jvm - Java Virtual Machine - среда ("виртуальная"), в которой выполняется твой скомпилированный код. Она выполняет твой код.  
jre - Java Runtime Environment. Просто пакет, который включает в себя JVM и минимальный набор библиотек для работы программ. И еще браузерный плагин, где будут выполняться апплеты.  
jdk - Java Development Kit - это уже полноценный набор библиотек и инструментов для для создания, компилирования и дебага программ. Включает в себя jre

JVM -> JRE -> JDK

JDK включает в себя JRE. Ядром JRE является JVM. Hеобходимо запомнить, что JRE – это минимальная реализация для исполнения Java-приложений, состоящая из JVM и библиотеки Java-классов. JDK включает в себя JRE и является комплектом для разработки приложений на языке Java

Итак, история подарила нам байт – минимальный объём памяти, который мы можем использовать. И состоит он из 8 бит. Самый маленький целый тип данных в java – byte. Это знаковый 8-битовый тип.

compile error - красная линия в коде, ее находит компилятор

runtime error - внизу выскакивает ошибка, там , где должен быть результат.

Положительная бесконечность - это положительное число, настолько большое, что оно не может быть представлено нормально. Отрицательная бесконечность - это отрицательное число, настолько большое, что оно не может быть представлено нормально. NaN означает "Не число" и результат математической операции, которая не дает числового деления 0 на 0.

System.out.println(0f / 0f);

System.out.println(1f / 0f);

System.out.println(-1f / 0f);

Вывод:

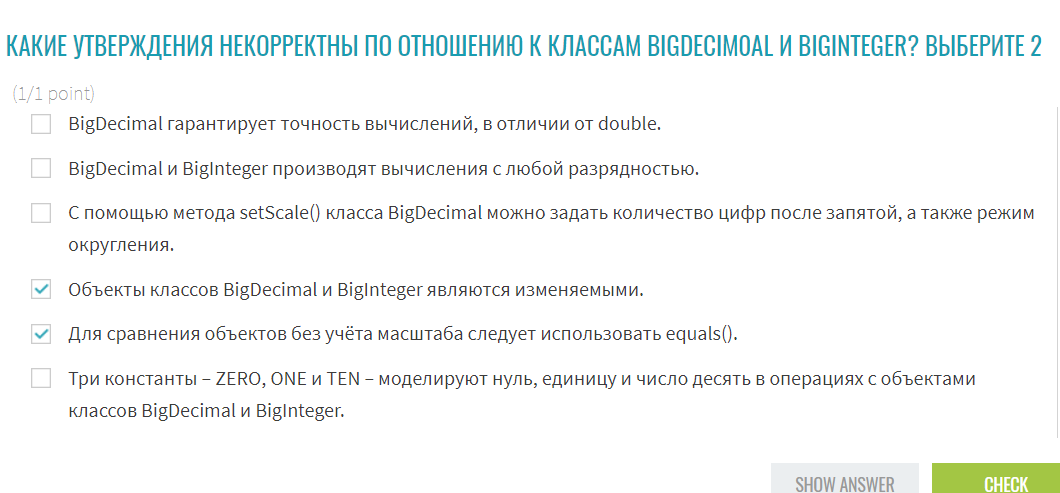
NaN

Infinity

-Infinity

К операциям, приводящим к появлению NaN в качестве ответа, относятся:

* все математические операции, содержащие NaN в качестве одного из операндов;
* деление нуля на нуль;
* деление бесконечности на бесконечность;
* умножение нуля на бесконечность;
* сложение бесконечности с бесконечностью противоположного знака;
* вычисление квадратного корня отрицательного числа[1];
* логарифмирование отрицательного числа.



**short** j = **'ъ'**;*// символ можно привести ко всем кроме byte, boolean т.к. char 16 бит, а byte 8 бит***char** ch = 100; *//без кавычек, работает, получаются символы.* диапазон хранимых значений от 0 до 65535

**char ch = ‘1’ - если привести к int выведет 49.**

**В ‘ ’ помещается только 1 символ (число, буква, знак...). От 0 до 9 будет выведено цифра-символ, который не будет означать соотв. Число. Это символ. Неоходимо приведение к int.**

Нередки ситуации, когда приходится применять различные операции, например, сложение и произведение, над значениями разных типов. Здесь также действуют некоторые правила:

если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double(-float-long)

иначе все операнды операции преобразуются к типу int (даже если складывается byte + short = int)

Если в операциях участвуют данные типа char, то они преобразуются в int:

Метод remainderUnsigned () - это метод класса Java Integer, который возвращает беззнаковый остаток от деления первого аргумента на второй аргумент, где каждый аргумент и результат интерпретируются как беззнаковое значение.

