**Представление чисел в памяти**

В Java отрицательные числа представляются в дополнительном коде, который формируется следующим образом: сначала число переводится в двоичный вид, затем инвертируется (0 заменяется на 1 и наоборот) и к результату прибавляется 1. Чтобы получить положительное число из отрицательного нужно также инвертировать его и прибавить 1.

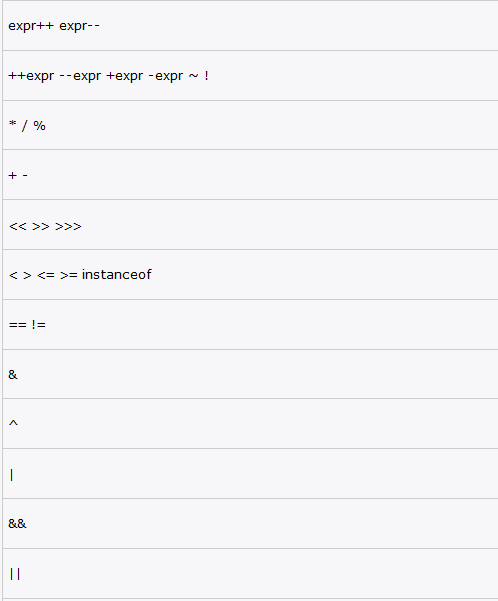
Старший двоичный разряд хранит цифру знака: 1 – отрицательное, 0 - положительное. Таким образом, когда нужно сложить положительное число с отрицательным, мы складываем их в двоичном виде.

Все операторы Java можно разделить на следующие группы:

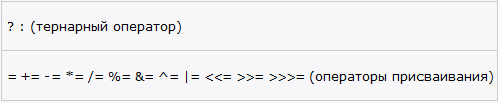
* арифметические операторы;
* операторы сравнения;
* побитовые операторы;
* логические операторы;
* операторы присваивания;
* прочие операторы.

|  |  |
| --- | --- |
| Арифметические операторы | |
| + | операция сложения двух чисел |
| - | операция вычитания двух чисел |
| \* | операция умножения двух чисел |
| / | операция деления двух чисел |
| % | получение остатка от деления двух чисел |
| ++ | Префиксный и постфиксный инкремент |
| -- | Префиксный и постфиксный декремент |
| Побитовые операторы | |
| & | побитовое и |
| | | побитовое или |
| ^ | побитовое исключающее или |
| ~ | побитовое дополнение |
| << | Побитовый сдвиг влево, при сдвиге влево старший бит теряется |
| >> | Побитовый сдвиг вправо, при сдвиге младшие байты теряются. При этом сохраняется бит знака. |
| >>> | Сдвиг вправо с заполнением нулями (не сохраняет знаковый бит, ставит его 0) |
| операторы сравнения | |
| == | Проверка на равенство |
| != | Проверка на неравенство |
| < | Меньше чем |
| > | Больше чем |
| >= | Больше или равно |
| <= | Меньше или равно |
| логические операторы (одиночные - поразрядные) | |
| | или || | или |
| & или && | и |
| ! | не |
| ^ | Исключающее или |
| Операторы |, & и ||, && идентичны, но во втором случае выражения могут не высчитываться до конца, если их значение уже очевидно | |
| операторы присваивания; | |
| = | обычное |
| += | Сложение с присваиванием |
| -= | Вычитание с присваиванием |
| \*= | Умножение с присваиванием |
| /= | Деление с присваиванием |
| %= | Остаток от деления с присваиванием |
| &= | Логическое и с присваиванием |
| |= | Логическое или с присваиванием |
| ^= | Исключающее или с присваиванием |
| <<= | Сдвиг влево с присваиванием |
| >>= | Сдвиг вправо с присваиванием |
| >>>= | Нулевой сдвиг вправо с присваиванием |

**Приоритет операций**



В лямбда выражениях



**УСЛОВНЫЕ КОНСТРУКЦИИ**

**If /else**

if ([логическое выражение]) {

//do some things

} else {

//do other things

}

**switch/case**

switch(var){

case 1:

//do

break;

case 2:

//do

break;

default:

//do

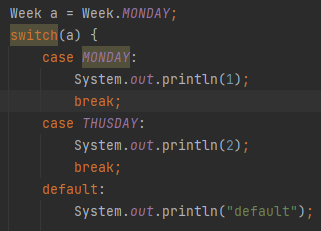
}

В операторе switch можно использовать типы char, byte, short, int, Character, Byte, Short, Integer, String, or an enum.

