**LESSON 3**

**Object**

Понятие «объект» не имеет конечного понятия. Можно сказать, что объект – это осязаемая сущность, которая чётко проявляет своё поведение. Если рассмотреть объект в контексте ООП, то можно дать следующее понятие. Объект ООП – это совокупность переменных состояния и связанных с ним методов (операций). Эти методы показывают, как данный объект взаимодействует с окружающей системой.

Объект служит для того, чтобы держать в себе описание класса и добавить доступ к некоторым специальным операциям над ним.

*Пример*

Можно работать с классами, не создавая объектов. В таком случае мы сможем работать только со статическим полями и методами. Если нам нужно создать 1000 экземпляров класса Man, каждый из которых содержит свою начинку (хотя бы разные имена: Петя, Вася, Никита), тогда возникает необходимость в создании конкретной реализации такого класса.

*Способы создания объектов?*

|  |  |  |
| --- | --- | --- |
| С помощью оператора new | Зачем нужен такой способ? |  |
| Пример | *Obj object = new Obj();* |
| Какие механизмы срабатывают? | 1. Загрузка класса (.class -> .java) 2. Выделение памяти в куче для хранения свойств объекта 3. Создание объектной ссылки, которая будет ссылаться на эту ячейку памяти 4. Локальной переменной присваивается эта ссылка |
| Клонирование | Зачем нужен такой способ? | В случае, если нам нужен точно такой объект. Если сделать через присваивание, то, изменяя исходный объект, изменится новый. |
| Пример | *Book book = new Book();*  *Book book2 = book.clone();* |
| Какие механизмы срабатывают? | Заметим то, чтобы данный способ был реализуемым, класс должен реализовывать интерфейс-маркер Cloneable и переопределять метод clone(). Иначе будет выброшен exception CloneNotSupportedException.     1. Вызывается метод clone на объекте *book* => super.clone() 2. Выделение памяти в куче для хранения свойств объекта 3. **НЕ** вызываем new! Копируем значение полей базового типа 4. Если в качестве поля используется объектная ссылка, то смотрим, реализует ли класс, к котором она относится, интерфейс Cloneable и переопределяет ли метод clone. Если не реализует, то к такому же полю нового объекта присваивается ссылка на такое же поле клонируемого.     То есть, изменяя поле author, оно изменится у обоих объектов. Такое клонирование называется *неполным (ранним).*  Если класс реализует Cloneable, то метод clone() должен быть переопределён так:    Тогда поле, содержащее объектную ссылку, НЕ будет ссылаться на тот же объект, что и исходный объект, а будут ссылаться на новый объект. Такое клонирование называется *полным(поздним)*. |
| Десериализация | Зачем нужен такой способ? | Для общения Java компонентам друг с другом через системы и сети нужен протокол, о котором знает и отправитель, и получатель. Сериализация создана для этого. Отправитель сериализует объект, отправляет получателю. А он в свою очередь десериализует. |
| Пример |  |
| Какие механизмы срабатывают? | Заметим то, что для реализации этого способа необходимо реализовать интерфейс-маркер Serializable. Иначе   1. Создаём объект класса FileInputStream (предназначен для чтения строки байтов; пакет java.io), передавая в конструктор файл, в котором записан байт-код. 2. Создаём объект класса ObjectInputStream (предназначен для десериализации примитивных данных или объекта, ранее записанные в строку байтов с помощью ObjectOutputStream; пакет java.io), передавая в конструктор созданный объект с байт строкой. 3. Вызываем метод readObject(), который считывает объект с ObjectInputStream. Он считывает класс объекта, сигнатуру класса, значения не transient и не static полей класса и все супертипы объекта. 4. Явное приведение типов |
| Рефлексия | Зачем нужен такой способ? |  |
| Пример | *Obj object2=(Obj)Class.forName("by.epam.Obj").newInstance();* |
| Какие механизмы срабатывают? |  |

**Class Object**

*Где лежит?*

java.lang.Object

*Кого реализует или наследует?*

Никого

*Назначение и использование данного класса?*

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

*Состояние/поведение данного класса?*

Состояние не имеет (нет полей).

