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

Во время наследования возможно изменения модификаторов доступа в сторону большей видимости

## **О чем говорит ключевое слово final?**

* Параметры методов не могут изменять своё значение внутри метода;
* Локальные переменные не могут быть изменены после присвоения им значения.

## **Что вы знаете о функции main()?**

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

## **Какие побитовые операции вы знаете?**

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

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

Наследуясь от абстрактного, класс «растворяет» собственную индивидуальность. Реализуя интерфейс, он расширяет собственную функциональность.

Поэтому абстрактные классы применяются в случае построения иерархии однотипных, очень похожих друг на друга классов. В этом случае наследование от абстрактного класса, реализующего поведение объекта по умолчанию может быть полезно, так как позволяет избежать написания повторяющегося кода. Во всех остальных случаях лучше использовать интерфейсы.

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

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

## **Зачем нужны и какие бывают блоки инициализации?**

Блоки инициализации представляют собой код, заключенный в фигурные скобки и размещаемый внутри класса вне объявления методов или конструкторов.

* Существуют статические и нестатические блоки инициализации.
* Блок инициализации выполняется перед инициализацией класса загрузчиком классов или созданием объекта класса с помощью конструктора.
* Несколько блоков инициализации выполняются в порядке следования в коде класса.
* Блок инициализации способен генерировать исключения, если их объявления перечислены в throws всех конструкторов класса.
* Блок инициализации возможно создать и в анонимном классе.

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

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

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

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

## **Может ли статический метод быть переопределён или перегружен?**

Перегружен - да. Всё работает точно так же, как и с обычными методами - 2 статических метода могут иметь одинаковое имя, если количество их параметров или типов различается.

Переопределён - нет. Выбор вызываемого статического метода происходит при раннем связывании (на этапе компиляции, а не выполнения) и выполняться всегда будет родительский метод, хотя синтаксически переопределение статического метода - это вполне корректная языковая конструкция.

## **Могут ли нестатические методы перегрузить статические?**

Да. В итоге получится два разных метода. Статический будет принадлежать классу и будет доступен через его имя, а нестатический будет принадлежать конкретному объекту и доступен через вызов метода этого объекта.

## **Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?**

При переопределении метода сужать модификатор доступа не разрешается

Изменять тип возвращаемого значения при переопределении метода разрешено только в сторону сужения типа (вместо родительского класса - наследника).

При изменении типа, количества, порядка следования аргументов вместо переопределения будет происходить overloading (перегрузка) метода.

## **Можно ли объявить метод абстрактным и статическим одновременно?**

Нет. В таком случае компилятор выдаст ошибку: "Illegal combination of modifiers: ‘abstract’ and ‘static’". Модификатор abstract говорит, что метод будет реализован в другом классе, а static наоборот указывает, что этот метод будет доступен по имени класса.

## **В чем разница между членом экземпляра класса и статическим членом класса?**

Модификатор static говорит о том, что данный метод или поле принадлежат самому классу и доступ к ним возможен даже без создания экземпляра класса. Поля, помеченные static инициализируются при инициализации класса. На методы, объявленные как static, накладывается ряд ограничений:

* Они могут вызывать только другие статические методы.
* Они должны осуществлять доступ только к статическим переменным.
* Они не могут ссылаться на члены типа this или super.

В отличии от статических, поля экземпляра класса принадлежат конкретному объекту и могут иметь разные значения для каждого. Вызов метода экземпляра возможен только после предварительного создания объекта класса.

## **Где разрешена инициализация статических/нестатических полей?**

* Статические поля можно инициализировать при объявлении, в статическом или нестатическом блоке инициализации.
* Нестатические поля можно инициализировать при объявлении, в нестатическом блоке инициализации или в конструкторе.

