1. Интерфейсы

Экран	Слова
Титул	Здравствуйте, добро пожаловать на курс посвящённый инструментарию разработчика на джава
Отбивка	и сегодня на повестке дня у нас интерфейсы, но не те интерфейсы, что графические, а те, что программные.
На прошлом уроке	На прошлом уроке мы поговорили как раз о графических интерфейсах пользователя. создали немного окон, разместили немного компонентов, порисовали. Поговорили о графических интерфейсах через призму создания и взаимодействия объектов, нарисовали парутройку диаграмм, чтобы лучше запомнить, что как и с чем связывается, кто кого вызывает и зачем.
На этом уроке	Поговорим об интерфейсах, рассмотрим понятие и принцип работы, поговорим о ключевом слове implements; Наследование и множественное наследование интерфейсов, реализация, значения по умолчанию и частичная реализация интерфейсов. Естественно, что говоря об интерфейсах нельзя не сказать о функциональных интерфейсах и анонимных классах. Коротко рассмотрим модификаторы доступа при использовании интерфейсов.
06-01	Разговор об интерфейсах хотелось бы построить не так, как мы это делали на лекциях обычно, а от некоторой практической составляющей, чтобы где-то в середине лекции у вас сложилось явное впечатление, что что-то тут явно можно улучшить, а когда мы применим интерфейсы, вы подумали «аааaа так вот зачем оно нужно и как применяется», а потом уже поговорим о теоретической составляющей и особенностях.
06-02	Итак начнём с преамбулы, здесь я призываю вас не особенно обращать внимание на то, какие именно классы и методы используются, а внимательно следить за взаимодействием и отношениями объектов, потому что интерфейсы, о которых мы сегодня планируем поговорить - это как раз механизм упрощающий и универсализирующий взаимодействия объектов. Код может показаться непростым, но зато задачу мы поставим таким образом, что если делать нормально, то без интерфейсов не обойтись, и когда мы закончим - у вас, как я только что и сказал, должно появиться ощущение что «о, так вот зачем они нужны». Почему будет сложно? потому что несмотря на то что вы уже знаете принципы ооп - надо уметь их применять.

Экран	Слова
06-03	Итак, что мы будем делать? мы сделаем некий небольшой набросок своего собственного игрового 2д движка, без физики и прочего, а просто с объектами и анимацией, чисто демонстрационный. На его основе можно что угодно запилить, и будет отлично работать. На слайде вы видите окно, кружки летают (имейте ввиду, они перемещаются плавно, если у кого-то что-то дёргается, это интернет с низким ФПС передаёт, а не приложение). пока что ничего тут не обрабатывается, ничего толком не происходит. Итак, самое главное, что нам понадобится, это окно, которое будет как-то взаимодействовать с операционной системой, на окне будет канва, на которой мы будем всё рисовать, и собственно объекты, которые мы будем рисовать.
06-04	Быстро создаём окно, буквально константы с размерами, координатами, и конструктор окна прямо здесь же вместе с методом мейн, не вдаваясь в подробности того, как работает фреймворк свинг на котором мы всё это пишем, на эту тему у нас был отдельный урок, на котором уже закрепили информацию об ООП. Самое важное для нас сейчас - это то, что окно - это объект с какими-то свойствами и каким-то поведением. Если коротко, стартуем программу, создаём объект окна, говорим, что окно с каким-то заголовком, какого-то размера и находится по каким-то координатам, а когда мы нажмём на крестик, то закрыть нужно будет не только окно, но и программу целиком.
06-05	Припоминаем, что есть такой компонент JPanel, с ним можно много что делать но самое интересное, что можно на нём рисовать. Итак создали наследника Jpanel с названием MainCanvas. Что умеет любой компонент в свинге? правильно, он умеет перерисовываться, посредством вызова метода paintComponent. Применяем полиморфизм. Из документации мы знаем, что метод пэинтКомпонент вызывается тогда, когда фреймворку надо перерисовать панельку. Мы этот метод переопределим и напишем свою собственную реализацию перерисовки. Давайте сделаем так: создадим конструктор панельки, и будем в конструкторе делать что то незначительное, например, менять цвет фона на синий. Заоверрайдили пэинтКомпонент и тут-же вызвали родительский метод. То есть по сути мы говорим, что нас вполне устраивает то, как перерисовывается панелька, но мы потом захотим туда что-то добавить. Ну и добавим четыре метода, возвращающие границы нашей канвы, левую, правую, верхнюю, нижнюю, для удобства дальнейшего взаимодействия с получившейся канвой.

