**LESSON 1**

**JVM JRE JDK**

Java Virtual Machine (JVM) – это часть программного обеспечения Java, которая исполняет байт код, который был создан из .java файла компилятором Java (javac).

JVM может использоваться для выполнения байт кода, написанного на других языках. JVM можно рассматривать как самостоятельную платформу.

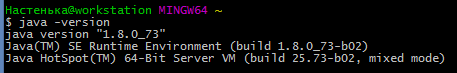
Java Virtual Machine специфицируется набором команд байт-кода, набором регистров, стеком, сборщиком мусора и пространством хранения методов. Виртуальная машина Java полностью стековая.

Java можно охарактеризовать следующим выражением «compile once, run anywhere» из-за того, что байт код может быть использован на многих платформах.

Java байт код – это машинно-независимый код, который генерирует java-компилятор, а выполняется java-интерпретатором

*Как проверить какая версия java стоит на пк?*

1. Через cmd: java -version



1. В меню Пуск -> Programs -> Java -> About java
2. Панель управления –> Программы –> Программы и компоненты
3. Панель управления –>Программы –> Java –> Вкадка General -> About

*English*

*A Java virtual machine (JVM), an implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform.*

JRE (Java Runtime Environment) – это среда выполнения для Java. Предназначена только для выполнения готовых приложений на Java, поэтому содержит только реализацию виртуальной машины (минимальную) и библиотеки java классов (стандартные библиотеки).

JRE используется в браузерах, умеющих выполнять программы java, операционных системах.

JRE является необходимым и достаточным условием выполнения java программ. Поэтому, скачав jre (например с oracle.com), можно начать работать с java.

JRE предназначена для пользователей. С её помощью можно запускать апплеты и приложения, написанные на java.

*English*

*Java Runtime Environment (JRE) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library.*

JDK (Java Development Kit) – специальный пакет разработчика приложений на языке Java, включающий в себя компилятор (javac), стандартные библиотеки классов, примеры, документацию, различные утилиты, JRE (исполнительную систему).

JDK предназначена для разработки.

*English*

*Java Development Kit (JDK) is a superset of a JRE and contains tools for Java programmers, e.g. a javac compiler.*

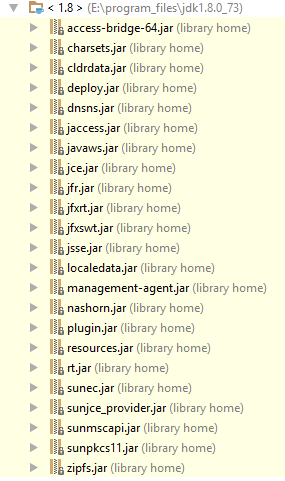
JVM -> JRE -> JDK

*Какие стандартные классы и библиотеки?*

Библиотеки классов предоставляют большую часть функциональных возможностей, обеспечиваемых средой Java.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Название jar** | **Расположение** | **Название пакетов** |
| 1 | Rt.jar | Jre/lib | 1. Java.lang ( включает классы для создания таких структур, как динамические массивы, стеки и словари. Есть классы для работы с генератором псевдослучайных чисел, для разбора строк на составляющие элементы, для работы с календарной датой и временем) 2. java.util ( включает классы, имеющие отношение к вводу и выводу данных через потоки. Заметим, что с использованием этих классов можно работать не только с потоками байт, но также и с потоками данных других типов, например числами int или текстовыми строками) 3. java.math (предназначенный для выполнения математических операций, таких как вычисление синуса, косинуса и тангенса ) 4. java.net (библиотека классов для работы в сети, содержит классы, с помощью которых можно работать с универсальными сетевыми адресами URL, передавать данные с использованием сокетов TCP и UDP, выполнять различные операции с адресами IP ) 5. java.awt ( для создания пользовательского интерфейса аплеты Java могут и должны использовать библиотеку классов java.awt. AWT - это сокращение от Abstract Window Toolkit (инструментарий для работы с абстрактными окнами) ) |
| 2 |  |  |  |
| 3 |  |  |  |
| 4 |  |  |  |

Rt.jar contains all the classes provided in the Java Runtime Environment. If you don't have it on your classpath you will not have access to any of those classes you need to use like java.lang.String or java.io.File.



