1. Специализация: ООП и исключения

Экран	Слова
Титул	Здравствуйте, продолжим беседу об ООП
Отбивка	и дополнительно затронем вопрос исключений.
На прошлом уроке	мы поговорили о достаточно большой теме - реализации ООП в джава. рассмотрели классы и объекты, а также наследование, полиморфизм и инкапсуляцию. Дополнительно немного поговорили об устройстве памяти. На следующей лекции рассмотрим внутренние и вложенные классы, перечисления и исключения, нас ждут очень интересные темы, не переключайтесь
На этом уроке	На этой лекции в дополнение к предыдущей, разберём такие понятия как внутренние и вложенные классы; процессы создания, использования и расширения перечислений. Детально разберём уже знакомое вам понятие исключений и их тесную связь с многопоточностью в джава. Посмотрим на исключения с точки зрения ООП, обработаем немного исключений, а также раз и навсегда разделим понятия штатных и нештатных ситуаций.
отбивка Перечисления	Начнём с небольшой темы, с перечислений
Перечисление - это упоминание объектов, объединённых по какому-либо признаку	Кроме восьми примитивных типов данных и классов в Java есть специальный тип, выведенный на уровень синтаксиса языка - enum или перечисление. 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.
лайвкод 04-сезоны	Если копнуть немного глубже, перечисления - это такие специальные классы, содержащие внутри себя собственные статические экземпляры. Сложноватая мысль, если нужно, повторите её про себя несколько раз. а я пока напишу перечисление времён года enum Season WINTER, SPRING, SUMMER, AUTUMN . Когда мы доберёмся до рассмотрения внутренних и вложенных классов, в том числе статических, дополнительно это проговорим.
лайвкод 04-один-сезон	Перечисление фактически представляет новый тип данных, поэтому мы можем определить переменную данного типа и использовать её. Переменная типа перечисления может хранить любой объект этого исключения. Season current = Season.SPRING; System.out.println(current); Интересно также то, что вывод в терминал и запись в коде у исключений полностью совпадают, поэтому, в терминале мы видим ....

Экран	Слова
лайвкод 04-перечислить	Каждое перечисление имеет статический метод values(). Он возвращает массив всех констант перечисления, далее мы можем этим массивом манипулировать как нам нужно, например, вывести на экран все его элементы. Season[] seasons = Season.values(); for (Season s : seasons) System.out.printf(”s s); Именно в этом примере, я использую цикл foreach для прохода по массиву, потому что у перечислений нет индексов, а показать их нужно. Если коротко, данный цикл возьмёт последовательно каждый элемент перечисления, присвоит ему имя s точно также, как мы это делали в примере на две строки выше, и сделает эту переменную С доступной в теле цикла в рамках одной итерации, на следующей итерации будет взят следующий элемент, и так далее
лайвкод 04-порядковый номер	Также в перечисления встроен метод ordinal() возвращающий порядковый номер определенной константы (нумерация начинается с 0). System.out.println(current.ordinal()) Обратите внимание на синтаксис, метод можно вызвать только у конкретного экземпляра перечисления, а при попытке вызова у самого класса перечисления System.out.println(Seasons.ordinal()) мы ожидаемо получаем ошибку невозможности вызова нестатического метода из статического контекста.
03-статические-поля	как мы с вами помним из пояснения связи классов и объектов, такое поведение возможно только если номер элемента как-то хранится в самом объекте. Мы видим в перечислениях очень примечательный пример инкапсуляции - мы не знаем, хранятся ли на самом деле объекты перечисления в виде массива, но можем вызвать метод вельюс. Мы не знаем, хранится ли в каждом объекте перечисления его номер, но можем вызвать его метод ординал. А раз перечисление - это класс, мы можем определять в нём поля, методы, конструкторы и прочее.

