43. Каким образом можно обратиться к локальной переменной метода из анонимного класса, объявленного в теле этого метода? Есть ли какие-нибудь ограничения для такой переменной?  
44. Как связан любой пользовательский класс с классом Object?  
45. Расскажите про каждый из методов класса Object.  
46. Что такое метод equals(). Чем он отличается от операции ==.  
47. Если вы хотите переопределить equals(), какие условия должны удовлетворяться для переопределенного метода?  
48. Если equals() переопределен, есть ли какие-либо другие методы, которые следует переопределить?  
49. В чем особенность работы методов hashCode и equals? Каким образом реализованы методы hashCode и equals в классе Object?  Какие правила и соглашения существуют для реализации этих методов? Когда они применяются?  
50. Какой метод возвращает строковое представление объекта?  
51. Что будет, если переопределить equals не переопределяя hashCode? Какие могут возникнуть проблемы?  
52. Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode?  
53. Как вы думаете, будут ли какие-то проблемы, если у объекта, который используется в качестве ключа в hashMap изменится поле, которое участвует в определении hashCode?  
54. Чем отличается абстрактный класс от интерфейса, в каких случаях что вы будете использовать?  
55. Можно ли получить доступ к private переменным класса и если да, то каким образом?  
56. Что такое volatile и transient? Для чего и в каких случаях можно было бы использовать default?  
57. Расширение модификаторов при наследовании, переопределении и сокрытии методов. Если у класса-родителя есть метод, объявленный как private, может ли наследник расширить его видимость? А если protected? А сузить видимость?  
58. Имеет ли смысл объявлять метод private final?  
59. Какие особенности инициализации final переменных?  
60. Что будет, если единственный конструктор класса объявлен как final?  
61. Что такое finalize? Зачем он нужен? Что Вы можете рассказать о сборщике мусора и алгоритмах его работы.  
62. Почему метод clone объявлен как protected? Что необходимо для реализации клонирования?

### **Ответы. Часть 3**

#### 43. Каким образом можно обратиться к локальной переменной метода из анонимного класса, объявленного в теле этого метода? Есть ли какие-нибудь ограничения для такой переменной?

Также как и локальные классы, анонимные могут захватывать переменные, доступ к локальным переменным происходит по тем же правилам:

* Анонимный класс имеет доступ к полям внешнего класса.
* Анонимный класс не имеет доступ к локальным переменным области, в которой он определен, если они не финальные (final) или неизменяемые (effectivelyfinal).
* Как и у других внутренних классов, объявление переменной с именем, которое уже занято, затеняет предыдущее объявление.
* Вы не можете определять(добавлять) статические члены анонимного класса.

Анонимные классы также могут содержать в себе локальные классы. Конструктора в анонимном классе быть не может.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | publicclassAnimal{        IntegerclassAreaVar2=25;        publicvoidanonymousClassTest(){          finalInteger[]localAreaVar={25};          //Анонимныйкласс          ActionListener listener=newActionListener(){  *@Override*              publicvoidactionPerformed(ActionEvente){                  //можно использовать переменные класса без указания final                  classAreaVar2=classAreaVar2+25;                    //нельзя использовать локальные переменные, если они не final;                  /\*Local variable is accessed from within inner class: needs to be declared final \*/                  localAreaVar[0]=localAreaVar[0]+5;              }          };      }  } |

#### 44. Как связан любой пользовательский класс с классом Object?

Все классы являются наследниками суперкласса Object. Это не нужно указывать явно. В результате объект Object может ссылаться на объект любого другого класса.

#### 45. Расскажите про каждый из методов класса Object.

* publicfinalnativeClassgetClass() — возвращает в рантайме класс(метаданные: аннотации, методы, поля … ПОЛНУЮ ИНФОРМАЦИЮ О КЛАССЕ ЧЕРЕЗ МЕХАНИЗМ РЕФЛЕКСИИ) данного объекта.
* publicnativeinthashCode() — возвращаетхеш-код(числовое представление объекта)
* publicboolean equals(Object obj) — сравниваетобъекты.
* protected native Object clone() throws CloneNotSupportedException — клонированиеобъекта(полнаякопия. Новый объект в новой ячейке памяти). Если клонируемый объект не реализует Cloneable,то вызов метода clone() выбросит исключение CloneNotSupportedException.