## **Какие типы классов бывают в java?**

* Top level class (Обычный класс):
  + Abstract class (Абстрактный класс);
  + Final class (Финализированный класс).
* Interfaces (Интерфейс).
* Enum (Перечисление).
* Nested class (Вложенный класс):
  + Static nested class (Статический вложенный класс);
  + Member inner class (Простой внутренний класс);
  + Local inner class (Локальный класс);
  + Anonymous inner class (Анонимный класс).

## **Расскажите про вложенные классы. В каких случаях они применяются?**

Класс называется вложенным (Nested class), если он определен внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать обрамляющий его класс.

## **Какие существуют особенности использования вложенных классов: статических и внутренних? В чем заключается разница между ними?**

* Вложенные классы могут обращаться ко всем членам обрамляющего класса, в том числе и приватным.
* Для создания объекта статического вложенного класса объект внешнего класса не требуется.
* Из объекта статического вложенного класса нельзя обращаться к не статическим членам обрамляющего класса напрямую, а только через ссылку на экземпляр внешнего класса.
* Обычные вложенные классы не могут содержать статических методов, блоков инициализации и классов. Статические вложенные классы - могут.
* В объекте обычного вложенного класса хранится ссылка на объект внешнего класса. Внутри статической такой ссылки нет. Доступ к экземпляру обрамляющего класса осуществляется через указание .this после его имени. Например: Outer.this.

## **Что такое «локальный класс»? Каковы его особенности?**

****Local inner class**** (Локальный класс) - это вложенный класс, который может быть декларирован в любом блоке, в котором разрешается декларировать переменные. Как и простые внутренние классы (Member inner class) локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр только тогда, когда применяются в нестатическом контексте.

Локальные классы имеют следующие особенности:

* Видны только в пределах блока, в котором объявлены;
* Не могут быть объявлены как private/public/protected или static;
* Не могут иметь внутри себя статических объявлений методов и классов, но могут иметь финальные статические поля, проинициализированные константой;
* Имеют доступ к полям и методам обрамляющего класса;
* Могут обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/core.md" \l "java-core)

## **Что такое «анонимные классы»? Где они применяются?**

Это вложенный локальный класс без имени, который разрешено декларировать в любом месте обрамляющего класса, разрешающем размещение выражений. Создание экземпляра анонимного класса происходит одновременно с его объявлением. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический вложенный класс - в нестатическом контексте появляется окружающий его экземпляр.

Анонимные классы имеют несколько ограничений:

* Их использование разрешено только в одном месте программы - месте его создания;
* Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;
* Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

* создания объекта функции (function object), например, реализация интерфейса Comparator;
* создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных;
* в статическом методе генерации;
* инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

## **Каким образом из вложенного класса получить доступ к полю внешнего класса?**

Статический вложенный класс имеет прямой доступ только к статическим полям обрамляющего класса.

Простой внутренний класс, может обратиться к любому полю внешнего класса напрямую. В случае, если у вложенного класса уже существует поле с таким же литералом, то обращаться к такому полю следует через ссылку на его экземляр. Например: Outer.this.field.

## **Для чего используется оператор assert?**

****Assert**** (Утверждение) — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Утверждение может автоматически сигнализировать об обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.

Обычно утверждения оставляют включенными во время разработки и тестирования программ, но отключают в релиз-версиях программ.

## **Что такое Heap и Stack память в Java? Какая разница между ними?**

****Heap (куча)**** используется Java Runtime для выделения памяти под объекты и классы. Создание нового объекта также происходит в куче. Это же является областью работы сборщика мусора. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться из любой части приложения.

****Stack (стек)**** это область хранения данных также находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO (Последний-зашел-Первый-вышел)РЕКУРСИЯ

Различия между Heap и Stack памятью:

* Куча используется всеми частями приложения в то время как стек используется только одним потоком исполнения программы.
* Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится лишь ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче.
* Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков.
* Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы.
* Если память стека полностью занята, то Java Runtime бросает исключение java.lang.StackOverflowError. Если заполнена память кучи, то бросается исключение java.lang.OutOfMemoryError: Java Heap Space.
* Размер памяти стека намного меньше памяти в куче.
* Из-за простоты распределения памяти, стековая память работает намного быстрее кучи.

