Введение в ООП. Часть 1

ООП

Объектно-ориентированное программирование (ООП) представляет собой подход в разработке программного обеспечения, где основной упор делается на использование **объектов** — структур данных, объединяющих в себе как **состояние**, так и **поведение**, соответствующее этому состоянию. Основная идея заключается в том, что программы строятся как взаимодействующие между собой коллекции объектов, каждый из которых является э**кземпляром определенного класса**. Классы в этом контексте выступают как **шаблоны** для создания объектов, определяя их атрибуты и методы. ООП облегчает структурирование и понимание программ, способствует повторному использованию кода через механизмы наследования и полиморфизма, а также упрощает процесс отладки и обслуживания программного обеспечения.

В дополнение к объектно-ориентированному подходу, существуют и другие парадигмы программирования, такие как **процедурное** программирование, **функциональное** программирование и **логическое** программирование.

Процедурное программирование фокусируется на выполнении последовательностей вычислительных операций или процедур; оно предшествовало ООП и акцентирует внимание на написании функций и их взаимодействии.

Функциональное программирование, в свою очередь, основывается на использовании функций как первоклассных граждан и стремится к минимизации изменяемого состояния и побочных эффектов, что обеспечивает высокую степень модульности и возможности для параллельных вычислений.

Логическое программирование основано на формализации логики задачи, где программа составляется как набор фактов и правил, позволяя системе самостоятельно строить выводы и решать задачи.

Классы

Класс в объектно-ориентированном программировании выступает как **абстрактный** тип данных и шаблон, предназначенный для создания объектов. Это ключевая концепция, позволяющая разработчикам определять пользовательские типы данных, внутри которых могут быть описаны разнотипные переменные, известные как **поля** класса. Поля определяют состояние, которым будут обладать объекты, созданные на основе данного класса, предоставляя тем самым структуру данных для моделирования реальных или

абстрактных сущностей в программе. Принцип абстракции позволяет выделить и описать только те характеристики сущности, которые важны для решения конкретной задачи, игнорируя нерелевантные детали.

```
class Human {

        // поля
        int age;
        boolean isEmployed;
        String name;
}
```

Класс **`Human`**, представленный в примере, демонстрирует, как может быть реализована абстракция в программировании. В этом классе определены поля, такие как **`age`** (возраст), **`isEmployed`** (статус занятости) и **`name`** (имя), которые являются значимыми характеристиками для моделирования человека в контексте решаемой задачи. Например, если задача состоит в вычислении среднего возраста трудоустроенных людей, то поля **`age`** и **`isEmployed`** будут иметь ключевое значение, в то время как другие возможные аспекты, такие как цвет волос или рост, могут быть опущены, поскольку они не влияют на решение задачи.

Классы являются **ссылочными** типами данных, что означает, что переменные класса хранят ссылку на место в памяти, где фактически расположен объект, а не сам объект. Это предоставляет дополнительную гибкость в управлении объектами, например, позволяя множеству переменных ссылаться на один и тот же объект.

Объекты

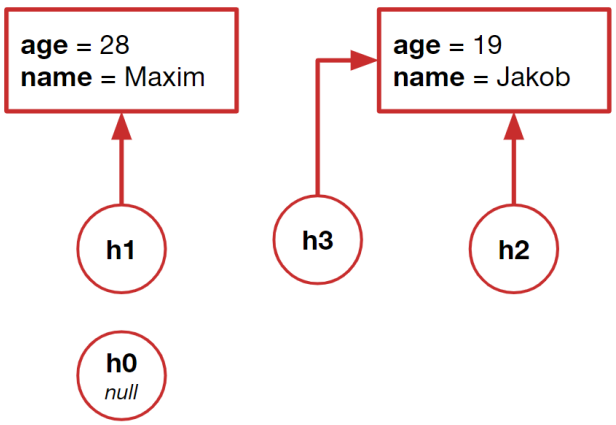
Объект в объектно-ориентированном программировании является конкретным экземпляром класса, который создается и используется для выполнения определенных задач в программе. Каждый объект обладает уникальным состоянием, которое определяется значениями его полей, заданными в соответствии с шаблоном, описанным в классе. Эти поля хранят данные, специфичные для каждого объекта.