**Native** означает, что метод написан НЕ НА Java !!!!

* publicStringtoString() — возвращает строковое представление объекта.
* publicfinalnativevoidnotify() — просыпается один поток, который ждет на “мониторе” данного объекта.
* publicfinalnativevoidnotifyAll() — просыпаются все потоки, которые ждут на “мониторе” данного объекта.
* publicfinalnativevoidwait(longtimeout) throwsInterruptedException — поток переходит в режим ожидания в течение указанного времени.
* publicfinalvoidwait() throwsInterruptedException — приводит данный поток в ожидание, пока другой поток не вызовет notify() или notifyAll() методы для этого объекта.
* publicfinalvoidwait(longtimeout, intnanos) throwsInterruptedException — приводит данный поток в ожидание, пока другой поток не вызовет notify() или notifyAll() для этого метода, или пока не истечет указанный промежуток времени.
* protectedvoidfinalize() throwsThrowable — вызывается сборщиком мусора, когда garbagecollector определил, что ссылок на объект больше нет.

#### 46. Что такое метод equals(). Чем он отличается от операции ==.

* == → является ссылочным сравнением, то есть оба объекта указывают на одно и то же место памяти
* .equals() → оценивает сравнение значений в объектах

#### \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* .equals(...) будет сравнивать только то, что написано для сравнения, не больше, не меньше.
* Если класс не переопределяет метод equals, то по умолчанию используется метод equals(Object o) ближайшего родительского класса, который переопределил этот метод.
* Если родительские классы не предоставили переопределение, то по умолчанию используется метод из конечного родительского класса Object и поэтому вы остаетесь с методом Object#equals(Object o). В Object API это то же самое, что и ==; то есть он возвращает true тогда и только тогда, когда обе переменные относятся к одному и тому же объекту, если их ссылки одно и то же. Таким образом, вы будете тестировать на **равенство объектов,** а не на **функциональное равенство**.
* Всегда помните, чтонужно переопределить hashCode если вы переопределяете equals чтобы не "разорвать контракт". Согласно API, результат, возвращаемый методом hashCode() для двух объектов, **должен** быть таким же, если их методы equals показывают, что они эквивалентны. Обратное утверждение не обязательно верно.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Сравнение объектов в Java выполняется с помощью оператора сравнения == и метода *equals()*.

Оператор сравнения == можно использовать с:

* примитивами, т.к. они используют ПУЛ значений(схоже с Set), который поддерживает уникальность содержащихся в нем значений(схоже с contains()).
* классами-оболочками, т.к. они лишь объектное представление примитивов и в итоге, всё равно ссылаются на примитивы..
* String. Также использует ПУЛ значений.
* ENUM.Он содержит определённое количество значений(экземпляров Енама), ссылки на которые ВСЕГДА ПОЛУЧАЮТ из одного источника(по имени Енама).

Метод *equals()* используется только экземплярами классов.

Позволяет проверить ссылочные типы данных на истинное равенство(не по ссылкам, а на реальное содержимое)

Рассмотрим пример:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Object ob1 = new Object();  Object ob2 = ob1;    System.out.println(ob1 == ob2); // true  System.out.println(ob1.equals(ob2)); //true |

Переменная *ob1* содержит в себе ссылку на объект класса [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html), а переменная *ob2* просто ее копирует при объявлении, в итоге две переменных ссылаются на один и тот же объект в памяти, что при сравнении этих двух переменных всегда возвращается *true*.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Object ob1 = new Object();  Object ob2 = new Object();    System.out.println(ob1 == ob2); // false  System.out.println(ob1.equals(ob2)); //false |

Теперь при объявлении каждой из переменных вызывается оператор *new*, который для каждой из них создает свой экземпляр класса [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html), поэтому при сравнении этих двух переменных каждый раз возвращается *false*, потому что обе переменные указывают на разные объекты в разных участках памяти. Давайте теперь сравним что-то более конкретное, например, две строки:

|  |  |
| --- | --- |
| 1  2  3  4  5 | String str1 = new String("ABC");  String str2 = new String("ABC");    System.out.println(str1 == str2); // false  System.out.println(str1.equals(str2)); //true |

Как видите поведение метода [equals()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "equals(java.lang.Object)) для класса [String](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html) несколько отличается. Это происходит по одной простой причине, метод [equals()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "equals(java.lang.Object)) определен в классе [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html), поэтому все классы в Java его наследуют и вольны переопределять, что и было сделано в классе [String](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html). Как видите методом [equals()](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html" \l "equals(java.lang.Object)) проверяются на соответствие символы двух строк, если они совпадают, то метод возвращает *true*. Перед тем как приступить к переопределению метода [equals()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "equals(java.lang.Object)) и комплексному сравнению объектов созданных нами классов, мы рассмотрим инструменты, которые нам для этого понадобятся: оператор [instanceof](https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html" \l "d5e24757) и метод [getClass()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "getClass()).

