Вопросы к коллоквиуму 2-го модуля по курсу КПО

Максим Кузнецов

Илья Шиндяпкин

1. **Что такое класс, объект, интерфейс? Основные принципы ООП.**

   Класс - это ссылочный тип, шаблон для создания объекта, описывающий структуру и поведение объекта.

   Объект - это экземпляр класса, содержащий структуру и поведение, которые определены классом.

   Интерфейс - это ссылочный тип, представляющий собой совокупность абстрактных методов. Однако также может содержать константы, статические методы и вложенные типы. Все методы интерфейса являются абстрактными.

   Отличия интерфейса от класса:

   - Вы не можете создать экземпляр интерфейса.
   - В интерфейсе не содержатся конструкторы.
   - Все методы в интерфейсе абстрактные.
   - Интерфейс не может содержать поля экземпляров. Поля, которые могут появиться в интерфейсе, обязаны быть объявлены и статическими, и final.
   - Интерфейс не расширяется классом, он реализуется классом.
   - Интерфейс может расширить множество интерфейсов.

   a. Множественное наследование и как его реализовать?

      В Java не поддерживается множественное наследование классов из-за проблемы ромбовидного наследования, однако существует множественное наследование для интерфейсов. То есть возможна такая конструкция:

      interface InterfaceC extends InterfaceA, InterfaceB

   b. Наследование – это антипаттерн?

      Наследование зачастую используют неверно, заменяя логику, которую следовало бы выделить в интерфейсы и абстрактные классы. Из-за этого иерархии классов могут излишне усложняться. В целом наследование не является антипаттерном, если его правильно использовать, поскольку он предлагает такой мощный инструмент, как полиморфизм, который поможет выстроить сложную и специфическую логику в иерархии классов.

   c. Полиморфизм и его преимущества.

      Полиморфизм - одна из фундаментальных концепций объектно-ориентированного программирования, которая делает код более гибким за счет реализации переопределения методов и перегрузки (перегрузка не всегда считается полиморфизмом). Полиморфизм используется, например, ссылка на родительский

класс используется для ссылки на объект дочернего класса. Также происходит улучшение инкапсуляции, так как класс-родитель не знает как конкретно реализован метод в классе-наследнике.

2. **Преимущества и недостатки ООП.**

   (Преимущества) Позволяет конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кода и появляется возможность создания библиотек классов для различных применений. Могут выстраиваться зависимости между объектами. То есть в целом код становится более модульным, гибким для использования и безопасным. Дает возможность абстрагироваться от деталей реализации. Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

   (Недостатки) От программиста при ООП подходе требуется разбираться в больших библиотеках классов. К тому же, проектирование архитектуры класса и способов взаимодействия с другими классами может оказаться сложнее, чем реализация непосредственно методов, которые и несут всю смысловую нагрузку программы. Также программы написанные с использованием ооп парадигмы зачастую получаются больше по объему кода и памяти на диске, а также слегка медленнее работают, чем просто процедурный код.

3. **Абстрактные классы и интерфейсы.**

   Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал. Интерфейс – это ссылочный тип в Java. Он схож с классом. Это совокупность абстрактных методов. Класс реализует интерфейс, таким образом наследуя абстрактные методы интерфейса. Вместе с абстрактными методами интерфейс в Java может содержать константы, обычные методы, статические методы и вложенные типы. Тела методов существуют только для обычных методов и статических методов.

   a. Вложенные и анонимные классы: когда они используются?

      Вложенные классы подробно расписаны в вопросе 25.

      Анонимные классы - это классы, что не имеют имени и их создание происходит в момент инициализации объекта.

      За счет вложенных классов можно описать дополнительный объект, что принадлежит к классу. Анонимные классы позволяют описать новый функционал для создаваемого объекта.

   b. Когда необходимо использовать интерфейс и абстрактный класс?

