1. Специализация: ООП

Экран	Слова
Титул	Перейдём к интересному: что можно хранить в джаве, как оно там хранится, и как этим манипулировать
На прошлом уроке	На прошлом уроке мы рассмотрели базовый функционал языка, то есть основную встроенную функциональность, такую как математические операторы, условия, циклы, бинарные операторы. Также разобрали способы хранения и представления данных в Java, и в конце поговорили о способах манипуляции данными, то есть о функциях (в терминах языка называющиеся методами)
На этой лекции: классы и объекты; управление памятью; инкапсуляция; наследование; полиморфизм.	После разбора типов данных попробуем с помощью примеров разобраться в таких основополагающих в джава вещах, как классы и объекты, а также с тем, как применять на практике основные принципы ООП: наследование, полиморфизм и инкапсуляцию. Дополнительно поговорим об устройстве памяти в джава.
Класс	Начнём с базового, фундаментального понятия класс.
Чертёж (набросок рисунка) кота	Что такое класс? С точки зрения ООП, класс определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java. Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа, т.е. класс — это шаблон (чертёж), по которому создаются объекты (экземпляры класса). Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные.
Котик и стопки документов	Если мы хотим работать в нашем приложении с документами, то необходимо для начала объяснить приложению, что такое документ, описать его в виде класса (чертежа) Document. Рассказать какие у него должны быть свойства: название, содержание, количество страниц, информация о том, кем он подписан и т.д. В этом же классе мы описываем что можно делать с документами: печатать в консоль, подписывать, изменять содержание, название и т.д. Результатом такого описания и будет класс Document. Однако это по-прежнему всего лишь чертеж.

Экран	Слова
Буквы, написанные по трафарету чтобы было видно сам трафарет. документы, одинаковые по структуре и разные по содержанию.	Если нам нужны конкретные документы, а нам они обязательно нужны, то необходимо создавать объекты: документ №1, документ №2, документ №3. Все эти документы будут иметь одну и ту же структуру (название, содержание, ...), с ними можно выполнять одни и те же действия (печатать, подписать, ...) НО наполнение будет разным (например, в первом документе содержится приказ о назначении работника на должность, во втором, о выдаче премии отделу разработки и т.д.).
03-Структура	Начнём с малого, напишем свой первый класс. Представим, что нам необходимо работать в нашем приложении с котами. Java ничего не знает о том, что такое коты, поэтому нам необходимо создать новый класс (тип данных), и объяснить что такое кот. Создадим проект, его структура нам не в новизну, и будет иметь вид как на слайде, в мейне пока что простой хелловорлд, а вот с новым файлом кота пойдём разбираться, Создадим его для простоты в том же пакете, что и мейн
лайвкод 03-кот	Начнем описывать в классе Cat так называемый API кота. как мы все прекрасно помним, имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс Cat должен находиться в файле Cat.java. Пусть у котов есть три свойства: name (кличка), color (цвет) и age (возраст); совокупность этих свойств называется состоянием, и коты пока ничего не умеют делать. Класс Cat будет иметь следующий вид. (String name; String color; int age;) свойства класса, записанные таким образом в виде переменных называются полями
03-экземпляр-кота	Мы рассказали Java что такое коты, теперь если мы хотим создать в нашем приложении конкретного кота, а я напомню, что в 90% случаев мы сильно хотим создавать те или иные экземпляры, следует воспользоваться оператором new Cat(); в основном классе программы. Более подробно разберём, что происходит в этой строке, чуть позже. Пока же нам достаточно знать, что мы создали объект типа Cat (экземпляр класса Cat), и запомнить эту конструкцию. Для того чтобы с ним (экземпляром) работать, можем положить его в переменную, которой дать идентификатор cat1.

Экран	Слова
лайвкод 03-разные-коты	Припоминаем разницу между объявлением, присваиванием и инициализацией, становится понятно, что тут произошло и как ещё можно создавать котов. Припоминаем также, что в переменной не лежит сам объект, а только ссылка на него, подробнее, как и обещал, позже. Объект cat1 создан по чертежу Cat, и значит у него есть поля name, color, age, с которыми можно работать (получать или изменять их значения). Для доступа к полям объекта служит оператор точка, которая связывает имя объекта с именем поля. Например, чтобы присвоить полю color объекта cat1 значение ”Белый нужно выполнить следующий код: cat1.color = ”Белый”; Прошу, не запутайтесь, класс кота мы описали в отдельном файле, а создавать объекты и совершать манипуляции следует в мейне, не может же кот сам себе имя назначить
лайвкод 03-два-кота	Операция-точка служит для доступа к полям и методам объекта по его имени. Мы уже пользовались оператором точка для доступа к полю с длиной массива, например. Рассмотрим пример консольного приложения, работающего с объектами класса Cat. создадим двух котов, один будет белым барсиком 4х лет, второй чёрным мурзиком шести лет, и просто выведем информацию о них в терминал
03-класс-объекты1	Вначале мы создали два объекта типа Cat: cat1 и cat2, соответственно они имеют одинаковый набор полей name, color, age, почему? потому что они принадлежат одному классу, созданы по одному шаблону. Однако каждому из них мы в эти поля записали разные значения. Как видно из результата печати в консоли, изменение значения полей одного объекта, никак не влияет на значения полей другого объекта. Данные объектов cat1 и cat2 изолированы друг от друга. А значит мы делаем вывод о том, поля хранятся в классе, а значения полей хранятся в объектах.
Объекты	Как создавать новые типы данных (классы) мы разобрались, мельком посмотрели и как создаются объекты наших классов
Cat cat1; cat1 = new Cat();	Подробнее разберем как создавать объекты, и что при этом происходит. Создание объекта как любого ссылочного типа данных проходит в два этапа. Мы это помним из рассказа о массивах. Сначала создается переменная, имеющая интересующий нас тип, в нее мы можем записать ссылку на объект. Затем необходимо выделить память под наш объект, создать и положить объект в выделенную часть памяти, и сохранить ссылку на этот объект в памяти в нашу переменную. Для непосредственного создания объекта применяется оператор new, который динамически резервирует память под объект и возвращает ссылку на него, в общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new.

Экран	Слова
03-разные-коты	В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока ещё не ссылается на конкретный объект (первоначально значение переменной cat1 равно null). В следующей строке выделяется память для объекта типа Cat, и в переменную cat1 сохраняется ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat. Обычно новый объект создается в одну строку (Cat cat1 = new Cat()).
Оператор new [квалификаторы] ИмяКласса имяПеременной = new ИмяКласса();	Раз уж начали про объекты - нельзя не сказать про оператор new. Оператор new динамически выделяет память для нового объекта, общая форма применения этого оператора имеет вид как на слайде, но на самом деле справа - не имя класса, как могло показаться на первый взгляд ИмяКласса() в правой части выполняет вызов конструктора данного класса, который подготавливает вновь создаваемый объект к работе.
лайвкод 03-один-кот	Именно от оператора нью будет зависеть, сколько именно объектов будет создано в программе. На первый взгляд может показаться, что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти. Но это не так. На самом деле cat1 и cat2 будут ссылаться на один и тот же объект. Присваивание переменной cat1 значения переменной cat2 не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная cat2 содержит ссылку на тот же объект, что и переменная cat1.
03-один-кот-2	Таким образом, любые изменения, внесённые в объекте по ссылке cat2, окажут влияние на объект, на который ссылается переменная cat1, поскольку это один и тот же объект в памяти, как и в примере на слайде, где мы как будто бы указали возраст второго кота 6 лет, а при выводе, возраст 6 лет оказался и у первого кота. Это довольно легко себе представить как ярлыки для запуска одной и той же программы на рабочем столе и в меню Пуск. Или если мы на один и тот же шкафчик в раздевалке наклеим два номера - сам шкафчик можно будет найти по двум ссылкам
Метод - это функция, принадлежащая классу	Ранее мы уже говорили о том, что в языке Java любая программа состоит из классов и функций, которые могут описываться только внутри них. Именно поэтому все функции в языке Java являются методами. А метод, как мы помним, это - функция, являющаяся частью некоторого класса, которая может выполнять операции над данными этого класса.

