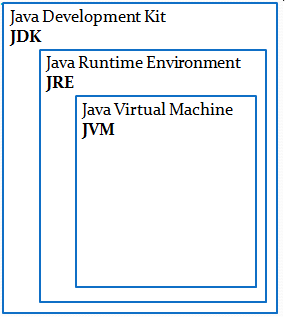
1. **Чем различаются JRE, JVM и JDK?**



**JVM** (Java Virtual Machine) - виртуальная машина Java - основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM обеспечивает платформо-независимый способ выполнения кода. Программисты могут писать код не задумываясь, как и где он будет выполняться.

**JRE** (Java Runtime Environment) - минимальная реализация виртуальной машины, необходимая для исполнения Java приложений, без компилятора и других средств разработки. Состоит из виртуальной машины и библиотек Java классов.

**JDK** (Java Development Kit) - комплект разработчика приложений на языке Java, включающий в себя компилятор, стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему JRE.

## Какие существуют модификаторы доступа?

**private** (приватный): члены класса доступны только внутри класса. Для обозначения используется служебное слово private.

**default**, package-private, package level (доступ на уровне пакета): видимость класса/членов класса только внутри пакета. Является модификатором доступа по умолчанию - специальное обозначение не требуется.

**protected** (защищённый): члены класса доступны внутри пакета и в наследниках. Для обозначения используется служебное слово protected.

**public** (публичный): класс/члены класса доступны всем. Для обозначения используется служебное слово public.

Последовательность модификаторов по возрастанию уровня закрытости: public, protected, default, private.

Во время наследования возможно изменения модификаторов доступа в сторону большей видимости (для поддержания соответствия принципу подстановки Барбары Лисков).

Следующая таблица суммирует, к каким конструкциям Java может применяться каждый модификатор:

private default protected public

Класс No Yes No Yes

Вложенный класс Yes Yes Yes Yes

Конструктор Yes Yes Yes Yes

Метод Yes Yes Yes Yes

Поля Yes Yes Yes Yes

Private

Если метод или переменная помечены как private (им назначен модификатор частного доступа), то только код внутри того же класса может получить доступ к переменной или вызвать метод. Код внутри подклассов не может получить доступ к переменной или методу, равно как и код из любого внешнего класса.

Классы не могут быть помечены таким модификатором. Подобная пометка класса будет означать, что никакой другой класс не сможет получить к нему доступ, а это означает, что вы вообще не сможете использовать его.

Вот пример назначения к полю:

public class Clock {

private long time = 0;

}

Переменная – long time была помечена как частная. Это означает, что переменная long time внутри класса Clock недоступна из кода вне него.

**Доступ к частным полям**

Поля часто объявляются частными для контроля доступа к ним из внешнего мира. В некоторых случаях поля являются действительно закрытыми, то есть они используются только внутри класса. В других случаях к полям можно получить доступ через методы доступа (например, геттеры и сеттеры). Вот пример:

public class Clock {

private long time = 0;

public long getTime() {

return this.time;

}

public void setTime(long theTime) {

this.time = theTime;

}

}

В приведенном выше примере два метода getTime() и setTime() могут получить доступ к переменной long time. Два метода объявлены как публичные, то есть они могут быть вызваны из кода в любом месте вашего приложения.

**Частные конструкторы**

Если конструктору в классе назначен частный модификатор, это означает, что конструктор не может быть вызван из-за пределов класса. Закрытый конструктор все еще может вызываться из других конструкторов или из статических методов того же класса:

public class Clock {

private long time = 0;

private Clock(long time) {

this.time = time;

}

public Clock(long time, long timeOffset) {

this(time);

this.time += timeOffset;

}

public static Clock newClock() {

return new Clock(System.currentTimeMillis());

}

}

Эта версия класса Clock содержит приватный конструктор и публичный конструктор. Закрытый конструктор вызывается из публичного (оператор this();). Он также вызывается из статического метода newClock().

Приведенный выше пример служит только для того, чтобы показать вам, что закрытый конструктор можно вызывать из открытых  и из статических методов внутри одного и того же класса.

Default

Объявляется без записи модификатора доступа вообще. Означает, что код внутри самого класса, а также код внутри классов в том же пакете, что и этот класс, могут обращаться к классу, полю, конструктору или методу, которому назначен такой модификатор. Поэтому эту разновидность также иногда называют модификатором доступа к пакету.

Подклассы не могут получить доступ к методам и переменным-членам (полям) в суперклассе, если эти методы и поля помечены этим модификатором, если только подкласс не находится в том же пакете, что и суперкласс.

Вот пример:

public class Clock {

long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

Поле времени в классе Clock не имеет модификатора, что означает, что ему неявно назначен по умолчанию. Следовательно, класс ClockReader может считывать переменную-член time объекта Clock при условии, что ClockReader и Clock находятся в одном и том же пакете Java.

Protected

Обеспечивает тот же доступ, что и модификатор по умолчанию, кроме того, подклассы могут обращаться к защищенным методам и переменным (полям) членов суперкласса. Это верно, даже если подкласс не находится в том же пакете, что и суперкласс.