Экран	Слова
06-06	Наладим взаимодействие компонентов и привяжем все действия нашего игрушечного движка ко времени физического мира. В основном окошке, в конструкторе, создали переменную класса MainCanvas и расположили её на окне прям в центре, пусть на окне будет только она. Пока что всё понятно и хорошо. Всё работает. В принципе можно начать писать логику игры прям здесь, в классе канвы, но это архитектурно не очень хорошо, ведь это же канва на которой мы рисуем, значит здесь мы должны по логике только рисованием заниматься. Давайте примем архитектурное решение писать логику игры в нашем классе с кружками, а MainCanvas будет универсальным, чтобы иметь возможность в дальнейшем рисовать вообще всё что угодно. Для этого описали в нашем основном окошке метод, назвали его как-нибудь нормально onDrawFrame и в нём будем описывать реализацию цикла для нашего модного приложения, то есть так называемую бизнес-логику. На данный момент это будут два метода - апдейт который будет как то изменять состояние нашего приложения, и рендер, который будет отдавать команды всяким рисующим компонентам. То есть получается, что канва время от времени будет говорить, что она нарисовалась, а основное окно по этому событию будет что-то умное предпринимать

Экран	Слова
06-07	Теперь надо чтобы при перерисовке наш с вами MainCanvas этот метод дёргал, и тем самым изображение как-то менялось. Для этого канве надо знать как минимум чей метод она будет дёргать, то есть ей нужно знать на каком окне она находится. создаём в классе канвы локальную переменную, которая умеет хранить объекты класса MainWindow и передадим значение этой переменной в конструкторе. А в пэинтКомпоненте будем вызывать нужный метод controller.onDrawFrame(); Далее, чтобы зациклить это действие мы можем пойти несколькими путями: самый простой - создать постоянно обновляющуюся канву, то есть в методе пэинтКомпонент взять и написать repaint() но это вариант прямо скажем ”так себе он полностью нагрузит одно из ядер процессора только отрисовкой окна - не самое лучшее применение одного из ядер. Второй путь - это применение магии из занятия по потокам. мы можем заставить наш поток какое-то время поспать, поэтому мы вызываем статический метод класса Трэд, он называется слип и принимает на вход количество миллисекунд, которое поток должен обязательно поспать. это даст нам фпс близкий к 60, приемлемо для условной игры, не надо ни больше ни меньше, пожалуй. Получится, что мы создали внутри этого метода некий бесконечный цикл отрисовки, своеобразный ду-вайл, который сам себя заставляет крутиться дальше с некоторой периодичностью и на каждой своей итерации сообщает контроллеру, что прошло около одной шестидесятой секунды.
06-08	Довольно интересно, кстати, как при этом изменился код, вызывающий конструктор канвы, ведь мы из основного класса с окном теперь как-то должны в канву передать это самое основное окно, как вы могли догадаться, для этой цели мы применяем указатель на текущий объект класса this, то есть в конструкторе основного окна мы передали ссылку на экземпляр этого окна канве. немного ломает привычное использование для обращения к полям в конструкторе, но, поверьте, то ли ещё будет. Предлагаю запомнить такой способ применения ключевого слова this, он нам сегодня ещё пригодится. Ещё раз: создавая экземпляр основного окна мы передаём ссылку на этот создаваемый экземпляр канве, чтобы она знала, у какого окна дёргать метод обновления.

Экран	Слова
06-09	Закончим с отрисовкой и методом onDrawFrame. Он будет обновлять сцену и рендерить её. Для обновления сцены было бы очень неплохо знать дельту времени, которая прошла с предыдущего кадра, чтобы обновлять мир. Конечно можно писать, опираясь на частоту кадра, или на то что мы там спим 16миллисекунд, но это очень сомнительная опора, потому что мы гарантированно спим 16миллисекунд, но сколько именно мы будем выполнять остальные действия - неизвестно, потому что отрисовка происходит не через фиксированные промежутки времени а по очереди сообщений окна и ещё куча факторов, поэтому, лучше всего точно знать сколько времени прошло с предыдущего кадра. Сделаем так чтобы метод onDrawFrame эту самую дельту у канвы получал и отдавал методу обновления. Возможно, метод ещё захочет знать, какая именно канва отрисовалась, вдруг их будет несколько, да и вдруг нам для логики понадобится узнавать размеры канвы... и нам нужен будет объект графики с канвы, чтобы отдавать ей команды на рисование. Соответственно считаем дельту в канве. Важно, чтобы привести всё к привычному времени, например, пиксель-в-секунду, отдавать время в секундах, поэтому дополнительно переводим из наносекунд в секунды. Вот, собственно и вся физика, которая нам понадобится. отдадим себя и отдадим свой объект графики, чтобы основная логика могла узнать наши размеры и на нас же рисовать. Все изменения канвы вы видите на слайде, все изменения основного окна ему соответствуют. Пока самое главное, что нам нужно понять об этих двух объектах - канва считает для нас время в физическом мире и постоянно перерисовывает себя, сообщая об этом факте основному окну, а основное окно на этот факт как-то реагирует. Ну и ООП вокруг этого тоже было бы хорошо понимать, объекты передают ссылки друг на друга и вызывают друг у друга всякие интересные методы.