Если не указывать break, то при нахождении первого истинного условия, за ним также выполняться все последующие блоки case и default;

* Оператор switch отличается от опреатора if тем, что может проверять проверку только равенства, а if любого булевого типа.
* Два case в одном switch не могут иметь одинаковые значения.
* Можно использовать несколько операторов ветвей case.
* Оператор switch эффективнее операторов else-if.
* Switch можно вкладывать друг в друга. Внутренний switch может иметь такие же константы в case как и внешний.

Применение switch-case для enum.



**Тернарный оператор**

int z = x < y ? (x+y) : (x-y);

Выполняет приведение типов. Поэтому при его использовании могут возникать неочевидные ошибки



Если оба флага false, выскочит NPE, т.к. 2 и null приведутся к типу Integer, а затем 1 и Integer должны будут быть приведены к int, потребуется Unboxing что вызовет NPE.

**ЦИКЛЫ**

for ([инициализация счетчика]; [условие]; [изменение счетчика])

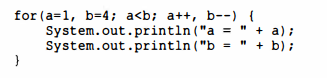
{

    // действия

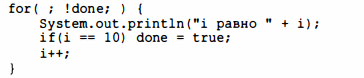
}

Переменную управления циклом можно объявить в инициализирующей части оператора for. Такая переменная будет видна только внутри цикла.

В каждом блоке можно использовать по несколько переменных



В блоке условие не обязательно должна быть проверка переменной, он может содержать любое логическое выражение. Отдельные части оператора for могут остутствовать



Цикл **for each** называется усовершенствованным циклом for. Предназначен для перебора коллекций и массивов.



Эта разновидность упрощает синтаксис и исключает возможность ошибок, связанных с выходом за пределы массива.

Изменение переменной итерации не приводит к изменению исходного массива или коллекции, поскольку она доступна только для чтения.

**Do while**

int j = 7;

do{

    System.out.println(j);

    j--;

}

while (j > 0);

Тело цикла выполняется хотя бы один раз. Проверка условия происходит после выполнения.

**while**

int j = 6;

while (j > 0){

    System.out.println(j);

    j--;

}

Цикл while может быть пустым. While(++i < --j); Обычно тело цикла пустое, если управляющее выражение само может выполнить все необходимые действия. Условие циклов должно быть логическим выражением

Оператор **break** позволяет выйти из цикла.

Но он может также использоваться в качестве альтернативы goto



Блоки помечаются с помощью метки *метка:.* При использовании break с меткой выполнение будет продолжено с конца помеченного блока. Блок с меткой можно использовать только внутри помеченного блока. Используется в основном для выхода из вложенных циклов.

оператор **continue** позволяет перейти к следующей итерации цикла. Этот оператор также можно использовать с меткой. Например для начала следующей итерации внешнего цикла.

**КЛАССЫ, АБСТРАКТНЫЕ КЛАССЫ, ИНТЕРФЕЙСЫ**

Шаблоном или описанием объекта является **класс**. Класс так же представляет собой тип данных. Код класса определяет интерфейс для объекта, а также возможные состояния.

Параметр – определенная в методе переменная, которая принимает значение при вызове. Аргумент – непосредственно значение, передаваемое методу при его вызове.

Когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная скрывает переменную экземпляра.

**Конструктор –** специальный метод, вызывающийся при создании нового объекта данного класса и выполняют инициализацию объекта. Он не имеет явно указанного типа, но на самом деле возвращаемым типом является тип самого класса.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается **конструктор по умолчанию** без параметров.

Конструктор подкласса всегда вызывает конструктор базового, причем в начале своего конструктора. По умолчанию вызывается конструктор без параметров базового класса. Если конструктора без параметров нет, то требуется явный вызов конструктора базового класса используя ключевое слово super();

С помощью super можно обращаться к методам и полям родительского класса.

