет имя метода. И третий параметр представляет порядковый индекс параметра в списке параметров.

Определим декоратор для параметра метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | function logParameter(target: any, key : string, index : number) {      var metadataKey = `\_\_log\_${key}\_parameters`;        if (Array.isArray(target[metadataKey])) {          target[metadataKey].push(index);        }        else {          target[metadataKey] = [index];      }  }  function logMethod(target: any, key: string, descriptor: PropertyDescriptor) {        var originalMethod = descriptor.value;      descriptor.value = function (...args: any[]) {            var metadataKey = `\_\_log\_${key}\_parameters`;          var indices = target[metadataKey];            if (Array.isArray(indices)) {              for (var i = 0; i < args.length; i++) {                    if (indices.indexOf(i) !== -1) {                      var arg = args[i];                      var argStr = JSON.stringify(arg) || arg.toString();                      console.log(`${key} arg[${i}]: ${argStr}`);                  }              }              var result = originalMethod.apply(this, args);              return result;          }          else {              var a = args.map(a => (JSON.stringify(a) || a.toString())).join();              var result = originalMethod.apply(this, args);              var r = JSON.stringify(result);              console.log(`Call: ${key}(${a}) => ${r}`);              return result;          }      }      return descriptor;  }    class User {        private name: string;      constructor(name: string){          this.name = name;      }     @logMethod      setName(@logParameter name: string){          this.name = name;      }      print():void{          console.log(this.name);      }  }  let tom = new User("Tom");  tom.setName("Bob");  tom.setName("Sam"); |

Декоратор logParameter добавляет в прототип класса новое свойство metadataKey. Это свойство представляет массив, который содержит индексы декорированных параметров.

Для чтения метаданных из свойства metadataKey применяется декоратор метода logMethod, который перебирает все параметры метода, находит значения параметров по индексам, которые определены декоратором параметра, и выводит на консоль названия и значения декорированных параметров.



## 6.3.Декораторы свойств и методов доступа

### **Декораторы свойств**

Декоратор свойства представляет функцию, которая принимает два параметра:

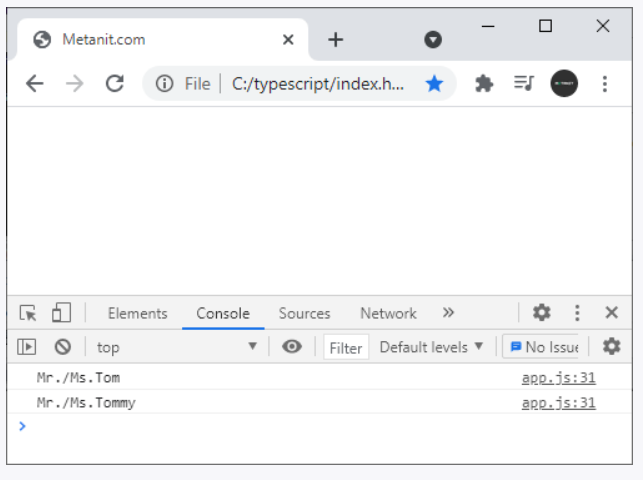
|  |  |
| --- | --- |
| 1  2  3 | function MyPropertyDecorator(target: Object, propertyKey: string){      // код декоратора  } |

Где первый параметр представляет конструктор класса, если свойство статическое, либо прототип класса, если свойство нестатическое. А второй параметр представляет имя свойства.

Определим простейший декоратор для свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | function format(target: Object, propertyKey: string){        let \_val = this[propertyKey];   // получаем значение свойства        // геттер      var getter = function () {          return "Mr./Ms." + \_val;      };        // сеттер      var setter = function (newVal: string) {          \_val = newVal;      };        // удаляем свойство      if (delete this[propertyKey]) {            // И создаем новое свойство с геттером и сеттером          Object.defineProperty(target, propertyKey, {              get: getter,              set: setter          });      }  }    class User {        @format      name: string;      constructor(name: string){          this.name = name;      }      print():void{          console.log(this.name);      }  }  let tom = new User("Tom");  tom.print();  tom.name = "Tommy";  tom.print(); |

Декоратор format выполняет небольшое форматирование значение свойства. Для этого вначале мы получаем значение свойства. Создаем геттер, который возвращает отформатированное значение. Далее определяется сеттер, который устанавливает новое значение для свойства. И в конце удаляется старое свойство и создается новое с геттером и сеттером.