Экран	Слова
03-нестатик	Метод для своей работы может использовать поля объекта и/или класса, в котором определен, напрямую, без необходимости передавать их во входных параметрах. Это похоже на использование глобальных переменных в функциях, но в отличие от глобальных переменных, метод может получать прямой доступ только к членам класса. Самые простые методы работают с данными объектов. Методы формируют АПИ классов, то есть способ взаимодействия с классами, интерфейс
Лайвкод 03-метод	Вернёмся к примеру с котиками. Все мы знаем, что котики умеют урчать, мяукать и смешно прыгать. В целях демонстрации мы в описании этих действий просто будем делать разные выводы в консоль, хотя мы и можем научить нашего котика выбирать минимальное значение из массива, но это было бы неожиданно. Итак опишем метод например подать голос и прыгать.
Лайвкод 03-метод-вызов	Обращение к методам выглядит очень похожим на стандартный способом, через точку, как к полям. Теперь когда мы хотим позвать нашего котика, он нам скажет, мяу, я имя котика, а если мы решили что котику надо прыгать, он решит, прилично-ли это - прыгать в его возрасте. Как видно, барсик замечательно прыгает, а мурзик от прыжков воздержался, хотя попрыгать мы попросили их обоих
Вопросы для проверки 1. Что такое класс? 2. Что такое поле класса?	Что такое класс? Класс – это шаблон группы объектов. Класс - это новый тип данных в программе Что такое поле класса? Поле (атрибут) класса — это характеристика объекта, также можно сказать, что это часть его состояния. Вопрос: На какие три этапа делится создание объекта? Создание идентификатора, Выделение памяти под наш объект, сохранение ссылки на объект в переменную.
шрифтом курьер слово static викисловарь - статика этимология	Теперь, когда мы более-менее хорошо знаем, что такое класс и объект, хотелось бы остановиться на специальном модификаторе - static, делающем переменную или метод ”независимыми”от объекта. Если формально, то: Static — модификатор, применяемый к полю, блоку, методу или внутреннему классу, он указывает на привязку субъекта к текущему классу. Для использования таких полей и методов, соответственно, объект создавать не нужно. В Java большинство членов служебных классов являются статическими.

Экран	Слова
— статические методы; — статические переменные; — статические вложенные классы; — статические блоки.	Мы можем использовать это ключевое слово в четырех контекстах: статические методы; статические переменные; статические вложенные классы; статические блоки. Рассмотрим подробнее только первые два пункта, о третьем поговорим чуть позже, а четвёртый потребует от нас знаний, выходящих не только за этот урок, но и за десяток следующих.
Статические методы - методы класса	Статические методы также называются методами класса, потому что статический метод принадлежит классу, а не его объекту. Что логично, нестатические называются методами объекта. Статические методы обладают интересным свойством - их можно вызывать напрямую через имя класса, не обращаясь к объекту и вообще объект не создавая. Что это и зачем нужно? Например, умение кота мяукать можно вывести в статическое поле, потому что, например, мы весной можем открыть окно, не увидеть ни одного экземпляра котов, но зато услышать их, и точно знать, что мяукают не дома и не машины, а коты
03-статические-поля	Аналогично статическим методам, статические поля принадлежат классу и совершенно ничего не знают об объектах. Важной отличительной чертой статических полей является то, что их значения также хранятся в классе, в отличие от обычных полей, чьи значения хранятся в объектах. На слайде довольно понятно это показано. Изображение на слайде именно в этом виде я настоятельно рекомендую если не заучить, то хотя бы хорошо запомнить, оно нам ещё пригодится. Из этой же картинки можно сделать несколько выводов, о которых сейчас поговорим
лайвкод 03-статическое-поле-код	Помимо того, что статические поля - это полезный инструмент создания общих свойств это ещ§ и опасный инструмент создания общих свойств. Так, например, мы знаем, что у котов четыре лапы, а не 6 и не 8. Не создавая никакого барсика будет понятно, что у кота - 4 лапы. Это полезное поведение
лайвкод 03-статическое-поле-ошибка	Посмотрим на опасность. Мы видим, что у каждого кота есть имя, и помним, что коты хранят значение своего имени каждый сам у себя. А знают экземпляры о названии поля потому что знают, какого класса они экземпляры. Но что если мы по невнимательности добавим свойство статичности к имени кота?

Экран	Слова
03-статическое-поле-признак	Создав тех же самых котов, которых мы создавали весь урок, мы получим двух мурзиков и ни одного барсика. Почему это произошло? По факту переменная у нас одна на всех, и значение тоже одно, а значит каждый раз мы меняем именно его, а все остальные коты ничего не подозревая смотрят на значение общей переменной и бодро его возвращают. Поэтому, чтобы не запутаться, к статическим переменным, как правило, обращаются не по ссылке на объект — cat1.name, а по имени класса — Cat.name.
03-статические-поля	К слову, статические переменные — редкость в Java. Вместо них применяют статические константы. Они определяются ключевыми словами static final и по соглашению о внешнем виде кода пишутся в верхнем регистре.

Экран	Слова
Вопросы для проверки 1. Какое свойство добавляет ключевое слово static полю или методу? (a) неизменяемость (b) принадлежность классу (c) принадлежность приложению 2. Может ли статический метод получить доступ к полям объекта? (a) не может (b) может только к константным (c) может только к неинициализированным	Какое свойство добавляет ключевое слово static полю или методу? неизменяемость принадлежность классу принадлежность приложению. Конечно, принадлежность классу, как следствие отсутствие необходимости создавать объект для таких полей и методов. Может ли статический метод получить доступ к полям объекта? не может - может только к константным - может только к неинициализированным. не может, потому что объект всегда знает какого он класса, а классы никогда не знают, какие именно объекты были от него созданы.
отбивка Введение в управление памятью	Понимая поверхностно, как организовано создание и хранение объектов, можем углубиться в эту тему. Это факультативная часть, выходящая достаточно далеко за рамки джуниор позиции.

Экран	Слова
03-память	Это глубокое погружение в управление памятью Java позволит расширить ваши знания о том, как работает куча, ссылочные типы и сборка мусора. Поможет лучше понять глубинные процессы и, как следствие, писать более хорошие программы. Для оптимальной работы приложения JVM делит память на область стека (stack) и область кучи (heap). Всякий раз, когда мы объявляем новые переменные, создаем объекты или вызываем новый метод, JVM выделяет память для этих операций в стеке или в куче. Далее будет представлена модель организации памяти в Java. Чуть позже мы её рассмотрим подробнее, а начнем со стека.
стопка блинов LIFO (Last-In, First-Out. Последний-зашел, Первый-вышел)	Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи. Данная память в Java работает по схеме LIFO (Последний-зашел-Первый-вышел).
НУЖЕН РЕКВИЗИТ ТРИ ТАРЕЛКИ С ПЕЧЕНЬЯМИ ИЛИ КОНФЕТАМИ	Кроме того, переменные в стеке имеют определенную область видимости. Программой используются только объекты из активной области. Например, если ЖВМ выполняет тело метода, она помещает весь используемый в методе контекст на стек и может получить доступ только к объектам из стека, которые находятся внутри тела метода. Она не может получить доступ к другим локальным переменным, так как они не выходят в область видимости. Когда метод завершается и возвращается результат его работы, верхняя часть стека выталкивается, и активная область видимости изменяется.
03-многопоточность	Немного забегая вперёд можно сказать, что все потоки, работающие в JVM, имеют свой стек. Для нас пока достаточно отождествлять поток и собственно исполнение нашей программы. Стек в свою очередь держит информацию о том, какие методы вызвал поток. Я буду называть это «стеком вызовов». Стек вызовов возобновляется, как только поток завершает выполнение своего кода. каждый слой стека вызовов содержит все локальные переменные для вызываемого метода и потока. Все локальные переменные примитивных типов полностью хранятся в стеке потоков и не видны другим потокам

