TypeScript 2

Object types, property modifiers

Объектные типы могут быть анонимными, определяться интерфейсами или синонимами типов. Каждое свойство в объектном типе может определять несколько вещей: тип свойства, опциональность, и мутабельность.

Опциональные свойства помечаются вопросительным знаком после имени. Как и с опциональными параметрами, к их типу примешивается undefined.

```
interface PaintOptions {
  shape: Shape
  xPos?: number
  yPos?: number
}
```

Свойства только для чтения помечаются модификатором readonly перед именем. Этот модификатор не делает свойство иммутабельным, оно лишь запрещает присваивать ему другие значения.

```
interface SomeType {
  readonly prop: string
}

function doSomething(obj: SomeType) {
  console.log(obj.prop) // OK
  obj.prop = 'hello'    // Error
}
```

Index signatures

Иногда нам необходимо использовать тип, всех свойств которого мы не знаем, но при этом знаем форму его значений. Для описания подобных типов в TypeScript используются индексы:

```
interface StringArray {
  [index: number]: string
}
```

В приведенном примере интерфейс StringArray содержит сигнатуру индекса. Она означает, что значение типа StringArray содержит свойства с ключами типа number и значениями типа string. Сигнатура индекса типа свойства должна быть строкой или числом.

При индексации с помощью number, JS преобразует его в string, поэтому тип, возвращаемый из числового индексатора, должен быть подтипом типа, возвращаемого строковым индексатором.

Сигнатуры можно использовать вместе с обычными свойствами. В таком случае тип их значения должен быть объединением всех типов значений обычных свойств.

Сигнатуры индексов можно использовать вместе с модификатором readonly.

Generic Types

Часто нам требуется использовать похожие объектные типы, отличающиеся одним или несколькими типами свойств. TypeScript позволяет реализовать подобное поведение с помощью общих типов (generic types). Такие типы определяются в угловых скобках после имени интерфейса или синонима:

```
type Box<Type> = {
  contents: Type
}
```

```
function setContents<Type>(box: Box<Type>, newContents: Type) {
  box.contents = newContents
}
```

В отличие от интерфейсов, синонимы могут описывать не только объектные, но и любые другие типы. Используя это вместе с дженериками, мы можем создать следующие типы:

```
type OrNull<Type> = Type | null

type OneOrMany<Type> = Type | Type[]

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>    // null | Type | Type[]

type OneOrManyOrNullStrings = OneOrManyOrNull<string>   // null | string | string[]
```

Общие типы также можно использовать вместе с ограничениями.

Classes

TypeScript также позволяет типизировать и классы. Чтобы определить типы полей, достаточно указать их через двоеточие. Аннотация типа является необязательной, по умолчанию поля имеют тип any.

```
class Point {
  x: number
  y: number
}
```

Поля также могут иметь начальные значения. В этом случае типы выводятся автоматически.

```
class Point {
  x = 0
  y = 0
}
```

Вместе с полями можно использовать модификатор readonly. Он запрещает присваивать полю значения вне конструктора:

```
class Greeter {
  readonly name: string = 'Hello!'

  constructor(otherName?: string) {
    if (otherName !== undefined) {
      this.name = otherName
    }
  }

  err() {
    this.name = 'newName' // Error
  }
}
const g = new Greeter()
g.name = 'newName'        // Error
```

Классы также могут определять сигнатуры индекса, работающие аналогично сигнатурам других объектных типов:

```
class MyClass {
  [s: string]: boolean | ((s: string) => boolean)
}
```

Constructors

Конструкторы класса похожи на функции. В мы можем аннотировать их параметры, добавлять значения по умолчанию, а также перегружать. Но между сигнатурами конструктора класса есть некоторые отличия:

- Конструкторы не могут иметь параметры типа (но класс - может)
- Конструкторы не могут иметь аннотацию возвращаемого типа - это всегда экземпляр класса.

Так же, как и в JS, при наличии суперкласса, перед использованием this обязателен вызов super(). Несоблюдение этого правила TypeScript приравнивает к ошибке.