Экран	Слова
06-10	Рисовать несложные штуки мы научились буквально на прошлой лекции. Наше приложение будет рисовать какие-то объекты, будут это кружки, квадратики, картинки, человечки или какие-то другие объекты - не важно. Важно, чтобы у программы было описан механизм и поведение этих объектов. Это как раз то, о чём я говорил и говорю - применение архитектуры, применение ООП. Мало просто посмотреть уроки, почитать книжки, посидеть на семинаре. Надо сидеть и думать в рамках парадигмы ООП. Это ещё простенькая архитектура. Я догадываюсь, что вы смотрите сейчас, и вроде всё понятно, а сами в жизни бы такое не написали. Я сам несколько лет назад такое в жизни не написал бы, так что не переживайте, всё придёт с опытом. Главное, не путайте понятия уметь программировать и знать язык программирования. В какой-то момент вы поймаете себя на мысли, что вы не думаете о циклах и условиях, а думаете на следующем уровне абстракции, думаете о взаимосвязях, об архитектуре, а пальцы сами набирают какие-то языковые конструкции.
06-11	Соответственно, рисовать просто линии круги и прочее - довольно скучно, поэтому будем рисовать объекты, Создадим класс Спрайт. Ни от чего наследоваться не будем. Просто опишем общее для всех рисуемых объектов в нашей программе поведение. Что может быть у всех объектов в приложении общего? размеры, местоположение. Обычно, когда вы начинаете изучать какой-то графический фреймворк вы замечаете, что начало координат у этого фреймворка находится в верхнем левом или нижнем левом углу. Однако очень часто, когда пишутся какие-то игры или другие приложения с использованием графики в качестве координат используется центр объекта. То есть надо условиться - что икс и игрек - это центр любого визуального объекта на нашей канве. И соответственно удобно хранить не длину-ширину, а половину длины и половину ширины. А границы объекта соответственно будем отдавать через геттеры и сеттеры. Дополнительно научим наш спрайт рисоваться. Ну как научим, просто скажем, что он умеет обновляться и рендериться, а его наследники пусть уже решают, как именно они хотят это делать. Но спрайты лишены логики, они ничего не знают о том, как именно будут меняться их свойства.

Экран	Слова
06-12	Поэтому естественно нужно создать класс собственно шарика, который будет по нашему экрану прыгать, а то непорядок какой-то. В конструкторе задаём мячику рэндомные размеры. Давайте придумаем ему какие-нибудь глобальные свойства. Можно конечно придумать класс, который будет направлять наш спрайт и вообще задавать ему скорость и прочие физические величины, но мы ж с вами пример пишем, так что придумаем ему просто - скорость по осям x и y соответственно и цвет. Для шарика переопределяем апдейт и рендер. суперы здесь не нужны, они всё равно пустые. Самый простой рендер - мы объекту графики зададим цвет текущего шарика и сделаем fillOval, которому передадим лево, верх, ширину и высоту. Несмотря на то что наши объекты содержат поля типа флоут, мы работаем с пиксельной системой координат, а значит надо переводить в целые числа, (что конечно же не подходит для реальных проектов, там нужно всё сразу переводить в мировые координаты (например принять центр экрана за 0, верх и лево за -1 низ и право за 1, как это делает OpenGL) чтобы рендерить экраны). Но это нам и пяти лекций не хватит чтобы вникнуть так что не будем. А в методе апдейт мы просто прибавляем к текущим координатам шарика его скорость, умноженную на дельту времени, то есть как в третьем классе. Расстояние, которое должен был преодолеть шарик за то время пока канва спала и рендерилась. ну и обрабатываем отскоки, то есть описываем 4 условия, что при достижении границы мы меняем направление вектора.
06-13	быстренько допилим основной класс и будем переходить к беседе об интерфейсах уже, а то чувствую вы под-устали от вступлений. В основном классе мы делаем очень прямолинейно - создаём массив из спрайтов и называться он будет sprites и мы говорим что будет у нас допустим 10 кружчков. В методе апдейт мы пробежимся по всем спрайтам и скажем каждому из них - апдейться так как ты умеешь. В методе рендер мы сделаем тоже самое - пробежимся по всем спрайтам и скажем - отрисуйся так, как ты хочешь. И всё, реализацию обновления и отрисовки мы оставили самим объектам, то есть инкапсулировали в них, только каждый объект сам по себе знает, как именно ему обновляться с течением времени, и как рисоваться, а основной экран уже управляет - на какой канве, когда и кого рисовать. В конструкторе же добавим простой цикл инициализирующий приложение десятью шариками.

Экран	Слова
06-14	Получается, мы меньше чем за час, с учётом того что этот код надо как-то набрать, написали довольно простое и очень хорошо расширяемое приложение, которое рисует штуки. Напомню, что самое главное, что мы должны из этого приложения извлечь - это взаимодействия и взаимовлияния объектов. Наследование, полиморфизм, инкапсуляция поведений и свойств. Если честно, я слегка устал от синего цвета фона, а внимательный зритель мог заметить, что в одном из последних снимков с кодом я убрал строку, изменяющую цвет фона из конструктора. Начать разговор об интерфейсах я решил с создания отдельного класса фона, но сразу столкнулся с необходимостью думать головой.
06-15	Логично было бы предположить, что фон - это спрайт, имеющий прямоугольную форму и всегда рисующийся первым. Но, вот беда, при изменении размеров окна фон тоже желательно изменить в размерах, а это лишние слушатели и десятки строк кода, поэтому в отрисовке класса Фон я просто говорю канве, что она должна изменить свой цвет фона. А что, ссылка то на канву у меня есть, имею право. Цвет фона я меняю синусоидально по каждому из трёх компонент цвета, поэтому изменение происходит плавно. В общем, получается, что от спрайта, фактически, нужно только поведение, а свойства не нужны. Но и отказаться от наследования нам бы не хотелось, потому что тогда мы не сможем фон единообразно в составе массива спрайтов обновлять. Это наталкивает нас на мысль об унификации поведения, на мысль об интерфейсе.
06-16	Механизм наследования очень удобен, но он имеет свои ограничения. В частности мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование

Экран	Слова
06-17	В языке Java эту проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить к себе множество интерфейсов. Правильно говорить реализовать интерфейс, будем сразу говорить правильно. Если сказать проще, интерфейс можно очень-очень грубо представить как очень-очень абстрактный класс. До седьмой джавы это был просто набор методов без реализации. Начиная с восьмой наделали много тонкостей, с ними и будем разбираться. Итак интерфейс - это описание методов. Примером интерфейса в реальной жизни может быть интерфейс управления автомобилем, интерфейс взаимодействия с компьютером или даже интерфейс USB, так, компьютеру не важно, что именно находится по ту сторону провода, флешка, веб-камера или мобильный телефон, а важно, что компьютер умеет работать с интерфейсом USB, отправлять туда байты или получать. Потоки ввода-вывода, которые мы проходили чуть раньше - это тоже своего рода интерфейс, соединяющий не важно какой программный код и не важно какой, например, файл.
06-18	Интерфейсы объявляются также, как классы, и вообще могут иметь очень похожую на класс структуру, то есть быть вложенным или внутренним, но чаще всего каждый отдельный интерфейс описывают в отдельном файле, также как класс, но используя ключевое слово интерфейс. Создадим пару интерфейсов, например, человек и бык, опишем в них методы, например, ходить и издавать звуки. Все методы во всех интерфейсах всегда публичные, и в классическом варианте не имеют реализации. Ну и поскольку все методы всегда паблик то этот модификатор принято просто не писать. Для новичка это неочевидно и сбивает с толку, может показаться что модификатор дефолтный, а на самом деле он публичный, видите, среда разработки отметила ключевое слово паблик как лишнее? поблагодарим разработчиков джавы и пойдём дальше.

Экран	Слова
06-19	Для чего мы так сделали? Продолжим пример, создадим пару классов, класс мужчина и класс бык. Класс мужчины будет реализовывать интерфейс человека, а класс быка внезапно быка. Для того, чтобы реализовать интерфейс - мы должны переопределить все его методы, либо сделать класс абстрактным. Статический анализатор кода в идее нам об этом явно скажет. Выведем модное сообщение о том что это методы мужчины или быка. То есть множественного наследования нет, но мы можем реализовать сколько угодно интерфейсов. И теперь получается самая соль - мы можем в объявлять не только классы и создавать объекты, но и создать переменную, которая реализовывает интерфейс. То есть тут могут лежать абсолютно никак не связанные между собой объекты, главное, чтобы они реализовывали интерфейс. И мы можем работать с методами интерфейса, которые могут быть для разных классов вообще по-разному реализованы. Такой вот полиморфизм. Понимаете насколько сильно это отличается от наследования, когда мы с вами создавали общий абстрактный класс животное и от него наследовали наших котиков?
06-20	Чтобы стало сильно понятнее, создадим класс минотавра. Для тех, кто не застал античную грецию, напоминаю это такой товарищ, который с телом человека и головой быка скучно сидел в лабиринте и ждал заблудившихся путников. Соответственно, реализовывал интерфейсы человека и быка каким-то своим способом, а именно, ходил на ногах человека, но не мычал, как бык, а загадки загадывал. Интересно то, что в программе мы можем к минотавру обратиться не только как к человеку, но и как к быку, то есть гипотетически, можно создать некоторого Тесея, погонщика минотавровых стад. Но это уже, так сказать, полёт фантазии, нас интересует только техническая часть вопроса - классы не связаны между собой наследованием, а обращение к ним единообразное.
06-21	Также важно, что в интерфейсах разрешено наследование. То есть у нас интерфейс может наследоваться от другого интерфейса, соответственно при реализации такого, наследующего интерфейса, мы должны переопределять не только методы интерфейса, но и методы всех его родителей, то есть тут картина очень похожа на наследование классов, но внимание не запутайтесь, в интерфейсах разрешено множественное наследование. То есть ваш милый домашний пушистый котик может быть также одновременно сумасшедшим ночным тыгыдыком и хищником-убийцей из дикой природы. Ну или вот, как на слайде, много разной мифологии. Это у меня сюда ещё химера не поместилась.

Экран	Слова
06-22	Давно я не задавал никаких вопросов, уже устали, наверное. Давайте отвлечёмся и ответим на пару вопросов. 1. Программный интерфейс - это: 1. окно приложения в ОС; 2. реализация методов объекта; 3. объявление методов, реализуемых в классах. ... 30сек ... это объявление методов, которые потом будут реализованы каждым классом по-своему. Далее... Интерфейсы нужны для: 1. компенсации отсутствия множественного наследования; 2. отделения API и реализации; 3. оба варианта верны. ... 30 сек... оба варианта верны, интерфейсы это достаточно гибкий инструмент, который выступает в роли безопасной замены множественного наследования и позволяет отделить АПИ от реализации. И последнее в этом подразделе. Интерфейсы позволяют: 1. удобно создавать новые объекты, не связанные наследованием; 2. единообразно обращаться к методам объектов, не связанных наследованием; 3. полностью заменить наследование. ... 30 сек ... Конечно, полностью заменить наследование не получится, и ни о каком удобстве создания объектов речи не идёт, интерфейсы действительно позволяют единообразно обращаться к объектам, не связанным наследованием, но реализующим один интерфейс. как флешки-мышки и прочая ЮЗБи периферия, помните?
06-23	На несколько минут вернёмся к фону, пока мы его полностью не забыли, применим наши новые интерфейсные знания на практике. Фон наследуется от спрайта, но ему от спрайта вообще ничего не нужно, кроме двух методов, в которых он реализует своё собственное поведение, которое в свою очередь никак не коррелирует с тем, как ведут и что хранят в себе остальные спрайты. И вот сложилась ситуация в которой нам надо хранить в одном массиве спрайтов в основной программе очень похожие объекты но наследовать их друг от друга не совсем логично. Как поправить?