Экран	Слова
на слайд перечисление справа	Особенности стека: — Он заполняется и освобождается по мере вызова и завершения новых методов; — Переменные на стеке существуют до тех пор, пока выполняется метод в котором они были созданы; — Если память стека будет заполнена, Java бросит исключение java.lang.StackOverflowError; — Доступ к этой области памяти осуществляется быстрее, чем к куче; — Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек.
03-устройство памяти	Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект. Неважно, был ли объект создан и присвоен локальной переменной или создан как переменная-член другого объекта, он хранится в куче. со всеми своими примитивными и не примитивными данными внутри. В случае, когда локальная переменная примитивного типа, она хранится на стеке потока.
03-началось	Локальная переменная также может быть ссылкой на объект. В этом случае ссылка (локальная переменная) хранится на стеке, но сам объект хранится в куче. Объект использует методы, эти методы содержат локальные переменные. Эти локальные переменные также хранятся на стеке, несмотря на то, что объект, который использует метод, хранится в куче. Переменные-члены класса хранятся в куче вместе с самим классом. Это верно как в случае, когда переменная-член имеет примитивный тип, так и в том случае, если она является ссылкой на объект. Статические переменные класса также хранятся в куче вместе с определением класса. В общем случае, эти объекты имеют глобальный доступ и могут быть получены из любого места программы.
— Young generation — Old (Tenured) Generation — Permanent Generation	Куча разбита на несколько более мелких частей, называемых поколениями: Young Generation — область где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора; Old (Tenured) Generation — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога «возраста», они перемещаются в Old Generation; Permanent Generation — эта область содержит метаинформацию о классах и методах приложения, но начиная с Java 8 данная область памяти была упразднена. В Java 8 PermGen заменён на Metaspace - его динамически изменяемый по размеру аналог. Важно упоминуть, что именно здесь живут статические поля.

Экран	Слова
Особенности кучи: список из правой части	— В общем случае, размеры кучи на порядок больше размеров стека — Когда эта область памяти полностью заполняется, Java бросает java.lang.OutOfMemoryError; — Доступ к ней медленнее, чем к стеку; — Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора; — В отличие от стека, который создаётся для каждого потока свой, куча не является потокобезопасной, поскольку для всех одна, и ее необходимо контролировать, правильно синхронизируя код.
Сборщик мусора	Вроде разобрались, разделили память так, как её разделяет ЖВМ. Предлагаю взглянуть на то, как ЖВМ осуществляет управление неиспользуемыми объектами
список справа	Но, прежде чем углубиться в детали, давайте сначала упомянем несколько вещей: — Этот процесс запускается автоматически Java, и Java решает, запускать или нет этот процесс; — На самом деле это дорогостоящий процесс. При запуске сборщика мусора все потоки в вашем приложении приостанавливаются (в зависимости от типа GC); — На самом деле это гораздо более сложный процесс, чем просто сбор мусора и освобождение памяти.
03-System.gc();	Несмотря на то, что Java решает, когда запускать сборщик мусора, нам доступен метод класса System, который мы можем явно вызвать и ожидать, что сборщик мусора будет запускаться при выполнении этой строки кода, верно? Это ошибочное предположение. Вы только как бы просите Java запустить сборщик мусора, но, опять же, Java решать, делать это или нет. В любом случае явно вызывать System.gc() не рекомендуется.

Экран	Слова
03-устройство памяти	Поскольку это довольно сложный процесс и может повлиять на производительность всего приложения, он реализован весьма разумно. Для этого используется так называемый процесс «Mark and Sweep» (отмечай и подметай). Java анализирует переменные из стека и «отмечает» все объекты, которые необходимо поддерживать в рабочем состоянии. Затем все неиспользуемые объекты очищаются. Фактически, чем больше мусора и чем меньше объектов помечены как живые, тем быстрее идет процесс. Чтобы сделать это еще более оптимизированным, память кучи состоит из нескольких частей, как показано на слайде.
03-молодое-поколение	Все новые объекты начинаются с молодого поколения. Как только они выделены в коде Java, они попадают в этот подраздел, называемый eden space. В конце концов пространство эдема заполняется объектами. На этом этапе происходит незначительная сборка мусора, так называемая minor collection. Некоторые объекты (те, на которые есть ссылки) помечаются, а некоторые (те, на которые нет ссылок) - нет. Те, которые были отмечены, затем переходят в другой подраздел молодого поколения под названием S0 пространства выживших (обратите внимание, что само пространство выживших разделено на две части, S0 и S1). Те, которые остались немаркированными, удаляются автоматической сборкой мусора.
03-выжившее-поколение	Так будет продолжаться до тех пор, пока пространство eden снова не заполнится; на этом этапе начинается новый цикл. События minor collection повторяются, но в этом цикле немного иначе. S0 был заполнен, и поэтому все отмеченные объекты, которые выживают как из пространства eden, так и из S0, фактически попадают во вторую часть пространства survivor, называемую S1. На приведенной на слайде диаграмме видно, что они помечены как из пространства выживших и в пространство выживших соответственно.

Экран	Слова
03-третье-поколение	Следует отметить одну очень важную вещь: любые объекты, попадающие в пространство выживших, помечаются счетчиком возраста. Алгоритм проверит это, чтобы увидеть, соответствует ли он пороговому значению для перехода в старое поколение. При следующей второстепенной сборке повторяется тот же процесс. Однако на этот раз пространства выживших переключаются. Объекты, на которые даны ссылки, перемещаются в S0. Главная мысль в том, что объекты не обязательно переходят из S0 в S1 пространства выживших. На самом деле, они просто чередуются с тем, куда они переключаются при каждой minor сборке мусора. Если эти процессы обобщить, то по существу все новые объекты начинаются в пространстве eden, а затем в конечном итоге попадают в пространство survivor, поскольку они переживают несколько циклов сборки мусора.
03-старое-поколение	Старое поколение можно рассматривать как место, где лежат долгоживущие объекты. По сути, если объекты достигают определенного возрастного порога после нескольких событий сборки мусора в молодом поколении, то затем они могут быть перемещены в старое поколение. Когда объекты собирают мусор из старого поколения, происходит крупное событие сборки мусора. Предлагаю рассмотреть, как выглядит продвижение из пространства выживших молодого поколения в старое поколение. В приведенном выше примере все выжившие объекты, достигшие возрастного порога в 8 циклов — и это всего лишь пример, поэтому специально не запоминайте это число — перемещаются алгоритмом в старое поколение. Старое поколение состоит только из одной секции, называемой постоянным поколением.
Постоянное поколение	Обратите внимание, что постоянное поколение не заполняется, когда объекты старого поколения достигают определенного порога, а затем перемещаются (повышаются) в постоянное поколение. Скорее, постоянное поколение немедленно заполняется JVM метаданными, которые представляют классы и методы приложений во время выполнения. JVM иногда может следовать определенным правилам для очистки постоянного поколения, и когда это происходит, это называется полной сборкой мусора major collection. Также, хотелось бы ещё раз упомянуть событие под названием остановить мир. Когда происходит небольшая сборка мусора (для молодого поколения) или крупная сборка мусора (для старого поколения), мир останавливается; другими словами, все потоки приложений полностью останавливаются и должны ждать завершения события сборки мусора.

Экран	Слова
Сборщик мусора. Реализации — последовательный — параллельный — CMS — G1 — ZGC	Стоит упомянуть, что JVM имеет пять типов реализаций GC: - Последовательный сборщик мусора. Это самая простая реализация GC, поскольку она в основном работает с одним потоком. В результате эта реализация GC замораживает все потоки приложения при запуске. Поэтому не рекомендуется использовать его в многопоточных приложениях, таких как серверные среды; - Параллельный сборщик мусора. Это GC по умолчанию для JVM, который иногда называют сборщиками пропускной способности. В отличие от последовательного сборщика мусора, он использует несколько потоков для управления пространством кучи, но также замораживает другие потоки приложений во время выполнения GC. Если мы используем этот GC, мы можем указать максимальные потоки сборки мусора и время паузы, пропускную способность и занимаемую площадь (размер кучи); - Сборщик мусора CMS. Реализация Concurrent Mark Sweep (CMS) использует несколько потоков сборщика мусора для сбора мусора. Он предназначен для приложений, которые требуют более коротких пауз при сборке мусора и могут позволить себе совместно использовать ресурсы процессора со сборщиком мусора во время работы приложения. Проще говоря, приложения, использующие этот тип GC, в среднем работают медленнее, но не перестают отвечать, чтобы выполнить сборку мусора. Здесь следует отметить, что, поскольку этот GC является параллельным, вызов явной сборки мусора, такой как использование System.gc() во время работы параллельного процесса, приведет к сбою или прерыванию параллельного режима; - Сборщик мусора G1. Сборщик мусора G1 (Garbage First) предназначен для приложений, работающих на многопроцессорных компьютерах с большим объемом памяти. Он доступен с обновления 4 JDK7 и в более поздних версиях. Сборщик G1 заменит сборщик CMS, поскольку он более эффективен; - Z сборщик мусора. ZGC (Z Garbage Collector) - это масштабируемый сборщик мусора с низкой задержкой, который дебютировал в Java 11 в качестве экспериментального варианта для Linux. JDK 14 представил ZGC под операционными системами Windows и macOS. ZGC получил статус production начиная с Java 15. ZGC выполняет всю дорогостоящую работу одновременно, не останавливая выполнение потоков приложений более чем на 10 мс, что делает его подходящим для приложений, требующих низкой задержки.