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Метод equals сравнивает символы из объекта типа String, а операция == - две ссылки на объекты, определяя ссылаются ли они на один и тот же экземпляр.

В примере отображено два разных объекта типа String, которые могут содержать одинаковые символы, но ссылки на эти объекты при сравнении не будут равнозначны:

// метод equals() в сравнении с операцией ==

publicclassEqualsNotEqualTo {

publicstaticvoidmain(String[] args) {

Strings1="Сравнение";

Strings2=newString(s1);

System.out.println(s1 + " равно " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));

}

}

Переменная s1 ссылается на экземпляр класса String, созданный присваиванием ей строкового литерала "Сравнение". А объект, на который ссылается переменная s2, создается с использованием переменной s1 в качестве инициализатора.

Таким образом, содержимое обоих объектов типа String одинаково, но это разные объекты. Следовательно, переменные s1 и s2 ссылаются не на один и тот же объект, поэтому они не равны, в сравнении операции ==.

## **Результат:**

Сравнение равно Сравнение ->true

Сравнение == Сравнение ->false

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

**finalclass A**

{

publicstaticString s;

A(){

this.s = newString("aTest" );

}

}

**finalclass B**

{

privateString s;

B(){

this.s = newString("aTest" );

}

publicStringgetS(){

return s;

}

}

**publicfinalclassMyEqualityTest**

{

publicstaticvoid main( Stringargs[] ){

String s1 = newString("Test" );

String s2 = newString("Test" );

System.out.println("\n1 - PRIMITIVES ");

System.out.println( s1 == s2 ); // false

System.out.println( s1.equals( s2 )); // true

A a1 = newA();

A a2 = newA();

System.out.println("\n2 - OBJECT TYPES / STATIC VARIABLE" );

System.out.println( a1 == a2 ); // false

System.out.println( a1.s == a2.s ); // true

System.out.println( a1.s.equals( a2.s ) ); // true

B b1 = newB();

B b2 = newB();

System.out.println("\n3 - OBJECT TYPES / NON-STATIC VARIABLE" );

System.out.println( b1 == b2 ); // false

System.out.println( b1.getS() == b2.getS() ); // false

System.out.println( b1.getS().equals( b2.getS() ) ); // true

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

classComparisonDemo {

public static void main(String[] args){

int value1 = 1;

int value2 = 2;

if(value1 == value2)

System.out.println("value1 == value2");

if(value1 != value2)

System.out.println("value1 != value2");

if(value1 > value2)

System.out.println("value1 > value2");

if(value1 < value2)

System.out.println("value1 < value2");

if(value1 <= value2)

System.out.println("value1 <= value2");

}}**Output: value1 != value2 value1 < value2 value1 <= value2**

**47. Если вы хотите переопределить equals(), какие условия должны удовлетворяться для переопределенного метода?**

Метод equals() может быть переопределён только для ссылочных типов(наследников класса Object). ??? ( + см.ответ 49!!! )

**48. Если equals() переопределен, есть ли какие-либо другие методы, которые следует переопределить?**

При переопределении equals() обязательно нужно переопределить метод hashCode(). Равные объекты должны возвращать одинаковые хэш-коды.

* *Всегда помните, чтобы переопределить hashCode если вы переопределяете equals() чтобы не "разорвать контракт". Согласно API, результат, возвращаемый методом hashCode() для двух объектов, должен быть таким же, если их методы equals() показывают, что они эквивалентны. Обратное утверждение не обязательно верно.*

Т.е. одинаковые объекты всегда обязаны возвращать одинаковый hashCode,

при этом разные объекты могут возвращать одинаковый hashCode, в случае переполнения типа int(коллизия).

**https://habr.com/ru/post/168195/**

**49. В чем особенность работы методов hashCode и equals? Каким образом реализованы методы hashCode и equals в классе Object? Какие правила и соглашения существуют для реализации этих методов? Когда они применяются?**

Если очень просто, то хеш-код — это число. Если более точно, то

***hashCode - это битовая строка фиксированной длинны, полученная из массива произвольной длинны.***  
Пример №1Выполнимследующийкод:

**publicclassMain**{

**publicstaticvoidmain**(String[] args){

Object object = **new**Object();

**int**hCode;

hCode = object.hashCode();

System.out.println(hCode);