Экран	Слова
06-24	Напишем некий интерфейс, назовём его Interactable и скажем, что у него есть методы апдейт и рендер, без реализации. Вообще, если по хорошему, нужно было создавать два интерфейса, Updatable и Renderable, Чтобы иметь возможность отделить рисуемые объекты от обновляемых, но у нас же с вами пример, поэтому мы создадим единый интерфейс. То есть это будут некие объекты которые должны уметь рисоваться и обновляться. Идём в спрайт, и говорим, что мы реализуем интерфейс взаимодействуемого. Смотрим, что сломалось, кто навскидку скажет, почему? Не нужно судорожно хвататься за клавиатуру, но мысль правильная - модификатор должен быть паблик, потому что интерфейс по умолчанию содержит публичные методы, а сужать область видимости запрещено. И Фон тоже теперь у нас реализует интерфейс Interactable. При этом получается, то фон вообще никак не связан со спрайтом, у них даже набор полей разный. Но при этом, оба умеют рисоваться и апдейтиться, благодаря интерфейсу. Быстро поправив основное окно и логику, сменив массив спрайтов на массив интерактивностей можем запустить и увидеть, как снова весьма бодро по экрану летают наши шарики.
06-25	И вот мы подошли к самому главному, той гибкости, которую даёт нам работа с интерфейсами. Если вы обратите внимание, как развивается наше повествование по курсу, вы заметите, что сначала мы выходили за пределы одного метода, потом за пределы одного класса, затем за пределы одного пакета, за пределы программного кода, а теперь вовсе хотим написать код, который возможно будет использовать несколькими программами. Посмотрим со стороны, что мы тут написали. У классов канвы и спрайта, а также у интерфейса нет никакой специфики, их можно применять, по сути, где угодно, не только в этой конкретной программе с этими конкретными классами. Универсальные получились штуковины. Создадим какую-то условную вторую игру: новый пакет, новый класс, скопипастим туда немного кода от мячиков. И не писать же нам по новой спрайты и интерфейсы. сделаем правильное дробление по пакетам и станет очевидно, что у нас есть некий общий библиотечный пакет, и какие-то игры с конкретными реализациями. создадим пакет common, переносим туда канву, спрайт и геймобъект, и будем всё чинить.

Экран	Слова
06-26	Те кто делает это прямо сейчас, смотря эту лекцию в записи, могут обратить внимание, что общий пакет перенёсся вовсе без проблем, шарики перенеслись с минимальными изменениями, только публичные модификаторы понадобились. Во втором главном окне создали такой же конструктор, размеры, положение. И вот хотим воспользоваться общей канвой. Но не даёт же. Почему? мы же так классно всё придумали, никакой специфики. Но нет, канва то может принимать в конструкторе только мэйн окна из пакета кружочков. и канва уже у этого класса вызывает метод onDrawFrame(). Привязались к классу, связали себя по рукам и ногам сразу. Как это решается? это решается интерфейсом. нам надо написать какой-то интерфейс вроде canvasRepaintListener, который будет уметь ждать от канвы вызов метода и как-то по своему его реализовывать. А то у нас получается катастрофа - игры зависят от общего пакета, а по хорошему общий пакет вообще ничего не должен знать о том, какие игры он помогает создать. Создаём в общем пакете интерфейс с одним методом, который имеет сигнатуру из mainWindow. И переписываем канву так, чтобы она не какой-то класс с шариками на вход принимала, а canvasRepaintListener. и везде используем слушателя через интерфейс. Внимательно смотрите на ещё один фокус. Мы помним, что интерфейс может быть реализован классом, а наши окна - это тоже классы. И вот они наследуются от ДжейФрейма. Привычные нам наследование и полиморфизм утверждают, что это всё, что мы можем сделать. А мы нашим классам внезапно говорим что классы вы конечно хорошие, с прекрасным наследованием от фреймворка, но теперь ещё будете реализовывать интерфейс слушателя канвы. Следовательно, можете продолжать передавать себя в конструктор, а метод интерфейса у нас уже правильно реализован, с верной сигнатурой. Чтобы подчеркнуть, что это реализация интерфейса, допишем аннотацию оверрайд.
06-27	Насладимся летающими шариками в одном приложении, и летающими квадратиками в другом. Теперь на основе нашего очень хорошо отделённого интерфейсами общего пакета можно штамповать такие приложения практически без усилий. Понимаете, как разработчики в игровых студиях делают по сотне очень похожих игр в год?

