INTERVIEW JAVA CORE JAVA DEVELOPER

Ответы на вопросы на собеседование Java core (часть 3).

Vasyl K 8:23:00 8 Комментарии

- **ОПИШИТЕ ИЕРАРХИЮ ИСКЛЮЧЕНИЙ.**

Все классы-исключения расширяют класс Throwable – непосредственное расширение класса object.

У класса Throwable и у всех его расширений по традиции два конструктора:

- Throwable o – конструктор по умолчанию;
- Throwable (String message) – создаваемый объект будет содержать произвольное сообщение message.

Записанное в конструкторе сообщение можно получить затем методом getMessage (). Если объект создавался конструктором по умолчанию, то данный метод возвратит null.

Метод toString возвращает краткое описание события, именно он работал в предыдущих листингах.

Три метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

- printstackTrace() – выводит сообщения в стандартный вывод, как правило, это консоль;
- printStackTrace(PrintStream stream) – выводит сообщения в байтовый поток stream;
- printStackTrace(PrintWriter stream) – выводит сообщения в символьный поток stream.

У класса Throwable два непосредственных наследника – классы Error и Exception. Они не добавляют новых методов, а служат для разделения классов-исключений на два больших семейства – семейство классов-ошибок (error) и семейство собственно классов-исключений (exception).

Классы-ошибки, расширяющие класс Error, свидетельствуют о возникновении сложных ситуаций в виртуальной машине Java. Их обработка требует глубокого понимания всех тонкостей работы JVM. Ее не рекомендуется выполнять в обычной программе. Не советуют даже выбрасывать ошибки оператором throw. Не следует делать свои классы-исключения расширениями класса Error или какого-то его подкласса.

Имена классов-ошибок, по соглашению, заканчиваются словом Error.
Классы-исключения, расширяющие класс Exception, отмечают возникновение обычной нештатной ситуации, которую можно и даже нужно обработать. Такие исключения следует выбросить оператором throw. Классов-исключений очень много, более двухсот. Они разбросаны буквально по всем пакетам J2SDK. В большинстве случаев вы способны подобрать готовый класс-исключение для обработки исключительных ситуаций в своей программе. При желании можно создать и свой класс-исключение, расширив класс Exception или любой его подкласс.

Среди классов-исключений выделяется класс RuntimeException – прямое расширение класса Exception. В нем и его подклассах отмечаются исключения, возникшие при работе JVM, но не столь серьезные, как ошибки. Их можно обрабатывать и выбрасывать, расширять своими классами, но лучше доверить это JVM, поскольку чаще всего это просто ошибка в программе, которую надо исправить. Особенность исключений данного класса в том, что их не надо отмечать в заголовке метода пометкой throws.
Имена классов-исключений, по соглашению, заканчиваются словом Exception.

- **КАКИЕ ВИДЫ ИСКЛЮЧЕНИЙ В JAVA ВЫ ЗНАЕТЕ, ЧЕМ ОНИ ОТЛИЧАЮТСЯ?**

Все исключительные ситуации можно разделить на две категории: проверяемые(checked) и непроверяемые(unchecked).
Все исключения, порождаемые от Throwable, можно разбить на три группы. Они определяются тремя базовыми типами: наследниками Throwable – классами Errorи Exception, а также наследником Exception – RuntimeException.
Ошибки, порожденные от Exception (и не являющиеся наследниками RuntimeException), являются проверяемыми. Т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.
Исключения, порожденные от RuntimeException, являются непроверяемыми и компилятор не требует обязательной их обработки.
Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, IndexOutOfBoundsException- выход за границы массива, java.lang.ArithmeticException- деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков try-catch.
Исключения, порожденные от Error, также не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной

ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести StackOverflowError (переполнение стека), OutOfMemoryError (нехватка памяти).
Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово throws с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется.
Переопределенный (overridden) метод не может расширять список возможных исключений исходного метода.

- **ЧТО ТАКОЕ CHECKED И UNCHECKED EXCEPTION?**

Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными.
Unchecked исключения в Java – наследованные от RuntimeException, Checked – от Exception.
Пример unchecked исключения – NullPointerException, checked исключения – IOException.

- **КАК СОЗДАТЬ СВОЙ UNCHECKED EXCEPTION?**

Унаследоваться от RuntimeException.