}

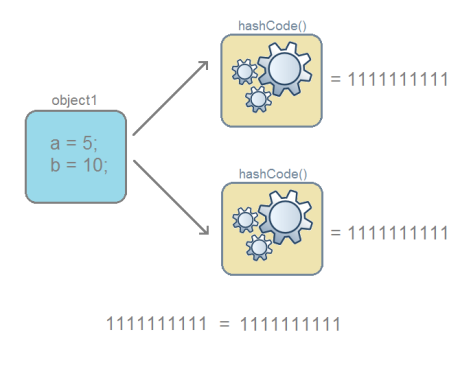
}  
В результате выполнения программы в консоль выведется целое 10-ти значное число. ***Это число и есть наша «…битовая строка фиксированной длины».*** В java она представлена в виде числа примитивного типа int, который равен 4-м байтам, и может помещать числа от -2 147 483 648 до 2 147 483 647. На данном этапе важно понимать, что хеш-код это число, у которого есть свой предел, который для java ограничен примитивным целочисленным типом int.  
Вторая часть объяснения гласит: «***… полученная из массива произвольной длины.»***

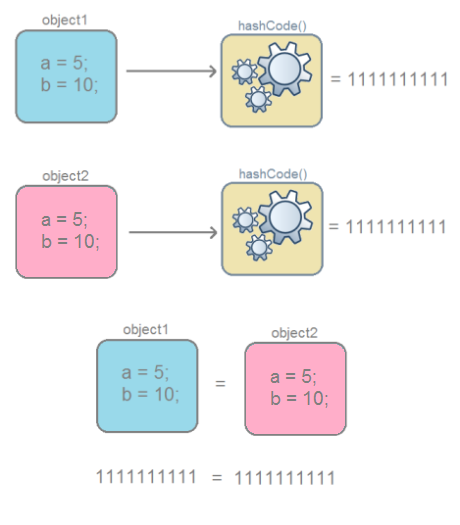
Под массивом произвольной длины мы будем понимать объект. В 1 примере в качестве массива произвольной длины у нас выступает объект типа Object.  
  
В итоге, в терминах Java, *hashCode* — это целочисленный результат работы метода, которому в качестве входного параметра передан объект.  
Этот метод реализован таким образом, что для одного и того-же входного объекта, *hashCode*всегда будет одинаковым. Следует понимать, что множество возможных *hashCode*ов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: **“Множество объектов мощнее множества *hashCode*ов”.** Из-за этого ограничения, вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.  
  
Здесь главное понять, что:

* Если *hashCode* разные, то и входные объекты гарантированно разные.
* Если *hashCode*равны, то входные объекты не всегда равны.

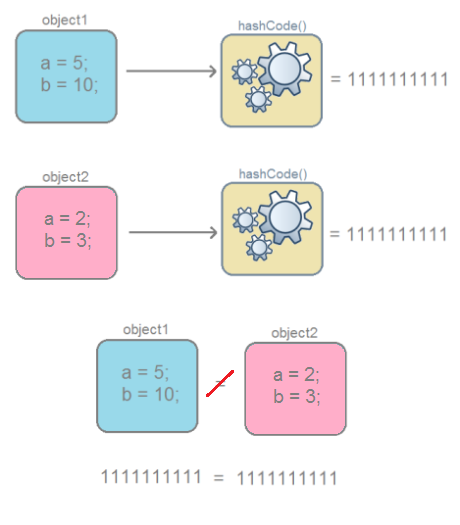
Ситуация, когда у *разных*объектов *одинаковые hashCode*называется — **коллизией**. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

##### **Подведём итог:** Сперва, что-бы избежать путаницы, определимся с терминологией. **Одинаковые объекты — это объекты одного класса с одинаковым содержимым полей.**

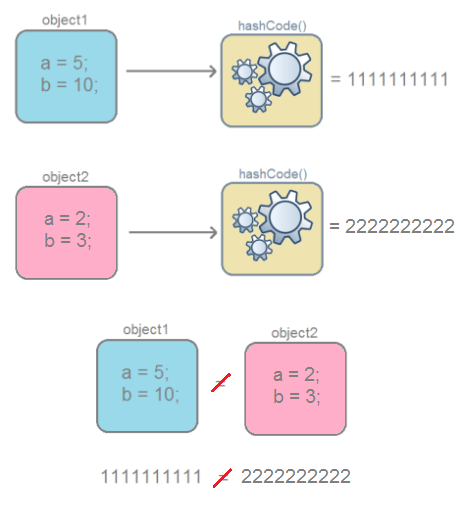
1. для одного и того-же объекта, хеш-код всегда будетодинаковым;
2. если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот, см. правило**3**)



1. если хеш-коды равны, то входные объекты не всегда равны (коллизия);



1. если хеш-коды разные, то и объекты гарантированно разные;



#### *Понятие эквивалентности. Метод equals()*

Начнем с того, что в java, каждый вызов оператора new порождает новый объект в памяти. Для иллюстрации создадим какой-нибудь класс, пускай он будет называться “BlackBox”.  
Пример №2Выполним следующий код:

**publicclassBlackBox**{

**int**varA;

**int**varB;

BlackBox(**int**varA, **int**varB){

**this**.varA = varA;

**this**.varB = varB;

}

}

Создадим класс для демонстрации BlackBox.

