**LESSON 3**

**Object**

«Объект» не имеет конечного понятия. Объект ООП – это совокупность переменных состояния и связанных с ним методов (операций). Эти методы показывают, как данный объект взаимодействует с окружающей системой. Или объект – это конкретная реализация класса.

*Пример*

Другими словами, решать бизнес задачи проще, используя ООП. ООП позволяет разложить проблему на связанные между собой задачи. Каждая проблема становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом случае исходная задача в целом упрощается, и программист получает возможность оперировать с гораздо большими по объему программами

*Способы создания объектов?*

|  |  |  |
| --- | --- | --- |
| С помощью оператора new | Когда применяется? | Для создания объекта с помощью конструктора. |
| 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 | Когда применяется? | Если необходимо создать быстро копию объекта или нет public API. |
| 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. Та же ошибка будет, если переопределён метод, но не реализован интерфейс.     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 побитово. 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, например, 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 | Когда применяется? | 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 ошибка java.io.NotSerializableException   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   **Теория (конспект Ольги) смотри после таблицы** |
| Рефлексия | Когда применяется? | Рефлексия - это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов, выполнять операции над полями и методами. Рефлексия в Java осуществляется с помощью Java Reflection API. Этот интерфейс API состоит из классов пакетов java.lang и java.lang.reflect. Одна из возможностей интерфейса Java Reflection API является создание объектов классов.  Применяется в случае, если мы знаем только имя класса и параметры. |
| Пример |  |
| Какие механизмы срабатывают? | 1. На классе Class вызываем метод forName, который возвращает объект Class, ассоциированный с классом или интерфейсом, имя которого было задано как параметр. Имя класса – полное имя описываемого класса (java.lang.Thread) 2. Загрузка класса 3. На объекте класса Class вызываем метод newInstance, который возвращает новый экземпляр заданного класса. Экземпляр класса создастся как будто был вызван *конструктор без аргументов*. Класс инициализируется, если он не был ещё инициализирован. Если класс или его конструктор без параметров недоступны, то будет выброшено исключение IllegalAccessException. Если попытаемся создать экземпляр абстрактного класса или интерфейса, или массива, или примитивного типа, или нет конструктора без параметров, то будет выброшено исключение InstantiationException 4. Явное приведение типов |

**Рефлексия**

Рефлексия - это механизм исследования данных о программе во время её выполнения.

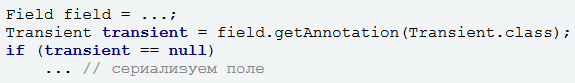
Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов, выполнять операции над полями и методами. Рефлексия в Java осуществляется с помощью Java Reflection API. Этот интерфейс API состоит из классов пакетов java.lang и java.lang.reflect.

Применяется в случае, если мы знаем только имя класса и параметры.

Практические аспекты:

1. С введением аннотаций в java 1.5 стало легко добавлять метаинформацию к классам, полям, методам и параметрам. По наличию аннотации (или ее отсутствию) можно во время выполнения программы принимать различного рода решения.

Допутим, мы пишем свой модуль для сериализации объектов. Пусть некоторая описанная нами аннотация @Transient говорит о том, что конкретное поле не нужно сериализовать. Нам поможет следующий фрагмент кода:



1. Продлжая тему о сервлетах. В веб-фреймворке, который писали мои друзья, используется следующий контракт: действие представляет собой вызов некоторого метода someMethod у некоторого класса SomeClass. При этом url имеет вид, например, somedomain.com/SomeAction.someMethod.

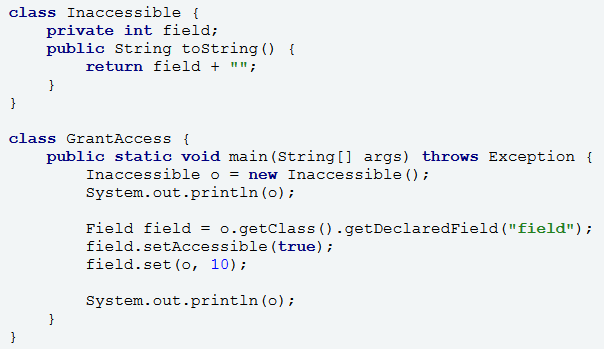
Вот небольшой фрагмент кода, который показывает, как можно вызвать метод у класса используя reflection. Пример очень упрощен в обучающих целях:





1. Часто программистам приходится работать с чужими проприетарными библиотеками. Случается, что где-то в библиотеке значения по-умолчанию некоторых полей вас не устраивают, но изменить их через API нет возможности. Или некоторый метод закрыт модификатором доступа private.