      Абстрактный класс используется когда нам нужна какая-то реализация по умолчанию. Интерфейс используется когда классу нужно указать конкретное поведение. Часто интерфейс и абстрактный класс комбинируют, т.е. имплементируют интерфейс в абстрактном классе, чтоб указать поведение и реализацию по умолчанию.

c. Приведите примеры абстрактных классов и интерфейсов

```java
package pro.java.animal;

abstract class Animal { // абстрактный класс

    private String type;

    abstract void getSound(); // абстрактный метод

    Animal(String aType) {
        type = aType;
    }

    String getType() {
        return type;
    }

}
```

```java
interface Printable{

    void print();
}
class Book implements Printable{

    String name;
    String author;

    Book(String name, String author){

        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
```

d. Может ли абстрактный класс не иметь абстрактных методов?

Может.

e. Почему в некоторых интерфейсах вообще нет методов и полей и зачем нужны такие интерфейсы?

Такие интерфейсы являются некими маркерами, обозначающими, что класс относится к определенной группе классов. Например интерфейс Clonable указывает на то, что класс поддерживает механизм клонирования.

f. Использование иерархий интерфейсов

Можно наследовать один интерфейс от другого, таким образом создавая иерархии интерфейсов. На таком принципе реализована Java Collections Framework. В нем главными интерфейсами являются Map и Collection. И так, например, от Collection далее наследуются Set, List, Queue, образуя тем самым иерархию интерфейсов.

4. Лямбда-выражения в *Java*

a. Синтаксис

(параметры) -> (тело)

b. Назначение

Используется как упрощённая запись анонимного класса.

Позволяет реализовать callback функции. Callback или функция обратного вызова в программировании — передача исполняемого кода в качестве одного из параметров другого кода. Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при ее вызове.

c. Примеры

(int x, int y)->x+y;

()-> 30 + 20;

(int x, int y)->{x += y; return x;};

5. **Модификаторы доступа в *Java***
   a. Расположите модификаторы доступа от большего к меньшему

      public: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов

      protected: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах

      private: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.

   b. Сравните с модификаторами в *C#* и *Python*

      В Python так же, как в Java.

      C#:

      private: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.

      private protected: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.

      protected: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.

      internal: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.

      protected internal: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.

      public: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

   c. Сокрытие данных и инкапсуляция

      Инкапсуляция (encapsulation) — это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств.

      Благодаря сокрытию реализации за внешним интерфейсом класса можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы.

d. Почему важно ставить модификаторы доступа?

Рекомендуется как можно больше ограничивать доступ к данным, чтобы защитить их от нежелательного доступа извне (как для получения значения, так и для его изменения). Использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное сокрытие данных внутри некоторой области видимости называется инкапсуляцией.

Явное написания модификатора доступа в коде необходимо для улучшения читаемости кода, поскольку не будут использованы модификаторы доступа по умолчанию.

e. Ключевое слово *final* – что это, в каких конструкциях используется

Для класса это означает, что класс не сможет иметь подклассов, т.е. запрещено наследование.

Это полезно при создании immutable (неизменяемых) объектов, например, класс String объявлен, как final.

Следует также отметить, что к абстрактным классам (с ключевым словом abstract), нельзя применить модификатор final, т.к. это взаимоисключающие понятия.

Для метода final означает, что он не может быть переопределен в подклассах. Это полезно, когда мы хотим, чтобы исходную реализацию нельзя было переопределить.

Для ссылочных переменных это означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. Это важно! Ссылку изменить нельзя, но состояние объекта изменять можно.

f. Проиллюстрируйте на практических примерах использование каждой конструкции

```java
public final class String{
}

class SubString extends String{ //Ошибка компиляции
}
```

```java
final int[] array = {1,2,3,4,5};
array[0] = 9;   //ок, т.к. изменяем содержимое массива – {9,2,3,4,5}
array = new int[5]; //ошибка компиляции
```

```java
public class SuperClass{
    public final void printReport(){
        System.out.println("Report");
    }
}

class SubClass extends SuperClass{
    public void printReport(){  //Ошибка компиляции
        System.out.println("MyReport");
    }
}
```

6. Класс *Object* в Java

Является суперклассом для всех классов.

a. Расскажите о методах класса

- getClass() возвращает: объект Class, представляющий класс времени исполнения (runtime class) этого объекта.
- hashCode() возвращает значение хэш-кода для объекта.

- equals(Object obj) указывает, равен ли какой-либо другой объект этому объекту.
- clone() создает и возвращает копию этого объекта.
- toString() возвращает строковое представление объекта
- finalize() вызывается сборщиком мусора на объекте, когда сборщик мусора определяет, что больше нет ссылок на объект.

Далее идут методы, которые мы не изучали, но тем не менее они есть в классе object:

- notify() пробуждает один поток, который ожидает на мониторе этого объекта.
- notifyAll() пробуждает все потоки, которые ожидают на мониторе этого объекта.
- wait(long timeout) заставляет текущий поток ждать, пока другой поток не вызовет метод notify() или метод notifyAll() для этого объекта, или пока не истечет указанное количество времени. Метод wait бывает также с 2 параметрами и без параметров.

b. Зачем нужны методы *equals* и *hashCode*?

Эти методы широко используются в стандартных библиотеках Java при вставке и извлечении объектов в HashMap. Метод equals также используется для обеспечения хранения только уникальных объектов в HashSet и других Set реализациях, а также в любых других случаях, когда нужно сравнивать объекты.

Также их можно переопределять при создании пользовательских типов данных.

c. Что произойдет, если в finalize будет бесконечный цикл?

OutOfMemoryError, если сборщик мусора отработает. Если нет, то ничего не произойдет.

d. Как бы Вы решали такое с точки зрения разработчика?

Мы можем поймать Error, поскольку он наследуется от класса Throwable. Однако делать это крайне не рекомендуется, поскольку возникновение Error сообщает, что возникли серьезные проблемы при работе операционной системы, которые наша программа не должна пытаться поймать.

Лучшее решение - не делать бесконечных циклов в finalize (и вообще).

e. Тонкости применения *equals* и *hashCode*

Используя equals, мы должны придерживаться основных правил, определённых в спецификации Java:

- Рефлексивность — x.equals(x) возвращает true.
- Симметричность — x.equals(y) <=> y.equals(x).
- Транзитивность — x.equals(y) <=> y.equals(z) <=> x.equals(z).
- Согласованность — повторный вызов x.equals(y) должен возвращать значение предыдущего вызова, если сравниваемые поля не изменялись.
- Сравнение null — x.equals(null) возвращает false.

Правила для hashCode:

- Повторный вызов hashCode для одного и того же объекта должен возвращать одинаковые хеш-значения, если поля объекта, участвующие в вычислении значения, не менялись.
- Если equals() для двух объектов возвращает true, hashCode() также должен возвращать для них одно и то же число.
- При этом неравные между собой объекты могут иметь одинаковый hashCode.

7. Особенности работы со строками в *Java*

Если строка создана с помощью конструктора, то она записывается в кучу, а если же она была создана, как строковый литерал, то она уже будет записана в пуле строк.

a. Что такое пул строк, как он работает?

Пул строк - это специальное отведенное место в куче, в котором хранятся только уникальные представления строковых литералов. При создании строк с одинаковым содержимым, у нас единожды создается идентичная строка в пуле строк, а далее только увеличивается количество ссылок к этой строке.

b. Бывают ли аналогичные пулы для числовых типов?

Для числовых примитивов таких пулов нету, поскольку они не являются ссылочными типами, однако для их wrapperов такие пулы существуют подобно строковым.

8. *Autoboxing* / *unboxing*

a. Зачем нужны классы-обертки?

Autoboxing происходит:

- При присвоении значения примитивного типа переменной соответствующего класса-обертки.
- При передаче примитивного типа в параметр метода, ожидающего соответствующий ему класс-обертку

Например: Integer iOb = new Integer(7); Double dOb = new Double(7.0);.

