Абстрактный класс (abstract class)

Если класс или метод класса имеет спецификатор abstract, вы не можете создать экземпляр класса. Абстрактный класс предназначен для использования в качестве суперкласса, в котором реализованы или определены некоторые, но не все, составляющие. Он также называется базовым классом.

Пояснение -

1 - Что такое класс?

Класс - это шаблон для создания объектов с похожим поведением и свойствами (переменными и методами). Чтобы получить объект с реальными значениями, необходимо создать экземпляр класса. Классы являются иерархическими, то есть класс может наследовать информацию от родительского класса (суперкласса) и передавать эту информацию своим дочерним элементам (подклассам). Классы могут представлять собой настраиваемые типы, определяемые пользователем.

Пример класса в коде на verse -

| class Car {  // Свойства класса (переменные)  var brand: string; // Бренд машины  var model: string; // Модель машины  var speed: int = 0; // Текущая скорость   // Метод для разгона машины  fn accelerate(amount: int) -> void {  speed = speed + amount;  }   // Метод для остановки машины  fn stop() -> void {  speed = 0;  } } |
| --- |

Что происходит в коде - brand, model, speed - это переменные, хранящие информацию о машине. Они заданы в типе данных str и int. accelerate - метод для увеличения скорости машины. stop - метод для остановки машины.

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА (ОБЪЕКТ)

| let myCar = new Car(); // Создаём экземпляр класса (объект)  myCar.brand = "Tesla"; // Присваиваем значение свойству brand myCar.model = "Model S"; myCar.accelerate(50); // Разгоняем машину |
| --- |

new Car() - создаем экземпляр класса Car, то есть объект myCar.

myCar.brand = “Tesla” - мы задаём конкретное значение для свойства brand.

myCar.accelerate(50); – вызываем метод accelerate(), который увеличивает скорость.

### **Разница между классом и экземпляром**

| Класс | Экземпляр(объект) |
| --- | --- |
| Шаблон, который описывает структуру объекта. | Конкретный объект, созданный по шаблону класса. |
| Не содержит реальных данных. | Хранит реальные значения в переменные. |
| Не может быть использован напрямую. | Можно вызвать методы и изменить св-ва. |

Пример с кода -

Car (класс) - это типо чертежа машины, а

myCar (экземпляр) - это реальная машина сделанная по чертежу.

Каждый раз, когда ты создаешь new Car() - ты получаешь **новую машину** с теми же характеристиками, но с уникальными значениями свойств.

2 - Что такое метод?

Метод — это функция, прикрепленная к классу, которая может быть вызвана только для экземпляра этого класса.

Из примера выше можно сделать вывод, что

**fn accelerate(amount: int) -> void { ... }**

* Это метод, который принимает число (amount (типа int в данном случае)) и увеличивает скорость (speed).
* **void** означает, что метод **ничего не возвращает**.

**fn stop() -> void { ... }**

* Этот метод просто обнуляет скорость.

Пример кода -

| let myCar = new Car(); // Создаём объект машины  myCar.accelerate(50); // Увеличиваем скорость на 50  myCar.stop(); // Останавливаем машину |
| --- |

Что здесь происходит -

**new Car();** – создаём экземпляр класса Car.

**myCar.accelerate(50);** – вызываем метод accelerate(), увеличивая скорость на 50.

**myCar.stop();** – вызываем метод stop(), сбрасывая скорость в 0.

Виды методов -

| Виды метода | Как вызывается | Когда используется |
| --- | --- | --- |
| Обычный метод | myCar.accelerate(50); | Когда метод работает с данными **конкретного объекта**. |
| Статический метод | Car.getInfo(); | Когда метод работает **без создания объекта**. |

Пример статического метода -

| class Car { // Статический метод  static fn getInfo() -> string {  return "Это класс машин!";  }  } // Вызов статического метода  Car.getInfo(); // Можно вызвать без создания объекта |
| --- |

**Важно: Обычные методы работают с конкретными объектами, а статические методы – с классом в целом.**

Вывод -

1. метод - это функция внутри класса
2. Методы могут изменять данные объекта и вызываться только у экземпляра.
3. Есть обычные методы (для объектов) и статические методы (для класса в целом).

3 - Что такое спецификатор (specifier)?