Экран	Слова
Итоги рассмотрения устройства памяти — куча доступна везде, объекты доступны отовсюду — все объекты хранятся в куче, все локальные переменные хранятся на стеке — стек недолговечен — и стек и куча могут быть переполнены — куча много больше стека, но стек гораздо быстрее	- Куча используется всеми частями приложения в то время как стек используется только одним потоком исполнения программы; - Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков; - Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче; - 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 java.lang.StackOverflowError, а если память кучи заполнена, то бросается исключение java.lang.OutOfMemoryError: Java Heap Space; - Размер памяти стека намного меньше памяти в куче. Из-за простоты распределения памяти (LIFO), стековая память работает намного быстрее кучи.
Вопросы для проверки По какому принципу работает стек? Что быстрее, стек или куча? Что больше, стек или куча?	По какому принципу работает стек? лифо. Что быстрее, стек или куча? стек Что больше, стек или куча? куча
КОНСТРУКТОР	После такого, копания в шестерёнках, время расслабиться и перейти к разбору конструкторов Java. Вернёмся к нашим барсикам.

Экран	Слова
03-два-кота	Чтобы создать объект мы тратим одну строку кода (Cat cat1 = new Cat()). Поля этого объекта заполнятся автоматически значениями по-умолчанию (целочисленные - 0, логические - false, ссылочные - null и т.д.). Нам бы хотелось дать коту какое-то имя, указать его возраст и цвет, поэтому мы пишем ещё три строки кода. В таком подходе есть несколько недостатков: во-первых, мы напрямую обращаемся к полям объекта (чего не стоит делать, в соответствии с принципами инкапсуляции, о которых речь пойдет чуть позже), а во-вторых, если полей у класса будет намного больше, то для создания всего лишь одного объекта будет уходить 5-10-15 строк кода, что очень громоздко и утомительно. Было бы неплохо иметь возможность сразу, при создании объекта указывать значения его полей.
03-плохой-конструктор лайвкод	Для инициализации объектов при создании в Java предназначены конструкторы. Конструктор - это частный случай метода в том смысле, что он тоже выполняет какие-то действия. Имя конструктора обязательно должно совпадать с именем класса, возвращаемое значение не пишется. Если создать конструктор класса Cat как показано на слайде, он автоматически будет вызываться при создании объекта. Теперь, при создании объектов класса Cat, все коты будут иметь одинаковые имена, цвет и возраст (это будут белые двухлетние Барсики).
03-параметризированный-конструктор лайвкод	При использовании конструктора из предыдущего примера, все создаваемые коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей наших объектов необходимо создать параметризированный конструктор. В приведенном на слайде примере, в параметрах конструктора используется первая буква от названия поля, это сделано для упрощения понимания логики заполнения полей объекта. И очень скоро будет заменено на более корректное использование ключевого слова this.

Экран	Слова
03-вызов-параметризированного-лайвкод	При такой форме конструктора, когда мы будем создавать в программе кота, необходимо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, вы определяете сами, то есть вы не обязаны заполнять все поля, которые есть в классе записывать в параметры конструктора, но при вызове обязаны заполнить аргументами все, что есть в параметрах, как при вызове метода. Наборы значений имён, цветов и возрастов будут переданы в качестве аргументов конструктора (n, c, a), а конструктор уже перезапишет полученные значения в поля объект (name, color, age). То есть начальные значения полей каждого из объектов будет определяться тем, что мы передадим ему в конструкторе. Как видите, теперь нам нет необходимости обращаться напрямую к полям объектов, и мы в одну строку можем инициализировать наш новый объект.
03-перегрузка-конструктора лайвкод	Мы можем как не объявлять ни одного конструктора, так и объявить их несколько. Также как и при перегрузке методов, имеет значение набор аргументов, не может быть нескольких конструкторов с одинаковым набором аргументов. В наш пример мы допишем конструктор, принимающий только один параметр - имя, а значит цвет будет неизвестен, а возраст, например, равен единице.
03-вызов-перегрузки лайвкод	В этом случае допустимы будут следующие варианты создания объектов - с полным набором параметров и только с именем. Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new. Не лишним будет напомнить что у классов всегда есть или лучше сказать всегда должен быть конструктор, даже если вы не пропишите свою реализацию конструктора.
03-конструктор-по-умолчанию лайвкод	Пока мы недалеко ушли от вызовов конструкторов, важно упомянуть, что как только вы создали в классе свою реализацию конструктора, пустой конструктор, называемый также конструктором по-умолчанию автоматически создаваться не будет. И если вам понадобится такая форма конструктора (в которой нет параметров, и которая ничего не делает), необходимо будет создать его вручную: public Cat() . То есть компилятор как-бы думает, что если вы не описали конструкторы, значит вам не важно, как будет создаваться объект, а значит, хватит пустого конструктора, ну а если конструктор написан, значит выкручивайтесь сами. (здесь можно показать конструкторы в скомпилированных класс-файлах)

Экран	Слова
ключевое слово this this - это указатель на текущий экземпляр класса. нужен когда одинаковые имена полей и параметров нужен чтобы вызвать конструктор из конструктора	Рассмотрим ещё один момент. Пару слайдов назад я упомянул о ключевом слове this, значит посмотрим, на часть ситуаций, в которых его используют. В контексте конструкторов, применять this нужно в двух случаях: — Когда у переменной экземпляра класса и переменной метода/-конструктора одинаковые имена; — Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора. на самом деле, это ключевое слово используется чаще, но в основном с этими целями, не так много, — всего два случая, в контексте конструкторов. Рассмотрим обе ситуации на примерах.
03-параметризированный-конструктор	Вспоминаем наш конструктор, видим, что переменные в параметрах называются не так, как проименованы поля класса. Многим программистом казалось это странным — вводить переменную с новым именем, если в итоге речь идет об одном и том же. Об имени, цвете или возрасте в классе Cat. Поэтому, разработчики языка задумались о том, чтобы удобно сделать использование одного имени переменной. Другими словами, зачем иметь два имени для переменной, обозначающей одно и то же.
03-параметризированный-конструктор-2	хотелось бы сделать как-то так. Но в этом случае возникает проблемный. У нас теперь две переменные, которые называются одинаково. Один String name принадлежит классу Cat, а другой String name находится в локальной видимости конструктора. А жвм как и любой другой электрический прибор всегда идёт по пути наименьшего сопротивления, когда не знает, какую переменную вы имеете ввиду. То есть, когда пишете строку name = name; Java берёт самую близкую name из конструктора и для левой и для правой части оператора присваивания, и получается, что вы просто присваиваете значение переменной name из конструктора, ей же. Что не имеет никакого смысла. Для решения этой проблемы и некоторых других было введено ключевое слово this, которое в данном случае укажет, что нужно вызывать переменную не конструктора, а класса Cat.
03-правильный-конструктор	То есть this сошлется на вызвавший объект, как было сказано в начале. В результате чего имя котика через конструктор будет установлено создаваемому объекту. Таким образом, здесь this позволяет не вводить новые переменные для обозначения одного и того же, что позволяет сделать код менее перегруженным дополнительными переменными.