Вот пример:

public class Clock {

protected long time = 0; // time in milliseconds

}

public class SmartClock() extends Clock{

public long getTimeInSeconds() {

return this.time / 1000;

}

}

В вышеприведенном примере у подкласса SmartClock есть метод getTimeInSeconds(), который обращается к переменной времени Clock суперкласса. Это возможно, даже если Clock и SmartClock не находятся в одном и том же пакете, поскольку поле времени помечено соответствующим модификатором.

Public

Общедоступный модификатор означает, что весь код может получить доступ к классу, полю, конструктору или методу независимо от того, где находится код доступа (в другом классе и в другом пакете):

public class Clock {

public long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

Поле времени в классе Clock помечено общедоступным модификатором. Поэтому класс ClockReader может обращаться к полю времени в Clock независимо от того, в каком пакете находится ClockReader.

**Доступ к классу**

Важно помнить, что модификатор, назначенный классу, имеет приоритет над любыми модификаторами, назначенными полям, конструкторам и методам этого класса. Если класс помечен модификатором по умолчанию, то никакой другой класс вне того же пакета не может получить доступ к этому классу, включая его конструкторы, поля и методы. Это не поможет, если вы объявите эти поля общедоступными.

Частные и защищенные модификаторы не могут быть назначены классу. Только для конструкторов, методов и полей внутри классов. Классам могут быть назначены только по умолчанию (пакет) и общедоступный.

**К интерфейсу**

Java-интерфейсы предназначены для указания общедоступных полей и методов в классах, которые они реализуют. Поэтому вы не можете использовать модификаторы частного и защищенного доступа в них. Поля и методы в интерфейсах неявно объявляются общедоступными, если вы пропустите модификатор, поэтому вы также не можете использовать модификатор по умолчанию (без него).

**Модификаторы и наследование**

Когда вы создаете подкласс некоторого класса, у методов в подклассе не может быть назначено им менее доступных модификаторов, чем в суперклассе.

Например, если метод в суперклассе является общедоступным, то он также должен быть таким в подклассе, в случае, если подкласс переопределяет метод. Если метод в суперклассе защищен, то он должен быть защищенным или общедоступным в подклассе.

Хотя не разрешено уменьшать доступность переопределенного метода, разрешено расширять доступность переопределенного метода. Например, если методу назначен модификатор по умолчанию в суперклассе, то разрешается назначать переопределенный метод в подклассе модификатору открытого доступа.

Источник: https://java-blog.ru/osnovy/modifikatory-dostupa-java

## О чем говорит ключевое слово final?

Модификатор final может применяться к переменным, параметрам методов, полям и методам класса или самим классам.

* Класс не может иметь наследников;
* Метод не может быть переопределен в классах наследниках;
* Поле не может изменить свое значение после инициализации;
* Параметры методов не могут изменять своё значение внутри метода;
* Локальные переменные не могут быть изменены после присвоения им значения.

1. **Какими значениями инициализируются переменные по умолчанию?**

* Числа инициализируются 0 или 0.0;
* char — \u0000;
* boolean — false;
* Объекты (в том числе String) — null.

## Что вы знаете о функции main()?

Метод main() — точка входа в программу. В приложении может быть несколько таких методов. Если метод отсутствует, то компиляция возможна, но при запуске будет получена ошибка `Error: Main method not found`.

Это точка входа в класс. Виртуальная машина Java выполняет определенный класс через main () определенного класса. Когда в классе есть метод main (), выполнение команды «имя класса java» запускает виртуальную машину для выполнения класса. метод main ().

**Синтаксический анализ основной функции main ()**

1. Поскольку виртуальная машина Java должна вызывать метод main () класса, права доступа этого метода должны быть открытыми.
2. Поскольку виртуальной машине Java не нужно создавать объект при выполнении метода main (), метод должен быть статическим, а метод main () - статическим, поэтому JVM не создает объект экземпляра класса, в котором находится метод main, при выполнении метода main. В методе main () мы не можем напрямую обращаться к нестатическим членам в классе. Мы должны создать экземпляр объекта класса, прежде чем мы сможем получить доступ к нестатическим членам в классе через этот объект.
3. Этот метод получает параметр массива типа String, в котором хранятся параметры, переданные классу выполнения при выполнении команды java. Почему входной параметр является массивом символов? Поскольку строковый массив может принимать несколько параметров, и любой тип данных может быть преобразован в строку.

# **Всё о методе main()**

Метод main() отличается от всех остальных методов тем, что является, как правило, точкой входа в программу. Этот метод вызывается виртуальной машиной Java. Как только заканчивается выполнение метода main(), так сразу же завершается, тем самым, работа самой программы.

Метод main(), так и любой другой метод, должен быть обязательно вложен в класс. После компиляции класс, содержащий метод main(), запускается на выполнение командой

java ПолноеИмяКласса

Эта команда приводит к выполнению метода main(), вложенного в данный класс. Оговоримся, что после имени класса могут следовать параметры командной строки, которые будут обсуждаться ниже.

