Какова разница между абстрактным классом и интерфейсом?

1. **АК** это класс, снабжённый модификатором abstract. Нельзя создать экземпляр АК, так же как нельзя объявить АК final. АК может содержать абстрактные методы (без реализации, с модификатором abstract).
2. И это особая структура, описывающая тип данных в виде его интерфейса (списка сигнатур методов для работы с этим типом)
3. **АК** может содержать всё, что может содержать обычный класс, а также объявления абстрактных методов. Интерфейс же может содержать только объявления методов, *static nested classes & interfaces и константы*.
4. АК может, как и интерфейс, содержать только объявления методов (абстрактные методы), но, в отличие от И, в АК эти методы могут быть private/default/protected, а в И только public
5. использовать несколько “полностью абстрактных” классов (содержащие только абстрактные методы) в одном классе-наследнике нельзя, а несколько интерфейсов можно (множественное наследование)

Как «насильно» вызвать сборку мусора?

1. Сборку мусора (освобождение памяти, занимаемой объектами
2. GC насильно вызвать нельзя, можно только “попросить” JVM произвести сборку мусора по возможности.
3. Это можно сделать как программно, через обращение к System.gc(), так и через “внешние” инструменты, например, с помощью jvisualvm.
4. Полная сборка мусора, выполняемая при этом, может занять довольно существенное время, поэтому выполнять её в приложениях, чувствительных ко времени выполнения отдельных операций, применять её не рекомендуется, особенно учитывая тот факт, что момент, когда JVM реально выполнит её, неизвестен.

Когда требуется явное приведение классов?

1. Явное приведение классов (explicit casting) служит для указания, каким типом должен обладать объект, с тем, чтобы можно было использовать интерфейс этого типа (методы или поля), применительно к этому объекту, или присвоить ссылку на него переменной соответствующего типа Например:
   String s = (String) object;
   ((String) object).length()
2. type cast применяется только к типам, связанным иерархией наследования. Нельзя сделать приведение (A) b, если A не является наследником B.
3. это называется нисходящим преобразованием (от типа-родителя к типу-потомку)
4. восходящее преобразование типов (типа-потомка к типу-родителю) делается неявно (implicit casting), например: Object o = new String(“one”);
5. при восходящем преобразовании типов возможна потеря доступа к части интерфейса приводимого типа.
6. при явном приведении возможна ситуация, когда объект на самом деле не обладает нужным типом, в этом случае может возникнуть ClassCastException: Object o = new Integer(100); String s = (String) o; <- ошибка
7. предотвратить это можно, предварительно проверив тип с помощью instanceof

Чем конструкторы отличаются от других методов?

1. Конструкторы служат для создания экземпляров класса, у методов более широкое применение
2. Имя конструктора должно совпадать с именем класса, при этом в классе могут быть и методы, совпадающие с именем класса
3. Конструктор не имеет типа возвращаемого значения
4. в каждом конструкторе, если нет иного, неявно вызывается конструктор без параметров супер-класса (если он есть) в первой же строке (перед всеми остальными)
5. в конструкторах можно использовать явное обращение к другим конструкторам с помощью super(...) и this(…), при этом они должны быть в первой строке. А в методах экземпляра можно использовать super. в произвольном месте, а можно и не использовать.
6. главное тут то, что любой конструктор обязательно обращается либо к другому конструктору этого же класса, либо к какому-либо конструктору супер-класса, явно или неявно.
7. конструкторы не могут иметь модификаторов static, abstract, final. можно только public/protected/default/private
8. конструкторы нельзя переопределить (override).
9. если в классе нет ни одного конструктора, при компиляции автоматически генерируется public конструктор без параметров (конструктор по умолчанию)

Можно ли вызывать конструкторы один из другого, если их в классе несколько?

1. (см. предыдущий ответ в целом про конструкторы)
2. да, можно, с помощью this: A() {this(10); … }
3. обращение к this должно быть первой строкой в конструкторе
4. при этом нельзя делать рекурсию (цикл) из подобных вызовов (первый вызывает второй, а второй первый)

В чем разница между JDK и JRE?

1. JDK: Java Development Kit - набор разработчика. JRE: Java Runtime Environment - среда выполнения.
2. JRE содержит JVM (Java Virtual Machine) и набор стандартных библиотек. С её помощью пользователь может запускать программы, написанные на Java
3. JDK состоит из JRE, исходных кодов стандартных библиотечных классов, а также набора вспомогательных программ для разработки, отладки, публикации и мониторинга приложений на Java. Например:
   1. javac - компилятор
   2. jmap - для анализа памяти
   3. jstack - для анализа нитей (потоков) выполнения
   4. jvisualvm - для мониторинга

Имеет ли значение в каком порядке перехватывать исключения FileNotFoundException и IOExceptipon?

1. рассказать про Exceptions в целом
2. рассказать про try/catch
3. multiple catch
   1. standard
   2. compact form (Java 7)
4. блоки catch применяются последовательно
5. если объект exception обладает соответствующим типом, то он будет перехвачен
6. соответственно, если в первом блоке тип exception будет более общий (супер класс), то второй блок никогда не сработает.
7. в данном вопросе FNFE является подклассом IOE, поэтому блок, перехватывающий FNFE, должен быть перед блоком IOE
8. это имеет смысл только если обработка этих исключений разная. если код одинаков, можно обойтись просто перехватом IOE.
9. если exceptions не связаны как родитель/потомок, то порядок неважен

Могут ли внутренние классы, описанные внутри метода, иметь доступ к локальным переменным этого метода?

1. рассказать кратко про внутренние классы
2. в методе может быть описан локальный или анонимный класс
3. созданные в методе экземпляры этих классов могут быть “отданы” за пределы метода (через ссылки на них)
4. но время жизни локальной переменной - только в блоке метода
5. поэтому внутри классов создаются и используются копии значений этих переменных
6. поскольку после определения класса программист мог бы менять значение локальной переменной, что привело бы к семантической ошибке (он мог бы считать, что изменение переменной в методе отразится на значении переменной в локальном классе), в Java ввели требование делать такие переменные final, явно запрещая изменение их значений
7. то же самое относится и к параметрам метода, в котором определён внутренний класс: они тоже должны быть final, если их используют внутри локального/анонимного класса в методе.

Как подкласс может обратиться к методу или конструктору из суперкласса?

1. привести пример A extends B, демонстрируя знание наследования
2. уточнить, что речь идёт о доступе из метода экземпляра
3. super() для конструкторов (только в первой строке, всегда есть неявно)
4. super. для методов: произвольное место, если нет, то не вызывается

В чем разница между очередью и стеком?

1. Очередь (Queue) и стек (Stack) - структуры данных, служащие для хранения и выдачи данных (объектов).
2. Очередь, как минимум, обладает двумя операциями: add (добавить данные) и remove (извлечь данные).
3. Операция add добавляет данные в конец очереди (часто называется tail, хвост), а операция remove удаляет данные из начала очереди (head). Таким образом, каждый следующий элемент добавляется “после” предыдущего, а самый “старый” элемент (добавленный раньше всех), извлекается из очереди первым (при этом новым первым становится элемент, следовавший за ним).
4. Этот принцип называется FIFO: First In, First Out (первый пришёл, первый ушёл).
5. Стек (другое часто используемое название - “магазин”), как минимум, обладает двумя операциями: push (втолкнуть) и pop (вытолкнуть).
6. Стек мысленно можно представлять себе как стопку из хранимых элементов. Операция push помещает новый элемент наверху стопки, поверх остальных. Операция pop извлекает верхний элемент из стопки, при этом “верхним” элементом становится элемент, находившийся “под" ним.
7. Этот принцип называется LIFO: Last In First Out (последним пришёл, первым ушёл).
8. Главное отличие стека от очереди состоит как раз в разных принципах работы: Queue - FIFO, Stack - LIFO.
9. Помимо обязательных операций (add/remove, push/pop), как правило, есть и дополнительные:
   1. в очереди: head/element/peek/first - посмотреть, какой элемент первый в очереди, но не удалять; tail/last - посмотреть, какой последний элемент в очереди
   2. в стеке: tos/top/peek - посмотреть, какой элемент находится на верхушке стека, но не выталкивать его.
   3. isEmpty, size - проверка на пустоту и получение размера (количества хранящихся элементов)
   4. поиск элемента и просмотр всех элементов

Что вам приходит в голову, когда вы слышите о новом поколении (young generation) в Java?