Выделите текст, чтобы посмотреть примеры

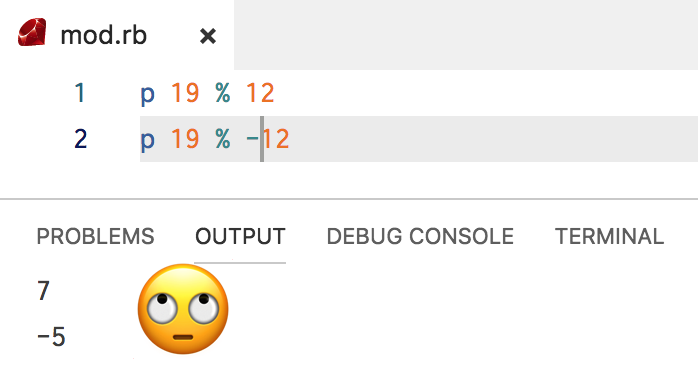
1. **public** **class** IntegerRemainderUnsignedExample1 {
2. **public** **static** **void** main(String[] args) {
3. **int** a = 55;
4. **int** b = 16;
5. System.out.println("Output Remainder: "+Integer.remainderUnsigned(a, b));

===7 - остаток от деления

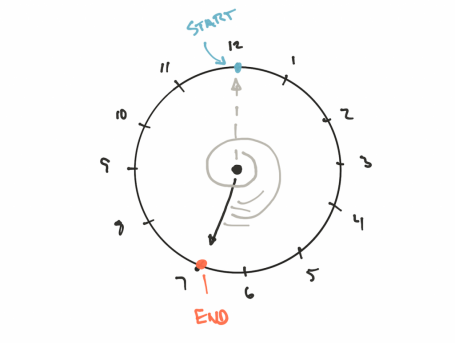
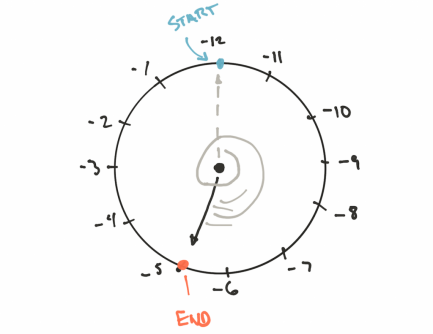
Помимо &, | и ^ в Java также используются поразрядные операторы:

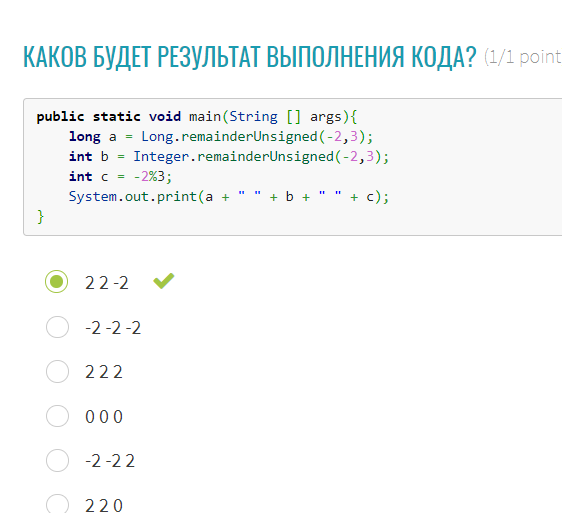
* ~ поразрядный оператор отрицания
* >> побитовый сдвиг вправо
* >>> беззнаковый побитовый сдвиг вправо
* << побитовый сдвиг влево

