Java Junior Resume Questions

OOP

- Что такое ООП?
  - методология программирования основанная на представлении программы в виде объектов
  - каждый объект является экземпляром абстрактного типа данных, класса,
  - при этом классы образуют иерархию наследования
- Что такое класс?
  - класс абстрактный тип данных, который включаетв себя данные и методы
- Что такое объект?
  - Объект это часть пространства задачи, экземпляр класса
  - сущность которая имеет поля данных и набор операций для взаимодействия
  - сущность является экземпляром класса после операций компиляции и связывания
- Как создать неизменяемый объект в Java? Перечислите все преимущества
  - объявить final class чтобы отключить наследование и переопределение методов
  - final private все поля
  - один конструктор на все поля, так как они final
  - отключить setters() методы, чтобы нельзя было изменить поля
  - создать immutable все поля объектов должны быть также Immutable
  - возвращать полные копии полей объектов, примитивные поля можно оригиналы
  -
  - Преимущества
  - потоки можно безопасно использовать в многопоточной системе
  - не требуют конструктор копирования так как уже есть методы создания полного клона
  - реализации клонирования
  - защищенное копирование при использовании полей, т.к. делается полная копия
  - работают как хорошие ключи для Map так как их нельзя изменить
- Пример. реализация
- ```
  private static final class User {
      private final String name;
      private final boolean isActive;
      private final String userId;
      // can be constructed using this constructor ONLY !
      public User(String name, boolean isActive, String userId) {
          this.name = name;
          this.isActive = isActive;
          this.userId = userId;
      }
      public String getName() {
          return name;
      }
      public boolean isActive() {
          return isActive;
      }
      public String getUserId() {
          return userId;
      }
  }
  ```
  ```

- Преимущества
  - синхронизация — immutable объекты потокобезопасны
  - lazy hashCode — так как объекты неизменны, можно вычислять hashCode один раз перед использованием и потом кэшировать hashCode так как он не изменится
  - failure atomicity — при сбое объект не зависнет в промежуточном состоянии, они неизменны
  - не требует — защищенного копирования при использовании как поле другого объекта
  - удобны — в качестве ключей для Map<> или элементов для Set<>
  - проверку — такого объекта можно делать один раз при создании, больше не проверять
- Назовите основные принципы ООП.
  - абстракция — выделение значимых характеристик
  - инкапсуляция — объединение данных и методов в классе и сокрытие реализации класса
  - наследование — построение нового класса на базе существующего
  - полиморфизм — использование разных объектов с одинаковым интерфейсом
- Что такое наследование?
  - наследование — это процесс наследования данных и методов существующего класса, т.е. повторное использование и добавление новых данных и методов
  - свойства — отсутствие обратной совместимости
  - наследование может быть только от одного класса
  - класс наследник может иметь дополнительный or измененный функционал
  - применяется — чтобы избежать дублирования данных или кода
- Что такое полиморфизм? Какие проявления полиморфизма в Java Вы знаете?
  - полиморфизм — использование разных объектов с одинаковым интерфейсом при этом поведение каждого объекта будет разным
  - свойства — полиморфизм реализуется через наследование и переопределения методов и позднего связывания
  - абстрактные классы и реализацию методов
  - интерфейсы и переопределение методов
  - проявления — функциональный через методы класса
  - параметрический через Generics класса
- Что такое инкапсуляция?
  - инкапсуляция — объединение данных и методов в класс
  - по сути реализация принципа абстракции данных
  - сокрытие это ограничения видимости реализации класса для пользователя
  - применяется — когда нужно скрыть реализацию
  - от изменения реализации клиентом
  - для сохранения стандартного интерфейса
  - для защиты данных, работа идет только через интерфейс
  - от излишне низкого уровня абстракции и перегрузки клиента деталями
  - практика — доступ к полям класса всегда реализуется через геттеры и сеттеры
  - это позволяет осуществлять проверку и обработу данных поля
-
-

- Incapsulation Example **java04/incap**
  - пример реализации incapsulation с использованием методов Objects Java8
  - Objects nullable() NullPointerException на null, реализации с String, Supplier<String>
  - equals() реализует сравнение заданных объектов для метода User.equals()
  - hash() использует список параметров для создания hashCode
  - @NonNull аннотация пользователя, регистрируется в IDEA >> Settings>>Compiler
  - используется для генерации IllegalArgumentException если аргумент равен null
- Пример. реализация
- ```
  public class MobilePhone {
      private static final Random rnd = new Random();
      private static final int RANGE_ID = 10000;
      private final int mId;
      private final String mModel;
      private final String mBrand;
      private final String mColor;
      private final double mPrice;
      private MobilePhone(int id, String brand, String model, String color, double price) {
          this.mModel = model;
          this.mBrand = brand;
          this.mColor = color;
          this.mPrice = price;
          this.mId = id;
      }
      private MobilePhone(String brand, String model, String color, double price) {
          this(rnd.nextInt(RANGE_ID), brand, model, color, price);
      }
      public String getBrand() {
          return mBrand;
      }
      double getPrice() {
          return mPrice;
      }
      @Override
      public boolean equals(Object o) {  // autogenerated by Java8
          if (this == o) return true;
          if (!(o instanceof MobilePhone)) return false;
          MobilePhone that = (MobilePhone) o;
          return mId == that.mId &&
                  Double.compare(that.mPrice, mPrice) == 0 &&
                  Objects.equals(mModel, that.mModel) &&
                  Objects.equals(mBrand, that.mBrand) &&
                  Objects.equals(mColor, that.mColor);
      }
      @Override
      public int hashCode() {
          return Objects.hash(mId, mModel, mBrand, mColor, mPrice);
      }
      @Override
      public String toString() {
          return String.format("ID: %02d Brand: %-8s Model: %-10s Color: %-8s Price: %-8.2f",
                  mId, mBrand, mModel, mColor, mPrice);
      }
      private static <T> void checkForNull(T obj) {
          if (obj == null) {
              throw new NullPointerException();
          }
          if (obj instanceof Double && (Double) obj < 0) {
              throw new NullPointerException();
          }
          if (obj instanceof Integer && (Integer) obj < 0) {
              throw new NullPointerException();
          }
      }
  ```
  ```

- Пример. реализация продолжение статические методы

```
// NonNull with User Annotation and IDEA support
    static MobilePhone newInstance(int id, @NonNull String brand, String model,
 String color, double price) {
        validateParameters(id, brand, model, color, price);
        return new MobilePhone(id, brand, model, color, price);
    }

    static MobilePhone newInstance(String brand, String model, String color, double price) {
        validateParameters(brand, model, color, price);
        return new MobilePhone(brand, model, color, price);
    }

    // NonNull with standard Methods
    private static void validateParameters(String brand, String model, String color, double
price) {
//  brand check  inactive and blocked by @NonNull
        Objects.requireNonNull(brand);                                  // no message
        Objects.requireNonNull(model, () -> "No Model Message");  // supplier
        Objects.requireNonNull(color, "No Color data");    // string
        checkForNull(price);
    }

    // NonNull with custom Methods
    private static void validateParameters(int id, String brand, String model, String color,
double price) {
        checkForNull(id);
        validateParameters(brand, model, color, price);
    }
}
```

- Реализация MobileFactory
  - интерфейс использует методы
  - save() сохранения экземпляра в Map<String,MobilePhone>
  - getBrand() извлечения элементов одного бренда
  - getTotal() сумму цен элементов одного бренда и модели
  - Utils класс со статическим методом создания MobileFactory
  - constructor сделан private да еще и выдает Exception
  - random() UUID.randomUUID() метод генерирует строку со случайным ID
  - stream() map.values().stream() потоки используются для генерации List<MobilePhone>
- ВНИМАНИЕ. Constructor сделан private да еще и выдает Exception
- Пример. реализация MobileFactory
-

```
public interface IMobileFactory {
    MobilePhone save(MobilePhone item);
    List<MobilePhone> getBrand(String brand);
    double getTotal(String brand, String model);
}
```

```
public final class MobileFactoryUtils {
    public static MobileFactory newInstance(){
        return new MobileFactory();
    }

    private MobileFactoryUtils() {
        throw new UnsupportedOperationException("Utility class");
    }
}
```

- Пример. реализация MobileFactory продолжение
- ```
  public class MobileFactory implements IMobileFactory {
      private HashMap<String, MobilePhone> map;

      public MobileFactory() {
          this.map = new HashMap<>();
      }

      @Override
      public MobilePhone save(MobilePhone item) {
          Objects.requireNonNull(item);
          if (item.getId() == null) {
              map.put(item.getId(), item);
              return item;
          }
          final String id = UUID.randomUUID().toString();
          final MobilePhone saveItem = MobilePhone.newInstance(id,
                  item.getBrand(), item.getModel(), item.getColor(), item.getPrice());
          map.put(id, saveItem);
          return saveItem;
      }
      @Override
      public List<MobilePhone> getBrand(String brand) {
          Objects.requireNonNull(brand);
          return map.values().stream()
                  .filter(p -> p.getBrand().toLowerCase().equals(brand.toLowerCase()))
                  .collect(Collectors.toList());
      }

      @Override
      public double getTotal(String brand, String model) {
          return map.values().stream()
                  .filter(p -> p.getBrand().toLowerCase().equals(brand.toLowerCase()))
                  .filter(p -> p.getModel().toLowerCase().equals(model.toLowerCase()))
                  .mapToDouble(MobilePhone::getPrice)
                  .sum();
      }

      private static void printBrandPhones(MobileFactory factory, String brand) {
          System.out.println("All " + brand + " phones:");
          factory.getBrand(brand)
                  .forEach(System.out::println);
          System.out.println("---------------------------------");
      }

      private static void printBrandModelPhonesTotalPrice( MobileFactory factory,
  ```
- ```
                                                String brand, String model) {
          System.out.println("Total " + brand + " " + model +
  ```
- ```
                                    " phones price: " + factory.getTotal(brand, model));
      }

      public static void main(String[] args) {
          final MobileFactory factory = MobileFactoryUtils.newInstance();
          factory.save(MobilePhone.newInstance("Samsung", "Galaxy 4", "Black", 7700));
          factory.save(MobilePhone.newInstance("Samsung", "Galaxy A", "Green", 6600));
          factory.save(MobilePhone.newInstance("Samsung", "Galaxy A", "Silver", 6900));
          factory.save(MobilePhone.newInstance("Apple", "iPhone 5s", "White", 5500));
          factory.save(MobilePhone.newInstance("Apple", "iPhone 6", "Red", 4444));
          factory.save(MobilePhone.newInstance("Apple", "iPhone 6", "Gray", 5400));

          printBrandPhones(factory, "Samsung");
          printBrandPhones(factory, "Apple");

          printBrandModelPhonesTotalPrice(factory, "Samsung", "Galaxy A");
          printBrandModelPhonesTotalPrice(factory, "Apple", "iPhone 6");
      }
  }
  ```
-
  ```

- Что такое абстракция?
  - абстракция — выделение значимых характеристик объекта и отбрасывание ненужных
  - идея выбрать уровень абстракции чтобы работать с простыми объектами
  - и при этом с достаточной точностью.
  - частные случаи — абстракции это интерфейсов, абстрактных классов
  - интерфейс — позволяет представить разные объектв системе одним типом
  - абстрактный класс — позволяет задать единый интерфейс группе потомков
  - Пример. — на дороге объект абстракции машина обладает габаритами, массой и скоростью
  - в мастерской объект машина обладает иерархией узлов и агрегатов
  -
- В чем преимущества объектно-ориентированных языков программирования?
  - объекты представляют реальные объекты, позволяют напрямую перевести систему в программу
  - инкапсуляция — защищает реализацию и упрощает работу
  - наследование — обеспечивает повторное использование данных и методов
  - полиморфизм — позволяет расширять систему не меняя конструкции существующей
- В чем недостатки ООП
  - неэффективность
  - на этапе выполнения — операция динамического связывания работает медленнее
  - использование геттеров и сеттеров медленнее прямого обращения
  - распределения памяти — каждый объект использует две ссылки, на объект и на класс
  - излишняя универсальность — в библиотечном классе часто больше методов чем требуется
  - это увеличивает время выполнения и размер кода
  - понимание ООП — серьезные затраты по времени на изучение концепций ООП
  - ООП порождает большие иерархии классов
  - отследить логику бывает очень сложно
  - многоразовое использование требует изучения библиотек классов
  - инкапсуляция — сложно изучать классы библиотек доступные только через JavaDoc
  - проектирование классов — сложная задача, требует большого опыта
  -
  -
- Как использование объектно – ориентерованного подхода улучшает разработку программного обеспечения?
  - повторное использование — наследование
  - реальное отображение — объекты напрямую отражают реальные объекты
- Имеется выражение «является» и «имеет». Что они подразумевают в плане принципов ООП? В чем разница между композицией и агрегацией?
  - IS-A — является — реализует наследование
  - HAS-A — имеет — реализует композицию
  - HAS-A — содержит — реализует агрегацию
  - Композиция отличается от агрегации, объект при агрегации может существовать независимо
  - House HAS_A Room — при удалении House объект Room исчезнет вместе с House
  - House HAS_A Student — при удалении House объект Student может существовать

- Что вы подразумеваете под полиморфизмом, инкапсуляцией и динамическим связыванием?
  - полиморфизм — способность объекта данного типа ссылающегося на объекты разного типа
  - вызывать методы, реализованные по разному для конкретного типа
  - полиморфизм — использует динамическое связывание, когда реализация метода
  - присваивается объекту во время выполнения
  - инкапсуляция — сокрытие внутренней реализации класса
- Что такое интерфейс?
  - интерфейс — является шаблоном поведения
  - содержит шаблон поведения через абстрактные методы
  - свойства — определяет отношение между объектами через поведение и никак иначе
  - допускает множественное наследование
  - решает проблему ромбовидного наследования т.к наследуется только сигнатура
  - реализует концепцию контракта на поведение классов, реализующих интерфейс
  - компенсируют отсутствие множественного наследования
- ВНИМАНИЕ. Java8 интерфейсы ???
- Что такое абстрактный класс?
  - абстрактный класс — является шаблоном структуры класса
  - содержит поля данных которые определяют стояние объектов класса
  - содержитшаблон поведения через абстрактные методы
  - может содержать реализацию поведения через реализованные методы
  - свойства — нельзя создать экземпляр абстрактного класса, можно только наследовать
  - каждый потомок обязан реализовать абстрактные методы
  - как интерфейсы, абстрактный класс реализует концепцию контракта

JavaCore

- Чем отличается JRE, JVM и JDK?
  - JRE минимальная реализация JVM для запуска application состоит из JVMи библиотеки классов
  - JVM собственно виртуальная машина, основная часть исполняемой системы JRE
  - интерпретирует байт код приложения, созданный компилятором javac из исходного кода
  - JDK полный комплект разработки, состоит из компилятора javac, стандартных классов Java,
  - документации, утилит и исполнительной системы JRE
- Опишите модификаторы доступа в Java.
  - private доступ внутри класса
  - default доступ внутри пакета
  - protected доступ внутри пакета и в других пакетах если есть наследники
  - public доступ везде
- Что такое package level access
  - это доступ внутри пакета default
- Чем абстрактный класс отличается от интерфейса?
  - абстрактный класс описывает структуру класса, модификатор abstract
  - нельзя создать экземпляр абстрактного класса
  - содержит поля, статические и нестатические
  - нестатические методы могут быть абстрактные или реализованные
  - статические методы обязаны иметь реализацию
  - интерфейс описывает контракт поведения, модификатор implements
  - содержит абстрактные методы
  - статические методы обязаны иметь реализацию
  - default методы это методы реализованные в интерфейсе по умолчанию
  - наследование интерфейсав предпочтительнее из-за множественного наследования
- В каких случаях Вы бы использовали абстрактный класс, а в каких интерфейс?
  - интерфейс для классов которые уже имеют структуру и встраивание абстрактного
  - класса не оправдывает переработку структуры классов
  - когда надо собрать коллекцию разнородных объектов с одним поведением
  - абстрактный класс много объектов с одинаковой или похожей структурой и поведением
  - много объектов с одинаковым поведением и чтобы сократить код и не
  - писать для каждого класса implements
  - для ускорения кода, вызов метода абстрактного класса работает быстрее
- Может ли объект получить доступ к private-переменной класса? Если, да, то каким образом?
  - стандартный для объекта этого класа напрямую, через методы класса
  - для объекта другого класса через методы класса
  - reflections через reflections защита доступа работает только для методов
- Для чего в джаве статические блоки?
  - инициализация статических полей класса
- Можно ли перегрузить static метод?
  - статические методы можно перегрузить, как у обычных методов использовать другую сигнатуру
- Можно ли переопределить static метод?
  - статические методы нельзя переопределить
  - при наследовании, статический метод можно переписать, это называется сокрытие

- Расскажите про внутренние классы. Когда вы их будете использовать?
  - static nested class — статический вложенный класс
  - — по сути это внешний класс, который упакован как внутренни
  - — полное наследование через экземпляр внешнего класса
  - inner class — нестатический вложенный класс
  - — не может иметь статические методы, может иметь static final поля
  - — обращение к такому же методу внешнего класса Ext.this.method()
- ВНИМАНИЕ. Всегда использовать Static Nested Class когда можно, внутренние классыМЕШАЮТ работе gc
  - method local class — внутренний класс метода
  - — полноценный внутренний класс
  - — не может иметь статические методы, может иметь static final поля
  - anonymous inner class — анонимный внутренний класс
  - — не имеет конструктора, так как класс не имеет имени
  - — обращение к полям и методам внешнего класса Ext.this.method()
  - — не может иметь статические методы, может иметь static final поля
  -
- ВНИМАНИЕ. Anonymous Classes создают безымянный внутренний класс и ТАКЖЕ МЕШАЮТ работе gc
- Использование
  - static nested class — используется для удобства упаковки классов в коде
  - inner class — используются когда нужна структура только внутри класса
  - — и внутренний класс НЕ ИМЕЕТ смысла без ЭКЗЕМПЛЯРА внешнего класса
  - method local class — используется когда нужна струкутра только внутри метода
  - anonymous class — используются для реализации интерфейса и расширения объекта методами
  - — для расширения объекта другого класса новыми методами
- В чем разница между переменной экземпляра и статической переменной? Приведите пример.
  - переменная экземпляра создается при создании объекта
  - статическая переменная создается при загрузке класса
- Пример. реализация
- ```
  public class A {
      private static int sA;
      private int mA;
  }
  ```
- Приведите пример когда можно использовать статический метод?
  - статические методы используются для инициализации статических переменных или выполнения
  - сервисных операций, то есть таких когда не требуется создание экземпляра
  - Пример. реализация
  - Библиотека методов математических операций Math класса.
- Расскажите про классы- загрузчики и про динамическую зарузку классов.
  - загрузчики классов — Bootstrap, System Classloader, Extension Class Loader
  - Bootstrap — базовый загрузчик на базе JVM, загружает классы java/* из $JAVA_HOME/lib
  - — пользователь к нему доступа не имеет, но можно управлять –Xbootclasspath
  - Extension Classloader — загрузчик расширений, загружает классы из $JAVA_HOME/lib/ext
  - System Classloader — загрузчик на уровне JRE, загружает классы $CLASSPATH
  - — можно управлять –classpath
  - custom загрузчик — должен расширять java.lang.Classloader
  ```