- **КАКИЕ ЕСТЬ UNCHECKE EXEPTION?**

Исключение	Значение
ArithmeticException	Арифметическая ошибка: деление на нуль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректная модификация коллекции
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном экземпляре
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

- **ЧТО ТАКОЕ ERROR?**

Исключения, порожденные от Error, не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести StackOverflowError (переполнение стека), OutOfMemoryError (нехватка памяти).
Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово throws с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется.

- **ОПИШИТЕ РАБОТУ БЛОКА TRY-CATCH-FINALLY.**

Если срабатывает один из блоков catch, то остальные блоки в данной конструкции try-catch выполняться не будут.
Свойством транзакционности исключения не обладают - действия, произведенные в блоке try до возникновения исключения, не отменяются после его возникновения.

- **ВОЗМОЖНО ЛИ ИСПОЛЬЗОВАНИЕ БЛОКА TRY-FINALLY (БЕЗ CATCH)?**

try может быть в паре с finally, без catch. Работает это точно так же - после выхода из блока try выполняется блок finally. Это может быть полезно, например, в следующей ситуации. При выходе из метода вам надо произвести какое-либо действие. А return в этом методе стоит в нескольких местах. Писать одинаковый код перед каждым return нецелесообразно. Гораздо проще и эффективнее поместить основной код в try, а код, выполняемый при выходе - в finally.

- **ВСЕГДА ЛИ ИСПОЛНЯЕТСЯ БЛОК FINALLY?**

Не всегда например в следующих ситуациях:

1. Существуют потоки-демоны - потоки предоставляющие некие сервисы, работая в фоновом режиме во время выполнения программы, но при этом не являются ее неотъемлеммой частью.Таким образом когда все потоки не демоны завершаются, программа завершает свою работу. В потоках демонах блок finally не выполняеться, они прерываются внезапно.
2. System.exit(0)
3. если в блоке finally произошло исключение и нет обработчика, то оставшийся код в блоке finally может не выполнятся.

- **КАКИЕ ЕСТЬ ОССОБЕННОСТИ КЛАССА STRING? ЧТО ДЕЛАЕТ МЕТОД INTERN().**

1. Внутреннее состояние класса String нельзя изменить после его создания, т.е. этот класс неизменяемый (immutable) поэтому когда вы пишете String str = "One" + "Two"; создается три! объекта класса String.
2. От него нельзя унаследоваться, потому что класс String объявлен как final: public final class String
3. Метод hashCode класса String переписан и возвращает: s[0]*31^(n–1) + s[1]*31^(n–2) + … + s[n–1]
4. У класса String есть метод public String intern(), который возвращает строку в каноническом ее представлении из внутреннего пула строк, поддерживаемого JVM, он нужен чтобы вместо String.equals() использовать ==.

Понятно, что оператор сравнения ссылок выполняется гораздо быстрее, чем посимвольное сравнение строк.
Используют в основном, где приходится сравнивать много строк, например в каких нибудь XML парсерах.
А вообще по увеличению производительности ещё вопрос. Ибо метод intern() тогда должен выполняться быстрее чем equals(), каждый раз когда вы вызываете метод intern() просматривается пул строк на наличие такой строки и если такая уже есть в пуле, то возвращается ссылка на нее.Сравниваются они через equal().

- **МОЖНО ЛИ НАСЛЕДОВАТЬ СТРОКОВЫЙ ТИП, ПОЧЕМУ?**

Классы объявлены final, поэтому наследоваться не получится.

- **ПОЧЕМУ СТРОКА ЯВЛЯЕТСЯ ПОПУЛЯРНЫМ КЛЮЧОМ В HASHMAP В JAVA?**

Поскольку строки неизменны, их хэшкод кэшируется в момент создания, и не требует повторного пересчета. Это делает строки отличным кандидатом для ключа в Map и они обрабатываются быстрее, чем другие объекты-ключи HashMap. Вот почему строки преимущественно используются в качестве ключей HashMap.