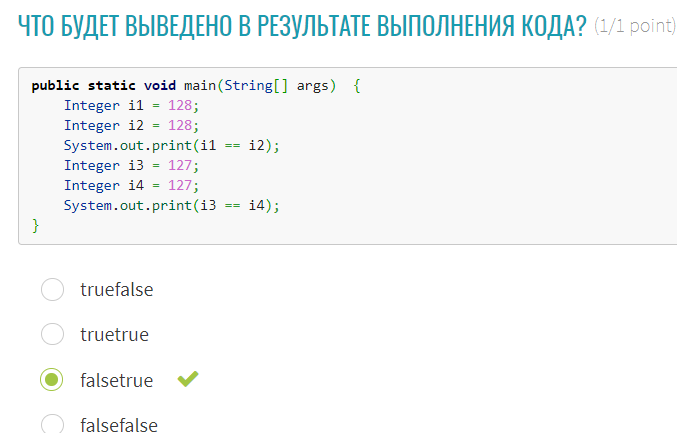
<https://habr.com/ru/post/421071/>

Ruby согласен с Google:  
  
  
  
**Во имя Дейкстры, что здесь происходит?**

# Вращение часов назад

Чтобы ответить на вопрос, следует понять разницу между *остатком* и *modulo*. **Программисты объединяют эти операции**, но не должны этого делать, потому что они дают одинаковый результат только в случае, если делитель (в нашем случае 12) положителен. Вы можете легко отправить баги в продакшн, если делитель отрицательный.  
  
Но почему существует разница? Рассмотрим положительный делитель 19 mod 12 на часах:  
  
  
  
Конечный результат 7. Мы это знаем и мы можем доказать математически. Но что насчёт 19 mod -12? **Здесь нужно использовать другие часы**:  
  
  
  
Модуль равен -12, и мы не можем игнорировать или изменить его, умножив на -1, поскольку модульная арифметика так не работает. Единственный способ правильно рассчитать результат — переставить метки на часах так, чтобы мы двигались от -12 или вращали часы против часовой стрелки, что даёт тот же результат.  
  
Почему не начать метки с -1, двигаясь к -2, и т.д.? *Потому что в таком случае мы будем двигаться назад* и постоянно уменьшать результат, пока не достигнем -12, и в этот момент сделаем прыжок +12, а modulo так не работает.





Ключ к ответу называется интернированием объектов. Java ставит небольшие числа (-128 до 127 вкл), поэтому все экземпляры Integer(n) с n в интернированном диапазоне одинаковы. Числа, которые больше или равны 128, не интернированы, поэтому объекты Integer(1000) не равны друг другу.

Если вы посмотрите на исходный код для Integer, вы увидите, что Integer.valueOf(int) ****пулы**** все значения от -128 до 127. Причина в том, что небольшие значения Integer используются часто и, таким образом, достойный объединения/кэширования.

Взято прямо из Integer.java:

public static Integer valueOf(int i) {

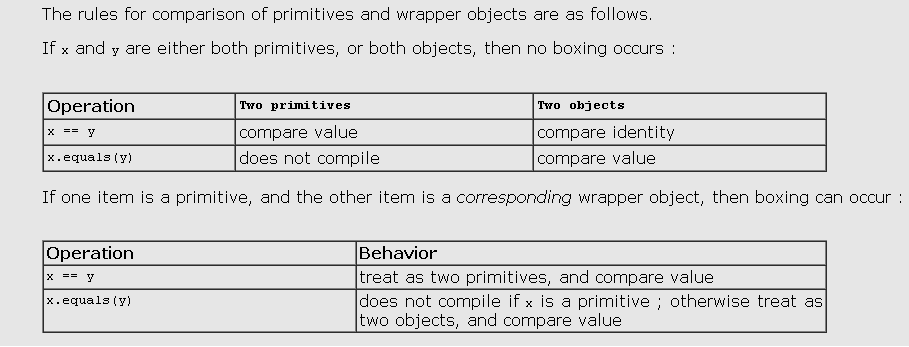
if(i >= -128 && i <= IntegerCache.high)