Вместо декомпиляции-исправления-компиляции есть способ, который иногда выручает. Это установка флага acccessible. Давайте установим значение 10 для поля field класса Inaccessible:



**Сериализация и десериализация**

Сериализация – процесс сохранения состояния объекта в последовательность байт.

Десериализация – процесс восстановления объекта из этих байт.

Java Serialization API предоставляет стандартные механизмы для создания сериализуемых объектов. То есть класс разбирается как набор полей, каждое из которых пишется в выходной поток.

Процесс сериализации выглядит так:

1. Сериализуется каждое поле
2. Transient не сериализуется и получает значение по умолчанию при десериализации
3. Static не сериализуется и может получить значение по умолчанию или принять существующее значение при десериализации

Процесс десериализации выглядит так:

1. Выделяется память под новый объект
2. Восстанавливаются все поля (ссылки)
3. Значение поля transient получает значение по умолчанию
4. Значение поля static получает значение по умолчанию, если в памяти не хранится объект данного класса, иначе получает значение из этого класса.

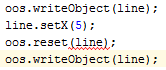
Сериализация нарушает ООП. Объект создаётся без конструкора.

Для сериализации необходимо поле *private static final long serialVersionID*. Это единственное static поле, которое сериализуется. Если класс подвергся модификации (изменилась структура класса), то значение этого поле должно быть изменено. Так как при десериализации мы обращаемся к классу, хранящемся в памяти, но создать объект не сможем, так как *serialVersionID* было изменено, и мы получим **InvalidClassException**.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь также могут хранить ссылки. При десериализации все ссылки должны быть восстановлены (всё сериализуется, независимо от глубины).

При сериализации объект помечается, если уже был сериализован. Это нужно в том случае, если другой объект тоже ссылается на этот объект. Вместо повторной сериализации вставляется ссылка. При десериализации архитектура сохраняется.

Дозаписывать в байт файл после сериализации нельзя. Но можно исправить ситуацию следующим образом:



При десериализации производного класса, наследуемого от несериализуемого, вызывается конструктор **без параметров** родительского, а конструктор производного вызываться не будет.

Сериализацию и десериализацию можно осуществить **двумя** способами:

|  |  |  |
| --- | --- | --- |
| Что должен реализовать класс? | java.io.Serializable  Интерфейс-маркер | java.io.Externalizable  Он содержит два метода, которые необходимо реализовать – writeExternal(ObjectOutput) и readExternal(ObjectInput). В этих методах как раз и находится логика сериализации и десериализации. |
| Родитель должен быть сериализуемым? | Совершенно необязательно. |  |
| Как проходит десериализация? | Под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.  При десериализации вызывается конструктор без параметров родительского НЕсериализуемого класса. И если такого конструктора не будет – при десериализации возникнет ошибка. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается, как и было сказано выше. | Сначала вызывается конструктор без параметров, а потом уже на созданном объекте вызывается метод readExternal, который и вычитывает, собственно, все свои данные.  Потому – любой реализующий интерфейс Externalizable класс обязан иметь public конструктор без параметров! Более того, поскольку все наследники такого класса тоже будут считаться реализующими интерфейс Externalizable, у них тоже должен быть конструктор без параметров! |
| Transient поле | Не сериализуется.  При десериализации – значение по умолчанию. | Сериализуется.  При десериализации – значение по умолчанию. |
| Static поле | Не сериализуется. | Можно сериализовать, но не рекомендуется. |
| Final поле | Сериализуются и десериализуется | Не десериализуется. Ибо final-поля должны быть инициализированы в конструкторе. |
| serialVersionID | Есть (=равным дате модификации кода)  Для получения стандартного значения (того, которое вычисляется внутренним механизмом) можно использовать утилиту serialver, входящую в поставку SDK.  Крайне рекомендуется всем сериализуемым классам декларировать это поле в явном виде, ибо вычисление по умолчанию очень чувствительно к деталям структуры класса, которые могут различаться в зависимости от реализации компилятора, и вызывать таким образом неожиданные InvalidClassException при десериализации. Объявлять это поле лучше как private, т.к. оно относится исключительно к классу, в котором объявляется. Хотя в спецификации модификатор не оговорен. | Нет |
| Наследование | При наследовании от класса, реализующего Serializable, никаких дополнительных действий предпринимать не надо. Сериализация будет распространяться и на дочерний класс | При наследовании от класса, реализующего Externalizable, необходимо переопределить методы родительского класса readExternal и writeExternal. Иначе поля дочернего класса сериализованы не будут. В этом случае надо бы не забыть вызвать родительские методы, иначе не сериализованы будут уже родительские поля. |
| Что будет , если реализовать и Serializable, и Externalizable? | Выиграет Externalizable | |