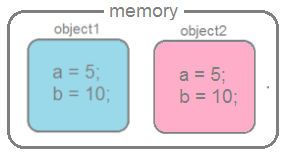
**publicclassDemoBlackBox**{

**publicstaticvoidmain**(String[] args){

BlackBox object1 = **new**BlackBox(5, 10);

BlackBox object2 = **new**BlackBox(5, 10);

}

}  
Во втором примере, в памяти создастся два объекта.  
  
  
  
Но, как вы уже обратили внимание, содержимое этих объектов одинаково, то есть эквивалентно. Для проверки эквивалентности в классе Object существует метод equals(), который сравнивает содержимое объектов и выводит значение типа booleantrue, если содержимое эквивалентно, и false — если нет.

object1.equals(object2);// должно быть true, поскольку содержимое объектов эквивалентно

**Эквивалентность(equals()) и хеш-код(hashCode()) тесно связанны между собой, поскольку хеш-код вычисляется на основании содержимого объекта (значения полей) и если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые (см.**[**правило 2**](https://habr.com/ru/post/168195/#2)**).**  
Иными словами:

object1.equals(object2)// должно быть true

object1.hashCode() == object2.hashCode()// должно быть true  
Я написал “должно быть”, потому что если вы выполните предыдущий пример, то на самом деле результатом выполнения всех операций будет false. Для пояснения причин, заглянем в исходные коды класса Object.

#### *Класс Object.*

Как известно, все java-классы наследуются от класса Object. В этом классе уже определены методы hashCode() и equals().  
Определяя свой класс, вы автоматически наследуете все методы класса Object. И в ситуации, когда в вашем классе не переопределены (@overriding) hashCode() и equals(), то используется их реализация из Object.  
  
*Рассмотрим исходный код метода equals() в классе Object.*

**publicbooleanequals**(Objectobj){

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

}  
При сравнение объектов, операция “==” вернет true лишь в одном случае — когда ссылки указывают на один и тот-же объект. В данном случае не учитывается содержимое полей.

Выполнив приведённый ниже код, equals вернет true.

**publicclassDemoBlackBox**{

**publicstaticvoidmain**(String[] args){

BlackBox object3 = **new**BlackBox(5, 10);

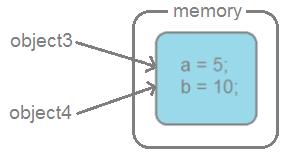
BlackBox object4 = object3;// Переменная object4 ссылаетсяна

//тот-же объект что и переменная object3

object3.equals(object4)//true

}

}

  
  
Теперь понято, почему Object.equals() работает не так как нужно, ведь он сравнивает ссылки, а не содержимое объектов.  
Далее на очереди hashCode(), который тоже работает не так как полагается.  
  
*Заглянем в исходный код метода hashCode() в классе Object:*

**publicnativeinthashCode**();

Вот собственно и вся реализация. Ключевое слово **native** означает, что реализация данного метода выполнена на другом языке, например на C, C++ или ассемблере. Конкретный nativeinthashCode() реализован на C++, вот исходники — <http://hg.openjdk.java.net/jdk7/jdk7/hotspot/file/tip/src/share/vm/runtime/synchronizer.cpp> функция get\_next\_hash.  
  
При вычислении хэш-кода для объектов класса Object по умолчанию используется [Park-Miller RNG](http://en.wikipedia.org/wiki/Park-Miller_random_number_generator) алгоритм. В основу работы данного алгоритма положен генератор случайных чисел. Это означает, что при каждом запуске программы у объекта будет разный хэш-код.  
  
Получается, что используя реализацию метода hashCode() от класса Object, мы при каждом создании объекта класса newBlackBox(), будем получать разные хеш-коды. Мало того, перезапуская программу, мы будем получать абсолютно разные значения, поскольку это просто случайное число.  
  