Поведение определяется следующими методами (11 публичных методов, 5 обычных и 6 с нативной реализацией):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | При выполнении метода clone() сначала проверяется, можно ли клонировать исходный объект. Если разработчик хочет сделать объекты своего класса доступными для клонирования через Object.clone(), то он должен реализовать в своем классе интерфейс Cloneable. Если проверка не выполняется успешно, метод порождает ошибку **CloneNotSupportedException**. Если интерфейс Cloneable реализован, то порождается новый объект от того же класса, от которого был создан исходный объект. При этом копирование выполняется на уровне виртуальной машины, никакие конструкторы не вызываются. Затем значения всех полей, объявленных, унаследованных либо объявленных в родительских классах, копируются. Полученный объект возвращается в качестве клона. | | |
|  | | | Предназначен для переопределения в подклассах с  выполнением общих соглашений о сравнении содержимого двух объектов.  При переопределении должны выполняться соглашения:  - **рефлексивность** – объект равен самому себе (если он не null);  - **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда  возвращает значение **true**;  - **транзитивность** – если метод **equals()** возвращает значение true при сравнении  объектов **x** и **y,** а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение  **true**;  - **непротиворечивость** – при многократном вызове метода для двух не подвергшихся  изменению за это время объектов возвращаемое значение всегда должно быть  одинаковым;  - **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**. | | | | | |
|  | | | | | | Возвращает объект типа Class, который описывает класс(имя, методы, поля), от которого был порожден этот объект; | | |
|  | | | | | Вызывается перед уничтожением объекта  автоматическим сборщиком мусора (garbage collection). Отрабатывает автоматически. Если нужно мы можем обратиться напрямую к нему.  Java не контролирует, какой поток вызовет метод finalize. Однако, гарантирует, что, поток, который вызвал этот метод, не будет проводить какую-либо видимую пользователю синхронизацию при выполнении метода. Вызывается только один раз JVM.  Если будет выброшено исключение, это приведёт к тому, что финализация метода будет приостановлена, в противном случае проигнорирована. | | | |
|  | Возвращает хэш-код объекта, вычисление которого  управляется следующими соглашениями:  **\_** во время работы приложения значение хэш-кода объекта **не изменяется**, если объект  не был изменен;  \_ все **одинаковые** по содержанию **объекты** одного типа должны иметь **одинаковые хэш-коды**;  \_ различные по содержанию объекты одного типа могут иметь различные хэш-коды.  Следует переопределять всегда, когда переопределен метод **equals().** | | | | | | | |
|  | | | | | | | | Возвращает представление объекта в виде строки. Для класса Object и его наследников, не переопределивших toString() |
|  | | Будит поток, который находится в состоянии ожидания на мониторе объекта. Если таких поток несколько, то выбирается один из них случайным образом. Поток, который разбудили, не будет иметь возможности выполнять код до тех пора, пока текущий поток не уступит ему lock этого объекта.  Может выбрасить ошибку **IllegalMonitorStateException** в случае, если текущий поток не является владельцем монитора объекта.  Поток является владельцем если:   1. Выполняя синхронизированный метод этого объекта 2. Выполняя синхронизированного выражения, которое синхронизируется на объекте 3. Для объектов типа Class, путём выполнения синхронизированного статического метода этого класса | | | | | | |
|  | | | | Будит все потоки, которые находятся в состоянии ожидания на мониторе объекта. Потоки, которые были разбужены, не будут иметь возможности выполнять код до тех пора, пока текущий поток не уступит ему lock этого объекта.  Может выбрасить ошибку **IllegalMonitorStateException** в случае, если текущий поток не является владельцем монитора объекта. | | | | |
|  | | | | | | Вынуждает текущий поток подождать, пока другой поток вызывает метод notify() или notifyAll().  Может выбпасить **IllegalMonitorStateException** в случае, если текущий поток не является владельцем монитора объекта, и **InterruptedException** в случае, если любой поток прервёт текущий поток до или во время того, так текущий поток ждёт уведомления | | |
|  | | | | | | Вынуждает текущий поток подождать, пока другой поток вызывает метод notify() или notifyAll() или ждёт определённое количество времени.  Может выбпасить **InterruptedException** в случае, если любой поток прервёт текущий поток до или во время того, так текущий поток ждёт уведомления | | |
|  | | | | | | | Вынуждает текущий поток подождать, пока другой поток вызывает метод notify() или notifyAll(), или некоторый другой поток прерывает текущий, или ждёт определённое количество времени.  Может выбпасить **IllegalMonitorStateException** в случае, если текущий поток не является владельцем монитора объекта, и **InterruptedException** в случае, если любой поток прервёт текущий поток до или во время того, так текущий поток ждёт уведомления, и **IllegalArgumentException** в случае, если значение timeout отрицательное или значение nanos не принадлежит промежутку от 0 до 999999 | |

*Контракт между equals & hashCode*

Если equals возвращает true, то hashCode должен вернуть одно и то же значение. Обратное не верно.