- Процедура загрузки класса практически ВСЕ классы загружаются Bootstrap из rt.jar пакета /java.lang*
    - Процедура поиска класса в Cache
    - System Classloader Cache? >> Extension Classloader Cache? >> Bootstrap Classloader Cache?
    - Если поиск не нашел класс в разделах Cache, управление на загрузку
    - Bottstrap Load ? >> Extension Classloader ? >> System Classloader ? >> ClassNotFoundException
    - Если класс найден или загружен на любом этапе, процесс загрузки завершается
- Пример. реализация
- ```
  public class Main {
      public static void main(String[] args) {
          C c = new C();
          B b = new B();
          A a = new A();
      }
  }
  ```
- ```
  javac *.java
  java -verbose:class Main.class
  ```
- Статическая загрузка классов
    - при помощи оператора new A()
- Динамическая загрузка классов
    - загрузка в процессе работы при помощи метода forName() или при помощи ClassLoader
- Что такое статическая и что такое динамическая загрузка класса?
- Динамическая загрузка классов
    - загрузка в процессе работы при помощи метода forName() или при помощи ClassLoader
- Пример. реализация

```
public class Main {
    public static void main(String[] args) {
        A a = new A();                                  // статическая загрузка класса
        B b = new B();
        try {
            Class classA1 = Class.forName("week1.classloader.bin.A1"); // динамическая
            ClassLoader loader = D1.class.getClassLoader();
            Class cD1 = loader.loadClass("week1.classloader.bin.D1"); // динамическая
            Class cB1 = loader.loadClass("week1.classloader.bin.B1");

            A1 a1 = (A1) classA1.newInstance();
            A1 d1 = (D1) cD1.newInstance();
            A1 b1 = (B1) cB1.newInstance();
            System.out.println(a1 + " " + d1 + " " + b1);
        } catch (ClassNotFoundException | IllegalAccessException |
                                                InstantiationException e) {
            e.printStackTrace();
        }
    }
}
```

- Для чего нужен оператор "assert" в джава?
    - оператор проверки утверждений, если условие не выполняется (=false), то вызывается
    - исключение AssertionError, для использования включить флаг -ea
- Пример. реализация

```
public class Main {
    public static void main(String[] args) {
        System.out.println("To Activate Assert use switch -ea for JVM");
        boolean flag = false;
        System.out.println("flag value:"+flag);
        assert flag: "flag is not true";
    }
}
```

- Почему в некоторых интерфейсах вообще не определяют методов?
  - это интерфейсы маркеры, они указывают компилятору Java, что данный класс поддерживает особый тип поведения. С Java 1.5 надобность в них отпала, используются Аннотации
  - java.lang.Serializable, java.lang.Clonable, java.lang.Remote, java.lang.ThreadSafe
- Пример. реализация

```
Если класс поддерживает Serializable, то компилятор проверяет все подтипы и если
встречает NonSerializable, то выдается исключение  NotSerializableException
```

- Какая основная разница между String, StringBuffer, StringBuilder?
  - String immutable class следовательно всегда создается новый объект при изменении
  - использовать когда есть объявление в одну строку
- ВНИМАНИЕ. НИКОГДА не использовать в цикле, в каждой итерации будет создан отдельный объект
  - StringBuilder mutable class, nonsynchronized, нельзя использовать в multithreading, но он быстрее
  - использовать в одном потоке, когда надо много добавлять в строку
  - StringBuffer mutable class, synchronized, можно использовать в многопоточной среде
  - использовать в многопоточной среде, когда надо много добавлять в строку
- Расскажите про потоки ввода-вывода Java.
  - байтовые потоки InputStream, OutputStream работают побайтно
  - символьные потоки InputStreamReader, OutputStreamWriter работают посимвольно
  - символ Unicode это 2 байта, позволяют подключить Charset
  - FileReader, FileWriter производный от Reader, Writer
  - стандартные потоки StdIn, StdOut, Err
  - StdIn стандартный ввод, с клавиатуры System.in
  - StdOut стандартный вывод в поток System.out
  - Err стандартный поток ошибок System.err
  - буферизованные версии используются для ускорения обмена
  - отличаются наличием метода flush() сброса буфера на носитель
- Что такое String Pool?
  - это cache вида WeakHashMap для String которые создаются кавычками (не new String)
  - String pool существенно экономит память в Heap за счет скорости
  - intern() значение объекта из pool
  - расположен в Permanent Generation (Java6) или обычный Heap (Java8)
  - String s = new String("Cat")
  - создает ОДНУ строку, если уже есть в String pool "Cat", создает строку в String Pool
  - создает ДВЕ строки, если нет в String pool "Cat", одну в String pool, другую в Heap
  - Пример. реализация

```
String s1 = "Cat";
String s2 = new String("Cat");
String s3 = "Cat";
String s4 = new String("Cat");
String s5 = s2.intern();
String s6 = s3.intern();
```

- Когда использовать String.intern()
  - intern() это метод поиска и размещения строки в String pool
  - строка созданная "" или new String() автоматом пополняет String pool
  - non intern StringBuilder() или считанная с файла строка не размещается в String pool
  - intern() позволяет разместить строку в String pool и сравнивать == по ССЫЛКЕ
- Преимущества
  - intern() ускоряет сравнение строк по ссылке, если сравнений строк много

- Недостатки
  - intern() РАСХОДУЕТ память, так как String pool не очищается при удалении строк
  - блокирует глобальный ресурс и ЗАМЕДЛЯЕТ работу в многопоточном приложении
- Tricks
  - CONSTANTS просто константа при изменении в основном классе, в использующем останется
  - прежней, если не сделано перекомпилирование
  - intern() отменяет константу и любое изменение кода не требует перекомпиляции
  - использующих библиотек, так как они используют не константу, а значение метода
  - `public static final String CONSTANT = "value".intern();`
  - `public static final String configOption1 = "some option".toString();`
  -
- Что такое Integer Pool?
  - cache для Character, Byte, Integer, Long значений в диапазоне -128..127 включительно
  - переменные с такими значениями всегда и автоматом становятся примитивами
  - и их можно сравнивать по ссылке
  - расположен в пуле постоянной среды выполнения
  -
- Пример. реализация
- ```
  public static void main(String[] args) {
      Integer i1 = 127;
      Integer i2 = 127;

      Character c1 = '\u007f';
      Character c2 = '\u007e';
      c2++;

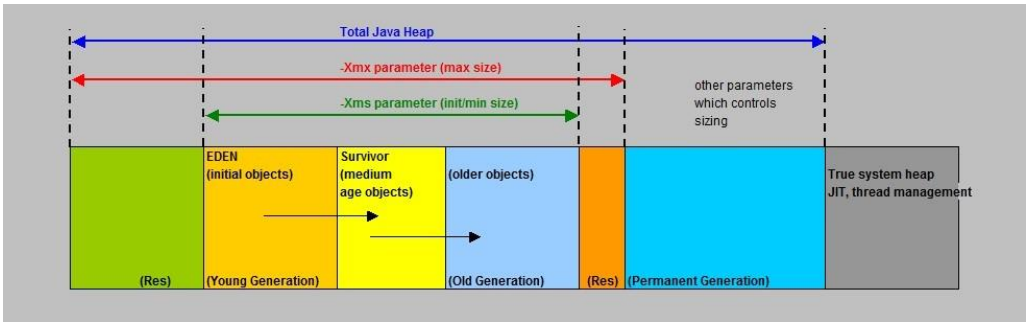
      Byte b1 = 127;
      Byte b2 = 127;

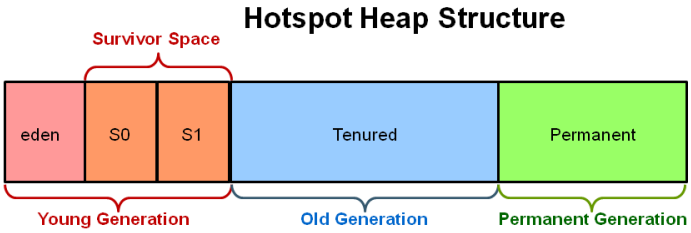
      Short s1 = 126;
      Short s2 = 127;
      s1++;

      final Long l1 = 127L;
      final Long l2 = 127L;

      System.out.println("i1==i2: "+(i1==i2));
      System.out.println("c1==c2: "+(c1==c2));
      System.out.println("b1==b2: "+(b1==b2));
      System.out.println("s1==s2: "+(s1==s2));
      System.out.println("l1==l2: "+(l1==l2));
  }
  ```
- Примитивные типы в Java?
- byte: 8-bit signed two's complement integer. -128..127 (inclusive)
- short: 16-bit signed two's complement integer. -32,768..32,767
- int: 32-bit signed two's complement integer. -2^31..2^31-1 Java SE 8 unsigned 32-bit integer, 0..23^2-1.
- long: 64-bit two's complement integer. -2^63..2^63-1. Java SE 8 unsigned 64-bit long, 0.. 2^64-1.
- float: 32-bit IEEE 754 floating point.
- double: 64-bit IEEE 754 floating point.
- boolean: two possible values: true and false.
- char: 16-bit Unicode character. '\u0000' (or 0)..'\uffff' (or 65,535 inclusive).
  ```

- Что такое Heap и Stack память в Java?
  - Heap — динамическая память создается при старте JVM, хранит все объекты и JRE классы
  - здесь работает GC, удаляет объекты, на которые нет ссылок в приложении
  - объект, созданный в Heap, имеет глобальный доступ из любой точки программы
  - JRE классы загружаются в Heap при старте программы
  - свойства — все объекты создаются в Heap и существуют только в Heap
  - в Heap хранится тело объекта, размер в памяти зависит от числа полей объекта
  - размер Heap задается при старте приложения JVM -Xms, -Xmx ключами
  - при переполнении Heap выбрасывается исключение java.lang.OutOfMemoryError
  - Stack — непрерывная область памяти по принципу LIFO, как стандартный стек
  - хранит примитивы и ссылки на объекты в Heap
  - при вызове любого метода в Stack создается новый блок с примитивами, ссылками
  - при выходе из метода, блок удаляется
  - свойства — все переменные метода хранятся в стеке, доступ только из метода, локальный
  - объект метода создается в Heap, ссылка на него хранится в стеке
  - при выходе из метода ссылки удаляются, объекты собирает GC.
  - размер стека задается JVM –Xss ключом
  - при переполнении Stack выбрасывается исключение java.lang.StackOverflowError
- Какая разница между Stack и Heap памятью в Java?
  - Heap — используется всеми частями приложения
  - содержит все объекты созданные в приложении
  - объекты в Heap доступны из любой части приложения
  - резервирование памяти по системе Young, Old Generation System, медленнее Stack
  - Heap существует пока работает приложение
  - размер памяти Heap намного больше размера памяти Stack
  - Stack — используется только текущим методом
  - хранит локальные переменные, параметры методов и ссылки на объекты в Heap
  - переменные доступны только из метода, для которого создан блок памяти в Stack
  - резервирование памяти по системе LIFO, быстрее Heap из за простой логики LIFO
  - блок памяти существует ограниченное время
- Расскажите про модель памяти в джава?
  - Heap
  - Young Generation
  - Eden (init) — начальная область памяти сюда попадают все созданные объекты
  - когда заполняется на некий процент, GC проводит minor collection
  - Survivor(mid) — сюда попадают объекты выжившие после быстрой сборки в Eden
  - время от времени долгоживущие объекты перемещаются в Tenured
  - Tenured(old) — хранятся долгоживущие объекты,singletones, менеджеры ресурсов
  - когда заполняется эта область проводится major collection
  - MetaSpace Java8 — не является ЧАСТЬЮ Heap, растет автоматом по мере заполнения
  - ~~Permanent Generation~~ — хранит данные и методы класса (статические поля и все методы)
  - Runtime Constant Pool находится ЗДЕСЬ
  - Stack
  - Thread Stack — блок памяти под конкрентый поток
  - в нем хранятся локальные переменные, параметры методов





- JMM Java Memory Model
  - JMM модель памяти Java для работы с потоками описывается JSR133 стандартом
  - постулаты каждое значение читаемое одним потоком записано каким-то другим потоком
  - правила упорядочения
  - каждое действие с переменной в потоке «happens-before» кода который идет ниже
  - освобождение мониторов «happens-before» следующего захвата того же монитора
  - запись в volatile «happens-before» следующего чтения этого же поля
  - вызов start() потока происходит «happens-before» любых действий в этом потоке
  - все действия в потоке «happens-before» действий в другом потоке join() к данному
  - работа final полей
  - значения final полям надо задать в конструктор и затем видимы другим потокам
- ВНИМАНИЕ. Нельзя давать доступ к переменной пока незавершен конструктор
  - Пример. реализация ПРАВИЛЬНО

```
class FinalFieldExample {
    final int x;
    int y;
    static FinalFieldExample f;
    public FinalFieldExample() {
        x = 3;
        y = 4;
    }
    static void writer() {
        f = new FinalFieldExample();
    }
    static void reader() {
        if (f != null) {
            int i = f.x;
            int j = f.y;
        }
    }
}
```

  - Пример. реализация НЕПРАВИЛЬНО

```
public FinalFieldExample() { // bad!
    x = 3;
    y = 4;
    global.obj = this; // bad construction - allowing this to escape
}
```

  - ВНИМАНИЕ. Конструктор еще незавершен, а объект obj уже получил ссылку на экземпляр

- Как работает сборщик мусора (garbage collector)?
  - GC методы Marking, Deletion, Deletion with Compacting
  - Marking помечает объекты на удаление
  - Deletion удаляет объекты из памяти
  - Deletion with Compacting упаковывает выжившие объекты
  - GC стратегия сборки
  - отслеживает заполнение Eden (init), при заполнении, GC проводит minor collection
  - Eden удаление мусора и если места мало, упаковка выживших объектов вместе
  - перенос выживших после быстрой сборки объектов в Survivor(mid)
  - перенос выживших после многократных сборок мусора объектов в Tenured(old)
  - GC утилиты
  - jstat утилита сбора статистики GC по <PID> jstat -gc <PID>
  - jmap утилита статистики по <PID> на JVM jmap -heap <PID>
  - jmap -dump:file=<FILE> <PID>
  - jcmd утилита диагностики по <PID> jcmd <PID> GC.heap_dump filename=<FILE>
  - jhat утилита парсинга jcmd dump файла jhat <FILE>
  - hprof агент профайлинга heap и CPU java -agentlib:hprof <ProfiledClass>
  - java -agentlib:hprof=heap=sites <ProfiledClass>
  - allocation profile javac –J-agentlib:hprof=heap=sites Hello.java
  - heap dump javac –J-agentlib:hprof=heap=dump Hello.java
  - sampling profile results javac –J-agentlib:hprof=cpu=samples Hello.java
  - jps утилита процессов запущенных на JVM
- Расскажите про приведение типов. Что такое понижение и повышение типа? Когда вы получаете ClassCastException?
  - приведение типа это установка типа объекта отличное от текущего
  - автоматическое вверх при расширении типа от int к double, от потомка к предку
  - не автоматическое вниз при сужении типа от double к int, от предка к потомку
  - ClassCastException выдается когда тип объекта и приведение не совпадают
- Что такое статический класс, какие особенности его использования?
  - статический класс это вложенный класс, в котором нет ссылки на объект внешнего класса
  - по сути статический вложенный класс аналогичен внешнему классу
  - может содержать статические поля и методы
  - для доступа к статическому методу вложенного класса не нужен экземпляр
  - каждый вложенный статический класс может реализовать интерфейс
  - static nested class статический вложенный класс
  - по сути это внешний класс, который упакован как внутренний
- ВНИМАНИЕ. Всегда использовать Static Nested Class когда можно, внутренние классы МЕШАЮТ работе gc
  -
- Каким образом из вложенного класса получить доступ к полю внешнего класса?
  - доступ из static nested class как из любого класса только через экземпляр внешнего класса
  - можно сделать поле static и тогда доступ напрямую как к своему
  - доступ из non-static nested class или inner class через ExtClass.this
  -

- Какие существуют типы вложенных классов? Для чего они используются?
  - static nested class — статический вложенный класс
  - — по сути это внешний класс, который упакован как внутренни
  - — доступ к non-static полям только через экземпляр внешнего класса
  - inner class — нестатический вложенный класс
  - — не может иметь статические методы, может иметь static final поля
  - — обращение к такому же методу внешнего класса Ext.this.method()
- ВНИМАНИЕ. Всегда использовать Static Nested Class когда можно, внутренние классыМЕШАЮТ работе gc
  - method local class — внутренний класс метода, видимость только внутри блока, метода
  - — полноценный внутренний класс
  - — не может иметь статические методы, может иметь static final поля
  - anonymous inner class — анонимный внутренний класс,видимость только внутри блока метода
  - — не имеет конструктора, так как класс не имеет имени
  - — к переменным блока или final, или effectively final т.е. не менялась
  - — не может иметь статические методы, может иметь static final поля
- ВНИМАНИЕ. Anonymous Classes создают безымянный внутренний класс и ТАКЖЕ МЕШАЮТ работе gc
- Возможно ли при переопределении (override) метода изменить:
  - Модификатор доступа — да, в сторону расширения, то есть вверх, доступа
  - Возвращаемый тип — да, в сторону сужения, то есть вниз и только для объектов
  - Тип аргумента или количество — нет
  - Имя аргументов — да, главное тип и число аргументов
  - Изменять порядок, количество или вовсе убрать throws?
  - — да можно в наследуемых методах убрать throws
  - — да можно добавлять новые, наследующие те, что у предка
- Что такое autoboxing?
  - автоматические приведение типа между объектом и примитивными типами
- Что такое Generics?
  - это обобщенный тип, применяется для методов и классов работающих с несколькими типами
  - на этапе компиляции происходит замена обобщенного типа на реальный тип данных
  - Пример. реализация
  - LinkedList<E> — коллекция для любого типа в Java
- Какова истинная цель использования обобщенных типов в Java?
  - первоначальная цель — создание коллекций с обобщенным типом данных
- Каким образом передаются переменные в методы, по значению или по ссылке?
  - параметры примитивных типов — передаются по значению
  - параметры объектов — передается только ссылка на объект
- Что такое native в определении метода?
  - это механизм запуска кода JNI — метод реализуется в платформо зависимом коде на C/C++
  - — метод завершается ";" как абстрактный метода
  - — метод вызывается из библиотеки C/C++ из класса Java
- Пример. реализация