Экран	Слова
лайвкод 04-перечисление-цвет	Перечисление Color определяет приватное поле code для хранения кода цвета, а с помощью метода getCode оно возвращается. enum Color RED(”#FF0000”), GREEN(”#00FF00”), BLUE(”#0000FF”); String code; Color (String code) this.code = code; String getCode() return code; Через конструктор передается для него значение. Следует отметить, что конструктор по умолчанию приватный, то есть имеет модификатор private. Любой другой модификатор будет считаться ошибкой. Поэтому создать константы перечисления с помощью конструктора мы можем только внутри перечисления. И что косвенно намекает нам на то что объекты перечисления это статические объекты внутри самого класса перечисления. Также важно, что механизм описания конструкторов класса работает по той же логике, что и обычные конструкторы, то есть создав собственный конструктор мы уничтожили конструктор по-умолчанию, впрочем, мы его можем создать, если это будет иметь смысл для решаемой задачи.
лайвкод 04-перечисление-с-полем	Исходя из сказанного ранее можно сделать вывод, что с объектами перечисления можно работать точно также, как с обычными объектами, что мы и сделаем, например, выведя информацию о них в консоль for (Color c : Color.values()) System.out.printf(”s(s) c, c.getCode());
отбивка Вложенные и внутренние (Nested) классы	Взглянем на чуть более, скажем так, комплексный момент - вложенные классы. На самом деле мы очень хорошо подготовились к этой теме и сейчас должно быть не так уж и сложно.
03-вложенные-классы	В Java есть возможность создавать классы внутри других классов и их разделяют на два вида: Non-static nested classes — нестатические вложенные классы. По-другому их еще называют inner classes — внутренние классы; Static nested classes — статические вложенные классы. Непосредственно внутренние классы подразделяются ещё на два подвида. Помимо того, что внутренний класс может быть просто внутренним классом, он еще бывает: локальным классом (local class); анонимным классом (anonymous class). Анонимные классы мы пока что не будем рассматривать, отложим на несколько лекций.

Экран	Слова
лайвкод 04-класс-апельсин	Разберём всё по порядку и начнём мы с внутренних классов. Почему их так называют? public class Orange public void squeezeJuice() System.out.println(”Squeeze juice ...”); class Juice public void flow() System.out.println(”Juice dripped ...”); Всё просто, их создают внутри другого класса. Рассмотрим на примере апельсина с реализацией, как это предлагает официальная документация оракла.
04-использование-апельсина	В основной программе мы должны будем создать отдельно апельсин, отдельно его сок через вот такую интересную форму вызова конструктора и можем отдельно работать как с апельсином, так и его соком. Orange orange = new Orange(); Orange.Juice juice = orange.new Juice(); orange.squeezeJuice(); juice.flow(); Здесь всё прозрачно и последовательно, но не очень хорошо соответствует жизни.
лайвкод 04-технологичный-апельсин	Важно, что мы же программисты, разработчики. Ещё немного и инженеры, уж мы то понимаем, что когда мы сдавливаем апельсин из него сам по себе течёт сок, а когда апельсин попадает к нам в программу он сразу снабжается соком. Поэтому мы можем слегка модифицировать наш код public class Orange private Juice juice; public Orange() this.juice = new Juice(); public void squeezeJuice() System.out.println(”Squeeze juice ...”); juice.flow(); private class Juice public void flow() System.out.println(”Juice dripped ...”); Что мы сделали? Мы создали объект апельсина. Создали один его, если можно так выразиться, «подобъект» — сок. Далее, мы описали потенциальное наличие у апельсина сока, как его части, поэтому создали внутри класса апельсин класс сока. При создании апельсина создали сок, то есть можно сказать что описали некоторое созревание, Решив выдавить сок у апельсина - объект сока сообщил о том, что начал течь
лайвкод 04-апельсиновый сок	И в основной программе осталось выполнить довольно привычное нам создание апельсина Orange orange = new Orange(); orange.squeezeJuice(); После чего произойдёт достаточно логичное для сдавливания апельсина действие - вытекание сока