*Использование prime=31 в hashCode*

Во-первых, это простое число. Хешкод, сгенерированный умножением на простое число, даёт возможность чутко реагировать на изменение состояния объекта, то есть изменять хешкод тоже, и даёт нормальное распределение, то есть меньше коллизий . Во-вторых, умножение на 31 (= 24 - 1) представляет собой сдвиг на 5 байтов, следовательно, умножение не является тяжёлой операцией.

*Что будет, если hasCode переопределить таким образом, чтобы он возвращал всегда единицу?*

*Почему в Object есть методы, связанные с многопоточностью? Почему бы их все не положить в класс Thread?*

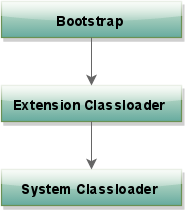
Одна из причин заключается в том, что Java обеспечивает блокировку на уровне объекта, а не на уровне потоков. Каждый объект имеет блокировку, которая приобретается потоком. Таким образом, если поток должен ожидать блокировки, то есть смысл вызвать метод wait на объекте, а не на потоке. Если бы wait метод был объявлен в классе Thread, то не было бы ясно

**ClassLoading**

*Что такое ClassLoader?*

ClassLoader - часть jre которая занимается динамической загрузкой классов.

ClassLoader - это стандартный абстрактный java класс java.lang.ClassLoader. Предназначен для загрузки классов в память JVM и помещения их в собственный кэш. Но сам, непосредственно, их не загружает, это делают его наследники.

*Виды ClassLoader*

Bootstrap ClassLoader - базовый загрузчик;

Extension Classloader - загрузчик расширений;

System Classloader - системный загрузчик.ды загрузчиков

Загрузчики классов образуют иерархию, корнем которой является базовый загрузчик. Все остальные загрузчики при инициализации сохраняют ссылку на родительский загрузчик. Таким образом достигается реализация модели делегирования загрузки.

Право загрузки класса рекурсивно делегируется от самого нижнего загрузчика в иерархии к самому верхнему. Такой подход позволяет загружать классы тем загрузчиком, который максимально близко находится к базовому. Так достигается максимальная область видимости классов. Под областью видимости подразумевается следующее: каждый загрузчик ведет учет классов, которые были им загружены. Множество этих классов и называется областью видимости. При этом загрузчик видит только «свои» классы и классы «родителя» и понятия не имеет о классах, которые были загружены его «потомком»

• **Bootstrap ClassLoader** реализован на уровне JVM и не имеет обратной связи со средой исполнения. Данным загрузчиком загружаются классы из директории $JAVA\_HOME/lib и все базовые классы. Поэтому, попытка получения загрузчика у классов java.\* всегда заканчивается null'ом. Но если очень хочется, то управлять загрузкой базовых классов можно с помощью ключа -Xbootclasspath, который позволяет переопределять наборы базовых классов.

• **Extension Classloader** загружает классы из директории $JAVA\_HOME/lib/ext. В Sun JRE — это класс sun.misc.Launcher$ExtClassLoader. Управлять загрузкой расширений можно с помощью системной опции java.ext.dirs.

• **System Classloader** реализованный уже на уровне JRE. В Sun JRE — это класс sun.misc.Launcher$AppClassLoader. Этим загрузчиком загружаются классы, пути к которым указаны в переменной окружения CLASSPATH. Управлять загрузкой системных классов можно с помощью ключа -classpath или системной опцией

*Процесс создания класса*

После того, как байт-код для запрошенного класса был найден, необходимо на его основе создать класс - получить полноценный объект, представляющий Java-класс. Для этого используется метод [ClassLoader.defineClass()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html#defineClass%28java.lang.String,%20byte%5B%5D,%20int,%20int%29).