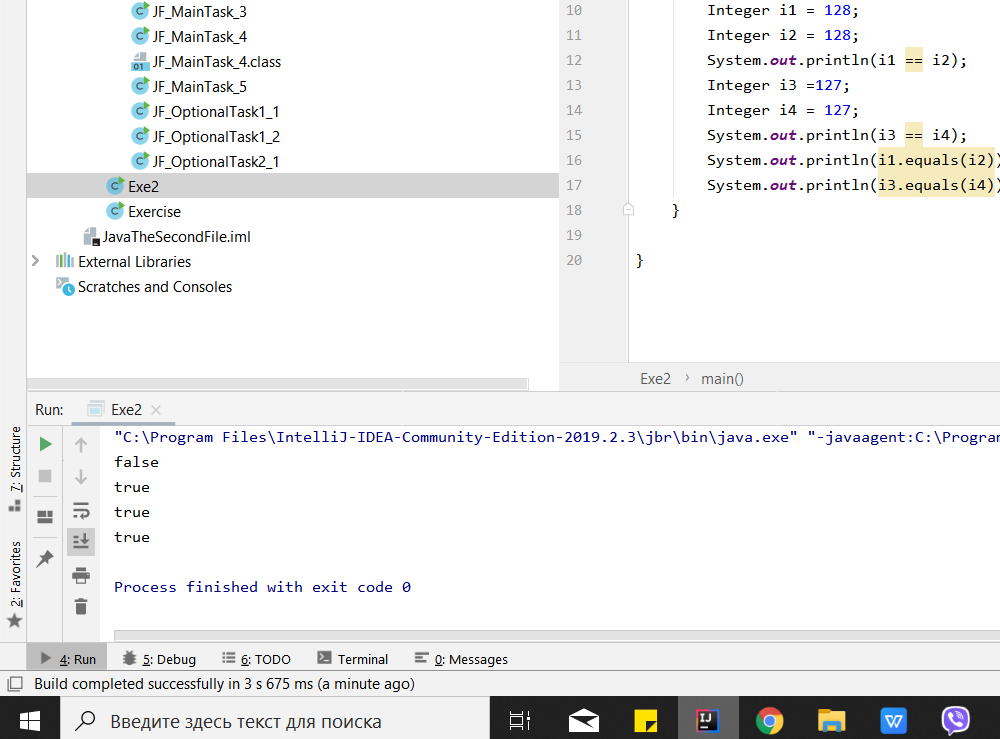
return IntegerCache.cache[i + 128];

else

return new Integer(i);

Простое правило, которое нужно помнить при работе с объектами, - используйте .equals если вы хотите проверить, равны ли два объекта(их значения), используйте == когда вы хотите увидеть, указывают ли они на один и тот же экземпляр(ссылки).





Динамическое и статическое связывание

Существует два типа связывания методов в языке Java: ранее связывание (его ещё называют статическим) и позднее (соответственно, динамическое) ****связывание****. Вызов метода в Java означает, что этот метод привязывается к конкретному коду или в момент компиляции, или во время выполнения, при запуске программы и создании объектов. Можно понять из названия, статическое связывание носит более статический характер, так как происходит во время компиляции, то есть код «знает», какой метод вызывать после компиляции исходного кода на Java в файлы классов. А поскольку это относится к ранней стадии жизненного цикла программы, то называется также ранним связыванием (early binding). С другой стороны, динамическое связывание происходит во время выполнения, после запуска программы виртуальной машиной Java. В этом случае то, какой метод вызвать, определяется конкретным объектом, так что в момент компиляции информация недоступна, ведь объекты создаются во время выполнения. А поскольку это происходит на поздней стадии жизненного цикла программы, то называется в языке Java поздним связыванием (late binding).

Итак, ****фундаментальное различие между статическим и динамическим связыванием в Java**** состоит в том, что первое происходит рано, во время компиляции на основе типа ссылочной переменной, а второе – позднее, во время выполнения, с использованием конкретных объектов.

this() vs super()

Как я уже сказал в начале, главное отличие между ****this**** и ****super**** в Java в том, что ****this**** представляет текущий экземпляр класса, в то время как ****super**** - текущий экземпляр родительского класса.