Но, как мы помним, должно выполняться правило: *“если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые ”*. **Поэтому, при создании пользовательского класса, принято переопределять методы hashCode() и equals() таким образом, что бы учитывались поля объекта.**  
Это можно сделать вручную либо воспользовавшись средствами генерации исходного кода в IDE. Например, в Eclipse это *Source* → *GeneratehashCode() andequals()...*

В итоге, класс *BlackBox* приобретает вид:

**publicclassBlackBox**{

**int**varA;

**int**varB;

BlackBox(**int**varA, **int**varB){

**this**.varA = varA;

**this**.varB = varB;

}

@Override

**publicinthashCode**(){

**finalint** prime = 31;

**int** result = 1;

result = prime \* result + varA;

result = prime \* result + varB;

**return** result;

}

@Override

**publicbooleanequals**(Object obj){

**if** (**this** == obj)

**returntrue**;

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

**returnfalse**;

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

**returnfalse**;

BlackBox other = (BlackBox) obj;

**if** (varA != other.varA)

**returnfalse**;

**if** (varB != other.varB)

**returnfalse**;

**returntrue**;

}

}

Теперь методы hashCode() и equals() работают корректно и учитывают содержимое полей объекта:

object1.equals(object2); //true

object1.hashCode() == object2.hashCode(); //true

#### Итоги: Создавая пользовательский класс, нужно переопределять методы hashCode() и equals(), что бы они корректно работали и учитывали данные объекта. Кроме того, если оставить реализацию из Object, то при использовании java.util.HashMap возникнут проблемы, поскольку HashMap активно используют hashCode() и equals() в своей работе, но про это хорошо написано у [tarzan82](https://habrahabr.ru/users/tarzan82/) в посте [Структуры данных в картинках.HashMap](http://habrahabr.ru/post/128017/).

#### 50. Какой метод возвращает строковое представление объекта?

someObject.toString(); (+ см.ответ 62 «Вывод на консоль».. *'com.cloneable.core.services.CloneableService@4563e9ab'*)

#### 51. Что будет, если переопределить equals() не переопределяя hashCode? Какие могут возникнуть проблемы?

Нарушится контракт. Классы и методы, которые использовали правила этого контракта могут некорректно работать. Так для объекта HashMap это может привести к тому, что пара, которая была помещена в Map, возможно не будет найдена в ней при обращении к Map, если используется новый экземпляр ключа.

#### 52. Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode?

Те, которые используют при определении метода equals(). Хэш-код должен быть равномерно распределен на области возможных принимаемых значений.

В идеале использовать простое уникальное поле объетка(либо совокупность простых полей, при отсутствии такового), [простое поле – примитив,класс-оболочка или строка]. При отсутствии вышеуказанного, можно использовать техническое поле, генерируемое для каждого экземпляра для определения его уникальности(guid, uuid).

#### 53. Как вы думаете, будут ли какие-то проблемы, если у объекта, который используется в качестве ключа в HashMap изменится поле, которое участвует в определении hashCode?

Будут. При обращении по ключу мы можем не найти значение.

#### 54. Чем отличается абстрактный класс от интерфейса, в каких случаях что вы будете использовать?

*Абстрактные классы* используются только тогда, когда есть «is a» тип отношений; *интерфейсы* могут быть реализованы классами, которые не связаны друг с другом.

*Абстрактный класс* может реализовывать методы; интерфейс может реализовывать статические методы начиная с 8й версии.

*Интерфейс* может описывать константы и методы. Все методы интерфейса по умолчанию являются публичными (public) и абстрактными (abstract), а поля — public static final. С java 8 в интерфейсах можно реализовывать default и static методы.

В Java класс может наследоваться (реализовывать) от многих интерфейсов, но только от одного абстрактного класса.

С *абстрактными классами* вы теряете индивидуальность класса, наследующего его; с *интерфейсами* вы просто расширяете функциональность каждого класса.