Методы можно переопределять и перегружать. Конструктор можно перегрузить, но переопределить нельзя. У каждого класса он свой собственный. Конструктор вызывает конструктор родительского класса (явно и неявно).

Перегружаемые методы должны отличаться по типу или количеству их параметров. Возвращаемые типы могут различаться, но только этого недостаточно чтобы отличить 2 разных варианта метода.

В JDK5 были добавлены **varargs (**variable-length arguments – аргументы переменной длины**).** Для указания аргументов переменной длины служат три точки (…). Все параметры помещаются в массив с указанным именем.



В методе могут быть и обычные параметры, но параметры переменной длины должны размещаться в конце. Метод может содержать только один параметр переменной длины. Такие методы так же можно перегружать.

Оператор **new** выделяет память для объекта и возвращает ссылку на него.

Ключевое слово **this** представляет ссылку на текущий экземпляр класса.

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью **инициализатора** объекта. Инициализатор выполняется до любого конструктора. То есть в инициализатор мы можем поместить код, общий для всех конструкторов

[модификатор доступа][static] class [имя класса] extends [список наследуемых классов] implements [список реализуемых интерфейсов] {

//поля

//методы и конструктор

//блок инициализации

{

//инициализация

}

}

**Класс Object**

Object это базовый класс для всех остальных объектов в Java. Любой класс наследуется от Object и, соответственно, наследуют его методы:

public boolean equals(Object obj) – служит для сравнения объектов по значению; по умолчанию сравнивает ссылки.

int hashCode() – возвращает hash код для объекта; По умолчанию использует целочисленное представление адреса памяти.

String toString() – возвращает строковое представление объекта; По умолчанию реализация toString создает вывод в виде package.class@hashCode

Class getClass() – возвращает класс объекта во время выполнения;

protected Object clone() – создает и возвращает копию объекта;

void notify() – возобновляет поток, ожидающий монитор;

void notifyAll() – возобновляет все потоки, ожидающие монитор;

void wait() – остановка вызвавшего метод потока до момента пока другой поток не вызовет метод notify() или notifyAll() для этого объекта;

protected void finalize() – может вызываться сборщиком мусора в момент удаления объекта при сборке мусора.

**Абстрактный класс –** класс, экземпляр которого нельзя создать. призваны предоставлять базовый функционал для классов-наследников. Для создания используется ключевое слово **abstract.**

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Если класс имеет хотя бы один абстрактный метод, то этот класс должен быть объявлен абстрактным.

**Интерфейсы** определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

С помощью интерфейса можно полностью абстрагировать интерфейс класса от его реализации. Мы указываем, что именно должен выполнять класс, но не как это делать. Мы также изолируем определения методов от иерархии наследования.

Интерфейс неспособен сохранять данные состояния (нет переменных экземпляра).

Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Если класс реализует интерфейс, то он должен реализовать все методы интерфейса. Если класс не полностью реализует методы интерфейса, то он должен быть объявлен как abstract.

Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам класса.

По умолчанию все методы в интерфейсе фактически имеют модификатор public. Однако начиная с Java 9 мы также можем определять в интерфейсе методы с модификатором private. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию. Могут использоваться только внутри самого интерфейса, в котором они определены. Не наследуются. Введены так как начиная с Java 8 интерфейс начал содержать не только интерфейс, но и реализацию. Реализация может содержать вспомогательные методы, которые без private становились бы частью интерфейса.

Интерфейсы, как и классы, могут наследоваться:

interface BookPrintable extends Printable{

    void paint();

}

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах с модификаторами доступа public, private или protected. При использовании за пределами объемлющей области, его имя должно быть уточнено именем класса или интерфейса, членом которого он является (*ClassA.InterfaceA*).

Вложенный интерфейс, объявленный как private может быть реализован только внутри объемлющей области.

В интерфейсах могут быть объявлены **переменные.** Они неявно объявляются как static и final и должны быть инициализированы, их нельзя изменить в классе, реализующем интерфейс (константы).

Если в классе реализуются 2 интерфейса, в которых объявлен один и тот же метод, то реализуемый метод будет использоваться клиентами любого из двух интерфейсов.