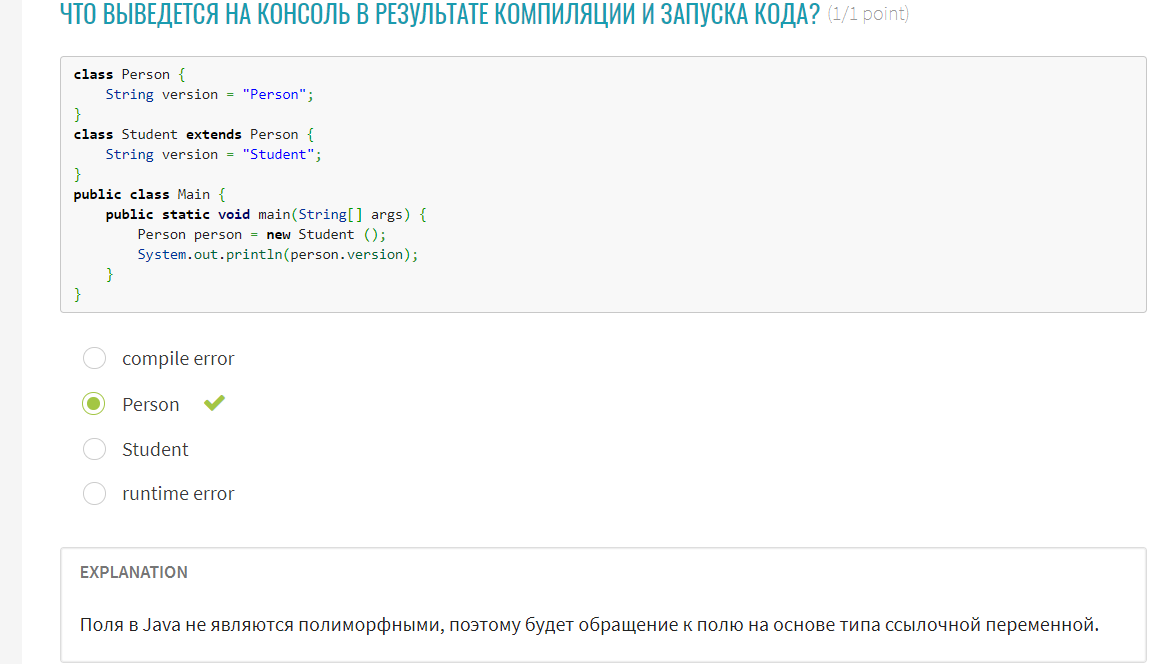
Внутри класса для вызова своего конструктора без аргументов используется ****this()****, тогда как ****super()****используется для вызова конструктора без аргументов, или как его ещё называют, конструктора по умолчанию родительского класса. Между прочим, таким способом вызывать можно не только конструктор без аргументов, а и вообще любой другой конструктор, передав ему соответствующие параметры.

****this**** и ****super**** в Java используются для обращения к переменным экземпляра класса и его родителя. Вообще-то, к ним можно обращаться и без префиксов ****super**** и ****this****, но только если в текущем блоке такие переменные не перекрываются другими переменными, т.е. если в нем нет локальных переменных с такими же именами, в противном же случае использовать имена с префиксами придется обязательно, но это не беда, т.к. в таком виде они даже более читабельны.

1. И this, и super — это нестатические переменные, соответственно их нельзя использовать в статическом контексте, а это означает, что их нельзя использовать в методе main. Это приведет к ошибке во время компиляции "на нестатическую переменную this нельзя ссылаться из статического контекста". То же самое произойдет, если в методе main воспользоваться ключевым словом super.
2. И this, и super могут использоваться внутри конструкторов для вызова других конструкторов по цепочке, нпр., this() и super() вызывают конструктор без аргументов наследующего и родительского классов соответственно.
3. Внутри конструктора this и super должны стоять выше всех других выражений, в самом начале, иначе компилятор выдаст сообщение об ошибке. Из чего следует, что в одном конструкторе не может быть одновременно и this(), и super().

Каждый конструктор при отсутствии явных вызовов других конструкторов неявно вызывает с помощью super() конструктор без аргументов родительского класса, при этом у вас всегда остается возможность явно вызвать любой другой конструктор с помощью либо this(), либо super().

Класс не может наследоваться от самого себя. В конструкторе класса нельзя совместно использовать вызовы super() и this(), поскольку такой вызов должен быть всегда первым оператором конструктора. Компилятор не помешает переопределить статические методы в подклассах, однако при их вызове будет использоваться механизм раннего связывания. Аннотацию @Override к статическим методам применять нельзя. Статические методы можно перегружать в подклассах, доступность таких методов зависит от типа ссылки и атрибута доступа. При динамическом связывании версия вызываемого метода определяется на этапе выполнения.



Эта хрень относится к статическим методам и переменным экземпляра

 Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного ссылке в сообщении типа объекта называется полиморфизмом.

* при переопределении методы суперкласса и подкласса носят одинаковое имя и одинаковую сигнатуру типов параметров (см. **[рис. 1](https://www.bestprog.net/wp-content/uploads/2019/06/01.jpg)**)****;
* при перегрузке методы суперкласса и подкласса носят одинаковое имя но разные сигнатуры типов параметров. На рисунке 2 продемонстрирована перегрузка метода в иерархии наследования.