Переменные, которые объявляются для хранения ссылок на объекты, неформально называются **объектными переменными**. Они могут принимать значение **null**, что означает отсутствие ссылки на какой-либо объект, либо могут указывать на конкретный объект, созданный с использованием оператора **new** и **конструктора** класса. Таким образом, объектная переменная может либо указывать на реальный объект в памяти, либо быть неинициализированной (null), что отражает отсутствие объекта.

```
// h0, h1, h2 - объектные переменные

Human h0 = null;

// создание объектов
Human h1 = new Human();
Human h2 = new Human();
Human h3 = h2;

h1.age = 28;
h1.name = “Maxim”;

h2.age = 19;
h2.name = “Jakob”;
```

Пример кода демонстрирует создание нескольких объектных переменных и объектов класса **`Human`**. Переменная **`h0`** инициализируется значением **null**, что означает, что она не ссылается ни на один объект. С другой стороны, переменные **`h1`** и **`h2`** используют оператор **`new`** для создания новых объектов класса **`Human`**, каждый из которых имеет собственные уникальные значения полей **`age`** и **`name`**. Переменная **`h3`** демонстрирует, что две разные объектные переменные (**`h2`** и **`h3`**) могут ссылаться на один и тот же объект.

Такое поведение объектных переменных и механизм создания объектов лежит в основе работы с данными в объектно-ориентированных языках программирования, обеспечивая гибкость и мощные возможности для организации и управления состоянием программы.



Конструкторы

Конструктор выполняет **инициализацию** объекта непосредственно после его создания. Этот специальный блок инструкций задает **начальное** состояние объекта, внося начальные значения в его поля, что позволяет объекту быть готовым к использованию

сразу после создания. Инициализация объекта обеспечивает установку определенных значений для всех его атрибутов, гарантируя, что объект начинает свою работу в предсказуемом и контролируемом состоянии.

Конструктор по умолчанию в языках программирования, поддерживающих ООП, таких как Java, — это **конструктор без параметров**, который автоматически генерируется компилятором для любого класса, если только разработчик явно не определил другой конструктор. Этот конструктор не содержит кода инициализации и предоставляет самые базовые условия для создания объекта. Однако, если в классе явно определен один или несколько пользовательских конструкторов с параметрами или без них, компилятор не добавляет конструктор по умолчанию автоматически, ожидая, что разработчик полностью контролирует процесс инициализации объектов своего класса.

```
class Human {

	// поля
	int age;
	boolean isEmployed;
	String name;

	// собственный конструктор без параметров
	Human() {
		age = 18;
		isEmployed = true;
		name = “Anonymous”;
	}
}

// h1 ссылается на работающего человека 18 -ти лет с именем Anonymous
Human h1 = new Human();
```

Пример с классом Human демонстрирует создание собственного конструктора без параметров. В этом конструкторе полям **`age`**, **`isEmployed`** и **`name`** присваиваются начальные значения, таким образом определяя начальное состояние каждого объекта Human как 18-летнего трудоустроенного человека с именем "Anonymous". Это позволяет разработчику обеспечить более глубокий контроль над процессом создания объектов, задавая конкретные начальные характеристики, которые соответствуют логике и требованиям приложения.

Класс может содержать множество конструкторов, каждый из которых отличается набором формальных параметров. Эта возможность называется **перегрузкой конструкторов** и служит для предоставления различных способов инициализации объектов одного и того же класса. Перегруженные конструкторы позволяют создавать экземпляры класса с разными начальными состояниями, в зависимости от предоставленных аргументов или их отсутствия, что обеспечивает гибкость в процессе разработки программ.

```
class Human {
        // …
        // конструктор без параметров
        Human() {
                age = 18;
                isEmployed = true;
                name = “Anonymous”;
        }

        // конструктор с параметрами
        Human(int a, boolean emp, String n) {
                age = a;
                isEmployed = emp;
                name = n;
        }
}

// имеем возможность использовать оба конструктора для создания объектов

Human h1 = new Human();

Human h2 = new Human(33, false, “Max”);
```

В приведенном примере мы видим два различных конструктора: один без параметров и один с параметрами. Конструктор без параметров инициализирует объекты с некоторым предопределенным состоянием, в данном случае устанавливая возраст в 18 лет, статус занятости в `true` и имя в "Anonymous". Это обеспечивает удобный способ создания объектов с стандартным набором начальных значений без необходимости явно указывать их каждый раз при создании нового объекта.