Экран	Слова
04-схема-работы-внутреннего-класса	Таким образом очевидно что мы создаем апельсин и внутри него создается сок при создании каждого объекта апельсина то есть у каждого апельсина будет свой собственный сок который мы можем выжать сдавив апельсин. в этом смысл внутренних классов не статического типа. Нужные нам методы вызываются у нужных объектов. Все просто и удобно. И кстати вполне возможно что в будущем нам это пригодится такая связь объектов и классов называется композиции есть ещё ассоциация и агрегация а именно эта композиция.
На одном слайде 04-апельсин и 04-технологичный-апельсин	Если класс полезен только для одного другого класса, то вполне логично встроить его в этот класс и хранить их вместе. Использование внутренних классов увеличивает инкапсуляцию. Оба примера достаточно отличаются реализацией. Мой пример подразумевает ”более сильную”инкапсуляцию, так как извне ко внутренниму классу доступ получить нельзя, поэтому создание объекта внутреннего класса происходит в конструкторе основного класса - в апельсине. Вы можете создавать объект сока где вам это нужно, не обязательно в конструкторе. С другой стороны, у примера из документации есть доступ извне ко внутреннему классу сок но всё равно только через основной класс апельсина. Как и собственно создать объект сока можно только через объект апельсина.

Экран	Слова
Особенности внутренних классов (последовательное появление элементов перечисления) — внутренний объект не существует без внешнего; — внутренний имеет доступ ко всему внешнему; — внешний не имеет доступа ко внутреннему без создания объекта;	Давайте познакомимся с важными особенностями внутренних классов: объект внутреннего класса не может существовать без объекта внешнего класса. Это логично: для того мы и сделали Juice внутренним классом, чтобы в нашей программе не появлялись то тут, то там апельсиновые соки из воздуха. код внутреннего класса имеет доступ ко всем полям и методам экземпляра (так же как и к статическим членам) окружающего класса, включая все члены, даже объявленные как private, на самом деле, от нас это скрыто, но объект внутреннего класса получает неявную ссылку на внешний объект, который его создал, и поэтому может обращаться к членам внешнего объекта без дополнительных уточнений; экземпляр внешнего класса не имеет доступа ни к каким членам экземпляра внутреннего класса на прямую, то есть без создания экземпляра внутреннего класса внутри своих методов (И это логично, так как экземпляров внутреннего класса может быть создано сколько угодно много, и к какому же из них тогда обращаться?);

Экран	Слова
— у внутренних классов есть модификаторы доступа; — внутренний класс не может называться как внешний; — во внутреннем классе нельзя иметь не-final статические поля; — Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса — Со внутренними классами работает наследование и полиморфизм.	у внутреннего класса, как и у любого члена класса, может быть установлен один из трех уровней видимости: public, protected или private. Если ни один из этих модификаторов не указан, то по умолчанию применяется пакетная видимость. Это влияет на то, где в нашей программе мы сможем создавать экземпляры внутреннего класса. Единственное сохраняющееся требование — объект внешнего класса тоже обязательно должен существовать и быть видимым; внутренний класс не может иметь имя, совпадающее с именем окружающего класса или пакета. Это важно помнить. Правило не распространяется ни на поля, ни на методы; внутренний класс не может иметь полей, методов или классов, объявленных как static (за исключением полей констант, объявленных как static и final). Статические поля, методы и классы являются конструкциями верхнего уровня, которые не связаны с конкретными объектами, в то время как каждый внутренний класс связан с экземпляром окружающего класса; Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса. Это объясняется особенностями устройства внутренних классов. У внутреннего класса могут быть конструкторы с параметрами или только конструктор по умолчанию. Но независимо от этого, когда мы создаем объект внутреннего класса, в него незаметно передается ссылка на объект внешнего класса. Ведь наличие такого объекта — обязательное условие. Иначе мы не сможем создавать объекты внутреннего класса. Но если метод внешнего класса статический, значит, объект внешнего класса может вообще не существовать. А значит, логика работы внутреннего класса будет нарушена. В такой ситуации компилятор выбросит ошибку; Также, внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования. Это уже более сложная тема, которую вы можете рассмотреть самостоятельно при желании.