Метод main() должен объявляться в классе следующим образом:

1. **public** **static** **void** main(String[] args)
2. {
3. ... ... ... ...//Код метода
4. }

Ключевое слово public означает, что метод main() доступен везде, где доступен содержащий его класс. Ключевое слово static означает, что метод является статическим, т. е. не требует для своего вызова наличие экземпляра класса. Ключевое слово void означает, что метод не возвращает никакого значения. Все эти три слова обязательно должны присутствовать в описании метода.

Запись (String[] args) после имени метода представляет собой список его параметров. Имеется лишь один параметр args, содержащий ссылку на массив строк. Этот массив формируется из строки, переданной классу, содержащему метод main(), при запуске этого класса на выполнение командой java.

Если строка передана не была, то массив args содержит 0 элементов. В противном случае строка разбивается на части, которые в строке отделёны друг от друга пробельными символами. Эти части будем в дальнейшем называть аргументами командной строки. Данными аргументами и заполняется массив args.

Вместо имени массива args можно использовать любое другое имя. Например, следующее описание main() вполне корректно.

1. **public** **static** **void** main(String[] strings)
2. {
3. ... ... ... ...//Код метода
4. }

Что касается типа и количества параметров метода main(), то они изменению не подлежат.

Заметим, что все требования к методу main(), приведённые выше, необходимо соблюдать только в случае, если мы хотим, чтобы данный метод мог вызываться виртуальной машиной Java.

Рассмотрим пример, в котором на консоль построчно выводятся все аргументы командной строки, переданные программе при её запуске. Под программой в данном случае мы будем понимать класс, содержащий метод main(). Этот класс мы назовём MainTest. Вот его код:

1. **class** MainTest
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **for** (String s : args)
6. System.out.println(s);
7. }
8. }

Здесь в методе main() содержится цикл, поочерёдно перебирающий элементы массива args. Ссылки на элементы массива поочерёдно присваиваются переменной s и выводятся на консоль методом println() объекта System.out. Этот метод после каждого вывода осуществляет перевод строки.

Сохраним код класса MainTest в файле с именем MainTest.java и скомпилируем этот файл командой

javac MainTest.java

В результате компиляции получаем файл MainTest.class. Запускаем его на выполнение с несколькими аргументами командной строки:

java MainTest Это всего лишь проверка!

Если нет проблем с отображением кириллицы, то в результате выполнения команды на консоль будет выведено:

Это  
всего  
лишь  
проверка!

Метод main() может вызываться не только виртуальной машиной Java, но и любым другим методом. Например, main() может быть рекурсивным, т. е. может вызывать сам себя. В этом отношении main() ничем не отличается от остальных методов.

В следующем примере (весьма вычурном) метод main() рекурсивен. При каждом вызове метода, за исключением последнего, на печать выводится один аргумент командной строки.

1. **class** RexMain
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **if** (args.length != 0)
6. {
7. System.out.println(args[0]);
8. main(java.util.Arrays.copyOfRange(args, 1, args.length));
9. }
10. }
11. }

Здесь в методе main() выясняется, не является ли пустым массив args. Если нет, то на печать выводится первый его элемент. После этого метод main() вызывается рекурсивно. В качестве параметра ему передаётся новый массив, отличающийся от старого отсутствием первого элемента. Если же массив args пуст, то работа main() на этом завершается.

Результат выполнение класса RexMain совпадает с результатом выполнения класса MainTest.

Метод main(), так же как и любой другой метод, можно перегружать, т. е. создавать в том же классе одноимённые методы, отличающиеся от исходного списком параметров. При запуске на выполнение класса, содержащего несколько методов main(), виртуальная машина Java выбирает нужный (если он имеется), ориентируясь на список параметров.

Несколько разных классов, входящих в одну программу, могут содержать методы main(). Ничто не мешает запускать на выполнение любой из этих классов.

## Новичкам и не только

В самом начале статьи было сказано, что метод main() является точкой входа в программу с оговоркой “как правило”. В этом разделе мы установим, что, вообще говоря, выполнение программы необязательно начинается с вызова метода main().

В Интернете на форумах, посвящённых Java, я нередко встречал любопытную задачку: написать класс, метод main() которого имеет пустое тело, выводящий на консоль надпись “Hello world!”. Полагаю, что эта задача известна многим из тех, кто интересуется Java.

В основе решения задачи лежит использование статического блока, представляющего собой программный код, заключённый в фигурные скобки, которому предшествует ключевое слово static. Статический блок вкладывается в класс и исполняется при первом же обращении к классу, до выполнения любых методов класса, в том числе, и метода main().

Статический блок имеет некоторое сходство со статическим методом. Отличается он от последнего тем, что не имеет имени, не принимает параметров, не возвращает значения (а значит, не может содержать инструкции return) и не вызывается явно. Так же как и статический метод, статический блок может содержать обращения к статическим полям и методам класса.

