Принципы ООП

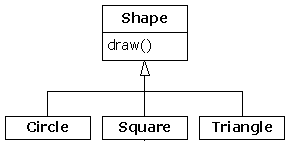
Java, как и любой другой язык с поддержкой ООП, реализует следующие принципы:

* **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".

Полиморфизм в Java

**Приведение к базовому типу**

[1] Объяснение этого термина кроется в том, что он основан она диаграмме наследования классов:



Преобразование (casting) дочернего к базовому происходит при движении вверх (up) по диаграмме наследования, так что получается - *upcasting* (приведение к базовому типу). Приведение к базовому типу всегда безопасно, поскольку Вы переходите от более общего типа, к более конкретному.

Так что дочерний класс является супермножеством базового класса. Он может содержать больше методов, чем базовый класс, но он должен содержать минимум все те методы, что есть в базовом классе.

Только одна вещь может случится при приведении к базовому типу, это, что могут потеряться некоторые методы. Поэтому то компилятор и позволяет осуществлять приведение к базовому типу без каких-либо ограничений на приведение типов или специальных замечаний.

Таким образом мы можем сослаться на объект класса-наследника посредством его базового класса.

Например:

List list = **new** ArrayList<String>();

**Раннее и позднее связывание**

Соединение вызова метода с телом метода называется связыванием.

Когда связывание осуществляется до запуска программы (компилятором и компоновщиком, если такой используется), то оно называется *ранним связыванием*. Компиляторы, не поддерживающие ООП (например, С) реализуют только раннее связывание.

Связывание называется *поздним связывание**м*, если оно происходит во время работы программы и основывается на типе объекта. Позднее связывание так же иногда называют *динамическим связыванием* или *связыванием во время выполнения*.

Если язык поддерживает позднее связывание, то у него должен быть механизм определения типа объекта во время работы программы и вызова соответствующего метода. Механизм позднего связывания меняется от языка к языку, но можно представлять себе, что в объект должна быть встроена некоторая информация о его типе.

**Реализация полиморфизма**

Рассмотрим иерархию классов, изображенную на рисунке выше:

**class** Shape {  
 **public void** draw() {  
 *//Nothing to do* }  
}  
  
**class** Square **extends** Shape {  
 **public void** draw() {  
 System.***out***.println(**"Square"**);  
 }  
}  
  
**class** Сircle **extends** Shape {  
 **public void** draw() {  
 System.***out***.println(**"Сircle"**);  
 }  
}  
  
**class** Triangle **extends** Shape {  
 **public void** draw() {  
 System.***out***.println(**"Triangle"**);  
 }  
}

Как можно заметить в наследниках у переопределен метод draw(). Создадим объекты классов Triangle, Circle, Square при этом ссылки на эти объекты будут иметь тип базового класса Shape:

Shape[] a = **new** Shape[]{**new** Triangle(), **new** Square(), **new** Сircle()};

**for**(**int** i = 0; i < a.**length**; i++) {  
 a[i].draw();  
}

Возникает вопрос какой из методов будет вызван? Ведь все ссылки имеют тип Shape. Вывод будет таким:

Triangle

Square

Сircle

Таким образом каждый класс наследник вызвал именно свой метод draw(), вместо того, чтобы вызвать пустой метод draw из класса Shape.

Это и есть реализация **полиморфизма**. Какой метод вызывать определяется по типу объекта с помощью механизма *позднего связывания.* [1]

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

Если нужно запретить возможность дальнейшего переопределения метода в подклассах используется ключевое слово final.

Понятие overloading (перегрузка) нужно отличать от понятия overriding (переопределение).

При переопределении (overriding) методов речь идет только о паре классов - базовом и порожденном. В порожденном классе определяется метод полностью идентичный по сигнатуре (имя, тип возврата, набор параметров) тому, что есть в базовом.

При перегрузке (overloading) одинаковым остается только имя, остальные части сигнатуры могут отличаться.

Generic в Java

**Обобщённое программирование** - это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания.

В Java, начиная с версии JDK 5.0, добавлены средства обобщённого программирования, синтаксически основанные на C++.

Ниже будут рассматриваться **generics** (дженерики) или <<контейнеры типа T>> - подмножество обобщённого программирования.

Можно сказать, что generic это класс или интерфейс, который параметризирован типами. Типы-параметры дженериков будут указаны позже, после определения самого дженерика. Конкретизация типа происходит при создании объекта.

Отдельные методы и конструкторы тоже могут быть параметризтрованными.

**Зачем нужны дженерики**

Дженерики используются если один и тот же алгоритм, блок кода, должен работать с разными типами данных. Дженерики реализуют подход: "один алгоритм - множество типов данных".

Самым ярким примером использования дженериков являются коллекции.

**Синтаксис**

Дженерики определяются следующим образом:

class name<T1, T2, ..., Tn> { /\* ... \*/ }

Секция параметров разделена угловыми скобками (<>), далее идет имя класса. Параметры-типы также называются переменными типов T1, T2, ..., and Tn.

Задекларированные переменные типов могут использоваться везде внутри класса.

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

{

**Соглашения об именах параметров типов**

Имя тип-параметра должно быть представлено одной буквой в верхнем регистре. Без такого соглашения было бы трудно обеспечить разницу между типом переменной, обычным классом или именем интерфейса.

Наиболее часто используемые имена параметров:

E - Element (used extensively by the Java Collections Framework)

K - Key

N - Number

T - Type

V - Value

**Экземпляры и вызов**

Ссылка должна иметь вид: Box<Integer> integerBox; Для создания экземпляра используется конструкция Box<Integer> integerBox = new Box<Integer>();

**Diamond syntax**

Чтобы упростить жизнь программистам в Java 7 был введён алмазный синтаксис (diamond syntax), в котором можно опустить параметры типа. Т.е. можно предоставить компилятору определение типов при создании объекта: Pair<Integer, String> pair = new Pair<>(6, " Apr");

1. Thinking in Java 4th Edition, Bruce Eckel
2. <http://www.quizful.net/post/java-generics-tutorial>
3. <https://docs.oracle.com/javase/tutorial/java/generics/types.html>