Глава 5 : Наследование

1) При наследовании методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым условиям в него добавляют дополнительные поля и методы

2) Способность переменной (например, е) ссылаться на объекты, имеющие разные фактические типы, называется полиморфизмом. Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием. Оба эти понятия будут подробнее обсуждаться далее в главе.

3) Совокупность всех классов, производных от общего суперкласса, называется иерархией наследования.

4) Основная выгода полиморфизма — легкость, с которой можно создавать новые классы, «ведущие себя» аналогично родственным, что, в свою очередь, позволяет достигнуть расширяемости и модифицируемости.(Если говорить кратко, **полиморфизм** — это способность обьекта использовать методы производного класса, который не существует на момент создания базового)

5)Когда у нас есть несколько методов с одинаковыми названиями и одинаковыми аргументами в одном классе , то выводится ошибка. Если такая же проблема возникает, но один метод в суперклассе, а другой в подклассе, то стоит смотреть на то, что стоит перед методом (static, final,privat), если стоят приватные, статические и final-методы, то выполняется метод из суперкласса(статическое связывание или как его еще называют раннее связывание), в противном случает выполняется метод из подкласса(динамическое связывание или позднее связывание)

6) Иногда наследование оказывается нежелательным. Классы, которые нельзя расширить, называются конечными. Для указания на это в определении класса используется модификатор доступа final.

7) На заре развития Java некоторые программисты пытались использовать ключевое слово final для того, чтобы исключить издержки, связанные с динамическим связыванием. Если метод не переопределяется и невелик, компилятор применяет процедуру оптимизации, которая состоит в непосредственном встраивании кода

8) Таким образом, можно сформулировать следующие основные правила приведения типов при наследовании:

• Приведение типов можно выполнять только в иерархии наследования.

• Чтобы проверить корректность приведения суперкласса к подклассу, следует выполнить операцию instanceof.

9)Абстракция - это принцип ООП, согласно которому при проектировании классов и создании объектов необходимо выделять только главные свойства сущности, и отбрасывать второстепенные.( абстрактный класс — это просто чертеж для будущих «нормальных» классов) Абстрактные классы избавляют нас от «просто объектов». Они дают нам базовое состояние и поведение. Например, у всех машин должна быть модель, цвет и максимальная скорость, а еще они должны уметь газовать и тормозить. Вот и все. Это — общая абстрактная схема, дальше ты уже сам проектируешь нужные тебе классы.

10)Можно выделить несколько следующих особенностей абстрактных классов или методов:

• **Класс является абстрактным, если хотя бы один из его методов является абстрактным**. Хоть один из двух, хоть один из тысячи методов — без разницы.

• При этом, если метод помечен словом abstract, каждый класс-наследник должен его реализовать или быть объявленным как абстрактный. Иначе компилятор выбросит ошибку.

11) Итак, в Java предоставляются следующие четыре модификатора доступа, определяющие границы области действия компонентов программы:

• Модификатор доступа private — ограничивает область действия классом.

• Модификатор доступа public — не ограничивает область действия.

• Модификатор доступа protected — ограничивает область действия пакетом и всеми подклассами.

• Модификатор доступа отсутствует — область действия ограничивается пакетом (к сожалению) по умолчанию.

12) Класс Object является исходным предшественником всех остальных классов, поэтому каждый класс в Java расширяет класс Object. Но явно отражать этот факт, как в приведенной ниже строке кода, совсем не обязательно.

13) Спецификация Java требует, чтобы метод equals () обладал следующими характеристиками:

1. Рефлексивность. При вызове х.equals(х) по любой ненулевой ссылке х должно возвращаться логическое значение true.

2. Симметричность. При вызове х.equals(у) по любым ссылкам х и у должно возвращаться логическое значение tru e тогда и только тогда, когда при вызове у.equals (х) возвращается логическое значение true.

3. Транзитивность. Если при вызовах х.equals(у) и у.equals(z) по любым ссылкам х, у и z возвращается логическое значение true , то и при вызове х.equals (z) возвращается логическое значение true.

4. Согласованность. Если объекты, на которые делаются ссылки х и у, не изменяются, то при повторном вызове х.equals(у) должно возвращаться то же самое значение. При вызове х.equals(null) по любой непустой ссылке х должно возвращаться логическое значение false .

14) Таким образом, возникают два разных варианта:

• Если проверка на равенство реализована в подклассе, правило симметричности требует использовать метод getClass ().