*Сколько версий jdk может быть?*

Много.

*Если много версий, то как выбрать нужное для выполнения?*

1. Переопределить переменные среды окружения.
2. Через командную строку зайти в папку с нужной jre и там выполнить команды

*Что есть в bin ?*

Исполняемые файлы, такие как: java, javac, Javadoc, jar, jdb (helps you find and fix bugs in Java language programs).

*Могут ли параллельно быть запущены несколько jvm? Сколько? Как проверить состояние?*

Могут, сколько позволит память, ведь jvm – это просто исполняемая программа (java.exe). Проверить можно:

1. Через task manager
2. Использовать visual vm

*Когда jvm начинает работу и заканчивает?*

Jvm начинает работу с выполнением программы и заканчивает после её завершения.

*Как пощупать jvm?*

Jvm – это просто исполняемая программа (java.exe). Находится в jre -> bin -> java.exe . Так же используя visual vm

*Параметры java*

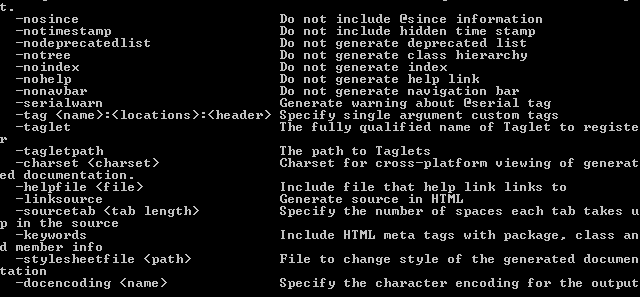
**

*Параметры javac*



*Параметры Javadoc*

**

**

*Кроссплатформенность.*

Для обеспечения кроссплатформенности сначала идёт компиляция программы в язык низкого уровня – байткод.

**Переменные среды окружения (environment variables)**

Переменные среды окружения используются для хранения текстовых строк пользователя и информации о настройках операционных систем.

|  |  |  |
| --- | --- | --- |
| PATH | Указывает путь поиска исполняемых файлов | E:\program\_files\jdk1.8.0\_73\bin |
| JAVA\_HOME | If install jdk you’ll be setting this variable | E:\program\_files\jdk1.8.0\_73 |
| JRE\_HOME | If install jre you’ll be setting this variable | E:\program\_files\Java\jre1.8.0\_73 |

Чтобы установить значение переменной, например TEMP : SET TEMP=C:\TEMP

**Байткод**

Java байткод – это машинно-независимый код, который генерирует java-компилятор, а выполняется java-интерпретатором.

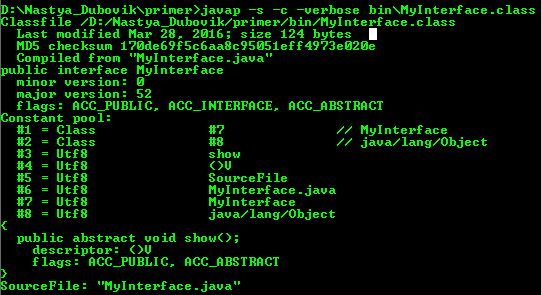
Каждому классу в java соответствует один откомпилированный файл. Это справедливо даже для подклассов или анонимным классов. Такой файл содержит информацию об имени класса, его родителе, список интерфейсов которые он реализует, перечисление его полей и методов. Важно отметить, что после компиляции информации, которая содержит директива import, теряется и все классы именуются теперь через полный путь. Например в место String будет записано java/lang/String.

Байт код стеко-ориентированный язык, похожий по своей структуре на ассемблер. Что бы произвести операции с данными их сначала нужно положить на стек. Мы хотим взять поле у объекта. Что бы это сделать нужно его положить в стек. В байт-коде нет имен переменных, у них есть номера. Нулевой номер у ссылки на текущий объект или у переменой this. Потом идут параметры исполняемого метода. Затем остальные переменные.