```
private static native void registerNatives();
```

- Какие методы есть у класса Object?
  - Object базовый класс — от него наследуются все остальные классы
  - методы класса Object переопределяются при необходимости
- `public class Object {`

- Native implementation

  `public final native Class<?> getClass()`
  - возвращает runtime класс объекта, тот реальный класс который определен во время runtime
  -

  `public native int hashCode()`
  - возвращает hashCode объекта

  `public final native void notify()`
  - уведомляет другой поток что можно продолжать работу

  `public final native void notifyAll()`
  - уведомляет все потоки что можно продолжать работу

  `public final native void wait(long timeout) throws InterruptedException`
  - реализация wait ожидания на время timeout

  `protected native Object clone() throws CloneNotSupportedException;`
  - создает shallow copy объекта если класс объекта РЕАЛИЗУЕТ интерфейс Clonable
- ВНИМАНИЕ. Object class НЕ РЕАЛИЗУЕТ Clonable, поэтому НЕВОЗМОЖНО вызвать new Object.clone()

- Standard implementation

  `public boolean equals(Object obj)`
  - возвращает сравнивает объекты по методу equals, а не по ссылке

  `public String toString()`
  - возвращает String объект состоящий из имени класса и hash кода

  `public final void wait(long timeout, int nanos) throws InterruptedException`
  - реализация wait ожидания на время timeout в наносекундах

  `public final void wait() throws InterruptedException`
  - реализация wait бесконечное время пока не освобожден монитор

  `protected void finalize() throws Throwable`
  - метод который вызывается GC при удалении объекта
- Почему все классы наследуются от Object
  - все методы могут принимать объекты любого класса как параметры класса Object
  - коллекции любых классов можно было создавать на базе Object, это использовалось до Generics
  - все методы класса Object становятся методами любого класса созданного в Java
- Правила переопределения метода Object.equals()
  - проверить объект по ссылке — если равно, — вернуть true
  - проверить объект по типу — если тип разный — вернуть false
  - преобразовать аргумент в корректному типу
  - провести сравнение по значимым полям
  - объекты сравнивать методом equals
  - float преобразовать в int или сразу использовать Float.compare()
  - double преобразовать в long или сразу использоват ь Double.compare()
  - проверить является ли метод симметричным, транзитивным, рефлективным,непротиворечивым
  - ВСЕГДА переопределять hashCode()
  - ВСЕГДА использовать простое сравнение, то есть объекты сравнивать по полям не сложнее

- Что такое симметричный, транзитивный и непротиворечивый метод в Java?
  - симметричный возвращает true для x.eq(y) только тогда когда y.eq(x) тоже true
  - транзитивный если x.eq(y) true и y.eq(z) true тогда обязательно x.eq(z) тоже true
  - непротиворечивый если x.eq(y) true то сколько бы раз ни вызвать x.eq(y) тоже true
  - ре флективность если x.eq(x) true то есть объект должен быть равен самому себе
- Если вы хотите переопределить equals(), какие условия должны удовлетворяться для переопределенного метода?
  - надо соблюдать рефлексивность, симметричность, транзитивность и непротиворечивость
  - object.equals(null) всегда возвращает false
  - null.equals(null) не сработает так как вызовет NullPointerException
- Какая связь между hashCode и equals?
  - equals()=true hashcode() = true одинаковые объекты обязательно равный hashcode
  - hashcode = true equals() = true | false равный hashcode необязательно равные объекты
  - hashcode=false equals()=false неравный hashcode обязательно неравные объекты
  -
- Каким образом реализованы методы hashCode и equals в классе Object?
  - equals() реализовано сравнение по ссылке
  - hashCode() реализовано не в Java, через механизм native на языке C/C++.
- Что будет, если переопределить equals не переопределяя hashCode? Какие могут возникнуть проблемы?
  - коллекции Map, Set будут работать неправильно
  - Object.hashCode() использует native System.IdentityHashCode()
  - по умолчанию использует алгоритм Park-Miller RNG или генератор rnd()
  - поэтому одинаковые объекты будут иметь разный hashcode
- System. IdentityHashCode() implementation
  - выбор метода IdentityHashCode `-XX:hashCode=n` выбирает n=0..5
  - JDK7 (0 по умолчанию), JDK8 (5 по умолчанию)
  - методы генерации hashCode
  - `0 Park-Miller RNG random() default`
  - `1 (address>>3)^(address>>8)^random() код на базе ссылки`
  - `2 constant 1`
  - `3 counter++ последовательный счетчик`
  - `4 address адрес ссылки`
  - `5 XOR shift Marsaglia алгоритм на базе сдвига и XOR`
- ВНИМАНИЕ. Опция 5 наилучшая по производительности и качеству используется с JDK8
- Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode?
  - использовать поля, которые идентифицируют объект в рамках приложения
  - примитивные поля лучше, так как сравнение по ним идет быстрее
  - поля, которые будут с большой вероятностью отличаться у разных объектов
- Для чего нужен метод hashCode()?
  - метод hashCode выполняет функцию хэширования, то есть сопоставления объекту
  - уникальной битовой строки или hash Code
  - коллекции Map, Set используют hashCode как индекс для хранения данных
  - один из методов hashCode остаток от деления ключа k на m возможных значений
  - непротиворечивость метод должен возвращать один и тот же hashCode для одного объекта
  - равность метод должен возвращать одинаковый hashCode для равных объектов

- Правила переопределения метода Object.hashCode().
  - использовать — те же поля, которые участвуют в методе equals()
  - применить — рекурсивную формулу result = PRIME*result + field
  - PRIME = 31
- Расскажите про клонирование объектов. В чем отличие между поверхностным и глубоким клонированием?
  - clone copy — shallow copy клонирование, примитивные поля будут скопированы
  - поля объекты и массивы будут ссылаться на одни и те же объекты и массивы
  - изменение полей объектов клона, проявится в полях объектах оригинала
  - deep copy — полное клонирование объекта с созданием новых объектов полей делается руками
- Правила переопределения метода Object.clone().
  - вызвать — super.clone() для клонирования родителя
  - свойства — shallow copy — поверхностное копирование по умолчанию
  - поля примитивные копируются
  - поля объекты создаются новые ссылки на те же самые объекты
  - deep copy — глубокое копирование надо рекурсивно создать новые объекты
  -
- Где и как вы можете использовать закрытый конструктор?
  - Singleton — private conctructor можно запустить только из метода newInstance() этого объекта
  - таким образом создание новых экземпляров под контролем их нельзя создать извне
- Что такое конструктор по умолчанию?
  - это конструктор без аргументов
  - свойства — всегда есть у класса, где не определен ни один конструктор
  - конструктор по умолчанию всегда неявно вызывается у предка если нет других вызовов
- Опишите метод Object.finalize().
  - этот метод — вызывает GC при удалении объекта, аналогичен деструктору в C++
  - этот метод должен освободить занятые ресурсы, закрыть соединения с Database
  - в Java если надо освободить ресурсы, лучше сделать это явно, без метода finalize()
- Чем отличаются слова final, finally и finalize?
  - final — модификатор определяет как константу переменную, метод, класс
  - переменная — нельзя изменить значение
  - метод — нельзя переопределить метод
  - класс — нельзя наследоваться от класса
  - finally — блок try/catch конструкции вызывается всегда при выходе из блока try/catch
  - независимо от того, произошла ошибка или нет
  - например для закрытия открытого файла, потока или соединения
  - НЕ СРАБОТАЕТ
  - если в блоке try/catch main.Thread вызывается System.exit(0)
  - если в блоке try/catch child.Thread, и внезапно закрыт main.Thread поток
  - МЕШАЕТ
  - если finally — возбуждает Exception оно перебивает любое Exception в блоке
  - если finally — содержит return то это значение перебивает все возвраты из метода
  - если finally — меняет переменную возврата в норме ВЕРНЕТСЯ ЗНАЧЕНИЕ try
  - НОВОЕ значение будет возвращено только при Exception
  - механизм такой push value >> exec finally >> pop value >> return

    - поэтому любое значение в finally будет перебито нормальным из try
- ВНИМАНИЕ. Есть нюансы возврата из метода при работе с final, может возвращать неверное значение
    - finalize метод освобождения ресурсов при удалении экземпляра класса
    - использовать только для освобождения ресурсов, при завершения приложения
    - проверять если ресурс активен, закрывать его
- ВНИМАНИЕ. Вызывается случайно GC, поэтому НЕ ИСПОЛЬЗОВАТЬ для быстрого освобождения ресурсов

- Опишите иерархию исключений.
  - Object >> Throwable >> Exception >> RuntimeException
  - >> Other Exception
  - >> Error
  - Throwable базовый класс всех Exception в Java
  - конструкторы
  - Throwable() и Throwable(Stirng message)
  - методы
  - getMessage() выдать строку которая задана в конструкторе
  - printStackTrace() выдать ошибку на консоль
  - printStackTrace(PrintStream) вывести ошибку в байтовый поток PrintStream
  - printStackTrace(PrintWriter) вывести ошибку в символьный поток PrintStream
  - Exception исключения в результате нештатных ситуаций при выполнении приложения
  - РЕКОМЕНДУЕТСЯ обрабатывать ошибки на уровне программы
  - возможно расширить чтобы создать свое Exception
  - RuntimeException ошибки программирования, то что программист сделал в коде
  - например приведение типов, выход за массив, обращение к null
  - РЕКОМЕНДУЕТСЯ не обрабатывать по возможности и исправить код
  - Other Exception ошибки ввода вывода IOException, URL exception, Class Exception
  - по соглашению все имена Exception завершаются словом Exception
  - Error Exception внутренние ошибки в JRE
  - НЕ РЕКОМЕНДУЕТСЯ обрабатывать и выбрасывать Error Exception
  - по соглашению все имена Error Exception завершаются словом Error
- ВНИМАНИЕ. RuntimeException ЗАМЕДЛЯЕТ работу программы так как при создании копирует стек вызова
- Для УСКОРЕНИЯ надо переопределить метод fillStackTrace()
- Какие виды исключений в Java вы знаете, чем они отличаются?
  - все исключения наследуют класс Throwable
  - три группы Exception, RuntimeException и Error
  - Exception проверяемые, компилятор провеярет есть ли обработка исключений
  - IOException, компилятор проверяет обрабатывается ли ошибка файла
  - методы которые не проверяют исключение должны пробрасывать с throws
  - RuntimeException непроверяемые, компилятор не следит, есть ли обработка
  - задача программиста обработать исключения вида NullPointerException
  - Error непроверяемые, так как ошибки такого класса говорят о фатальном сбое,
  - программа не в состоянии справиться с ошибками такого уровня
  - Например StackOverflowError, OutOfMemoryError
- Что такое checked и unchecked Exception?
  - checked исключения которые должны обрабатываться в блоке try/catch или throws метода
  - все исключения от класса Exception
  - unchecked исключения, которые можно не обрабатывать и не описывать
  - все исключения от классов RuntimeException и Error
  -
- Как создать свой unchecked Exception?
  - создать свой класс Exception который наследует RuntimeException класс

- Какие есть Unchecked exception?
  - RuntimeErrorException
  - ArrayIndexOutOfBoundsException
    IllegalArgumentException
    NullPointerException
    NumberFormatException
    AssertionError
    ExceptionInInitializerError
    StackOverflowError
    NoClassDefFoundError
- Какие есть Checked Exception
  - Exception
    IOException
    FileNotFoundException
    ClassNotFoundException
    NoSuchMethodException
    NoSuchFieldException
- Что такое Error?
  - Error Exception unchecked не проверяются компилятором
  - так как это ошибки фатальные на уровне работы JVM программа не может
  - их исправить если они уже произошли
- Опишите работу блока try-catch-finally.
  - срабатывает один из блоков catch, остальные блоки в этом не участвуют
  - после обработки блоков try или catch всегда выполняется блок finally
  - finally используется для заключительных операций перед return из метода
- ВНИМАНИЕ. Работа с finally предосторожности
  - finally НЕ МОЖЕТ изменить значение return из блока try
  - finally МОЖЕТ заменить Exception подмену надо отслеживать
- Возможно ли использование блока try-finally (без catch)?
  - возможно используется когда return из метода выполняет однотипные операции
  - блок finally перехватывает управление и выполняет эти операции на выходе
- Всегда ли исполняется блок finally?
  - всегда кроме фатальных операций
  - System.exit(0) в главном потоке
  - прерывание потока демона в котором задействован finally
  - в блоке finally срабатывает Exception, тогда оставшаяся часть блока не выполняется
- ВНИМАНИЕ. В блоке finally Exception ПОДМЕНЯЕТ внешнее Exception и это надо отслеживать
- Какие есть оссобенности класса String? что делает метод intern().
  - final class это финальный класс от него нельзя наследовать
  - immutable значение Stringo финальне, нельзя изменить, только создать новый объект
  - hashCode метод рассчитывается по формуле h = 31 * h + val[i];
  - intern() возвращает значение строки из внутреннего пула строк, если там нет, то создает
  - используется при массивном сравнении строк, для сравнения по ссылке ==
  - увеличивает расход памяти в пуле строк

- Можно ли наследовать строковый тип, почему?
  - нельзя так как класс String является final.
- Почему для расчета hashCode выбрано число 31
  - это простое число
  - его можно получить сдвигом и вычитанием (2<<5) -1
- Почему строка является популярным ключом в HashMap в Java?
  - String immutable это означает что хэш код вычисленный при создании остается неизменным
  - hashCode кэшируется при создании строки, поэтому используется снова
  - это работает если используется та же строка или String.intern()
- ВНИМАНИЕ. Создание новых длинных строк ключей МОЖЕТ снизить скорость,hachCode зависит от длины
- ВНИМАНИЕ. String.intern() ведет к ЗАТРАТАМ памяти,скорости,использовать ТОЛЬКО если низкая скорость
- Дайте определение понятию конкатенация строк.
  - конкатенация это объединение строк
  - String().concat() метод объединения строк String НЕ ИСПОЛЬЗОВАТЬ в цикле
  - StringBuilder().append() метод объединения строк StringBuilder использовать в цикле
- Как перевернуть строку?
  - вручную использовать массив char[]
  - StringBuilder.reverse() метод реверса строки
- Как сравнить значение двух строк?
  - String.equals() метод сравнения строк по содержанию
- Как обрезать пробелы в начале и конце строки?
  - String.trim() метод обрезки пробелов в начале и конце строки
- Дайте определение понятию "пул строк".
  - String pool это HashMap в разделе PemGen(Java6) или Heap(Java8)
  - свойства хранит содержимое String объектов, которые расположены в heap
  - является cache для строк, созданных присваиванием
  - intern() позволяет создать значение строки в пуле строк если строка создана new String()
  - это замедляет создание строки, увеличивает размер пула строк
  - это ускоряет работу при сравнении строк по ссылке
- Можно ли синхронизировать доступ к строке?
  - нет смысла это неизменный объект, поэтому синхронизации не требуется
- Как правильно сравнить значения строк двух различных объектов типа String и StringBuffer?
  - привести к одному типу и сравнить при помощи метода equals()
- Почему строка неизменная и финализированная в Java?
  - кэширование поддержка пула строк возможен благодаря immutable
  - безопасность в целях безопасности загрузка классов, сетевые соединения, доступе к базе
  - хэширование hashCode создается один раз при создании строки
  - многопоточность не требует синхронизации
- Напишите метод удаления данного символа из строки.
  - String.replace() для многократного удаления конкретного символа или последовательности
  - String.replaceAll() для многократного удаления с использованием regex

- Что такое рефлексия?
  - Reflection API — представлен в пакете java.lang.reflect, позволяет получить структуру класса
  - — из объекта во время RunTime исполнения приложения
  - определить — тип класса, модификаторы доступа,
  - — полях, методы, конструкторы, интерфейсы и суперклассы
  - создать — экземпляр класса, определенного Reflection
  - получить и задать — значение поля объекта
  - вызвать — метод объекта
  - создать — коллекцию из класса, определенного Reflections
- Пример. реализация **java02/reflections**
- Что произойдет со сборщиком мусора (GC), если во время выполнения метода finalize() некоторого объекта произойдет исключение?
  - ничего — сборщик мусора проигнорирует любое исключение, которое выбрасывает finalize()
- Что такое интернационализация, локализация?
  - интернационализация — метод разработки приложение легко адаптирует под разные языки
  - локализация — адаптация интерфейса приложения под конкретный язык
- Что такое Аннотации в Java?
  - это теги — добавляются к объявлениям полей, методов, конструкторов, классов, пакетов
  - — а также к локальным переменным и параметрам
  - связывают — элементы языка Java поля, классы и т.д. с новой информацией
  - — по сути это модификаторы, которые не меняя кода добавляют новый функционал
  - Пример. реализация
  - @Override — аннотация переопределенного метода
  - @Test — аннотация метода тестирования в JUnit
- Какие функции выполняет Аннотации?
  - дает — информацию для компилятора
  - — информацию инструментам для генерации другого кода, конфигурации
  - — информацию, используемую во время выполнения приложения
- Какие встроенные аннотации в Java вы знаете?
  - java.lang.annotation — @Documented, @Inherited, @Native, @Repeateble, @Retention, @Target
  - java.lang — @Deprecated, @FunctionalInterface, @Override, @SafeVarargs,
  - — @SupressWarnings
- Что делают аннотации @Retention, @Documented, @Target и @Inherited?
  - @Documented — используется при объявлении нового класса аннотаций
  - — включает аннотацию в JavaDoc класса при генерации
  - @Inherited — используется при объявлении нового класса аннотаций
  - — позволяет аннотации быть унаследованной
  - — действует только для классов, не действует для остального и интерфейсов
  - @Native — показывает, что поля могут быть использованы методами JNI
  - @Repeateble — используется при объявлении нового класса аннотаций
  - — показывает, что аннотация может быть использована несколько раз подряд
  - @Retention — используется при объявлении нового класса аннотаций
  - — показывает жизненный цикл аннотации в коде, в классе или runtime
  - @Target — используется при объявлении нового класса аннотаций
  - — задает объекты к которым может быть применена аннотация

- Что делают аннотации @Override, @Deprecated, @SafeVarargs и @SuppressWarnings?
  - @Deprecated — показывает что метод не рекомендован к использованию
  - @FunctionalInterface — показывает что интерфейс это functional interface
  - применяется к интерфейсам с одним абстрактным методом
  - не абстрактные методы интерфейса НЕ УЧИТЫВАЮТСЯ аннотацией
  - @Override — показывает что метод переопределяет метод предка
  - @SafeVarargs — подавляет предупреждения для методов с переменным числом
  - параметров неопределенного типа
  - применяется только для static методов, final member, конструкторов
  - @SupressWarnings — подавляет предупреждения заданные в аннотации
- Какой жизненный цикл аннотации можно указать с помощью @Retention?
  - @Retention(RetentionPolicy.SOURCE) — применяется только в исходнике
  - @Retention(RetentionPolicy.CLASS) — применяется в исходнике и после компиляции
  - @Retention(RetentionPolicy.RUNTIME) — применяется в исходнике, после компиляции и
  - во время исполнения
- К каким элементам можно применять аннотацию, как это указать?
  - @Target({ ElementType.TYPE, ElementType.FIELD, ElementType.METHOD,
  - ElementType.PARAMETER, ElementType.CONSTRUCTOR,
  - ElementType.LOCAL_VARIABLE})
  - PACKAGE — пакет
  - TYPE — класс
  - FIELD — поле
  - METHOD — метод
  - PARAMETER — параметр метода
  - CONSTRUCTOR — конструктор
  - LOCAL_VARIABLE — локальная переменная
- Как создать свою Аннотацию?
  - аннотация — создается как интерфейс **java04/annotation/**
- Атрибуты каких типов допустимы в аннотациях?
  - A primitive type
  - String
  - Class or an invocation of Class (§4.5)
  - An enum type
  - An annotation type
  - An array — type whose component type is one of the preceding types (§10.1).String
  - одномерный массив String[], Class[], enum[], Annotation[]
- Annotation Reflection Example **java04/annot**
  - аннотации — можно проверить только тестируя конкретный метод при помощи Reflection
- Процедура
  - аннотации — вытаскиваются в массив аннотаций
  - метод — вытаскивается из экземпляра класса и запускается с заданными параметрами
  - exception — ловится если метод генерирует Exception
  - сравнивается с массивом аннотаций
  - handler — вытаскивается из массива аннотаций
  - запускается на обработку и генерит нужное Exception

- Пример. реализация **java04/annot**
- ```
  private interface ExceptionHandler {
      void handleException(Throwable t);
  }