1. Это термин, связанный с динамической памятью в Java (HEAP, “куча”) и со сборкой мусора (GC, Garbage Collection).
2. Все объекты, создающиеся JVM (виртуальной Java-машиной), размещаются в динамической памяти (куче).
3. пока на объект ссылается какой-то другой объект (через переменную, хранящую ссылку), он нужен в работе. как только на объект уже никто не ссылается, доступ к нему потерян и он становится “мусором”.
4. сборщик мусора периодически проверяет динамическую память и “очищает” её от “мусора”, освобождая занятую прежде память, делая её пригодной для размещения новых объектов
5. вероятность того, что недавно созданный объект скоро станет “мусором” весьма велика. поэтому, для экономии времени при работе сборщика мусора, он просматривает в основном только “свежие”, “молодые” объекты. это и есть young generation.
6. сборщик мусора ведёт учёт таких объектов. внутри young generation он делятся на
   1. Eden Space (только что созданные объекты, ни разу не проверенные при сборке мусора)
   2. Survivor Space (объекты, пережившие несколько сборок мусора (от 1 до N, зависит от типа сборщика и настроек)
7. если объект пережил много сборок мусора, он становится “старожилом” и перемещается в Tenured Generation (или Old)
8. Tenured(Old) Generation подвергается сборке мусора значительно реже, чем Young Generation.
9. Помимо Young и Tenured есть ещё и PermGen (Permanent Generation) - место, в котором JVM хранит мета-информацию о классах (байт-код методов) и другую информацию.

Есть два класса: A и B. Класс B должен информировать класс A когда случается некое важное событие. Какой design-pattern вы должны реализовать?

1. Этот шаблон проектирования (desing pattern) называется Observer (Наблюдатель). Часто его также называют Listener (Слушатель) и Publisher/Subsciber(Издатель/Подписчик)
2. В данном примере класс B это наблюдаемый объект, а класс A - наблюдатель (слушатель)
3. Как правило, наблюдаемый объект позволяет наблюдать за ним сразу нескольким наблюдателям(слушателям)
4. для этого в нем есть методы addOberver/removeObserver (add/removeListener).
5. наблюдатели, как правило, регистрируются у объекта в виде интерфейсов (например, MouseListener)
6. при наступлении определённого события объект перебирает список из наблюдателей и посылает каждому из них сообщение о событии
7. сообщение может быть “пустым”, а может и предоставлять наблюдателю дополнительную информацию
8. это делается либо передачей ссылки на сам объект, с тем чтобы наблюдатель мог сам извлечь из объекта необходимые ему данные, либо объект сам формирует набор данных, интересующих наблюдателя и передает его прямо в сообщении в виде объекта, описывающего произошедшее событие.

Какой модификатор доступа надо указать в классе, чтобы доступ к нему имели только классы из того же пакета?

1. Класс может иметь несколько различных видов доступа к нему, описываемые модификаторами public/protected/private
2. если модификатор доступа отсутствует, то это уровень доступа “по умолчанию”, default
3. именно он гарантирует доступ к классу из любого класса из этого же пакета
4. однако, это не относится анонимным и локальным классам, у которых не может быть никаких модификаторов и которые недоступны вне блока, в котором они описаны
5. модификатор protected, дающий доступ к классу из других пакетов, также гарантирует доступ к классу для всех классов из его же пакета
6. внешние (outer) классы могут иметь только доступ public или default (пакетный). вложенные (nested) классы (static, inner, local, anonymous) могут иметь доступ public/protected/default/private

Чем отличается статический внутренний класс от просто внутреннего класса?

1. Внутренние (или часто используется термин вложенные, nested) классы бывают 4-х видов:
   1. static - статический внутренний класс, описывается внутри класса
   2. inner - просто внутренний класс, описывается внутри класса
   3. local - локальный внутренний класс, описывается в блоке (например, в методе или инициализаторе класса/экземпляра)
   4. anonymous - анонимный класс, экземпляр которого создаётся прямо в месте описания на основе другого класса-родителя или интерфейса, описывается в блоке.
2. static класс описывается внутри внешнего класса, обязательно имеет модификатор static и может иметь любой из модификаторов public/protected/private/abstract/final
3. static класс можно инстанцировать (создать экземпляр), просто используя его конструктор в сочетании с именем внешнего класса, в котором он описан. например: new A.B()
   указание внешнего класса необязательно (если есть import или используем в классе-владельце)
4. просто внутренний класс (далее inner) описывается также внутри внешнего класса и у него отсутствует модификатор static, остальные модификаторы любые
5. inner класс можно инстанцировать только используя конкретный экземпляр внешнего класса, в котором он описан, это делается с помощью специального вида оператора new: A a = new A(); a.new B(); (где A внешний класс, а B - inner в нём)
6. inner класс хранит неявную внутреннюю ссылку на экземпляр, с помощью которого он был создан
7. за счёт этого экземпляр inner класса может обращаться напрямую к членам экземпляра внешнего класса (полям и методам)

можно ли обратиться к не-статической переменной из статического метода?