Class -> .class

Enum -> .class

Interface (не является классом, так как не является наследником Object) -> .class



|  |  |  |  |
| --- | --- | --- | --- |
| **byte offset** | **size** | **type or value** | **description** |
| 0 | 4 bytes | u1 = 0xCA hex | [magic number](https://en.wikipedia.org/wiki/Magic_number_(programming)) (CAFEBABE) used to identify file as conforming to the class file format |
| 1 | u1 = 0xFE hex |
| 2 | u1 = 0xBA hex |
| 3 | u1 = 0xBE hex |
| 4 | 2 bytes | u2 | minor version number of the class file format being used |
| 5 |
| 6 | 2 bytes | u2 | major version number of the class file format being used. Java SE 8 = 52 (0x34 hex), Java SE 7 = 51 (0x33 hex), Java SE 6.0 = 50 (0x32 hex), Java SE 5.0 = 49 (0x31 hex), JDK 1.4 = 48 (0x30 hex), JDK 1.3 = 47 (0x2F hex), JDK 1.2 = 46 (0x2E hex), JDK 1.1 = 45 (0x2D hex). For details of earlier version numbers see footnote 1 at [The JavaTM Virtual Machine Specification 2nd edition](https://docs.oracle.com/javase/specs/jvms/se6/html/VMSpecTOC.doc.html) |
| 7 |
| 8 | 2 bytes | u2 | constant pool count, number of entries in the following constant pool table. This count is at least one greater than the actual number of entries; see following discussion. |
| 9 |
| 10 | *cpsize*(variable) | table | constant pool table, an array of variable-sized constant pool entries, containing items such as literal numbers, strings, and references to classes or methods. Indexed starting at 1, containing (*constant pool count* - 1) number of entries in total (see note). |
| ... |
| ... |
| ... |
| 10+*cpsize* | 2 bytes | u2 | access flags, a bitmask |
| 11+*cpsize* |
| 12+*cpsize* | 2 bytes | u2 | identifies *this* class, index into the constant pool to a "Class"-type entry |
| 13+*cpsize* |
| 14+*cpsize* | 2 bytes | u2 | identifies *super* class, index into the constant pool to a "Class"-type entry |
| 15+*cpsize* |
| 16+*cpsize* | 2 bytes | u2 | interface count, number of entries in the following interface table |
| 17+*cpsize* |
| 18+*cpsize* | *isize*(variable) | table | interface table, an array of variable-sized interfaces |
| ... |
| ... |
| ... |
| 18+*cpsize*+*isize* | 2 bytes | u2 | field count, number of entries in the following field table |
| 19+*cpsize*+*isize* |
| 20+*cpsize*+*isize* | *fsize*(variable) | table | field table, variable length array of fields |
| ... |
| ... |
| ... |
| 20+*cpsize*+*isize*+*fsize* | 2 bytes | u2 | method count, number of entries in the following method table |
| 21+*cpsize*+*isize*+*fsize* |
| 22+*cpsize*+*isize*+*fsize* | *msize*(variable) | table | method table, variable length array of methods |
| ... |
| ... |
| ... |
| 22+*cpsize*+*isize*+*fsize*+*msize* | 2 bytes | u2 | attribute count, number of entries in the following attribute table |
| 23+*cpsize*+*isize*+*fsize*+*msize* |
| 24+*cpsize*+*isize*+*fsize*+*msize* | *asize*(variable) | table | attribute table, variable length array of attributes |
|  |

*Трансляция, компиляция, интерпретация?*

Трансляция (лат. translatio — перевод) - преобразование программы с одного языка программирования на другой, который понятен процессору или программе-интерпретатору.

Компиляция - трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера).

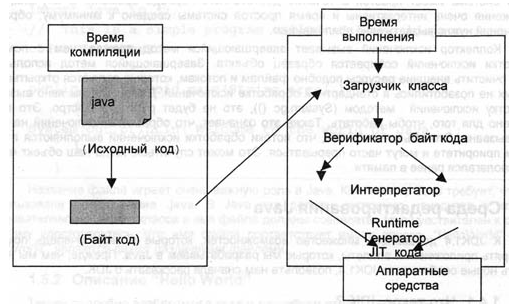
Интерпретация - пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется до её выполнения целиком).

*JIT компиляция*

JIT-компиляция ( Just-in-time compilation, компиляция «на лету»), динамическая компиляция (англ. dynamic translation) — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию.