Methods

Метод - свойство класса, значением которого является функция. Аннотировать их можно точно так же, как и функции с конструкторами:

```
class Point {
  x = 10
  y = 10

  scale(n: number): void {
    this.x *= n
    this.y *= n
  }
}
```

В теле метода к полям и методам класса необходимо обращаться через this. Иначе название будет указывать на лексическое окружение:

```
let x: number = 0
class C {
  x: string = 'привет'
  m() {
    x = 'world' // Error! Type 'string' is not assignable to type 'number'.
  }
}
```

Accessors

Классы могут иметь акцессоры (вычисляемые свойства, accessors):

В TypeScript для них существуют особые правила:

- Если set отсутствует, свойство автоматически становится readonly
- Параметр типа сеттера предполагается на основе типа, возвращаемого геттером
- Если параметр сеттера имеет аннотацию типа, она должна совпадать с типом, возвращаемым геттером
- Геттеры и сеттеры должны иметь одинаковую видимость членов

```
class C {
  _length = 0
  get length() {
    return this._length
  }
  set length(value) {
    this._length = value
  }
}
```

Implements

Ключевое слово implements используется для проверки, соответствует ли класс определенному интерфейсу. При несовпадении сигнатур возникает ошибка:

```
interface Pingable {
  ping(): void
}
class Sonar implements Pingable {
  ping() {
    console.log('ping!')
  }
}
class Ball implements Pingable {
  pong() {                  // Property 'ping' is missing in type 'Ball' but required in type 'Pingable'.
    console.log('pong!')
  }
}
```

Классы могут реализовывать несколько интерфейсов одновременно. В таком случае интерфейсы указываются через запятую.

Implements только проверяет соответствие сигнатур, но не изменяет тип класса.

Extends

В JavaScript классы могут расширяться другими классами с помощью ключевого слова extends. Расширенный класс получает все свойства и методы базового, а также может определять свои собственные.

Класс может перезаписывать свойства и методы своего суперкласса. Для доступа к методам базового класса используется синтаксис *super*.

TypeScript обеспечивает соответствие класса подтипу суперкласса. Нарушение контракта приведет к ошибке:

```
class Base {
  greet() {
    console.log('Hello!')
  }
}
class Derived extends Base {
  greet(name: string) { // Error! Type '(name: string) => void' is not assignable to type '() => void'.
    console.log(`Привет, ${name.toUpperCase()}`)
  }
}
```

Class members visibility

TypeScript может использоваться для определения видимости полей класса за пределами класса. Для этого он предоставляет 3 модификатора видимости, указывать которые необходимо перед именем поля:

- public - публичные члены класса доступны везде. Является значением по умолчанию, указывать его не обязательно.
- protected - защищенные члены доступны только самому классу, в котором они определены, и его подклассам.
- private - частные члены доступны только классу, в котором определены.

```
class SomeClass {
  public a = 42          // Optional use
  protected b = '42'
  private c = false
}
```

Class static members

В классах доступно определение статических членов. Они привязываются не к конкретным экземплярам, а к самому классу. Чтобы определить статический член, используется модификатор static:

```
class MyClass {
  static x = 0
  static printX() {
    console.log(MyClass.x)
  }
}
console.log(MyClass.x)
MyClass.printX()
```

Статические члены могут использоваться вместе с модификаторами видимости.

Статические члены могут наследоваться.

Классы в JavaScript являются функциями, вызываемыми с помощью ключевого слова new, поэтому некоторые слова нельзя использовать в качестве названий статических членов класса. Например, name, length и call.

Generic classes

Класса, как и интерфейсы, могут использоваться с дженериками. Они также указываются в угловых скобках после имени класса:

```
class Box<Type> {
  contents: Type
  constructor(value: Type) {
    this.contents = value
  }
}

const b = new Box('Hello!') // Box<string>
```

Как и в интерфейсах, в классах допускается использование ограничений дженериков, а также значений по умолчанию.

Статические члены не могут ссылаться на параметры типа класса. Иначе предполагалось бы разное поведение статических членов в зависимости от дженерика. Так как при компиляции типы полностью удаляются, подобное поведение невозможно.