  @Target(ElementType.ANNOTATION_TYPE)  // for annotations only
  @Retention(RetentionPolicy.RUNTIME)
  private @interface Catch {
      Class<? extends ExceptionHandler> targetCatchHandler();

      Class<? extends Throwable> targetException() default Exception.class;
  }

  @Target(ElementType.METHOD)
  @Retention(RetentionPolicy.RUNTIME)
  private @interface CatchGroup {
      Catch[] catchers();  // method gets Catch[] array
  }

  private interface Caller {
      void callMethod() throws Throwable;              // any Exception
  }
  private static class MethodCaller {
      private static void callMethod(Caller instance) throws Exception {
          Method m = instance.getClass().getMethod("callMethod");
          Annotation[] as = m.getAnnotations();
          Catch[] catches = null;
          for (Annotation a : as) {
              catches = ((CatchGroup) a).catchers();
          }
          try {
              instance.callMethod();
          } catch (Throwable e) {
              Class<?> ec = e.getClass();
              if (catches == null) return;
              for (Catch c : catches) {
                  if (c.targetException().equals(ec)) {
                      ExceptionHandler h = c.targetCatchHandler().newInstance();
                      h.handleException(e);
                      break;
                  }
              }
          }
      }
      static class Bar implements ExceptionHandler {// класс генератор Exception
          @Override
          public void handleException(Throwable t) {
              System.out.println("NullPointerException: bar");
              System.out.println(t.getMessage());
          }
      }
      private static class Foo implements Caller {  // собственно класс и метод пользователя
          @Override
          @CatchGroup(catchers = {
          @Catch(targetCatchHandler = Bar.class, targetException = ArithmeticException.class),
          @Catch(targetCatchHandler = Bar.class, targetException = NullPointerException.class)
          })
          public void callMethod() throws Throwable {
              int a = 0;
              int b = 10;
              System.out.println(b/a);
          }
          public static void main(String[] args) throws Exception {
              Foo foo = new Foo();  // тестирование метода снаружи через Reflection и аннотации
              MethodCaller.callMethod(foo);  // метод запускается,затем ловятся его аннотации
          }
      }
  }
  ```
  ```

- Что такое JMX?
  - JMX Java Management Extension это фреймворк который позволяет встроить
  - приложение в существующие системы сетевого управления
  - определяет стандарт для написания JMX объектов MBean к которым имеет доступ JMX клиент
  - позволяет вызывать методы MBean объектов
  - задавать Listeners которые реагируют на события MBean объектов
  -
- Какие выгоды предлагает JMX?
  - простота реализации JMX основана на понятии сервера и объектов управления MBeans
  - сервер выступает как агент управления, объекты инструментарий
  - масштабируемость службы агентов JMX независимы, и могут быть встроены в агент JMX
  - гибкость JMX позволяет расширять концепцию в будущем, создавать любые
  - решения в будущем
  - управляемость JMX API представляет сервисы для работы в распределенной среде
- Пример. реализация JMX Local **java02/jmx/jmxc**
  - процедура запустить два Terminal
  - Terminal 1 запустить приложение check.cmd
  - Terminal 2 запустить jconsole check1.cmd
  - проверка JConsole выбрать процесс с портом 10999
  - MBean найти SimpleAgent
  - запустить sayHello(), изменить message, запустить sayHello()
- Пример. реализация JMX Remote **java02/jmx/jmxr**
  - процедура запустить Main() прямо из IDE
  - проверка JConsole
  - RemoteProcess вставить строку "service:jmx:rmi:///jndi/rmi://localhost:9999/server "
  - MBean найти SimpleAgent
  - запустить sayHello(), изменить message, запустить sayHello()
- Пример. реализация JMX Remote manual **java02/jmx/jmxr**
  - процедура встать на каталог project (parent of src)
  - создать три окна Terminal
  - Treminal1 src/java02/jmxr/check.cmd регистрация порта
  - Treminal2 src/java02/jmxr/check1.cmd компиляция и запуск агента
  - Treminal3 src/java02/jmxr/check2.cmd запуск JConsole
  - проверка JConsole
  - RemoteProcess вставить строку "service:jmx:rmi:///jndi/rmi://localhost:9999/server "
  - MBean найти SimpleAgent
  - запустить sayHello(), изменить message, запустить sayHello()
- Пример. реализация JMX Html **java02/jmx/jmxh**
  - процедура запустить Main() прямо из IDE
  - проверка FireFox
  - http://localhost:8000
  - MBean найти SimpleAgent >> name=hellothere
  - запустить sayHello(), изменить message, запустить sayHello()
- Что еще умеет JMX кроме дистанционного управления?
  - просмотр и изменение конфигурации приложения

- сбор и публикация статистики о работе приложение
- извещение о нештатных ситуациях или изменении состояния приложения
    - event notification — извещение о событиях изменения атрибута
    - monitor service — уведомления о зарегистрированных событиях
    - timer service — уведомления по таймеру
    - m-let service — создание и регистрация экземпляров MBean Server

- Что такое MBean?
  - MBean объекты — реализуют JMX интерфейс, который включает
  - величины которые доступны
  - методы которые могут быть вызваны
  - извещения которые могут быть посланы
  - конструкторы
- Какие типы MBeans существуют?
  - существует четыре типа MBeans standard, dynamic, open и model
  - Standard — интерфейс определяется набором методов
  - Dynamic — специализированный интерфейс, доступный во время исполнения
  - Open — это dynamic MBean, использует только основные типы данных
  - Model — это dynamic MBean, полностью конфигурируется и может показать свой интерфейс
  - Notification Model — базовая модель основана на Event для передачи данных между
  - агентами JMX
  - MBean Metadata Classes — классы, которые используются для создания MBean интерфейсов
- Пример. реализация Model MBean **java02/jmxmodel/jmxm**
  - применение — создать класс приложения
  - зарегистрировать сервер
  - создать массив параметров
  - зарегистрировать массив приложение, массив параметров
  - запустить сервер
  - запустить JConsole
- Что такое MBean Server?
  - сервер — это реестр объектов MBeans, которые используются для управления
  - предоставляет стандартный интерфейс к объектам MBean
  - регистрация — обязательно для любого MBean к которому нужен доступ
  - регистрировать MBean может другой MBean, агент или удаленное приложение
  - при регистрации объекту MBean назначается уникальное имя
- Какие механизмы обеспечивают безопасность в технологии Java?
  - возможности языка — проверка границ массива, запрет на преобразование типов
  - контроль доступа — к файлам или сетевым ресурсам
  - цифровая подпись — используется для аутентификации кода
- Назовите несколько видов проверок которые выполняет верификатор байт-кода Java?
  - JRE — проверяет byte code на следующее и не допускает к исполнению без проверки
  - код не подделывает указатели
  - код не нарушает ограничения доступа
  - код корректно использует типы объектов
  - JRE — на момент исполнения после проверок знает что
  - There are no operand stack overflows or underflows.
  - All local variable uses and stores are valid.
  - The arguments to all the Java Virtual Machine instructions are of valid types.

- Что вы знаете о "диспетчере защиты" в Java?
  - диспетчер защиты — обеспечивает защиту доступа к классу методами Reflections
  - политики беопасености — создание нового загрузчика классов
  - создание нового Security Manager
  - создание новой политики безопасности
  - доступ к полю класса с помощью Reflections
  - доступ к файлу
  - установление соединения через сокет
  - запуск задания на печать
  - доступ к буферу обмена
  - обращение к окну верхнего уровня
  -
- Что такое JAAS?
  - JAAS — Java Authentification and Authorization Service Java API
  - обеспечивает авторизацию пользователя
  - основная задача отделить авторизацию от основной программы чтобы управлять
  - авторизацией и приложением независимо
  -
- Что такое Рефакторинг?
  - рефакторинг — процесс изменения кода, облегчающий его понимание и сопровождение
  - не затрагивает поведение программы
- Что такое JAR и как им пользоваться?
  - JAR — это Java ARchive архив с файлами содержит
  - файлы с расширением *.class байт кода Java
  - каталог META-INF
  - с файлом MANIFEST.MF, который может расширить функционал JAR
- material — смотреть в каталоге materials/jar — **materials/jar**
  - полная информация по JAR файлам — в документе jar.doc
  - проект с примерами использования — в проекте jar_project
  -
- Пример. реализация — **jar_project**
- f

JavaCollrctions Framework

- Что такое Коллекция?
  - это хранилища данных созданные для эффективного доступа к данным
  - построены на базе абстрактных структур данных
  - свойства добавление элемента
  - удаление элемента
  - изменение элемента
- Назовите основные интерфейсы коллекций и их имплементации.
- Collection интерфейс базовый для всех, кроме Map, коллекций Java
  - `interface Collection extends Iterable<E>`
  - `interface Set, List, Queue extends Collection`
  - `inteface Map`
- Map интерфейс базовый для всех Map
  - `class HashTable`
  - `>> abstract Dictionary`
  - `<> Map`
  - `class LinkedHashMap HashMap`
  - `>> HashMap >> abstract AbstractMap`
  - `<> Map <> Map`
  - `class WeakHashMap`
  - `>> abstract AbstractMap`
  - `<> Map`
  - `class TreeMap`
  - `>> abstract AbstractMap`
  - `<> NavigableMap NavigableMap`
  - `<> SortedMap SortedMap`
  - `<> Map`
  - `class abstract AbstractMap`
  - `<> Map`
- List интерфейс базовый для всех List, Deque, Queue
  - `class Stack >> Vector`
  - `>> AbstractList`
  - `<> List`
  - `class ArrayList`
  - `>> AbstractList AbstractList`
  - `<> List >> AbstractCollection AbstractCollection`
  - `<> List <> Collection Collection`
  - `class LinkedList <> Iterable`
  - `>> AbstractSequentialList AbstractSequentialList`
  - `<> List >> AbstractList`
  - `<> Deque`
  - `class AbstractList`
  - `>> AbstractCollection`
  - `<> List`
  - `interface Deque`
  - `<> Queue Queue`
  - `<> Collection`

- Set
  - class HashSet
  - >> AbstractSet
  - <> Set
  - class LinkedHashSet
  - >> HashSet
  - <> Set
  - class TreeSet
  - >> AbstractSet
  - <> NavigableSet NavigableSet
  - <> SortedSet SortedSet
  - <Set> Set
  - <> Collection
  - class AbstractSet
  - >> AbstractCollection AbstractCollection
  - <> Set <> Collection Collection
  - <> Iterable
- Описание коллекций
  - List
  - Vector синхронный List в одном потоке работает медленно, рекомендуется ArrayList
  - ArrayList быстрый произвольный доступ O(1)
  - LinkedList последовательный доступ O(n), добавление и удаление с конца за O(1)
  - очень удобен для реализации стека и очереди, где движение на концах коллекции
  - Set интерфейс
  - TreeSet сортированный Set
  - HashSet работает по хэшкодам, самый быстрый вариант
  - LinkedHashSet работает по хэшкодам, медленнее HashSet, хранит в порядке добавления
  -
  - Queue интерфейс очереди
  - LinkedList реализация списка, очереди, стека
  - PriorityQueue реализация приоритетной очереди
  -
  - Map интерфейс Map пара ключ, значение
  - HashTable синхронная коллекция, устаревшая и медленно работает в одном потоке
  - HashMap поиск элементов по хэш ключу, наиболее быстрая реализация
  - LinkedHashMap поиск элементов по хэш ключу, медленнее HashMap, в порядок добавления
  - TreeMap сортированная Map
  - Синхронизация
  - методы Collection можно методом Collections.synchronizedList(Map)
  - Set можно методом Collections.synchronizedSet(List)
  - Map можно методом Collections.synchronizedMap(List)
  - коллекции CopyOnWriteArrayList синхронный ArrayList
  - CopyOnWriteArraySet синхронный Set на базе CopyOnWriteArrayList
  - ConcurentHashMap синхронная Map похожа на HashTable
  - ConcurrentLinkedDeque синхронная очередь Deque
  - ConcurrentSkipListMap синхронная sorted NavigableMap
  - ConcurrentSkipListSet синхронный sorted NavigableSet на базе
  - ConcurrentSkipListMap

- Чем отличается ArrayList от LinkedList? В каких случаях лучше использовать первый, а в каких второй?
  - ArrayList произвольный доступ, строится на базе массивов
  - лучше использовать, когда нужен быстрый произвольный доступ за время O(1)
  - медленно удаляет, добавляет элементы из середины коллекции
  - LinkedList последовательный доступ
  - лучше использовать, когда нужен быстрый последовательный доступ
  - удаление, вставка в середине коллекции за время O(n)
  - строится на базе связных списков
- Производительность

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

- Чем отличается HashMap от Hashtable?
  - HashTable является синхронной коллекцией, работает медленнее, чем HashMap
  - устарела, альтернатива ConcurrentHashMap
- Чем отличается ArrayList от Vector?
  - Vector является синхронизированной коллекцией, поэтому низкая производительность
  - устарел, альтернатива ArrayList
- Как сравниваются елементы коллекций?
  - метод equals() сравнивает объекты
  - hashCode() выдает уникальный hashCode()
  - ОДИНАКОВЫЕ элементы имеют ОДИНАКОВЫЙ hashCode
  - РАЗНЫЕ элементы могут иметь ЛЮБОЙ hashCode
- Расположите в виде иерархии следующие интерфейсы: List, Set, Map, SortedSet, SortedMap, Collection,Iterable, Iterator, NavigableSet, NavigableMap.
  - Iterable >> Collection >> List, Set >> SortedSet >> NavigableSet
  - Map >> SortedMap >> NavigableMap
  - Iterator
- Почему Map - это не Collection, в то время как List и Set являются Collection?
  - Collection это контейнер для одиночныхэлементов
  - Map это контейнер для элементов пара ключ, значение
- Дайте определение понятию "iterator".
  - iterator объект на базе интерфейса Iterator, позволяет перебирать все элементы коллекции
  - Collection.iterator() возвращает объект интерфейса Iterator
  - используется в конструкции foreach по очереди
  - реализует три метода hasNext(), next() и remove()
  - позволяет удалять элементы из оригинальной коллекции во время перебора

- Что вы знаете об интерфейсе Iterable?
  - Iterable интерфейс, который наследует Collection и следовательно все коллекции
  - метод iterator() возвращает итератор для данного объекта коллекции
- Как одной строчкой преобразовать HashSet в ArrayList?
  - использовать конструктор new ArrayList(Collection c)
  -
- Как одной строчкой преобразовать ArrayList в HashSet?
  - использовать конструктор new HashSet(Collection c)
- Как перебрать все ключи Map учитывая, что Map - это не Iterable?
  - получить Map.keyset() объект Set<K> перебрать Set итератором или forEach
- Пример. реализация
  - ```
    Set<String> k = tMap.keySet();
    Iterator<String> ik = k.iterator();
    for (String s1 : k) {
    }
    ```
- Как перебрать все значения Map учитывая, что Map - это не Iterable?
- Пример. реализация
  - ```
    Collection<Integer> v = tMap.values();
    Iterator<Integer> iv = v.iterator();
    for (Integer integer : v) {
    }
    ```
  -
- Как перебрать все пары ключ-значение в Map учитывая, что Map - это не Iterable?
  - получить EntrySet<K,V> и перебрать его итератором или forEach
- Пример. реализация
  - ```
    Set<Map.Entry<String,Integer>> e = tMap.entrySet();
    Iterator<Map.Entry<String,Integer>> ie = e.iterator();
    for (Map.Entry<String, Integer> entry : e) {

    }
    ```
- В чем проявляется "сортированность" SortedMap, кроме того, что toString() выводит все по порядку?
  - sortedMap выводит в сортированном порядке
  - toString() пары значений, упорядоченные по ключу
  - keyset() ключи, упорядоченные по ключу
  - values() значения, упорядоченные по ключу
- Пример. реализация
  - ```
    System.out.println("toString     : "+sMap.toString());

    System.out.print("Iterator  key:");
    for (Integer key : sMap.keySet()) {
        System.out.print(key+"="+sMap.get(key)+" ");
    }

    System.out.println();
    System.out.print("Iterator  val:");
    for (Integer value : sMap.values()) {
        System.out.print("v="+value+" ");
    }
    System.out.println();
    ```
  -

