**https://javarush.ru/quests/lectures/questmultithreading.level01.lecture04**

**Методы equals & hashCode: зачем, где используются, как работают**

Java Multithreading

1 уровень , 4 лекция

**ОТКРЫТА**

— Теперь я расскажу о не менее полезных методах **equals(Object o) & hashCode()**.

Как ты уже, наверное, успел запомнить, в Java при сравнении ссылочных переменных сравниваются не сами объекты, а ссылки на объекты.

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| Integer i = **new** Integer(1);  Integer j = **new** Integer(1);  System.out.println(i==j); | i не равно j Переменные указывают на различные объекты. Хотя объекты содержат одинаковые данные; |
| Integer i = **new** Integer(1);  Integer j = i;  System.out.println(i==j); | i равно j Переменные содержат ссылку на один и тот же объект. |

— Да, я это помню.

— Есть также стандартное решение этой ситуации – метод **equals**.

Цель метода **equals** – определить идентичны ли объекты внутри, сравнив внутреннее содержание объектов.

— И как он это делает?

— Тут все аналогично методу toString().

У класса Object есть своя реализация метода equals, которая просто сравнивает ссылки:

**public** **boolean** equals(Object obj)

{

**return** (**this** == obj);

}

— М-да. С чем боролись, на то и напоролись.

— Не вешай нос. Тут все тоже очень хитро.

Этот метод создавался, чтобы разработчики переопределяли его в своих классах. Ведь только разработчик класса знает, какие данные важны, что учитывать при сравнении, а что – нет.

— А можно пример такого метода?

— Конечно. Допустим, у нас есть класс, описывающий математические дроби, тогда он выглядел бы так (для ясности, я переведу английские названия на русский язык):

**Пример**

**class** Дробь

{

**private** **int** числитель;

**private** **int** знаменатель;

Дробь(**int** числитель, **int** знаменатель)

{

**this**.числитель = числитель;

**this**.знаменатель = знаменатель;

}**public** **boolean** equals(Object obj)

{

**if** (obj==**null**)

**return** false;

**if** (obj.getClass() != **this**.getClass() )

**return** false;

Дробь other = (Дробь) obj;

**return** **this**.числитель\* other.знаменатель == **this**.знаменатель \* other.числитель;

}

}

|  |
| --- |
| **Пример вызова:** |
| Дробь **one** = **new** Дробь(2,3); Дробь **two** = **new** Дробь(4,6); **System.out.println(one.equals(two));** |
| **Результат вызова будет true.** **дробь 2/3 равна дроби 4/6** |

— Для большей ясности я использовала русские названия. Так можно делать только в обучающих целях.

Теперь разберем пример.

Мы переопределили метод **equals**, и теперь для объектов класса **Дробь** у него будет своя реализация.

В этом методе есть несколько проверок:

**1)** Если переданный для сравнения объект – **null**, то объекты не равны. Объект, у которого вызвали метод **equals** ведь точно не **null**.

**2)** Проверка на сравнение классов. Если объекты разных классов, то мы не будем пробовать их сравнить, а сразу скажем, что это различные объекты – **return false**.

**3)** Со второго класса школы все помнят, что дробь 2/3 равна дроби 4/6. А как это проверить?

|  |
| --- |
| **2/3 == 4/6** |
| **Умножим обе части на оба делителя (6 и 3), получим:** |
| **6 \* 2 == 4 \* 3** |
| **12 == 12** |
| **Общее правило:** |
| **Если** **a / b == c / d** **То** **a \* d == c \* b** |

— Поэтому в третьей части метода **equals** мы преобразуем переданный объект к типу Дробь и сравниваем дроби.

— Понятно. Если бы мы просто сравнивали числитель с числителем и знаменатель со знаменателем, то дробь 2/3 не была бы равной 4/6.

Теперь понятно, что ты имела ввиду, когда говорила, что только разработчик класса знает, как правильно его сравнивать.

— Да, но это только половина дела. **Есть еще второй метод – hashCode()**

— С методом equals все понятно, а зачем нужен **hashCode**()?

— Метод **hashCode** нужен для быстрого сравнения.

У метода **equals** есть большой минус – он слишком медленно работает. Допустим, у тебя есть множество(Set) из миллиона элементов, и нам нужно проверить, содержит ли оно определенный объект или нет. Как это сделать?

— Можно в цикле пройтись по всем элементам и сравнить нужный объект с каждым объектом множества. Пока не найдем нужный.

— А если его там нет? Мы выполним миллион сравнений, чтобы узнать, что там нет этого объекта? Не многовато ли?

— Да, даже мне понятно, что слишком много сравнений. А что, есть другой способ?

— Да, для этого и используется **hashCode**().

Метод **hashCode**() для каждого объекта возвращает определенное число. Какое именно – это тоже решает разработчик класса, как и в случае с методом equals.