## **Верно ли утверждение, что примитивные типы данных всегда хранятся в стеке, а экземпляры ссылочных типов данных в куче?**

Не совсем. Примитивное поле экземпляра класса хранится не в стеке, а в куче. Любой объект (всё, что явно или неявно создаётся при помощи оператора new) хранится в куче

## **Каким образом передаются переменные в методы, по значению или по ссылке?**

В Java параметры всегда передаются только по значению, что определяется как «скопировать значение и передать копию». С примитивами это будет копия содержимого. Со ссылками - тоже копия содержимого, т.е. копия ссылки. При этом внутренние члены ссылочных типов через такую копию изменить возможно, а вот саму ссылку, указывающую на экземпляр - нет.

## **Для чего нужен сборщик мусора?**

Сборщик мусора (Garbage Collector) должен делать всего две вещи:

* Находить мусор - неиспользуемые объекты. (Объект считается неиспользуемым, если ни одна из сущностей в коде, выполняемом в данный момент, не содержит ссылок на него, либо цепочка ссылок, которая могла бы связать объект с некоторой сущностью приложения, обрывается);
* Освобождать память от мусора.

## **Как работает сборщик мусора?**

Механизм сборки мусора - это процесс освобождения места в куче, для возможности добавления новых объектов.

Объекты создаются посредством оператора new, тем самым присваивая объекту ссылку. Для окончания работы с объектом достаточно просто перестать на него ссылаться, например, присвоив переменной ссылку на другой объект или значение null; прекратить выполнение метода, чтобы его локальные переменные завершили свое существование естественным образом. Объекты, ссылки на которые отсутствуют, принято называть мусором (garbage), который будет удален.Мусор собирается системой автоматически, без вмешательства пользователя или программиста,

## **Что такое «пул строк»?**

****Пул строк**** – это набор строк, хранящийся в Heap.

* Пул строк возможен благодаря неизменяемости строк в Java и реализации идеи интернирования строк;
* Пул строк помогает экономить память, но по этой же причине создание строки занимает больше времени;
* Когда для создания строки используются ", то сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка на неё;
* При использовании оператора new создаётся новый объект String. Затем при помощи метода intern() эту строку можно поместить в пул или же получить из пула ссылку на другой объект String с таким же значением;

## **Что такое finalize()? Зачем он нужен?**

наследуется от Java.lang.Object

переопределяя finalize() можно запрограммировать действия необходимые для корректного удаления экземпляра класса - например, закрытие сетевых соединений, соединений с базой данных, снятие блокировок на файлы и т.д.

Вызов этого метода не гарантируется, т.к. приложение может быть завершено до того, как будет запущена сборка мусора.

System.gc(); // Принудительно зовём сборщик мусора

## **Что произойдет со сборщиком мусора, если выполнение метода finalize() требует ощутимо много времени, или в процессе выполнения будет выброшено исключение?**

Непосредственно вызов finalize() происходит в отдельном потоке Finalizer (java.lang.ref.Finalizer.FinalizerThread), который создаётся при запуске виртуальной машины (в статической секции при загрузке класса Finalizer). Методы finalize() вызываются последовательно в том порядке, в котором были добавлены в список сборщиком мусора. Соответственно, если какой-то finalize() зависнет, он подвесит поток Finalizer, но не сборщик мусора. Это в частности означает, что объекты, не имеющие метода finalize(), будут исправно удаляться, а вот имеющие будут добавляться в очередь, пока поток Finalizer не освободится, не завершится приложение или не кончится память.

## **Расскажите про приведение типов. Что такое понижение и повышение типа?**

способ преобразования значения переменной одного типа в значение другого типа.

