Принципы ООП

Java, как и любой другой язык с поддержкой ООП, реализует следующие принципы:

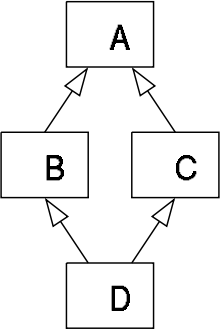
* **Абстракция**. Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. Подразумевается набор значимых характеристик объекта, доступный остальной программе. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации. Это позволяет работать с объектами, не вдаваясь в особенности их реализации.
* **Инкапсуляция -** это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. Код и данные связанные посредством инкапсуляции образуют объект. Инкапсуляция осуществляет сокрытие данных и деталей реализации от конечного пользователя объекта.
* **Наследование -** свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс - потомком, наследником, дочерним или производным классом. Наследование позволяет создавать иерархии классов, что улучшает абстракцию и инкапсуляцию и исключает дублирование кода. Кроме того, наследование является необходимым условием реализации полиморфизма.
* **Полиморфизм** - возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию - например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: "Один интерфейс, множество реализаций".

Множественное наследование

Множественное наследование порождает несколько проблем, понимание которых объясняет почему поддержка ООП в Java реализована именно так, а не иначе.

В Java запрещено множественное наследование (посредством ключевого слова extends), но разрешено множественное наследование интерфейсов (реализация посредством ключевого слова implements).

**Ромбовидное наследование**



(англ. *diamond inheritance*) - ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если метод класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C? Конечно эта проблема решается в современных языках программирования.

В Java было реализовано множественное наследование только для интерфейсов, потому что они не содержат реализаций методов, а значит у метода всегда есть только одна реализация.

Абстрактные классы и методы

Наверное, все языки, которые поддерживаю ООП, поддерживают абстрактные классы. **Абстрактный класс -** это базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике принцип полиморфизма. Такие классы в Java объявляются с помощью ключевого слова abstract.

**Абстрактный метод** - это метод, который не имеет реализации. Смысл абстрактного метода в том чтобы обязать класс-наследник реализовать описанный метод позже.

Синтаксис определения абстрактного класса:

*// Простой пример абстрактного класса.***abstract class** А {  
 **abstract void** callme();  
  
 *// абстрактные классы все же могут содержать конкретные методы* **protected void** callmetoo() {  
 System.***out***.println(**"Это конкретный метод класса А."**);  
 }  
}  
  
**class** В **extends** А {  
 **void** callme() {  
 System.***out***.println(**"Реализация метода callme класса В."**);  
 }  
}

**Особенности абстрактных классов**

* Абстрактный класс обязан наследоваться;
* Абстрактный класс может иметь конструктор, но не может создавать экземпляров;
* Конструктор абстрактного класса не может быть абстрактным. Так происходит потому, что конструктор не может быть переопределен позже, как обычный метод;
* Абстрактный класс может иметь как абстрактные методы, так и обычные;
* Наследник абстрактного класса должен реализовать все абстрактные методы;
* Абстрактный класс может не иметь абстрактных методов
* Хотя абстрактные классы не могут быть использованы для создания объектов, их можно применять для создания ссылок на объекты;

**Приведение типов классов**

Рассмотрим классы из примера выше. Класс A, как родитель, является более универсальным, чем потомок B. Это значит, что любой объект класса потомка может быть явно или автоматически приведён к классу родителя.

То есть вполне корректным будет:

A ob1;  
B ob2 = **new** B();  
ob1 = (A) ob2; *// явное приведение ob1 = ob2; автоматическое приведение*ob1.callme();

Приведение всегда возможно, когда переменную или объект мы пытаемся привести к более универсальному (базовому) типу. Таким образом мы можем сослаться на объект класса-наследника посредством его базового класса.

**Виртуальность методов в Java**

Пускай есть классы:

**class** A {  
 **protected void** call() {  
 System.***out***.println(**"class A called"**);  
 }  
}  
  
**class** B **extends** A {  
 **protected void** call() {  
 System.***out***.println(**"class B called"**);  
 }  
}

Вызовем метод:

A obj = **new** B();  
obj.call();

Возникает вопрос какой из методов будет вызван? Из класса А или класса В? Будет вызван метод класса В, **несмотря на то ссылка имеет тип класса А**.