Давай рассмотрим ситуацию на примере:

Представь, что у тебя есть миллион 10-тизначных чисел. Тогда в качестве hashCode для каждого числа можно выбрать остаток от его деления на 100.

Пример:

|  |  |
| --- | --- |
| **Число** | **Наш hashCode** |
| 1234567890 | 90 |
| 9876554321 | 21 |
| 9876554221 | 21 |
| 9886554121 | 21 |

— Да, с этим понятно. И что нам делать с этим hashCode-числом?

— Вместо того чтобы сравнивать числа, мы будем сравнивать их **hashCode**. Так быстрее.

И только если **hashCode**-ы равны, сравнивать уже посредством **equals**.

— Да, так быстрее. Но нам все равно придется сделать миллион сравнений, только уже более коротких чисел, а для тех чисел, чьи hashCode совпадают, опять вызвать equals.

— Нет, можно обойтись гораздо меньшим числом.

Представь, что наше множество хранит числа, сгруппированные по hashCode или отсортированные по **hashCode**(что равносильно их группировке, т.к. числа с одинаковым hashCode находятся рядом). Тогда можно очень быстро и легко отбросить ненужные группы, достаточно один раз для каждой группы проверить совпадает ли ее hashCode с hashCode заданного объекта.

Представь, что ты студент, и ищешь своего друга, которого знаешь в лицо и про которого известно, что он живет в 17 общаге. Тогда ты просто проходишься по всем общежитиям универа и в каждом общежитии спрашиваешь «это 17 общага?». Если нет, то ты отбрасываешь всех из этой общаги и переходишь к следующей. Если «да», то начинаешь ходить по всем комнатам и искать друга.

В данном примере номер общаги – 17 – это и есть hashCode.

Разработчик, который реализует функцию hashCode, должен знать следующие вещи:

**А)** **у двух разных объектов может быть одинаковый hashCode** (разные люди могут жить в одной общаге)

**Б)** **у одинаковых объектов** (**с точки зрения equals**) **должен быть одинаковый hashCode**.

**В)** **хеш-коды должны быть выбраны таким образом, чтобы не было большого количества различных объектов с одинаковыми hashCode.** Это сведет все их преимущество на нет (Ты пришел в 17 общагу, а там живет пол универа. Облом-с).

И теперь самое важное. Если ты переопределяешь метод **equals**, обязательно нужно переопределить метод **hashCode**(), с учетом трех вышеописанных правил.

Все дело в том, что **коллекции в Java перед тем как сравнить объекты с помощью equals всегда ищут/сравнивают их с помощью метода hashCode()**. И если у одинаковых объектов будут разные hashCode, то объекты будут считаться разными - до сравнения с помощью equals просто не дойдет.

В нашем примере с Дробью, если бы мы взяли hashCode равный числителю, то дроби 2/3 и 4/6 имели бы разные hashCode. Дроби – одинаковые, equals говорит, что они одинаковые, но hashCode говорит, что они разные. И если перед сравнением с помощью equals сравнивать по hashCode, то получим что объекты разные и до equals просто не дойдём.

Пример:

|  |
| --- |
|  |
| HashSet<Дробь>set = new HashSet<Дробь>(); set.add(new Дробь(2,3));System.out.println( set.contains(new Дробь(4,6)) ); |
| Если метод **hashCode()** будет возвращать числитель дроби, то результат будет **false**. Объект new **Дробь(4,6)**не будет найден в коллекции. |

— А как правильно реализовать hashCode для дроби?

— Тут надо помнить, что одинаковым дробям обязательно должен соответствовать одинаковый hashCode.

**Вариант 1**: hashCode равен целой части от деления.

Для дроби 7/5 и 6/5 это будет 1.

Для дроби 4/5 и 3/5 это будет 0.

Но этот вариант плохо годится для сравнения дробей, которые заведомо меньше 1. Целая часть (hashCode) всегда будет 0.

**Вариант 2**: hashCode равен целой части от деления знаменателя на числитель.

Этот вариант подойдет для случая, когда значение дроби меньше 1. Если дробь меньше 1, значит перевернутая дробь больше 1. А если мы переворачиваем все дроби – это никак не скажется на их сравнении.

Итоговый вариант будет совмещать в себе оба решения:

**public** **int** hashCode()

{

**return** числитель/знаменатель + знаменатель/числитель;

}

Проверяем для дробей 2/3 и 4/6. У них должны быть равные hashCode:

|  |  |  |
| --- | --- | --- |
|  | **Дробь 2/3** | **Дробь 4/6** |
| **числитель / знаменатель** | 2 / 3 == 0 | 4 / 6 == 0 |
| **знаменатель / числитель** | 3 / 2 == 1 | 6 / 4 == 1 |
| **числитель / знаменатель** **+** **знаменатель / числитель** | 0 + 1 == 1 | 0 + 1 == 1 |

На этом – все.

— Спасибо, Элли, было действительно интересно.