#### 55. Можно ли получить доступ к private переменным класса и если да, то каким образом?

1. Внутри класса доступ к приватной переменной открыт без ограничений(инфа: модификаторы доступа).
2. Доступ вне класса осуществляется через открытые(public) методы : геттеры и сеттеры.
3. Если геттеры и сеттеры отсутствуют - рефлексия(крайне не рекомендуется). Reflection API.
4. Ограничения накладываемые модификаторами доступа внешнего класса не распростроняются на внутренние и вложенные классы(внутренний или вложенный класс имеет полный доступ ко всем членам содержащего его класса, в том числе к членам, объявленным как private).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | publicclassSomeClass{        private String name="SomeNameString";      private Integer x=25;    }    publicclassTestPrivateAccess{        publicstaticvoidmain(String[]args){          SomeClass someClass=newSomeClass();            try{              Field reflectField=SomeClass.class.getDeclaredField("name");//NoSuchFieldException e              Field reflectField2=SomeClass.class.getDeclaredField("x");//NoSuchFieldException e                /\* Если не дать доступ, то будет ошибка              java.lang.IllegalAccessException: Class .. .TestPrivateAccess              can not access a member of class .. .SomeClass with modifiers "private"              \*/              reflectField.setAccessible(true);              reflectField2.setAccessible(true);                String fieldValue=(String)reflectField.get(someClass);//IllegalAccessException ex              Integer fieldValue2=(Integer)reflectField2.get(someClass);//IllegalAccessException ex              System.out.println(reflectField);//private java.lang.Stringru.javastudy.interview.oop.privateFieldAccess.SomeClass.name              System.out.println(fieldValue);//SomeNameString                System.out.println(reflectField2);//private java.lang.Integerru.javastudy.interview.oop.privateFieldAccess.SomeClass.x              System.out.println(fieldValue2);//25          }catch(NoSuchFieldExceptione){              e.printStackTrace();          }catch(IllegalAccessExceptionex){              ex.printStackTrace();;          }        }  } |

#### 56. Что такое volatile и transient? Для чего и в каких случаях можно было бы использовать default?

volatile  — *не используется кэш при обращении к полю(имеется ввиду область памяти в которой JVM может сохранять локальную копию переменной, чтобы уменьшить время обращения к переменной). Для volatile переменной JVM гарантирует синхронизацию для операций чтения/записи, но не гарантирует для операций изменения значения переменной.*

Если было изменено значение какого-либо поля одним потоком(изменяется состояние объекта), *volatile* гарантирует, что все остальные потоки будут видеть актуальное состояние объекта, т.е. после изменения любым потоком.

transient — указание того, что при сериализации/десериализации данное поле не нужно сериализовать/десериализовывать.

Сериализация – перевод описания объекта из частного(конкретный язык программирования) в общий синтаксис(json, xml), понятный всем языкам программирования.

Десериализация – обратный процесс.

default :

Изначально (с Java 1.5) слово default [использовалось](http://www.java2s.com/Tutorial/Java/0020__Language/UsingDefaultValues.htm) для объявления **дефолтного значения элементов аннотации**.  
  
В Java 8 вместе с лямбдами и стримами [появилась](http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html) острая необходимость дополнить стандартные интерфейсы новыми методами. Никто естественно не собирался ломать обратную совместимость, и [было предложено](https://www.jcp.org/en/jsr/detail?id=335) добавить **методы по умолчанию**.  
  
Теперь добавление ключевого слова default к методу интерфейса позволяет добавить ему тело. Все новые методы старых интерфейсов снабжаются дефолтной реализацией.  
  
В реализации такого метода его дефолтный вариант вызывается тем же синтаксисом, что и внешний класс из [вложенного](https://itsobes.ru/JavaSobes/kakie-sushchestvuiut-tipy-v-java): InterfaceName.super.methodName().  
  
Методы по умолчанию подошли еще на шаг к введению в Java беспроблемной версии множественного наследования – [примесям (mixin)](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%81%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Интерфейс не может иметь состояния, поэтому полноценные примеси всё ещё недоступны.

#### 57. Расширение модификаторов при наследовании, переопределении и сокрытии методов. Если у класса-родителя есть метод, объявленный как private, может ли наследник расширить его видимость? А если protected? А сузить видимость?

Действует общий принцип: расширять видимость можно, сужать нельзя.

private методы видны только внутри класса, для потомков они не видны. Поэтому их и расширить нельзя.

#### 58. Имеет ли смысл объявлять метод private final?

Нет, такой метод и так не виден для наследников, а значит не может быть ими переопределен.

#### 59. Какие особенности инициализации final переменных?

* Для поля. Поле помеченное при помощи слова final не может изменить свое значение после инициализации.  
  Не статическое final поле можно инициализировать:
* при описании,
* в конструкторе (во всех),
* в динамическом блоке.  
  Статическое final поле (static final) инициализируется либо в статическом блоке, либо при описании, либо в динамическом блоке(не рекомендуется).
* Значение локальных переменных, а так же параметров метода помеченных при помощи слова final не могут быть изменены после присвоения. *Помечать локальные переменны как final НЕ РЕКОМЕНДУЕТСЯ, т.к.это противоречит code convention.*

#### 60. Что будет, если единственный конструктор класса объявлен как final?

К конструктору не применимо ключевое слово final.