Abstract classes & members

В TypeScript классы и их поля могут быть абстрактными. Абстрактными называются члены класса, не имеющие реализации. Они должны находиться внутри абстрактного класса. Такой класс не может инстанцироваться напрямую. Такие классы выступают в роли базовых классов (суперклассов).

```
abstract class Base {
  abstract getName(): string

  greet() {
    console.log('Hello!')
  }
}
class Derived extends Base {
  getName() {
    return 'name'
  }
}
```

Иногда нам требуется работать с конструкторами, создающими какой-нибудь экземпляр класса, производный от абстрактного. Так как экземпляр абстрактного класса нельзя создать, в этом случае нельзя просто использовать его тип. Вместо этого необходимо использовать сигнатуру конструктора:

```
function someFunction(ctor: typeof Base) {
  const instance = new ctor()
  // Cannot create an instance of an abstract class.
}
```

```
function someFunction(ctor: new () => Base) {
  const instance = new ctor()
  instance.greet()
}
```

Relationships between classes

TypeScript сравнивает типы структурно. И классы в данном случае подчиняются тому же правилу. Например, следующие 2 класса имеют одинаковые сигнатуры, поэтому типом одного можно аннотировать экземпляр второго:

```
class Point1 {
  x = 0
  y = 0
}
```

```
class Point2 {
  x = 0
  y = 0
}
```

```
const p: Point1 = new Point2()
```

Подобное поведение справедливо и для подтипов даже при отсутствии явного наследования:

Пустые классы не имеют членов, поэтому в структурном плане могут называться супертипами для всех других классов.

```
class Person {
  name: string
  age: number
}

class Employee {
  name: string
  age: number
  salary: number
}

const p: Person = new Employee()
```

Design Patterns

Patterns

Паттерны описывают типичные способы решения часто встречающихся проблем при проектировании программ.

Виды паттернов:

- **Порождающие** – отвечают за создание объектов
- **Структурные** – отвечают за построение иерархий классов
- **Поведенческие** – решают задачи эффективного взаимодействия между объектами

Module

Модуль – паттерн, задачей которого является предоставление возможности инкапсуляции в JavaScript.

С помощью модулей можно эмулировать приватные методы объектов.

```
const Module = (() => {
    const privateVariable = "Private variable";
    const privateFunc = () => {
        console.log("Private method");
    };
    return {
        publicFunc: () => {
            privateFunc();
            console.log(privateVariable);
        },
    };
})();

Module.publicFunc(); // Private method
// Private variable
```

Singleton

Singleton – порождающий паттерн, позволяющий использовать один экземпляр объекта класса в любом месте кода без необходимости его пересоздания.

Таким образом создается меньше объектов, что уменьшает нагрузку на память.

Этот паттерн подходит для реализации компонентов, используемых в разных частях программы (логгеры, сервисы доступа к базам данных).

Singleton

```
class Logger {
  private static _instance: Logger;
  private constructor() {}
  public static getInstance(): Logger {
    if (!Logger._instance) {
      Logger._instance = new Logger();
    }
    return Logger._instance;
  }
  log(logs: string) {
    console.log(logs);
  }
}

const logger = Logger.getInstance();
logger.log("Some logs..."); // Some logs...
const anotherLogger = Logger.getInstance();
console.log(logger === anotherLogger); // true
```

Adapter

Adapter – структурный паттерн, позволяющий классам с несовместимыми сигнатурами методов взаимодействовать между собой.

Адаптер инкапсулирует адаптируемый класс, при этом предоставляя новые реализации существующих методов.

С помощью этого паттерна можно изменять объекты, возвращаемые сторонней библиотекой или старым кодом, в необходимый формат.

Adapter

```
class DataProvider {
  request() {
    return "file.xml";
  }
}

class XmlToPdfAdapter {
  adaptee: DataProvider;
  constructor(adaptee: DataProvider) {
    this.adaptee = adaptee;
  }
  request() {
    return this.adaptee.request().replace("xml", "pdf");
  }
}

const adapter = new XmlToPdfAdapter(new DataProvider());
console.log(adapter.request()); // file.pdf
```

Decorator

Декоратор – структурный паттерн, позволяющий добавлять классам функциональность оборачивая в “обёртки”.

Применимы, если требуется много различных комбинаций разного функционала.