В Java существуют несколько разновидностей приведения:

* ****Тождественное (identity)****. Преобразование выражения любого типа к точно такому же типу всегда допустимо и происходит автоматически.
* ****Расширение (повышение, upcasting) примитивного типа (widening primitive)****. Означает, что осуществляется переход от менее емкого типа к более ёмкому. Например, от типа byte (длина 1 байт) к типу int (длина 4 байта). Такие преобразование безопасны в том смысле, что новый тип всегда гарантировано вмещает в себя все данные, которые хранились в старом типе и таким образом не происходит потери данных. Этот тип приведения всегда допустим и происходит автоматически.
* ****Сужение (понижение, downcasting) примитивного типа (narrowing primitive)****. Означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа int было больше 127, то при приведении его к byte значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, при этом все старшие биты, не умещающиеся в новом типе, просто отбрасываются - никакого округления или других действий для получения более корректного результата не производится.
* ****Расширение объектного типа (widening reference)****. Означает неявное восходящее приведение типов или переход от более конкретного типа к менее конкретному, т.е. переход от потомка к предку. Разрешено всегда и происходит автоматически.
* ****Сужение объектного типа (narrowing reference)****. Означает нисходящее приведение, то есть приведение от предка к потомку (подтипу). Возможно только если исходная переменная является подтипом приводимого типа. При несоответствии типов в момент выполнения выбрасывается исключение ClassCastException. Требует явного указания типа.
* ****Преобразование к строке (to String)****. Любой тип может быть приведен к строке, т.е. к экземпляру класса String.
* ****Запрещенные преобразования (forbidden)****. Не все приведения между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся приведения от любого ссылочного типа к примитивному и наоборот (кроме преобразования к строке). Кроме того, невозможно привести друг к другу классы, находящиеся на разных ветвях дерева наследования и т.п.

При приведении ссылочных типов с самим объектом ничего не происходит, - меняется лишь тип ссылки, через которую происходит обращение к объекту.

Для проверки возможности приведения нужно воспользоваться оператором instanceof:

Parent parent = new Child();

if (parent instanceof Child) {

Child child = (Child) parent;

}

**Оператор** instanceof **нужен, чтобы проверить, был ли объект, на который ссылается переменная** X**, создан на основе какого-либо класса** Y(тип)**.**

Оператор instanceof сравнивает объект и указанный тип.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Cat cat = **new** Cat();

System.out.println(cat **instanceof** Animal);

System.out.println(cat **instanceof** MaineCoon);

}}

Вывод: true

false

Конструктор MaineCoon не вызывался при создании Cat, что логично. Ведь MaineCoon — потомок Cat, а не предок. Но шаблоном для Cat он не является.

## **Когда в приложении может быть выброшено исключение ClassCastException?**

ClassCastException (потомок RuntimeException) - исключение, которое будет выброшено при ошибке приведения типа.

|  |  |
| --- | --- |
| Кэширование —  простыми словами | размещение данных в специально отведенном месте для ускоренного доступа к ним в будущем. Примеры:   * Кэширование сайта — при посещении веб-страницы, [браузер](https://www.dmosk.ru/terminus.php?object=brouser) может сохранить данные на компьютере, чтобы потом их не скачивать второй раз. * Кэширование файла — процесс чтения/записи в специально отведенной памяти диска. Таким образом, при записи данные отправляются из оперативной память в системный кэш файла, а затем на сам диск. |

Хранение больших объёмов информации требует организации быстрого доступа к ней. Хэширование является одним из способов организации такого доступа. Основная идея хэширования состоит в вычислении по входным данным некоторого числа фиксированного размера, которое как бы будет указывать адрес данных в таблице. При этом такое число будет называться хэшем, а функция, с помощью которой оно вычисляется, - хэш-функцией.

Основное свойство всех хэш-функций - их необратимость, то есть по хэшу невозможно восстановить первоначальные данные, по которым он вычислен.