Статические блоки используются редко. Как правило, в их задачи входит инициализация статических полей класса.

Ну а мы поместим в статический блок инструкцию вывода на консоль строки “Hello world!”:

1. **class** HelloWorld
2. {
3. **static**
4. {
5. System.out.println("Hello world!");
6. }
7. **public** **static** **void** main(String[] args)
8. {
9. }
10. }

Можно скомпилировать класс, запустить на выполнение и удостовериться в том, что с поставленной задачей он вполне успешно справляется.

Тот факт, что статический блок выполняется до метода main(), легко проверить, скомпилировав и выполнив следующий класс:

1. **class** HelloHello
2. {
3. **static**
4. {
5. System.out.println("Привет от static-блока!");
6. }
7. **public** **static** **void** main(String[] args)
8. {
9. System.out.println("Привет от метода main()!");
10. }
11. }

На консоль будет выведено:

Привет от static-блока!  
Привет от метода main()!

Заметим, что задачу, поставленную в начале статьи, можно решить, не прибегая к использованию статического блока. Для этого достаточно написать статический метод, печатающий строку “Hello world!” и возвращающий значение какого-либо типа, например, типа int, после чего создать статическое поле того же типа, инициализирующееся посредством вызова данного метода.

Дело в том, что инициализация статических полей происходит до выполнения любых статических методов, (и даже до выполнения статического блока). Исключением из этого правила является вызов статических методов в процессе инициализации. Поэтому метод, участвующий в инициализации, выполнится раньше main().

Вот код, демонстрирующий данный подход:

1. **class** WithoutStatic
2. {
3. **private** **static** **int** print()
4. {
5. System.out.println("Hello world!");
6. **return** 0;
7. }
8. **private** **static** **int** i = print();
9. **public** **static** **void** main(String[] args)
10. {
11. }
12. }

Выполнение класса WithoutStatic приводит к тому же выводу на консоль, что и выполнение класса HelloWorld.

Резонно задаться вопросом: а можно ли вообще обойтись без метода main(), пусть даже имеющего пустое тело? Для ответа на вопрос изменим класс HelloWorld, удалив из него main():

1. **class** WithoutMain
2. {
3. **static**
4. {
5. System.out.println("Hello world!");
6. }
7. }

И вот тут начинается самое интересное! Я компилировал и запускал этот класс с использованием комплектов разработки на языке Java (JDK) трёх разных версий: jdk1.6\_021, jdk1.6\_024 и jdk1.7.0\_01. Во всех трёх случаях код компилировался без проблем. А вот с запуском класса проблемы возникали.

Для начала, оговорюсь, что каждый файл с расширением class запускался под управлением “своей” виртуальной машины Java, т. е. входящей в тот JDK, посредством которого этот файл был получен в результате компиляции.

Итак, в последних двух случаях на экран выводилась надпись:

Error: Main method not found in class WithoutMain, please define the main method as: public static void main(String[] args)

Таким образом, не выполнялся даже код, входящий в статический блок. А вот в первом случае на консоль выводилось следующее:

HelloWorld!  
Exception in thread "main" java.lang.NoSuchMethodError: main

На этот раз виртуальная машина, всё же, выполняет статический блок и только после этого, не обнаружив метода main(), спохватывается и выдаёт по данному поводу ругательство. Можно ли избежать сообщения об исключении? Оказывается, можно!

Для этого достаточно после инструкции печати вызвать статический метод exit() класса System. Данный метод прерывает работу виртуальной машины и возвращает родительскому процессу (как правило, операционной системе) целое значение, переданное методу в качестве аргумента. Считается, что ненулевое значение свидетельствует об аварийном прерывании.

Итак, ниже приведён код работоспособной программы, не содержащей метода main().

1. **class** DownWithMain
2. {
3. **static**
4. {
5. System.out.println("Долой main()!");
6. System.out.exit(0);
7. }
8. }

Выполнение программы приводит к следующему выводу на консоль:

Долой main()!

Как мы выяснили, программа вполне может обходиться без метода main(), правда, с оговоркой, что запущена на выполнение она будет под управлением виртуальной машины достаточно старой версии. На этом всё. Спасибо за внимание!

## Какие логические операции и операторы вы знаете?

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| & | Логическая операция И (AND) или конъюнкция |
| | | Логическая операция ИЛИ (OR) или дизъюнкция |
| ^ | Логическая операция исключающее ИЛИ (XOR) |
| ! | Логическая унарная операция НЕ (NOT) |
| || | Укороченная логическая операция ИЛИ (short-circuit) |
| && | Укороченная логическая операция И (short-circuit) |
| == | Равенство |
| != | Неравенство |
| &= | Логическая операция И с присваиванием |
| |= | Логическая операция ИЛИ с присваиванием |
| ^= | Логическая операция исключающее ИЛИ с присваиванием |

## 1.. OR, AND, XOR, NOT.

OR (|) - результат будет true*,* если хотя бы одно значение равно true*.*

AND (&) - результат будет true, только если и A, и B равны true.

