***КРИТЕРИЙ ПРИМЕНИМОСТИ НАСЛЕДОВАНИЯ***

Если имеются классы A1 и A2 и можно считать, что A2 является модифицированным (усложненным или измененным) вариантом A1 с сохранением всех особенностей поведения A1, то A2 должен описываться как потомок A1. На уровне абстракции, описывающей поведение, объект типа A2 должен вести себя, как объект типа A1 *при любых значениях полей данных*.

пять основных **критериев правильности построения иерархии**:

**1. В процессе наследования должно идти расширение (усложнение, специализация, конкретизация) классов, а не наоборот.**

**2. Наследование должно идти только от абстрактных классов (или интерфейсов как варианта полностью абстрактных классов — *см. главу 9*).**

**3. Названия всех методов должны давать точное представление о том, что делает метод. Причем эти имена должны восприниматься как команды (установить что-то, прочитать что-то, показать что-то и т. д.). Заметим, что названия методов при построении иерархии очень важны, так как именно они отражают абстракцию поведения объектов.**

**4. Недопустимы названия методов, содержащие связку "и" ("and"). Например, readAndShowSpeed, calculateIntegralAndWriteToFile и т. п. Такого рода гибриды следует разделять на два и более независимых метода — readSpeed и ShowSpeed; calculateIntegral и writeToFile, и т. д.**

**5. Названия всех классов должны давать четкое представление о соответствующих абстракциях поведения, в том числе для неабстрактных классов.**

Последнее требование кажется слишком расплывчатым, поэтому можно рекомендовать следующий простой способ его проверки — завести объектную переменную с соответствующим классу именем. В этом случае вызов метода figure.show() будет означать "показать фигуру", и эта расшифровка смысла вызова должна быть правильной по логике независимо от того, на объект какого типа ссылается переменная figure. Точно так же вызов dot.show() будет означать "показать точку", вызов scalableFigure.show()— "показать масштабируемую фигуру" и т. д.

**Но если у вас имеется иерархия, не удовлетворяющая этим критериям, сразу встает вопрос о необходимости ее доработки или отказа от нее.**

Встречается следующий критерий: **"если имеются классы A1 и A2 и можно сказать, что A2 является частным случаем A1, то A2 должен описываться как потомок A1"**. Это не совсем корректно.

***Наследовани****е* опирается на инкапсуляцию. Оно позволяет строить на основе первоначального класса другие, добавляя в классы новые поля данных и методы. Первоначальный класс называется ***прародителем* (ancestor),** новые классы — его ***потомками* (descendants).**

Набор классов, связанных отношением наследования, называется ***иерархией классов*.** А класс, стоящий во главе иерархии, от которого унаследованы все остальные (прямо или опосредованно), называется ***базовым классом иерархии*.**

***Полиморфизм***опирается как на инкапсуляцию, так и на наследование.

В иерархии принято рисовать стрелки в направлении от наследника к прародителю. Такое направление называется **Generalization** (обобщение, генерализация). Оно противоположно направлению наследования, которое принято называть **Specialization** (специализация). Стрелки символизируют направление в сторону упрощения.

На деле же ***потомки должны обладать более сложным устройством и поведением по сравнению с прародителем*.**

В концепции наследования основное внимание уделяется *поведению объектов*. Объекты с разным поведением имеют другой тип, а значения полей данных характеризуют *состояние объекта*, но не его тип. Поведение задается методами, и это делается в классах. А состояние относится к конкретному объекту, причем в конкретный момент выполнения программы.

Каждый объект класса-потомка ***при любых значениях полей***нужно рассматривать как экземпляр класса-прародителя. Он должен обладать той же абстракцией поведения, что и прародитель, но только с некоторыми изменениями на уровне реализации этого поведения. Мы говорим про абстракцию поведения как о тех характерных действиях, которые могут быть описаны на уровне полиморфного кода, безотносительно к конкретной реализации в конкретном классе.

Для класса AbstractFigure следует вводить переменную с именем figure, для класса AbstractName — переменную с именем name.

**Базовый класс *Object***

Класс Object базовый для всех классов Java, все его поля и методы наследуются и содержатся во всех классах. В классе Object содержатся следующие методы:

**public Boolean equals(Object obj**) — возвращает true в случае, когда равны значения объекта, из которого вызывается метод, и значения объекта, передаваемого через ссылку obj в списке параметров. Если объекты не равны, то возвращается false. В классе Object равенство рассматривается как равенство ссылок и эквивалентно оператору сравнения ==. Но в потомках этот метод может быть переопределен, и может сравнивать объекты по их содержимому (например, так происходит для объектов оболочечных числовых классов).

