INTERVIEW JAVA CORE JAVA DEVELOPER

Ответы на вопросы на собеседование Java core (часть 2).

Vasyl K 8:02:00 6 Комментарии

- **ВОЗМОЖНО ЛИ ПРИ ПЕРЕОПРЕДЕЛЕНИИ (OVERRIDE) МЕТОДА ИЗМЕНИТЬ:**

    1. **Модификатор доступа**
    2. **Возвращаемый тип**
    3. **Тип аргумента или количество**
    4. **Имя аргументов**
    5. **Изменять порядок, количество или вовсе убрать секцию throws?**

    1. Да, если расширять (package -> protected -> public)
    2. Да, если выполняется Downcasting(понижающее преобразование, преобразование вниз по иерархии) то есть возвращаемый тип в переопределенном методе класса наследника должен быть НЕ шире чем в классе родителе (Object -> Number -> Integer)
    3. Нет, в таком случае происходит Overload(перегрузка)
    4. Да
    5. Возможно изменять порядок. Возможно вовсе убрать секцию throws в методе, так как она уже определена. Так же возможно добавлять новые исключения, которые наследуются от объявленных или исключения времени выполнения.

Переопределение методов действует при наследовании классов, т.е. в классе наследнике объявлен метод с такой же сигнатурой что и в классе родителе. Значит этот метод переопределил метод своего суперкласса.
Несколько нюансов по этому поводу:

- Модификатор доступа в методе класса наследника должен быть НЕ уже чем в классе родителе, иначе будет ошибка компиляции.
- Описание исключения в переопределенном методе класса наследника должен быть НЕ шире чем в классе родителе, иначе ошибка компиляции.
- Метод обьявленный как "private" в классе родителе нельзя переопределить!

- **ЧТО ТАКОЕ AUTOBOXING?**

Autoboxing/Unboxing – автоматическое преобразование между скалярными типами Java и соответствующими типами-врапперами (например, между int – Integer). Наличие такой возможности сокращает код, поскольку исключает необходимость выполнения явных преобразований типов в очевидных случаях.

- **ЧТО ТАКОЕ GENERICS?**

"Java Generics" – это технический термин, обозначающий набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры.
Примером дженериков или обобщенных типов может служить библитека с коллекциями в Java. Например, класс LinkedList<E> – типичный обобщенный тип. Он содержит параметр E, который представляет тип элементов, которые будут храниться в коллекции. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, мы можем использовать LinkedList<String> или LinkedList<Integer>. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Класс типа LinkedList<E> – обобщенный тип, который содержит параметр E. Создание объектов, типа LinkedList<String> или LinkedList<Integer> называются параметризированными типами, а String и Integer – реальные типы аргументов.

- **КАКОВА ИСТИННАЯ ЦЕЛЬ ИСПОЛЬЗОВАНИЯ ОБОБЩЕННЫХ ТИПОВ В JAVA?**

Обобщенные типы в Java были изобретены, в первую очередь, для реализации обобщенных коллекций.

- **КАКИМ ОБРАЗОМ ПЕРЕДАЮТСЯ ПЕРЕМЕННЫЕ В МЕТОДЫ, ПО ЗНАЧЕНИЮ ИЛИ ПО ССЫЛКЕ?**

В java параметры в методы передаются по значению, тоесть создаются копии параметров и с ними ведется работа в методе. В случае с примитивными типами, то при передачи параметра сама переменная не будет меняться так как в метод просто копируется ее значение.

А вот при передачи объекта копируется ссылка на объект, тоесть если в методе мы поменяем состояние объекта, то и за методом состояние объекта тоже поменяется. Но если мы этой копии ссылки попытаемся присвоить новую ссылку на обьект, то старая ссылка у нас не изменится.

В случае передачи по значению параметр копируется. Изменение параметра не будет заметно на вызывающей стороне.

В Java объекты всегда передаются по ссылке, а примитивы – по значениюю.