**Декоратор метода доступа**

Декоратор метода доступа принимает три параметра:

|  |  |
| --- | --- |
| 1  2  3 | function decorator(target: Object, propertyName: string, descriptor: PropertyDescriptor){      // код декоратора  } |

Первый параметр представляет конструктора класса для статического метода, либо прототип класса для обычного метода.

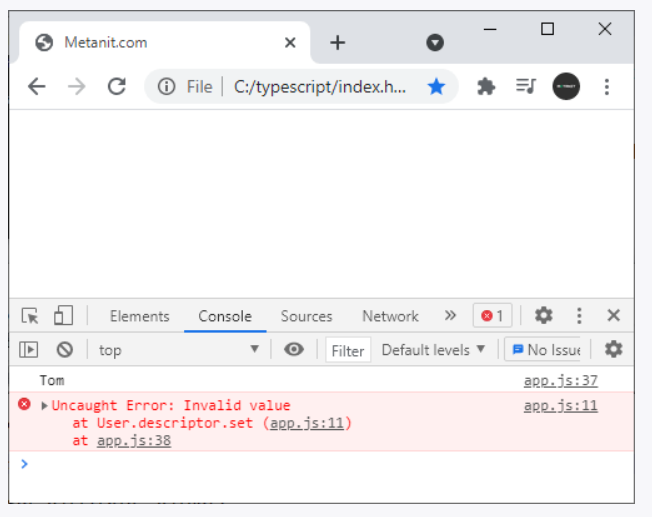
Второй параметр представляет название метода.

Третий параметр представляет объект PropertyDescriptor.

Определим простейший декоратор метода доступа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | function validator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {      const oldSet = descriptor.set;        descriptor.set = function(value: string) {          if (value === "admin") {              throw new Error("Invalid value");          }          if(oldSet!==undefined) oldSet.call(this, value);      }  }  class User {        private \_name: string;      constructor(name: string){          this.name = name;      }        public get name(): string {          return this.\_name;      }      @validator      public set name(n: string) {          this.\_name = n;      }  }  let tom = new User("Tom");  console.log(tom.name);  tom.name= "admin";  console.log(tom.name); |

Декоратор validator переопределяет поведение сеттера с помощью свойства descriptor.set. Если передаваемое сеттеру значение представляет строку "admin", то генерируется ошибка.



Декоратор достаточно применить только к геттеру или к сеттеру, в любом случае он будет сразу применяться к обоим аксессорам.

И, к примеру, при вызове инструкции tom.name= "admin"; мы столкнемся с ошибкой:

## 6.4.Фабрики декораторов

Декоратор класса, свойства, метода представляет обычную функцию, которая принимает заданное количество параметров. Но что если мы хотим передавать в декоратор какие-то дополнительные данные, которые могут быть известны только при применении декоратора? В этом случае мы можем сконструировать фабрику декораторов (factory decorator). Фабрика декоратора представляет функцию, которая в свою очередь возвращает функцию декоратора.

Например, определим протейшую функцию декоратора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | function regex(pattern: string){      let expression = new RegExp(pattern);      return function regex(target: Object, propertyName: string){          let propertyValue = this[propertyName];            // геттер          var getter = function () {              return propertyValue;          };            // сеттер          var setter = function (newVal: string) {              let isValid: boolean = expression.test(newVal);              if(isValid === false){                  throw new Error(`Value ${newVal} does not match ${pattern}`);              }              else{                  console.log(`${newVal} is valid`);              }          };          // удаляем свойство          if (delete this[propertyName]) {                // И создаем новое свойство с геттером и сеттером              Object.defineProperty(target, propertyName, {                  get: getter,                  set: setter              });          }      }  }  class Account{        @regex("^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$")      email: string;        @regex("^[\+]?[(]?[0-9]{3}[)]?[-\s\.]?[0-9]{3}[-\s\.]?[0-9]{4,6}$")      phone: string;        constructor(email: string, phone: string){          this.email = email; this.phone = phone;      }  }  let acc = new Account("bir@gmail.com", "+23451235678");  acc.email = "bir\_iki\_yedi"; |

Декоратор regex принимает в качестве параметра регулярное выражение и при этом возвращает функцию декоратора свойства. В декораторе в сеттере при установке для свойства нового значения проверяется соответствие нового значения регулярному выражению. Если нет соответствия, то генерируется ошибка.

Далее в классе Account к его свойствам email и phone применяется декоратор regex, но в каждом случае в декоратор передается свое регулярное выражение, с помощью которого можно произвести проверку:

