Java 2.0

BOOKS

* Хорстманн Java. Библиотека профессионала Т1 (79)(215-265)(302-325)(350)
* Шилдт Java8. Полное руководство (198)(286-438)(470-563)(564)
* Файн Java24 (83 - 114)(127-150)(171)

SITES

* Java course (http://java-course.ru/begin/class\_description\_ext/)( http://java-course.ru/begin/simpleclasses/)
* Metanit (https://metanit.com/java/tutorial/3.12.php) (<https://metanit.com/java/tutorial/3.8.php>) (https://metanit.com/java/tutorial/8.1.php)

VIDEO

* Файн (<https://www.youtube.com/playlist?list=PLkKunJj_bZefB1_hhS68092rbF4HFtKjW>)(5-6)(8 – 0m)
* Уроки java (<https://www.youtube.com/user/husivm/videos>) (44-95)(96 – 0m)(328 – 0m)
* Мирончик (<https://www.youtube.com/playlist?list=PL27FCF69EBF48228B>) (1.2 – 2.25m)

**Основы ООП**

* Java характеризуется как язык: простой, объектно-ориентированный (в центре внимания объекты и их интерфейсы), распределенный, надежный, безопасный, не зависящий от архитектуры компьютера, переносимый, интерпретируемый, высокопроизводительный, многопоточный, динамичный
* Основные черты языка ОПП: 1) все является объектом (объект как усовершенствованная переменная) 2) программа это группа объектов указывающих друг другу что делать посредством сообщения (вызовов методов) 3) каждый объект имеет собственную память, состоящую из других объектов 4) у каждого объекта есть свой тип, т.е. он является экземпляром какого-либо класса 5) все объекты определенного типа могут получать одинаковые сообщения
* Объект обладает состоянием, поведением и индивидуальностью
* Принципы ООП: инкапсуляция – механизм, связывающий код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства. Скрывает подробности реализации. Наследование – один объект получает свойства другого. Обеспечивает принцип иерархической классификации. Полиморфизм – позволяет использовать один и тот же интерфейс для общего класса действий.
* Раньше считалось что нужно придумать алгоритм, а потом структуру для хранения информации, теперь информация хранится в классе
* Основой для ООП является декомпозиция – разделение задачи на крупные блоки, потом на мелкие.
* Файл – единица компиляции
* 4 строительных блока: класс, интерфейс, енум, аннотации
* Класс – логическая структура (шаблон, описание объекта), объект (экземпляр класса) – его реализация, переменные в классе – переменные экземпляра
* Любое понятие необходимое в java следует инкапсулировать в пределах класса
* Аннотация – описание метаданных
* Методы описывают поведение, а параметры состояние
* Наследование происходит посредством extends
* Наследуемые объекты не имеют доступа к private членам суперкласса
* Все классы неявно наследуются от Object
* В иерархии наследования вызов конструкторов идет от базового класса
* В конструкторе использование super – вызывает конструктор базового класса (должен быть первым вызовом в конструкторе), затем можно обратиться ко всем (не private) элементам, а this вызывает другой конструктор из этого класса (this и super в одном конструкторе недопустимы), и даже если super() не указан, то его добавит компилятор, чтобы дойти до Object (т.е. если не указать super, то будет вызван конструктор по умолчанию)
* По сути конструктор можно сравнить с static методом
* Наблюдается необычное поведение если в конструкторе вызвать метод, который переопределен, то в родительском конструкторе вызовется версия из подкласса, но поля будут иметь значения родительского
* Обратиться к классу родителю можно тоже через super (нужно например при переопределении метода если нужно вызвать метод из родителя)
* super это не аналог ссылки this, он просто сообщает компилятору что нужно вызвать метод из родительского класса (а this нельзя присвоить другое значение, т.к. она объявлена как final)
* Существует 3 варианта отношения между классами: Ассоциация, Агрегация и композиция, Обобщение и расширение (наследование). Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. Агрегация — отношение когда один объект является частью другого. Композиция —отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то (т.е. это более жесткий вариант Агрегации, отношение один к одному (машина и двигатель например))
* Или по-другому: есть три степени отношения между классами: зависимость – использует что-то, агрегирование – содержит что-то, наследование – является чем-то. Прежде чем применять наследование, стоит обратить внимание на композицию.
* если объект специализируется на чем-то (например ящик апельсинов), то это уже шаблон и тут нужны генерики
* При переопределении метода все-таки должно учитываться и возвращаемое значение (хотя в некоторых случаях это допустимо, т.к. возвращаемое значение не входит в сигнатуру метода), ковариантность возвращаемых значений - подразумевается, что переопределённый метод может вернуть тип, производный от типа, возвращаемого методом базового класса, тоже самое касается exception которые методы могут сгенерить
* Т.к. на поиски вызываемого метода уходит довольно долго времени, то VM заранее создает таблицу методов, в которой перечисляются все методы с сигнатурами
* При переопределении метода мы можем только расширять модификаторы доступа (и если например указать у родителя public, то и метод в потомке должен быть public, иначе будет ошибка)
* Так же мы не можем сужать throws Exception при переопределении метода (касается checked exeption)
* Одно из применений статических методов – фабричные методы. Одна из причин их использования это то что конструкторы нельзя назвать произвольным именем, так же при пользовании конструктором тип объекта фиксирован, а фабричные метод может возвращать например класс родителя
* Если указать public final class, то мы запретим его наследование, если указать final перед методом, то мы запретим его переопределение. Все методы конечного класса являются конечными
* Порядок перегрузки методов: 1) сначала все пытается привести к целочисленному (с расширением) 2) затем к дробному 3) затем к оберткам 4) затем к Object 5) в итоге к массиву …
* Порядок инициализации класса: 1) инициализируются статические поля 2) вызываются статические блоки инициализации 3) все поля инициализируются значениями 4) вызывается родительский конструктор 5) выполняются блоки инициализации в порядке их следования 6) если указан то вызывается другой конструктор 7) выполняется конструктор
* Рекомендации по разработке классов: 1) всегда хранить данные в переменных private 2) инициализировать поля явно с помощью конструктора 3) не употреблять в классе много простых типов (если есть несколько связанных простых типов нужно объединить их в класс) 4) разбивать на части слишком крупные классы 5) выбирать для классов и методов осмысленные имена 6) отдавать предпочтение неизменяемы классам (таким как LocalDate, который возвращает новый объект при изменении). Часто возникает проблема с изменяемыми классами в потоках