Unboxing происходит:

- При присвоении экземпляра класса-обертки переменной соответствующего примитивного типа.
- В выражениях, в которых один или оба аргумента являются экземплярами классов-оберток (кроме операции == и !=).
- При передаче объекта класса-обертки в метод, ожидающий соответствующий примитивный тип.

Например: int i = iOb; double d = dOb;

9. Дженерики в Java

Дженерики (обобщения) — это особые средства языка Java для реализации обобщенного программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

a. Что значит <?>, <? *extends T*>, <? *super T*>?

Запись вида "? extends ..." или "? super ..." — называется символом подстановки, с верхней границей (extends) или с нижней границей (super). List<? extends Number> может содержать объекты, класс которых является Number или наследуется от Number. List<? super Number> может содержать объекты, класс которых Number или у которых Number является наследником (супертип от Number).

Запись вида Collection<?>равносильна Collection<? extends Object> , а значит — коллекция может содержать объекты любого класса, так как все классы в Java наследуются от Object – поэтому подстановка называется неограниченной.

10. Контейнеры в *Java*: особенности реализации

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

- Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций
- Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
- Deque: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
- List: наследует интерфейс Collection и представляет функциональность простых списков
- Set: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
- SortedSet: расширяет интерфейс Set для создания сортированных коллекций
- NavigableSet: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
- Map: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

11. *LinkedList* – устройство

LinkedList — реализует интерфейс List. Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Итератор поддерживает обход в обе стороны. Реализует методы получения, удаления и вставки в начало, середину (по индексу) и конец списка. Позволяет добавлять любые элементы в том числе и null.

12. *ArrayList* – устройство

Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. Проще говоря, ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано.

Хотя мы можем свободно добавлять в объект ArrayList дополнительные объекты, в отличие от массива, однако в реальности ArrayList использует для хранения объектов опять же массив. По умолчанию данный массив предназначен для 10 объектов. Если в процессе программы добавляется гораздо больше, то создается новый массив, который может вместить в себя все количество. Подобные перераспределения памяти уменьшают производительность. Поэтому если мы точно знаем, что у нас список не будет содержать больше определенного количества элементов, например, 25, то мы можем сразу же явным образом установить это количество, либо в конструкторе: ArrayList<String> people = new ArrayList<String>(25);, либо с помощью метода ensureCapacity: people.ensureCapacity(25);

a. Нужен ли метод trimToSize по и почему?

Метод trimToSize() класса java.util.ArrayList в Java урезает емкость этого экземпляра ArrayList до текущего размера списка. Приложение может использовать эту операцию для минимизации хранения экземпляра ArrayList.

Использовать метод бесполезно, если ArrayList включает в себя мало элементов или если элементы будут в него добавляться после использования метода. (В общем, метод практически не нужен)

13. Сравнение *LinkedList* и *ArrayList*

Почти всегда лучше использовать ArrayList вместо LinkedList.

В LinkedList не работает кэширование, так как ноды расположены в разных частях памяти.

К тому же, LinkedList занимает в среднем больше памяти из-за ссылок на другие ноды.

14. Устройство *HashMap*

Эта структура данных устроена по принципу “ключ-значение”, ключи хешируются.

При определенном количестве HashMap перестраивается в TreeMap.

a. Что такое коллизия?

Коллизией называется случай, когда для двух различных пар ключ-значение у нас совпадает хэш для ключа.

b. Как решаются проблемы с коллизиями?

При возникновении коллизии Hashmap создает связный список для корзины текущего хэша, что может ухудшить асимптотику до O(n). При большом количестве коллизий в одной ячейке, если элемент реализует интерфейс Comparable, связный список перестраивается в дерево, что улучшает асимптотику до O(log n).

15. Можно ли потерять элемент в *HashMap*?

a. Обоснуйте ответ

Неправильно имплементированные методы equals() и hashcode() для пользовательского типа ключа могут стать причиной потери элемента в HashMap.