Экран	Слова
06-28	Непосредственно об интерфейсах осталось сказать, что интерфейсы были значительно переработаны в джаве 1.8, туда добавили довольно много интересных механизмов, и об одном из них нельзя не сказать. Дефолтная реализация интерфейса. Как и всю остальную сегодняшнюю лекцию, хочу начать с примера применения. И пример я хочу построить на основе тех интерфейсов, которые у нас уже написаны - человек и бык. Понятно же, что именно у этих интерфейсов явно есть реализации по-умолчанию, например, для действия ходить, человек ходит на двух ногах, а бык на четырёх копытах. Есть и исключительные случаи, но они чаще всего связаны с чем-нибудь грустным, поэтому поступим логично и скажем, что у обоих интерфейсов будет реализация метода перемещения по-умолчанию. для этого понадобится ключевое слово дефолт. Обратите внимание, что в интерфейсе быка я не использовал это ключевое слово, и среда разработки мне подсказала, что так писать нельзя.
06-29	Какие особенности есть у реализующих методы по-умолчанию интерфейсов? Первое, и самое очевидное - отсутствие необходимости переопределять вообще все методы в классах, реализующих эти интерфейсы, что делает интерфейс чуть более похожим на класс. Реализованные по-умолчанию интерфейсы могут задействовать созданные в этом интерфейсе поля, а наличие в интерфейсе полей делает его ещё более похожим на класс. На слайде можно видеть, что написанные перед полями модификаторы были отмечены средой разработки как избыточные и необязательные к написанию. Если убрать ключевые слова стэтик или файнал - ничего не изменится, а вот если заменить паблик на, скажем, прайвэт - будет ошибка компиляции, поля интерфейса всегда публичные и явно их приватить - нельзя.
06-30	Программные интерфейсы открывают перед разработчиком широчайшие возможности по написанию более выразительного кода. Одна из наиболее часто используемых возможностей - анонимные классы.

Экран	Слова
06-31	Итак, мы уже неплохо знаем, что классы – это такой новый для нашей программы тип данных. Мы уже привычными движениями создаём публичные классы в отдельных файлах и пишем в них логику, но это не всё, что позволяет нам джава. Классы также бывают вложенными и внутренними. Внутренние классы - это классы, которые пишутся внутри класса, который описан в файле, мы его даже можем использовать и вызывать его методы. А также вложенные или локальные классы, которые мы можем объявлять прямо в методах, и работать с ними, как с обычными классами, но область видимости у них будет только внутри метода. Это мы уже тоже прошли. К чему я это веду? к анонимным классам. Анонимный класс, что довольно очевидно - это класс без названия. Создадим интерфейс, назовём его как-нибудь хорошо, например, MouseListener и опишем в нём пару методов, например mouseUp(), mouseDown(). Перейдя в основную часть программы можем написать, что некоторый локальный класс будет реализовывать интерфейс и значит должен переопределять все его методы. После чего мы привычно можем экземпляр этого класса создать и даже попользоваться его методами. Удобно, привычно, хорошо.
06-32	Очень часто, какие-то элементы управления (кнопчки, события входящих датчиков (клавиатура, мышка), сеть требуют на вход каких-нибудь обработчиков своих данных, которые будут внимательно слушать конкретный источник данных, отлавливать события и вообще знать что делать. Это всё делается через интерфейсы. С точки зрения программы, создаётся какой-то метод, например, добавления к кнопке слушателя, вы должны помнить, мы такие делали для кнопок в приложении, которое играет с пользователем в крестики-нолики, там как раз интерфейс назывался экшн листенер. Значит если мы в элемент управления в качестве слушателя передадим что-то, что реализует нужный интерфейс, то это нечто, очевидно, объект, начнёт ловить события и как-то их обрабатывать. Мы это делали и в летающих шариках, чтобы отвязать вызов метода объектом от знания о классе реализующем этот метод.