В Java все нестатические, неприватные (то есть, protected, package и public) методы которые переопределены являются виртуальными.

Ключевое слово final запрещает возможность дальнейшего переопределения метода в подклассах.

Это и есть реализация **полиморфизма**. Какой метод вызывать определяется по типу объекта.

Описанный момент очень важен, поскольку не во всех языках переопределенные методы являются по умолчанию виртуальными.

Например, в С++ переопределенные методы по умолчанию не виртуальны. То есть в описанной ситуации был бы вызван метод класса А. Чтобы сделать их виртуальными, нужно добавить ключевое слово **virtual**.

Интерфейсы в Java

Сложно декларативно, в двух словах, описать что такое интерфейс. Ниже приведены описания с разных точек зрения, сумма которых формирует понятие интерфейса.

# Несколько определений из разных источников

* Назначение интерфейса - **описание** функциональности, которая **должна** быть реализована имплементирующими классами в дальнейшем [1, c. 150].
* Интерфейс определяет имена, список аргументов, возвращаемые значения методов, но не их реализацию. Интерфейс предоставляет только **форму**, но не **реализацию**. Интерфейс говорит: «все классы что реализуют этот конкретный интерфейс будут выглядеть так». То есть любой код, который использует данный интерфейс знает какие методы могут быть вызваны. Можно сказать, что интерфейс устанавливает «протокол» общения между классами [2, c. 222].
* Интерфейс - это регламент взаимодействия. Класс, который реализует интерфейс обязан реализовывать все его методы. В интерфейсе вы описываете лишь сигнатуры методов, то есть вы указываете что класс наследник **должен** уметь делать, но **как** он будет это делать, тот решает сам. Таким образом, вы уверенны что, если класс реализует тот или иной интерфейс, все объекты данного класса имеют определенный набор методов. Интерфейс - это еще одна абстракция (более высокого уровня чем наследование), позволяющая отделить **описание** от **реализации**.

# Примеры интерфейсов в реальном мире

* Интерфейс «способность летать», то есть способность к движению в воздухе. Эта способность присуща бабочкам, птицам, самолетам. На уровне объектов между ними родства нет - бабочка не является самолетом, а самолет не является птицей. То есть реализация у всех сущностей разная, а интерфейс общий.
* Интерфейс «управление наземным транспортным средством» предоставляет органы управления: руль, педали, средства подачи сигналов. При этом транспортное средство может быть реализовано в виде троллейбуса, легкового автомобиля, электропоезда.

# Отличия интерфейсов и абстрактных классов

Интерфейсы тесно связаны с абстрактными классами. Не все языки выделяют интерфейсы как отдельный элемент. Например, в С++ нет интерфейсов, а есть только абстрактные классы. То есть чисто абстрактный класс в С++ (где есть только абстрактные методы) можно рассматривать как интерфейс. Выяснение отличий подводит нас к тому, что такое интерфейс конкретно в языке Java [3], [4].

|  |  |
| --- | --- |
| **Классы** | **Интерфейсы** |
| Абстрактные классы предназначены для **наследования** (**inheritance**), а значит реализуют **«is a»** тип отношений. Например: класс «собака» унаследован от класса «животное». Значит мы можем сказать: «собака» есть (is a) животное. Но обратное неверно. При этом «собака» имеет доступ к свойствам и методам родительской сущности «животное». | Интерфейсы реализуют тип отношений **«has a»**,который называется **композиция** (**composition**). Например: есть класс «пчела» и класс «собака». Эти сущности никак не связаны друг с другом, но обе реализуют интерфейс «возможность укусить». То есть, мы можем сказать: «собака», «пчела» имеют (has a) «возможность укусить». |
| Множественное наследование в Java запрещено для классов. | В Java один класс может реализовывать множество интерфейсов. |
| Абстрактный класс может реализовывать методы, все или только некоторые. | Интерфейс не может реализовывать методы |
| К неабстрактным членам класса применимы любые модификаторы. Однако абстрактные методы (с модификатором abstract) могут иметь либо public, либо protected модификатор. Иными словами, абстрактные методы не могут быть приватными. | Все методы по умолчанию являются публичными (public) и абстрактными (abstract), а поля - public static final. Даже если они таковыми не объявлены. |
| При наследовании абстрактного класса **не все** методы могут быть переопределены. | При реализации интерфейса **все** методы должны быть реализованы. |
| В абстрактном классе можно описать конструктор (или несколько конструкторов). | Интерфейс не может содержать никаких конструкторов |

# Зачем нужны интерфейсы

Причины появления интерфейсов и устранения множественного наследования в Java заключаются в том, что Java стремиться более строго следовать паттернам в объектно-ориентированном проектировании (General Responsibility Assignment Software Patterns - общие шаблоны распределения обязанностей) [5].