**Основы**

* В состав JRE входит JVM и ClassLibraries (основная из них rt.jar), lib/tools.jar – создает байт код (причем мы всегда можем сбросить версию байт-кода до java предыдущих версий)
* Переменной CLASSPATH не должно быть (она не желательна)
* В состав JVM входит интерпретатор (Interpreter), JIT компилятор (компилирует байт-код в машинный и затем исполняет), Верификатор (Verifier), сборщик мусора (Garbage collector)
* При старте JVM стартует 3 основных загрузчика классов: bootstrap(загружает внутренние классы), загрузчик расширений (загружает классы из <JAVA\_HOME>/lib/ext), системный загрузчик (загружает классы из параметра –classpath или –cp)
* Java Core = Java SE
* Java это не только язык, это набор технологий
* пакет java.lang автоматически подключается к выполняемой программе
* до 1.4 язык был упрощен от C++, была убрана арифметика указателей, множественная наследовательность, но потом с 1.5 опять стали усложнять, добавили типизированные коллекции, программирование с помощью метаданных
* если в классе использовать просто скобки {}, то получится инициализатор, вызывается один раз при создании объекта причем до вызова конструктора, но на самом деле после вызова super в конструкторе
* Программу можно запустить без метода main указав в блоке static {} и дописав в конце System.exit(0);
* Апплеты запускаются без main, но на самом деле запускается appletRunner, а уже он запускает апплет (на апплетах построен oracle forms)
* Классы пишутся с большой буквы и имеются существительным, интерфейсы – прилагательным
* Методы пишем маленькими буквами, так же в соответствии с конвенцией JavaBEAN должны быть геттеры и сеттеры (get, set) а если метод возвращает Boolean то возможно использовать is..
* Имя файла должно совпадать с именем класса, причем класс должен быть public и хотя в файле можно объявить несколько классов, но public должен быть только один (остальные будут доступны только в этом пакете, это бывает очень удобно, своеобразная инкапсуляция на уровне классов)
* Явная загрузка классов Class.forName()
* Каждый загрузчик имеет свое пространство имен, поэтому если классы одинаковы и загружены разными загрузчиками, то они считаются разными. Таким образом можно например создать два синглтона
* В EE входит стандартная java + описание как сервисы должны работать (по сути мы можем запихнуть в jvm некоторые сервисы и приложение будет работать внутри них), есть много решений от разных компаний
* По сути ME, SE, EE отличается набором классов в виртуальной машине
* Объекты создаются через new и по сути мы вызываем конструктор
* Порядок вызовов конструкторов при наследовании всегда от базового класса, причем если в наследнике в конструкторе не указан super, то будет вызван по умолчанию
* Типы делятся на примитивные и ссылочные, но в методы все передаются по значению
* методы работают в контексте экземпляра класса, в каждый метод неявно передается ссылка this (это ссылка на текущий экземпляр)
* до java8 методы существовали только в контексте экземпляра, с появлением лямбды теперь методы могут быть отдельно (называем их функциями)
* Все переменные в методе должны быть инициализированы, пусть даже null иначе не будет компилироваться
* Но поля класса неявно инициализируются [0, null, false]
* инициализировать поля класса необязательно константой, можно и вызовом метода
* в метод можно передать произвольное число параметров, в этом случае ставим "…" (int getSum(int ...nums)), а далее пробегаемся как for(int x:nums), по сути происходит автоупаковка int[] nums. Это является необязательным параметром и поэтому если метод принимает несколько параметров, то необязательный ставим последним
* локальные переменные метода если называются одинаково, то перекрывают переменные экземпляра
* Память ПК делится на: регистры, стек, куча, постоянная память (если переменные в коде записаны), неоперативная память (потоковая, либо на диске)
* Примитивы и ссылки хранятся в стеке, объекты в куче. На каждый поток создается стек
* void – пустой тип
* Базовые типы данных : boolean(размерность зависит от VM) , byte (127), short (32767), int (2\_147\_483\_648), long (8 байт), float (4 байта), double (8байт), char (2 байта)
* Размер данных строго фиксирован для всех операционных систем
* Прикол в том, что int / int = int (т.к. результат справа приводится всегда к самому вместительному типу, чтобы исправить делаем 1 / (double) 2 )
* Значение с плавающей точкой автоматически интерпретируется как double, поэтому чтобы привести к float используют псевдонимы float fl = 36.6f;
* приведение типов указывается как a = (aType) b; при этом сохраняется возможность потери точности или усечения значения. Так же приведение типов можно выполнять только в иерархии наследования
* Есть три специальных значения с плавающей точкой +/- бесконечность (для double Double.NEGATIVE\_INFINITY и Double. POSITIVE\_INFINITY) и NaN (Double.NaN), вещественные можно делить на ноль, получим бесконечность, а ноль на ноль даст нам NaN (проверить можно только как Double.isNaN, т.к. nan никогда не равен самому себе)
* char (описывается в UTF16) можно делать арифметические операции и он беззнаковый ( и если записывать char в Юникоде, то преобразование будет во время синтаксического анализа кода). Способы описания символа: число, в ' ', шестнадцатеричное число после \u ('\u0077'), существую так же синонимы к некоторым символам : '\t', '\n', '\r'
* вычисления с числами с плавающей точкой проходят иногда по-разному (на разных процессорах), с округлением промежуточных результатов, можно использовать strictfp перед методом или классом чтобы операции над числами с плавающей точкой были воспроизводимы, гарантирует точное вычисление и исключает переполнение (используется крайне редко)
* Если недостаточно точности примитивов, то используем BigInteger, BigDecimal (BigInteger a = BigInteger.valueOf(100);
* String как правило хранится в пуле стрингов, так же обстоит и с обертками (но есть интересная особенность, Integer равны друг другу только если значение лежит в пределах до 127)
* Для каждого примитивного типа есть класс обертка (Integer.valueOf(primitive) или new Integer(primitive)), часто используется в коллекциях (можно и не писать new, а просто число, java автоупакует его), числовые обертки наследуются от Number
* Обертки можно конвертить в строку и обратно (toString и parse)
* Для оберток есть перегрузка оператора = и +
* String и классы обертки имеют свойства синглетона (т.к. valueOf это метод синглетона)
* В java реализована динамическая диспетчеризация методов, т.е. нужный метод будет выбираться во время выполнения
* Иногда final даже ускоряет работу программы, т.к. происходит не позднее, а раннее связывание
* Переопределенные методы лучше предварять @Override чтобы компилятор точно проверял корректность переопределения
* Переопределенный метод вызывается исходя из типа объекта на который ссылается ссылка, а не типа ссылочной переменной
* Переопределение полей класса не существует, поэтому лучше не называть одинаковым именем, т.к. будет существовать копия для каждого из объектов
* Модификатор static означает статический класс, методы можно вызывать без создания экземпляра класса (часто используют если свойства или методы присущи всему классу, а не экземпляру)
* в статических методах мы можем вызывать только другие статические методы и использовать только статические переменные (this не передается в методы)
* Статический импорт import static .. (мы можем использовать статические методы без полного указания класса, не Math.sqrt, а sqrt, например)
* К статическим методам не применим принцип полиморфизма, т.к. связывание происходит на этапе компиляции
* Модификаторы доступа: public (видны другим классам из всех пакетов), private (доступен только из кода в том же классе), protected (доступен из текущего класса, из текущего пакета или производного класса даже из другого пакета), по умолчанию (package-private) (видны всем классам в текущем пакете)
* Если в метод передать объект другого того же класса, то он будет иметь доступ к private полям
* Protected вызывает всегда споры, в общем не рекомендуется так обозначать поля, а вот методы можно

**Конструкции языка**

* возможно сочетание арифметических операций с присваиванием x += 4
* операции с присваиванием могут быть объединены в цепочку: x = y = z = 100; (т.к. присваивание идет справа)
* инкремент/декремент бывает префиксный (сначала изменяется значение переменной) и постфиксный (сначала операция, потом значение)
* Существует понятие блок операций – обозначается как {} и может быть в любом месте, так же блок определяет область видимости переменных. Нельзя внутри блока определить переменную совпадающую с именем во внешнем блоке {int n = 1; { int n = 2; //ошибка}}
* условные конструкции if..else if.. else, скобки не обязательны, else объединяется с ближайшим if
* switch(num){ case 1: .. break; default: .. }, с JDK 7 в выражении кроме примитивных типов можно использовать строки, если используем перечисление, то не нужно вводить имя перечисления (switch (sz){ case SMALL: .. break;} )
* тернарная операция : [условие] ? [второй операнд] : [третий операнд]
* интересная работа оператора XOR (^) он принимает значение истина только если один из аргументов истина (с помощью него можно обменять числа местами)
* одинарные |, & проверяют всегда обе части выражения независимо от результата
* switch в общем работает быстрее чем if, т.к. компилятор строит таблицу переходов
* циклы :  
  for (int i = 1, j = 10; i < j; i++, j--){..}  
  for (int i : array){..} (работает с любым классом реализующим интерфейс Iterable)  
  do{..} while (j > 0);  
  while (j > 0){..}
* управление циклами: break; (можно использовать с меткой, метка обозначается как name: и управление передается сразу за метку) и continue;
* при компиляции циклы for и while компилируются в одно и тоже
* циклы могут не содержать тела, если вычислений хватает в управляющей конструкции
* рекурсии работают медленнее циклов, может возникнуть переполнение стека, но некоторые алгоритмы (например быстрая сортировка) сложно реализовать без рекурсии
* для обозначения констант используем final и пишем как правило большими буквами, часто есть потребность в константах класса, их обозначаем public static final (к локальным переменным можно применять только final)
* значение для final переменной задается либо при инициализации, либо в конструкторе (иначе ошибка)
* рекурсивные функции – вызывающие сами себя, обязательно должны иметь базовый вариант чтобы остановиться
* проверить является ли объект экземпляром класса – instanceof (x instanceof Manager), возвращает true/false и если x == null, то исключения не будет, а просто вернет false. Вообще лучше меньше использовать такой подход (оператор ссылается не на тип ссылки, а на свойства объекта на который она ссылается)

**Стандартные классы**

* Массивы (хранят только одинаковые элементы): обозначаются как int[] nums = new int[4]; либо int[] nums2 = new int[] { 1, 2, 3, 5 }; либо int[] nums3 = { 1, 2, 3, 5 }; а двумерный массив int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } }; (при создании массива все его элементы инициализируются 0 или null или false)
* Length у массива никогда не отражает реальную длину массива, а лишь разрешенное количество элементов
* Массивы созданы фиксированной длинны чтобы быстро по ним итерироваться, т.к. хранятся на стеке
* Инициализация массива с помощью {} возможна только при определении
* Атрибуты командной строки (java ClassName g h), передаются в метод main тоже в виде массива и использовать их можно как args[0]
* Для работы с массивами есть специальный класс java.util.Arrays : Копирование массивов Arrays.copyOf(numbers, numbers.length) или очень быстро работает System.arraycopy(), Arrays.sort(numbers); - сортирует массивы (используется усовершенствованный вариант алгоритма быстрой сортировки), Arrays.equals(a,b) попарно сравнивает элементы массивов
* Для математических вычислений используем класс Math (почти все методы статические) :   
  - Math.sqrt() – вычисление корня  
  - Math.pow() – возведение в степень  
  - Math.floorMod() – остаток от деления (положительный т.к. у % ошибно они бывают отрицательные)
* Для работы с большими числами используем BigInteger, BigDecimal:  
  - valueOf() – получение значений  
  - add() – сумма  
  - multiply() – умножение  
  - divide() – деление
* Дата определяется классом Date (считается устаревшим) или LocalDate (LocalDate.now())

**Перечисления**

* объявляются как public enum Name {}
* каждое из значений enum это экземпляр класса Enum, поэтому к ним можно применить методы или конструкторы ( enum Size { SMALL("S"); private String a; private Size(String \_a){ a = \_a}; String getSize(){return a;}}, соответственно потом можно вызывать метод getSize
* для какого-то значения enum можно переопределять методы, записываются как: SMALL("S") {String getSize(){return "XL";}};

**Сборщик мусора**

* Реализован механизм автоматической сборки мусора, когда он запускается у класса вызывается метод finalize(), но мы не можем гарантировать его вызов (на самом деле есть способ отправив Runtime.addShutdownHook() )