Экран	Слова
03-один-из-другого	Второй случай частого использования this с конструкторами - вызов одного конструктора из другого. это может пригодиться когда у вас (как ни странно) несколько конструкторов и вам не хочется в новом конструкторе переписывать код инициализации, приведенный в конструкторе ранее. Все не так страшно как кажется. Напишем немного синтетический пример, который, тем не менее, должен многое объяснить. Здесь вызывается обычный конструктор с тремя параметрами, который принимает имя цвет и возраст, но, допустим, когда котята рождаются возраст им задавать смысла немного, поэтому нам может пригодиться и конструктор просто с именем и цветом, а зачем писать присваивание имени и цвета несколько раз, если можно вызвать соответствующий конструктор? Но на такой вызов есть ограничение, конструктор из конструктора можно вызвать только один раз и только на первой строке конструктора.
public Cat (Cat cat) this(cat.name, cat.color, cat.age);	Есть еще один вид конструктора - это конструктор копирования. Чтобы создать конструктор копирования, мы можем объявить конструктор, который принимает объект того же типа, в нашем случае котика, в качестве параметра, а в самом конструкторе аналогично конструктору, заполняющему все параметры, заполнить каждое поле входного объекта в новый экземпляр. Но у нас уже есть конструктор, который заполняет все поля, зачем нам очень похожий, спросите вы и будете правы (лайвкод) благодаря имеющемуся у нас конструктору со всеми нужными параметрами, с помощью ключевого слова this явно вызвать конструктор заполняющий все поля кота, значениями из переданного объекта, фактически, его копирующий. То, что мы имеем здесь, – это неглубокая копия, что удобно и довольно просто, поскольку все наши поля – int и два String в данном случае являются либо примитивными, либо неизменяемыми типами. Если класс Java имеет изменяемые поля, например, массивы, то мы можем вместо простой сделать глубокую копию внутри его конструктора копирования. При глубокой копии вновь созданный объект не должен зависеть от исходного, потому что мы создаем отдельную копию каждого изменяемого объекта, а значит, например, просто скопировать ссылку на массив будет недостаточно. Такая вложенность может быть любая и определяется поставленной задачей.

Экран	Слова
Вопросы для проверки Для инициализации нового объекта абсолютно идентичными значениями свойств переданного объекта используется - пустой конструктор - конструктор по-умолчанию - конструктор копирования Что означает ключевое слово this?	Для инициализации нового объекта абсолютно идентичными значениями свойств переданного объекта используется - пустой конструктор - конструктор по-умолчанию - конструктор копирования. Конечно же копирования Что означает ключевое слово this? это ссылка на текущий экземпляр класса, на используемый в данный момент объект
Инкапсуляция	Получше узнав о классах и объектах в джава, добавив для себя понимания организации этого добра в памяти, можно приступить к полному погружению в ООП. Начнём с Инкапсуляции.
Инкапсуляция - определение (от лат...)	Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными. То есть класс должен представлять собой ”черный ящик которым возможно пользоваться, но его внутренний механизм защищен от повреждений.
чёрный ящик, 03-useless-box	Как работает той или иной поисковик, которым вы пользуетесь? Как именно он ищет информацию по тем или иным словам, которые вы вводите? Почему одни результаты находятся сверху в результатах поиска, а не какие-то другие? Хотя мы и используем поисковые средства каждый день, но, мало кто знает ответы на эти вопросы. Но это и не важно. Ведь это никому не нужно знать, когда поисковики выполняются свою задачу - ведь только это и важно. Все это возможно благодаря одному из главных принципов объектно-ориентированного программирования — инкапсуляции. Изначальное значение слова «инкапсуляция» в программировании — объединение данных и методов работы с этими данными в одной упаковке (”капсуле”).

Экран	Слова
кот в мешке	В Java в роли упаковки-капсулы выступает класс. Класс содержит в себе и данные (поля класса), и действия (методы класса) для работы с этими данными. Также, можно встретить такое понятие как ”сокрытие”. Оно реализует два направления: сокрытие реализации и сокрытие данных. Хорошо, но с помощью чего достигается сокрытие? Все члены класса в языке Java - поля и методы - имеют модификаторы доступа. Мы уже сталкивались с модификатором public, создавая публичные классы и публичную точку входа в программу. public означает доступность отовсюду, обычно используется для методов. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.
03-МД-без-протекта-и-привата	Есть два модификатора, которые нам уже известны, это паблик и пекеж-приват, также известный как дефолтный, пакетный или отсутствующий модификатор. Что это значит? это значит, что вообще всё что мы пишем в джаве имеет уровень доступа и если мы не определили этот уровень явно, то джава отнесёт наши данные к уровню доступности внутри пакета. Про пакеты мы говорили в самом начале, и вот, если публичные классы находятся в одном пакете - им доступны поля и методы друг друга, имеющие модификатор пекеж-прайват.
03-МД-без-протекта	модификатор private определяет доступность только внутри класса, и предпочтительнее всех. Вообще, хорошая практика предоставлять минимальный доступ к переменным в своём классе, чтобы те, кто им пользуются не внесли какие-то неожиданные изменения в его работу. Возникает законный вопрос, как так получается, что мы пользуемся всякими сложными механизмами без особенно долгого осмысливания, как они устроены и на чем основана их работа? Как мы можем дёргать за все эти ниточки ЖРЕ, если максимальная приватность - это очень хорошо? Все просто: создатели всех библиотек предоставляют простой и удобный интерфейс к своим разработкам.
cat.name = ; cat.color = ”GREEEEEN”; cat.age = -12345; лайвкод	Внимательно рассмотрим класс котика, хотя, конечно, мы его писали, что может пойти не так? А не так здесь может быть очень многое. Например, кто-то может создать хорошего кота, а потом его переименовать, перекрасить в зелёный цвет или сделать ему отрицательный возраст. В результате в программе находятся объекты с некорректным состоянием. Коту такое явно не понравится.

Экран	Слова
03-кот-с-дефолтными-полями лайвкод	Какая допущена ошибка? Все поля находятся в пакетном доступе. К ним можно обратиться в любом месте пакета: достаточно просто создать объект Cat — и всё, у любого программиста есть доступ к его данным напрямую через оператор точки. Нам, создателям класса Кот, нужно как-то защитить наши данные от некорректного вмешательства извне. Во-первых, все поля необходимо помечать модификатором private.
03-кот-с-приватными-полями	Private — самый строгий модификатор доступа в Java. Если его использовать, поля класса Cat не будут доступны за его пределами. Теперь поля вроде защищены от посягательства других программистов, но есть проблема, доступ к ним закрыт “намертво” - в программе нельзя даже получить вес существующей кошки, если это понадобится. Это тоже довольно неудобно, в таком виде наш класс практически нельзя использовать.
03-публичные-геттеры лайвкод	Нам нужно решить вопросы с получением и изменением значений полей. На помощь к нам приходят геттеры и сеттеры. Название происходит от английского “get” — “получать” (т.е. “метод для получения значения поля”) и set — “устанавливать”. (написать по одному а далее сгенерировать геттеры и сеттеры идеей) Теперь у нас есть возможность получать данные и устанавливать их. Например, с помощью getColor мы можем получить значение окраса котика.
03-вызов-через-геттеры лайвкод	Теперь из другого класса (Main) есть доступ к полям Cat, но только через геттеры. Обратите внимание — у геттеров стоит модификатор доступа public, то есть они доступны из любой точки программы. А что, если нам хочется обновить возраст котика? Не проблема, тут можно использовать сеттер. У многих мог возникнуть вопрос, а в чём разница? Разница в первую очередь в том, что сеттер — это полноценный метод. А в метод, в отличие от поля, можно заложить необходимую логику проверки, чтобы не допустить неприемлемых значений. Например, можно не позволить назначение отрицательного числа в качестве возраста
03-убираем-лишние-сеттеры лайвкод	Важно, что создавая для класса геттеры и сеттеры мы не только получаем возможность накручивать на поля дополнительную логику, но и получаем возможность регулировать доступ к полям. Например, если в программе нужно запретить менять котикам окрас, то мы просто удаляем из класса соответствующий сеттер.