## **Какие есть особенности класса String?**

* Это неизменяемый (immutable) и финализированный тип данных;
* Все объекты класса String JVM хранит в пуле строк;
* Объект класса String можно получить, используя двойные кавычки;
* Можно использовать оператор + для конкатенации строк;
* Начиная с Java 7 строки можно использовать в конструкции switch.

## **Почему String неизменяемый и финализированный класс?**

Есть несколько преимуществ в неизменности строк:

* Пул строк возможен только потому, что строка неизменяемая, таким образом виртуальная машина сохраняет больше свободного места в Heap, поскольку разные строковые переменные указывают на одну и ту же переменную в пуле. Если бы строка была изменяемой, то интернирование строк не было бы возможным, потому что изменение значения одной переменной отразилось бы также и на остальных переменных, ссылающихся на эту строку.
* Если строка будет изменяемой, тогда это станет серьезной угрозой безопасности приложения. Например, имя пользователя базы данных и пароль передаются строкой для получения соединения с базой данных и в программировании сокетов реквизиты хоста и порта передаются строкой. Так как строка неизменяемая, её значение не может быть изменено, в противном случае злоумышленник может изменить значение ссылки и вызвать проблемы в безопасности приложения.
* Неизменяемость позволяет избежать синхронизации: строки безопасны для многопоточности и один экземпляр строки может быть совместно использован различными потоками.
* Строки используются classloader и неизменность обеспечивает правильность загрузки класса.
* Поскольку строка неизменяемая, её hashCode() кэшируется в момент создания и нет необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в HashMap т.к. его обработка происходит быстрее.

## **Почему char[] предпочтительнее String для хранения пароля?**

С момента создания строка остаётся в пуле, до тех пор, пока не будет удалена сборщиком мусора. Поэтому, даже после окончания использования пароля, он некоторое время продолжает оставаться доступным в памяти и способа избежать этого не существует. Это представляет определённый риск для безопасности, поскольку кто-либо, имеющий доступ к памяти сможет найти пароль в виде текста. В случае использования массива символов для хранения пароля имеется возможность очистить его сразу по окончанию работы с паролем, позволяя избежать риска безопасности, свойственного строке.

## **Почему строка является популярным ключом в HashMap в Java?**

Поскольку строки неизменяемы, их хэш код вычисляется и кэшируется в момент создания, не требуя повторного пересчета при дальнейшем использовании. Поэтому в качестве ключа HashMap они будут обрабатываться быстрее.

## **Что делает метод intern() в классе String?.**

Метод intern() используется для сохранения строки в пуле строк или получения ссылки, если такая строка уже находится в пуле.

## **Можно ли использовать строки в конструкции switch?**

Да, начиная с Java 7 в операторе switch можно использовать строки, ранние версии Java не поддерживают этого. При этом:

* участвующие строки чувствительны к регистру;
* используется метод equals() для сравнения полученного значения со значениями case, поэтому во избежание NullPointerException стоит предусмотреть проверку на null.
* согласно документации, Java 7 для строк в switch, компилятор Java формирует более эффективный байткод для строк в конструкции switch, чем для сцепленных условий if-else.

## **Какая основная разница между String, StringBuffer, StringBuilder?**

Класс String является неизменяемым (immutable) - модифицировать объект такого класса нельзя, можно лишь заменить его созданием нового экземпляра.

Класс StringBuffer изменяемый - использовать StringBuffer следует тогда, когда необходимо часто модифицировать содержимое.

Класс StringBuilder был добавлен в Java 5 и он во всем идентичен классу StringBuffer за исключением того, что он не синхронизирован (одновременно несколько потоков могут использовать класс)и поэтому его методы выполняются значительно быстрей.