**Исключения**

* Типы ошибок: 1) ошибки ввода 2) сбой оборудования (недоступная веб страница или закончилась бумага в принтере) 3) физические ограничения (закончилась оперативная память или место на диске) 4) ошибки программирования
* базовым классом исключений является Throwable, от него наследуются Error (описывает внутренние ошибки в среде исполнения) и Exception. От Exception наследуется RuntimeException и остальные
* Exception – являются проверяемыми (checked), RuntimeException и Error – непроверяемые (unchecked - исключения в следствии ошибки разработчика, их можно не указывать в throws)
* если метод вызывает, но не обрабатывает исключения, то к определению дописываем throws Exceptname
* по сути trows дописывается если вызывается метод генерирующий проверяемое исключение и мы его не обрабатываем или мы явно вызываем исключение с помощью throw
* Есть особенность – метод из подкласса не может генерировать более общие исключения чем метод из суперкласса, если в суперклассе их вообще нет, то и подкласс генерить не может. Если в объявлении метода указан класс исключений, то он может генерить и все подклассы данного исключения
* Можно создавать и вызывать свои исключения, нужно создать класс наследуемый от Exception (или любого другого подкласса). Так же по соглашению должно быть два конструктора: по умолчанию и с сообщением (указываем super(message))
* для того чтобы вызвать исключение пишем throw new Exception("..")
* перехват исключения, общий вид: try {..} catch(Exception e){} finally {}, если нужно обработать несколько исключений, то пишем еще один catch
* можно обрабатывать сразу несколько типов исключений catch(FirstException | SecondException e), только если они не являются подклассами друг друга (иначе ошибка)
* e – конечная (final)
* если в блоке catch вызывается исключение, то мы получаем цепочку исключений
* Хорошей практикой перехвата исключения считается заключение его в оболочку и прокинуть дальше:  
  catch (SQLException e){ throw new ServletException(".." + e.getMessage()); }  
  или  
  catch (SQLException e){ Throwable se = new ServletException("…");  
  se.initCause(e); throw se; } и далее можно извлечь Throwable e = se.getCause();
* Цепочки нужны чтобы проинформировать о причине исключения (например, деление на ноль – причина ошибка ввода), помимо initCause (используется в старых классах) можно просто в конструкторе указать (message, causeExeption)
* Блок finally можно использовать и без catch, рекомендуется даже так try { try {..} finally{} } catch {}
* Блок finally будет выполнен даже если в try содержится return, и если в finally тоже есть return, то первоначальный return будет замаскирован. Более того если в final стоит return, то будет проигнорировано исключение которое вызывается как throw. По сути finally выполняется в любом случае при передаче управления из метода
* может быть ситуация когда в finally при закрытии ресурсов тоже выскочит ошибка – для этого используем try с ресурсами try (Resource r = new Resource() ) {} catch{}  
  если ресурсов несколько – перечисляем через ";"
* если в try с ресурсами возникнет исключение – оно добавится как подавленное (suppressed) и если в своем классе реализовать интерфейс AutoClosable – то можно и его использовать как ресурс (так же есть интерфейс Closable, это для ошибок типа IOException)
* получить подавленные сообщения можно через getSuppressed()
* все исключения наследуют getMessage() и getStackTrace() методы, getStackTrace возвращает массив StackTraceElement – необходим для анализа трассировки (содержит информацию о имени файла, номере строки и т.д.)
* метод Thread. getAHStackTraces () позволяет получить трейс со всех потоков, возвращает Map<Thread, StackTraceElement []>
* так же для поимки ошибок используется механизм assert. (assert (a>0);) Если условие не удоблетворяет, то программа упадет по AssertionError