JRE выполняет код, откомпилированный для JVM:

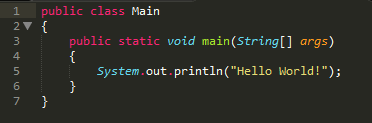
1. Загрузка .class файлов (выполняется с помощью “Загрузчика классов”, загрузчик класса делает проверку защиты, если файлы используются в сети)
2. Проверка байт-кода (выполняется “верификатором байт-кода”, верификатор байт-кода проверяет формат кода, преобразования типов объектов и проверяет нарушение прав доступа)
3. Выполнение кода (выполняется “интерпретатором во время выполнения”)
4. Интерпретатор выполняет байт-коды и делает запросы на используемое оборудование.



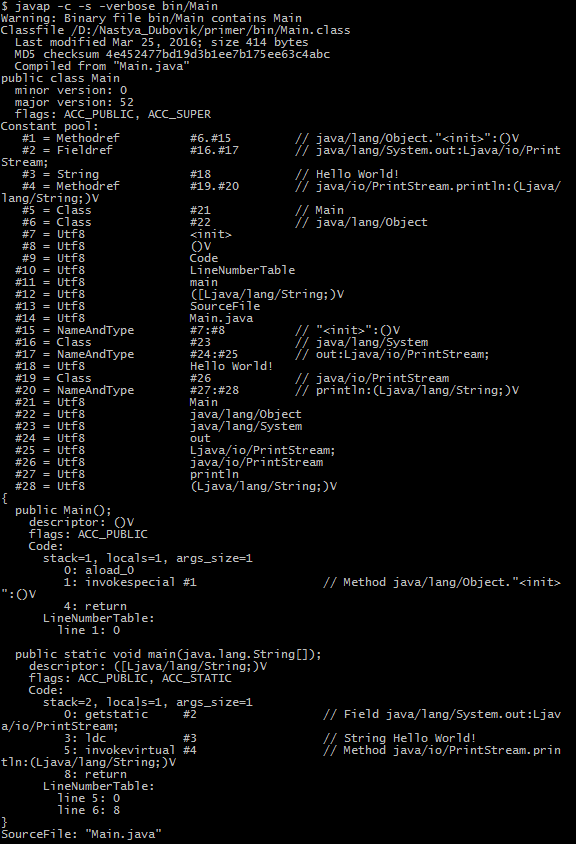
**Console App Run**

*Как установить и написать первую программу на Java?*

1. С сайта oracle.com скачиваем jdk, которое будет удовлетворять системным требованиям (выбираем систему и её разрядность).
2. Разархивируем скачанный файл (по умолчанию установиться на диск С в папку Java).
3. В переменную среды окружения PATH указываем путь к jdk/bin.
4. Проверяем установилась ли java с помощью команды java -version (или другим способом).
5. Создаём папку bin и src. В src будем хранить файлы с расширением java, а в bin – их байткод с расширением class. Создаём файл (с помощью любого редактора), в котором пишем java код. Сохраняем файл с расширением .java в папку src.



1. Через командную строку заходим в корневую папку (на папку выше чем src и bin). Компилируем наш файл в папку bin: **javac –d bin src/Main.java**
2. Далее выполняем скомпилированный файл: **java -classpath bin Main**
3. Просмотреть байт код фала class можно с помощью команды: **javap -c -s –verbose bin/Main**



*Какая кодировка по умолчанию в java?*

Java использует представление строк в формате Unicode. Кодировка по умолчанию зависит от системы и её настроек. Для русских виндов принята кодировка Cp1251.

*Если создать пустой файл, он скомпилируется?*

Ошибки не возникнет. Так как было ноль классов, то получим ноль файлов с расширением class.

*Если создать пустой класс, он скомпилируется?*

Ошибки не возникнет. В байт коде будет прописан конструктор по умолчанию.

*Если нет метода main?*

Файл скомпилируется, но не выполнится.

****

**Структура класса**

*Class как тип и как файл*

Class как тип:

Класс – это абстракция сущности некоторой предметной области.

Класс – это описание совокупности объектов с общими атрибутами, методами, отношением и семантикой.

Классы определяют структуры и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.

Class как файл:

После компиляции .java файла компилятором javac (стандартный), создаётся .class файл, который содержит байт код.