1. Предположим, речь идёт не о переменных вообще (параметры метода, локальные - это всё тоже переменные), а о поле: переменной-члене класса, у которой отсутствует модификатор static:
   ```
   class A {
     int i;
   }
   ```
2. это поле существует только в рамках объекта - экземпляра класса
3. поэтому обращение к этому полю возможно только через экземпляр класса
4. в методах экземпляра (не статических) использование экземпляра класса обычно делается неявно, опуская обращение к объекту через служебное ключевое слово this. например:
   ```
   class A {
      int i;
      void m() {
         this.i = 10;   // явное указание экземпляра
         System.out.println(i);   // неявное указание экземпляра
      }
   }
   ```
5. в методах класса (static) использование обращения к this невозможно, поскольку в контексте выполнения статического метода нет сведений о конкретном экземпляре класса, к которому его применять
6. соответственно, доступ к нестатическому полю экземпляра класса в статическом методе невозможен:
   ```
   class A {
      int i;
      static void m() {
         this.i = 10;   // ERROR: явное указание экземпляра НЕВОЗМОЖНО в static
         System.out.println(i);   // ERROR: неявное указание экземпляра НЕВОЗМОЖНО в static
      }
   }
   ```
7. однако, если есть ссылка на экземпляр класса, доступная в статическом методе этого класса (через параметр метода или статическую переменную), то доступ к полю возможен, даже если оно private:
   ```
   class A {
      private int i;
      static void m(A a) {
         a.i = 10;   // явное указание экземпляра
         System.out.println(a.i);   // явное указание экземпляра
      }
   }
   ```

какие типы данных есть в Java?

1. При объявлении переменной в языке Java необходимо всегда указывать её тип данных, например: int a;
2. типы данных делятся на
   1. примитивные
      1. перечислимые
         1. целочисленные byte, short, int, long
         2. символьный char
      2. логический boolean
      3. вещественные float, double
   2. ссылочные
      1. классы (class)
      2. интерфейсы (interface)
      3. перечисления (enum)
3. для примитивных типов есть классы-обёртки: Byte, Short, Integer, Long, Float, Double, Character, Boolean
4. при смешанном использовании объектов классов-обёрток и примитивных типов в Java работает механизм авто-упаковки (auto-boxing), например: Integer i = 20; // неявно создаётся new Integer(20), авто-упаковка

Чем отличаются переопределение (Override) и перегрузка (Overload)

1. Оба эти термина относятся к методам, описанным в классе
2. Override это переопределение метода экземпляра, унаследованного от родительского класса
3. также этот термин в Java применяется при описании методов интерфейса, реализуемого в конкретном классе
4. при переопределении метода родителя возможно обращение к переопределённым методам-членам класса-родителя с помощью super.
5. Overload это перегрузка, имеется в виду перегрузка имени метода: описание в классе нескольких методов с одним и тем же именем, но с разным набором параметров
6. Overload может применяться как к методам экземпляра (не статическим), так и к методам класса (статическим, с модификатором static)
7. Overload не имеет прямого отношения к наследованию, хотя, конечно, наследники могут объявлять новые методы с именами как в родительском классе, но с другим набором параметров.
8. И тот, и другой приём используются для поддержки полиморфизма

Что такое итератор?

1. Итератор это приём, известный также как шаблон проектирования (design pattern), который широко используется для перебора элементов из какого-либо набора данных
2. Он служит альтернативой перебору элементов массива, который возможен благодаря обращению к конкретному элементу по его номеру (индексу). при переборе элементов массива в цикле, например, переменная цикла используется для перемещения к следующему элементу, один за другим.
3. В некоторых наборах данных нет индексов, позволяющих обращаться к элементам напрямую. Однако, как правило, есть возможность запомнить текущий элемент и получить ссылку на следующий (например, в списке).
4. Соответственно, итератор - это некий объект, работающий с конкретным набором данных и умеющий перебирать все элементы этого набора один за другим
5. Для этого итератор предоставляет клиенту как минимум два метода: hasNext() и next()
   1. boolean hasNext() - возвращает true, если есть следующий необработанный элемент
   2. Object next() - возвращает ссылку на следующий необработанный объект из набора
6. пример:
   ```
   Iterator it = list.iterator();
   while (it.hasNext()) {
     Object item = it.next();
     // обработка объекта
   }
   ```
7. в Java интерфейс Iterator содержит также метод remove для удаления текущего элемента (последнего возвращенного методом next) из набора данных. он необязателен для реализации (обычно оставляем пустым)