XOR (^) - результат будет true, только если или A равно true, или В равно true.

NOT (!) - инвертирование значения. Если значение было true, то станет false, и наоборот.

public class BooleanLogic1 {

public static void main(String[] args) {

boolean a = true;

boolean b = false;

boolean c = a | b;

boolean d = a & b;

boolean e = a ^ b;

boolean f = (!a & b) | (a & !b);

boolean g = !a;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("a | b = " + c);

System.out.println("a & b = " + d);

System.out.println("a ^ b = " + e);

System.out.println("(!a & b) | (a & !b) = " + f);

System.out.println("!a = " + g);

}

}

## 2, Укороченные логические операторы (short-circuit). && , ||

Чаще всего в языке Java используются так называемые укороченные логические операторы (short-circuit):

||   -  Укороченный логический оператор ИЛИ

&&  - Укороченный логический оператор И

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, то есть если левый операнд конъюнкции имеет значение true, или левый операнд дизъюнкции имеет значение false.

В формальной спецификации языка Java укороченные логические операции называются условными.

В следующем примере правый операнд логического выражения вычисляться не будет, так как условие d!=0 не выполняется и нет смысла дальше вычислять это выражение:

public class BooleanLogic2 {

public static void main(String[] args) {

int d = 0;

int num = 10;

if (d != 0 && num / d > 10) {

System.out.println("num = " + num);

}

}

}

Если же логическое выражение переписать как d != 0 & num / d > 10, то правое выражение вычисляться будет. И в этом случае программа выбросит исключение времени выполнения с сообщением - на ноль делить нельзя.

Если нам надо сравнить возможный диапазон значений для какой-то переменной, например - a<x<b, такое условие разбивается на два: a<x &&x<b:

public class BooleanLogic3 {

public static void main(String[] args) {

int a = 1;

int b = 2;

int x = 3;

System.out.print(a < x && x < b);

// System.out.print(a < x < b);//Ошибка компиляции

}

}

## 3. Операции ==, !=.

Здесь все просто - чтобы сравнить два значения типа boolean, можно использовать знаки == (проверка на равенство) и != (проверка на неравенство):

public class BooleanLogic4 {

public static void main(String[] args) {

boolean b1 = true;

boolean b2 = false;

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

System.out.println(b1 != b2);

## 4. Операции с присваиванием.

Также существуют операции с присваиванием для AND, OR, XOR. Посмотрим пример:

public class BooleanLogic5 {

public static void main(String[] args) {

boolean b1 = true;

boolean b2 = true;

b1 &= b2;//равносильно b1 = b1 & b2;

System.out.println(b1);

b1 |= b2; //равносильно b1 = b1 | b2;

System.out.println(b1);

b1 ^= b2; //равносильно b1 = b1 ^ b2;

System.out.println(b1);

}

}

## 7 Что такое тернарный оператор выбора?

Тернарный условный оператор ?: - оператор, которым можно заменить некоторые конструкции операторов if-then-else.

Выражение записывается в следующей форме:

условие ? выражение1 : выражение2

Если условие выполняется, то вычисляется выражение1 и его результат становится результатом выполнения всего оператора. Если же условие равно false, то вычисляется выражение2 и его значение становится результатом работы оператора. Оба операнда выражение1 и выражение2 должны возвращать значение одинакового (или совместимого) типа.

**8 Какие побитовые операции вы знаете?**

* ~: Побитовый унарный оператор NOT;
* &: Побитовый AND;
* &=: Побитовый AND с присваиванием;
* |: Побитовый OR;
* |=: Побитовый OR с присваиванием;
* ^: Побитовый исключающее XOR;
* ^=: Побитовый исключающее XOR с присваиванием;
* >>: Сдвиг вправо (деление на 2 в степени сдвига);
* >>=: Сдвиг вправо с присваиванием;
* >>>: Сдвиг вправо без учёта знака;
* >>>=: Сдвиг вправо без учёта знака с присваиванием;
* <<: Сдвиг влево (умножение на 2 в степени сдвига);
* <<=: Сдвиг влево с присваиванием.

## 9 Где и для чего используется модификатор abstract?

Класс, помеченный модификатором abstract, называется абстрактным классом. Такие классы могут выступать только предками для других классов. Создавать экземпляры самого абстрактного класса не разрешается. При этом наследниками абстрактного класса могут быть как другие абстрактные классы, так и классы, допускающие создание объектов.

Метод, помеченный ключевым словом abstract - абстрактный метод, т.е. метод, который не имеет реализации. Если в классе присутствует хотя бы один абстрактный метод, то весь класс должен быть объявлен абстрактным.

Использование абстрактных классов и методов позволяет описать некий шаблон объекта, который должен быть реализован в других классах. В них же самих описывается лишь некое общее для всех потомков поведение.