При переопределении метода equals обязательно переопределение метода hashCode

**public int hashCode()** — выдает *хэш-код* объекта. Как уже говорилось, хэш-кодом называется условно-уникальный числовой идентификатор, сопоставляемый какому-либо элементу. Из соображений безопасности выдавать адрес объекта прикладной программе нельзя. Поэтому в Java хэш-код заменяет адрес объекта в тех случаях, когда для каких-либо целей необходимо хранить таблицы адресов объектов;

**protected Object clone() throws CloneNotSupportedException** — метод занимается копированием объекта и возвращает ссылку на созданный *клон* (дубликат) объекта. В наследниках класса Object его обязательно нужно переопределить, а также указать, что класс реализует интерфейс Clonable. Попытка вызова метода из объекта, не поддерживающего *клонирования*, вызывает возбуждение исключительной ситуации CloneNotSupportedException (клонирование не поддерживается).

Различают два вида клонирования: *мелкое* (shallow), когда в клон один к одному копируются значения полей оригинального объекта, и *глубокое* (deep), при котором для полей ссылочного типа создаются новые объекты, клонирующие объекты, на которые ссылаются поля оригинала. При мелком клонировании и оригинал, и клон будут ссылаться на одни и те же объекты. Если объект имеет поля только примитивных типов, то различия между мелким и глубоким клонированием нет. Реализацией клонирования занимается программист, разрабатывающий класс, автоматического механизма клонирования нет. И именно на этапе разработки класса следует решить, какой вариант клонирования выбрать. В подавляющем большинстве случаев требуется глубокое клонирование.

**public final Class getClass()** — возвращает ссылку на метаобъект типа класс. С его помощью можно получать информацию о классе, к которому принадлежит объект, и вызывать его методы класса и поля класса;

**protected void finalize() throws Throwable** — вызывается перед уничтожением объекта. Должен быть переопределен в тех потомках Object, в которых требуется совершать какие-либо вспомогательные действия перед уничтожением объекта (закрыть файл, вывести сообщение, отрисовать что-либо на экране и т. п.). Вообще говоря, данный метод не следует использовать.

**public String toString()** — возвращает строковое представление объекта (настолько адекватно, насколько это возможно). В классе Object этот метод реализует выдачу в

строку полного имени объекта (с именем пакета), после которого следует символ @, а затем в шестнадцатеричном виде хэш-код объекта. В большинстве стандартных классов этот метод переопределен. Для числовых классов возвращается строковое представление числа, для строковых — содержимое строки, для символьного — сам символ (а не строковое представление его кода!).

**Листинг 6.21. Примеры вызова метода toString() для объектов разных типов**

Object obj=new Object();

System.out.println(" obj.toString() дает "+obj.toString());

Double d=new Double(1.0);

System.out.println(" d.toString()дает "+d.toString());

Character c='A';

System.out.println("c.toString() дает "+c.toString());

Например, фрагмент кода из листинга 6.21 обеспечит такой вывод:

obj.toString() дает java.lang.Object@fa9cf

d.toString()дает 1.0 c.toString()дает A

**notify(), notifyAll()и** несколько перегруженных вариантов метода **wait,** предназначенных для работы с потоками (threads).

**Порядок вызовов при создании объекта некого класса** (будем называть его дочерним классом):

1. Создается объект, в котором все поля данных имеют значения по умолчанию  
 (нули на двоичном уровне представления).

2. Вызывается конструктор дочернего класса.

3. Конструктор дочернего класса вызывает конструктор родителя (непосредственного прародителя),  
 а также по цепочке все прародительские конструкторы и инициализацию полей, заданных в этих  
 классах, вплоть до класса Object.

4. Поля родительской части объекта инициализируются значениями, заданными в декларации  
 родительского класса.

5. Выполняется тело конструктора родительского класса.

6. Поля дочерней части объекта инициализируются значениями, заданными в декларации дочернего  
 класса.

7. Выполняется тело конструктора дочернего класса.

Модификаторы class ИмяКласса extends ИмяРодителя {

Задание полей;

static {

тело блока инициализации класса

}

{

тело блока инициализации объекта

}

Задание подпрограмм — методов класса, методов объекта конструкторов

}

Блоков инициализации класса и блоков инициализации объекта может быть несколько.

**Порядок выполнения операторов при наличии блоков инициализации главного класса приложения   
(содержащего метод main):**

1. Инициализация полей данных и выполнение блоков инициализации класса  
 (в порядке записи в декларации класса).

2. Метод main.

3. Выполнение блоков инициализации объекта.

4. Выполнение тела конструктора класса.

**Для других классов порядок аналогичен**, но без вызова метода main:

1. Инициализация полей данных и выполнение блоков инициализации класса  
 (в порядке записи в декларации класса).

2. Выполнение блоков инициализации объекта.

3. Выполнение тела конструктора класса.

Чем лучше пользоваться — блоками инициализации или конструкторами? Ответ, конечно, неоднозначен: в одних ситуациях — конструкторами, в других — блоками инициализации.

Для придания начальных значений переменным класса в случаях, когда для этого требуются сложные алгоритмы, можно пользоваться только статическими блоками инициализации.  
Для инициализации полей объектов в общем случае предпочтительнее конструкторы, но если необходимо выполнить какой-либо код инициализации до вызова унаследованного конструктора, можно воспользоваться блоком динами-ческой инициализации.

При вызове конструктора с помощью слова this требуется, чтобы вызов this был пер-вым оператором в реализации вызывающего конструктора.