Спецификаторы в языке программирования Verse — это ключевые слова, которые добавляют дополнительную информацию к определению классов, методов или переменных. Они могут изменять поведение объекта, функции или класса, а также контролировать доступ к данным и их взаимодействие с системой. Спецификаторы можно разделить на две подкатегории: спецификаторы видимости и спецификаторы эффектов.

## 

## **виды спецификаторов**

Спецификаторы видимости (Visibility Specifiers)

Спецификаторы видимости контролируют **доступ** к данным, методам и классам из других частей программы. Они позволяют или ограничивают доступ к элементам в зависимости от того, где они используются.

| Спецификатор | Описание |
| --- | --- |
| public | Разрешает доступ к элементу извне класса и модуля |
| private | Запрещает доступ к элементу извне, элемент доступен только внутри того же класса или модуля. |
| protected | Элемент доступен внутри класса и его подклассов (наследников). |

Пример кода -

| class Player {  public var name: string; // Доступно в любом коде  private var health: int; // Доступно только внутри класса Player  protected var level: int; // Доступно внутри класса и его подклассов } |
| --- |

Спецификаторы эффектов

Спецификаторы эффектов определяют, как функция взаимодействует с состоянием программы и каковы её побочные эффекты.

| Спецификатор | Описание |
| --- | --- |
| pure | Функция не изменяет состояние программы и всегда возвращает одинаковый результат при одинаковых входных данных. |
| mutating | Функция изменяет состояние объекта или переменной. |
| decides | Функция возвращает случайный или неопределённый результат, например, при случайных событиях. |

Пример кода с pure и mutating -

| class Counter {  var count: int = 0;    // pure: всегда даёт один и тот же результат  fn getDoubleValue(x: int) : pure int {  return x \* 2;  }    // mutating: изменяет состояние count  fn increment() : mutating void {  count = count + 1;  } } |
| --- |

**getDoubleValue()** — это чистая (pure) функция, которая всегда возвращает один и тот же результат при одинаковом входе.

**increment()** — это изменяющая (mutating) функция, которая изменяет значение count.

Пример кода с decides -

| fn flipCoin() : decides bool {  // Функция случайно выбирает true или false } |
| --- |

**decides** означает, что функция возвращает неопределённый результат, например, случайное значение.

Спецификаторы abstract

Спецификатор **abstract** используется для создания абстрактных классов и методов. Абстрактные классы не могут быть инстанцированы напрямую; вместо этого они служат шаблоном для создания других классов. Абстрактные методы определяют интерфейс для подклассов, которые должны реализовать эти методы.

* Абстрактный класс содержит абстрактные методы (без реализации), которые обязательно должны быть реализованы в подклассах.
* Абстрактные классы и методы помогают создавать общие шаблоны для различных объектов.

пример кода -

| abstract class Shape {  // Абстрактный метод: не имеет реализации  fn area() : abstract float;    // Конкретный метод: с полной реализацией  fn display() -> void {  print("Displaying shape...");  } }  class Circle : Shape {  var radius: float;    // Реализация абстрактного метода  fn area() : float {  return 3.14 \* radius \* radius;  } }  class Square : Shape {  var side: float;    // Реализация абстрактного метода  fn area() : float {  return side \* side;  } } |
| --- |

* **Shape** — абстрактный класс, который содержит абстрактный метод **area()**, который должен быть реализован в дочерних классах.
* **Circle** и **Square** — это конкретные классы, которые реализуют метод **area()** для вычисления площади соответствующих фигур.
* **display()** — это обычный метод, который уже имеет свою реализацию.

Итог -

**Спецификаторы видимости** (public, private, protected) **контролируют доступ** к данным и методам.

**Спецификаторы эффектов** (pure, mutating, decides) определяют, **как функция влияет на состояние** программы.

**Спецификатор abstract** позволяет **создавать абстрактные классы и методы**, которые задают **структуру** для других классов и не могут быть использованы напрямую.

3 - Что такое экземпляр (instance) и создание экземпляра (instantiate) (**ПОДРОБНО В ПЕРВОМ ПУНКТЕ**)?

**Экземпляр** класса — это **конкретный объект**, созданный на основе шаблона (класса). Экземпляр обладает всеми **характеристиками и поведением** класса, но с **реальными значениями** для его свойств. Это позволяет работать с данными, которые представляют **конкретные сущности**.