Экран	Слова
06-33	Придумаем для текущего примера какой-нибудь метод, принимающий на вход только что созданный МаусЛистенер. И в метод передаём объект, а внутри объекта для данного конкретного случая написали, как именно должна вести себя программа, когда кто-то нажал или отпустил кнопку мышки. Соответственно, весьма часто такие классы создаются не просто без названия экземпляра, но и вовсе без имени, прямо в аргументе методов. Ну и действительно, зачем ему имя, если он будет использован только один раз и только в этом методе? То есть, передать в метод экземпляр адаптера, реализующего маусЛистнер - несложно и привычно, мы это можем и понимаем. Вот допустим у нас есть параметр метода MouseListener m и нужно создать туда какой-то экземпляр, который реализует этот интерфейс. Но у нас уже есть класс, который это делает: маусАдаптер. Можем создать новый экземпляр адаптера, но без идентификатора, зачем он нужен, если адаптер сразу передаётся в метод и больше никак не используется? Вроде классно, избавились от одной лишней переменной, но можно и ещё лучше. Класс маусАдаптера идеально выполняет критерий С из принципов СОЛИД - сингл респонсибилити - делает только одно полезное дело - реализует интерфейс маус листнера. Но раз дело только одно - можно его выполнять и без размышлений о названии класса. Создадим сразу интерфейсную переменную и сохраним реализацию в неё. Для этого есть такой немного необычный синтаксис, который мы видим на 36й строке - new MouseListener, круглые скобки и дальше пишем интерфейс, который реализуем в фигурных скобках. Пустые круглые скобки намекают нам на то, что это вызов конструктора, а фигурные намекают на тело класса, то есть по сути мы пишем класс, где мы пишем реализацию и который тут же инстанциируем. Более формально, получается что мы создаём один экземпляр анонимного класса, который реализует интерфейс MouseListener. И конечно потом мы этот анонимный класс уже кладём в идентификатор. А можем даже не создавать интерфейсный идентификатор, а сразу передать реализующий экземпляр в аргумент метода. Получается, что мы в метод передаём новый экземпляр анонимного класса который РЕАЛИЗУЕТ ИНТЕРФЕЙС слушателя, и вот здесь же, как раз описание этого класса, мы в нём переопределяем соответствующие методы. Вот так это читается. Получается, что сейчас на слайде представлены все способы реализации интерфейса и передачи его в функцию при помощи анонимных классов. Ещё раз, анонимные классы это классы, не имеющие названия и реализующие какой-то интерфейс. Тут можно поговорить о лямбдах, в так называемых функциональных интерфейсах, то есть в интерфейсах, содержащих только один методд, для того чтобы сократить синтаксис, можно просто убрать то, что неизменно для этого анонимного клас-

Экран	Слова
06-34	Конечно, мы можем избежать использования анонимных классов, тем более, что часто они могут занимать довольно много места, сейчас поговорим как, но в чужом коде их использвоание можно встретить сплошь и рядом. Первое, что придумали делать - это адаптеры. Например, для панели на которой мы только что рисовали летающие шарики есть слушатель и метод добавления слушателя мышки, очень похожий на тот, который был только что создан в учебных целях, только не игрушечный. Если начать в аргументах этого метода писать «нью что-нибудь», идея предложит реализовать интерфейс маус листнера, но также предложит ещё один вариант - какой-то маус адаптер. Я напоминаю, что все исходные коды всего языка джава открыты, их можно и даже нужно читать время от времени. Так вот, если открыть исходники маус адаптера, можно увидеть, что это класс, реализующий несколько интерфейсов, но только формально, то есть все реализации пустые. Это позволяет нам вполне легально переопределять не все, а только некоторые методы интерфейса, значительно экономя место в коде приложения, если нам нужна реакция только на одно какое-то действие. Если попытаться это корректно прочитать по русски, должно получиться что-то вроде: создай новый экземпляр анонимного класса, который НАСЛЕДУЕТСЯ ОТ КЛАССА MouseAdapter, реализующего нужные тебе интерфейсы и переопредели вот этот метод. Остальные оставляй пустыми, потому что остальные действия можно игнорировать.
06-35	Как можно избежать таких многоэтажных и многострочных конструкций и при этом получить понятный код без лишних заморочек? Очень просто. Сказать, что класс, в котором мы в данный момент пишем код - реализует тот или иной интерфейс, благо интерфейсов можно реализовывать сколько угодно, в отличие от наследования от одного единственного родителя, и переопределить все методы. В некоторые из них даже можно написать реализацию, а туда, где требуется интерфейс - передать ссылку на себя самого. Мы же и слушатели мышки и слушатели действий и вообще что хочется.

Экран	Слова
06-36	Отвлечёмся на несколько вопросов 1. Программный интерфейс — это способ 1. рисования графических объектов 2. взаимодействия объектов 3. взаимодействия программы с пользователем ... 30сек... объекты знают за какие ниточки друг друга дёргать именно благодаря интерфейсам, то есть API. конечно же для взаимодействия объектов. 2. Анонимный класс — это класс без чего? 1. без интерфейса 2. без объектов этого класса 3. имени у самого класса ... 30сек... само слово анонимность предполагает неизвестность или полное отсутствие имени, поэтому анонимный класс - это когда без имени. 3. Поле в интерфейсе 1. невозможно создать 2. должно обязательно быть public static final 3. или private final ... 30сек... Как мы могли видеть на одном из слайдов - поля в интерфейсах публичные статические и неизменяемые. И последнее 4. Метод по-умолчанию 1. можно переопределять 2. можно не переопределять 3. можно использовать с полем интерфейса 4. все варианты верны ... 30сек... все варианты верны, метод по умолчанию - это удобство, а не дополнительные ограничения.
06-37	Осталось коротко поговорить о некоторых особенностях работы приложений, использующих графические интерфейсы.

