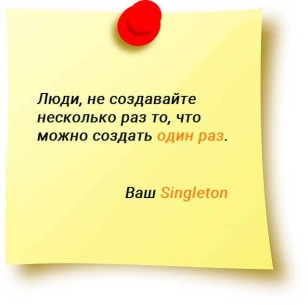
**Что такое Singleton?**

Что такое Singleton можно объяснить одной фразой:



Звучит логично, правда? Мы привыкли, что обычно создаем несколько объектов одного класса - вспомним хотя бы **String**. Но иногда наоборот - что-то необходимо использовать только **ОДИН** раз. Можно найти примеры этого и в реальной жизни:

* Вы не берете новую ручку каждый раз, когда пишете новое слово?
* Вы не покупаете новый автомобиль каждый раз, когда нужно куда-то ехать?

Вот так и в программировании -**иногда нам хватает только одного экземпляра класса** для того, чтобы выполнить все поставленные задачи.

Итак, представьте, что в программе Вам нужно:

* получить класс, который имеет всего **один экземпляр класса**
* в таком случае  все, кто обращается к этому классу, будут получать **один и тот же экземпляр**.
* желательно - для потокобезопасности - должен быть **stateless**(информацию про **stateless**можно прочитать [тут](https://vertex-academy.com/tutorials/ru/stateless-i-immutable-java/)).

Именно такой класс называется **Singleton**-ом.

**Чем Singleton НЕ ЯВЛЯЕТСЯ?**

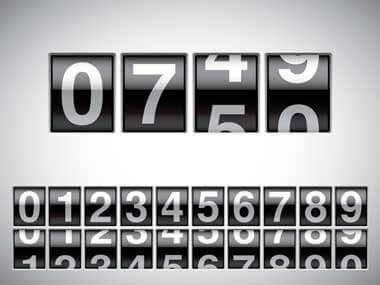
* **Singleton** - это не метод, а класс.
* **Singleton** - это не готовый рецепт создания класса. Есть несколько способов сделать класс синглтоном - мы их рассмотрим далее. Но "начинка" этих классов остается за Вами.

**Может ли меняться Singleton?**

Да, состояние **Singleton**-а может быть как изменяемым, так и не изменяемым.

* **Пример 1 - изменяемый Singleton**

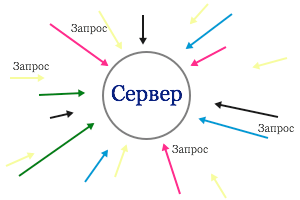
Представьте, что у Вас есть сосед Вася. Вася хакер и хочет взломать Вашу почту. Он подбирает пароли - один за другим - и вводит их по очереди.



Именно для защиты от таких Вась и была придумана **блокировка.** То есть, например, ввел 5 раз неправильно пароль, все - ты заблокирован - и 30 минут не можешь вводить никакие пароли.

А теперь представим, что именно **Вам** нужно реализовать систему защиты от таких вот Вась на своем сервере.

Обратите внимание, что множественный ввод не так легко отследить как кажется. На Ваш сервер одновременно поступает огромное количество информации.



Кроме того, Вася может пытаться вводить пароль с разных IP-адресов, что еще больше усложняет задачу. А нам ведь нужно как-то контролировать количество попыток входа в конкретную учетную запись.

В такой ситуации нам как раз и поможет **Singleton**!



Каждый раз, когда будет запрашиваться доступ к определенной учетной записи, мы будем перенаправлять этот запрос на некий класс, реализующий паттерн **Singleton** (ведь нам же надо хранить эту информацию только в единственной "точке").Данный класс будет считать количество неудачных попыток зайти на сайт.

В данном случае наш **Singleton** будет изменяемым - каждый раз, когда будет осуществляться попытка доступа к нашей учетной записи, переменная, которая считает количество попыток, будет меняться.

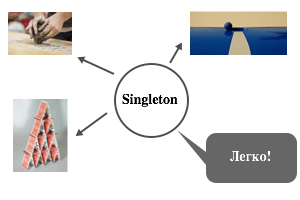
Так, чтобы решить эту проблему, нам понадобился изменяемый **Singleton**.

**Пример 2 - неизменяемый Singleton**



Одним из примеров неизменяемого **Singleton**-а может быть класс, который ставит водяной знак (watermark) на изображения, чтобы предотвратить их кражу.

В принципе, поставить вотермарку на изображение - задача достаточно простая. Классу нужно только получить изображение и отдать уже отредактированную картинку. Если водяной знак будет **один и тот же** на всех изображениях, тогда нет необходимости менять что-либо и в состоянии класса, поэтому он будет **неизменяемым**. С этой задачей сможет справиться один экземпляр этого класса,  а значит можем создать **класс на основе паттерна Singleton**, который будет выполнять **одну и ту же задачу** - ставить водяные знаки на изображения.



Несмотря на то, что паттерн **Singleton** может показаться легким в теории, на практике можно наткнуться на "подводные камни". Давайте посмотрим на примерах.

**Как и когда создавать Singleton?**

* **"Я создаю класс. Мне нужно будет несколько инстансов в будущем или нет?"**

Любой класс, которого Вам **понадобится только один экземпляр**можно сделать **Singleton**-ом. Вот подумайте - сколько памяти Вы сэкономите... Сколько времени... Ммм... Красота.

* Должен ли **Singleton**быть доступен из любой точки программы?

В Интернете часто можно увидеть, что **Singleton** якобы должен быть доступен "из любой точки программы". Это не совсем так.  **Singleton**, как и любые другие данные, должны быть доступны только там, где они нужны. Если они нужны в части программы А - нет необходимости чтобы ее видели части В, C и D.

Тем не менее, обратите внимание на то, какая видимость должна быть у Вашего **Singleton**-а.

* Обратите внимание - нужно ли Вам работать с потоками.

В зависимости от того, работаете Вы с потоками или нет, можно будет выбрать оптимальную реализацию**Singleton**-а.

**Пример  - Самый простой вариант реализации**

|  |  |
| --- | --- |
| 1  2  3  4 | public class Singleton {      private Singleton(){...}      public static final Singleton INSTANCE = new Singleton();  } |

Это самый простой пример реализации **Singleton**.

**Плюсы и минусы**

* Честных Singleton в Java не существует

Из-за reflection, нескольких ClassLoader-ов и других причин, нельзя создать настоящий-настоящий Singleton.

* Трудности  при тестировании

Singleton трудно изолировать - это усложняет написание юнит-тестов.

* Нарушает «Принцип единственной обязанности» класса

«Принцип едиственной обязанности» считается хорошим тоном в ООП. Он состоит в том, что класс должен отвечать только за что-то одно. Например, класс "КореньКвадратный" должен отвечать только за извлечения квадратного корня. Он не должен одновременно печь пирог и гладить кота - только извлекать квадратный корень. Класс Singleton, помимо своих прямых обязанностей, занимается контролем количества своих классов. Это считается плохим тоном.

В этой статье мы расскажем 5 реализаций **Singleton**-а:

1. Самый простой вариант реализации

2. Ленивая реализация

3. Ленивая реализация с быстрой инициализацией

4. Хитрая реализация с внутренним классом

5. Самая хитрая реализация

**Пример 1  - Самый простой вариант**

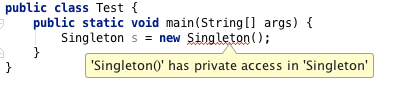
|  |  |
| --- | --- |
| 1  2  3  4 | public class Singleton {      private Singleton(){...}      public static final Singleton INSTANCE = new Singleton();  } |

Итак, что произошло? У нас есть **ПРИВАТНЫЙ**конструктор, а это значит, что:

1. Мы не сможем создать объект класса как мы это привыкли делать. Например, если мы запустим такой код:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Test {      public static void main(String[] args) {          Singleton s = new Singleton();      }  } |

Получим ошибку:



Итак, наш приватный конструктор не позволяет никому создавать новый экземпляр класса. Но как тогда вообще создать хотя бы первый объект? Это происходит внутри самого класса - в этой строке:

|  |  |
| --- | --- |
| 1 | public static final Singleton INSTANCE = new Singleton(); |

Как Вы могли заметить, **INSTANCE** имеет доступ **public**. Это значит, что мы можем обращаться к нему из любого другого класса.

2. Обратите внимание, что мы сделали приватным **конструктор по умолчанию**. Это значит, что мы не сможем наследовать класс **Singleton**.

Например, создадим какой-нибудь класс, который будет наследовать наш **Singleton**:

|  |  |
| --- | --- |
| 1  2 | public class SingletonChild extends Singleton{  } |

Получим ошибку:



А теперь давайте попробуем реализовать его и запустить.

**Реализация**

Пишем такой **Singleton**:

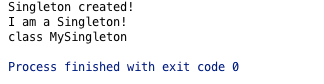
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class MySingleton {        private MySingleton() {          System.out.println("Singleton created!");      }        public static final MySingleton INSTANCE = new MySingleton();        public void printName() {          System.out.println("I am a Singleton!");      }  } |

Тут все как и в примере выше. Но чтобы было интереснее, мы добавили метод **printName()**. Он печатает строку "I am a Singleton!".

Теперь, запустим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Test {      public static void main(String[] args) {            MySingleton.INSTANCE.printName();            System.out.println(MySingleton.INSTANCE.getClass());      }  } |

**Получаем:**



**Комментарии к коду:**

* Так как первым делом при загрузке класса был создан **INSTANCE**, первым в консоли мы видим сообщение из конструктора - "Singleton created!".
* В **main** мы обратились к нашему единственному экземпляру -  **INSTANCE**, через название класса:

MySingleton.INSTANCE

* Сначала мы вызвали метод **printName()**:

MySingleton.INSTANCE.printName();

В консоли получаем "I am a Singleton!".

* Что еще делать с этим **Singleton**-ом?  Применяем на нем универсальный метод **getClass()** и выводим в консоль:

System.out.println(MySingleton.INSTANCE.getClass());

В консоли получаем "class MySingleton".

Поздравляем - теперь мы реализовали наш первый простейший **Singleton**!

**Пример 2  - Ленивая реализация**



* **Почему "ленивая"?**

Реализация называется ленивой не потому, что Вы можете сэкономить на ней время и пойти попить кофе, а потому, что объект **Singleton**-а создается не сразу при загрузке программы, а только "по вызову".

* **Зачем?**

"Ленивая инициализация" полезна, когда создание объекта класса **Singleton** очень ресурсоемкое. Оно может тормозить программу, если будет выполняться при запуске. Мы же создаем объект только когда первый раз вызываем специальный метод.

Код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Singleton {      private static Singleton instance;        private Singleton(){...}        public static synchronized Singleton getInstance() {          if (instance == null) {              instance = new Singleton();          }          return instance;      }  } |

Давайте попробуем реализовать его и запустить.

**Реализация**

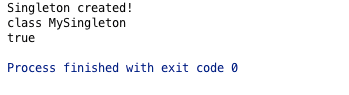
Напишем такой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class MySingleton {      private static MySingleton instance;        private MySingleton() {          System.out.println("Singleton created!");      }        public static synchronized MySingleton getInstance() {          if (instance == null) {              instance = new MySingleton();          }          return instance;      }  } |

Запустим его с помощью следующего кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Test {      public static void main(String[] args) {            MySingleton firstInstance = MySingleton.getInstance();            System.out.println(firstInstance.getClass());            MySingleton secondInstance = MySingleton.getInstance();            System.out.println(firstInstance == secondInstance);      }  } |

**Получаем:**



**Комментарии к коду:**

Теперь наш **instance** создается не при загрузке программы, а только по вызову.

Что мы имеем в коде?

* Сначала мы запрашиваем **instance** с помощью метода **getInstance()**. В консоли видите сообщение, которое появляется при вызове конструктора - "Singleton created!"
* Потом используем наш **firstInstance** - например, применяем на нем метод **getClass()** и выводим в консоль. Получаем "class MySingleton".
* Второй раз вызываем метод **getInstance()**. Как и ожидалось, новый объект не создается. Поэтому, в консоли мы не видим нового сообщения "Singleton created!" (оно появится только при вызове конструктора).
* Кроме того, мы сравниваем **firstInstance** и **secondInstance**. Как видите, они равны - т.е. мы получаем один и тот же объект. Значит, наш **Singleton** настоящий!
* Обратите внимание на модификатор **synchronized** возле метода **getInstance()**. Он нужен для корректной работы в многопоточной среде. Но теперь этот метод - "узкое место" в нашей программе. Только один поток в один момент времени может получить к нему доступ - а это замедляет выполнение программы. Если хотите больше узнать про **synchronized** - прочитайте эту статью.