*Зачем нужен Externalizable?*

Во-первых, она дает гораздо большую гибкость (мы можем управлять процессами сериализации и десериализации как хотим, что делает нас независимыми от любых изменений в классе). Во-вторых, зачастую она может дать немалый выигрыш по объему сериализованных данных. В-третьих, существует такой аспект как производительность.

*Почему Cloneable и Serializable не реализованы в Object?*

Для безопасности, иначе можно легко создать объект и использовать в своих целях. Создать «свой» connection pool и доставать connection.

*Маркерные интерфейсы*

Плюсы: возможность внести информацию о дополнительном поведении класса.

Минусы: если базовый класс реализует интерфейс, то и производные классы тоже будут его реализовывать (нет «анти-маркеров»).

**Class Object**

*Где лежит?*

java.lang.Object

*Кого реализует или наследует?*

Никого

*Назначение и использование данного класса?*

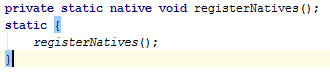
Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

*Состояние/поведение данного класса?*

Состояние не имеет (нет полей).

Есть статический блок, который используется для вызова статической функции registerNatives.

Для того, чтобы JVM нашла ваши нативные функции, они должны быть проименованы определённым образом. Например, этой функции java.lang.Object.registerNatives соответствует C функция Java\_java\_lang\_Object\_registerNatives. Используя функцию registerNatives, вы можете именовать C функции как хотите.



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.

Все поля, которые участвуют в определении hashCode, должны участвовать в определении equals.

*Использование prime=31 в hashCode*

1. 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 .
2. Умножение числа на 31 (= 24 - 1) представляет собой последовательность операций: сдвиг на 5 байтов влево и вычитание этого же числа, следовательно, умножение не является тяжёлой операцией.



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

Получим огромное число коллизий.

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

Так как wait, notify и noyifyAll работают на уровне блокировок (lock), имеет смысл определить их в классе Object, так как lock принадлежит объекту.

**ClassLoading**

*Что такое ClassLoader?*

СlassLoader - 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;

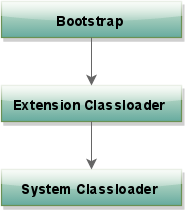
Extension Classloader - загрузчик расширений;

System Classloader - системный загрузчик.

• **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



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»

*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.

*Зачем нужна иерархия загрузчиков?*

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

*Зачем нужно делегирование?*

Для расширения области видимости класса и для безопасности, а именно: если попытаться загрузить свой класс java.lang.Integer, то попытка завершится провалом, так как такой класс с таким путем будет уже находится в кэше базового класса.

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. Так же базовый загрузчик создаёт объекты Extension и System ClassLoader, а они подгружают нужные классы по мере необходимости.