**10 Чем абстрактный класс отличается от интерфейса? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?**

* В Java класс может одновременно реализовать несколько интерфейсов, но наследоваться только от одного класса.
* Абстрактные классы используются только тогда, когда присутствует тип отношений «is a» (является). Интерфейсы могут реализоваться классами, которые не связаны друг с другом.
* Абстрактный класс - средство, позволяющее избежать написания повторяющегося кода, инструмент для частичной реализации поведения. Интерфейс - это средство выражения семантики класса, контракт, описывающий возможности. Все методы интерфейса неявно объявляются как public abstract или (начиная с Java 8) default - методами с реализацией по-умолчанию, а поля - public static final.
* Интерфейсы позволяют создавать структуры типов без иерархии.
* Наследуясь от абстрактного, класс «растворяет» собственную индивидуальность. Реализуя интерфейс, он расширяет собственную функциональность.

Абстрактные классы содержат частичную реализацию, которая дополняется или расширяется в подклассах. При этом все подклассы схожи между собой в части реализации, унаследованной от абстрактного класса, и отличаются лишь в части собственной реализации абстрактных методов родителя. Поэтому абстрактные классы применяются в случае построения иерархии однотипных, очень похожих друг на друга классов. В этом случае наследование от абстрактного класса, реализующего поведение объекта по умолчанию может быть полезно, так как позволяет избежать написания повторяющегося кода. Во всех остальных случаях лучше использовать интерфейсы.

* **Интерфейс описывает только поведение. У него нет состояния. А у абстрактного класса состояние есть: он описывает и то, и другое.**

Интерфейс только описывает поведение. Соответственно, мы не сможем реализовать внутри интерфейса геттеры и сеттеры. Такова природа интерфейса: он нужен для работы с поведением, а не состоянием.

В Java8 появились дефолтные методы интерфейсов, у которых есть реализация. О них ты уже знаешь, поэтому повторяться не будем.

* **Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.**
* **Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса.**

Множественного наследования в Java нет, а множественная реализация есть. Отчасти этот пункт вытекает из предыдущего: интерфейс связывает между собой множество разных классов, у которых часто нет ничего общего, а абстрактный класс создается для группы очень близких друг другу классов. Поэтому логично, что наследоваться можно только от одного такого класса. Абстрактный класс описывает отношения «is a».

## 11 Почему в некоторых интерфейсах вообще не определяют методов?

Это так называемые маркерные интерфейсы. Они просто указывают что класс относится к определенному типу. Примером может послужить интерфейс Clonable, который указывает на то, что класс поддерживает механизм клонирования.

Можно при помощи интерфейса-маркера(интерф1) сделать расширение(extends) другого интерфейса((интерф2) того, функциональность которого нам будет нужна!) и при этом в классе, который унаследует поведение (интерф1) со всеми его методами, мы КАК БОНУС, получаем ВСЕ МЕТОДЫ (интерф2), МЕТОДЫ КОТОРОГО НАМ НЕ НУЖНО ОБЯЗАТЕЛЬНО ПЕРЕОПРЕДЕЛЯТЬ !!!! ЭТОГО НЕ БУДЕТ ТРЕБОВАТЬСЯ!! Просто, когда нам надо, тогда и будем использовать необходимые нам методы.

## 12 Почему нельзя объявить метод интерфейса с модификатором final?

В случае интерфейсов указание модификатора final бессмысленно, т.к. все методы интерфейсов неявно объявляются как абстрактные, т.е. их невозможно выполнить, не реализовав где-то еще, а этого нельзя будет сделать, если у метода идентификатор final.

**13 Может ли объект получить доступ к члену класса, объявленному как private? Если да, то каким образом?**

* Внутри класса доступ к приватной переменной открыт без ограничений;
* Вложенный класс имеет полный доступ ко всем (в том числе и приватным) членам содержащего его класса;
* Доступ к приватным переменным извне может быть организован через отличные от приватных методов, которые предоставлены разработчиком класса. Например: getX() и setX().
* Через механизм рефлексии (Reflection API):

class Victim {

private int field = 42;

}

//...

Victim victim = new Victim();

Field field = Victim.class.getDeclaredField("field");

field.setAccessible(true);

int fieldValue = (int) field.get(victim);

## 14 Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?

Поля объекта инициализируются в следующем порядке:

1. Статические поля класса Parent;
2. Статический блок инициализации класса Parent;
3. Статические поля класса Сhild;
4. Статический блок инициализации класса Child;
5. Нестатические поля класса Parent;
6. Нестатический блок инициализации класса Parent;
7. Конструктор класса Parent;
8. Нестатические поля класса Сhild;
9. Нестатический блок инициализации класса Сhild;
10. Конструктор класса Сhild.