16. Виды и отличия исключений в *Java*

Исключения подразделяются на два основных типа Unchecked, унаследованные от RuntimeException, и Checked, наследуемые от Exception.

a. Что такое исключение и для чего оно необходимо?

Exception сообщает нам, что во время работы программы в определенном месте возникла исключительная ситуация. В зависимости от типа исключения мы можем настроить работу нашей программы, чтобы избежать ее аварийного завершения, а также добавить различные обработчики этих исключений.

b. Целесообразно ли “обертывать” всю программу в один *try catch*?

Нет, поскольку, если обернуть всю программу в try catch, то все исключения буду подниматься на самый высокий уровень. Это маскирует реальные ошибки программы. Также замедляет ее работу и потребляет дополнительную память, поскольку работа кода исключений не оптимизирована, а также каждое исключение содержит в себе еще и StackTrace.

17. Как устроен *Try-with-resources*

С помощью конструкции try-with-resources мы можем объявить ресурсы, которые будут использоваться в try-блоке, и эти ресурсы будут автоматически закрыты по окончании выполнения try-блока. Ресурсом будем называть объект, являющийся экземпляром класса, который реализует интерфейс java.lang.AutoCloseable.

a. Почему вызывается *close*?

Потому что ресурсы реализуют интерфейс AutoCloseable. То есть для безопасного завершения работы с этим ресурсом он должен быть закрыт с помощью метода close(). При этом ресурсы будут закрываться в порядке обратном порядку их написания.

b. Как это реализовано в *Python* и в *C#*?

В Python для использования ресурсов с последующим их закрытием используется with():

В C# для этого используется using() {}

18. Можно ли “поймать” *Error* и почему?

Да, мы можем поймать Error, поскольку он наследуется от класса Throwable. Однако делать это крайне не рекомендуется, поскольку возникновение Error сообщает, что возникли серьезные проблемы при работе операционной системы, которые наша программа не должна пытаться поймать.

a. А *Throwable*?

Очевидно мы можем ловить и Throwable, поскольку он представляет собой суперкласс, для всех исключений и ошибок в Java, но делать это опять же категорические не стоит, поскольку при попытке поймать Throwable, будут пойманы не только Exception, но и Error, которые, как уже говорилось выше ловить не рекомендуется.

19. Почему строка является популярным ключом в *HashMap* в *Java*?

Поскольку строки неизменны, их хэшкод кэшируется в момент создания, и не требует повторного пересчета. Это делает строки отличным кандидатом для ключа в Map и они обрабатываются быстрее, чем другие объекты-ключи HashMap. Вот почему строки преимущественно используются в качестве ключей HashMap.

20. На что это влияет и влияет ли вообще последовательность блоков *catch* в *try*?

Последовательность блоков влияет на то, в каком порядке будет проверяться возникло ли исключение указанного типа. При это если указать, например, IOException в блоке catch выше, чем FileNotFoundException, то возникнет ошибка компиляции, поскольку первый тип исключения является суперклассом для второго. То есть, если у нас возникнет FileNotFoundException, то мы все равно зайдем в ветку с IOException, тогда получится, что ветка catch с FileNotFoundException будет являться недостижимым кодом.

21. Отличия Override / Overloading / Hiding

Override - переопределение. Сигнатура метода не меняется, но меняется его реализация в дочернем классе. Работает полиморфизм.

Overloading - перегрузка. Сигнатура метода меняется (то есть количество параметров и их тип), а имя метода остается прежним.

Hiding - сокрытие. Сигнатура метода не меняется, но меняется его реализация в дочернем классе, однако полиморфизм не работает. То есть при использовании ссылки класса-родителя для объекта класса-наследника, вызванный метод будет содержать реализацию класса-родителя. А если бы ссылка была типа класса-наследника, то вызванный метод содержал бы, соответственно, реализацию класса-наследника после сокрытия.