- **ДАЙТЕ ОПРЕДЕЛЕНИЕ ПОНЯТИЮ КОНКАТЕНАЦИЯ СТРОК.**

Конкатенация – операция объединения строк. Результатом является объединения второй строки с окончанием первой. Операция конкатенации могут быть выполнены так:

```
StringBuffer stringBuffer = new StringBuffer();
StringBuilder stringBuilder = new StringBuilder();
String str = "ABC";

str += "DEF";
String str2 = "one".concat("two").concat("three");
stringBuffer.append("DDD").append("EEE");
stringBuilder.append("FFF").append("GGG");

System.out.println(str);// ABCDEF
System.out.println(str2);// onetwothree
System.out.println(stringBuffer.toString());// DDDEEE
System.out.println(stringBuilder.toString());// FFFGGG
```

- **КАК ПЕРЕВЕРНУТЬ СТРОКУ?**

Один из способов как это можно сделать:

```
String s = "ABCDEFG";
StringBuilder stringBuilder = new StringBuilder(s);
stringBuilder.reverse();
System.out.println(stringBuilder.toString()); //GFEDCBA
```

- **КАК СРАВНИТЬ ЗНАЧЕНИЕ ДВУХ СТРОК?**

Строка в Java – это отдельный объект, который может не совпадать с другим объектом, хотя на экране результат выводимой строки может выглядеть одинаково. Оператор == (а также !=) работает с ссылками объекта String. Если две переменные String указывают на один и тот же объект в памяти, сравнение вернет результат true. В противном случае результат будет false, несмотря на то что текст может содержать в точности такие же символы.

Для сравнения посимвольно на эквивалентность необходимо использовать метод equals().

```
String s1 = new String("ABC");
String s2 = new String("ABC");
String s3 = "ABC";
String s4 = "ABC";

System.out.println(s1 == s2); // false
System.out.println(s3 == s4); // true. Т.к. один набор литералов будет
                              // указывать на одну область памяти
System.out.println(s1.equals(s2));// true

s1 = s2;
System.out.println(s1 == s2); // true
if ("someString" == "someString") { // true
    System.out.println("true");
}
```

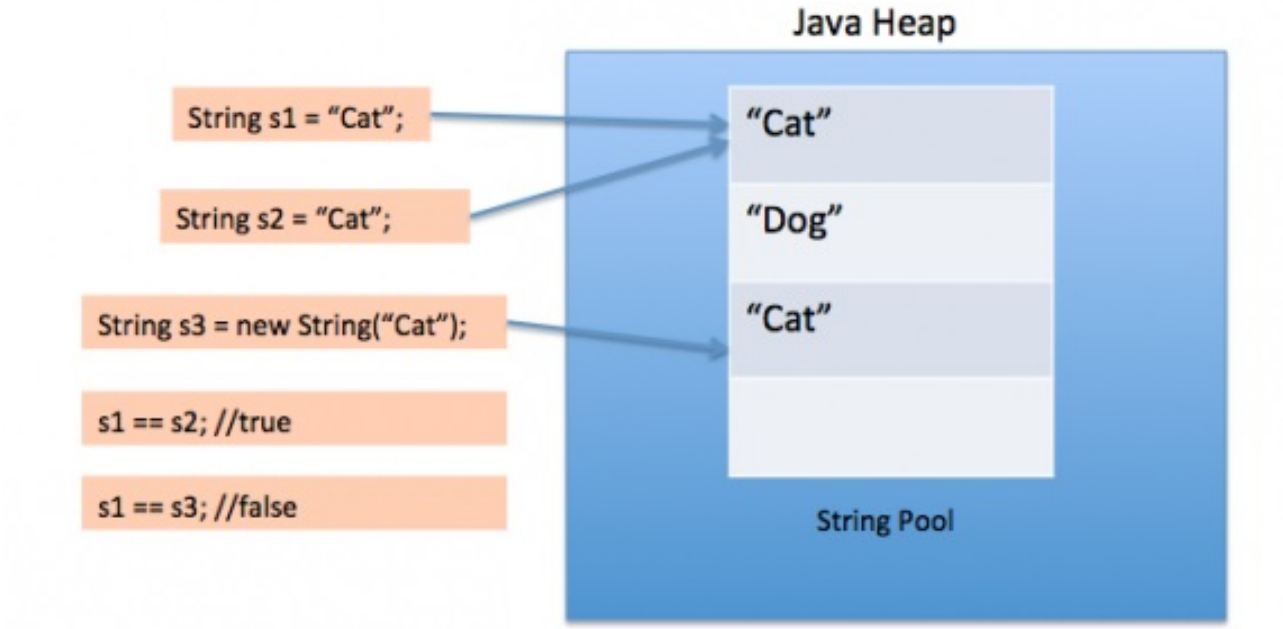
- **КАК ОБРЕЗАТЬ ПРОБЕЛЫ В НАЧАЛЕ И КОНЦЕ СТРОКИ?**

Небольшой пример:

```
String s = "    a    ";
System.out.println(s.trim() + "b");// ab
System.out.println(s + "b");//     a    b
```