Экран	Слова
продублировать 3-26 (поля класса кот) лайвкод	Внимательно осмотрев класс кота мы приходим к выводу, что хранить возраст котов очень неудобно, потому что каждый год нужно будет обновлять это значение для каждого объекта кота в программе, а это может оказаться утомительно. Выходом может оказаться хранение не возраста, а неизменяемого параметра даты рождения и подсчёт возрастка каждый раз, когда его запрашивают. Ведь человеку, который запрашивает возраст котика, не интересно, каким образом получено значение, прочитано из поля или вычислено, ему важен конечный результат. Это сокрытие реализации.
03-хранение-дат-вместо-возраста лайвкод	Допишем метод получения возраста и поле года рождения, чтобы сделать инкапсуляцию более явной. Теперь видно сокрытие реализации - у кота нет поля возраст, но есть метод получения возраста, и тому, кто быудет вызывать этот метод совершенно нет дела как устроен котик внутри.
Вопросы для проверки перечислите модификаторы доступа инкапсуляция - это - архивирование проекта - сокрытие информации о классе - создание микросервисной архитектуры	перечислите модификаторы доступа. приват, дефолт (пэкприв), протект, паблик. инкапсуляция - это - архивирование проекта - сокрытие информации о классе - создание микросервисной архитектуры. Конечно же сокрытие сведений об устройстве класса
отбивка Наследование	Второй кит ООП после инкапсуляции - наследование
03-класс-пса лайвкод	Представим, что мы хотим создать помимо класса котиков, класс собачек. Данный класс будет выглядеть очень похожим образом, только он будет не мяукать, а гавкать, и заменим обоим животным прыжок на простое перемещение на лапках (буквально скопипастить кота и заменить название на пса, заменить прыжок на движение на будущее) Вроде выглядит неплохо, пока мы не сталкиваемся с необходимостью описать классы для целого зоопарка. Мы видим, что в наших классах есть очень много одинаковых полей и методов. А как мы помним из занятия по циклам - если что-то повторяется в коде несколько раз - это очень плохо и надо срочно от этого избавиться, чтобы не плодить сотни строк кода, в которых даже разбираться никто не будет.

Экран	Слова
Наследование в программировании - это Наследование в Java реализуется ключевым словом extends (англ. - расширять)	Тут нам на помощь приходит наследование. И кот и пёс являются животными, и у всех описываемых нами животных есть имя, возраст, окрас. И все описываемые нами животные могут бегать, прыгать, и откликаться на имя. Создадим так называемый родительский, он же базовый, класс, животное. И поместим в него общие поля.
03-класс-животное-поля лайвкод	Создав так называемый родительский класс, или суперкласс, и поместив в него поля, мы можем убрать поля из кота и пса. Если полей много, лаконичность описания родственных классов может быть весьма ощутимой. Так что же нужно, чтобы унаследовать какой-то класс? Чтобы унаследовать один класс от другого, нужно после объявления нашего класса указать ключевое слово extends и написать имя родительского класса. Реализуем котика и собачку
03-класс-животное-методы лайвкод	В целом, достаточно безболезненно можно перенести и одинаковые методы, ведь из информации о хранении всего в памяти мы помним, что методы хранятся в классах, а значит хранить их лучше в одном экземпляре, а не тиражировать одинаковые методы в разных классах. Перенос кода в родительский класс показал наличие проблемы. Что же с этими классами не так? Вспоминаем информацию о модификаторах доступа и находим неприятность. Модификатор private определяет область видимости только внутри класса, а если нам нужно чтобы переменную было видно ещё и в классах-наследниках, нужен хотя бы модификатор доступа по умолчанию. А если мы создаём класс наследник в каком-то другом пакете, то и он нам не подойдёт.
03-МД-полный	Пришло время резюмировать наши знания о модификаторах доступа и поговорить о последнем. К членам данных и методам класса можно применять следующие модификаторы доступа: private - содержимое класса доступно только из методов данного класса; public - есть доступ фактически отовсюду. default - содержимое класса доступно из любого места пакета, в котором этот класс находится. protected (защищенный доступ) содержимое доступно также как с дефолтным модификатором, но ещё и для классов-наследников;

Экран	Слова
03-класс-животное-перенос-конструктора лайвкод	Про поля и методы проговорили, теперь умеем наследоваться и весьма сокращать количество написанного кода за счёт ООП, пришло время вернуться к конструкторам. Надо же разобраться какой и когда вызывается и после кого. Рассмотрим интересный момент - если мы перенесём одинаковые конструкторы кота и пса в общий класс животного - наша программа снова перестанет работать, давайте разбираться, почему.
03-конструкторы-вызов лайвкод	Начнём мы с того, что при создании объекта в первую очередь вызывается конструктор его базового класса, а только потом — конструктор самого класса, объект которого мы создаем. То есть при создании объекта Cat сначала отработает конструктор класса Animal, а только потом конструктор Cat. Чтобы убедиться в этом — упростим конструкторы и добавим в конструкторы Cat и Animal вывод в консоль. Как мы помним, если в классе был написан код хотя бы одного конструктора, то конструктор по-умолчанию не создаётся. А без конструктора по-умолчанию, классы-наследники не понимают, что им вызывать.
03-конструкторы-коррекция лайвкод	Мы можем явно вызвать конструктор базового класса в конструкторе класса-потомка. Базовый класс еще называют “суперклассом”, поэтому в Java для его обозначения используется ключевое слово super. Здесь такое же ограничение, как и при вызове конструкторов данного класса - вызов такого конструктора может быть только один и быть только первой строкой.
03-класс-птица лайвкод	Создадим ещё один класс, наследника животного, чтобы использовать наследование как положено, по назначению. Внимательный зритель мог заметить, что наследование реализуется через ключевое слово extends, это значит расширять. Для нас важно, что класс-родитель расширяется функциональностью или свойствами класса-наследника. Это позволяет нам, например. добавить в птичку такое свойство как высота полёта и такой метод как летать, в дополнение к тому, что умеет животное.
03-дополнительное-поле лайвкод	В конструкторе птицы мы вызвали конструктор Animal и передали в него три поля. Нам осталось явно проинициализировать только одно поле — высоту полёта, которого в Animal нет. Ранее, мы говорили о том, что при создании объекта в первую очередь вызывается конструктор класса-родителя. Именно поэтому слово super() всегда должно стоять в конструкторе первым, иначе логика работы конструкторов будет нарушена и программа выдаст ошибку.

Экран	Слова
this vs. super	this и super - это два специальных ключевых слова в Java, которые представляют соответственно текущий экземпляр класса и суперкласс. Начинающие Java-программисты часто путают эти слова и обнаруживают слабую осведомленность об их специальных свойствах, о которых нередко спрашивают на интервью по Java Core. Вот, например, пара вопросов, из того, что сразу приходит на ум, Можно ли присвоить другое значение ключевому слову this в Java? и какая разница между ключевыми словами this и super в Java.
this() super() 03-родитель-в-наследнике	Как уже было сказано, главное отличие между this и super в том, что this представляет текущий экземпляр класса, в то время как super - текущий экземпляр родительского класса. Один из примеров использования переменных this и super — мы уже разбирали примеры вызовов конструкторов одного из другого, так называемые вызовы конструкторов по цепочке, это возможно благодаря использованию ключевых слов this и super. Внутри класса для вызова своего конструктора без аргументов используется this(), тогда как super() используется для вызова конструктора без аргументов, или как его ещё называют, конструктора по умолчанию родительского класса. Ну или как мы сделали ранее, вообще любой другой конструктор, передав ему соответствующие параметры.
this.field this.method() super.field super.method()	Ещё this и super в Java используются для обращения к переменным текущего экземпляра класса и его родителя. Вообще-то, к ним можно обращаться и без префиксов super и this, но только если в текущем блоке нет локальных переменных с такими же именами, в противном же случае использовать имена с префиксами придется обязательно, но это не страшно, поскольку в таком виде они даже более читабельны. Классическим примером такого подхода является использование this внутри конструктора, который принимает параметр с таким же именем, как и у переменной экземпляра.
большими буквами: множественное наследование запрещено	Пока разговариваем о текущих и родительских классах, нужно явно сказать, что у всей этой прекрасной объектной ориентированности есть одно важное ограничение: для каждого создаваемого подкласса можно указать только один суперкласс. В Java не поддерживается множественное наследование, то есть наследование одного класса от нескольких суперклассов. Зато возможно каскадное наследование, то есть класс-наследник вполне может быть чьим-то родителем. Пример такого поведения рассмотрим немного позже.