##### **Получение доступа из метода подкласса к методу суперкласса в случае, когда совпадают имена методов и сигнатуры их параметров. Ключевое слово super.**



В одном файле может быть много классов, но публичным должен быть тем, что назван как файл. Остальные в этом файле - без паблик.

В одном файле не может быть двух public классов.

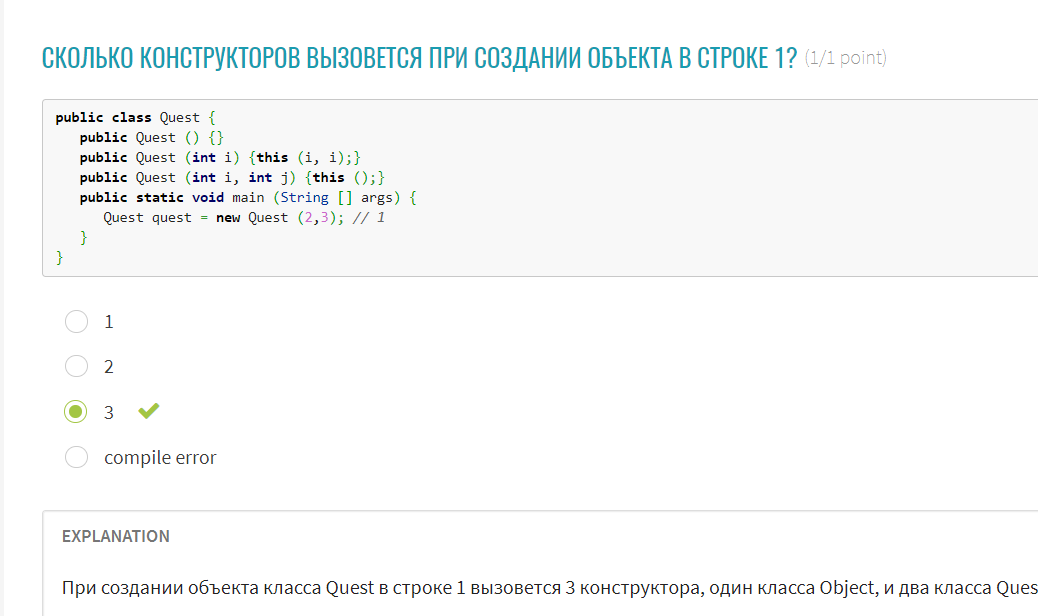
Конструктор может быть объявлен только со спецификаторами public, private, protected или без спецификатора. В случае использования private и protected объект класса может быть создан с помощью статического метода класса, который вызывает такой конструктор.

Применение final или abstract при объявлении конструктора не имеет смысла, потому что он не участвует в наследовании.

Методы, объявленные как private, не наследуются, поэтому на них не распространяются принципы полиморфизма. Так что метод с такой же сигнатурой, объявленный в подклассе, не имеет никакой связи с методом из суперкласса. В таком случае при вызове через ссылку на суперкласс происходит попытка вызвать его private-метод, что приводит к ошибке компиляции.

а также при переопределении метод может иметь более конкретный тип возвращаемого значения. Тип Number can be уточнен как Integer, Float.....

Для разрешения перегрузки сначала ищется метод, тип формального параметра которого совпадает с типом фактического параметра, и только в случае неудачи ищется другой метод, к типу формального параметра которого можно преобразовать передаваемый объект.



Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным:

В тот момент, когда мы пишем:

MonitoringSystem generalModule = new MonitoringSystem() {

};