С другой стороны, конструктор с параметрами позволяет явно задать значения для полей `age`, `isEmployed`, и `name` при создании объекта, что дает возможность инициализировать объект с конкретными характеристиками, отличными от стандартных. Это особенно полезно, когда нужно создать объект с уникальными свойствами, отражающими конкретные требования или условия.

this

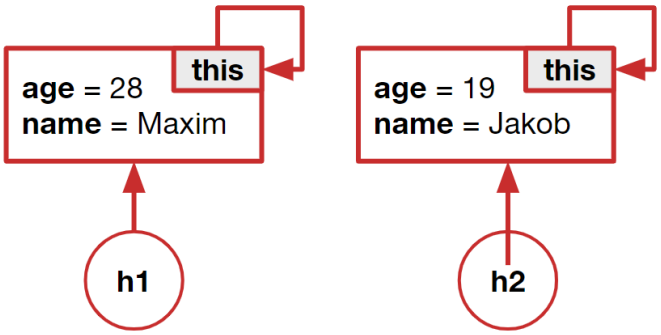
Ключевое слово **this** является важным инструментом, который используется внутри конструкторов и методов класса. Оно служит ссылкой на текущий объект — экземпляр класса, в контексте которого вызван конструктор или метод. Использование **this** позволяет однозначно идентифицировать поля и методы объекта, особенно в ситуациях, когда имена параметров метода или конструктора совпадают с именами полей класса, что может привести к коллизии имен.

```
class Human {
```

```
        // ...

        Human(int age, boolean isEmployed, String name) {
                this.age = age;
                this.isEmployed = isEmployed;
                this.name = name;
        }
}
```

В приведенном примере конструктор принимает три параметра (**`age`**, **`isEmployed`**, **`name`**), имена которых совпадают с именами полей класса. Чтобы правильно инициализировать поля объекта значениями, переданными в конструктор, и избежать путаницы между полями класса и параметрами конструктора, используется ключевое слово **`this`**. Префикс **`this`** перед именем поля (**`this.age`**, **`this.isEmployed`**, **`this.name`**) указывает, что присваивание должно производиться именно полям текущего объекта, а не локальным переменным и параметрам.

Такой подход не только разрешает конфликт имен, но и повышает читаемость кода, делая явным обращение к атрибутам и методам объекта. Использование **this** подчеркивает, что операция или доступ осуществляются в контексте текущего экземпляра класса, что особенно полезно в сложных классах с множеством полей и методов.



Ключевое слово **this** обладает дополнительной функциональностью, позволяя не только обращаться к полям и методам текущего объекта, но и вызывать один конструктор класса из другого. Эта возможность особенно ценна для минимизации дублирования кода при инициализации объектов класса, когда несколько конструкторов выполняют общие операции инициализации, но также включают дополнительные шаги для разных вариантов создания объектов.

```
class Human {
```

```
        // ...
        Human() {
                age = 18;
                isEmployed = true;
                name = “Anonymous”;
        }

        Human(String name) {
                this(); // вызов первого конструктора
                this.name = name; // инициализация имени
        }
}
```

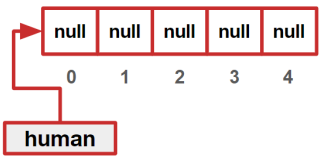
В примере представлен метод вызова конструктора без параметров из конструктора с параметром **`name`**. Конструктор без параметров задает начальные значения для полей **`age`**, **`isEmployed`** и **`name`**, обеспечивая базовую инициализацию объекта. В конструкторе, принимающем параметр name, сначала выполняется вызов конструктора без параметров с помощью **`this(),`** что гарантирует, что все поля объекта будут инициализированы базовыми значениями. Затем, поле **`name`** инициализируется переданным значением, позволяя таким образом дополнить или изменить начальное состояние объекта без необходимости повторного кодирования инициализации общих полей.

Массив объектов

В Java, как и во многих других объектно-ориентированных языках программирования, массив объектов представляет собой мощный инструмент для хранения и управления коллекциями объектов одного типа. Создание массива объектов в Java подразумевает инициализацию массива объектных переменных, где каждая переменная может хранить ссылку на объект определенного класса. Важным моментом при работе с массивами объектов является то, что после инициализации массива все его элементы по умолчанию инициализируются значением **null**, что означает отсутствие ссылки на какой-либо объект.

Пример создания массива объектов класса **`Human`** демонстрирует, как можно выделить память под массив из пяти объектных переменных типа **`Human`**:

```
Human[] humans = new Human[5];
```



На этом этапе каждый элемент массива **`humans`** является **null**, поскольку объекты еще не были созданы и присвоены элементам массива. Для того чтобы использовать элемент массива для хранения объекта, необходимо явно создать объект и присвоить его одному из элементов массива. Это делается с помощью оператора **`new`** и конструктора класса:

```
humans[2] = new Human("Marsel");
```

В этом примере объект класса **`Human`** с именем "Marsel" создается и его ссылка присваивается третьему элементу массива **`humans`** (поскольку индексация элементов массива начинается с нуля). Таким образом, элемент массива **`humans[2]`** теперь ссылается на конкретный объект, в то время как остальные элементы массива до сих пор имеют значение **`null`**, если только для них не были созданы и присвоены другие объекты.