**Пример**: Представьте, что класс **Car** описывает шаблон автомобиля с такими характеристиками, как цвет и марка. Каждый экземпляр этого класса будет представлять конкретный автомобиль с конкретными значениями для этих характеристик.

Создание экземпляра (instantiate)

**Создание экземпляра** или **инстанцирование** означает создание **конкретного объекта** из шаблона класса. Для этого используется специальный синтаксис, который обычно включает в себя вызов конструктора класса, если таковой имеется.

Более подробный пример -

| class Car {  var make: string; // марка автомобиля  var model: string; // модель автомобиля  var year: int; // год выпуска автомобиля    // Конструктор класса  fn init(make: string, model: string, year: int) -> void {  self.make = make;  self.model = model;  self.year = year;  }   // Метод для отображения информации о машине  fn displayInfo() -> void {  print("Car: " + make + " " + model + " (" + year.toString() + ")");  } }  fn main() {  // Создание экземпляра класса Car  var myCar = Car("Toyota", "Camry", 2021);    // Вызов метода экземпляра  myCar.displayInfo(); // Выведет: Car: Toyota Camry (2021) } |
| --- |

**Класс Car**:

* В этом примере **Car** — это шаблон (класс) для создания объектов автомобилей. Он имеет три переменные: **make** (марка), **model** (модель) и **year** (год выпуска).
* Конструктор init используется для инициализации этих переменных.

**Создание экземпляра**:

* В функции **main** создаётся **экземпляр класса Car** с конкретными значениями: **"Toyota"** для марки, **"Camry"** для модели и **2021** для года выпуска.
* Экземпляр myCar теперь содержит все характеристики, определённые в классе, но с реальными значениями.

**Метод экземпляра**:

* После создания экземпляра, мы вызываем метод **displayInfo()** для вывода информации о машине. Этот метод использует данные, привязанные к конкретному экземпляру **myCar**.

3 - Что такое суперкласс (superclass) и подкласс (subclass)

Что такое суперкласс?

**Суперкласс** — это **класс**, который предоставляет общие **свойства** (поля) и **методы** для других классов. Он служит **основой** для других классов, называемых **подклассами**. Когда один класс наследует другой, то класс, от которого наследуется поведение и свойства, называется **суперклассом**. Класс, который наследует, называется **подклассом**.

Суперкласс включает в себя **все поля** и **методы**, которые могут быть использованы в **подклассах**. Это позволяет создавать **иерархию классов**, где общий функционал определён в одном месте, а конкретные детали — в подклассах.

Наследование

**Наследование** — это механизм, при котором **подкласс** получает **свойства и поведение** от **суперкласса**. Это позволяет использовать повторно код суперкласса, минимизируя дублирование.

* **Суперкласс** определяет общие характеристики и поведение, которые могут быть унаследованы.
* **Подкласс** может расширять или модифицировать поведение суперкласса, добавляя новые характеристики или переопределяя методы.

Пример кода -

| // Суперкласс Vehicle class Vehicle {  var make: string; // марка  var model: string; // модель    fn init(make: string, model: string) -> void {  self.make = make;  self.model = model;  }   // Метод, который можно переопределить в подклассах  fn displayInfo() -> void {  print("Vehicle: " + make + " " + model);  } }  // Подкласс Car, который наследует от Vehicle class Car : Vehicle {  var doors: int; // количество дверей    fn init(make: string, model: string, doors: int) -> void {  // Вызов конструктора суперкласса  super.init(make, model);  self.doors = doors;  }   // Переопределение метода displayInfo для конкретного класса  fn displayInfo() -> void {  print("Car: " + make + " " + model + " with " + doors.toString() + " doors");  } }  // Подкласс Bike, который наследует от Vehicle class Bike : Vehicle {  var type: string; // тип велосипеда   fn init(make: string, model: string, type: string) -> void {  // Вызов конструктора суперкласса  super.init(make, model);  self.type = type;  }   // Переопределение метода displayInfo для конкретного класса  fn displayInfo() -> void {  print("Bike: " + make + " " + model + " of type " + type);  } }  fn main() {  var myCar = Car("Toyota", "Camry", 4);  var myBike = Bike("Giant", "Escape", "Mountain");   // Вызов методов displayInfo для каждого экземпляра  myCar.displayInfo(); // Выведет: Car: Toyota Camry with 4 doors  myBike.displayInfo(); // Выведет: Bike: Giant Escape of type Mountain } |
| --- |

Пояснение -

**Суперкласс Vehicle**:

* Это класс, который предоставляет **общие поля** make (марка) и model (модель) для всех типов транспортных средств.
* Метод **displayInfo()** в суперклассе выводит общую информацию о транспортном средстве.