## **Что такое класс Object? Какие в нем есть методы?**

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; int hashCode() – возвращает hash код для объекта; String toString() – возвращает строковое представление объекта; 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; void wait(long timeout) – остановка вызвавшего метод потока на определённое время или пока другой поток не вызовет метод notify() или notifyAll() для этого объекта; void wait(long timeout, int nanos) – остановка вызвавшего метод потока на определённое время или 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.

## **Чем отличаются конструктор по-умолчанию, конструктор копирования и конструктор с параметрами?**

У конструктора по умолчанию отсутствуют какие-либо аргументы. Конструктор копирования принимает в качестве аргумента уже существующий объект класса для последующего создания его клона. Конструктор с параметрами имеет в своей сигнатуре аргументы (обычно необходимые для инициализации полей класса).

## **Что такое Reflection?**

****Рефлексия (Reflection)**** - это механизм получения данных о программе во время её выполнения (runtime).

1. Можно узнать всю информацию о классе — методы, поля, конструкторы, константы, суперклассы, реализуемые классом интерфейсы.
2. Можно работать с классом (объектом), а именно — создать объект класса, выполнить методы, получить или установить значения полей.

## **Зачем нужен equals(). Чем он отличается от операции ==?**

Метод equals() - определяет отношение эквивалентности объектов.

При сравнении объектов с помощью == сравнение происходит лишь между ссылками. При сравнении по переопределённому разработчиком equals() - по внутреннему состоянию объектов.

## **Какая связь между hashCode() и equals()?**

Равные объекты должны возвращать одинаковые хэш коды. При переопределении equals() нужно обязательно переопределять и метод hashCode().

## **Что будет, если переопределить equals() не переопределяя hashCode()? Какие могут возникнуть проблемы?**

Классы и методы, которые используют правила этого контракта могут работать некорректно. Так для HashMap это может привести к тому, что пара «ключ-значение», которая была в неё помещена при использовании нового экземпляра ключа не будет в ней найдена.

## **Каким образом реализованы методы hashCode() и equals() в классе Object?**

Реализация метода Object.equals() сводится к проверке на равенство двух ссылок:

public boolean equals(Object obj) {

return (this == obj);

}

Реализация метода Object.hashCode() описана как native, т.е. определенной не с помощью Java кода и обычно возвращает адрес объекта в памяти:

public native int hashCode();

## **Для чего нужен метод hashCode()?**

Метод hashCode() необходим для вычисления хэш кода переданного в качестве входного параметра объекта. В Java это целое число, в более широком смыле - битовая строка фиксированной длины, полученная из массива произвольной длины. Этот метод реализован таким образом, что для одного и того же входного объекта, хэш код всегда будет одинаковым. Следует понимать, что в Java множество возможных хэш кодов ограничено типом int, а множество объектов ничем не ограничено. Из-за этого, вполне возможна ситуация, что хэш коды разных объектов могут совпасть:

* если хэш коды разные, то и объекты гарантированно разные;
* если хэш коды равны, то объекты могут не обязательно равны.

Метод hashCode() не гарантирует уникальность возвращаемого значения. Ситуация, когда у разных объектов одинаковые хэш коды называется коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хэш кода.

## **В чем разница между this.getClass() == that.getClass() и that instanceof MyClass?**

Оператор instanceof сравнивает объект и указанный тип.

Его можно использовать для проверки является ли данный объект экземпляром некоторого класса, либо экземпляром его дочернего класса, либо экземпляром класса, который реализует указанный интерфейс.

this.getClass() == that.getClass() проверяет два класса на идентичность, поэтому для корректной реализации контракта метода equals() необходимо использовать точное сравнение с помощью метода getClass().

## **Есть класс Point{int x, y;}. Почему хэш код в виде 31 \* x + y предпочтительнее чем x + y?**

Множитель создает зависимость значения хэш кода от очередности обработки полей, что в итоге порождает лучшую хэш функцию.