• Если же проверка на равенство производится средствами суперкласса, можно выполнить операцию instanceof. В этом случае возможна ситуация, когда два объекта разных классов будут признаны равными.

5. Приведите тип объекта otherObject к типу переменной требуемого класса:

ИмяКласса other = {ИмяКласса) otherObject;

6. Сравните все поля, как показано ниже. Для полей примитивных типов служит операция = , а для объектных полей — метод Objects.equals(). Если все поля двух объектов совпадают, возвращается логическое значение tru e, а иначе — логическое значение false,

return поле1 == other.поле!

&& поле2.equals(other.поле2)

&& …;

15) Хеш-код — это целое число, генерируемое на основе конкретного объекта. Хеш-код можно рассматривать как некоторый шифр: если х и у — разные объекты, то с большой степенью вероятности должны различаться результаты вызовов х.hashCode() и у.hashCode().

16) Еще одним важным в классе Object является метод toString (), возвращающий значение объекта в виде символьной строки.

17) Динамические массивы : ArrayList и LinkedList

Обратите внимание на следующие особенности данной версии программы:

• Не нужно задавать размер массива.

• С помощью метода add () можно добавлять сколько угодно элементов в массив.

• Вместо свойства length для подсчета количества элементов в массиве служит метод size ().

• Вместо выражения а [ i ] для доступа к элементу массива вызывается метод a.get( i ) .

18)Особенность динамического массива

К сожалению, с помощью параметра типа в угловых скобках нельзя задать примитивный тип, например, выражение ArrayList< int > недопустимо. И здесь приходит на помощь класс объектной оболочки Integer.

19)Автоупаковка и автораспаковка

Правда, в Java имеется удобное языковое средство, позволяющее добавлять и извлекать элементы из массива. Рассмотрим следующую строку кода:

list.add(3);

Она автоматически преобразуется в приведенную ниже строку кода. Подобное автоматическое преобразование называется автоупаковкой. list.add (new Integer(3) ) ;

20)Класс Enume:

Пример:

**public** **enum** DayOfWeek {

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

}

Внутри нашего Enum находятся 7 констант со статическим доступом. Мы уже можем его использовать для реализации логики в программе.

21)Рефлексия:

**Рефлексия**- это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов.

Рефлексия — очень мощный механизм, который можно применять для решения перечисленных ниже задач. А в последующих разделах поясняется, как пользоваться этим механизмом.

* Узнать/определить класс объекта;
* Получить информацию о модификаторах класса, полях, методах, константах, конструкторах и суперклассах;
* Выяснить, какие методы принадлежат реализуемому интерфейсу/интерфейсам;
* Создать экземпляр класса, причем имя класса неизвестно до момента выполнения программы;
* Получить и установить значение поля объекта по имени;
* Вызвать метод объекта по имени.

Объяснение рефлексии: <https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java>

22) Класс Class:

В Java почти все сущности являются объектами, за исключением примитивных типов. У каждого объекта есть класс. Сами классы тоже является объектами, и они принадлежат классу Class.

У класса Class нет публичных конструкторов. Class - это generic тип. Методы Class предназначены для получения информации о классе (объекте типа Class). Например, можно узнать полное имя класса, какие у него аннотации, какие конструкторы и т.п. Эти методы нужны для reflection. С помощью reflection вы можете создавать объекты, которые принадлежат этому классу, и при этом заранее класс объекта вы можете не знать.

**Рекомендации по применению ООП**

1.Размещайте общие операции и поля в суперклассе.

Поле паше было перемещено в класс Person именно для, чтобы не повторять его в классах Employee и Student.

2. Старайтесь не пользоваться защищенными полями.

Некоторые разработчики полагают, что следует "на всякий случай" объявлять большинство полей защищенными, чтобы подклассы могли обращаться к ним по мере надобности. Но имеются две веские причины, по которым такой механизм не гарантирует достаточной защиты. Во-первых, множество подклассов неограниченно. Всякий может создать подкласс, производный от данного класса, а затем написать программу, получающую непосредственный доступ к защищенным полям его экземпляра, нарушая инкапсуляцию. И во-вторых, в Java к защищенным полям имеют доступ все классы, находящиеся в том же самом пакете, независимо от того, являются ли они подклассами данного класса или нет. В то же время полезно объявлять защищенными методы, которые не предназначены для общего употребления и должны быть переопределены в подклассах.

3. Используйте наследование для моделирования отношений "является".