- Как одним вызовом копировать элементы из любой Collection в массив?
  - toArray() метод преобразования в коллекции в массив
  - обязательно аргумент массив нулевой длины,в последних версиях работает быстрее
  - Пример. реализация
  - ```
    Integer[] ints =sMap.keySet().toArray(new Integer[0]);
    System.out.println("Collection to Array:");
    System.out.println(Arrays.toString(ints));
    ```
- ВНИМАНИЕ. Приведение типа делать через АРГУМЕНТ, использовать массив НУЛЕВОЙ длины
  - в последних версиях Java это работает быстрее, чем массив заданной длины
- Реализуйте симметрическую разность двух коллекций используя методы Collection (addAll(), removeAll(), retainAll()).
  - симметрическая разность это получение объектов, не принадлежащих ни одной коллекции
  - retainAll() пересечение коллекций
  - addAll() объединение коллекций
  - removeAll удаление одной коллекции из другой
  - процедура найти пересечение retainAll(), найти объединение addAll()
  - удалить пересечение из объединения, получится symmetric разность
- Пример. реализации
- ```
  // symmetric
          List<String> list = new ArrayList<>();
          List<String> listA = new ArrayList<>();
          List<String> listB = new ArrayList<>();
          Random rnd = new Random();
          for (int i = 20; i < 40; i++) {
              list.add("listA" + i);
          }
          listA.addAll(list.subList(1,10));
          listB.addAll(list.subList(5,14));

          System.out.println("listA:"+listA);
          System.out.println("listB:"+listB);

          System.out.println("Symmetric");
          List<String> listI = new ArrayList<>(listA);
          listI.retainAll(listB);                          // intersection
          List<String>  listS = new ArrayList<>(listA);
          listS.addAll(listB);                             // both
          listS.removeAll(listI);                          // symmetric difference

          System.out.println("symm :"+listS);
  ```
- Сравните Enumeration и Iterator.
  - Enumeration интерфейс перебора элементов
  - hasNext() , nextElement()
  - Iterator интерфейс перебора элементов
  - hasNext(), next(), remove()
  - отличия Enumeration похож на Iterator, но нет метода remove()
  - рекомендуется вместо Enumeration использовать Iterator
- Как между собой связаны Iterable и Iterator?
  - Iterable интерфейс который имеет один метод iterator()
  - Iterable.iterator() возвращает объект Iterator для обхода коллекции

- Как между собой связаны Iterable, Iterator и "for-each " введенный в Java 5?
    - Iterable — коллекции которая реализует Iterable использоваться в forEach напрямую
    - — напрямую, если коллекция
- Пример. реализация
- ```
  System.out.println("Iterable: ");
  MList<String> mList = new MList<>();
  mList.addAll(list.subList(0,10));

  System.out.println(mList);
  System.out.println("Iterable elements:");
  for (String s1 : mList) {
      System.out.print(s1+" ");
  }
  System.out.println();
  ```
-
- ```
  private static class MList<T> implements Iterable<T> {
      private List<T> list;
      public MList() {
          this.list = new ArrayList<>();
      }
      public void addAll(List<T> subList) {
          if (list == null || subList == null) throw new IllegalArgumentException();
          list.addAll(subList);
      }
      @Override
      public Iterator<T> iterator() {
          return new Iterator<T>() {
              private int index = 0;
              @Override
              public boolean hasNext() {
                  return !(list == null || list.isEmpty() || index >= list.size());
              }
              @Override
              public T next() {
                  if (!hasNext()) throw new IndexOutOfBoundsException();
                  T value = list.get(index);
                  index++;
                  return value;
              }
          };
      }
      @Override
      public String toString() {
          if (list == null) return "";
          return list.toString();
      }
  }
  ```
-
- Сравните Iterator и ListIterator.
    - Iterator — три метода обхода коллекции, одно направление от начала в конец
    - — hasNext(), next(), remove()
    - ListIterator — двусторонний итератор с возможностью изменения элементов
    - — hasNext(), next(), nextIndex()
    - — hasPrevious, previousIndex(), previous
    - — remove(), set()

- Что произойдет, если я вызову Iterator.next() не "спросив" Iterator.hasNext()?
    - если Iterator пустой, вылетит исключение NoSuchElementException
- Что произойдет, если я вызову Iterator.next() перед этим 10 раз вызвав Iterator.hasNext()? Я пропущу 9 элементов?
    - нет так, как перемещение происходит по методу next()
    -
- Если у меня есть коллекция и порожденный итератор, изменится ли коллекция, если я вызову iterator.remove()?
    - iterator.remove() вызовет NoSuchElementException
    - iterator.next() iterator.remove() удалит элемент из коллекции
- Если у меня есть коллекция и порожденный итератор, изменится ли итератор, если я вызову collection.remove(..)?
    - сработает исключение ConcurrentModification Exception
- ВНИМАНИЕ. Удалять it.next() it.remove() МОЖНО, удалять list.remove() it.next НЕЛЬЗЯ
-
- Пример. реализация **java_c01/Main**

```
System.out.println("\nCollection Iterator:");
System.out.println("Remove from it >> it.next >> it.remove():");
List<String> listE = new ArrayList<>(listA);
it = listE.iterator();
System.out.println(listE);
try {
    it.next();
    it.remove();
    System.out.println(listE);
} catch (ConcurrentModificationException ex) {
    System.out.println(ex.getMessage()+" "+listE);
}

System.out.println("Remove from it >> listF.remove it.next():");
List<String> listF = new ArrayList<>(listA);
it = listF.iterator();
System.out.println(listF);
try {
    listF.remove(0); // remove 2nd elemebt
    it.next();
} catch (ConcurrentModificationException ex) {
    System.out.println(ex);
    System.out.println(listF);
}
```

- Зачем добавили ArrayList, если уже был Vector?
    - Vector является потокобезопасным, поэтому работает медленнее в одном потоке
    - ArrayList добавили как асинхронный но быстрый вариант
    -
- В реализации класса ArrayList есть следующие поля: Object[] elementData, int size. Объясните, зачем хранить отдельно size, если всегда можно взять elementData.length?
    - elementData[] это массив, в котором хранятся элементы списка
    - массив удваивается или уменьшается в двое при изменении размера
    - size реальное число заполненных элементов в массиве elementData[]
- ВНИМАНИЕ. При добавлении одного элемента минимальный размер elementData[] = 10

- LinkedList - это односвязный, двусвязный или четырехсвязный список?
    - двухсвязный каждый Node хранит ссылку на prev и next
- Пример. реализация
- ```
  private static class Node<E> {
      E item;
      Node<E> next;
      Node<E> prev;

      Node(Node<E> prev, E element, Node<E> next) {
          this.item = element;
          this.next = next;
          this.prev = prev;
      }
  }
  ```

- Какое худшее время работы метода contain() для элемента, который есть в LinkedList (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - O(N)
- Какое худшее время работы метода contain() для элемента, который есть в ArrayList (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - O(N)
- Какое худшее время работы метода add() для LinkedList (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - O(1) в начало и конец списка
    - O(N) в сортированный список add(value) или середину списка add(index,value)
- Какое худшее время работы метода add() для ArrayList (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - O(N)
- Сколько выделяется элементов в памяти при вызове ArrayList.add()?
    - если список пустой 10 элементв
    - если список полный в 1.5 раза больше
- Пример. реализация
  ```
  int newCapacity = oldCapacity + (oldCapacity >> 1);
  ```

- Сколько выделяется элементов в памяти при вызове LinkedList.add()?
    - один элемент создается экземпляр класса Node
- Оцените количество памяти на хранение одного примитива типа byte в LinkedList?
    - 32 битные система 8+4*3+8 =28 байт кратно 8 = 32 байта
    - 64 битные система 8+8*3+8 = 40 байт
-
- ```
  private static class Node<E> {    //  на объект 8 байт
      E item;                       //  ссылка 4(8) байта + 8 байт Byte объект
      Node<E> next;                 //  ссылка 4(8) байта
      Node<E> prev;                 //  ссылка 4(8) байта

  }
  ```
- Оцените количество памяти на хранение одного примитива типа byte в ArrayList?
    - байт является элементом массива Object[] поэтому 24 байта
    - 32bit(64) 8+4(8)+8 = 24 bytes

- Я добавляю элемент в середину List-а: list.add(list.size()/2, newElem). Для кого эта операция медленнее - для ArrayList или для LinkedList?
    - LinkedList поиск позиции O(N/2) и вставка O(1)
    - ArrayList копирование O(N/2) и вставка O(1) плюс расширение O(N) если совпало
    - в среднем этот быстрее
- Как перебрать элементы LinkedList в обратном порядке, не используя медленный get(index)?
    - использовать iterator ListIterator использует внутренний класс ListItr
    - iterator DescendingIterator использует внутренний класс ListItr
- Как одним вызовом из List получить List со всеми элементами, кроме первых и последних 3-х?
    - использовать метод `sublist(3,list.size()-3)`
- Могут ли у разных объектов в памяти (ref0 != ref1) быть ref0.hashCode() == ref1.hashCode()?
    - одинаковый hashCode у разных объектов да может
- Могут ли у разных объектов в памяти (ref0 != ref1) быть ref0.equals(ref1) == true?
    - одинаковое значение у разных объектов да может
    - обязательно должен быть переопределен метод equals()
    - так как базовый сравнивает по адресу ссылки
- Могут ли у разных ссылок на один объект в памяти (ref0 == ref1) быть ref0.equals(ref1) == false?
    - нет один и тот же объект должен быть равен самому себе свойство рефлективности
- Есть класс Point{int x, y;}. Почему хэш-код в виде 31 * x + y предпочтительнее чем x + y?
    - множитель дает смещение по значению и гарантированно разные хэш коды для y < 31
- Если у класса Point{int x, y;} "правильно " реализовать метод equals (return ref0.x == ref1.x && ref0.y == ref1.y), но сделать хэш-код в виде int hashCode() {return x;}, то будут ли корректно такие точки помещаться и извлекаться из HashSet?
    - да только HashMap для значений с одним x и разными y превратится просто в HashSet
    - приведет к более медленному доступу
- equals() порождает отношение эквивалентности. Какими из свойств обладает такое отношение: коммутативность, симметричность, рефлексивность, дистрибутивность, ассоциативность, транзитивность?
    - рефлексивность равенство самому себе x==x
    - симметричность равенство друг другу x == y если y == x
    - транзитивность равенство по цепочке x==y и y==z то x==z
    - повторяемость равенство постоянно
    - неравенство null x!= null && y!=null
- Можно ли так реализовать equals(Object that) {return this.hashCode() == that.hashCode()}?
    - нельзя так как hashCode может быть одинаковым у разных объектов
    - говнокод допустимо для объектов Object если метод equals не переопределен
- В equals требуется проверять, что аргумент (equals(Object that)) такого же типа как и сам объект. В чем
- разница между this.getClass() == that.getClass() и that instanceof MyClass?
    - первое проверяет класс на точное совпадение
    - второе может дать true для потомков или интерфейса
- ВНИМАНИЕ. В equals() правильнее использовать ТОЧНОЕ совпадение класса
- Можно ли реализовать метод equals класса MyClass вот так: class MyClass {public boolean equals(MyClass that) {return this == that;}}?
    - можно по сути это тоже самое сравнение по ссылке как у Object()

- Будет ли работать HashMap, если все ключи будут возвращать int hashCode() {return 42;}?
  - да — вопрос тот же самый, но работать будет медленно, так как получится список
  - ключей вместо HashMap
- Зачем добавили HashMap, если уже был Hashtable?
  - HashTable — синхронная реализация Map в одном потоке работает медленно
  - не может хранить key = null
  - не может хранить value = null
- Согласно Кнуту и Кормену существует две основных реализации хэш-таблицы: на основе открытой адресацией и на основе метода цепочек. Как реализована HashMap? Почему так сделали (по вашему мнению)? В чем минусы и плюсы каждого подхода?
  - хэш — таблица строится на базе hashCode и обработке коллизий
  - обработка коллизий и есть методы с открытой адресацией и методе цепочек
- Открытая адресация
  - принцип — используется массив table[] который хранит key
  - процедура — добавления
  - получить стартовый индекс index = hashCode
  - проверить пустой ли элемент table[index], если да, то записать key
  - если занято, наращивать индекс до первого свободного элемента
  - процедура — получения
  - при получении первого элемента по адресу table[hashCode] проверяется key
  - извлечь ключ key = table[hashCode] — сравнить с искомым, если есть выход
  - получить следующий key= table[hashCode+1] — сравнить с искомым, если есть выход
  - повторить с индексом следующего ключа пока не достигнут конец table
  - вывод — по сути идет перебор всех элементов подряд со стартовой позиции hashCode
  - при заполнении таблицы на 70% работает очень медленно
- Цепочки
  - процедура — добавления
  - получить стартовый индекс index = hashCode
  - проверить пустой ли элемент table[index], если да, то записать Node(key,null)
  - если занято, получить Node = table[index] и проверить Node.next==null
  - если нет, идти по цепочке пока Node.next==null
  - когда найдено записать Node.next=new Node(key,null)
  - процедура — получения
  - при получении первого элемента по адресу table[hashCode] считывается Node
  - получить Node = table[hashCode] — сравнить Node.key если есть выход
  - проверить Node.next == null если да, то выход
  - получить следующий Node = Node.next — сравнить Node.key если есть выход
  - вывод — работает более менее оптимально даже для загруженных таблиц
- Сравнение
  - открытая адресация — работает крайне медленно при заполнении таблицы
  - удаление элемента проблематично при работе с массивами
  - метод цепочек — производительность снижается незначительно при заполнении
  - собственно удаление элемента списка выполняется за O(1)

- Пример. реализация HashMap

```
static final int TREEIFY_THRESHOLD = 8; // гарантированный размер бакета 8
```

-
```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);         // добавление в пустой элемент
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {                 // добавление в конец списка
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;                                  // key найден в Node.key
                p = e;
            }
        }
        if (e != null) {                                    // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

- Сколько переходов по ссылкам происходит, когда вы делаете HashMap.get(key) по ключу, который есть в таблице?
  - максимум 8 определяется значением TREEIFY_THRESHOLD
- Сколько создается новых объектов, когда вы добавляете новый элемент в HashMap?
  - создается один объект Node
- Как работает HashMap при попытке сохранить в нее два элемента по ключам с одинаковым hashCode, но для которых equals == false?
  - создается новый объект Node(key,null) который добавляется по ссылке Node.next
  - последнего в цепочке элемента в списке по адресу table[hashCode]
  - если количество элементов в узле превышает 8, все узлы в бакете заменяются на RBBST
  - далее чтение и запись в данный бакет происходит по алгоритму RedBlack BST
- HashMap может выродиться в список даже для ключей с разным hashCode. Как это возможно?
  - это возможно если метод hashCode() будет возвращать один и тот же hashCode

- Какое худшее время работы метода get(key) для ключа, которого нет в таблице (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - worst case O(N) для вырожденного набора HashMap в HashSet
- Какое худшее время работы метода get(key) для ключа, который есть в таблице (O(1), O(log(N)), O(N), O(N*log(N)), O(N*N))?
    - worst case O(N) для вырожденного набора HashMap в HashSet просто он последний
- Объясните смысл параметров в конструкторе HashMap(int initialCapacity, float loadFactor).
    - initialCapacity исходный размер таблицы, по умолчанию для HashMap равно 16
    - loadFactor по умолчанию для HashMap равно 0.75 или threshold = 12 для size = 16
- В чем разница между HashMap и IdentityHashMap? Для чего нужна IdentityHashMap? Как может быть полезна для реализации сериализации или клонирования?
    - IdentityHashMap не использует equals() и hashCode()
    - вместо equals() сравнение ссылок
    - вместо hashCode() native System.IdentityHashCode()
    - производительность выше за счет equals() сравнения ссылок
    - за счет нативного hashCode()
    - допускает одинаковые ключи так как сравнение идет по ссылкам
    - используется для Serialization/Deserialization так как могут быть объекты с equals()==true
- ВНИМАНИЕ. При Serialization разные объекты для которых equals()==true должны рассматриваться как
- разные, поэтому System.IdentityHashMap удобна для Serialization
- В чем разница между HashMap и WeakHashMap? Для чего нужна WeakHashMap?
    - WeakHashMap использует в качестве ключей WeakReferences
    - not synchronized, но может быть с помощью Collections.synchronizedMap()
- ВНИМАНИЕ. Объекты value НЕ ДОЛЖНЫ иметь ссылок на ключи WeakHashMap так как
- это не позволит удалить ключи и поломает схему WeaKReference
    - SoftReference ссылка на объект говорит о том, что возможно GC удалит объект
    - используется для крупных объектов, которые JVM выгружает
    - при нехватке памяти
    - позволяет ВОССТАНОВИТЬ Strong ссылку на объект при необходимости
    - WeakReference ссылка на объект говорит о том, что GC удалит объект
    - используется для автоматического удаления объектов из WeakHashMap
    - когда они больше не используются
    - используется для хранения дополнительной информации на объекты
    - ключи, которая удаляется автоматом вместе с
    - неиспользуемым объектом ключом
    - используется для ссылок на потоки, которые могут быть удалены ОС
    - позволяет ВОССТАНОВИТЬ Strong ссылку на объект при необходимости
    - PhantomReference ссылка на объект говорит о том, что GC удалит после нескольких запусков
    - используется совместно с ReferenceQueue и только с ней
    - PhantomReference позволяет узнать что объект точно попал под удаление
    - ДОБАВЛЯЕТСЯ в очередь только ПОСЛЕ вызова метода finalize()
    - СПАСТИ объект невозможно, можно очистить связанные с ним ресурсы
    - ОКОНЧАТЕЛЬНО он будет удален только после удаления PhantomReference
- ВНИМАНИЕ. Использование PhantomReference непонятно,скорее всего для очистки критических ресурсов

- Разница между HashMap и WeakHashMap **java_c01/weakHashMap**
  - работа GC — HashMap — GC не пытается удалять Entry<K,V>, даже если на объект K нет ссылок нигде в программе
  - WeakHashMap — Entry<K,V> в WeakHashMap будет удален, если на объект K нет strong references
  - ссылки — HashMap — Key имеет тип ссылки Strong Reference
  - WeakHashMap — Key имеет тип ссылки WeakReference поэтому будет удален GC
  - size — HashMap — размер не меняется если нет вызовов add(), remove()
  - WeakHashMap — размер меняется автоматом в результате работы GC
  - clone() — HashMap — возвращает shallow копию HashMap
  - WeakHashMap — НЕ РЕАЛИЗУЕТ интерфейс Clonable и не имеет clone()
  - Serialize — HashMap — реализует Serializable интерфейс
  - WeakHashMap — НЕ РЕАЛИЗУЕТ Serializable интерфейс
- Реализация User WeakHashMap
  - Entry<K,V> — надо сделать его на базе WeakReference и этого достаточно чтобы получить реализацию WeakHashMap
- Пример. реализация WeakHashMap
- ВНИМАНИЕ. Это ИЛЛЮСТРАЦИЯ реализации WeakHashMap
-
```
public class WeakHashMapUser<K,V> implements Map<K,V> {

private static class Entry<K,V> extends WeakReference<K>
  implements Map.Entry<K,V> {
    private V value;
    private final int hash;
    private Entry<K,V> next;
          <... code ...>
}
public V get(Object key) {
    int hash = getHash(key);
    Entry<K,V> e = getChain(hash);
    while (e != null) {
        K eKey= e.get();
        if (e.hash == hash && (key == eKey || key.equals(eKey)))
            return e.value;
        e = e.next;
    }
    return null;
}
```