*Структура .java файла?*

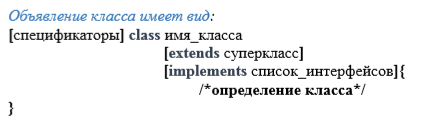
Первым в файле может быть написан ***комментарий***, потом package.

Одно объявление ***package***, иначе жалуется:



Так же объявление import только вначале. Та же ошибка.

*Объявление* ***класса***



Модификаторы – public, final, abstract.

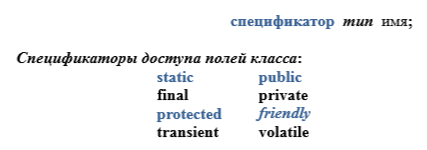
Кроме public класса (внешнего, enclosing, виден из других пакетов) могут быть ещё классы в этом файле, только их область видимости может быть только friendly. Число классов не ограничено (при компиляции создастся столько файлов .class , сколько классов написано в файле .java , учитывая внутренние, анонимные, вложенные, интерфейсы и перечисления).

Extends идёт перед implements, так как класс, от которого мы наследуемся, может реализовывать интерфейс, который мы перечислили в implements.

*Почему нет множественного наследования?*

Представим себе ситуацию, что множественное наследование есть. Класс А наследует класс В и С. Причём и в классе В, и в классе С переопределён метод show. Создав объект класса А и вызвав метод show, что мы получим? Конфликт (проблема алмаза).

*Объявление* ***поля***



Модификаторы – private, protected, friendly, public, static, final, transient (не будет сериализовано), volatile.

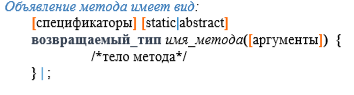
final + volatile = no

Константа **не** получает значение по умолчанию!

Константа может быть объявлена как поле экземпляра класса, но не проинициализирована. Тогда она должна быть проинициализирована в логическом блоке или в конструкторе (но только в одном месте).

Если константа объявлена как поле класса, то она может быть проинициализирована либо при объявлении, либо в с статическом блоке.

*Объявление* ***метода***



Модификаторы – private, public, protected, friendly, abstract, static, final, native, synchronized, strictfp.

Методы могут выбрасывать исключения: throws список\_ошибок\_через\_запятую

*Что такое сигнатура метода?*

Это имя и список параметров.

*Конструктор*

Конструктор имеет тоже имя что и класс, не возвращает значение. Конструктор **НЕ** может быть static, abstract или final.

*Зачем конструктору модификатор private?*

Например, для single tone.

*Конструктор может выбрасывать исключение?*

Может как checked, так и unchecked.

*Если нет модификатора доступа, то какой будет?*

Как у класса.

*Логические и статические блоки*

|  |  |  |
| --- | --- | --- |
|  | **Логические блоки** | **Статические блоки** |
| *Определение* |  |  |
| *Для чего?* | Используются для инициализации полей, но в них так же можно вызывать методы текущего класса, обращаться к полям текущего класса, создавать объекты, создавать классы. При обращении к полю, которое ещё не было явно объявлено, возможно только через ссылку this. | Выполняет те же функции, что и логический блок. Но он может обращаться **только** к статическим полям и методам. |
| *Когда вызывается?* | При создании объекта они выполняются последовательно, в порядке размещения, вместе с инициализацией полей. После выполнения последнего блока вызывается конструктор. | Вызывается один раз при создании объекта или при первом обращении к статическому полю или методу. |
| *Может ли выбрасывать исключение?* | Checked исключение не может, а unchecked может. Так как дальше нельзя отследить исключение. | Checked исключение не может, а unchecked может.  Так как дальше нельзя отследить исключение. |
| *Где располагается?* | В любом месте: внутри метода, конструктора, логического блока, статического блока. | Только внутри класса (глубже вложенности нет). Внутри статического вложенного. |

*Замечание:* в логических и статических блоках нельзя вызывать **super**();, так как он вызывается первой строчкой в конструкторе.