Процесс конструирования класса состоит из ряда последовательных фаз:

* Верификация байт-кода (verification)  
    
  Проверка переданного кода на соответствие ряду зачастую нетривиальных требований определенных в [спецификации JVM](http://java.sun.com/docs/books/vmspec/).
* Предварительная подготовка к загрузке (preparation)  
    
  Создание и инициализация необходимых структур, используемых для представления полей, методов, реализованных интерфейсов и т.п., определенных в загружаемом классе.
* Анализ зависимостей (resolution)  
    
  Загрузка набора классов, на которые ссылается загружаемый класс. Например:
  + родительский класс
  + реализованные интерфейсы
  + поля класса
  + типы, используемые в сигнатурах методов класса
  + локальные переменные в методах класса

Точный список зависит от конкретной реализации виртуальной машины. На данном этапе могут быть загружены как все классы, на которые тем или иным способом ссылается создаваемый класс, так и только их часть, если используется «ленивый» (lazy) алгоритм загрузки. В последнем случае, оставшиеся классы будут подгружаться по мере необходимости.

* Инициализация  
    
  Вызов статических блоков инициализации и присваивание полям класса значений по-умолчанию.

По завершению всех перечисленных фаз, класс полностью готов к использованию.

*Особенности динамической загрузки классов*

Динамическая загрузка классов в Java имеет ряд особенностей:

* отложенная (lazy) загрузка и связывание классов  
  Загрузка классов производится только при необходимости, что позволяет экономить ресурсы и распределять нагрузку.
* проверка корректности загружаемого кода (type safeness)  
  Все действия, связанные с контролем использования типов, производятся только во время загрузки класса, позволяя избежать дополнительной нагрузки во время выполнения кода.
* программируемая загрузка  
  Пользовательский загрузчик полностью контролирует процесс получения запрошенного класса—самому ли искать байт-код и создавать класс или делегировать создание другому загрузчику. Дополнительно существует возможность выставлять различные атрибуты безопасности для загружаемых классов, позволяя таким образом работать с кодом из ненадежных источников.
* множественные пространства имен  
  Каждый загрузчик имеет своё пространство имён для создаваемых классов. Соответственно, классы, созданные двумя различными загрузчиками на основе общего байт-кода, в системе будут различаться.

*Исключения*

[**ClassCastException**](https://www.ibm.com/support/knowledgecenter/SSEQTP_8.5.5/com.ibm.websphere.nd.doc/ae/rtrb_classload_viewer.html?lang=en-us#rtrb_classload_viewer__ccexc)

• Тип исходного объекта не является экземпляром выбранного класса (типа).

• ClassLoader, который загрузил исходный объект (класс) отличается от загрузчика классов, который загрузил выбранный класс.

• Приложение не выполняет или ненадлежащим образом выполняет узкую операцию

**ClassNotFoundException**, бросается, когда приложение пытается загрузиться класс по его названию (String) с помощью таких средств:

* forName метод в классе Class.
* findSystemClass метод в классе ClassLoader.
* loadClass метод в классе ClassLoader. Но класса с таким именем не существует.

**NoClassDefFoundError**, бросается в таких случаях:

* Когда архив, директория, или другой источник необходимых классов не был добавлен в источники текущего загрузчика классов или его предка.
* Загрузчик-предок не был установлен корректно.

## UnsatisfiedLinkError • Действие пользователя вызвало ошибку.

• System.mapLibraryName возвращает неправильный файл библиотеки.

• Native библиотека уже загружена.

*Выгрузка классов*

В общем случае, класс может быть выгружен только тогда, когда в приложении он более не используется. Конкретная же политика выгрузки классов во многом зависит от реализации виртуальной машины и в дальнейшем будет описываться поведение Sun HotSpot JVM.

Загруженные классы, несмотря на то, что являются полноценными Java-объектами, хранятся в особой системной области памяти, называемой permament generation (сокращенно, PermGen) и управляемой сборщиком мусора. Соответственно, класс будет выгружен только тогда, когда на него не осталось ссылок. В случае, если класс был создан пользовательским загрузчиком, прямая ссылка на него храниться загрузчиком и класс может быть выгружен только после успешной выгрузки загрузчика. Это правило распространяется и на загрузчики, управляемые системой. *Соответственно, классы, созданные базовым загрузчиком, не могут быть выгружены в принципе*. На практике это верно и для системного загрузчика с загрузчиком расширений—их выгрузка во время работы приложения не предусмотрена.