- Пример. реализация WeakHashMap демонстрация

```
// Created HashMap and WeakHashMap objects
Map hashmapObject = new HashMap();
Map weakhashmapObject = new WeakHashMap();
String hashmapKey = new String("hashmapkey");
String weakhashmapKey = new String("weakhashmapkey");
String hashmapValue = "hashmapvalue";
String weakhashmapValue = "weakhashmapvalue";
hashmapObject.put(hashmapKey, hashmapValue);
weakhashmapObject.put(weakhashmapKey, weakhashmapValue);
System.out.println("Map before GC :" + hashmapObject+ ":" + weakhashmapObject);
hashmapKey = null;
weakhashmapKey = null;
System.gc();
System.out.println("Map after GC :" + hashmapObject + ":" + weakhashmapObject);
```

- В WeakHashMap используются WeakReferences. А почему бы не создать SoftHashMap на SoftReferences?
    - WeakHashMap — создан для работы с данными которые можно безболезненно удалить
    - при этом они удаляются мгновенно, если объект не используется
    - SoftHashMap — создан для работы с данными которые выгружаются только под давлением
    - OutOfMemory Exception
    - Есть реализации у Oracle, Google, Apache
- Альтернативные решения
    - ConcurrentReferenceHashMap — реализована в Spring Framework 5
    - Maps.SoftHashMap — реализована в Oracle Fusion MiddleWare
    - Apache SoftHashMap — реализована в Apache Shiro code в репозитории github
    - Apache ReferenceMap — реализована в Apache Commons code в репозитории github
    -
- Пример. реализация SoftHashMap — **java_c01/softHashMap**
- Пример. реализация SoftHashMap с StrongReferences — **java_c01/softHashMap**
- В WeakHashMap используются WeakReferences. А почему бы не создать PhantomHashMap на PhantomReferences?
    - невозможно — создать PahntomHashMap так как PhantomReference get() возвращает null
    - то есть нет объектов, которые могли бы работать key или value
    - PhantomReference объекты помечены на удаление, а сами ссылки на
    - объекты помещаются в ReferenceQueue после вызова finalize() удаляемых
    - объектов но еще не удаленных
    - невозможно — восстановить объекты по PhantomReferences
    - невозможно получить доступ к этим объектам
    - единственное — что возможно это освободить ресурсы связанные с PhantomReference
    - объектами
- Сделайте HashSet из HashMap (используйте только множество ключей, но не множество значений).
- Пример. реализация
- ```
  HashSet<String> hashSet = new HashSet<>(hMap.keySet());
  System.out.println("keyset() : "+hashSet);
  ```
-
- Сделайте HashMap из HashSet (HashSet<Map.Entry<K, V>>).
- Пример. реализация
- ```
  HashSet<Map.Entry<String,String>> entryHashSet = new HashSet<>(hMap.entrySet());
  System.out.println("entries(): "+entryHashSet);
  ```
- Сравните интерфейсы java.util.Queue и java.util.Deque.
    - Queue — интерфейс для работы с очередями FIFO
    - методы — add(), offer() — добавляет элемент в конец очереди
    - remove(),poll() — удаляет элемент из начала очереди
    - peek(),element() — показать первый элемент без удаления
    - Deque — двусторонняя очередь, позволяет добавлять и удалять элементы с обоих концов
    - методы — add, offer, remove, peek, element и версии last, first каждого метода
- Сравнение
    - Queue — работает по принципу FIFO и это неизменно
    - Deque — может работать по FIFO и LIFO
    - обе очереди — не переопределяют методы equals() и hashCode()

- Кто кого расширяет: Queue расширяет Deque, или Deque расширяет Queue?
  - Deque расширяет Queue Deque << Queue << Collection << Iterable
- Почему LinkedList реализует и List, и Deque?
  - LinkedList связный список является идеальным для построения Deque
- В чем разница между классами java.util.Arrays и java.lang.reflect.Array?
  - Arrays это набор статических методов работы с массивами
  - Array класс работы с массивами в Reflections Framework
  - Reflections позволяет извлекать информацию о классах из объектов
  - имеет private constructor, объект создается методом Array.newInstance()
- Пример. реализация
- ```
  System.out.println("\nArrays int[]: "+Arrays.toString(new int[10]));
  int[] array = (int[]) Array.newInstance(int.class,10);
  array[1] = 1;
  array[9]= 18;
  System.out.println("Array int[]: "+Arrays.toString(array));
  ```
- В чем разница между классами java.util.Collection и java.util.Collections?
  - Collection это интерфейс, который является предком всех коллекций и расширяет Iterable
  - Collections это класс статических методов
- Напишите НЕмногопоточную программу, которая заставляет коллекцию выбросить ConcurrentModificationException.
- ```
  List<String> list = new ArrayList<>();
  Iterator<String> it = list.iterator();
  list.add("1");
  it.next(); // Concurrent
  ```
- ВНИМАНИЕ. ConcurrentModification срабатывает ТОЛЬКО на итераторах
- ```
  System.out.println(id + " " + list);  // используется Iterator в StringBuilder.append()
  ```
- Что такое fail safe поведение?
  - fail safe строго говоря такого опредления нет, есть weakly consistent и касается итераторов
  - weakly consistent итераторы является thread safe но необязательно освобождает коллекцию
  - во время итераций и может не отражать обновлений во время итераций
- Что такое "fail-fast поведение"?
  - fail-fast iterator который не препятствует изменению коллекции во время итерации
  - но выдает ConcurentModificationException если обнаруживает ситуацию
  - применение ConcurretnModificationException НЕ ДАЕТ гарантии безопасного доступа к
  - коллекции и должно использоваться как причина чтобы изменить код
- Что такое enum? **java_c01/enums/Main02Enum**
  - class static final class по умолчанию, создает константы на этапе компиляции
  - constructor private по умолчанию, поэтому невозможно сделать new enum()
  - если поля создаются с параметрами, ОБЯЗАТЕЛЬНО должен быть конструктор
  - redefine методов возможен при объявлении параметров enum, смотреть evalue.KOTLIN
  - private field access возможен только через геттеры и сеттеры при переопределении методов
  - аргументы enum НАСЛЕДУЮТ класс enum как СТАТИЧЕСКИЕ внутренние классы
- Enum наследование и работа с Generic **java_c01/enums/Main02Enum3**
  - enum<T> НЕ РАБОТАЕТ Generic, конструкция enum<T> НЕ ДОПУСТИМА
  - Enum<T> неявно final class поэтому НЕ МОЖЕТ БЫТЬ НАСЛЕДОВАН другим классом
  - T может быть только enum типа
  - instance можно получить только Enum.valueOf(enum.A, enum.B)
  - можно получить только Enum.valueAll(enum.class)
  ```

- Enum с Generic можно либо использовать в качестве **java_c01/enums/Main02Enum4**
- Реализация Generic с enum
  - стандартная конструкция с Generic с Class
  - `public class B<T> implements A<T>`
  - стандартная конструкция с Generic с enum НЕ РАБОТАЕТ
  - `public enum B<T> implements A<T>`
- Варианты Generic которые работают
  - `public class enum B implements A<T>`
  - `public class B<T> implements A<T>`
- Пример. реализация
  - в наследовании интерфейса
  - ```
    private enum B2 implements A<Integer> {
        B21,B22,B23;
        @Override
        public Integer getValue() {
            return null;
        }
    }
    ```
  - подменой обычным классом
  - ```
    private static class B4<T> implements A<T> {
        private final T s;
        public static final B4<String> A1 = new B4<>("S");
        public static final B4<Integer> A2 = new B4<>(12);
        B4(T s) {
            this.s = s;
        }
        @Override
        public T getValue() {
            return null;
        }
    }
    ```
- Пример. реализация Generic
- ```
  public static void main(String[] args) {
          System.out.println("Enum B :");

          System.out.println("Enum   :" + B.A1 + " " + B.A2);
          EnumSet<B> enumSet = EnumSet.of(B.A1, B.A2);
          System.out.println("EnumSet:" + enumSet);

          System.out.println("Class B:");
          System.out.println("Class B:" + B3.A1 + " " + B3.A2);

          System.out.println("\nGeneric:");
          A2 a2 = new A2("s1", 12);
          a2.setValue(a2);
          System.out.println(a2);

      }
  ```
  ```

- Абстрактный класс наследование и Generic Bounded **java_c01/enums/Main03Bounded**
  - bounded конструкция вида означает
  - `private static abstract class A1<T extends A1<T>> {}`
  - что тип T ограничен классом A1 и его наследниками
  - класс A1 абстрактный, то невозможно создать экземпляр <T> конструктором
  - <T> НЕ МОЖЕТ работать переменной в конструкторе
  - <T> экземпляр МОЖНО задать методом setValue()
  - <T> toString() НЕ РАБОТАЕТ переопределением,зацикливание, использовать show()
- Пример. реализация абстрактного класса с Generic Bounded <T extends A1<T>>

```
private static abstract class A1<T extends A1<T>> {
    T value;

    public T getValue() {                          // создание экземпляра закрытого класса
        return value;
    }
    public <E extends T> void setValue(E value) { // гарантированно проверяет тип checked
        this.value = (T) value;
    }
    public String show() {
        return "value";
    }
}
```

- Пример. реализация класса наследующего абстрактный класс Generic Bounded <T extends A1<T>>

```
private static class A2 extends A1 implements A<A1> {
    String s;
    int n;
    private A2(String s, int n) {
        this.s = s;
        this.n = n;
    }
    @Override
    public A1 getValue() {
        return value;
    }
    @Override
    public String toString() {
//            return "A2[" + s + "," + n + "," + value + ']';          // Stack Overflow
        if (value == null)
            return "A2[" + s + "," + n + "," + "null" + ']';
        else
            return "A2[" + s + "," + n + "," + value.show() + ']';
    }
}

public static void main(String[] args) {
    System.out.println("\nGeneric bounded:");
    A2 a2 = new A2("a2", 12);
    A2 a21 = new A2("a21", 521);
    a2.setValue(a21);
    System.out.println("a2          :" + a2);
    System.out.println("a2.getValue:" + a2.getValue());
}
```

- Что такое EnumSet? **java_c01/enums/Main02Enum**
  - EnumSet Set для enum объектов, использует внутри битовые вектора, заменяет BitSet
  - iterator() weakly consistent не выдает ConcurrentModificationException
  - но и не определяет изменений, которые сделаны во время итераций
  - not sync несинхронный класс по умолчанию, синхронная версия возможна если
  - применить метод Collections.synchronizedSet(EnumSet)

- Для множеств enum-ов есть специальный класс java.util.EnumSet? Зачем? Чем авторов не устраивал HashSet или TreeSet?
    - EnumSet используется как Set<> для хранения объектов Enum
    - внутри представлен как RegularEnumSet для малых наборов до 64 элементов
    - JumboEnumSet для больших наборов
    - элементы представляют бит вектор, то есть последовательность бит, компактно и эффективно
    - методы containsAll() и retainAll() работает очень быстро
    - iterator EnumSet.iterator() weakly consistent, не выдает ConcurrentModificationException
    - но может не заметить изменений во время итераций
- Пример. реализации **java_c01/enums/Main02Enum**
- ```
  EnumSet<eValue> eSet = EnumSet.allOf(eValue.class); // all values
  Iterator<eValue> eIt = eSet.iterator();
  while(eIt.hasNext()) {
      eValue e = eIt.next();
      System.out.print(e.name()+":"+eValue.valueOf(e.name())+" ");
  }
  for (eValue value : eSet) {
      System.out.print(value.name()+":"+eValue.valueOf(value.name())+" ");
  }
  ```
- ```
  private enum eValue implements Runnable {
      JAVA("Java"), SCALA("Scala"), KOTLIN("Kotlin", 12) {
          @Override
          public void show() {
              System.out.println("KOTLIN show():"+getS());
          }
          @Override
          public String toString() {
              return "ENUM[" + this.getS() + ", " + getN() + ']';
          }
      }, CPP("Cpp", 51), PASCAL {};
      private final String s;
      private int n;
      private eValue() {
          s = null;
      }
      private eValue(String s) {
          this.s = s;
      }
      private eValue(String s, int n) {
          this.s = s;
          this.n = n;
      }
      public void show() {
          System.out.println(s);
      }
      public String getS() {
          return s;
      }
      public int getN() {
          return n;
      }
      @Override
      public void run() {
          System.out.println("Enum:"+s+" "+n+" runnable");
      }
      @Override
      public String toString() {
          return "Enum[" + s + "]";
      }
  }
  ```

- java.util.Stack - считается "устаревшим". Чем его рекомендуют заменять? Почему?
  - Stack — расширяет класс Vector который работает медленно так как является синхронным
  - не совсем отвечает требованиям Stack, т.к. расширяет методы Vector
  - произвольного доступа к элементам по индексу
  - Deque — интерфейс рекомендуется к работе со Stack
  - LinkedList, ArrayDeque
- Какая коллекция реализует дисциплину обслуживания FIFO?
  - Queue — интерфейс реализует метод FIFO
- Какая коллекция реализует дисциплину обслуживания FILO?
  - Stack — класс на базе Deque реализует метод FILO
- Приведите пример, когда какая-либо коллекция выбрасывает UnsupportedOperationException.
  - операции — добавления над результатом методов выдают исключение
  - Arrays.asList(),
  - Collections.unmodifiableList(nList)
  - Collections.emptyList()
  - Collections.singletonList("String")
  - Map.keyset().add(), Map.values().add(), Map.entrySet().add()
- Пример. реализация **java_c01/Main02**
- Почему нельзя написать "ArrayList<List> numbers = new ArrayList<ArrayList>();" но можно "List<ArrayList> numbers = new ArrayList<ArrayList>();"?
  - Generic — должен расширять класс хранения, чтобы можно было сделать подстановку
  - либо точно соответствовать классу хранения
- Пример.
- ```
  // ArrayList
          ArrayList<? extends List> listA = new ArrayList<ArrayList>(); //аргумент должен точно
          ArrayList<? extends List> listB = new ArrayList<ArrayListM>();// соответствовать
          List<ArrayList> listC = new ArrayList<ArrayList>();           // либо расширять
          ArrayList<? extends A> listD = new ArrayList<B>();
      }

      private static class A {
      }
      private static class B extends A {
      }

      private static class ArrayListM<T> extends ArrayList<T> {
      }
  ```

- LinkedHashMap - что это еще за "зверь"? Что в нем от LinkedList, а что от HashMap?
  - LinkedHashMap — расширяет HashMap и все методы для добавления, удаления элементов
  - Entry<K,V> — расширяет HashMap.Node<K,V>
  - методы put() и get() меняют указатели p.before, p.after
  - EntrySetIterator перемещается по указателям в порядке добавления
  - accessOrder — позволяет итерации по элементам в порядке последнего доступа get()
- LinkedHashSet - что это еще за "зверь"? Что в нем от LinkedList, а что от HashSet?
  - LinkedHashSet — based on LinkedHashMap — итератор работает в порядке добавления
  - HashSet — based on HashMap — итератор работает в внутреннем порядке
  -
  ```

- Говорят, на LinkedHashMap легко сделать простенький кэш с "invalidation policy", знаете как?
  - LinkedHashMap — имеет поле accessOrder и метод removeEldestEntry()
  - accessOrder = true группирует часто используемые элементы в начале Map
  - removeEldestEntry() вызывается при каждом добавлении если надо удалить
  - принцип — наследуется LinkedHashMap
  - конструктор с флагом accessOrder = true
  - переопределяется метод removeEldestEntry() с флагом size() > MAX_CNT
  - процедура — при добавлении нового элемента и превышении лимита будет удален
  - первый добавленный элемент
  - при чтении из за accessOrder прочитанные элементы помещаются в начало
  - при новом добавлении, будет удален реже всего читаемый элемент
- Пример. реализации LRUCache **java_c01/Main02**
- ```
  private static class LRUCache<K, V> extends LinkedHashMap<K, V> {
      private static final int MAX_ENTRIES = 7;
      public LRUCache() {
          super(16, 0.75f, true);
      }
      @Override
      protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
          return size() > MAX_ENTRIES;  // request to remove node
      }
  }
  ```
- ```
  / LRU Cache
          System.out.println("\nLRUCache");
          LinkedHashMap<String, String> lruMap = new LRUCache<>();
          lruMap.put("s1", "v1");
          lruMap.put("s2", "v2");
          lruMap.put("s3", "v3");
          lruMap.put("s4", "v4");
          lruMap.put("s5", "v5");
          lruMap.put("s6", "v6");
          lruMap.put("s7", "v7");
          lruMap.put("s8", "v8");
          lruMap.put("s9", "v9");
          System.out.println("put:"+lruMap);
  // access
          lruMap.get("s2");
          lruMap.get("s8");
          lruMap.get("s5");
          lruMap.get("s3");
          System.out.println("get:"+lruMap);
          lruMap.put("s15", "v5");
          System.out.println("put:"+lruMap);
          lruMap.put("s16", "v6");
          System.out.println("put"+lruMap);
  ```

- Что позволяет сделать PriorityQueue?
  - Comparator — который позволяет сравнивать элементы списка или массива очереди
  - после сравнения по полям элемент вставляется или удаляется из очереди
- В чем заключаются отличия java.util.Comparator от java.lang.Comparable?
  - Comparable — интерфейс с методом compareTo(o)
  - сравнивает поле данного класса с параметром
  - Comparator — интерфейс с методом compare(o1, o2)
  - сравнивает два аргумента
  - Comparable<T> — используется в качестве интерфейса для коллекций с сортировкой
  - Comparator — используется в качестве аргумента метода Collestions.sort()

Java8

- Какие нововведения, появились в Java 8?
- lambda полноценная поддержка lambda выражений **java_08/Main01**
  - default метод в интерфейсах
  - ссылки на методы (s->s.toLowerCase()) >> (String::toLowerCase)
  - functional функциональный интерфейс
  - stream потоки для работы с коллекциями
  - Data API для работы с датами **java_08/Main01**
  - Nashorn Javascript support на Java **java_08/Main02**
- Как сортировать список строк с помощью лямбда-выражения?
  - использовать sorted(Comparator::comparing)
- Пример. реализация
- ```
  List<String> names = Arrays.asList("Bob","James","Red","Mike","John","Jimmy","Steve");
  names = names.stream()
          .sorted((s1,s2)->s2.compareTo(s1))
          .collect(Collectors.toList());
  System.out.println(names);