- **ДАЙТЕ ОПРЕДЕЛЕНИЕ ПОНЯТИЮ "ПУЛ СТРОК".**

Пул строк – это набор строк, который хранится в памяти Java heap. Мы знаем, что String это специальный класс в Java, и мы можем создавать объекты этого класса, используя оператор new точно так же, как и создавать объекты, предоставляя значение строки в двойных кавычках.
Диаграмма ниже объясняет, как пул строк размещается в памяти Java heap и что происходит, когда мы используем различные способы создания строк.



Пул строк возможен исключительно благодаря неизменяемости строк в Java и реализации идеи интернирования строк.
Пул строк помогает экономить большой объем памяти, но с другой стороны создание строки занимает больше времени.
Когда мы используем двойные кавычки для создания строки, сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка.
Тем не менее, когда мы используем оператор new, мы принуждаем класс String создать новый объект строки, а затем мы можем использовать метод intern() для того, чтобы поместить строку в пул, или получить из пула ссылку на другой объект String с таким же значением.

- **МОЖНО ЛИ СИНХРОНИЗИРОВАТЬ ДОСТУП К СТРОКЕ?**

String сам по себе потокобезопасный класс. Если мы мы работаем с изменяемыми строками, то нужно использовать StringBuffer.

- **КАК ПРАВИЛЬНО СРАВНИТЬ ЗНАЧЕНИЯ СТРОК ДВУХ РАЗЛИЧНЫХ ОБЪЕКТОВ ТИПА STRING И STRINGBUFFER?**

Привести их к одному типу и сравнить.

- **ПОЧЕМУ СТРОКА НЕИЗМЕННАЯ И ФИНАЛИЗИРОВАННАЯ В JAVA?**

Есть несколько преимуществ в неизменности строк:

- Строковый пул возможен только потому, что строка неизменна в Java, таким образом виртуальная машина сохраняет много места в памяти(heap space), поскольку разные строковые переменные указывают на одну переменную в пуле. Если бы строка не была неизмененяемой, тогда бы интернирование строк не было бы возможным, потому что если какая-либо переменная изменит значение, это отразится также и на остальных переменных, ссылающихся на эту строку.
- Если строка будет изменяемой, тогда это станет серьезной угрозой безопасности приложения. Например, имя пользователя базы данных и пароль передаются строкой для получения соединения с базой данных и в программировании сокетов реквизиты хоста и порта передаются строкой. Так как строка неизменяемая, её значение не может быть изменено, в противном случае любой хакер может изменить значение ссылки и вызвать проблемы в безопасности приложения.
- Строки используются в Java classloader и неизменность обеспечивает правильность загрузки класса при помощи Classloader. К примеру, задумайтесь об экземпляре класса, когда вы пытаетесь загрузить java.sql.Connection класс, но значение ссылки изменено на myhacked.Connection класс, который может осуществить нежелательные вещи с вашей базой данных.
- Поскольку строка неизменная, её hashcode кэшируется в момент создания и нет необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в Map и его обработка будет быстрее, чем других ключей HashMap. Это причина, почему строка наиболее часто используемый объект, используемый в качестве ключа HashMap.

- **НАПИШИТЕ МЕТОД УДАЛЕНИЯ ДАННОГО СИМВОЛА ИЗ СТРОКИ.**

Мы можем использовать метод replaceAll для замены всех вхождений в строку другой строкой. Обратите внимание на то, что метод получает в качестве аргумента строку, поэтому мы используем класс Character для создания строки из символа, и используем её для замены всех символов на пустую строку.

```
1  public static String removeChar(String str, char ch) {
2      return str == null ? null : str.replaceAll(Character.toString(ch), "");
3  }
```