Экран	Слова
03-вложенные-классы	Как мы помним классы это новый тип данных для нашей программы поэтому стоит ли упоминать что мы можем создавать классы а также их описывать например внутри методов это довольно редко используется но синтаксически язык позволяет это сделать. Первое, что нужно вспомнить перед изучением — их место в структуре вложенных классов. Исходя из схемы мы можем понять, что локальные классы — это подвид внутренних классов. Однако, у локальных классов есть ряд важных особенностей и отличий от внутренних классов. Главное заключается в их объявлении.
лайвкод 04-локальный-внутренний-класс	Локальный класс объявляется только в блоке кода. Чаще всего — внутри какого-то метода внешнего класса. Например, это может выглядеть так: public class Animal void performBehavior(boolean state) class Brain void sleep() if(state) System.out.println(”Sleeping”); else System.out.println(”Not sleeping”); Brain brain = new Brain(); brain.sleep(); некоторое животное, у которого утанавливается состояние спит оно или нет. метод performBehavior() принимает на вход булевое значение и определяет, спит ли животное. И внутри этого метода мы объявили наш локальный класс Brain
лайвкод 04-вызов-с-локальным-классом	Соответственно, снаружи это просто вызов метода Animal animal = new Animal(); animal.performBehavior(true); мог возникнуть логичный вопрос: зачем? Зачем объявлять класс именно внутри метода? Почему не использовать обычный внутренний класс? Действительно, можно было бы просто сделать класс Brain внутренним. Другое дело, что итоговое решение зависит от структуры, сложности и предназначения программы.

Экран	Слова
Особенности локальных классов (последовательное появление элементов перечисления) — сохраняет доступ ко всем полям и методам внешнего класса; — должен иметь свои внутренние копии всех локальных переменных; — имеют ссылку на окружающий экземпляр.	Соответственно, теперь рассмотрим особенности: локальный класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода, то есть полям и аргументам метода объявленным как final. Но начиная с JDK 8 локальный класс может обращаться к любым полям и аргументам метода объявленным в текущем блоке кода, даже если они не объявлены как final, но только в том случае если их значение не изменяется после инициализации; локальный класс должен иметь свои внутренние копии всех локальных переменных, которые он использует (эти копии автоматически создаются компилятором). Единственный способ обеспечить идентичность значений локальной переменной и ее копии – объявить локальную переменную как final. Опять же напомню, что это все было справедливо до JDK 7 включительно. В JDK 8 ситуация поменялась и можно обойтись без объявления переменной как final, но не менять ее значение в коде после инициализации. Хотя по большому счету лучше, для самоконтроля, все таки объявлять переменные как final; экземпляры локальных классов, как и экземпляры внутренних классов, имеют окружающий экземпляр, ссылка на который неявно передается всем конструкторам локальных классов. В результате определение внутренних классов можно описать так: подобно полям и методам экземпляра, каждый экземпляр внутреннего класса связан с экземпляром класса, внутри которого он определен (то есть каждый экземпляр внутреннего класса связан с экземпляром его окружающего класса). Вы не можете создать экземпляр внутреннего класса без привязки к экземпляру внешнего класса. То есть сперва должен быть создан экземпляр внешнего класса, а только затем уже вы можете создать экземпляр внутреннего класса.
отбивка Статические вложенные классы	Мы поговорили о нестатических внутренних классах (non-static nested classes) или, проще, внутренних классах. Рассмотрим статические вложенные классы (static nested classes). Чем они отличаются от остальных?