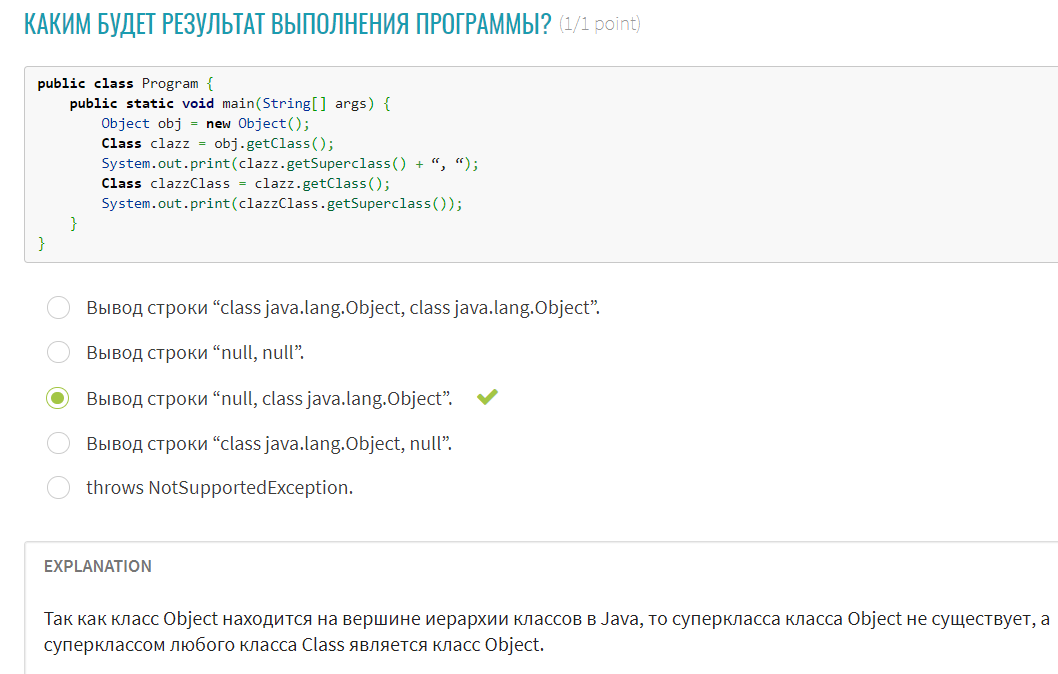
внутри Java-машины происходит следующее:

1. Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem.
2. Компилятор, увидев такой класс, требует от тебя реализовать все методы интерфейса MonitoringSystem (мы это и сделали 3 раза).
3. Создается один объект этого класса. Обрати внимание на код:

MonitoringSystem generalModule = new MonitoringSystem() {

};

В конце стоит точка с запятой! Она стоит там не просто так. Мы одновременно объявляем класс (посредством фигурных скобок) и создаем его объект с помощью ****();****

********

метод **hashCode()**, позволяющий получить уникальный целый номер для данного объекта. Когда объект сохраняют в коллекции типа [HashSet](http://java-online.ru/java-set.xhtml), то данный номер позволяет быстро определить его местонахождение в коллекции и извлечь.

Функция **hashCode()** объекта *Object* возвращает целое число *int*, размер которого равен 4-м байтам.

Следует понимать, что множество возможных хеш-кодов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: “Множество объектов мощнее множества хеш-кодов”. Из-за этого ограничения, вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.  
  
Здесь главное понять, что:

Если хеш-коды разные, то и входные объекты гарантированно разные.

Если хеш-коды равны, то входные объекты не всегда равны.

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

Заглянем в исходный код метода hashCode() в классе Object:

public native int hashCode();

Вот собственно и вся реализация. Ключевое слово native означает, что реализация данного метода выполнена на другом языке, например на C, C++ или ассемблере.

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 new BlackBox(), 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 → Generate hashCode() and equals()…

Сериализация (Serialization) — это процесс, который переводит объект в последовательность байтов, по которой затем его можно полностью восстановить. Зачем это нужно? Дело в том, при обычном выполнении программы максимальный срок жизни любого объекта известен — от запуска программы до ее окончания. Сериализация позволяет расширить эти рамки и «дать жизнь» объекту так же между запусками программы.

Необходимо, чтобы ваш класс реализовывал интерфейс Serializable. Это интерфейс — идентификатор, который не имеет методов, но он указывает jvm, что объекты этого класса могут быть сериализованы. Так как механизм сериализации связан с базовой системой ввода/вывода и переводит объект в поток байтов, для его выполнения необходимо создать выходной поток OutputStream, упаковать его в ObjectOutputStream и вызвать метод writeObject(). Для восстановления объекта нужно упаковать InputStream в ObjectInputStream и вызвать метод readObject().  
  
В процессе сериализации вместе с сериализуемым объектом сохраняется его граф объектов. Т.е. все связанные с этим объекто, объекты других классов так же будут сериализованы вместе с ним.