Экран	Слова
Главный класс Object 03-каскадное-наследование	Если класс-родитель не указан, таковым считается класс Object. Таким образом можно сделать вывод о том, что любой класс в джава так или иначе - наследник обжекта и соответственно всех его свойств и методов, мы ещё не раз будем возвращаться к этому примечательному факту. Вернёмся к нашим котикам, собачкам и птичкам. В этой иерархии классов можно проследить следующую цепь наследования: Object все классы неявно наследуются от него -> Animal -> Cat,Dog,Bird. Рассмотрим слайд и попробуем вспомнить о преобразовании примитивных типов.
03-через-общие-ссылки лайвкод	Суперклассы обычно размещаются выше подклассов, поэтому на вершине наследования находится класс Object, а в самом низу Cat Dog Bird. Объект подкласса представляет объект суперкласса, выражаясь проще мы можем ко всем котикам обращаться через общее название Животное и вообще ко всем объектам в программе мы можем обратиться через класс обжект. Поэтому в программе мы можем написать следующим образом: Object animal = new Animal(”Cat ”Black 3); Object cat = new Cat(”Murka ”Black 4); Object dog = new Dog(”Bobik ”White 2); Animal dogAnimal = new Bird(”Chijik ”Grey 3, 10); Animal catAnimal = new Cat(”Marusya ”Orange 1);
03-апкаст-даункаст лайвкод	Это так называемое восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии) или upcasting. Такое преобразование осуществляется автоматически. Обратное не всегда верно. Например, объект Animal не всегда является объектом Cat или Dog. Поэтому нисходящее преобразование или downcasting от суперкласса к подклассу автоматически не выполняется. В этом случае нам надо использовать операцию преобразования типов. Object animal = new Cat(”Murka ”Black 4); Cat cat = (Cat)animal; cat.move(); Обратите внимание, что в данном случае переменная animal приводится к типу Cat. И затем через объект cat мы можем обратиться к функционалу кота. Важно при этом, что изначально оператором new был создан объект кота, а не обжект или энимал
отбивка instanceof	Ещё один момент из темы преобразования типов - это оператор instanceof.

Экран	Слова
оператор instanceof возвращает истину, если объект принадлежит классу или его суперклассам и ложь в противном случае	Нередко данные приходят извне, и мы можем точно не знать, какой именно объект эти данные представляют. Соответственно возникает большая вероятность столкнуться с ошибкой преобразования типов. И перед тем, как провести преобразование типов, мы можем проверить, а можем ли мы выполнить приведение с помощью оператора instanceof
03-проверка-принадлежности лайвкод	Object cat = new Cat(”Murka ”Black 4); if (cat instanceof Dog) Dog dogIsCat = (Dog) cat; dogIsCat.jump(); else System.out.println(”Conversion is invalid”); Выражение cat instanceof Dog проверяет, является ли переменная cat объектом типа Dog. Но так как в данном случае явно кот не является собакой, то такая проверка вернет значение false, и преобразование не сработает. А вот выражение cat instanceof Animal выдало бы true и вывело бы Murka jumps (написать сравнение с животным)
отбивка ключевое слово final	помните, какую формулировку я просил запомнить? переменная с конечным значением
слайд из 2 про final добавить «класс с финальной реализацией, никаких других не будет»	Как хорошо, когда у нас есть возможность наследоваться от класса, забирать его свойства и поведение, но что, если нам стало нужно, чтобы от нашего класса никто больше не смог унаследоваться? На помощь приходит ключевое слово – final. Оно может применяться к классам, методам, переменным (в том числе аргументам методов). Нам на данный момент интересен момент с классом, как можно запретить наследование. Фактически мы говорим, что это класс с финальной реализацией, никаких других не будет
03-файнал-птица лайвкод	То есть, если мы логически понимаем, что у нас от птичек уже нельзя наследоваться, то мы их можем пометить этим ключевым словом и тогда, если мы в коде начнём писать класс например, попугайчика, и укажем, что наследуемся от птицы, то у нас выведется Cannot inherit from final.
отбивка Абстракция	иногда её даже выделяют как четвёртый принцип ООП, с чем я в корне не согласен

Экран	Слова
начальный эскиз животного желательно найти такую картинку, чтобы из одинакового кружка получались разные животные	Мы разобрались с наследованием в целом, с поведением конструкторов, с преобразованием типов и т.д. До беседы о полиморфизме осталось разобрать момент, который называется абстракцией. Есть мнение, что в ООП это означает, что при проектировании классов и создании объектов необходимо выделять только главные свойства сущности, и отбрасывать второстепенные, лично я с этим не согласен только лишь потому, что конкретно к ООП это отношения не имеет, а касается вообще всего программирования в целом. Так вот, абстрактный класс — это написанная максимально широкими мазками, о-о-о-чень приблизительная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в чистом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.
03-абстрактный-метод лайвкод	Но начнём мы не совсем с этого. Абстрактными могут быть не только классы, но и методы, это, кстати, порождает интересное ограничение, но об этом буквально через минуту. Абстрактный метод - это метод без реализации. Если внимательно посмотреть на наших животных, становится очевидно, что все они умеют издавать свой звук. Именно это поведение у них как бы общее, но для всех немного разное, нас это напрямую приводит к полиморфизму, но остановимся в шаге от него. Мы точно знаем, при проектировании животного, что все животные должны издавать звук, но не можем сказать, какой именно. Поэтому, мы говорим, что у животного есть метод издать звук, но реализацию этого метода в животном мы написать не можем, слишком мало сведений. Поэтому помечаем метод как абстрактный. Кому-то это может показаться, что это очень похоже на объявление функции в C++, да, вам не показалось.
03-абстрактное-животное лайвкод	Рассмотрим пример класса энимэл, что в нём изменилось, когда мы добавили в него абстрактные методы? верно, сам класс перестал быть верно описанным, компилятор утверждает, что что-то не так. А и правда, что будет, если программа попытается вызвать метод войс у животного, реализации-то нет? Прежде всего, класс энимал максимально абстрактно описывает нужную нам сущность — животное. Слово abstract на это недвусмысленно намекает. В мире не существует «просто животных». Есть губки, иглокожие, хордовые и т.д. Данный класс теперь слишком абстрактный, чтобы программа могла с ним нормально взаимодействовать, а значит просто является чертежом по которому будут создаваться дальнейшие классы животных. Отметим этот факт явно, написав ключевое слово abstract у класса, теперь компилятору всё нравится

Экран	Слова
абстрактный метод - это метод не содержащий реализации (объявление метода) абстрактный класс - класс содержащий хотя бы один абстрактный метод абстрактный класс нельзя инстанциировать	Что произошло? а произошло буквально следующее - мы сказали, что животное теперь слишком абстрактно, недостаточно детализирует объекты в нашей программе, Какого оно вида, к какому семейству относится, какие у него характеристики — непонятно. Было бы странно увидеть его в программе. Никаких «просто животных» в природе не существует. Только собаки, кошки, лисы и кто там ещё... В общем, мы взяли и просто-напросто запретили создавать экземпляры таких животных (можно ещё встретить термин инстанциировать). Очевидно, что абстрагирование метода вынуждает нас абстрагировать класс, но не наоборот, абстрактный класс необязательно должен содержать абстрактные методы, фактически, это просто запрещение создания экземпляров

Экран	Слова
Вопросы для проверки 1. Какое ключевое слово используется при наследовании? (a) parent (b) extends (c) как в C++, используется двоеточие 2. super - это (a) ссылка на улучшенный класс (b) ссылка на расширенный класс (c) ссылка на родительский класс 3. Не наследуются от Object (a) строки (b) потоки ввода-вывода (c) ни то ни другое	Какое ключевое слово используется при наследовании? parent - extends - как в C++, используется двоеточие. экстендс super - это - ссылка на улучшенный класс - ссылка на расширенный класс - ссылка на родительский класс. конечно родительский Не наследуются от Object - строки - потоки ввода-вывода - ни то ни другое. верный ответ ни то ни другое, потому что всё, кроме примитивов наследуется от обжекта