**Пример 3  -  Ленивая реализация с быстрой инициализацией**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Singleton {      private static volatile Singleton instance;        private Singleton(){...}        public static Singleton getInstance() {          if (instance == null) {              synchronized (Singleton.class) {                  if (instance == null) {                      instance = new Singleton();                  }              }          }          return instance;      }  } |

Давайте попробуем реализовать его и запустить.

**Реализация**

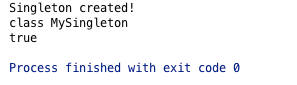
Представим, что у нас есть такой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class MySingleton {      private static volatile MySingleton instance;        private MySingleton(){          System.out.println("Singleton created!");      }        public static MySingleton getInstance() {          if (instance == null) {              synchronized (MySingleton.class) {                  if (instance == null) {                      instance = new MySingleton();                  }              }          }          return instance;      }  } |

Запускаем:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Test {        public static void main(String[] args) {            MySingleton firstInstance = MySingleton.getInstance();            System.out.println(firstInstance.getClass());            MySingleton secondInstance = MySingleton.getInstance();            System.out.println(secondInstance == firstInstance);      }  } |

**Получаем:**



**Комментарии к коду:**

В принциапе, вывод мало чем отличается от вывода в примере номер два. Давайте лучше разберемся с кодом самого **Singleton**-а:)

* Во-первых, у нас есть переменная с модификатором **volatile**. Этот модификатор делает так, что каждый поток не создает свою копию переменной, а все потоки работают с одной и той же. Действительно, зачем нам копии **Singleton**-а:) Он же **Singleton**!
* Дальше у нас есть метод **getInstance()**. Давайте еще раз на него посмотрим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static MySingleton getInstance() {          if (instance == null) {              synchronized (MySingleton.class) {                  if (instance == null) {                      instance = new MySingleton();                  }              }          }          return instance;  } |

Много условий, да?  Они нужны для обеспечения работы в многопоточной среде. Давайте рассмотрим два варианта - сначала более легкий на примере **одного** потока, а потом покажем как будет выполняться код если к методу будут обращаться одновременно **несколько** потоков.

1. **Сначала рассмотрим легкий вариант - когда у нас есть только один поток. Представим, что объект класса MySingleton еще не создан - то-есть наш instance = NULL. Что будет происходить?**

Наш поток зайдет в метод **getInstance()**. Как мы уже говорили, наш **instance**равен**NULL** - поэтому мы заходим в первое условие.

Потом мы видим что? Синхронизированный блок с ключом **MySingleton.class**. Если мы синхронизируем блок по ключу, это значит что в данный момент времени только один поток может иметь доступ к синхронизированным блкам с таким ключом и начинает "владеть" указанным ключом (в нашем случае - **MySingleton.class**). Что такое объект типа класс мы рассмотрим в отдельной статье - а сейчас давайте воспринимать **MySingleton.class**просто как "чертеж", по которому создается конкретный класс.  Без такого чертежа мы не можем создавать новый класс. И что это значит? Это значит что как только наш поток зашел в **synchronized** блок, новый объект данного класса не может быть создан никем, кроме нашего потока.

Итак, наш поток зашел в синхронизированнй блок. Теперь у него есть "чертеж" класса,  и никто не может создать новый объект класса **MySingleton** кроме него.

Теперь мы заходим во второй **if**:

|  |  |
| --- | --- |
| 1  2  3 | if (instance == null) {          instance = new MySingleton();      } |

Как мы уже говорили, наш **instance**пока равен **NULL**. Поэтому происходит что? Мы создаем новый объект класса  MySingleton() - и наш **instance**уже не равен null. Отлично!

Что теперь произойдет при повторном вызове метода **getInstance()**? Мы даже не зайдем в первый **if**потому что наш **instance**уже не пустой. Мы сразу вернем его значение.

2. А теперь давайте рассмотрим вариант если у нас **несколько** потоков. Опять же, начнем сначала - представим, что наш  **instance**пока равен **NULL**.