#### **Инициализация с помощью блоков**

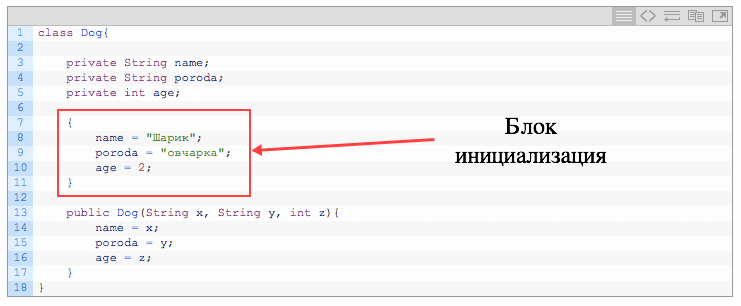
Представим, что мы хотим задать базовые значения переменных. Можно сделать это так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Dog{    private String name = "Шарик";  private String poroda = "овчарка";  private int age = 2;    public Dog(String x, String y, int z){        name = x;        poroda = y;        age = z;  }    } |

А можно использовать блок инициализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Dog{        private String name;      private String poroda;      private int age;        {          name = "Шарик";          poroda = "овчарка";          age = 2;      }        public Dog(String x, String y, int z){          name = x;          poroda = y;          age = z;      }  } |
|  |  |

Видите - две фигурные скобки, между которыми мы присвоили новые значения переменным:



Выглядит лучше, правда?

Давайте посмотрим какие существуют блоки инициализации.

#### **Виды блоков инициализации**

Существует всего два типа блоков:

* **нестатический** (*instance initializer*)
* **статический**(*class initializer*)

В предыдущем примере Вы видели первый тип - нестатический блок инициализации. Его общую форму можно показать так:

Как можно догадаться по названию:

* **статический блок** используется для инициализации **статических переменных**, а "обычный"  - **для всех остальных**.

#### **Зачем используются блоки инициализации**

Красивый и читабельный код - это конечно хорошо. Но зачем они нужны? Есть ли другие преимущества у блоков инициализации? **Есть.** И это - **больший функционал**.

Внутри блоков инициализации мы можем не только присваивать значения. Это **как метод** - тут можно писать любые команды. Например, вывод в консоль:

|  |  |
| --- | --- |
|  | class Test{  **static {**  **System.out.println("Это статический блок!");**  **}**    {              System.out.println("Это нестатический блок!");  }   } |

Для того, чтобы Вы могли протестировать у себя на компьютере, возьмем такой:

class Dog{

    private String name;

    private String poroda;

    private int age;

    {

        System.out.println("Это нестатический блок инициализации!");

    public Dog(String x, String y, int z){

        name = x;

        poroda = y;

        age = z;

    }

}

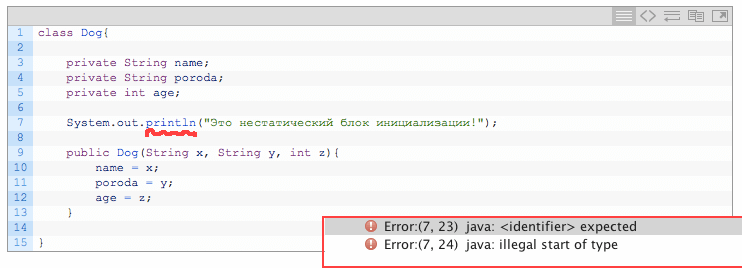
Теперь в нашем блоке инициализации есть только **System.out.println("Это нестатический блок инициализации!")**. Запустим его со следующим **main**-ом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Test {        public static void main(String[] args) {          Dog d = new Dog("Шарик", "овчарка", 2);      }  } |

И получим в консоли:

https://vertex-academy.com/tutorials/wp-content/uploads/2018/03/Capture-d%E2%80%99%C3%A9cran-2018-03-05-%C3%A0-14.11.14.png

Как видите, работает. Если бы мы просто написали **System.out.println**, без блока инициализации, программа бы не запустилась:



Как видите, такое можно делать только внутри блока инициализации. Это особенно полезно если нам нужно производить какие-нибудь вычисления перед инициализацией переменных.

#### **Подводим итог:**

* блоки инициализации используются для инициализации переменных внутри класса;
* есть два типа блоков инициализации - статический и нестатический;
* статический блок инициализации используется для статических переменных, нестатический - для всех остальных;
* блоки инициализации делают код читабельнее, и позволяют вызывать любые методы.

## 15 Для чего в Java используются статические блоки инициализации?

Статические блоки инициализация используются для выполнения кода, который должен выполняться один раз при инициализации класса загрузчиком классов, в момент, предшествующий созданию объектов этого класса при помощи конструктора. Такой блок (в отличие от нестатических, принадлежащих конкретном объекту класса) принадлежит только самому классу (объекту метакласса Class).

**16 К каким конструкциям Java применим модификатор static?**

* полям;
* методам;
* вложенным классам;
* членам секции import.

## 17 Как получить доступ к переопределенным методам родительского класса?

С помощью ключевого слова super мы можем обратиться к любому члену родительского класса - методу или полю, если они не определены с модификатором private.

super.method();

(1) **Composition vs. Aggregation:** https://holub.com/uml/#composition

Ни «агрегация», ни «композиция» действительно не имеют прямых аналогов во многих языках (Java, например).