Экран	Слова
отбивка Полиморфизм	Наконец, к теме, которая вызывает у новичков отторжение, но является основой для самых мощных инструментов программирования в ООП стиле.
полиморфизм – это возможность объектов с одинаковой спецификацией иметь различную реализацию (Overriding)	полиморфизм выражается возможностью переопределения поведения суперкласса (часто можно встретить утверждение, что при помощи перегрузки, но это мы обсудим позже). Основная суть в том, что в классе-родителе имеется некоторый метод, но реализация этого метода разная у каждого класса-наследника, фактически это и есть полиморфизм, нам осталось его только правильно оформить и рассмотреть в деталях, как мы любим.
03-полиморфизм-переопределение лайвкод	Что мы сделали? В наших классах-потомках мы определили такие же методы, как и объявленный метод класса родителя, который хотим изменить. Пишем в нем новый код. И все – как будто старого метода в классе-родителе и не было. У нас есть класс Animal, у которого есть метод voice(), мы наследуемся от него, создав класс котика и хотим назначить ему мяуканье.
Аннотации реализуют вспомогательные интерфейсы. Аннотация @Override проверяет, действительно ли метод переопределяется, а не перегружается	Важный момент, вы могли заметить в примерах наследования было уже переопределение из абстрактных классов, и там следует добавить так называемую аннотацию @Override. Она помогает компилятору понять, что в этом месте мы собираемся переопределить, а не перегрузить поведение предка. Если мы ошиблись в сигнатуре метода, то компилятор нам сразу об этом скажет. Настоятельно рекомендую всегда использовать данную аннотацию.
03-змейка лайвкод	То есть видим, что полиморфизм используется когда нам нужно описать поведение абстрактного класса или в целом назначить разным наследникам разное поведение, одинаково названное в родителе. Но есть и ситуации, когда все классы делают что-то одинаково, а один делает это как-то иначе. Создадим класс змейка. По очевидным причинам змейка не может ходить на лапках. Поэтому это поведение у змейки будет переопределено. Видим, что перемещение всех животных нас устраивает, а змейка делает это по-своему. Также, например, можно было создать черепаху, которая не умеет бегать, рыбу, которая не издаёт звуков, слона, который не умеет прыгать, в отличие от остальных, практикующих «среднее» поведение

Экран	Слова
03-хайдинг лайвкод	Стоить помнить, что переопределять можно только нестатические методы. Статические методы не наследуются в привычном смысле и, следовательно, не переопределяются. Можем припомнить, что создавать объекты котиков и птичек мы можем под любыми идентификаторами как самих соответствующих классов, так и их родителей. Создав в животном и коте статический метод с одинаковой сигнатурой мы сможем наблюдать то, что называется хайдингом, иначе сокрытием или перекрытием. Энимал статик войд идентифай()(энимал) Кэт статик войд идентифай()(кэт) видим, что класс унаследовался и метод должен переопределиться, внешне если создать энимал а и кэт ц, будет похоже, что так и произошло, но если сделать энимал а и энимал ц то видим что поведение изменилось. Это довольно очевидная вещь, если подумать. Статические члены класса относятся к классу, т.е. к типу переменной. Поэтому, логично, что если Cat имеет тип Animal, то и метод будет вызван у Animal, а не у Cat.
Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов параметрический полиморфизм и ad-hoc-полиморфизм Широко распространено определение полиморфизма, приписываемое Бьёрну Страуструпу: «один интерфейс — много реализаций»	Полиморфизм - это гораздо более широкое понятие, чем просто переопределение методов, в эту тему завязаны разные интересные теории типов и информации, множество парадигм программирования и другое. Мы же подойдём к этому вопросу с утилитарной точки зрения, один интерфейс - множество реализаций, как утверждает автор языка C++ мы рассмотрели, остался ещё один вариант, который тем не менее не дотягивает до истинного полиморфизма, но и до него мы доберёмся, когда будем говорить об обобщённом программировании.

Экран	Слова
к полиморфизму также относится перегрузка методов (Overloading)	Использование более одного метода с одним и тем же именем, но с разными параметрами в одном и том же классе или между суперклассом и подклассами в Java называется перегрузкой (Overloading). То есть, внешне выглядит, будто используется один метод вместо множества, выполняющих аналогичные действия. (в среде программирования) помним, что можно выводить в терминал как строки, так и цифры, также и символы, неужели строгая типизация сломалась и можно в один метод записать разные типы данных? нет на самом деле в классе System.out, который фактически PrintStream, есть довольно много перегрузок функции печати.
03-перегрузка-перемещения лайвкод	Перегрузка работает также, как работала без всех этих сложностей с ООП, ничего нового, но для порядка стоит создать возможность коту перемещаться не только абстрактно, но и на какое-то конкретное место или на какое-то конкретное количество шагов. void jump() void jump(String place) void jump(int count). Как видно, методы имеют одинаковые названия, но отличаются по количеству параметров и их типу. Их как и раньше можно использовать и вызывать. Просто в зависимости от введённого количества и типа параметров, будет выполняться соответствующее тело метода
абстрагируйте классифицируйте снова абстрагируйте инкапсулируйте	В итоге, чтобы стиль вашей программы соответствовал концепции ООП и принципам ООП java следуйте следующим советам: выделяйте главные характеристики объекта; выделяйте общие свойства и поведение и используйте наследование при создании объектов; используйте абстрактные типы для описания объектов; старайтесь всегда скрывать методы и поля, относящиеся к внутренней реализации класса.

Экран	Слова
Вопросы для проверки 1. Является ли перегрузка полиморфизмом (a) да, это истинный полиморфизм (b) да, это часть истинного полиморфизма (c) нет, это не полиморфизм 2. Что обязательно для переопределения? (a) полное повторение сигнатуры метода (b) полное повторение тела метода (c) аннотация Override	Является ли перегрузка полиморфизмом да, это истинный полиморфизм - да, это часть истинного полиморфизма - нет, это не полиморфизм. правильный ответ - это часть истинного полиморфизма, хотя если смотреть на этот вопрос узко, это совершенно точно не полиморфизм в ad-hoc понимании Что обязательно для переопределения? полное повторение сигнатуры метода - полное повторение тела метода - аннотация Override. Правильный ответ - это полное повторение сигнатуры, потому что смысл как раз в изменении тела, а аннотация оверрайд - это вспомогательный инструмент, который точно не является обязательным

Экран	Слова
на этом уроке мы	на этой лекции 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
ДЗ перечисление справа	1. написать класс кота так, чтобы каждому объекту кота присваивался личный порядковый целочисленный номер 2. написать классы кота, собаки, птицы, наследники животного. У всех есть три действия: бежать, плыть, прыгать. Действия принимают размер препятствия и возвращают булев результат. Три ограничения: высота прыжка, расстояние, которое животное может пробежать, расстояние, которое животное может проплыть. Следует учесть, что коты не любят воду. 3. * добавить механизм, создающий 25% разброс значений каждого ограничения для каждого объекта В качестве домашнего задания попробуйте описать класс кота таким образом, чтобы каждому объекту кота присваивался личный порядковый целочисленный номер. написать классы кота, собаки, птицы, наследующиеся от животного. У всех животных должено быть три действия: бежать, плыть, прыгать. Действия должны принимать целочисленный параметр размера препятствия и возвращать булев результат - успешно или неуспешно оно было преодолено. Соответственно, три ограничения: высота прыжка, расстояние, которое животное может пробежать, расстояние, которое животное может проплыть. Следует учесть, что коты не любят воду. в качестве задания со звёздочкой, добавить механизм, создающий 25% разброс значений каждого ограничения для каждого объекта, то есть если у кота максимальное расстояние бега 100 единиц, то каждый барсик и мурзик должен создаваться с ограничениями от 75 до 125 единиц.
Надо много учиться, чтобы знать хоть немного. Шарль Луи Монтескье	на этой ноте мы с вами возьмём паузу до следующей лекции, не забывайте быть умничками и беречь себя