**Подклассы Car и Bike**:

* Эти классы **наследуют** поведение от суперкласса **Vehicle**.
* В **Car** добавлено поле doors (количество дверей) и переопределён метод **displayInfo()**, чтобы выводить информацию о машине.
* В **Bike** добавлено поле type (тип велосипеда) и переопределён метод **displayInfo()**, чтобы выводить информацию о велосипеде.

**Использование super**:

* В каждом из подклассов вызывается конструктор суперкласса с помощью ключевого слова **super**, что позволяет инициализировать общие поля, такие как make и model.

ВЫВОД ПРОГРАММЫ -

| Car: Toyota Camry with 4 doors Bike: Giant Escape of type Mountain |
| --- |

### **Объяснение:**

* **Суперкласс Vehicle** определяет общий функционал для всех транспортных средств.
* Подклассы **Car** и **Bike** **расширяют** функциональность, добавляя специфичные характеристики (например, количество дверей у машины или тип велосипеда), а также **переопределяют** метод displayInfo для вывода информации, соответствующей их типу.

### **Итог:**

* **Суперкласс** предоставляет общую структуру и функциональность для других классов (подклассов).
* **Подклассы** могут **наследовать** эти свойства и методы, а также добавлять или изменять их для реализации более конкретных деталей.
* **Наследование** позволяет **повторно использовать код** и минимизировать дублирование, создавая иерархию классов с общими и специализированными характеристиками.

3 - Что такое составляющие (member)

В контексте программирования, **составляющая** (member) — это **элемент**, который является частью **структуры данных**, такой как **класс**, **структура**, **интерфейс** или **модуль**. Это может быть как **переменная**, так и **функция**.

#### **Типы составляющих:**

1. **Переменные (Fields)**:
   * Переменные, которые хранят **данные** или **состояние** объекта.
   * Они часто называются **полями**.
2. **Методы (Methods)**:
   * Функции, которые выполняют действия с данными объекта или класса.
   * Эти функции называют **методами**, так как они принадлежат классу или объекту.

Каждый объект или класс может иметь несколько составляющих, и эти составляющие определяют, как объект будет себя вести или какие данные он будет хранить.

Пример кода -

| class Person {  // Поля класса (составляющие переменные)  var name: string; // имя  var age: int; // возраст   // Конструктор, который инициализирует поля  fn init(name: string, age: int) -> void {  self.name = name;  self.age = age;  }   // Метод (составляющая функция), который выводит информацию о человеке  fn displayInfo() -> void {  print("Name: " + name + ", Age: " + age.toString());  }   // Метод для увеличения возраста  fn celebrateBirthday() -> void {  age = age + 1;  } }  fn main() {  // Создание экземпляра класса Person  var person = Person("Alice", 30);    // Вызов метода displayInfo() для отображения информации  person.displayInfo(); // Выведет: Name: Alice, Age: 30   // Вызов метода celebrateBirthday() для увеличения возраста  person.celebrateBirthday();   // Вновь вызываем displayInfo() после изменения возраста  person.displayInfo(); // Выведет: Name: Alice, Age: 31 } |
| --- |

Пояснение кода -

**Поля (составляющие переменные)**:

* **name** и **age** — это переменные (поля) класса **Person**, которые хранят данные о человеке (имя и возраст). Эти поля называются **составляющими переменными** или просто **переменными класса**.

**Методы (составляющие функции)**:

* **displayInfo()** — это метод, который выводит информацию о человеке, используя данные из полей **name** и **age**.
* **celebrateBirthday()** — это метод, который увеличивает возраст человека на 1.

**Экземпляр класса**:

* Когда создаётся объект **person**, он получает конкретные значения для полей: **"Alice"** для имени и **30** для возраста.