Экран	Слова
лайвкод 04-статический-класс	При объявлении такого класса мы используем уже знакомое нам ключевое слово static. Возьмём класс нашего котика и заменим метод voice() на статический класс. Объясняется это, как вы уже слышали на прошлом уроке, что допустим, мы находимся дома и у нас отрыто окно, мы слышим разные звуки, которые доносятся из окна. Из этих звуков мы можем разобрать звук мурчания котика. И тут мы понимаем, что котика мы не видим, а при этом слышим. public class Cat private String name, color; private int age; public Cat() public Cat(String name, String color, int age) this.name = name; this.color = color; this.age = age; static class Voice private final int volume; public Voice(int volume) this.volume = volume; public void sayMur() System.out.printf(”A cat purrs with volume dn volume); То есть, такое мурчание котика может присутствовать без видимости и понимания, что это такой за котик. Также, добавим возможность установить уровень громкости его мурчанья
04-отличия-статик-и-не	В чем отличие между статическим и нестатическим вложенными классами? Объект статического класса не хранит ссылку на конкретный экземпляр внешнего класса. Если помните, только что мы говорили о том, что в каждый экземпляр внутреннего класса незаметно для нас передается ссылка на объект внешнего класса. Без объекта внешнего класса объект внутреннего просто не мог существовать. Для статических вложенных классов это не так. Объект статического вложенного класса вполне может существовать сам по себе. В этом плане статические классы более независимы, чем нестатические.
лайвкод 04-использование-статического	Довольно важный и вместе с тем довольно очевидный момент заключается в том, что при создании такого объекта нужно указывать название внешнего класса, Cat.Voice voice = new Cat.Voice(100); voice.sayMur(); примерно так.

Экран	Слова
лайвкод 04-последнее-о-статике	И ещё одна особенность - разный доступ к переменным и методам внешнего класса. Статический вложенный класс может обращаться только к статическим полям внешнего класса. При этом неважно, какой модификатор доступа имеет статическая переменная во внешнем классе. Даже если это private, доступ из статического вложенного класса все равно будет. Все вышесказанное касается не только доступа к статическим переменным, но и к статическим методам. Слово static в объявлении внутреннего класса не означает, что можно создать всего один объект. for (int i = 0; i < 4; i++) Cat.Voice voice = new Cat.Voice(100 + i); voice.sayMur(); Не следует путать объекты с переменными. Если мы говорим о статических переменных — да, статическая переменная класса существует в единственном экземпляре. Но применительно ко вложенному классу static означает лишь то, что его объекты не содержат ссылок на объекты внешнего класса. В случае примера с котиком - мы слышим мурчание с разной громкостью и непонятно одного и того же котика или это другой. А самих объектов мы можем создать сколько угодно
отбивка Механизм исключительных ситуаций	Наконец-то, тема, которая почему-то вызывает у новичков отторжение недоумение и полное непонимание.
Исключение - это отступление от общего правила, несоответствие обычному порядку вещей	Думаю, тут надо начать с такой, немного философской части. Посмотрите пока что на этот вступительный слайд, он хорошая и в нём нет ничего сложного. Мы изучаем программирование, а что такое язык программирования? Это в первую очередь набор инструментов. Смотрите, например, есть строитель или вот лучше - художник. У художника есть набор всевозможных красок, кистей, холстов, карандашей, мольберт, ластик и куча-много-чего-ещё. Это всё его инструменты, с их помощью он делает свои важные художественные штуки. Тоже самое для программиста, у программиста есть язык программирования, который предоставляет ему инструменты: циклы, условия, классы, функции, методы, ООП, фрейморки, библиотеки... Исключения - это один из инструментов. Смотрите на исключения как на ещё один инструмент для работы программиста. Работает он достаточно специфично, и является достаточно высокоуровневым, исключения представляют из себя некую подсистему языка, которая является неотъемлемой частью любого мало-мальски серьёзного проекта.

Экран	Слова
В общем случае, возникновение исключительной ситуации, это ошибка в программе, но основным вопросом является следующий: — ошибка в коде программы, — ошибка в действиях пользователя — ошибка в аппаратной части компьютера	Итак бывают такие ситуации, когда в процессе выполнения программы возникают ошибки. При возникновении ошибок создаётся объект класса Исключение, и в этот объект записывается какое-то максимальное количество информации о том, какая ошибка произошла, чтобы потом прочитать и понять, где же проблема. Соответственно эти объекты можно ловить, связывать и бросать в подвал для дальнейших выяснений... Но в программировании это называется гуманным термином “обрабатывать”. То есть вы можете как-то повлиять на ход программы, когда она уже запущена, и сделать так, чтобы она не прекратила работу, увидев деление на ноль, например, а выдала пользователю сообщение, и отменила только одну последнюю операцию. Сегодня поговорим о том, как отличить штатную ситуацию от нештатной.
04-иерархия-исключений	Исключения все наследуются от класса Throwable и могут быть как обязательные к обработке, так и необязательные. Есть ещё подкласс Error, но он больше относится к аппаратным сбоям или серьёзным алгоритмическим или архитектурным ошибкам, и нас не интересует, потому что поймав что-то вроде OutOfMemory мы средствами Java прямо в программе ничего с ним сделать не сможем, такие ошибки надо обрабатывать и исключать в процессе разработки ПО. Или, возможно, в системном программировании, но не в прикладном. Нас интересует подкласс Throwable-Exception<RuntimeException и например какой-нибудь IOException. Вообще их куча много-много, на досуге можете полистать список на сайте оракл. Но можете считать, что все исключения, с которыми вы можете работать наследуются от Throwable-Exception. Важная информация. Все эксепшены, кроме наследников рантайма надо обрабатывать.