Decorator

```
interface Component {
  resolve(): string;
}

class ConcreteComponent implements Component {
  public resolve(): string {
    return "text";
  }
}

class Decorator implements Component {
  component: Component;
  constructor(component: Component) {
    this.component = component;
  }
  resolve(): string {
    return this.component.resolve();
  }
}
```

Decorator

```
class ReversingDecorator extends Decorator {
  resolve() {
    return super.resolve().split("").reverse().join("");
  }
}

class LoggingDecorator extends Decorator {
  resolve() {
    const res = super.resolve();
    console.log(res);
    return res;
  }
}
```

Decorator

```
const component = new ConcreteComponent();
console.log(component.resolve()); // text
const decorator = new ReversingDecorator(component);
console.log(decorator.resolve()); // txet
const anotherDecorator = new
LoggingDecorator(decorator);
anotherDecorator.resolve(); // "txet"
```

Strategy

Strategy – поведенческий паттерн, который определяет схожие алгоритмы и помещает их в разные классы, после чего эти алгоритмы можно взаимозаменять во время исполнения программы.

Этот паттерн отделяет алгоритмы от контекста, что позволяет проще их изменять и добавлять новые.

Strategy

```
interface Strategy {
  doAlgorithm(data: string[]): string[];
}

class SortingStrategy implements Strategy {
  doAlgorithm(data: string[]): string[] {
    return data.sort();
  }
}

class ReversingStrategy implements Strategy {
  doAlgorithm(data: string[]): string[] {
    return data.reverse();
  }
}
```

Strategy

```
class Context {
  private strategy: Strategy;
  constructor(strategy: Strategy) {
    this.strategy = strategy;
  }
  setStrategy(strategy: Strategy) {
    this.strategy = strategy;
  }

  doSomeBusinessLogic() {
    const res = this.strategy.doAlgorithm(["b", "z", "f", "e", "j"]);
    console.log(...res);
  }
}
```

Strategy

```
const context = new Context(new SortingStrategy());
context.doSomeBusinessLogic(); // b e f j z
context.setStrategy(new ReversingStrategy());
context.doSomeBusinessLogic(); // j e f z b
```

Command

Команда – поведенческий паттерн, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Command

```
interface Command {
  execute(orders: number[]): number[];
}

class PlaceOrderCommand implements Command {
  order: string;
  id: number;
  constructor(order: string, id: number) {
    this.order = order;
    this.id = id;
  }
  execute(orders: number[]): number[] {
    orders.push(this.id);
    console.log(`Вы заказали ${this.order} (${this.id})`);
    return orders;
  }
}
```

```
class CancelOrderCommand implements Command {
  id: number;
  constructor(id: number) {
    this.id = id;
  }
  execute(orders: number[]): number[] {
    orders = orders.filter((order) => order !== this.id);
    console.log(`Вы отменили ваш заказ ${this.id}`);
    return orders;
  }
}

class TrackOrderCommand implements Command {
  id: number;
  constructor(id: number) {
    this.id = id;
  }
  execute(orders: number[]): number[] {
    console.log(`Ваш заказ ${this.id} прибудет через 20 минут`);
    return orders;
  }
}
```

Command

```
class OrderManager {
  orders: number[];
  constructor() {
    this.orders = [];
  }
  execute(command: Command) {
    this.orders = command.execute(this.orders);
    return;
  }
}

const manager = new OrderManager();
manager.execute(new PlaceOrderCommand("Кофемашина", 2)); // Вы заказали Кофемашина (2)
manager.execute(new TrackOrderCommand(2)); // Ваш заказ 2 прибудет через 20 минут
manager.execute(new CancelOrderCommand(2)); // Вы отменили ваш заказ 2
```

Observer

Наблюдатель – поведенческий паттерн, который создает механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

С помощью этого паттерна обеспечивается низкая связанность компонентов.