Перечислите основные категории исключительных ситуаций

1. исключительные ситуации - это объекты, используемые в JVM для сообщения о возникших проблемах при выполнении программы
2. при возникновении проблемы можно “выбросить” исключительную ситуацию с помощью оператора throw, например:
   throw new RuntimeException(“problem description”);
3. Для перехвата исключений используется конструкция try/catch/(finally)
4. соответственно, для исключительных ситуаций используют экземпляры стандартных библиотечных классов или их потомков
5. базовый класс для всех исключительных ситуаций - Throwable (“выбрасываемый)
6. иерархия:
   - Throwable
     - Error - как правило, это ошибки самой JVM, например, OutOfMemoryError
     - Exception
       - RuntimeException
7. исключения делятся на две категории: проверяемые (checked) и непроверяемые (unchecked)
8. если метод выбрасывает проверяемое исключение, он обязан либо обработать его на месте, либо “передать” для обработки на уровень выше, в место вызова. это обеспечивается декларацией throws в заголовке метода.
9. непроверяемые исключительные ситуации не требуют немедленной обработки или декларации throws
10. к непроверяемым относятся Error и RuntimeException, а также все их потомки.
11. все остальные (Throwable, Exception и их потомки кроме Error и RuntimeException) являются проверяемыми.

Какая разница между throw и throws?

1. рассказать про Exceptions (см предыдущий вопрос)
2. дополнительно:
   1. throws часто используется в блоке catch для повторного выбрасывания исключительной ситуации. это необходимо для “каскадной” обработки проблемы, когда на каждом уровне можно написать код, выполняющий часть общей работы.
   2. при этом либо выбрасывают новый экземпляр, либо тот же самый
   3. при выбрасывании нового экземпляра часто “исходное” исключение “упаковывают” в новое (пердавая ссылку на него в конструктор нового исключения), а доступ к нему при обработке можно получить через метод getCause().
   4. декларация throws может содержать список из исключений
   5. в декларации throws необязательно указывать точный класс исключения, можно указать и более общий (класс-предок)

Зачем нужен блок finally?

1. рассказать про Exceptions (см.выше)
2. довольно часто после выполнения определённой части кода требуется обязательно выполнить завершающие действия. например, закрыть открытый файл или освободить захваченный монитор (Lock). то есть, говоря в целом, освободить ресурс:
   ```
   getResource
   … do something with it
   freeResource
   ```
3. однако, при выполнении кода с использованием ресурса могут возникнуть исключения и естественный ход выполнения может быть нарушен и завершающая часть, выполняющая освобождение ресурса, никогда не будет выполнена.
4. для гарантированного освобождения ресурса используют try/finally:
   ```
   getResource
   try {
     … do something with resource
   } finally {
     freeResource
   }
   ```
5. в Java 7 появилась новая конструкция: try with resource:
   ```
   try (Resource resource = getResource()) {
     … do something with resorce
   }
   ```
6. она гарантирует закрытие ресурса, если он был открыт. главное требование - реализация классом ресурса интерфейса AutoCloseable
7. блок finally выполняется всегда, причём он имеет приоритет. например, этот код вернёт 20:
   ```
   try {
     return 10;
   } finally {
     return 20;
   }
   ```
8. возможно и совместное использование try/catch/finally, но надо всегда помнить о приоритете блока finally: он выполнится всегда, последним.

Что такое finalize?

1. рассказать про сборку мусора (см.выше)
2. метод finalize описан в классе Object. он вызывается сборщиком мусора при удалении объекта из динамической памяти
3. при описании класса можно переопределять этот метод, причём super.finalize() вызывать не обязательно, он ничего не делает.
4. в этом переопределённом методе можно разместить код, освобождающий некие ресурсы, возможно использовавшиеся объектом.
5. “тяжёлый” код, размещённый в finalize может существенно затормозить сборку мусора, что, как правило, крайне нежелательно. поэтому использование finalize нежелательно, а при необходимости надо работать с ним крайне осмотрительно.

Перечислите все виды внутренних классов

1. Внутренние (или часто используется термин вложенные, nested) классы бывают 4-х видов:
   1. static - статический внутренний класс, описывается внутри класса
   2. inner - просто внутренний класс, описывается внутри класса
   3. local - локальный внутренний класс, описывается в блоке (например, в методе или инициализаторе класса/экземпляра)
   4. anonymous - анонимный класс, экземпляр которого создаётся прямо в месте описания на основе другого класса-родителя или интерфейса, описывается в блоке.
2. см. выше подробно про них в разных ответах
3. статические члены класса (методы, поля и инициализаторы) можно указывать только в статических и внешних классах