**Абстрактные классы и интерфейсы**

* Классы в которых методы определены частично называется абстрактным, обозначается как abstract (если есть abstract методы класс обязан быть abstract)
* Нельзя объявлять abstract конструкторы и статические методы, так же нельзя создать экземпляр
* Класс не может быть одновременно abstract и final
* Класс можно определить абстрактным даже если у него нет абстрактных методов (просто нельзя будет создать его экземпляр)
* Абстрактный класс может расширять обычный класс, т.е. никаких ограничений
* Интерфейс определяется как interface, а применяется как implements
* Называются часто с буквы "I" (IPayable)
* Интерфейсы используются для описания поведения, это 100% абстрактный класс (можно даже поставить abstract перед определением) и помогают решить проблему запрета множественного наследования (определяют некий функционал, но не имеют конкретной реализации)
* По сути интерфейс – это спецификация абстрактного типа данных, туда помещаются все методы открытые для использования в классе
* все методы интерфейса публичные и абстрактные, причем когда мы реализуем метод интерфейса он тоже должен быть объявлен как public
* Часто в интерфейсе описывают константы, доступ у них будет public static final
* Класс реализующий интерфейс должен реализовать все методы, либо быть абстрактным
* мы не можем напрямую создавать объекты интерфейсов, но зато если класс реализует интерфейс мы может привести к его типу (Printable p = new Journal("Хакер")), но если объект типа интерфейс хранит ссылку на класс и нам надо обратиться к методам класса неопределенным в интерфейсе, то необходимо делать возвышающие преобразование ( ((Journal)p).getName(); )
* ранее класс должен был реализовать все методы, в JDK 8 появилась возможность задать методы по умолчанию, т.е. интерфейсы могут иметь реализацию по умолчанию (модификатор default), так же добавляются и реализацию имеют static методы интерфейсов (они вызываются как и у класса)
* default методы можно переопределять, static методы не реализуются ни подчиненными интерфейсами (в отличии от классов в подчиненном интерфейсе нельзя вызвать static метод), ни классами
* default используются например при обработке событий от мыши мы переопределяем только нужные нам, остальные дефолтные
* в методах в интерфейсе доступным методы класса Object (такое ощущение что интерфейс тоже наследуется от Object, но на самом деле у него неявно дублируются все public методы из Object)
* в java9 можно создавать private методы, они используются только в конкретном интерфейсе
* интерфейсы могут расширять друг друга с помощью extends, причем можно расширять сразу несколько интерфейсов
* проверить что класс реализует интерфейс можно с помощью instanceof
* интерфейс так же может быть внутренним (вложенным), тогда он может быть public, private, protected (обращение к нему как и к вложенному классу)
* Часто интерфейсы используются для создания обратного вызова (т.е. мы создаем действие которое вызывается при других действиях, например нажатие на кнопку)
* но основное отличие от абстрактного класса в том что интерфейсы не содержат поля экземпляра
* разрешение конфликтов: 1) если интерфейс наследуется и суперкласс содержит метод default, то в потомках они игнорируются 2) если в суперклассе интерфейса есть метод с таким же именем, он перекрывается методом из наследуемого (обратиться к родителю можно через InterfaceName.super.MethodName) 3) если реализовать 2 интерфейса с методом одного имени, то только переопределить метод, иначе конфликт и ошибка
* Распространенные интерфейсы: 1) для сортировки по умолчанию реализуем Comparable (compareTo) 2) для сортировки реализуем Comparator (compare) 3) для клонирования объектов реализуем Cloneable (на самом деле является маркерным интерфейсом) (вызывается Object.clone(), но его обязательно нужно переопределить как public иначе нельзя вызывать напрямую). У массивов метод clone открыт, так что их можно клонировать
* Интерфейсы могут быть вложенными (с модификаторами public, private, protected)