- **КАКИЕ МЕТОДЫ ЕСТЬ У КЛАССА OBJECT?**

Object это базовый класс для всех остальных объектов в Java. Каждый класс наследуется от Object. Соответственно все классы наследуют методы класса Object.
Методы класса Object:

- public final native Class getClass()
- public native int hashCode()
- public boolean equals(Object obj)
- protected native Object clone() throws CloneNotSupportedException
- public String toString()
- public final native void notify()
- public final native void notifyAll()
- public final native void wait(long timeout) throws InterruptedException
- public final void wait(long timeout, int nanos) throws InterruptedException
- public final void wait() throws InterruptedException
- protected void finalize() throws Throwable

- **ПРАВИЛА ПЕРЕОПРЕДЕЛЕНИЯ МЕТОДА OBJECT.EQUALS().**

1. Используйте оператор == что бы проверить ссылку на объект, переданную в метод equals. Если ссылки совпадают – вернуть true. Это не обязательно, нужно скорее для оптимизации, но может сэкономить время в случае "тяжёлых" сравнений.
2. Используйте оператор instanceof для проверки типа аргумента. Если типы не совпадают, вернуть false.
3. Преобразуйте аргумент к корректному типу. Так как на предыдущем шаге мы выполнили проверку, преобразование корректно.
4. Пройтись по всем значимым полям объектов и сравнить их друг с другом. Если все поля равны – вернуть true. Для сравнения простых типов использовать ==. Для полей со ссылкой на объекты использовать equals. float преобразовывать в int с помощью Float.floatToIntBits и сравнить с помощью ==. double преобразовывать в long с помощью Double.doubleToLongBits и сравнить с помощью ==. Для коллекций вышеперечисленные правила применяются к каждому элементу коллекции. Нужно учитывать возможность null полей/

объектов. Очерёдность сравнения полей может существенно влиять на производительность.

5. Закончив реализацию equals задайте себе вопрос, является ли метод симметричным, транзитивным и непротиворечивым.

И ещё несколько дополнительных правил.

- Переопределив equals, всегда переопределять hashCode.
- Не использовать сложную семантику в equals (типа определения синонимов). equals должен сравнивать поля объектов, не более.

- **ЕСЛИ ВЫ ХОТИТЕ ПЕРЕОПРЕДЕЛИТЬ EQUALS(), КАКИЕ УСЛОВИЯ ДОЛЖНЫ УДОВЛЕТВОРЯТЬСЯ ДЛЯ ПЕРЕОПРЕДЕЛЕННОГО МЕТОДА?**

Метод equals() обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным и рефлексивным.

- Рефлексивность: для любого ненулевого x, x.equals(x) вернет true;
- Транзитивность: для любого ненулевого x, y и z, если x.equals(y) и y.eqals(z) вернет true, тогда и x.equals(z) вернет true;
- Симметричность: для любого ненулевого x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) вернет true.

Также для любого ненулевого x, x.equals(null) должно вернуть false.

- **КАКАЯ СВЯЗЬ МЕЖДУ HASHCODE И EQUALS?**

Объекты равны, когда a.equals(b)=true и a.hashCode==b.hashcode ->true Но необязательно, чтобы два различных объекта возвращали различные хэш коды(такая ситуация называется коллизией).

- **КАКИМ ОБРАЗОМ РЕАЛИЗОВАНЫ МЕТОДЫ HASHCODE И EQUALS В КЛАССЕ OBJECT?**

Реализация метода equals в классе Object сводится к проверке на равенство двух ссылок:

```
1 public boolean equals(Object obj) {
2   return (this == obj);
3 }
```

Реализация же метода hashCode класса Object сделана нативной, т.е. определенной не с помощью Java-кода:

```
1 public native int hashCode();
```

Он обычно возвращает адрес объекта в памяти.

- **ЧТО БУДЕТ, ЕСЛИ ПЕРЕОПРЕДЕЛИТЬ EQUALS НЕ ПЕРЕОПРЕДЕЛЯЯ HASHCODE? КАКИЕ МОГУТ ВОЗНИКНУТЬ ПРОБЛЕМЫ?**

Они будут неправильно хранится в контейнерах, использующих хэш коды, таких как HashMap, HashSet.
Например HashSet хранит элементы в случайном (на первый взгляд) порядке. Дело в том, что для быстрого поиска HashSet расчитывает для каждлого элемента hashCode и именно по этому ключу ищет и упорядочивает элементы внутри себя

- **ЕСТЬ ЛИ КАКИЕ-ЛИБО РЕКОМЕНДАЦИИ О ТОМ, КАКИЕ ПОЛЯ СЛЕДУЕТ ИСПОЛЬЗОВАТЬ ПРИ ПОДСЧЕТЕ HASHCODE?**

Есть. Необходимо использовать уникальные, лучше примитивные поля, такие как id, uuid, например. Причем, если эти поля задействованы при вычислении hashCode, то нужно их задействовать при выполнении equals.
Общий совет: выбирать поля, которые с большой долью вероятности будут различаться.