Представим, что несколько потоков заходят в первый **if**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static MySingleton getInstance() {          if (instance == null) { // все потоки здесь!              synchronized (MySingleton.class) {                  if (instance == null) {                      instance = new MySingleton();                  }              }          }          return instance;  } |

Потом у нас что? Синхронизированный блок! В нем может находится только один поток в один момент времени. Это значит, что один поток который успел первым, зайдет в блок, а остальные будут ждать "снаружи"

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static MySingleton getInstance() {          if (instance == null) {  // все потоки, кроме одного, ждут здесь!              synchronized (MySingleton.class) {  // один поток зашел в синхронизированный метод!                  if (instance == null) {                      instance = new MySingleton();                  }              }          }          return instance;  } |

Что происходит внутри синхронизированного метода? В принципе, тут все как и выше - поток заходит во второе условие и создает новый объект класса **MySingleton()**.

Отлично! Теперь наш первый поток вышел из синхронизированного блока и получает созданный **instance**.

Что происходит дальше? Как только первый поток вышел из синхронизированного блока, в него заходит следующий. Остальные потоки все еще ждут "снаружи".

Как Вы помните, мы уже создали объект класса **MySingleton()** и теперь **instance**не равен **NULL**. Поэтому этот поток не заходит во второй **if**, а просто возвращает **instance**. Так же и другие потоки - поскольку наш **instance**не равен **NULL**, а уже был инициализирован самым первым потоком, они вернут готовый **instance**.

Для этого нужна вторая проверка, что бы наш синглтон создавался только один раз даже в многопоточной среде.

**Пример 4  - Хитрая реализация с внутренним классом**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Singleton {        private Singleton(){...}        private static class SingletonHolder {          public static final Singleton HOLDER\_INSTANCE = new Singleton();      }        public static Singleton getInstance() {          return SingletonHolder.HOLDER\_INSTANCE;      }  } |

Давайте попробуем реализовать его и запустить.

**Реализация**

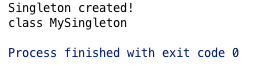
Напишем такой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class MySingleton {        private MySingleton() {          System.out.println("Singleton created!");      }        private static class SingletonHolder {          public static final MySingleton HOLDER\_INSTANCE = new MySingleton();      }        public static MySingleton getInstance() {          return SingletonHolder.HOLDER\_INSTANCE;      }  } |

Запустим его с помощью следующего кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Test {      public static void main(String[] args) {            MySingleton firstInstance = MySingleton.getInstance();            System.out.println(firstInstance.getClass());            MySingleton secondInstance = MySingleton.getInstance();        }  } |

**Получаем:**



**Комментарии к коду:**

Как видите, код в этом примере и в прошлом почти идентичный. Единственная разница - в реализации через внутренний класс.

В чем особенность внутреннего класса? Он позволяет нам, опять же, осуществить ленивую реализацию благодаря особенностям самой Джавы. Дело в том, что внутренние классы загружаются только тогда, когда мы обращаемся к ним впервые - в отличии от "внешних" классов, которые загружаются сразу при запуске программы.

Более того, она потокобезопасна - за счет той же особенности. Почему? Есть такая проблема - если несколько потоков одновременно "стучатся" к одному и тому же объекту, один из них может получить недогруженный объект. Но тут такого нет - благодаря особенностям загрузки внутренних классов

**Пример 5  - Самая хитрая реализация**

|  |  |
| --- | --- |
| 1  2  3 | public enum Singleton {      INSTANCE;  } |

Давайте попробуем реализовать его и запустить.

**Реализация**

Напишем такой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public enum MySingleton {        INSTANCE;        private MySingleton() {          System.out.println("Singleton created! By the way, in Enums the constructor is private by default - so there is no need to write private constructor by yourself");      }  } |

Обратите внимание - по умолчанию конструктор у **Enum** приватный. Это значит, что в отличии от других реализаций, в которых мы **обязательно** должны указывать, что конструктор приватный (иначе **Singleton**-а не выйдет), то тут мы можем явно не объявлять приватный конструктор.

Запустим его с помощью следующего кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Test {      public static void main(String[] args) {          System.out.println(MySingleton.INSTANCE);          System.out.println(MySingleton.INSTANCE);      }  } |

**Получаем:**



**Комментарии к коду:**

Теперь мы реализовали **Singleton**через **Enum**. Это очень удобно - мы можем не прописывать явно приватный конструктор (но тут мы прописали чтобы показать, как создается **Singleton**).

В методе **main** мы обратились к нему два раза. Как видите по сообщению в консоли, **INSTANCE** был создан всего один раз - сообщение **"Singleton created! By the way, in Enums the constructor is private by default - so there is no need to write private constructor by yourself "** было выведено всего один раз.