КОСНТРУКТОР – НЕ ЧЛЕН КЛАССА => НЕ УЧАСТВУЕТ В ПЕРЕОПРЕДЕЛЕНИИ.

#### 61. Что такое finalize? Зачем он нужен? Что Вы можете рассказать о сборщике мусора и алгоритмах его работы.

Метод finalize() вызывается перед тем, как объект будет удален garbage collector (сборщик мусора, далее gc). Существует много различных реализаций gc. Основа работы следующая: gc помечает объекты, на которые больше не ссылаются другие объекты для их удаления. Затем на одном из проходов помеченные объекты удаляются.  
Вызов finalize() не гарантируется, т.к. приложение может быть завершено до того, как будет запущена ещё одна сборка мусора. Да, можно отменить удаление объекта с помощью метода finalize(), присвоив его ссылку какому-то статическому полю.

Например: private static final String var = “”;

#### 62. Почему метод clone() объявлен как protected? Что необходимо для реализации клонирования?

Это указывает на то, что хоть метод и есть в классе Object и разработчик желает им воспользоваться, то его нужно переопределить. Для этого нужно реализовать интерфейс Cloneable, чтобы соблюсти контракт.

**Пример.**

Есть интерфейс Service, который расширяет интерфейс-маркер Cloneable. *Итерфейс-маркер – контракт(совокупность абстрактных методов[ которые требую реализацию!])такого интерфейса не содержит методов для реализации!*

*Если класс не реализует Cloneable, то при вызове ДАЖЕ ПЕРЕОРЕДЕЛЁННОГО метода clone(), будет выброшено исключение CloneNotSupportedException*.

Контракт интерфейса Service включает только 1 метод – copy().

package com.cloneable.core.services;  
  
public interface Service extends Cloneable {  
  
 Service copy();  
}

Класс CloneableService. Реализация.

CloneableService реализует интерфейс Service и его контракт – метод copy().

Задача данного метода нивелировать неудобства использования стандартного метода clone():

* Модификатор protected не позволяет вызвать clone() извне, что требует переопределения этого метода с расширением модификатора доступа до public и при этом вызова super-класса(super.clone()), что является bed practice.
* clone() возвращает Object, что требует дополнительной обработки при его использовании: требуется приведение к типу(в примере: (Service) clone();)
* clone() выбрасывает checked(проверяемое..на этапе компиляции) исключение, которое требует обработки(try/catch). Однако, мы уже реализуем Cloneable , что делает *CloneNotSupportedException не возможным исключением!*

package com.cloneable.core.services;  
  
import org.apache.commons.lang3.RandomStringUtils;  
  
import java.util.Objects;  
  
public class CloneableService implements Service {  
  
 public static final String *PRINTABLE\_FIELD* = "%s='%s'";  
 private final String name = RandomStringUtils.*random*(5, true, true);  
   
 @Override  
 public Service copy() {  
 try {  
 return (Service) clone();  
 } catch (CloneNotSupportedException e) {  
 throw new UnsupportedOperationException();  
 }  
 }  
  
 @Override  
 public String toString() {  
 return String.*join*(", ", convertFieldToString("name", name),  
 convertFieldToString("reference", super.toString()),  
 convertFieldToString("hash", Objects.*hash*(name)));  
 }  
  
 private String convertFieldToString(String name, Object value) {  
 return *PRINTABLE\_FIELD*.formatted(name, value);  
 }  
}

Вызов:

* переопределён toString()
  + каждое значение выводится в форматированном виде(метод *convertFieldToString()*);
  + каждое выводимое значение соединяется через *String.join()* по разделителю ", ";
  + "reference" в качестве значения вызывается *super.toString()* суперкласса(т.е.Object!). При этом выводится строковое представление ссылки.
  + "hash" – вывод результата хеш-функции для поля “name”(Objects.*hash*(name)).

public static void main(String... args){

Service service = new CloneableService();

System.*out*.printf("Original: name - %s, state - %s%n", service.getClass().getSimpleName(), service);  
 Object clone = service.copy();  
 System.*out*.printf("Clone: name - %s, state - %s%n",

clone.getClass().getSimpleName(), clone);  
}

**Вывод результата в консоль:**

Original: name - CloneableService, state - name='C8aBS', reference='com.cloneable.core.services.CloneableService@4563e9ab', hash='63639580'

Clone: name - CloneableService, state - name='C8aBS', reference='com.cloneable.core.services.CloneableService@11531931', hash='63639580'