  Collections.sort(names,String::compareTo);
  System.out.println(names);
  ```
- Какова структура лямбда-выражения?
  - структура (параметры) - > (исполнение в одну строку)
  - структура (параметры) - > ({ строка 1; строка2; return value;})
  -
- К каким переменным есть доступ у Лямбда-выражений?
  - доступ final или effective final переменным в области где были определены
  - final переменная которая объявлена как final
  - effective final переменная, объявлена как обычно, но не имеет новых присвоений
- Что такое ссылки на метод?
  - ссылка на метод или конструктор, это компактное выражение для конструкции
  - v1-> v1.method() User::method
  - (v1,v2)-> v1.method(v2) User::method
  - ()-> new User() User::new
- Какие виды ссылок на методы вы знаете?
  - статический User::staticMethod
  - member User::memberMethod v->v.size() String::size()
  - instance User::instanceMethod внешний List<>list v->list.add(v) list::add
  - конструктор User::new ()->new User User::new
- Пример. реализация
- ```
  List<String> names = Arrays.asList("Bob","James","Red","Mike","John","Jimmy","Steve");
  Collections.sort(names,String::compareTo);
  List<String> list = new ArrayList<>();
  names.stream().map(v->v.toLowerCase()).forEach(v->list.add(v));
  names.stream().map(String::toUpperCase).forEach(list::add);
  ```
- ```
  Map<Character,List<String>> tMap = list.stream()
          .collect(Collectors.groupingBy(v->v.toLowerCase().charAt(0),
                  ()->new TreeMap(),Collectors.toList()));

  Map<Character,List<String>> tMap2 = list.stream()
          .collect(Collectors.groupingBy(v->v.toLowerCase().charAt(0),
                    TreeMap::new,Collectors.toList()));
  ```
  ```

- Объясните выражение System.out::println
  - ссылка на метод v->System.out.println(v)
- Что такое функциональные интерфейсы?
  - интерфейс с одним абстрактным методом называется функциональным интерфейсом
  - может иметь любое число static методов
  - может иметь любое число default методов
- Для чего нужен функциональный интерфейс BiConsumer<T,U>?
  - выполняет операцию с двумя аргументами типа T и U, возвращает void
- Для чего нужен функциональный интерфейс BiFunction<T,U,R>?
  - выполняет операцию с двумя аргументами типа T и U, возвращает R
- Для чего нужен функциональный интерфейс BinaryOperator<T>?
  - выполняет операцию с двумя аргументами типа T, возвращает результат T
- Для чего нужен функциональный интерфейс BiPredicate<T,U>?
  - выполняет операцию над двумя аргументами типа Ти U и возвращает boolean
- Для чего нужен функциональный интерфейс BooleanSupplier?
  - создает объекты типа Boolean
- Для чего нужен функциональный интерфейс UnaryOperator<T>?
  - выполняет операцию над одним аргументом типа T и возвращает результат T
  - по сути это интерфейс Function<T,T>
- Для чего нужен функциональный интерфейс Predicate<T>?
  - выполняет операцию над одним аргументом типа T и возвращает результат Boolean
- Для чего нужен функциональный интерфейс Consumer<T>?
  - выполняет операции над аргументом типа T возвращает void
- Для чего нужен функциональный интерфейс DoubleBinaryOperator?
  - выполняет операцию как BinaryOperator<Double>
- Для чего нужен функциональный интерфейс DoubleConsumer?
  - выполняет операцию как Consumer<Double>
- Для чего нужен функциональный интерфейс DoubleFunction<R>?
  - выполняет операцию как Function<Double,R>
- Для чего нужен функциональный интерфейс DoublePredicate?
  - выполняет операцию как Predicate<Double>
- Для чего нужен функциональный интерфейс DoubleSupplier?
  - выполняет операцию как Supplier<Double>
- Для чего нужен функциональный интерфейс DoubleToIntFunction?
  - выполняет операцию Function<Double,Integer> принимает аргумент double, возвращает int
- Для чего нужен функциональный интерфейс DoubleToLongFunction?
  - выполняет операцию Function<Double,Long> принимает аргумент double, возвращает long
- Для чего нужен функциональный интерфейс DoubleUnaryOperator?
  - выполняет операцию UnaryOperator<Double>
- Для чего нужен функциональный интерфейс Function<T,R>?
  - выполняет операцию с одним аргументом типа T и возвращает результат типа R
- Для чего нужен функциональный интерфейс IntBinaryOperator?
  - выполняет операцию как BinaryOperator<Integer>
- Для чего нужен функциональный интерфейс IntConsumer?
  - выполняет операцию как Consumer<Integer>

- Для чего нужен функциональный интерфейс IntFunction<R>?
  - выполняет операцию как Function<Integer,R>
- Для чего нужен функциональный интерфейс IntPredicate?
  - выполняет операцию как Predicate<Integer>
- Для чего нужен функциональный интерфейс IntSupplier?
  - выполняет операцию как Supplier<Integer>
- Для чего нужен функциональный интерфейс IntToDoubleFunction?
  - выполняет операцию Function<Integer,Double> принимает аргумент int, возвращает double
- Для чего нужен функциональный интерфейс IntToLongFunction?
  - выполняет операцию Function<Integer,Long> принимает аргумент int, возвращает long
- Для чего нужен функциональный интерфейс IntUnaryOperator?
  - выполняет операцию UnaryOperator<Integer>
- Для чего нужен функциональный интерфейс LongBinaryOperator?
  - выполняет операцию как BinaryOperator<Long>
  - использует два аргумента типа Long и возвращает результат типа Long
- Для чего нужен функциональный интерфейс LongConsumer?
  - выполняет операцию как Consumer<Long>
- Для чего нужен функциональный интерфейс LongFunction<R>?
  - выполняет операцию как Function<Long,R>
  - использует аргумент типа Long и возвращает результат типа R
- Для чего нужен функциональный интерфейс LongPredicate?
  - выполняет операцию как Predicate<Long>
  - использует аргумент типа Long и возвращает результат типа Boolean
- Для чего нужен функциональный интерфейс LongSupplier?
  - выполняет операцию как Supplier<Long>
  - создает объекты типа Long
- Для чего нужен функциональный интерфейс LongToDoubleFunction?
- выполняет операцию как Function<Long,Double> **java_08/Main01A**
- Для чего нужен функциональный интерфейс LongToIntFunction?
  - выполняет операцию как Function<Long,Integer>
  - использует аргумент типа Long и возвращает результат типа Integer
- Для чего нужен функциональный интерфейс LongUnaryOperator?
  - выполняет операцию как UnaryOperator<Long>
  - использует аргумент типа Long и возвращает результат типа Long
- Для чего нужен функциональный интерфейс ObjDoubleConsumer<T>?
- выполняет операцию как BiConsumer<T,Double> **java_08/Main01A**
  - использует два аргумента, типа T и Double и выполняет операцию, возвращает void
- Для чего нужен функциональный интерфейс ObjIntConsumer<T>?
  - выполняет операцию как BiConsumer<T,Integer>
- Для чего нужен функциональный интерфейс ObjLongConsumer<T>?
  - выполняет операцию как BiConsumer<T,Long>
- Для чего нужен функциональный интерфейс Predicate<T>?
  - использует аргумент типа T и возвращает результат Boolean

- Для чего нужен функциональный интерфейс Supplier<T>?
  - создает объекты типа T
- Для чего нужен функциональный интерфейс ToDoubleBiFunction<T,U>?
- выполняет операцию как BiFunction<T,U,Double> **java_08/Main01A**
  - использует аргументы типа T и U и возвращает результат типа Double
- Для чего нужен функциональный интерфейс ToDoubleFunction<T>?
  - выполняет операцию как Function<T,Double>
  - использует аргумент типа T и возвращает результат типа Double
- Для чего нужен функциональный интерфейс ToIntBiFunction<T,U>?
  - выполняет операцию как BiFunction<T,U,Integer>
  - использует аргументы типа T и U и возвращает результат типа Integer
- Для чего нужен функциональный интерфейс ToIntFunction<T>?
  - выполняет операцию как Function<T,Integer>
  - использует аргумент типа T и возвращает результат типа Integer
- Для чего нужен функциональный интерфейс ToLongBiFunction<T,U>?
  - выполняет операцию как Function<T,U,Long>
  - использует аргументы типа T и U и возвращает результат типа Long
- Для чего нужен функциональный интерфейс ToLongFunction<T>?
  - выполняет операцию как Function<T,Long>
  - использует аргумент типа T и возвращает результат типа Long
- Для чего нужен функциональный интерфейс UnaryOperator<T>?
  - использует аргумент типа T и возвращает результат типа T
- Что такое StringJoiner?
  - StringJoiner класс который используется stream() для объединения строк
  - Collectors.joining() использует StringJoiner объект для выполнения операции
  - при создании можно задать delimeter, prefix, suffix и emptyValue
  - если задано emptyValue, то вместо {prefix,suffx} выводится {emtpyValue}
- Пример. реализация **java_08/Main01A**

```
StringJoiner stringJoiner = new StringJoiner(",", "{", "}");
stringJoiner.setEmptyValue("[]");
StringJoiner sj = new StringJoiner(":", "[", "]");
sj.add("George").add("Sally").add("Fred");
String desiredString = sj.toString();
System.out.println(sj.toString());
//  A StringJoiner may be employed to create formatted output from a Stream using
// Collectors.joining(CharSequence). For example:
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
String commaSeparatedNumbers = numbers.stream()
       .map(String::valueOf)
       .collect(Collectors.joining(", "));
```

- Что такое default методы?
  - default метод интерфейса, это метод реализованный в теле интерфейса
  - который не обязательно переопределять, кроме случая diamond conflict
  - если класс реализует два интерфейса имеющие одинаковый default метод
  - класс ОБЯЗАН реализовать этот метод чтобы избежать diamond конфликт

- Как вызывать default-метод интерфейса в классе?
  - вызов — default метода также как любого метода интерфейса который был реализован
  - с помощью имени интерфейса и объекта super — User.super.print("Message")
- Пример. реализация
- ```
  private interface IUser {
      static boolean valid(String s) {
          return s != null && !s.isEmpty();
      }
      static boolean valid(int i) {
          return i < 0;
      }
      default void print(String s) {
          if (!valid(s)) s = "Message: empty";
          System.out.println(s);
      }
      String print(Integer i);
  }
  private static class User implements IUser {
     @Override
      public String print(Integer i) {
          IUser.super.print("startInteger");
          print("Send Message");
          if (IUser.valid(i)) i = 0;
          return String.valueOf(i + 1);
      }
  }
  ```
- Что такое static методы?
  - static — метод интерфейса, это метод, реализованный в теле интерфейса
  - может быть вызван ТОЛЬКО из класса интерфейса
  - реализуют — вспомогательные функции, как контроль входных параметров, сортировка
  - не позволяют классам переопределить их, обеспечивают защиту интерфейса
  - нельзя — определить статические методы класс Object
  -
- Как вызывать static-метод интерфейса в классе?
  - вызывается — только из тела класса интерфейса
- Пример. реализация
- ```
  // default interface
          IUser iUser = (v) -> (String.valueOf(v + 1));
          iUser.print("Message:" + iUser.print(2));
          User user = new User();
          user.print("Message:"+user.print(2));
          user.print("");
  ```
- Что такое потоки(stream) в Java 8
  - пакет — java.util.stream реализует потоки в Java8 — **java8/stream**
  - stream() — это default метод интерфейса Collection который создает объект Stream<T>
  - Stream<T> — это интерфейс наследует интерфейс BaseStream<T,Stream<T>>
  - операции — intermediate, terminal
  - intermediate — подготавливают обработку данных — **java8/stream/Main02i**
  - filter, distinct, limit, map, sorted, range
  - terminal — запускают intermediate и выполняются сами — **java8/stream/Main02t**
  - forEach, min, max, count, sum, average, reduce,
  - collect, findFirst, findAny, anyMatch, allMatch

- Для чего нужен метод collect Java 8?
  - collect — метод, который преобразует поток в коллекцию List<>, Set<>, Map<>
  - можно переопределить Collector<>класс и создать свой формирователь коллекции
  - Пример. реализация своего класса Collector2 и метода toList **java8/stream/Main02tList**
- В чем разница между коллекцией(Collection) и потоком(Stream)?
  - коллекция — это массив данных, в котором можно элементы обрабатывать в любом порядке
  - поток — это одноразовый массив данных, в котором элементы доступны по одному
- Для чего предназначен метод forEach в потоках(stream)?
  - forEach — terminal операция на базе интерфейса Consumer<T>
- Как вывести на экран 10 случайных чисел, используя forEach?
  - реализация — с помощью методов range() и Random.ints()
- Пример. реализация **java_08/Main01A**

```
// stream
        Random rnd = new Random();
 // without boundaries
        rnd.ints().limit(10).forEach(v -> System.out.printf("%d ", v));
        System.out.println();
// with boundaries
        IntStream.range(0, 10)
                .forEach(v -> System.out.printf("%d ", (int) (Math.random() * 100)));
        System.out.println();
        IntStream.range(0, 10).forEach(v -> System.out.printf("%d ", rnd.nextInt(100)));
        System.out.println();
        rnd.ints(10, 0, 100).forEach(v -> System.out.printf("%d ", v));
        System.out.println();
        SecureRandom sRnd = new SecureRandom();
        sRnd.doubles(50,100).limit(10).forEach(v-> System.out.printf("%.1f ",v));
```

- Для чего предназначен метод map в потоках(stream)?
  - выполняет — преобразование поэлементную обработку с возможным преобразованием типа
- Как можно вывести на экран уникальные квадраты чисел используя метод map?
  - использовать — map(v->v*v)
  - сортированные значения sorted() или forEachOrdered
- Пример. реализация **java_08/Main01A**

```
rnd.ints(10, -10, 10)
        .map(v ->v * v)
        .distinct()
        .sorted()
        .forEach(v -> System.out.printf("%d ", v));
```

- Какова цель метода filter в потоках(stream)?
  - оставить — в потоке только те элементы, которые удовлетворяют условию фильтра
- Как вывести на экран количество пустых строк с помощью метода filter?
  - использовать — filter(String::empty)
- Пример. реализация **java_08/Main01A**

```
long count = listS.stream().filter(String::isEmpty) .count();
```

- Для чего предназначен метод limit в потоках(stream)?
  - limit() — ограничивает число элементов потока от начала до значения в методе
- Для чего предназначен метод sorted в потоках(stream)?
  - sorted() — сортирует элементы в потоках по встроенному или внешнему Comparator
  - IntStream.sorted() — нет аргументов и только acsending order
  - Stream.sorted(Comparator<T>) — использует custom Comparator

- Как вывести на экран 10 случайных чисел в отсортированном порядке в Java 8?
  - использовать sorted() или forEachOrder()
- Пример. реализации
- ```
  rnd.ints(10, 0, 100).forEachOrdered(v -> System.out.printf("%02d ", v));
  rnd.ints(0, 100).limit(10).sorted()
          .forEach(v -> System.out.printf("%02d ", v));
  ```

- Параллельная обработка в Java 8?
  - parallel() метод параллельной обработки потока
  - parallelStream() метод создания потока в параллельном потоке
- Пример. реализация **java_08/Main01A**
- ```
  long count2 = listS.parallelStream().filter(String::isEmpty).count();
  long count3 = listS.stream().parallel().filter(String::isEmpty).count();
  ```

- Как найти максимальное число в списке Java 8?
  - max() Stream<T>.max()
  - max() IntStream.max() метод прямого нахождения
  - statistics IntStream.summaryStatistics().getMax() через статистику
- Как найти минимальное число в списке Java 8?
  - min() Stream<T>.min()
  - min() IntStream.min() метод прямого нахождения
  - statistics IntStream.summaryStatistics().getMin() через статистику
- Как получить сумму всех чисел в списке, используя Java 8?
  - reduce() Stream<T>.reduce(0,(v1,v2)->v1+v2) через reduce()
  - sum() IntStream.sum() метод прямого нахождения
  - statistics IntStream.summaryStatistics().getSum() через статистику
- Как получить среднее значение всех чисел, в списке, используя Java 8?
  - внешний Stream<T> только через внешний подсчет
  - average() IntStream.average() метод прямого нахождения
  - statistics IntStream.summaryStatistics().getAverage() через статистику
- Пример. реализация **java_08/Main01A**
- ```
  System.out.println("listI:"+listI);
  int[] values = listI.stream().mapToInt(v->v).toArray();

  int maxInteger = listI.stream().max(Integer::compareTo).orElse(0);
  int minInteger = listI.stream().min(Integer::compareTo).orElse(0);
  int sumInteger = listI.stream().reduce(0, (v1, v2) -> v1 + v2);
  double averageInteger = (double) sumInteger / listI.size();

  IntSummaryStatistics statistics = listI.stream().mapToInt(v -> v).summaryStatistics();
  int max = listI.stream().mapToInt(v -> v).max().orElse(0);
  int min = listI.stream().mapToInt(v -> v).min().orElse(0);
  int sum = listI.stream().mapToInt(v -> v).sum();
  double average = listI.stream().mapToInt(v -> v).average().orElse(0);
  ```

- ```
  System.out.println("min: " + statistics.getMin() + " max: " + statistics.getMax() +
          " sum:" + statistics.getSum() + " avg:" + statistics.getAverage());

  statistics = IntStream.of(values).summaryStatistics();
  max = IntStream.of(values).max().orElse(0);
  min = IntStream.of(values).min().orElse(0);
  sum = IntStream.of(values).sum();
  average = IntStream.of(values).average().orElse(0);
  System.out.println("min: " + statistics.getMin() + " max: " + statistics.getMax() +
          " sum:" + statistics.getSum() + " avg:" + statistics.getAverage());
  ```

- Что такое Optional?
  - контейнер результата может иметь null или значение <T>
  - методы isPresent() false если null value если есть значение
  - orElse(other) other если null value если есть значение
  - orElseGet(Supplier) ()->other null value если есть значение
- Пример. реализация
- ```
  // Optional
  OptionalInt optInt = IntStream.of(values).max();
  OptionalDouble optDbl = listI.stream().mapToInt(v -> v).average();
  System.out.println("oInt:"+optInt.isPresent()+" value:"+optInt.orElseGet(()->-1));// supplier
  System.out.println("oDbl:"+optDbl.isPresent()+" value:"+optDbl.orElse(0));        // supplier
  ```