Наследование позволяет экономить время и труд при разработке программ, но иногда им злоупотребляют. Допустим, требуется создать класс Contractor. У работника, нанимаемого по контракту, имеется свое имя и дата заключения договора, но у него нет оклада. У него почасовая оплата, причем он работает не так давно, чтобы повышать оплату его труда. Ниже показано, как можно сделать класс Contractor подклассом, производным от класса Employee, добавив поле hourlyWage.

class Contractor extends Employee

{

private double hourlyWage;

}

Но это не совсем удачная идея. Ведь в этом случае получается, что каждый работник, нанятый по контракту, имеет и оклал и почасовую оплату. Если вы попробуете реализовать методы для распечатки платежных и налоговых ведомостей, то сразу же проявится недостаток такого подхода. Программа, которую вам придется написать, будет гораздо длиннее той, которую вы могли бы создать, не прибегая к неоправданному наследованию.

Отношение "контрактный работник-постоянный работник" не удовлетворяет критерию "является". Работники, нанятые по контракту, не являются постоянными и относятся к особой категории работников.

4. Не пользуйтесь наследованием, если не все методы имеет смысл сделать наследуемыми.

Допустим, требуется создать класс Holiday. Разумеется, праздники — это разновидность календарных дней, а дни можно представить в виде объектов типа GregorianCalendar, поэтому наследование можно применить следующим образом:

class Holiday extends GregorianCalendar(...)

К сожалению, множество праздников оказывается незамкнутым при наследовании. Среди открытых методов из класса GregorianCalendar имеется метод add (), который может превратить праздничные дни в будничные следующим образом:

Holiday Christmas;

Christmas.add(Calendar.DAY\_OF\_MONTH, 12);

Следовательно, наследование в данном случае не подходит. Следует, однако, иметь в виду, что подобные затруднения не возникают, если используется класс Local Date. Этот класс является неизменяемым, и поэтому в нем отсутствует метод, способный превратить праздничный день в будничный.

5. Переопределяя метод, не изменяйте его предполагаемое поведение. Принцип подстановки распространяется не только на синтаксис, но и на поведение, что важнее. При переопределении метода не следует без особых на то причин изменять его поведение. В этом компилятор вам не поможет. Ведь он не в состоянии проверить, оправдано ли переопределение метода. Допустим, требуется устранить упомянутый выше недостаток метода add () из класса Holiday, переопределив этот метод таким образом, чтобы он, например, не выполнял никаких действий или же возвращал следующий праздничный день. Но такое переопределение нарушает принцип подстановки. При выполнении приведенной ниже последовательности операторов пользователь вправе ожидать вполне определенного поведения и соответствующего результата, независимо от того, является ли объект х экземпляром класса GregorianCalendar или Holiday.

int dl = х.get(Calendar.DAY\_OF\_MONTH);

x.add(Calendar.DAY\_OF\_MONTH, 1);

int d2 = x.get(Calendar.DAY\_OF\_MONTH);

System.out.println(d2 - dl);

Безусловно, такой поход содержит камень преткновения. Разные пользователи посчитают естественным различное поведение программы. Так, по мнению некоторых, принцип подстановки требует, чтобы в методе Manager. equals () не учитывалась премия в поле bonus, поскольку она игнорируется в методе Employee. equals (). Подобные споры могут длиться бесконечно и не дать никакого результата. Поэтому, принимая конкретное решение, следует руководствоваться теми целями, для которых создается программа.

6. Пользуйтесь принципом полиморфизма, а не данными о типе.

Вспомните о принципе полиморфизма, как только увидите код, имеющий следующий вид:

if (х типа 1)

действие1(х);

else if (х типа 2)

действие2(х);

Имеют ли дейст вие\_1 и дейст вие\_2 общий характер? Если имеют, то поместите соответствующие методы в общий суперкласс или интерфейс обоих типов. Тогда можно просто сделать приведенный ниже вызов и выполнить правильное действие с помощью механизма динамического связывания, присущего полиморфизму.

х .действие();

Код, в котором применяется принцип полиморфизма или реализован интерфейс, намного легче сопровождать и расширять, чем код, изобилующий проверками типов.

7. Не злоупотребляйте механизмом рефлексии.

Механизм рефлексии позволяет создавать программы с высоким уровнем абстракции, где поля и методы определяются во время выполнения. Такая возможность чрезвычайно полезна для системного программирования, но для прикладного — практически не нужна. Рефлексия — очень хрупкий механизм, поскольку компилятор не может помочь в обнаружении ошибок. Все ошибки проявляются во время выполнения программы и приводят к возникновению исключений.