Observer

```
class Subject {
  observers: Observer[];
  state: number;
  constructor() {
    this.observers = [];
    this.state = 0;
  }
  attach(observer: Observer) {
    this.observers.push(observer);
  }
  detach(observer: Observer) {
    const observerIndex = this.observers.indexOf(observer);
    this.observers.splice(observerIndex, 1);
  }
  notify() {
    for (const observer of this.observers) {
      observer.update(this);
    }
  }
  someBusinessLogic() {
    this.state = Math.floor(Math.random() * 2 + 1);
    this.notify();
  }
}
```

Observer

```
interface Observer {
  update(subject: Subject): void;
}

class ObserverA implements Observer {
  update(subject: Subject) {
    if (subject.state === 1) {
      console.log("Event 1");
    }
  }
}

class ObserverB implements Observer {
  update(subject: Subject) {
    if (subject.state === 2) {
      console.log("Event 2");
    }
  }
}

const subject = new Subject();
const observer1 = new ObserverA();
subject.attach(observer1);
const observer2 = new ObserverB();
subject.attach(observer2);
subject.someBusinessLogic(); // Event 1 или Event 2
```

Chain of responsibility

Цепочка обязанностей – это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков.

Этот паттерн позволяет выстраивать цепь обработчиков динамически.

Chain of responsibility

```
class Data {
  username: string;
  password: string;
  role: string;
  constructor(username: string,
password: string, role: string) {
    this.username = username;
    this.password = password;
    this.role = role;
  }
}
```

```
interface Handler {
  setNext(handler: Handler): Handler;
  handle(request: Data): void;
}

class AbstractHandler implements Handler {
  nextHandler: Handler;
  setNext(nextHandler: Handler): Handler {
    this.nextHandler = nextHandler;
    return nextHandler;
  }
  handle(request: Data) {
    if (this.nextHandler) {
      return this.nextHandler.handle(request);
    }
    return;
  }
}
```

Chain of responsibility

```
class AuthenticationHandler extends AbstractHandler {
  handle(request: Data) {
    if (request.username !== "Ivan" || request.password !== "1234") {
      console.log("Invalid credentials");
      return;
    }
    console.log("Authentication passed");
    return super.handle(request);
  }
}

class AuthorizationHandler extends AbstractHandler {
  handle(request: Data) {
    if (request.role !== "ADMIN") {
      console.log("Access denied");
      return;
    }
    console.log("Authorization passed");
    return super.handle(request);
  }
}
```

Chain of responsibility

```
const authenticationHandler = new AuthenticationHandler();
const authorizationHandler = new AuthorizationHandler();
authenticationHandler.setNext(authorizationHandler);
authenticationHandler.handle(new Data("Ivan", "1234",
"USER"));
// Authentication passed
// Access denied
```

SOLID

S.O.L.I.D

SOLID – основные принципы объектно-ориентированного программирования

- **S** – Single responsibility principle
- **O** – Open/closed principle
- **L** – Liskov substitution principle
- **I** – Interface segregation principle
- **D** – dependency inversion principle

Single responsibility principle

Принцип единственной ответственности – для каждого класса должно быть определено единственное назначение.

Классы, содержащие много различной функциональности, могут быстро увеличиться в размерах, что усложнит их поддержку.

Такие классы стоит разделить на несколько классов.

Single responsibility principle

```
class Car {
  name: string;
  model: string;
  maxSpeed: number;
  http: HTTP;
  constructor(name: string, model: string, maxSpeed: number, http: HTTP) {
    this.name = name;
    this.model = model;
    this.maxSpeed = maxSpeed;
    this.http = http
  }
  getCar(id: number) {
    return this.http.get("api/cars/" + id);
  }
  saveCar() {
    return this.http.post("api/cars", {
      name: this.name,
      model: this.model,
      maxSpeed: this.maxSpeed,
    });
  }
}
```

Single responsibility principle

```
class Car {
  name: string;
  model: string;
  maxSpeed: number;
  constructor(name: string, model: string, maxSpeed: number) {
    this.name = name;
    this.model = model;
    this.maxSpeed = maxSpeed;
  }
}

class CarService {
  http: HTTP;
  constructor(http: HTTP) {
    this.http = http;
  }
  getCar(id: number) {
    return this.http.get("api/cars/" + id);
  }
  saveCar(car: Car) {
    return this.http.post("api/cars", car);
  }
}
```

Open-closed principle

Принцип открытости/закрытости – сущности должны быть открыты для расширения, но закрыты для модификации.

Появление новых классов/методов не должно влечь за собой изменение уже написанного кода.