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

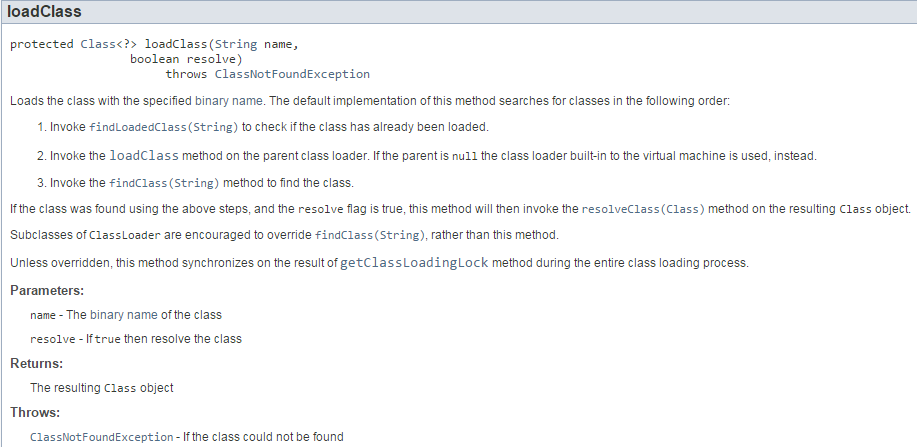
С учетом правил делегирования, класс может быть создан загрузчиком, отличным от того: который инициировал его загрузку. Поэтому для каждого конкретного класса особое значение имеют два загрузчика—инициировавший загрузку (initiating loader) и непосредственно создавший его (defining loader).

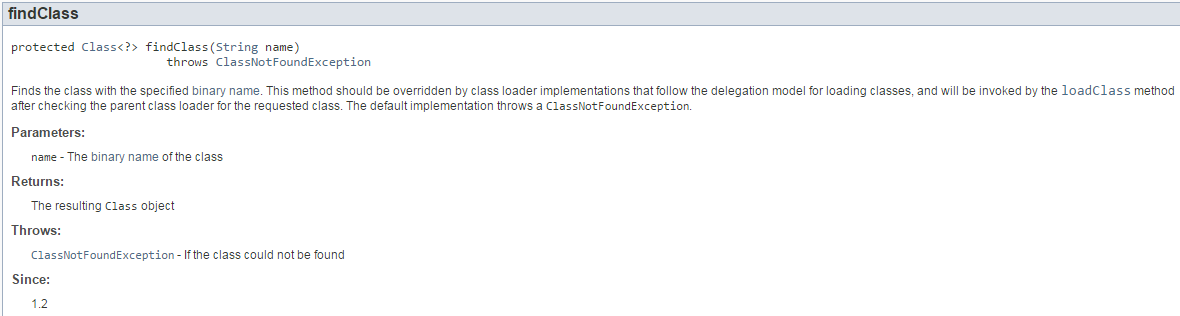
*Какие есть инструменты, чтобы посмотреть загрузку классов?*

Java -verbose -classpath ./bin com.qwertovsky.helloworld.HelloWorld

*Процесс создания класса*

|  |  |  |
| --- | --- | --- |
| **Loading** | **Linking** | **Initializing** |
| На вход приходит файл MyClass.class.  На данном этапе мы получаем поток байтов. | 1. Verify   Проверяем байт-код на magic\_number, sdk\_version. На этом этапе можем получить исключение (например не подходим под диапазон jdk) | Загрузка статических полей и блоков.  На выходе получаем объект класса Class |
| 1. Prepare   Инициализация структуры данных (типы данных). |
| 1. Resolve   (Возможно приостановит текущую загрузку) Загрузка ссылочных типов данных, родительствого класса. |







*Первое обращение к классу (active uses) это:*

1. Создание нового экземпляра класса

*Test test = new Test();*

1. Вызов статического метода

*Test.callToStaticMethod();*

1. Использование и присвоение значения статическому полю, объявленному в классе или интерфейсе, за исключением константных полей, которые были инициализированы во время компиляции
2. Вызов определённых reflection методов, таких как в классе Class или в классах in the java.lang.reflect package

*Class.forName(“Test”); test.getClass();*

1. Инициализация подкласса класса повлечёт за собой загрузку базового класса
2. Определение класса, как начального (содержащего метод main), при старте JVM.

*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.

*Как проверить, что класс загружен?*

If( test.getClass() == null ) {…}

*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.

**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.