О модификаторах в Java

<https://translated.turbopages.org/proxy_u/en-ru.ru.5591a089-64044af7-3e7d756a-74722d776562/https/stackoverflow.com/questions/2238730/what-is-the-difference-between-access-specifiers-and-access-modifiers>

<https://javarush.com/groups/posts/1988-modifikatorih-dostupa-private-protected-default-public>

<https://struchkov.dev/blog/modifiers-in-java/>

<https://java-blog.ru/osnovy/modifikatory-dostupa-java>

<https://java-blog.ru/osnovy/modifikatory-dostupa-java>

<https://translated.turbopages.org/proxy_u/en-ru.ru.8769edf6-64044d9b-56d461cf-74722d776562/https/www.baeldung.com/java-access-modifiers>

[https://www.magnumblog.space/java/translating-java-code-conventions#java\_source\_files](https://www.magnumblog.space/java/translating-java-code-conventions" \l "java_source_files) – соглашение об оформлении кода Java.

<https://habr.com/ru/post/440436/> Задачник

**Язык Java – Лекция 8. Основные особенности классов и их объектов. Пакеты классов**

Оглавление

[8.1 Фундаментальные отличия классов и объектов в Java 2](#_Toc129298557)

[8.2 Создание класса и объектов, состояние объекта, методы и поля, конструкторы 2](#_Toc129298558)

[8.2.1 Создание и использование класса 2](#_Toc129298559)

[8.2.2 Создание объекта класса 4](#_Toc129298560)

[8.2.3 Конструкторы класса 5](#_Toc129298561)

[8.2.3.1 Конструктор по умолчанию 5](#_Toc129298562)

[8.2.3.2 Специальные конструкторы 7](#_Toc129298563)

[8.2.4 Подведение промежуточных итогов 8](#_Toc129298564)

[8.2.5 Ключевое слово this 8](#_Toc129298565)

[8.3 Инициализаторы 9](#_Toc129298566)

[8.3.1 Что такое инициализация 9](#_Toc129298567)

[8.3.2 Инициализация с помощью блоков 9](#_Toc129298568)

[8.3.1 Нестатический инициализатор 10](#_Toc129298569)

[8.3.2 Блок статической инициализации 11](#_Toc129298570)

[8.3.3 Зачем используются блоки инициализации 11](#_Toc129298571)

[8.4 Жизненный цикл объекта 12](#_Toc129298572)

# 8.1 Фундаментальные отличия классов и объектов в Java

Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Программа на Java – это набор взаимодействующих между собой объектов, созданных на основе классов.

Аналогия со строительством дома:

* Из элементарных элементов –кирпичей, раствора, дорванных брусков для рам и дверных проёмов, наконец, каменщиков, плиточников и т.д. Для небольших (сравнительно) зданий (программ,) и строим долго. Это аналог структурного (процедурного) программирования.
* Из готовых стеновых блоков со вставленными оконными и дверными проёмами, которые остаётся только смонтировать. Нужны (в основном, практически) только монтажники. Поэтому строительство идёт быстрее и качественней. Это аналог объектно ориентированного программирования.
* Архитектор (инженер) должен знать, какие блоки понадобятся, существуют ли они в действительности (выпускаются промышленностью) или надо организовать их выпуск с нуля.

Где же здесь классы и объекты? Классы появляются, когда мы определяем, какого типа блоки нам нужны. Например, мы решили, что нам нужен блок с одним дверным проемом и двумя окнами. У нас такого блока еще нет, есть только его проект, или эскиз. **Такой проект или эскиз – как раз и есть класс**.

А вот когда мы по этому эскизу создаем реальный блок, появляется объект. По одному эскизу мы можем сделать один блок, два блока, сто блоков или ни одного. Все блоки, выполненные по одному эскизу, с одной стороны, похожи, поскольку все их штамповали по стандартной схеме (точнее, по эскизу), **но в то же время все это физически разные объекты**.

Каждый из них может использоваться по-разному: один покрасили, другой выбросили, третий разбили.

Так и программные объекты. Объекты, или экземпляры класса, создаются на основе одного шаблона (то есть класса), но каждый из них уникален.

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса.

# 8.2 Создание класса и объектов, состояние объекта, методы и поля, конструкторы

### 8.2.1 Создание и использование класса

Создание класса о человеке

***class Person{***

***String name; // имя***

***int age; // возраст***

***void displayInfo(){***

***System.out.printf ("Name: %s \tAge: %d\n", name, age);***

***}***

***}***

Использование класса: создание класса с main и создание объекта класса.

***public class Program{***

***public static void main(String[] args) {***

***Person Tom;***

***}***

***}***

***class Person{***

***String name; // имя***

***int age; // возраст***

***void displayInfo(){***

***System.out.printf("Name: %s \tAge: %d\n", name, age);***

}

}

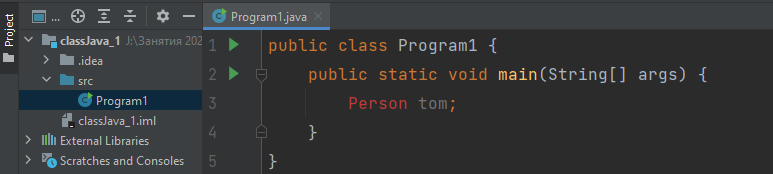


Рисунок 8.1. Попытка введения переменной tom типа Person. Красный цвет сигнализирует об ошибке

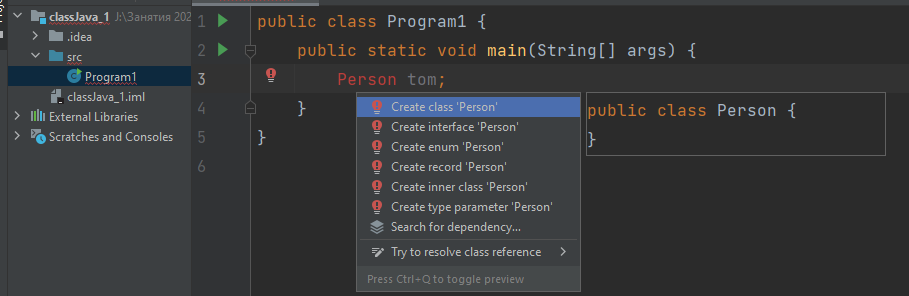
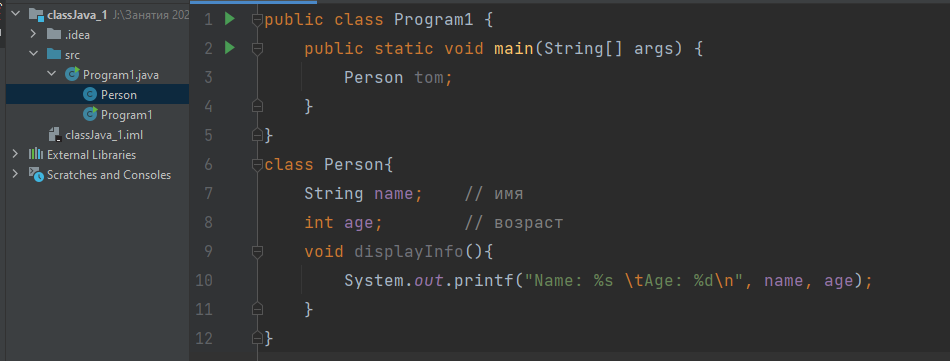


Рисунок 8.2. Предложения о возможном исправлении ошибки



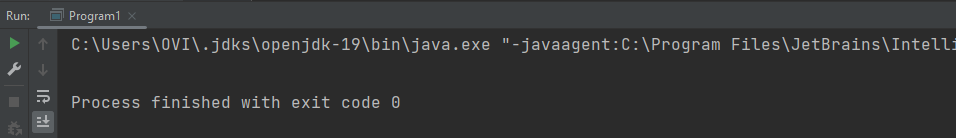


Рисунок 8.3. После создания класса Person ошибка исчезла. Переменная tom определена, но объект класса не создан! Запуск программы заканчивается без печати кодом 0.

Класс представляет новый тип, поэтому мы можем определять переменные, которые представляют данный тип. Так, здесь в методе main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение **null**.

По большому счету мы ее пока не можем использовать, поэтому вначале необходимо создать объект класса Person.

Любой объект может обладать двумя основными характеристиками: **состояние** - некоторые данные, которые хранит объект, и **поведение** - действия, которые может совершать объект.

### 8.2.2 Создание объекта класса

На рисунке 8.4 создаётся объект tom класса Person и вызывается метод displayInfo, который работает, но так как состояние объекта не задано (не определено!), метод выводит значения, присвоенные по умолчанию.

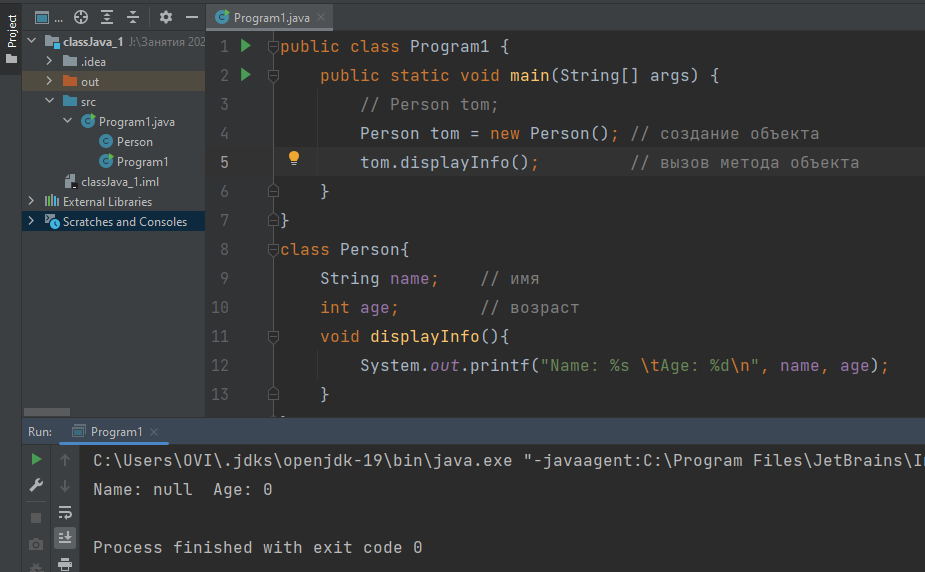


Рисунок 8.4. Вывод значений полей по умолчанию.

### 8.2.3 Конструкторы класса

#### 8.2.3.1 Конструктор по умолчанию

Кроме обычных методов классы могут определять **специальные методы**, которые называются конструкторами.

Конструкторы всегда вызываются при создании нового объекта данного класса и выполняют инициализацию объекта, а также заданные программистом специальные действия. Если программист не создаёт свой конструктор, то для этого класса автоматически создается конструктор без параметров – так называемый конструктор по умолчанию.

Выше определенный класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию, который можно использовать для создания объекта Person. В частности, создадим один объект, как и показано на рисунке 8.4.

Для создания объекта Person используется выражение new Person(). Оператор **new** выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти **будет выделен участок, где будут храниться все данные объекта класса Person**. **А переменная tom получит ссылку на созданный объект.**

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person **через переменную** tom и установить или получить их значения, например, tom.name = "Tom" и tom.age = 34 (рисунок 8.5). Эти значения присвоятся уже созданным в объекте переменным и переопределят их.

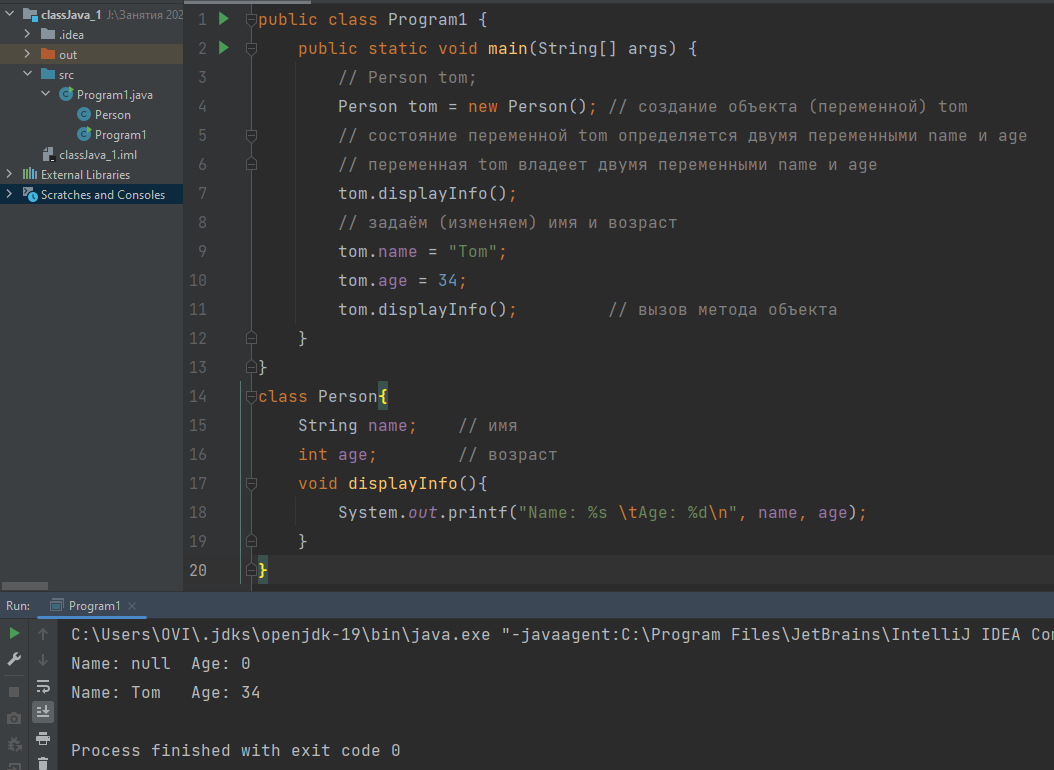


Рисунок 8.5. Переопределение переменных объекта.

На рисунке 8.5 метод displayInfo не имеет формальных параметров и выводит поля объекта. Можно реализовать метод с формальными параметрами, как показано на рисунке 8.6. В этом случае поля **name** и **age** выступают в роли фактических параметров.



Рисунок 8.6. Метод класса имеет формальные параметры.

#### 8.2.3.2 Специальные конструкторы

Если необходимо, чтобы при создании объекта производилась какая-то логика, то можно (и нужно) определить в классе свои конструкторы. Рассмотрим случай, когда конструктор обеспечивает присвоение полям класса определённых значений (рисунок 8.7).

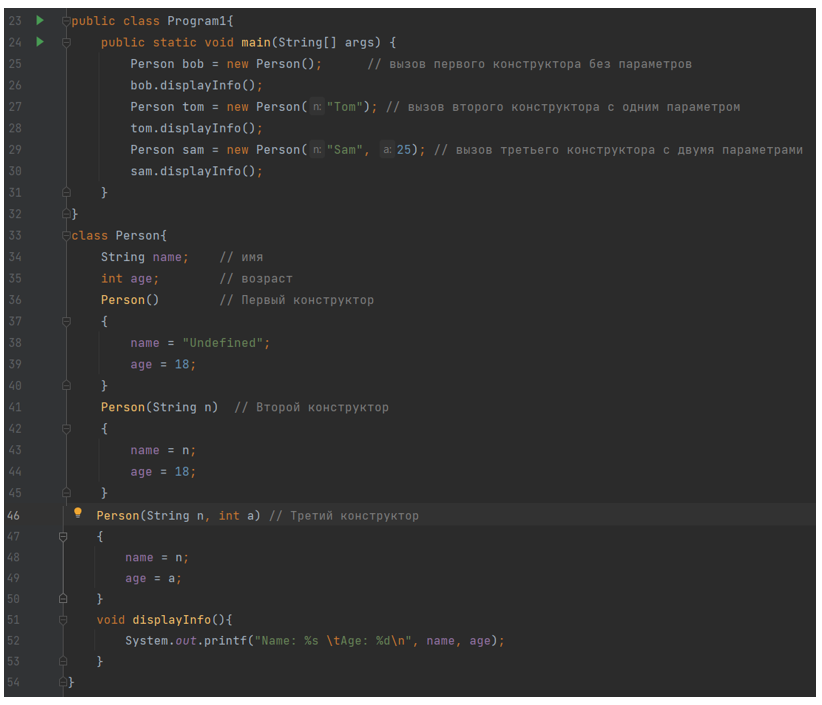


Рисунок 8.7. Класс с тремя конструкторами. Конструкторы имеют формальные параметры.

### 8.2.4 Подведение промежуточных итогов

* Конструктор – это метод, который выполняется вначале, при создании объекта.
* Конструктор создается практически так же, как и обычный метод класса, но есть некоторые ограничения.
* Конструктор не возвращает результат, а в сигнатуре конструктора не указывается идентификатор типа возвращаемого результата – никакой!
* У конструктора могут быть аргументы, которые описываются так же, как и аргументы обычного метода.
* Еще одно очень важное свойство конструктора состоит в том, что его, как и обычный метод класса, можно перегружать (рисунок 8.7, 8.8)

### 8.2.5 Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы (рисунок 8.8).

В третьем конструкторе параметры называются так же, как и поля класса. И чтобы разграничить поля и параметры, применяется ключевое слово this: **this.name = name**. Указываем, что значение **параметра name** присваивается **полю объекта name**.

Кроме того, у нас три конструктора, которые выполняют идентичные действия: устанавливают **поля** **name** и **age**. Чтобы избежать повторов, с помощью this можно вызвать один из конструкторов класса и передать для его параметров необходимые значения:

***Person(String name)***

***{***

***this(name, 18);***

***}***

В итоге результат программы будет тот же, что и в предыдущем примере.

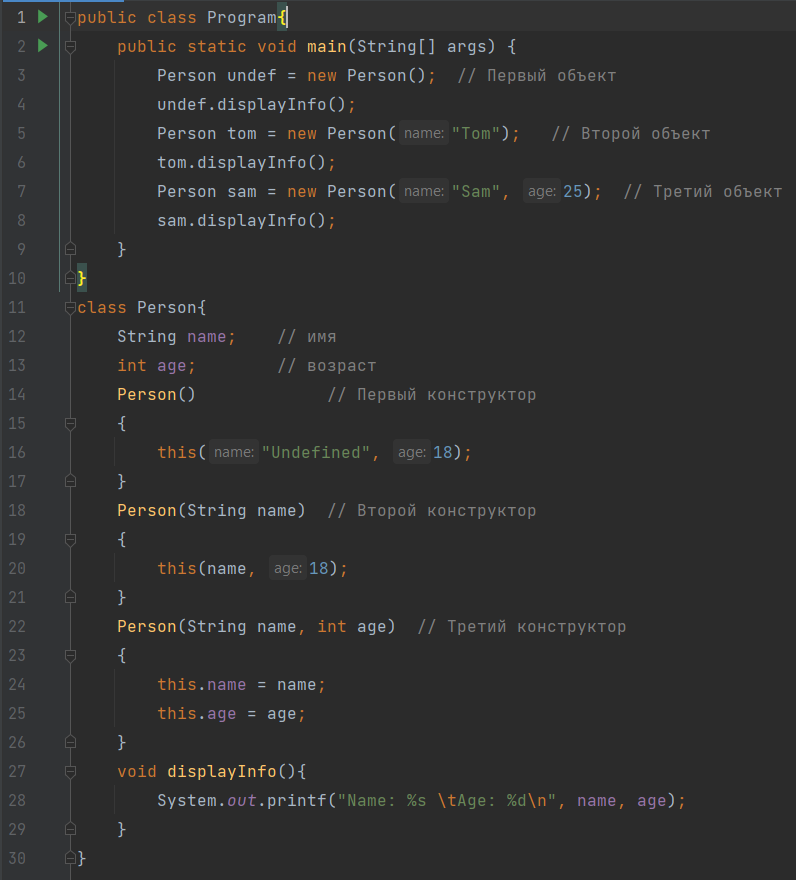


Рисунок 8.8. Перегрузка конструкторов и использование this

## 8.3 Инициализаторы

### 8.3.1 Что такое инициализация

Инициализация (от англ. initialize, от initial - "начальный, первоначальный") - действие, **впервые** задающее переменной какое-либо значение.

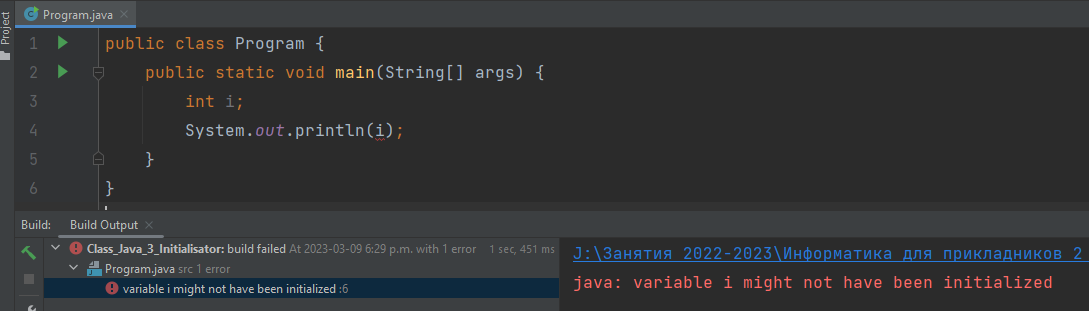
Выражение **int i** – объявление переменной, а не её инициализация. При попытке вывести её на экран (рисунок 8.9) получаем ошибку уже на стадии Build.

Рисунок 8.9. Иллюстрация необходимости инициализации переменной.

На рисунке 8.10 инициализация реализована.

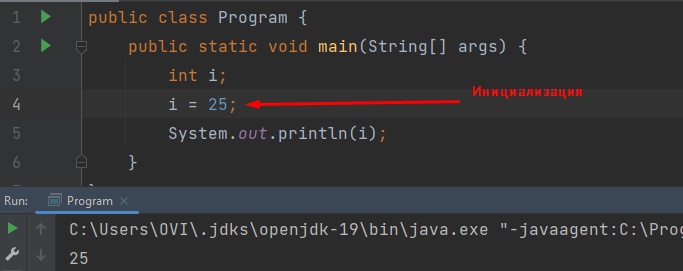


Рисунок 8.10. Пример инициализации.

### 8.3.2 Инициализация с помощью блоков

Представим, что мы хотим задать базовые значения переменных. Можно сделать это так (рисунок8.11а):

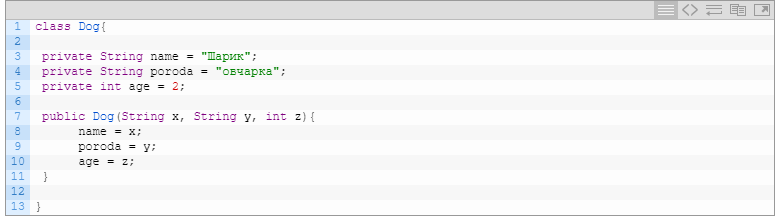


Рисунок 8.11а. Пример «обычной» инициализации.

А можно использовать блок инициализации:

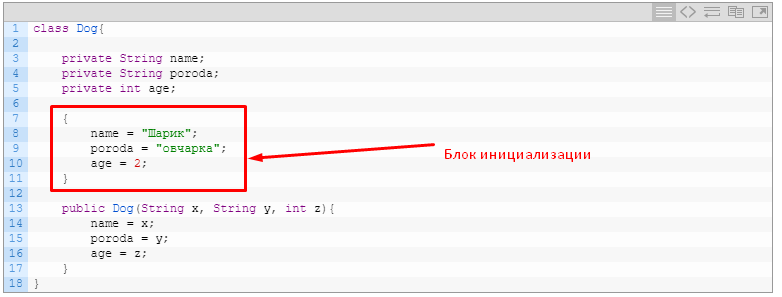


Рисунок 8.11б. Пример использования блока инициализации.

На рисунке 8.12 представлена полная версия использования блока инициализации.

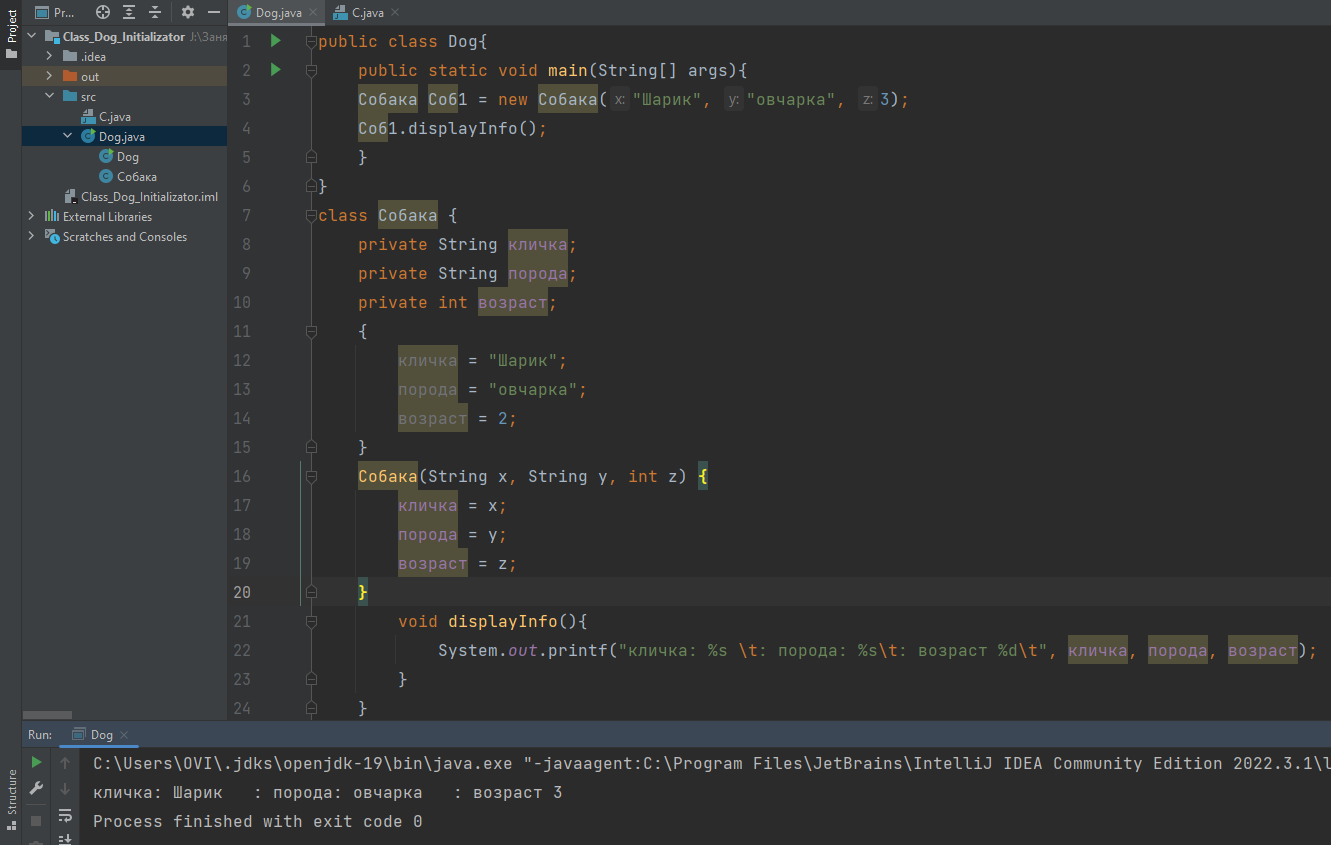


Рисунок 8.12. Использование блока инициализации в реальном контексте.

В Java **инициализатор - это блок кода, который не имеет связанного имени или типа данных** и размещается вне любого метода, конструктора или другого блока кода.

Java предлагает два типа блоков инициализаторов: статические и инициализаторы экземпляра –**нестатический** (*instance initializer – инициализатор объекта)* и **статический** (*class initializer*).

### 8.3.1 Нестатический инициализатор

Таким образом, начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта, а не только с использованием конструкторов.  **Инициализатор выполняется до любого конструктора**.

**В инициализатор мы можем поместить код, общий для всех конструкторов** (рисунок 8.13).



Рисунок 8.13. Использование инициализатора объекта.

### 8.3.2 Блок статической инициализации

Статический инициализатор, или статический блок, представляет собой блок кода, который используется для инициализации *статических* полей. Другими словами, это простой инициализатор, помеченный ключевым словом *static:*

***private static String forum;***

***static {***

***forum = "Java";***

***}***

### 8.3.3 Зачем используются блоки инициализации

Блоки инициализации обеспечивают более читабельный код. Но кроме этого они обеспечивают большую функциональность кода. Внутри блоков инициализации мы можем не только присваивать значения. Это **как метод** - тут можно писать любые команды. Например, вывод в консоль (рисунок 8.14):

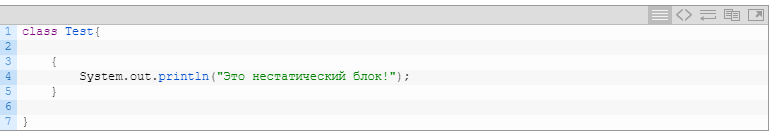


Рисунок 8.14.

**Реализовать печать информации в блоке инициализации в коде на рисунке 8.12. Доказать, что без блока инициализации печать не сработает!**

**Подводим итог**:

* блоки инициализации используются для инициализации переменных внутри класса;
* есть два типа блоков инициализации - статический и нестатический;
* синтаксис очень простой - просто пишем выражение внутри блока из двух скобок;
* в Java порядок инструкций инициализации следующий:
  + статические переменные и статические инициализаторы по порядку;
  + переменные экземпляра и инициализаторы экземпляра по порядку;
  + конструкторы.

## 8.4 Жизненный цикл объекта и сборщик мусора в Java

В Java не нужно беспокоиться об уничтожении объектов, Java заботится об устаревших объектах через свой сборщик мусора.

**Все объекты в Java хранятся в памяти кучи нашей программы**. Фактически, куча представляет собой большой пул неиспользуемой памяти, выделенный для нашего Java-приложения.

С другой стороны, **сборщик мусора - это программа Java, которая заботится об автоматическом управлении памятью** путем удаления объектов, которые больше недоступны.

**Чтобы объект Java стал недоступным, он должен столкнуться с одной из следующих ситуаций**:

* У объекта больше нет ссылок, указывающих на него.
* Все ссылки, указывающие на объект, выходят за рамки.

В заключение, объект сначала создается из класса, обычно с использованием ключевого слова *new.* Затем объект живет своей жизнью и предоставляет нам доступ к его методам и полям.

Наконец, когда он больше не нужен, сборщик мусора уничтожает его.

Сборка мусора — это процесс восстановления заполненной памяти среды выполнения путем уничтожения неиспользуемых объектов.

В таких языках, как C и C++, программист отвечает как за создание, так и за уничтожение объектов. Иногда программист может забыть уничтожить бесполезные объекты, и выделенная им память не освобождается. Расходуется все больше и больше системной памяти, и в конечном итоге она больше не выделяется. Такие приложения страдают от “утечек памяти”.

После определенного момента памяти уже не хватает для создания новых объектов, и программа нештатно завершается из-за OutOfMemoryErrors.

В C++ для сборки мусора можно воспользоваться методом delete(), а в C — методом free(). В Java сборка мусора происходит автоматически в течение всего времени работы программы. Это устраняет необходимость выделения памяти и, следовательно, позволяет избежать утечек.

**Сборка мусора в Java** — это процесс, с помощью которого программы Java автоматически управляют памятью. Java-программы компилируются в байт-код, который запускается на виртуальной машине Java (JVM).

Когда Java-программы выполняются на JVM, объекты создаются в куче, которая представляет собой часть памяти, выделенную для них.

Пока Java-приложение работает, в нем создаются и запускаются новые объекты. В конце концов некоторые объекты перестают быть нужны. **Можно сказать, что в любой момент времени память кучи состоит из двух типов объектов.**

* Живые — эти объекты используются, на них ссылаются откуда-то еще.
* Мертвые — эти объекты больше нигде не используются, ссылок на них нет.

Сборщик мусора находит эти неиспользуемые объекты и удаляет их, чтобы освободить память.

<https://vc.ru/dev/397062-sborka-musora-v-java-chto-eto-takoe-i-kak-rabotaet-v-jvm> !!!!!!!!!!!!!!!!

<https://habr.com/ru/company/otus/blog/553996/> !!!!!

<https://www.examclouds.com/ru/java/java-core-russian/metod-finalize>

<https://visualvm.github.io/index.html> !!!!

[**https://www.examclouds.com/ru/java/java-core-russian/struktura-pamyati**](https://www.examclouds.com/ru/java/java-core-russian/struktura-pamyati) **!!!**

## 8.5 Пакеты Java

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

<https://docs.oracle.com/en/java/javase/13/docs/api/allpackages-index.html>

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. **Принадлежность к пакету позволяет гарантировать однозначность имен**.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета: **package Название\_пакета;** .

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом. А путь к файлам внутри проекта соответствует названию пакета этих файлов. Например, если классы принадлежат пакету mypack, то эти классы помещаются в проекте в папку mypack.

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

### 8.5.1 Создание пакета в IntelliJ Idea

<https://russianblogs.com/article/2337997667/>

<https://mycomp.su/obzory/intellij-idea-skachat.html>

Например, создадим в папке для исходных файлов каталог study. В нем создадим файл Program.java со следующим кодом:

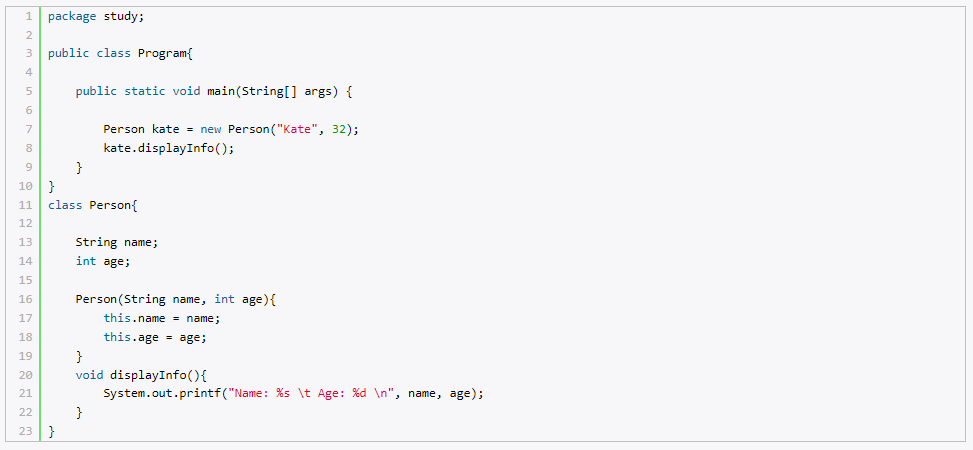


Рисунок 8.15.

<https://metanit.com/java/tutorial/3.2.php>

<https://www.bestprog.net/ru/2018/09/20/packages-using-packages-in-java-the-import-and-package-directives-compiled-modules-java-intermediate-class-files-project-structure-using-standard-java-libraries_ru/>