Экран	Слова
лайвкод 04-мартёшка-методов	Давайте рассмотрим примерчики, напишем пару-тройку методов, и сделаем матрёшку, из мэйна вызываем метод2, оттуда метод1, оттуда метод0, а в методе0 всё как всегда портим, пишем private static int div0() return 1 / 0; ArithmeticException является наследником класса RuntimeEcxeption поэтому статический анализатор его не подчеркнул, и ловить его вроде как не обязательно, спасибо большое разработчикам джава, надёжность кода не повысилась, единообразность работы всех исключений нарушилась, всё хорошо.
лайвкод 04-метод-деления	Выходит, на примере деления на ноль можно всё хорошо сразу и объяснить. допустим у нас есть какой-то метод который возможно мы даже сами написали, который, скажем, целочисленно делит два целых числа. немного его абстрагируем private static int div0(int a, int b) return a / b; Если посмотреть на этот метод с точки зрения программирования, он написан очень хорошо - алгоритм понятен, метод с единственной ответственностью, всё супер. Однако, из поставленной перед методом задачи очевидно, что он не может работать при всех возможных входных значениях. То есть если у нас вторая переменная равна нулю, то это неправильно. И что же с этим делать?
лайвкод 04-исключение-1	Нам нужно как-то запретить пользователю передавать в качестве делителя ноль. Самое простое - ничего не делать, но мы так не можем. private static int div0(int a, int b) if (b != 0) return a / b; return ???; Потому что метод должен что-то вернуть, а что вернуть, неизвестно, ведь от нас ожидают результат деления. Поэтому, допустим можем руками сделать проверку (б == 0ф) и выкинуть пользователю так называемый объект исключения throw new RuntimeException(”деление на ноль”) а иначе вернём а / б. private static int div0(int a, int b) if (b == 0) throw new RuntimeException(”parameter error”); return a / b;
лайвкод 04-исключение-2	Вызываем метод и пробуем делить 1 на 2. А вот если мы второй параметр передадим 0 то у нас выкинется исключение, System.out.println(div0(1,2)); System.out.println(div0(1,0)); то есть по сути new... это конструктор, нового объекта какого-то класса, в который мы передаём какой то параметр, в данном конкретном случае это строка с сообщением. Зафиксируем пока что эту мысль.
отбивка введение в многопоточность	Кажется многовато отступлений, но без этого точно никак нельзя продолжать.