22. Почему в *Java* или *C# char* занимает два байта, а в *C* – один?

Потому что в Java и C# для char используется кодировка Unicode, а для C используется кодировка ASCII.

a. Почему в этих языках такие “большие” *char*-ы?

Это сделано, для упрощения работы с различными кодировками. Поскольку C более низкоуровневый язык, то для него важнее оптимизация по памяти и унифицированная кодировка для различных вычислительных машин, а Java и C# более высокоуровневые языки, поэтому здесь используется расширенная кодировка, позволяющая использовать большее разнообразие символов в char.

23. *JRE*, *JVM*, *JDK* – что это, в чем отличие

- JDK нужен для разработки (это компилятор, отладчик и т.д.).
- JRE нужен для запуска Java программ (содержит в себе JVM).
- JDK и JRE содержат JVM, которая нужна для запуска программ на Java.
- JVM является сердцем языка программирования Java и обеспечивает независимость от платформы.

a. Основные компоненты JVM

   Загрузчик классов (Classloader), сборщик мусора (Garbage Collector) (автоматическое управление памятью), интерпретатор, JIT-компилятор, компоненты управления потоками.

b. Как создать исполняемый файл?
   - javac HelloWorld.java
   - Это создаст .class файл, необходимый для JAR файла.
   - Затем сделаем файл manifest (сохраним его используя расширение .txt)
   - Запишем в manifest.txt следующее: Main-Class: HelloWorld
   - Затем создадим JAR файл с помощью команды: jar cfm HelloWorld.jar Manifest.txt HelloWorld.class
   - Запускаем исполняемый файл командой: java -jar HelloWorld.jar

24. *Instanceof*, *getClass*( ) - в чём и разница, зачем нужны и где используются?

Ключевое различие между ними заключается в том, что getClass() возвращает значение true только в том случае, если объект на самом деле является экземпляром указанного класса, но оператор instanceof может возвращать значение true, даже если объект является подклассом указанного класса или интерфейса в Java.

Методы нужны для того, чтобы узнать имя типа, который мы используем

Используются, например, для реализация полиморфизма или для приведения типов.

25. Виды, различия и примеры применения вложенных классов

- Вложенные внутренние классы – нестатические классы внутри внешнего класса.
  - Они существуют только у объектов, потому для их создания нужен объект.
  - Внутри Java класса не может быть статических переменных. Если вам нужны какие-то константы или что-либо еще статическое, выносить их нужно во внешний класс.
  - У класса полный доступ ко всем приватным полям внешнего класса. Данная особенность работает в две стороны.
  - Можно получить ссылку на экземпляр внешнего класса.
  - Пример: класс “Крыло самолета” может быть вложенным классом класса “Самолет”
- Вложенные статические классы – статические классы внутри внешнего класса.
  - Позволяют укомплектовать схожие по логике работы классы в одну структуру
  - Пример: Класс “Building”, а внутри него статические классы “House”, “Shop”, и т.д.
- Локальные классы Java – классы внутри методов.
  - они обладают всеми свойствами нестатического вложенного класса, только создавать их экземпляры можно только в методе, причем метод не может быть статическим
  - Локальные классы способны работать только с final переменными метода.
  - Локальные классы нельзя объявлять с модификаторами доступа.
  - Локальные классы обладают доступом к переменным метода.
  - Пример: допустим, что у нас есть класс Person (будет считать, что это человек) со свойствами street (улица), house (дом). Нам бы хотелось возвращать какой-то объект для доступа только к местоположению человека. Для этого, мы создали интерфейс AddressContainer с методами getStreet и getHouse, который подразумевает собой хранилище данных о местоположении человека. Далее в

методе getAddressContainer() создаем класс PersonAddressContainer и возвращаем его (то есть тем самым упаковываем street и house в класс).