**Взаимодействие с составляющими**:

* В примере создаётся экземпляр класса **Person**. Мы вызываем метод **displayInfo()**, который использует поля класса для вывода информации.
* Также вызывается метод **celebrateBirthday()**, который изменяет поле **age**, увеличив его на 1.

Итог -

* **Составляющая** (member) — это **переменная** или **метод**, который является частью класса, структуры, интерфейса или модуля.
* **Переменные** класса обычно называются **поля** и содержат данные, которые определяют состояние объекта.
* **Методы** класса выполняют операции с этими данными и определяют поведение объекта.

Составляющие классов играют важную роль в моделировании данных и поведения объектов в объектно-ориентированном программировании.

ОКЕЙ МЫ ВРОДЕ РАЗОРАЛМ ОСНОВНЫЕ ПОНЯТИЯ, НО ОСТАЛОСЬ ТОЛЬКО ПОНЯТЬ ЧТО ТАКОЕ АБСТРАКЦИЯ И ПРИВЕСТИ ПРИМЕР КОДА

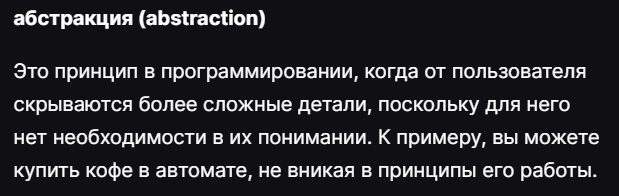


РИС. С ДОКУМЕНТАЦИИ EPIC GAMES

**Абстрактный класс (abstract class)**

Абстрактный класс — это класс, который не может быть инстанцирован напрямую. Он предназначен для использования в качестве базового (суперкласса) для других классов. Абстрактный класс может содержать как реализованные методы, так и абстрактные методы (методы без реализации), которые должны быть реализованы в подклассах.

Абстрактные классы часто используются для создания общих интерфейсов или шаблонов, которые должны быть доработаны в конкретных подклассах.

### Пример абстрактного класса в Verse

Предположим, у нас есть абстрактный класс Animal, который служит базой для конкретных животных, таких как Dog и Cat.

пРИМЕР кода ниже :

| // Абстрактный класс Animal abstract class Animal {  var name: string; // Имя животного   // Абстрактный метод (без реализации)  abstract fn makeSound() -> void;   // Конструктор класса  fn init(name: string) -> void {  self.name = name;  } }  // Подкласс Dog, наследующий от Animal class Dog : Animal {  // Реализация абстрактного метода makeSound  fn makeSound() -> void {  print(name + " says Woof!");  } }  // Подкласс Cat, наследующий от Animal class Cat : Animal {  // Реализация абстрактного метода makeSound  fn makeSound() -> void {  print(name + " says Meow!");  } }  fn main() {  // Создание экземпляров подклассов  var dog = Dog("Buddy");  var cat = Cat("Whiskers");   // Вызов методов  dog.makeSound(); // Выведет: Buddy says Woof!  cat.makeSound(); // Выведет: Whiskers says Meow! } |
| --- |

Пояснение кода:

**Абстрактный класс Animal**:

* Это абстрактный класс, который не может быть инстанцирован напрямую (например, мы не можем создать объект типа Animal).
* Класс **Animal** содержит абстрактный метод **makeSound()**, который **не имеет реализации** в самом абстрактном классе. Это обязательство для всех подклассов, чтобы реализовать этот метод.
* Также в абстрактном классе есть конструктор **init(name: string)**, который инициализирует поле **name**.

**Подклассы Dog и Cat**:

* Оба эти класса наследуют от абстрактного класса **Animal** и реализуют абстрактный метод **makeSound()**, предоставляя собственную реализацию для каждого типа животного.

**Создание экземпляров**:

* Мы не можем создать экземпляр класса **Animal** напрямую, так как это абстрактный класс.
* Однако, мы можем создать экземпляры подклассов **Dog** и **Cat**, которые реализуют все необходимые методы.

вывод

**Абстрактный класс** — это класс, который не может быть использован напрямую для создания объектов. Он служит как **шаблон** или **интерфейс** для других классов.

Абстрактные методы, определённые в абстрактном классе, **не имеют реализации** и должны быть реализованы в подклассах.

Это позволяет создавать **общие интерфейсы** для разных типов объектов, оставляя детали реализации на усмотрение подклассов.