Экран	Слова
04-метод-броска	Итак, что происходит? Ключевое слово throw заставляет созданный объект исключения начать свой путь по родительским методам, пока этот объект не встретится с каким-то обработчиком. в нашем текущем случае - это дефолтный обработчик виртуальной машины, который в специальный поток err выводит так называемый стектрейс, и завершает дальнейшее выполнение метода.
04-поток-epp	Далее по порядку: поток epp. Все программы в джава всегда многопоточны. Понимаете вы многопоточность или нет, знаете ли вы о её существовании или нет, не важно, многопоточность есть всегда. не будем вдаваться в сложности прикладной многопоточности, у нас ещё будет на это довольно много времени, пока что поговорим в общем. в чём смысл? смысл в том, что на старте программы запускаются так называемые потоки, которые работают псевдопараллельно и предназначены каждый для решения своих собственных задач, например, это основной поток, поток сборки мусора, поток обработчика ошибок, потоки графического интерфейса. Основная задача этих потоков - делать своё дело и иногда обмениваться информацией.
04-стектрейс	В упомянутый же парой минут ранее стектрейс кладётся максимальная информация о типе исключения, его сообщении, иерархии методов, вызовы которых привели к исключительной ситуации. Если не научиться читать стектрейс, если честно, можно расходиться по домам и не думать о серьёзном большом программировании. Итак стектрейс. Когда у нас случается исключение - мы видим, что случилось оно в потоке мэйн, и является объектом класса RuntimeException сообщение мы тоже предусмотрительно приложили. Первое что важно понять, что исключение - это объект класса. Далее читаем матрёшку - в каком методе создался этот объект, на какой строке, в каком классе. Далее смотрим кто вызвал этот метод, на какой строке, в каком классе. Это вообще самый простой стектрейс, который может быть. Бывают полотна по несколько десятков строк, клянусь, сам видел. Особенно важно научиться читать стектрейс разработчикам андроид, потому что именно такие стектрейсы будут вам прилетать в отчёт. У пользователя что то упало, он нажал на кнопку отправить отчёт, и вам в консоль разработчика прилетел стектрейс, который будет являться единственной доступной информацией о том, где вы или пользователь накосячили.

Экран	Слова
лайвкод 04-простой-пример-исключения	Если мы не напишем никакого исключения, кстати, оно всё равно произойдёт. Это общее поведение исключения. Оно где-то случается, прекращает выполнение текущего метода, и начинает лететь по стеку вызовов вверх. Возможно даже долетит до дефолтного обработчика, как в этом примере. int[] arr = 1; System.out.println(arr[2]) Некоторые исключения генерятся нами, некоторые самой джавой, они вполне стандартные, например выход за пределы массива, деление на ноль, и классический нуль-поинтер.
	Давайте наше исключение ловить. Первое, и самое важное, что надо понять - это почему что-то упало, поэтому не пытайтесь что то ловить, пока не поймёте что именно произошло, от этого понимания будет зависеть способ ловли. Исключение ловится двухсекционным оператором try-catch, его первой секцией try. Это секция, в которой предполагается возникновение исключения, и предполагается, что мы можем его поймать. А в секции catch пишем имя класса исключения, которое мы ловим, и имя объекта, в который мы положим экземпляр нашего исключения. Секция catch ловит указанное исключение и всех его наследников. Это важно. Рекомендуется писать максимально узко направленные секции catch, потому что надо стараться досконально знать как работает ваша программа, и какие исключения она может выбрасывать. Ну и ещё потому что разные исключения могут по-разному обрабатываться, конечно-же. Секций catch может быть сколько угодно много. Как только мы обработали объект исключения, он уничтожается, дальше он не поднимается, и в следующие catch не попадает. Мы, конечно, можем его насильно пульнуть выше, ключевым словом throw

Экран	Слова
	Так вот, когда какой-то наш метод выбрасывает исключение вы обязаны либо вынести объявление этого исключения в сигнатуру метода, что будет говорить тем, кто его вызывает о том, что в методе может возникнуть исключение, либо мы это исключение должны непосредственно в методе обработать, иначе у вас ничего не скомпилируется. Примером одного из таких исключений служит ИО это сокращение от инпут-аутпут и генерируется, когда, вы не поверите, возникает ошибка ввода-вывода. То есть при выполнении программы что-то пошло не так и она, программа не может произвести ввод-вывод в штатном режиме. На деле много чего может пойти не так, операционка заглючила, флешку выдернули, устройство телепортировалось в микроволновку, и всё, случился ИОЭксепшн, не смогла программа прочитать или написать что то в потоке ввода-вывода. Соответственно, уже от этого ИОЭ возникают какие-то другие, вроде FileNotFoundException, которое мы тоже обязаны обработать. Например, мы хотим чтобы наша программа что то там прочитала из файла, а файла на нужном месте не оказалось, и метод чтения генерирует исключение.