*Статические методы и поля*

1. Поле, объявленное с модификатором static, является общим для всех объектов. Называется ***переменной класса***.
2. Если один объект изменит значение такого поля, то и в другом объекте этого же класса значение будет изменено.
3. Для работы со статическими переменными используются статические методы. Нестатические методы могут обращаться к статическим методам и переменным, а статические методы только к статическим (так как они не имеют ссылки this).
4. Статические методы являются ***методами класса***.
5. Статические методы реализуют парадигму «раннего связывания» (жестоко определяем версию на этапе компиляции).
6. Статические методы применяются в случае, когда методу не нужен доступ к состоянию объекта или методу нужен доступ только к статическим полям класса.
7. Статические поля используются редко, кроме static final.
8. **Статические методы** наследуются, перегружаются, но **не переопределяются**.

*Можно ли писать throws у главного класса?*

Можно.

*Модификатор или спецификатор (private, public, friendly, protected)?*

Модификаторы доступа (область видимости).

*Синтаксис*

1. Первым символом в названии класса/метода/поля может быть только буква строчная или прописная (из юникода, причём можно их комбинировать) или $, или \_.
2. В названии класса/метода/поля могут присутствовать цифры, \_, $

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Class  enclose | Method | Поля | Переменные метода | Локальные переменные | Конст  руктор | Логический блок | Enum | Interface | Nested class | Inner class | Локальный  класс | Anonymous |
| Private | **-** | **+** | **+** | **-** | **-** | **+** | **-** | **-** | **-** |  | **+** | **-** | **-** |
| Protected | **-** | **+** | **+** | **-** | **-** | **+** | **-** | **-** | **-** |  | **+** | **-** | **-** |
| Friendly | **+** | **+** | **+** | **-** | **-** | **+** | **+** | **+** | **+** |  | **+** | **+** | **+** |
| Public | **+** | **+** | **+** | **-** | **-** | **+** | **-** | **+** | **+** |  | **+** | **-** | **-** |
| Static | **-** | **+** | **+** | **-** | **-** | **-** | **+** | **-** | **-** |  | **+** | **-** | **-** |
| Final | **+** | **+** | **+** | **+** | **+** | **-** | **-** | **-** | **-** |  | **+** | **+** | **-** |
| Abstract | **+** | **+** | **-** | **-** | **-** | **-** | **-** | **-** | **+** (redundant) |  | **+** | **+** | **-** |
| Native | **-** | **+** | **-** | **-** | **-** | **-** | **-** | **-** | **-** |  | **-** | **-** | **-** |
| Transient | **-** | **-** | **+** | **-** | **-** | **-** | **-** | **-** | **-** |  | **-** | **-** | **-** |
| Synchro  nized | **-** | **+** | **-** | **-** | **-** | **-** | **+** (только как часть метода) | **-** | **-** |  | **-** | **-** | **-** |
| Volatile | **-** | **-** | **+** |  |  | **-** | **-** | **-** | **-** |  | **-** | **-** | **-** |
| Strictfp | **+** | **+** | **-** | **-** | **-** | **-** | **-** | **+** | **+** |  | **+** | **+** | **+** |

**Перегрузка и переопределение**

|  |  |  |
| --- | --- | --- |
|  | **Перегрузка** | **Переопределение** |
| Определение | Определение методов с одинаковым именем, но различной сигнатурой. | Переопределённым методом называется метод в производном классе сигнатуры которых совпадают. |
| Реализует | *Раннее связывание*  То есть выбор производится в зависимости от типа ссылки на этапе *компиляции*. | *Позднее связывание (динамическое)*  Ассоциация между ссылкой и классом не устанавливается, пока объект с заданным именем не будет создан на стадии *выполнения*. |
| Пример | *Стиральная машина* имеет несколько режимов стирки. Мы говорим «стирай белое бельё», «стирай чёрное бельё». | *Человек* (как класс) имеет реализацию метода «дай совет, как лечить от простуды». Студент, наследующийся от класса Человек, имеет свою реализацию. Например, пей тёплый чай, не ходи в универ. Доктор, который наследуется тоже от человека, имеет совершенно иное понятие о простуде, поэтому переопределит его по-своему. |
| Замечание | 1. При разрешении перегрузки компилятор руководствуется типом ссылки. | 1. Область видимости сужать нельзя 2. В качестве возвращаемого типа может стоят подкласс данного класса. 3. В рамках одного класса переопределять метод нельзя. |
| Static | Статика перегружает не статику и наоборот. | Статика не может переопределить не статику и наоборот. |