**Функциональные интерфейсы и лямбды**

* функциональные интерфейсы – интерфейс с одним абстрактным методом (т.е. у него отсутствует реализация по умолчанию), можно пометить как @FunctionalInterface. Среди методов интерфейса могут быть с реализацией по умолчанию, статические и дублирующие методы Object, но именно абстрактный должен быть один
* Лямбда – набор инструкций, которые можно выделить в отдельную переменную
* Назовем их функции и в отличии от методов они абсолютно не привязаны к объекту класса
* Лямбда оператор -> разделяет выражение на две части левую – список параметров, правую – тело
* Лямбда не выполняется сама по себе, а образует реализацию метода определенного в функциональном интерфейсе
* Записыватся как: () –> {}, слева указываем параметры, если типы легко вычислимы, то определение можно опустить, если параметр один, то скобки можно опустить, справа выражение, если оно укладывается в один оператор, то скобки можно опустить (пример: Operation op = (int y)->{ int x=30; return x+y;}, () –> new BigDecimal(),(String a, String b) -> {return a.compare(b)}, (a , b) -> {return a.compareTo(b);}, самая простая лямбда () –> 1)
* При объявлении лямбды автоматически создается экземпляр класса реализующий функциональный интерфейс
* Лямбды могут и не возвращать значений, тогда их называют терминальными (s-> System.out.print(s))
* Использование переменных в лямбда: статическая переменная уровня класса может использоваться и ее значение может изменяться, а вот при использовании переменной уровня метода лямбда выражения при конструировании захватывают необходимые переменные (это нужно чтобы лямбда могла и в дальнейшем выполняться, т.е. после выхода из блока где определены захваченные переменные), поэтому в них можно использовать любые переменные класса (значения которых не изменяются), изменить захваченную переменную так же нельзя
* Поэтому при использовании переменных в лямбда лучше объявить их как final (хотя это не обязательно), т.к. по сути того и требуется
* Блок кода вместе с захваченными переменными называется замыкание
* Функциональные интерфейсы могут быть обобщенными, но лямбды не могут. Поэтому перед использованием мы уточняем тип (interface Operationable<T>{ T calculate(T x, T y);} Operationable<Integer> operation1 = (x, y)-> x + y;)
* Лямбду можно предавать в качестве аргумента в метод, для этого параметр объявляем как функциональный интерфейс int sum (int[] numbers, Expression func), либо сразу в метод передаем лямбду sum(nums, (n)-> n > 5);
* Лямбды так же могут возвращаться из метода, таким образом можно возвращать разные actions из одного метода
* Лямбду нельзя присвоить типу Object
* Создание ссылки на метод: ClassName::methodName, это тоже самое что x-> methodName(x)
* Позволяет ссылаться на метод не вызывая его, еще важно что метод может и называться не так, просто должны совпадать сигнатуры
* Существует три варианта ссылки на метод: Объект::МетодЭкземпляра, Класс::СтатическийМетод (в этих двух случаях это аналогично лямбде снабжающему метод параметрами x-> methodName(x)), Класс::МетодЭкземпляра (в этом случае параметр является целевым для метода x-> x.methodName()), так же допустимы записи this::МетодЭкземпляра и super::МетодЭкземпляра этот this и super относятся к классу из которого вызывается (this::equals равнозначна x -> this.equals(x) )
* Так же есть возможность обращаться к обобщенным методам Класс::<Тип>МетодЭкземпляра
* Аналогично можно сделать ссылку на конструктор ( Person::new ), так же например int[]::new равнозначно x -> new int[x], таким образом можно создавать экземпляры через лямбду. Так же в Java нельзя построить массив обобщенного типа Т, а с помощью лямбды можно
* Все встроенные функциональные интерфейсы перечислены в java.util.function, все они делятся на Consumer (потребители, принимают значение и ничего не возвращают) метод accept (так же andThen), Supplier (поставщики, не принимают параметры, но возвращают) метод get, Predicate (принимает тип и отдает Boolean) метод test – проверяет на совпадение какого либо условия (так же or, and,negate, isEqual), Function <T,R> (функция перехода от T к R) метод apply (так же compose, andThen, identity), BinaryOperator (принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T) метод apply, UnaryOperator (принимает в качестве параметра объект типа T, выполняет над ними операции) метод apply, Runnable (выполняет действие без возвращаемого значения) метод run
* так же для всех этих функциональных интерфейсов есть аналоги с двумя входящими и приставкой Bi (BiConsumer)
* есть в интерфейсах и доп. методы, например Pridicate.isEqual(a).or(Pridicate.isEqual(b)) равносильно x -> a.equals(x) || b.equals(x), у компараторов есть методы thenComparing
* Лямбды могут генерировать исключения, но оно должно быть совместимо с определением из функционального интерфейса
* NOTE: лямбда могут помочь избавиться от наследования (например в случаях если экземпляры отличаются лишь реализацией методов)
* NOTE: в линейном программировании мы не можем понять меняет ли метод состояние объекта, так же функциональное программирование борется с сайд-эффектами и скрытыми параметрами