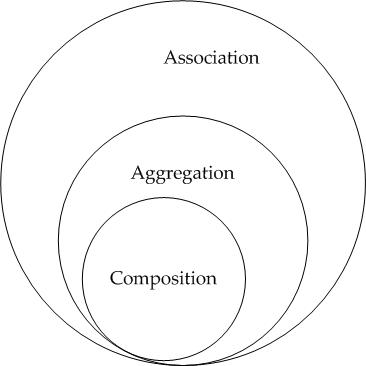
«Совокупность» представляет собой целое, состоящее из различных частей; Таким образом, Комитет представляет собой совокупность его членов. Собрание представляет собой совокупность Повестки дня, Комнаты и Участников. Во время реализации это отношение не является сдерживанием. (Собрание не содержит комнаты.) Точно так же части агрегата могут выполнять другие действия в другом месте программы, поэтому на них могут ссылаться несколько объектов. Другими словами, нет никакой разницы на уровне реализации между агрегацией и простым отношением «использует» (строка «ассоциация», на которой вообще нет ромбов). В обоих случаях объект имеет ссылки на другие объекты. Хотя нет никакой разницы в реализации, определенно стоит зафиксировать взаимосвязь в UML, потому что это поможет вам лучше понять модель предметной области, и потому что есть тонкие проблемы с реализацией. Я мог бы допустить более тесные отношения связи в агрегации, чем, например, с простым «использованием».

Композиция включает в себя еще более тесную связь, чем агрегация, и определенно предполагает включение. Основное требование заключается в том, что если класс объектов (назовем его «контейнером») состоит из других объектов (назовем их «элементами»), то элементы будут возникать, а также уничтожаться как побочный эффект создания. или уничтожение контейнера. Было бы редко, если бы элемент не был объявлен как private. Примером может быть имя и адрес клиента. Клиент без имени или адреса — бесполезная вещь. Точно так же, когда Заказчик уничтожен, нет смысла сохранять имя и адрес. (Сравните эту ситуацию с агрегацией, когда уничтожение комитета не должно вызывать уничтожение членов --- они могут быть членами других комитетов).

С точки зрения реализации, элементы отношения композиции обычно создаются конструктором или инициализатором в объявлении поля, но в Java нет деструктора, поэтому невозможно гарантировать, что элементы будут уничтожены вместе с контейнером. В C++ элемент был бы объектом (а не ссылкой или указателем), объявленным как поле в другом объекте, поэтому создание и уничтожение элемента было бы автоматическим. В Java такого механизма нет. Тем не менее, важно указать отношение включения в UML, потому что это отношение сообщает людям, занимающимся реализацией/тестированием, что ваше намерение состоит в том, чтобы элемент стал пригодным для сбора мусора (т. е. не должно быть никаких ссылок на него) при уничтожении контейнера).

Когда мы думаем об объектно-ориентированной природе, мы всегда думаем об объектах, классе (схемах объектов) и отношениях между ними. Объекты связаны и взаимодействуют друг с другом через методы. Другими словами, объект одного класса может использовать службы/методы, предоставляемые объектом другого класса. Такой тип отношений называется **ассоциацией.**.

**Агрегация и композиция являются подмножествами ассоциации** , что означает, что они являются конкретными случаями ассоциации.



* И в агрегации, и в композиции **объект одного класса «владеет» объектом другого класса** .
* Но есть тонкая разница. В **композиции** объект класса, который принадлежит объекту своего класса **, не может жить сам по себе** (также называемый «отношением смерти»). Он всегда будет жить как часть своего объекта-владельца, тогда как в **агрегации** зависимый объект является **автономным** и может существовать, даже если объект-владелец класса мертв.
* Таким образом, в композиции, если объект-владелец является сборщиком мусора, принадлежащий объект также будет, чего нельзя сказать о агрегации.

Смущенный?

**Пример композиции** : рассмотрим пример автомобиля и двигателя, который очень специфичен для этого автомобиля (это означает, что его нельзя использовать ни в каком другом автомобиле). Этот тип отношений между **Car** и классом **SpecificEngine** называется Composition. Объект класса Car не может существовать без объекта класса SpecificEngine, а объект SpecificEngine не имеет значения без класса Car. Проще говоря, класс Car исключительно «владеет» классом SpecificEngine.

**Пример агрегации** : Теперь рассмотрим класс **Car** и класс **Wheel** . Автомобилю нужен объект Wheel для работы. Это означает, что объект Car владеет объектом Wheel, но мы не можем сказать, что объект Wheel не имеет значения без объекта Car. Его можно очень хорошо использовать в велосипеде, грузовике или другом объекте автомобилей.

**Подводя итог -**

Подводя итог, ассоциация — это очень общий термин, используемый для обозначения того, когда класс использует функции, предоставляемые другим классом. Мы говорим, что это композиция, если один объект родительского класса владеет другим объектом дочернего класса, и этот объект дочернего класса не может осмысленно существовать без объекта родительского класса. Если это возможно, то это называется агрегацией.