- **ДЛЯ ЧЕГО НУЖЕН МЕТОД HASHCODE()?**

Существуют коллекции(HashMap, HashSet), которые используют хэш код, как основу при работе с объектами. А если хэш для равных объектов будет разным, то в HashMap будут два равных значения, что является ошибкой. Поэтому необходимо соответствующим образом переопределить метод hashCode().
X Хеширование – преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями или функциями свёртки, а их результаты называют хешем или хеш-кодом.
Хе Хеш-таблице – это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.
Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение i = hash(key) играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].
Одним из методов построения хеш-функции есть метод деления с остатком (division method) состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m – число возможных хеш-значений.

- **ПРАВИЛА ПЕРЕОПРЕДЕЛЕНИЯ МЕТОДА OBJECT.HASHCODE().**

При реализации hashCode используется несколько простых правил. Прежде всего, при вычислении хеш-кода следует использовать те же поля, которые сравниваются в equals. Это, во-первых, даст равенство хеш-кодов для равных обектов, во-вторых, распределено полученное значение будет точно так же, как и исходные данные. Теоретически, можно сделать так, чтобы хеш-код всегда был равен 0, и это будет абсолютно легальная реализация. Другое дело, что ее ценность будет равна тому же самому нулю.
Далее. Несмотря на то, что хеш-коды равных объектов должны быть равны, обратное неверно! Два неравных объекта могут иметь равные хеш-коды. Решающее значение имеет не уникальность, а скорость вычисления, потому как это приходится делать очень часто. Потому, в некоторых случаях имеет смысл посчитать хеш-код заранее и просто выдавать его по запросу. Прежде всего это стоит делать тогда, когда вычисление трудоемко, а объект неизменен.

- **РАССКАЖИТЕ ПРО КЛОНИРОВАНИЕ ОБЪЕКТОВ. В ЧЕМ ОТЛИЧИЕ МЕЖДУ ПОВЕРХНОСТНЫМ И ГЛУБОКИМ КЛОНИРОВАНИЕМ?**

Чтобы объект можно было клонировать, он должен реализовать интерфейс Cloneable(маркер). Использование этого интерфейса влияет на поведение метода "clone" класс Object. Таким образом
myObj.clone() создаст нам клон нашего объекта, но этот клон будет поверхностный.
Что значит поверхностным? Это значит что клонируется только примитивные поля класса, ссылочные поля не клонируются!
Для того, чтоб произвести глубокое клонирование, необходимо в клонируемом классе переопределить метод clone() и в нем произвести клонирование изменяемых полей объекта.

- **ПРАВИЛА ПЕРЕОПРЕДЕЛЕНИЯ МЕТОДА OBJECT.CLONE().**

Метод clone() в Java используется для клонирования объектов. Т.к. Java работает с объектами с помощью ссылок, то простым присваиванием тут не обойдешься, ибо в таком случае копируется лишь адрес, и мы получим две ссылки на один и тот же объект, а это не то, что нам нужно. Механизм копирования обеспечивает метод clone() класса Object.
clone() действует как конструктор копирования. Обычно он вызывает метод clone() суперкласса и т.д. пока не дойдет до Object.
Метод clone() класса Object создает и возвращает копию объекта с такими же значениями полей. Object.clone() кидает исключение CloneNotSupportedException если вы пытаетесь клонировать объект не реализующий интерфейс Cloneable. Реализация по умолчанию метода Object.clone() выполняет неполное/поверхностное (shallow) копирование. Если вам нужно полное/глубокое (deep) копирование класса то в методе clone() этого класса, после получения клона суперкласса, необходимо скопировать нужные поля.