Open-closed principle

```
class Rectangle {
  width: number;
  height: number;
  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }
}

class Circle {
  radius: number;
  constructor(radius: number) {
    this.radius = radius;
  }
}
```

Open-closed principle

```
class AreaCalculator {
  calculateRectangleArea(rectangle: Rectangle): number {
    return rectangle.width * rectangle.height;
  }

  calculateCircleArea(circle: Circle): number {
    return Math.PI * (circle.radius * circle.radius);
  }
}
```

Open-closed principle

```
interface Shape {
  calculateArea(): number;
}

class Rectangle implements Shape {
  width: number;
  height: number;
  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }
  calculateArea(): number {
    return this.width * this.height;
  }
}

class Circle implements Shape {
  radius: number;
  constructor(radius: number) {
    this.radius = radius;
  }
  calculateArea(): number {
    return Math.PI * (this.radius * this.radius);
  }
}
```

Open-closed principle

```
class AreaCalculator {
  public calculateArea(shape: Shape): number {
    return shape.calculateArea();
  }
}
```

Liskov substitution principle

Принцип подстановки Лисков – функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом.

Использование наследников классов в функции не должно нарушать работу программы.

Liskov substitution principle

```
class Bird {
  fly() {
    //..
  }
}

class Eagle extends Bird {
  dive() {
    //..
  }
}

class Penguin extends Bird {
  // Пингвины не могут летать
}
```

Liskov substitution principle

```
class Bird {
  layEgg() {
    // ..
  }
}

class FlyingBird {
  fly() {
    // ..
  }
}

class SwimmingBird extends Bird {
  swim() {
    // ..
  }
}

class Eagle extends FlyingBird {
  // ..
}

class Penguin extends SwimmingBird {
  // ..
}
```

Interface segregation principle

Принцип разделения интерфейса – много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.

Если классы, реализующие интерфейс, используют не все методы этого интерфейса, то стоит подумать о разделении этого интерфейса.

Interface segregation principle

```
interface ReportService {
  createEmployeeReport(employeeId: string): Report;
  createCustomerReport(customerId: string): Report;
  createManagementReport(projectId: string): Report;
}

class EmployeeReportService implements ReportService {
  createEmployeeReport(employeeId: string): Report {
    return {
      // ...
    };
  }
  createCustomerReport(customerId: string): Report {
    throw new Error("Method not implemented.");
  }
  createManagementReport(projectId: string): Report {
    throw new Error("Method not implemented.");
  }
}
```

Interface segregation principle

```
interface EmployeeReportService {
  createEmployeeReport(employeeId: string): Report;
}

interface CustomerReportService {
  createCustomerReport(customerId: string): Report;
}

interface ManagementReportService {
  createManagementReport(projectId: string): Report;
}

class EmployeeReportServiceImplementation implements
EmployeeReportService {
  createEmployeeReport(employeeId: string): Report {
    // ...
  }
}

// ..
```

Dependency inversion principle

Принцип инверсии зависимостей – модули верхних уровней не должны зависеть от модулей низких уровней.

Все модули должны зависеть от абстракций.

Абстракции не должны зависеть от деталей.

Детали должны зависеть от абстракций.

Dependency inversion principle

```
import { Logger } from "./logger";

class UserService {
  private logger = new Logger();

  async getAll() {
    try {
      this.logger.log("Retrieving all users...");
      return [];
    } catch (error: any) {
      this.logger.log(`An error occurred: ${error?.message}`);
      throw new Error("Something went wrong");
    }
  }
}
```

Dependency inversion principle

```
export interface ILogger {
  log(message: string): void;
  error(message: string): void;
}
```

Dependency inversion principle

```
class UserService {
  private logger: ILogger;
  constructor(logger: ILogger) {
    this.logger = logger;
  }
  async getAll() {
    try {
      this.logger.log("Retrieving all users...");
      return [];
    } catch (error: any) {
      this.logger.log(`An error occurred: ${error?.message}`);
      throw new Error("Something went wrong");
    }
  }
}
```

Темы для докладов

1) Utility Types: Record, Partial, Pick, Omit, Exclude - с реализацией.
   https://www.typescriptlang.org/docs/handbook/utility-types.html

2) Декораторы.
   https://www.typescriptlang.org/docs/handbook/decorators.html
   https://habr.com/ru/articles/494668/