При переопределении метода можно частично менять сигнатуру:

1. Изменять модификатор доступа к более открытому.
2. Изменять возвращаемый тип, менять базовые классы на их наследников.
3. Изменять throws исключения, так же от базового к наследникам.

Другие изменения сигнатуры приведут к перегрузке метода.

**ВНУТРЕННИЕ И ВЛОЖЕННЫЕ КЛАССЫ**

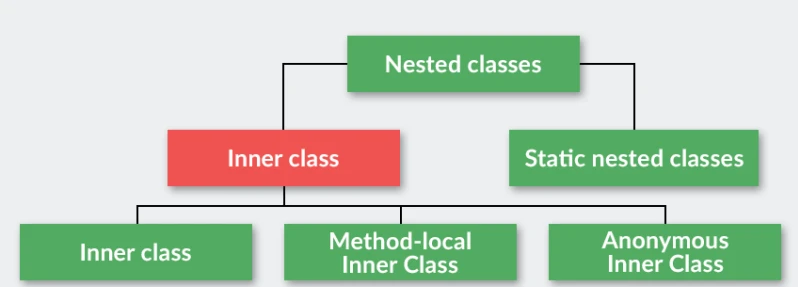
Класс называется **вложенным** (nested), если он определен внутри другого класса. Вложенный класс создается для того, чтобы обслуживать окружающий его класс.

Они делятся на 2 вида:

1. Non-static nested classes — нестатические вложенные классы. По-другому их еще называют inner classes — внутренние классы. Они бывают:

* Внутренний класс (inner class)
* Локальный класс (local class)
* анонимный класс (anonymous class)

1. Static nested classes — статические вложенные классы.



**Особенности внутренних классов**

Объект внутреннего класса не может существовать без объекта «внешнего» класса. Чтобы создать объект внутреннего класса, нужно создать объект внешнего.

У объекта внутреннего класса есть доступ к переменным «внешнего» класса (в том числе private). Но внешний класс не имеет доступа к членам вложенного.

Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса. Во внутренний класс неявно передается ссылка на объект внешнего класса.

Внутренний класс не может содержать статические переменные и методы. (До Java16)

При создании объекта внутреннего класса важную роль играет его модификатор доступа.

Модификаторы доступа для внутренних классов работают так же, как и для обычных переменных.

Можно получить ссылку на экземпляр внешнего класса. Пример: Airplane.this – ссылка на самолет, this – ссылка на крыло.

**Статический вложенный класс**

Для создания экземпляра такого класса, нужно через точку перечислить весь путь от внешнего класса до нужного. Может ссылаться только на статические члены внешнего класса.

**Локальные классы**

Данные классы объявляются внутри других методов. Они обладают всеми свойствами нестатического вложенного класса, только создавать их экземпляры можно только в методе, при чем метод не может быть статическим.

Локальные классы способны работать только с неизменяемыми локальными переменными (можно не объявлять их final, но они все равно не должны изменяться). Но можно сипользовать любые поля обрамляющего класса.

Локальные классы нельзя объявлять с модификаторами доступа.

Локальные классы обладают доступом к переменным метода.

**Анонимные классы –** вложенный класс без имени.

Под капотом анонимные классы – просто обычные нестатические вложенные классы.

Они обычно объявляются либо как подклассы существующего класса, либо как реализации некоторого интерфейса.

Вы можете объявить поля и методы внутри такого класса, но не можете объявить конструктор. Вместо этого возможно объявить статический инициализатор

***Использование анонимных классов оправдано во многих случаях, в частности когда:***

* тело класса является очень коротким;
* нужен только один экземпляр класса;
* класс используется в месте его создания или сразу после него;
* имя класса не важно и не облегчает понимание кода.

Внутренние и вложенные классы повышают инкапсуляцию. Если класс полезен только для одного другого класса, то логично встроить его в этот класс и хранить вместе.

**КЛЮЧЕВЫЕ СЛОВА STATIC И FINAL**

В языке программирования Java ключевым словом **static** помечают члены (поля или методы), которые принадлежат классу, а не экземпляру этого класса.

Это означает, что какое бы количество объектов вы не создали, всегда будет создан только один член, доступный для использования всеми экземплярами класса.

Ключевое слово static применимо к переменным, методам, блокам инициализации, импорту и вложенным классам (nested classes).