Больше всего с классами и интерфейсами связаны такие принципы:

* **High сohesion** (высокое **сцепление**, часто переводится как связность). Понятие, определяющее насколько тесно связаны между собой элементы одного модуля. Можно считать "модулем" класс, а "элементами" - его методы. Тогда **сцепление** определяет, по сути, насколько узкий (специфичный) функционал реализует данный класс.   
    
  Высокое **сцепление** означает что класс выполняет однородные обязанности, то есть лишь свою определённую часть функционала системы. И наоборот, за каждый элемент функционала системы отвечает только один класс.   
    
  Например, если в системе понадобилось изменить политику безопасности, то изменения должны быть сделаны лишь в одном месте программы, отвечающем за проверку пароля на безопасность.

Высокое **сцепление** - это положительная характеристика и его нужно стремиться повысить.

* **Low сoupling** (низкая **связанность**). Это принцип, который позволяет распределить обязанности между объектами таким образом, чтобы они как можно меньше зависели друг от друга. При соблюдении это принципа мы получаем:
  + Малое число зависимостей между классами (подсистемами).
  + Слабая зависимость одного класса (подсистемы) от изменений в другом классе (подсистеме).
  + Высокая степень повторного использования подсистем.

Высокое сцепление и низкая связанность на практике означают простоту обслуживания. То есть, лёгкость модификации существующей программы для изменения существующих и добавления новых возможностей.

Использование интерфейсов понижает связанность в то время как наследование повышает ее. Поддержка интерфейсов на уровне языка обязывает программиста придерживаться паттернов ООП.   
  
Интерфейсы плюс одинарное наследование помогают добиться оптимального баланса между сцеплением и связанностью.

В других языках (например, в C++) есть возможность использовать правильные подходы, паттерны ООП, но это полностью ответственность программиста. Управление сцеплением и связанностью на уровне языка - это неоспоримое преимущество Java.

**Рекомендации по применению** **абстрактных классов и интерфейсов**

* Абстрактный класс уместно использовать, если вы собираетесь использовать наследование, которое будет обеспечивать общую структуру.
* Абстрактный класс также уместно использовать, если вы хотите объявить приватные экземпляры. В интерфейсах, все методы должны быть публичными.
* Если вы считаете, что в будущем вам понадобится прибегать к добавлению новых методов, тогда абстрактный класс – лучший вариант. В случае интерфейса вам необходимо будет объявлять новый метод в каждом классе, который реализует ваш интерфейс.
* Интерфейсы – хороший выбор, если вам известно, что API не будет меняться некоторое время.
* Также, для реализации «множественного наследования» кроме интерфейсов вы не найдете альтернативных решений.
* Если абстрактный класс не содержит никакой частичной реализации, то вместо него лучше использовать интерфейс [4].

**Синтаксис**

Ниже приведен пример описания интерфейса и его реализации.

public interface Love {  
 bolean initLove(int money);

bolean initLove(Feelings feelings);

Pleasure makeLove();  
}

class Man implements Love {

bolean initLove(int money) {

…

}

bolean initLove(Feelings feelings) {

…

}

Pleasure makeLove(){

…

}

}

1. JAVA Methods Programming v2, 2015, Блинов, Романчик, с. 150, с. 253
2. Thinking in Java 4th Edition, Bruce Eckel, c. 222, с. 297
3. <http://www.quizful.net/interview/java/interface-vs-abstract-class>
4. <http://www.quizful.net/post/razlichie_v_primenenii_interfeysov_i_abstraktnih_klassov>
5. <https://ru.wikipedia.org/wiki/GRASP>
6. <http://kostin.ws/java/java-abstract-and-interfaces.html>