Как происходит поиск байт-кода загрузчиком

Во время загрузки, поиск запрошенного класса производится в следующей последовательности:

* поиск в списке ранее загруженных классов  
    
  Проверяется запрашивался ли ранее данный класс, и, если да, возвращается тот же самый класс, что и ранее.
* делегирование родительскому загрузчику  
    
  В случае, если класс ранее не был загружен, запрос на загрузку делегируется родительскому загрузчику. Это позволяет загружать классы тем загрузчиком, который находится ближе всего к базовому в иерархии делегирования.
* попытаться загрузить класс самому  
    
  Если родительский загрузчик не смог загрузить запрошенный класс, текущий загрузчик пытается сам произвести процесс загрузки требуемого класса: найти байт-код и на основе него создать требуемый класс.

Пользовательский загрузчик может не придерживаться данной последовательности шагов, но это чревато появлением трудноотловимых ошибок, связанных с ограничением области видимости загружаемых классов.

Как следствие, все системные классы загружаются базовым загрузчиком (например, [java.lang.Object](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html), [java.lang.String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) и т.д.). Более того, ряд системных классов, по соображениям безопасности, могут быть загружены только лишь базовым загрузчиком - любые попытки создать один из таких классов другим загрузчиком завершатся неудачно.

Загрузчик, подчиняющийся данной модели делегирования, имеет доступ только к тем классам, которые загружены им самим или другими загрузчиками, находящимися выше него в цепочке делегирования.

С учетом правил делегирования, класс может быть создан загрузчиком, отличным от того: который инициировал его загрузку. Поэтому для каждого конкретного класса особое значение имеют два загрузчика—инициировавший загрузку (initiating loader) и непосредственно создавший его (defining loader).

*Пример*

Пусть в системе исполнения встретилась декларация переменной пользовательского класс Student.  
1) Системный загрузчик попытается поискать в кеше класс Student.  
 1.1) Если класс найден, загрузка окончена.  
 1.2) Если класс не найден, загрузка делегируется загрузчику расширений.  
2) Загрузчик расширений попытается поискать в кеше класс Student.  
 2.1) Если класс найден, загрузка окончена.  
 2.2) Если класс не найден, загрузка делегируется базовому загрузчику.  
3) Базовый загрузчик попытается поискать в кеше класс Student.  
 3.1) Если класс найден, загрузка окончена.  
 3.2) Если класс не найден, базовый загрузчик попытается его загрузить.  
 3.2.1) Если загрузка прошла успешно, она закончена ;)  
 3.2.2) Иначе управление предается загрузчику раширений.  
 3.3) Загрузчик расширений пытается загрузить класс.   
 3.3.1) Если загрузка прошла успешно, она закончена ;)  
 3.3.2) Иначе управление предается системному загрузчику.  
 3.4) Системный загрузчик пытается загрузить класс.   
 3.4.1) Если загрузка прошла успешно, она закончена ;)  
 3.4.2) Иначе генерируется исключение java.lang.ClassNotFoundException.

**OOP Principles**

Абстракция – это выделение общих характеристик объекта, исключая набор.

Полиморфизм - возможность повторного использования имени в системе.

Полиморфизм - возможность использования родительского класса без создания объекта класса.

Полиморфизм – это свойство, которое позволяет одно и то же имя использовать для решения двух и более схожих, но технически различных задач.

Наследование - возможность создание класса на основе уже существующего.

Наследование(inheritance) - это процесс, посредством которого, один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Инкапсуляция - объединение данных и методов в единую оболочку с целью защиты и сокрытия реализации от пользователя.

Инкапсуляция(encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищающий и то, и другое от внешнего вмешательства или неправильного использования.

Позднее связывание (динамическое) – связывание, при котором ассоциация между ссылкой и классом не устанавливается, пока объект с заданным именем не будет создан на стадии выполнении программы.

Раннее связывание – ассоциация во время компиляции.