Синтаксис вызова clone() следующий:

```
Object copy = obj.clone();
```

или чаще:

```
MyClass copy = (MyClass) obj.clone();
```

Один из недостатков метода clone(), это тот факт, что возвращается тип Object, поэтому требуется нисходящее преобразование типа. Однако начиная с версии Java 1.5 при переопределении метода вы можете сузить возвращаемый тип.

Пару слов о clone() и final полях.

Метод clone() несовместим с final полями. Если вы попробуете клонировать final поле компилятор остановит вас. Единственное решение – отказаться от final.

Ну и пример использования clone():

```
class MyClass implements Cloneable{
    public Integer i = 10;
    public MyClass clone() throws CloneNotSupportedException{
        MyClass obj=(MyClass)super.clone();
        obj.i = i;
        return obj;
    }
    public String toString(){
        return i.toString();
    }
}

public class Temp {
    public static void main(String []args) throws  CloneNotSupportedException{
        MyClass a = new MyClass();
        a.i = 11;
        MyClass b = a.clone();
        MyClass c = a;
        System.out.println("a: " + a + " b: " + b + " c: " + c);
        a.i=12;
        System.out.println("a: " + a + " b: " + b + " c: " + c);
    }
}
```

Консоль:

```
a: 11 b: 11 c: 11
a: 12 b: 11 c: 12
```

Как видите, изменение объекта a повлекло за собой изменение объекта c, а вот с b всё в порядке.

- **ГДЕ И КАК ВЫ МОЖЕТЕ ИСПОЛЬЗОВАТЬ ЗАКРЫТЫЙ КОНСТРУКТОР?**

Например в качестве паттерна Синглетон. В том же классе создается статический метод. Где и создается экземпляр класса, конечно если он уже не создан, тогда он просто возвращается методом.

- **ЧТО ТАКОЕ КОНСТРУКТОР ПО УМОЛЧАНИЮ?**

В Java если нет явным образом определённых конструкторов в классе, то компилятор использует конструктор по умолчанию, определённый неявным способом, который аналогичен "чистому", конструктору по умолчанию. Конструктор по умолчанию – это

довольно простая конструкция, которая сводится к созданию для типа конструктора без параметров. Так, например, если при объявлении нестатического класса не объявить пользовательский конструктор (не важно, с параметрами или без них), то компилятор самостоятельно сгенерирует конструктор без параметров. Некоторые программисты явным образом задают конструктор по умолчанию по привычке, чтобы не забыть в дальнейшем, но это не обязательно
В Java если производный класс не вызывает явным образом конструктор базового класса (в Java используя super() в первой строчке), то конструктор по умолчанию вызывается неявно. Если базовый класс не имеет конструктора по умолчанию, то это считается ошибкой.

- **ОПИШИТЕ МЕТОД OBJECT.FINALIZE().**

Метод finalize(). Java обеспечивает механизм, который является аналогичным использованию деструкторов в C ++, который может использоваться для того, чтобы произвести процесс очистки перед возвращением управления операционной системе.
Применяя метод finalize(), можно определять специальные действия, которые будут выполняться тогда, когда объект будет использоваться сборщиком мусора. Данный метод вызывается при уничтожении объекта автоматическим сборщиком мусора (garbage collector). В классе Object он ничего не делает, однако в классе-наследнике позволяет описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом.
Его синтаксис:
protected void finalize( ) throws Throwable
Ссылки не являются собранным мусором; только объекты - собранный мусор.

- **ЧЕМ ОТЛИЧАЮТСЯ СЛОВА FINAL, FINALLY И FINALIZE?**

final - Нельзя наследоваться от файнал класса. Нельзя переопределить файнал метод. Нельзя изменить сначение файнал поля.
finally - используется при обработке ошибок, вызывается всегда, даже если произошла ошибка(кроме System.exit(0)). Удобно использовать для освобождения ресурсов.
finalize() - вызывается перед тем как сборщик мусора будет проводить освобождение памяти. Не рекомендуется использовать для освобождения системных ресурсов, так как не известно когда сборщик мусора будет производить свою очистку. Вообще данный метод мало кто использует. Единственно что можно использовать этот метод для закрытия ресурса что должен работать на протяжении всей работы программы и закрываться по ее окончанию. Еще можно использовать метод для защиты от так называемых «дураков», проверять, освобождены ли ресурсы, если нет, то закрыть их.