В языке Java, если поле объявляется статическим, то в независимости от количества созданных объектов класса — всегда будет существовать только один экземпляр статического поля.

Статические переменные размещаются в специальном пуле в памяти JVM, называемом Metaspace. Так как они относятся к классу, статические поля не сериализуются.

Статические переменные могут быть созданы только, как переменные класса. Они не могут быть локальными переменными.

**Статические методы** также принадлежат классу, а не объекту. При этом следует помнить, что из статического метода можно получить доступ только к статическим переменным или к другим статическим методам. Однако из методов экземпляра можно вызывать статические методы и обращаться к статическим переменным.

При наследовании статические методы наследуются, но их нельзя переопределить. При объявлении метода с идентичной сигнатурой, происходит перекрытие. Этот метод никак не связан с методом базового класса.

Статические методы можно определять и в интерфейсах, но они не могут быть переопределены методами в реализующих интерфейс классах.

* Абстрактные методы не могут быть статическими
* Статические методы не могут использовать ключевые слова this или super
* Методы экземпляра также могут непосредственно обращаться к статическим переменным и статическим методам
* Статические методы могут обращаться ко всем статическим переменным и другим статическим методам
* Статические поля и методы не являются потокобезопасными
* Статические методы связываются во время компиляции, в отличие от не статических методов, которые связываются во время исполнения. Из-за этого статические методы не могут быть переопределены, т.к. полиморфизм во время выполнения не распространяется на них.

Cтатический импорт позволяет обращаться к статическим членам класса непосредственно по имени члена, без дополнительного указания имени класса и пакета (println).

Статический блок инициализации используется для инициализации статических переменных. Обычно в тех случаях, когда нам требуется выполнить многострочную обработку. Статический блок выполняется только один раз при первой загрузке класса.

**статические вложенные классы** имеют доступ только к статическим членам внешнего класса. По сути является NameSpace (Чтобы иметь классы с одинаковыми именами).

Суть **модификатора final** - сделать дальнейшее изменение объекта невозможным.

*static final* позволяет объявить константу. Она будет одинаковой в любом экземпляре.

Вы можете применять этот модификатор тремя способами: для класса, для поля (переменной(в том числе аргументам методов)) и для метода.

При использовании с переменной, ее значение нельзя будет изменить после инициализации. Переменные с final - это константы. Для ссылочных переменных это означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. **Это важно!** Ссылку изменить нельзя, но состояние объекта изменять можно.

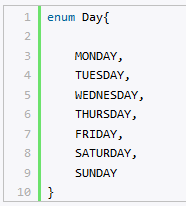
С java 8 появилось понятие — effectively final. Применяется оно только к переменным (в том числе аргументам методов). Суть в том, что не смотря на явное отсутствие ключевого слова final, значение переменной не изменяется после инициализации.

Использование final с методом будет значить, что при наследовании данный метод **нельзя переопределить.**

При использовании final с классом, от данного класса нельзя будет наследоваться.

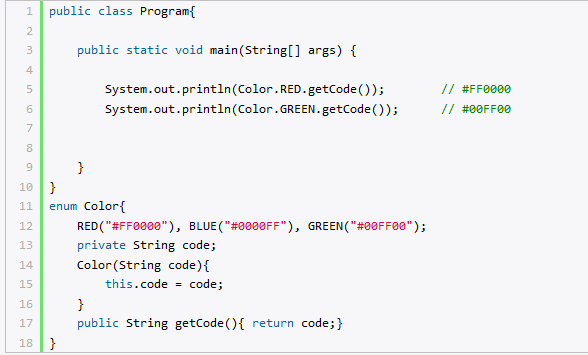
**ENUMS**

Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

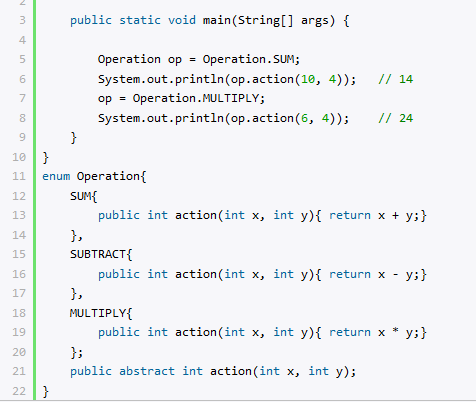


Каждое перечисление имеет статический метод values(). Он возвращает массив всех констант перечисления.