# Как правильно клонировать объект?

Использование оператора присваивания = не создает нового объекта, а лишь копирует ссылку на объект. Таким образом, две ссылки указывают на одну и ту же область памяти, на один и тот же объект. Для создания нового объекта с таким же состоянием используется клонирование объекта.

Класс Object содержит protected метод clone(), осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод clone() как public для обеспечения возможности его вызова. В переопределенном методе следует вызвать базовую версию метода super.clone(), которая и выполняет собственно клонирование.

Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс Cloneable. Интерфейс Cloneable не содержит методов относится к маркерным интерфейсам, а его реализация гарантирует, что метод clone() класса Object возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение CloneNotSupportedException. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора.

Это решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их обёрток или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым ссылочным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и сам объект поля класса.

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс Cloneable и переопределяет метод clone(). Так как, если это будет иначе вызов метода невозможен из-за его недоступности. Отсюда следует, что если класс имеет суперкласс, то для реализации механизма клонирования текущего класса-потомка необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений final для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

Помимо встроенного механизма клонирования в Java для клонирования объекта можно использовать:

* ****Специализированный конструктор копирования**** - в классе описывается конструктор, который принимает объект этого же класса и инициализирует поля создаваемого объекта значениями полей переданного.
* ****Фабричный метод**** - (Factory method), который представляет собой статический метод, возвращающий экземпляр своего класса.
* ****Механизм сериализации**** - сохранение и последующее восстановление объекта в/из потока байтов.

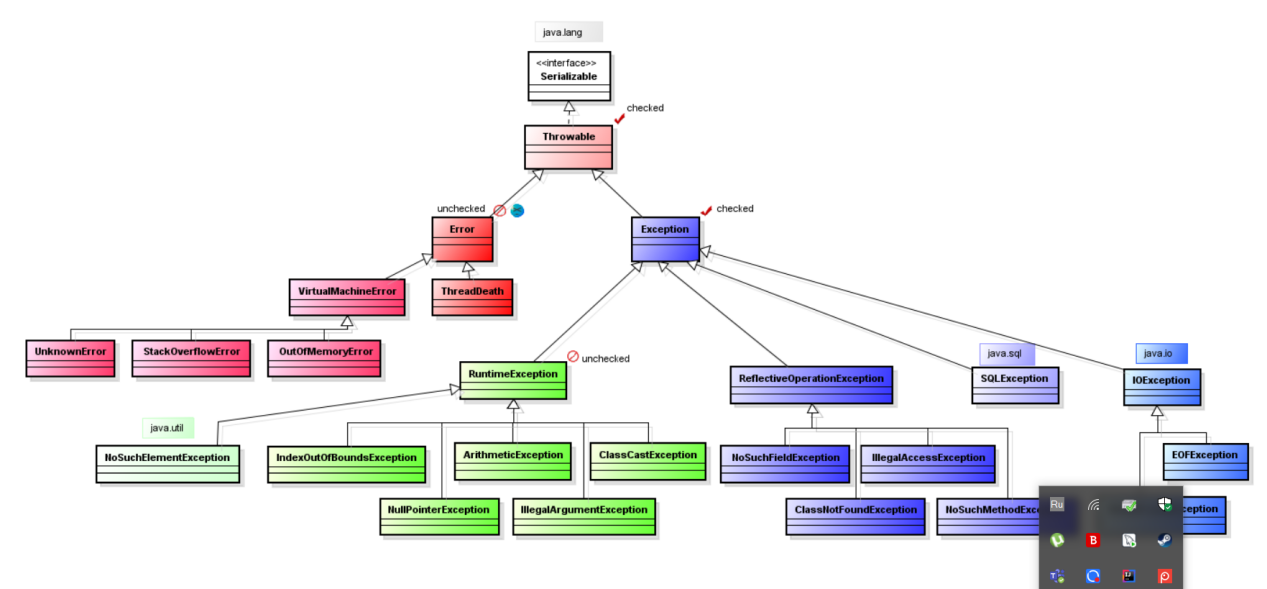
<https://habr.com/ru/post/246993/>

## **Опишите иерархию исключений.**

Исключения делятся на несколько классов, но все они имеют общего предка — класс Throwable, потомками которого являются классы Exception и Error.