- Анонимные Java классы – классы, которые создаются на ходу.
    - Использование анонимных классов оправдано, если тело класса является очень коротким, нужен только один экземпляр класса, класс используется в месте его создания или сразу после него или имя класса не важно и не облегчает понимание кода
    - Часто анонимные классы используются в графических интерфейсах для создания обработчиков событий. Например: для создания кнопки и реакции на её нажатие

26. Клонирование объектов

a. Почему метод *clone*( ) объявлен в классе *Object*, а не в интерфейсе *Cloneable*?

Контракт клонирования в Java требует, чтобы каждая реализация clone должна сначала получить клонированный экземпляр из super.clone(). Это создает цепочку, которая всегда заканчивается вызовом Object.clone, и этот метод содержит "магический" код нативного уровня, который делает двоичную копию базового raw struct, который представляет объект Java. Если этого механизма не существует, clone не будет полиморфным: метод Object.clone создает экземпляр любого класса, на который он вызван; это невозможно воспроизвести без собственного кода.

b. Как правильно клонировать *HashMap*?

В Java нет встроенной функции клонирования HashMap (глубокого), поэтому придется написать свою функцию с итеративным добавлением элементов из старого HashMap в новый

Также можно вместо итеративного добавления использовать Map.putAll() метод или .entrySet().stream().collect() (доступно в Java 8 и более поздних версиях). Еще один вариант: сериализовать в JSON и десериализовать обратно в новых объект.

27. Конструктор в *Java*

a. Можно ли сделать конструктор приватным, статическим?

Статический конструктор сделать нельзя

Конструктор можно сделать приватным, например, для использования его в паттерне проектирования Singleton, чтобы контролировать количество создаваемых объектов.

b. Что такое конструктор по умолчанию, конструктор копирования?

Конструктор по умолчанию - невидимый конструктор, который создается компилятором автоматически.

Конструктором копирования называется специальный конструктор, который принимает в качестве аргумента экземпляр того же класса для создания нового объекта на основе переданного.

Такие конструкторы применяются тогда, когда необходимо создать копию сложного объекта, но при этом мы не хотим использовать метод clone().

28. *Null* в *Java*

Null сообщает нам, что этот указатель ссылается на пустоту.

a. Как корректно обрабатывать *null*-значения?

Прежде, чем использовать значение и производить с ним какие-либо действия, всегда делать проверку, что значение не равно null. Можно это делать с помощью соответствующих конструкций if, а можно использовать Optional для нашего значения и проверять, что оно не null, с помощью уже готового метода ifPresent().

b. Что такое *Optional*?

Optional - специальная структура, состоящая из пары ключа-значение. При этом ключ - bool переменная, показывающая представлено значение или оно равняется null и значение - переменная определенного типа. При этом эта структура является полезной, поскольку будет появляться warning при попытке сразу получить значение, не проверив существует ли оно, что могут забывать делать разработчики без использования Optional.

29. Код стайл в *Java*

a. Правила наименования

При наименовании в Google codestyle не используются какие-либо префиксы и суффиксы.

Иногда в названиях могут появляться “_” для логического разделения компонент названия.

Для наименования классов используют UpperCamelCase.

Для наименования методов, переменных, параметров и не константных полей используют lowerCamelCase.

Для констант используется UPPER_SNAKE_CASE.

b. Что такое *JavaDoc*?

Javadoc — генератор документации в HTML-формате из комментариев исходного кода на Java. Javadoc — стандарт для документирования классов Java.

c. В чём отличие *JavaDoc* от строчного комментария?

Для строчного комментария обычно используется // либо /* */. При создании же JavaDoc у нас используются строго /** */. При этом в теле комментария мы описываем, что выполняет метод, параметры методов, возвращаемое значение и исключения.

30. Heap и Stack-память в *Java*

Куча используется всеми частями приложения в то время как стек используется только одним потоком исполнения программы.

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.

В куче работает сборщик мусора: освобождает память путем удаления объектов, на которые нет каких-либо ссылок.

В куче существует пул строк и пул для оберток примитивов числовых типов.

Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков.

Управление памятью в стеке осуществляется по схеме LIFO.

Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы.

31. Принципы *SOLID*

SOLID - это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

S - Single Responsibility Principle - принцип единственной ответственности. Каждый класс должен иметь только одну зону ответственности.

O - Open closed Principle - принцип открытости-закрытости. Классы должны быть открыты для расширения, но закрыты для изменения.

L - Liskov substitution Principle - принцип подстановки Барбары Лисков. Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться.

I - Interface Segregation Principle - принцип разделения интерфейсов. Данный принцип означает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

D - Dependency Inversion Principle - принцип инверсии зависимостей. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

a. Принципы *DRY vs WET*, *KISS*

DRY — don’t repeat yourself / не повторяйте себя. Если код не дублируется, то для изменения логики достаточно внесения исправлений всего в одном месте и проще тестировать одну (пусть и более сложную) функцию, а не набор из десятков однотипных. Следование принципу DRY всегда приводит к декомпозиции сложных алгоритмов на простые функции.

Концепция WET (англ. Write Everything Twice — «Пиши все дважды», или «We enjoy typing» — «Нам нравится печатать») является противоположностью DRY. Когда вы приступаете к созданию новой системы, вы не знаете, какими будут будущие

требования. Поэтому не спешите с абстракциями, ведь дублирование кода обойдется намного дешевле, чем плохая абстракция.

KISS (keep it short and simple) — это принцип проектирования и программирования, при котором простота системы декларируется в качестве основной цели или ценности. Не имеет смысла реализовывать дополнительные функции, которые не будут использоваться вовсе или их использование крайне маловероятно. Не стоит подключать огромную библиотеку, если вам от неё нужна лишь пара функций. Декомпозиция чего-то сложного на простые составляющие.

b. Примеры нарушения принципов *SOLID*

Пример того, как Singleton нарушает принципы SOLID

- S — принцип единственной ответственности. Очевидно, что синглтон противоречит ему, так как кроме своей основной функции еще управляет инстансами.
- O — принцип открытости/закрытости: объекты должны быть открыты для расширения, но закрыты для изменения. Синглтон нарушает данный принцип, так как контролирует точку доступа и возвращает только самого себя, а не расширение.
- L — принцип подстановки Барбары Лисков: объекты могут быть заменены экземплярами своих подтипов без изменения использующего их кода. Это неверно в случае с синглтоном, потому что наличие нескольких разных версий объекта означает, что это уже не синглтон.
- I — принцип разделения интерфейса: много специализированных интерфейсов лучше, чем один универсальный. Это единственный принцип, который синглтон нарушает не напрямую, но лишь потому, что он не позволяет использовать интерфейс.
- D — принцип инверсии зависимостей: вы должны зависеть только от абстракций, а не от чего-то конкретного. Синглтон нарушает его, потому что в данном случае зависеть можно только от конкретного экземпляра синглтона.

c. Что делает код STUPID?

STUPID — это акроним, который означает неудачный опыт в ООП и приводит к плохому коду. Расшифровывается как:

- Синглтон (Singleton);
- Сильная связанность (Tight Coupling);
- Невозможность тестирования (Untestability);
- Преждевременная оптимизация (Premature Optimization);
- Недескриптивное присвоение имени (Indescriptive Naming);
- Дублирование кода (Duplication).

(НЕ БУДУТ СПРАШИВАТЬ)

32. Шаблоны проектирования
   a. Определение понятия “шаблон проектирования”
   b. Назначение шаблонов, виды

33. Шаблон проектирования *Singleton*
   a. Способы реализации в *Java*
   b. Достоинства и недостатки паттерна

c. Является ли шаблон антипаттерном и почему?

34. Шаблон проектирования *Factory*

a. Назначение, достоинства и недостатки паттерна
b. Пример применения

35. Шаблон проектирования *Factory Method*

a. Назначение, достоинства и недостатки паттерна
b. Пример применения
c. Порядок действий по переходу от *Factory* к *Factory Method*