Перечисления, как и обычные классы, могут определять конструкторы, поля и методы. Конструктор всегда приватный, поэтому создавать константы перечисления можно только внутри enum.



Также можно определять методы для отдельных констант:



EnumMap, расширение Enum, Enum в switch.

**ПАКЕТЫ**

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.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Принадлежность к пакету позволяет гарантировать однозначность имен.

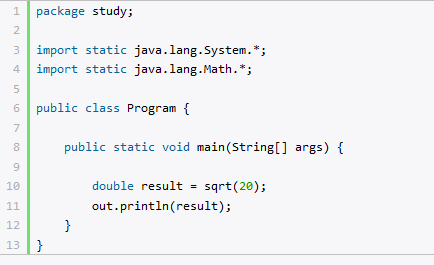
Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву **package**, после которой указывается имя пакета.

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом.

Классы необязательно определять в пакеты. Тогда считается что класс находится в пакете по умолчанию.

Если нам надо использовать классы из других пакетов, то нам надо подключить эти пакеты и классы (Можно использовать полный путь с названием всех пакетов). Директива **import** указывается в самом начале кода, после чего идет имя подключаемого класса. Чтобы подключить все классы из пакета можно использовать *packagename.\**.

В java есть также особая форма импорта - статический импорт. Для этого вместе с директивой import используется модификатор static:



Благодаря операции статического импорта мы можем использовать статические методы без названия класса.

**Transient –** модификатор, указываемый перед полем класса для обозначения того, что данное поле не должно быть сериализовано.

**ClassLoader**

Классы не загружаются в память не все сразу, а по мере необходимости. Этим занимается ClassLoader, являющийся частью JRE.

**Типы загрузчиков**

*Extension ClassLoader – загрузчик расширений.*

Загружает расширения основных классов Java из библиотеки расширений JDK. Он является дочерним элементом загрузчика Bootstrap ClassLoader и загружает расширения из каталога JRE / lib / text или любого другого каталога, указанного в системном свойстве java.ext.dirs.

*Application or System ClassLoader - системный загрузчик или загрузчик приложения.*

Этот тип загружает все классы уровня приложения, найденные в параметре командной строки -cp или в переменной среды CLASSPATH.

*Bootstrap ClassLoader -* базовый загрузчик, также называется Primordial ClassLoader. загружает стандартные классы JDK из архива rt.jar.

В Java используется иерархия загрузчиков классов, где корневым, разумеется, является базовый. Далее следует загрузчик расширений, а за ним уже системный. Естественно, каждый загрузчик хранит указатель на родительский для того, чтобы смочь делегировать ему загрузку в том случае, если сам будет не в состоянии этого сделать.

Любой класс, который расширяет java.lang.ClassLoader, может предоставить свой способ загрузки классов.

**Три принципа загрузки классов**

* *Делегирование –* запрос на загрузку передается родительскому классу, если родительский класс не смог загрузить класс, выполняется попытка загрузить класс самостоятельно. Каждый загрузчик ведет учет классов, загруженных им (помещает их в кеш).
* *Видимость –* загрузчик видит только свои классы и классы родители и не знает о классах потомка.
* *Уникальность –* класс может быть загружен только однажды. Механизм делегирования позволяет убедиться, что загрузчик не перегрузит загруженный ранее предком класс.

**Схема загрузки классов**

* Вызывается загрузка класса, идет поиск класса в кэше текущего загрузчика.
* Если нет в кэше, по принципу делегирования управление передается родительскому загрузчику. Он тоже ищет в кэше и т.д., пока не дойдет до базового загрузчика.
* Если в базовом загрузчике нет информации об искомом классе, будет выполнен поиск байт-кода данного класса по расположению классов, о котором знает данный загрузчик. Если загрузить класс не удается, управление вернется обратно загрузчику-потомку.
* Загрузчик-потомок будет пытаться выполнить загрузку из известных ему источников.
* Когда байт-код класса найден, происходит загрузка класса в JVM и получение экземпляра типа Class.