****Ошибки (Errors)**** представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине.

****Исключения (Exceptions)**** являются результатом проблем в программе, которые в принципе решаемы, предсказуемы и последствия которых возможно устранить внутри программы. Например, произошло деление целого числа на ноль.



## **Что такое checked и unchecked exception?**

В Java все исключения делятся на два типа:

* ****checked (контролируемые/проверяемые исключения)**** должны обрабатываться блоком catch или описываться в сигнатуре метода (например, throws IOException). Наличие такого обработчика/модификатора сигнатуры проверяются на этапе компиляции;
* ****unchecked (неконтролируемые/непроверяемые исключения)****, к которым относятся ошибки Error (например, OutOfMemoryError), обрабатывать которые не рекомендуется и исключения времени выполнения, представленные классом RuntimeException и его наследниками (например, NullPointerException), которые могут не обрабатываться блоком catch и не быть описанными в сигнатуре метода.

## **Какой оператор позволяет принудительно выбросить исключение?**

Это оператор throw:

throw new Exception();

## **О чем говорит ключевое слово throws?**

Модификатор throws прописывается в сигнатуре метода и указывает на то, что метод потенциально может выбросить исключение с указанным типом.

## **Как написать собственное («пользовательское») исключение?**

Необходимо унаследоваться от базового класса требуемого типа исключений (например от Exception или RuntimeException).

class CustomException extends Exception {

public CustomException() {

super();

}

public CustomException(final String string) {

super(string + " is invalid");

}

public CustomException(final Throwable cause) {

super(cause);

}

}

## **Что представляет из себя ошибки класса Error?**

Ошибки класса Error представляют собой наиболее серьёзные проблемы уровня JVM. Например, исключения такого рода возникают, если закончилась память доступная виртуальной машине. Обрабатывать такие ошибки не запрещается, но делать этого не рекомендуется.

## **Что вы знаете о OutOfMemoryError?**

OutOfMemoryError выбрасывается, когда виртуальная машина Java не может создать (разместить) объект из-за нехватки памяти, а сборщик мусора не может высвободить достаточное её количество.

## **Что такое механизм try-with-resources?**

Данная конструкция, которая появилась в Java 7, позволяет использовать блок try-catch не заботясь о закрытии ресурсов, используемых в данном сегменте кода. Ресурсы объявляются в скобках сразу после try, а компилятор уже сам неявно создаёт секцию finally, в которой и происходит освобождение занятых в блоке ресурсов.

## **Возможно ли использование блока try-finally (без catch)?**

Такая запись допустима, но смысла в такой записи не так много, всё же лучше иметь блок catch, в котором будет обрабатываться необходимое исключение.

## **Всегда ли исполняется блок finally?**

Код в блоке finally будет выполнен всегда, независимо от того, выброшено исключение или нет.

[к оглавлению](https://github.com/enhorse/java-interview/blob/master/core.md" \l "java-core)

## **Существуют ли ситуации, когда блок finally не будет выполнен?**

Например, когда JVM «умирает» - в такой ситуации finally недостижим и не будет выполнен, так как происходит принудительный системный выход из программы:

try {

System.exit(0);

} catch(Exception e) {

e.printStackTrace();

} finally { }

## **Может ли метод main() выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?**

Может и оно будет передано в виртуальную машину Java (JVM).

## **Что такое «интернационализация», «локализация»?**

****Интернационализация (internationalization)**** - способ создания приложений, при котором их можно легко адаптировать для разных аудиторий, говорящих на разных языках.

****Локализация (localization)**** - адаптация интерфейса приложения под несколько языков. Добавление нового языка может внести определенные сложности в локализацию интерфейса.