- Что такое Nashorn?
  - engine позволяет встраивать JavaScript в приложения Java
  - позволяет вызывать код Java из JavaScript
  - скорость в 2..10 раз быстрее по сравнению с Rhino, который поддерживает Mozilla
- Что такое jjs в Java 8?
  - утилита прямого запуска JavaScript в консоли cmd
- Пример. реализация запуск script.js из консоли и просмотр ответа в файле result.txt **java_08/Main02**
- ```
  //jjs
          System.out.printf(format, "Nashorn jjs:");
          File f = new File("./data/result.txt");
          if(f.exists()){
              if(!f.delete())throw new RuntimeException("Can't delete file:"+f.getPath());
          }
          Runtime.getRuntime().exec("cmd /c start .\\data\\start.cmd");
          String s ="";
          try {
              while (!f.exists() || f.length() <= 0) {
                  Thread.sleep(200);
                  System.out.print(".");
              }
              System.out.println();
              FileInputStream fs = new FileInputStream(f);
              BufferedReader br = new BufferedReader(new InputStreamReader(fs, "utf-8"));
              s = br.lines().parallel().collect(Collectors.joining());
          } catch (IOException e) {
              e.printStackTrace();
          }
          System.out.println("script.js: " + s);
      }
  ```
- Что такое LocalDateTime в Java 8?
  - LocalDateTime класс объединет LocalDate и LocalTime, хранит время без часового пояса
  - поддерживает многопоточность
  - хранит время с точностью до наносекунд
  - методы plus, minus,with, range, adjustInto, trancatedTo, isSupported ,isAfter
  - range() диапазон для заданного поля времени
  - isSupported() поддерживает ли данное время заданное поле
  - adjustInto() сделать копию времени по параметру
  - isAfter() показывает позже ли время чем параметр
- Что такое ZonedDateTime в Java 8?
  - ZonedDateTime класс времени, который включает зону, то есть часовой пояс в дату и время

- Как получить текущую дату с использованием time API из Java 8?
  - метод LocalDateTime.now()
- Как добавить 1 неделю к текущей дате с использованием time API?
  - метод LocalDateTime.plus()
- Как добавить 1 месяц к текущей дате с использованием time API?
  - метод LocalDateTime.plusMonths()
- Как добавить 1 год к текущей дате с использованием time API?
  - метод LocalDateTime.plusYears()
- Как добавить 10 лет к текущей дате с использованием time API?
  - метод LocalDateTime.plusYears()
- Как получить следующий вторник используя time API?
  - метод with()
- Как получить вторую субботу текущего месяца используя time API?
  - метод with() дважды, сначала с первым днем недели месяца, затем со следующим
- Как получить текущую дату в миллисекундах используя time API?
  - метод Instant.now().toEpochMilli()
- Как получить текущую дату по местному времени в миллисекундах используя time API?
  - метод ZonedDateTime.toInstant().toEpochMilli(
- Пример. реализация **java_08/Main01B**

```
LocalDateTime localDateTime = LocalDateTime.now();

LocalDateTime plusWeek = localDateTime.plus(1,ChronoUnit.WEEKS);

LocalDateTime plusMonth = localDateTime.plusMonths(1);

LocalDateTime plusYear = localDateTime.plus(1,ChronoUnit.YEARS);
//        LocalDateTime plusYear = localDateTime.plusYears(1);

LocalDateTime plus10Year = localDateTime.plus(1,ChronoUnit.DECADES);
//        LocalDateTime plus10Year = localDateTime.plusYears(10);

LocalDateTime nextTue = localDateTime.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));

LocalDateTime secondSat = localDateTime
       .with(TemporalAdjusters.firstInMonth(DayOfWeek.SATURDAY))
       .with(TemporalAdjusters.next(DayOfWeek.SATURDAY));

localDateTime = LocalDateTime.now();

long localMilli = localDateTime.toInstant(ZoneOffset.ofHoursMinutes(3,0)).toEpochMilli();

long timeMilliSeconds = Instant.now().toEpochMilli();

ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime,ZoneId.of("Europe/Moscow"));
System.out.println("Moscow ms       : "+zonedDateTime.toInstant().toEpochMilli());

zonedDateTime = ZonedDateTime.of(localDateTime,ZoneId.systemDefault());
System.out.println("milliseconds    : "+zonedDateTime.toInstant().toEpochMilli());

Instant instant = Instant.ofEpochMilli(timeMilliSeconds);
localDateTime = instant.atZone(ZoneId.systemDefault()).toLocalDateTime();

localDateTime = LocalDateTime.ofInstant(instant,ZoneId.systemDefault());

Calendar c = Calendar.getInstance();
c.setTimeInMillis(timeMilliSeconds);
System.out.println("calendar from ms: "+c.getTime());
```

- Классы Java8 кодирования данных?
  - Base64 класс состоит только из статических методов
  - Base64.Encoder класс кодировщика
  - Base64.Decoder класс декодировщика
  - типы Simple набор A-Za-z0-9+/ в наборе нет символов управления строкой
  - getDecoder(), getEncoder()
  - URL набор A-Za-z0-9+_ набор безопасен для имен файлов
  - getUrlDecoder(), getUrlEncoder()
  - MIME набор содержит MIME формат, по 76 символов в строке и \r\n
  - getMimeDecoder(), getMimeEncoder()
- Какой класс появился в Java 8 для декодирования данных?
  - Base64.Decoder класс декодировшика по схеме RFC4648 и RFC2045
  - класс поддерживает многопоточность
  - в конце файла допускается отсутствие символов padding "="
  - если они есть их число должно быть КОРРЕКТНО
  - методы decode() декодирует Base64 данные в byte[] массив или ByteBuffer
  - wrap() декодирует Base64 InputStream и возвращает InputStream данных
- Какой класс появился в Java 8 для кодирования данных?
  - Base64.Encoder класс кодировшика по схеме RFC4648 и RFC2045
  - класс поддерживает многопоточность
  - методы encode() кодирует массив byte[] в Base64 byte[] массив или ByteBuffer
  - wrap() кодирует OutpuStream данных в Base64 OutpuStream данных
- Как создать Base64 декодировщик?
  - метод Base64.getDecoder() создает объект декодировщика
- Как создать Base64 кодировщик?
  - метод Base64.getEncoder() создает объект кодировщика
- Пример. реализация Base64 кодирования и декодирования данных **java_08/Main03**
- ```
  //Base64
  ```

```
String format = "%n%s%n------------------------%n";

System.out.printf(format,"Base64 Encode:");
String originalInput = "test input";
String encodedString = Base64.getEncoder().encodeToString(originalInput.getBytes());
System.out.println("input           : "+originalInput);
System.out.println("Base64          : "+encodedString);


System.out.printf(format,"Base64 Decode:");

String encodedWoPadding = encodedString.replaceAll("=*$","");
byte[] decodedBytes = Base64.getDecoder().decode(encodedString);
byte[] decodedWoPadding = Base64.getDecoder().decode(encodedWoPadding);
System.out.println("decoded         : "+new String(decodedBytes));
System.out.println("decodedWoPadding: "+new String(decodedWoPadding));


System.out.printf(format,"MIME Encode:");
String originalMIME = UUID.randomUUID().toString();
String encodedMIME = Base64.getMimeEncoder().encodeToString(originalMIME.getBytes());

String decodedMIME = new String(Base64.getMimeDecoder().decode(encodedMIME));
System.out.println("original MIME:"+originalMIME);
System.out.println("encode MIME  :"+encodedMIME);
System.out.println("decode MIME  :"+decodedMIME);
```
```

Java IO Stream

- Что такоє символьная ссылка?
  - Linux
- Какая разница между I/O и NIO?
  - Java IO stream() ориентированный, в единицу времени доступен только один элемент
  - синхронный ввод/вывод, блокируется если одна сторона не успевает
  - JavaNIO ориентирован на буферный обмен
  - асинхронный ввод/вывод за счет использования буферов
  - используются селекторы, используют каналы для направления данных в буферы
- Какие особенности NIO вы знаете?
  - channel канал является абстракцией объектов файловой системы
  - каналы не блокируются
  - selector селектор, использует для передачи данных каналы или сокеты
  - buffer добавлены классы обертки буферизации всех типов данных кроме Boolean
  - методы класса Buffer clear, flip, mark
  - charset наборы символов для отображения байт и символов Unicode
- Какие существуют виды потоков ввода/вывода?
  - byte InputStream, OutputStream байт ориентированные потоки
  - char Reader, Writer символьно ориентированные потоки
- Назовите основные классы потоков ввода/вывода.
  - InputStream поток ввода байт данных,
  - FileInputStream работа с файлами
  - ByteArrayInputStream с массивом байт
  - PipedInputStream с Pipe объектои
  - DataInputStream позволяет работать с примитивными типами напрямую
  - PushBackInputStream позволяет возвращать элементы обратно в поток
  - BufferedInputStream буферизованная версия
  - PrintStream запись символов в поток байт
  -
  - Reader базовый класс потока ввода символов
  - InputStreamReader адаптер для подключения InputStream к Reader
  - FileReader чтение символов в файл
  - StringReader чтение строк
  - PipedReader работа с Pipe
  - PushbackReader позволяет возвращать символы в поток
  - BufferedReader буферизованный поток ввода символов данных
  - PrintWriter запись символов в байты или в символы
  -
- Чем отличаются и что общего у OutputStream, InputStream, Writer, Reader?
  - общее то, что это потоки которые получают свои данные из
  - массива байт, строки, файла, объекта Pipe
  - разница потоки InputStream работают с байтами
  - потоки Reader работают с символами
-

- Какие подклассы базового класса InputStream Вы знаете, для чего они предназначены?
  - FileInputStream работа с файлами
  - ByteArrayInputStream с массивом байт
  - PipedInputStream с Pipe объектои
  - DataInputStream позволяет работать с примитивными типами напрямую
  - BufferedInputStream буферизованная версия
  - FilterInputStream класс декоратор, у него protected Constructor
  - его ОБЯЗАТЕЛЬНО надо расширять чтобы использовать
  - PushBackInputStream позволяет возвращать элементы обратно в поток
  - PrintStream запись символов в поток байт
  -
- Что вы знаете о RandomAccessFile?
  - класс IO работы с файлом, позволяет получить произвольный доступ к файлу
  - наследуется напрямую от Object, это самостоятельный класс
  - реализует интерфейы DataInput, DataOutput
  - позволяет работать с примитивными типами данных и строками в UTF-8
  - имеет метод seek() произвольного доступа в любую позицию файла
- Какие есть режимы доступа к файлу есть у RandomAccessFile ?
  - режимы создание, чтение, запись
- Какие подклассы базового класса OutputStream ви знаєте, для чего они предназначены?
  - FileOutputStream работа с файлами
  - ByteArrayOutputStream с массивом байт
  - PipedOutputStream с Pipe объектои
  - BufferedOutputStream буферизованная версия
  - FilterOutputStream декоратор для потоков
  - DataOutputStream позволяет работать с примитивными типами напрямую
  -
- Для чего используется PushbackInputStream?
  - для анализа и рекурсивной обработки данных потока
  - позволяет анализировать последовательности байт
- Для чего используется SequenceInputStream?
  - позволяет объединять данные нескольких потоков, читает данные один за другим
- Какие подклассы базового класса Reader Вы знаете, для чего они предназначены?
  - InputStreamReader адаптер для подключения InputStream к Reader
  - FileReader чтение символов в файл
  - StringReader чтение строк
  - PipedReader работа с Pipe
  - PushbackReader позволяет возврашать символы в поток
  - BufferedReader буферизованный поток ввода символов данных
  -
-

- Какие подклассы базового класса Writer Вы знаете, для чего они предназначены?
    - OutputStreamReader — адаптер для подключения InputStream к Reader
    - FileWriter — запись символов в файл
    - StringWriter — запись строк
    - PipedReader — работа с Pipe
    - PushbackReader — позволяет возвращать символы в поток
    - BufferedReader — буферизованный поток ввода символов данных
    - PrintWriter — запись символов в байты или в символы
-
- Что такое абсолютный путь и относительный путь?
- В каких пакетах лежат классы-потоки?
- Что вы знаете о классах-надстройках?
- Какой класс-надстройка позволяет читать данные из входного байтового потока в формате примитивных типов данных?
- Какой класс-надстройка позволяет ускорить чтение/запись за счет использования буфера?
- Какие классы позволяют преобразовать байтовые потоки в символьные и обратно?
- В чем отличие класса PrintWriter от PrintStream?
- Какой класс предназначен для работы с элементами файловой системы?
- Какой символ является разделителем при указании пути в файловой системе?
- Какие методы класса File Вы знаете?
- Что вы знаете об интерфейсе FileFilter?
- Какие классы позволяют архивировать объекты?
-
-
- f

Java MultiThreading

- Как создать неизменяемый объект в Java? Перечислите все преимущества
-
-
- f

JavaCore Tricks

- Trick01
    - рекурсия инициализации объекта класса
- ВНИМАНИЕ. Здесь будет РЕКУРСИЯ инициализации объекта класса
- Пример. реализация

```
public class Main03 {
        Main03 b = new Main03();  // бесконечная рекурсия

        public int show(){
            return (true ? null : 0);
        }

        public static void main(String[] args)  {
            Main03 b = new Main03();
            b.show();
        }
}
```

- Trick02
    - null работает для статических методов
- ВНИМАНИЕ. Вызов СТАТИЧЕСКОГО метода работает даже для NULL объектов
- Пример. реализация
- ```
  public class Main04 {
          public static void show(){
              System.out.println("Static method called");
          }

          public static void main(String[] args)  {
              Main04 obj = null;
              obj.show();  // сработает даже для объекта null если метод статический
          }
  }
  ```
- Trick03
    - a-- - --a a-- дает a
    - - дает a-1
    - --a дает a-1
    - результат a – (a-2) = 2
- ВНИМАНИЕ. Здесь будет
- Пример. реализация
- ```
  public class Main05 {
      static int a = 1111;
      static {
          System.out.println("static:");
          a = a-- - --a;  // 1111 >> (a--) 1110 >> (--a)1109 = 1111 - 1109
          System.out.println("static:"+a);
      }
      {
          System.out.println("object:"+a);
          a = a++ + ++a;  // 2 >>(a++) 3 >> (++a) 4 = 2+4 = 6
          System.out.println("object:"+a);
      }
      public static void main(String[] args) {
          System.out.println(a);
          Main05 m = new Main05();
          Main05 m2 = new Main05();
      }
  }
  ```

- Trick04
    - int = null NullPointerException
- ВНИМАНИЕ. Здесь будет NullPointerException так как переменной int назначается NULL значение
- Пример. реализация
- ```
  public class Main06 {
      private int GetValue() {
          return (true ? null : 0);
      }
      public static void main(String[] args) {
          Main06 obj = new Main06();
          obj.GetValue();
      }
  }
  ```

- Trick05
    - autoboxing автобоксинг работает только для значений -128..127, поэтому
    - для 127 будут созданы переменные типа int
    - для 128 будут созданы стандартные объекты Integer с разным адресом ссылки
- ВНИМАНИЕ. Здесь будет
- Пример. реализация
- ```
  public class Main07 {
      public static void main(String[] args) {
          Integer i1 = -128;   // no autoboxing for int  -INF..129 > value  ||   127 < value
          Integer i2 = -128;
          System.out.println(i1 == i2);

          Integer i3 = 127;   // autoboxing to int  for -128..127
          Integer i4 = 127;
          System.out.println(i3 == i4);
      }
  }
  ```
-
- Trick06
    - сигнатура методов int и Integer разная, autoboxing тут не поможет
- ВНИМАНИЕ. Здесь будет ошибка компиляции
- Пример. реализация
- ```
  public class Main08 {
      private class A {
          void method(int i) {

          }
      }

      private class B extends A {
          @Override
          void method(Integer i) {

          }
      }
  }
  ```
  ```

- Trick07
    - ошибка компиляции new String(null) два конструктора имеют одну сигнатуру
    - ошибка исполнения new Integer(null) NullPointerException
- ВНИМАНИЕ. Здесь будет
- Пример. реализация
- ```
  public class Main09 {
      public static void main(String[] args) {
          Integer i = new Integer(null);  // ошибка на этапе исполнения
          String s = (StringBuffer)new String(null);      // ошибка на этапе компиляции
  два конструктора StringBuffer(), StringBuilder()
  //        StringBuffer s = new StringBuffer(null);
  //        StringBuilder s = new StringBuilder(null);
      }
  }
  ```
- Trick08
    - сумма чисел в строке String не вычисляется, они просто идут подряд
- ВНИМАНИЕ. Сумма чисел в строке НЕ ВЫЧИСЛЯЕТСЯ, они просто идут подряд
- Пример. реализация

```
public class Main10 {
    public static void main(String[] args) {
        String s = "ONE" + 3 + 2 + "TWO" + "THREE" + 5 + 4 + "FOUR" + "FIVE" + 5;
        System.out.println(s);
    }
}         //    ONE32TWOTHREE54FOURFIVE5
```

- Trick09
    - вычисление параметров происходит до входа в метод
- ВНИМАНИЕ. Здесь будет вычисление параметров ДО ВХОДА в метод
- Пример. реализация
- ```
  public class Main11 {
      static int method1(int i) {
          return method2(i *= 11);
      }
      static int method2(int i) {
          return method3(i /= 11);
      }
      static int method3(int i) {
          return method4(i -= 11);
      }
      static int method4(int i) {
          return i += 11;
      }
      public static void main(String[] args) {
          System.out.println(method1(11));
      }
  }                 //    11
  ```

- Trick10
    - ошибка компиляции, так как метод printlin() может принять null для char[] и String
- ВНИМАНИЕ. Здесь будет ошибка компиляции так как метод println() ПРИНИМАЕТ null для char[] и String
- Пример. реализация
- ```
  public class Main12 {
      public static void main(String[] args) {
          System.out.println(null);
      }
  }
  ```
  ```

-
-
- a

Materials

- Data Structure структуры данных
    - стандартные структуры данных
- Algorithm
    - стандартные алгоритмы
-
- f

Ключи JVM

- Ключи JVM
    - –Xbootclasspath ключ загрузки базовых классов
    - –classpath ключ загрузки классов пользователя

Java Developer

- Java Developer
    - language
    - framework
    - pattern
    - database
    - library
    - tools
    - api
    - flow
    - english
- Пример. реализация
    -
    -

Android Developer

- Android Developer
    - development
    - language
    - framework
    - pattern
    - flow
    - database
    - library
    - tools
    - api
    - flow
    - english
- Пример. реализация
    -
    -
    -

Lesson1

OOP

- OOP объектно ориентированное программирование
    - реализация OOD на языке программирования поддерживающего OOP
-

Tests

- Java основы
    - Java Основы Quizful java_basic_tests
    - Java Основы ProgHub java_pattern_tests
-

Вопросы Собеседования

- Q1
- Что покажет программа если

```
short value = 128
```

- Ответ

```
value = 128
```

- *Примечание*

```
byte   -2^7          2^7-1
short  -2^15         2^15-1
int    -2^31         2^31-1
long   -2^63         2^63-1
float  -1.4e-45f     3.4e+38f
double -4.9e-324     1.8e+308
```