Экран	Слова
06-38	Поскольку графический оконный интерфейс - это всегда многопоточность, да и привычного нам терминала под рукой нет, то тут сразу возникают особенности с обработкой исключений. Как ловить? Как показывать? Вот, например, есть у нас какое-то окно, на котором есть кнопка. У кнопки есть обработчик, в котором что-то идёт не так, скажем, выход за пределы массива. достаточно типичная ситуация. Возникает законный вопрос - как ловить? Посмотрите внимательно на этот слайд, на нём довольно много информации и его надо хорошо проанализировать. Дополнительно обратите внимание, как сделан обработчик - интерфейс слушателя действия реализован объектом класса основного окна и не захламляет конструктор. Если бы мы выбросили такое же исключение в консольном приложении, программа бы завершилась, здесь же мы видим, что несмотря на исключение в консоли окно всё ещё открыто и приложение работает. Если бы в обработчике выбрасывалось исключение, требующее обязательной обработки, всё понятно - обернули в трай кэтч и горя не знаем, но не тут то было, помните, что мы говорили о штатных и нештатных ситуациях? что если в случае исключения программу нужно завершить с исключением? Мы ясно видим, что исключения не завершают приложение. Да и если запускать приложение без среды разработки, куда выведется информация об исключении, в какой терминал, если терминала у пользователя не будет? Куда же можно положить обработчик исключения, чтобы всё точно было хорошо? Как именно мы можем поймать исключение, возникающее где-то в недрах графического фреймворка свинг? Если я поставил вас своими вопросами в тупик - это хорошо, таков был план, пошли разбираться.

Экран	Слова
06-39	Конечно, идеально было бы предварительно поговорить о многопоточности, но это отдельная сложная тема, рассмотрим её чуть позже. Достаточно того, что я уже несколько раз упоминал слово многопоточность в контексте графических интерфейсов и в контексте обработки исключений. Посмотрите пока что на правильный способ создания главных фреймов в свинге, для нас эта конструкция уже не должна быть чем-то сильно магическим, у класса свинг утилит есть статический метод, в который передаётся экземпляр анонимного класса, реализующего интерфейс, и в переопределяемом методе создаётся окно. А пока вы разглядываете эту конструкцию, упомяну ещё раз. Исключение происходит в таком специальном потоке, который называется EDT – Event Dispathing Thread. Этот поток совершает диспетчеризацию всех событий, происходящих во фреймворке свинг и является фактически генератором других потоков. Метод, который вы видите на экране заставляет объект фрейма не просто создаться, а явно создаёт его именно под управлением ЕДТ. Без этой конструкции тоже будет работать, но документация на свинг утверждает, что правильный способ именно такой. Вернуться на слайд назад Если внимательно посмотреть на текст исключения внизу экрана, становится видно, что исключение возникло в потоке со странным названием AWT-EventQueue-0. Наличие у потока номера говорит о том, что таких очередей событий у приложения может быть достаточно много, и в каких-то из них могут возникать исключения Вернуться обратно вперёд на два Подумаем чуть получше, где происходит исключение? в потоке. Как может называться обработчик не пойманных исключений? просто переведём с русского на английский и получим тр-эд.анкотэксепшнхэндлер. Совершенно внезапно обнаружим возможность реализовать интерфейс именно с таким названием. Интерфейс содержит один метод - непойманное исключение, который принимает на вход поток, в котором произошло исключение и объект исключения, которое произошло. Ну а если серьёзно, то такие обработчики уже написаны и встроены в среду исполнения джава, но они не очень умные, только и умеют, что в консоль писать стектрейс. Но мы то с вами уже крутые программисты, понимаем, что почти любое поведение можно переопределить, в том числе у обработчика исключений потока. И вот самое красивое – в конструкторе на 11й строке устанавливаем для потока обработчик непойманных исключений по-умолчанию, передаём себя, потому что мы и есть обработчик. В самой функции обработки, например, просто выведем на экран модальное окошко с текстом исключения, ронять приложение пока не будем, хоть и можем. Запустив увидим, что в консоли среды разработки прекрасная пустота, а это значит, что мы полностью перехватили контроль над обработкой исключений в этом окне и наша цель достигнута. Всё

Экран	Слова
06-40	Подведём некоторые итоги. На этой лекции были рассмотрены программные интерфейсы, что это, зачем они и как работают. Почему с ними неразрывно связано ключевое слово ‘implements‘, как осуществлять Наследование и множественное наследование интерфейсов, как интерфейсы помогают описывать множественное наследование классов, запрещённое в джаве. Реализовали немного интерфейсов, и даже рассмотрели как можно реализовать интерфейсы не полностью. Начали дословно понимать, что же именно написано в этих страшных многоэтажных обработчиках событий и что такое анонимные классы. Напоследок даже обработали одно исключение, возникающее на графическом интерфейсе пользователя.
06-41	В качестве домашнего задания нужно полностью разобраться с кодом, написанным на этой лекции, это слишком очевидно, даже не буду считать это за задание, поэтому предлагаю во первых, для приложения с шариками описать появление и убирание шариков по клику мышки левой и правой кнопкой соответственно, при этом пользоваться списками я настоятельно не рекомендую, работу с разного рода коллекциями мы рассмотрим на одной из следующих лекций, во вторых написать, выбросить и обработать такое исключение, которое не позволит создавать более, чем 15 шариков и третье, отмечу его аж двумя звёздочками, тут потребуется немного погуглить - описать ещё одно приложение, в котором на белом фоне будут перемещаться не шарики или квадратики, а изображения формата png, лежащие в виде файла в папке проекта.
06-42	Напоследок, соглашаясь с уважаемым древним мудрецом, напомню, что если чувствуете, что материал лекций слишком прост для вас, не ждите, пока станет сложнее, идите и сами изучайте то, что давно хотели, а на лекциях и семинарах мы будем подкладывать и подкладывать под эти знания какой-нибудь хороший фундамент. Развитие - это важно. Всем пока.