Mestre dos Códigos – Território Java

Mateus Felipe Teixeira

Nível Escudeiro

Instruções:

Da compilação:

Os desafios foram todos desenvolvidos na versão 14 do Java e utilizando a IDE IntelliJ. Foi utilizado o Maven para o gerenciamento de dependências. As dependências adicionadas foram o JUnit e Mockito, para ser possível fazer os testes dos desafios. Caso for ser feita a compilação via linha de comando há a necessidade, para algumas classes, compilar outras classes que são nelas usadas.

Da execução:

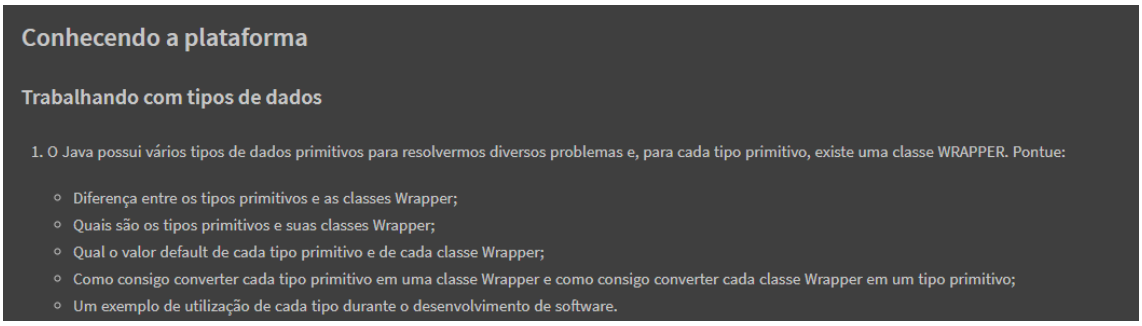
Há um desafio que fiz com interação de entradas de dados via teclado, que é: CompareBigDecimal.java. Além disso, os desafios do Judge também há essa característica.

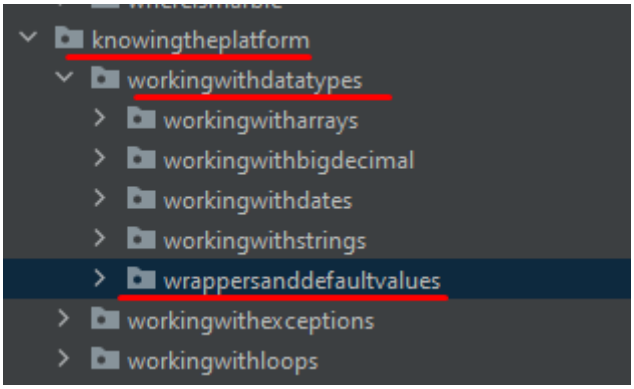
Além das respostas contidas neste documento, nos fontes também há outputs que auxiliam nas respostas e resultados.

Disposição dos códigos fontes:

Como comentado anteriormente, utilizei o IntelliJ para desenvolvimento dos desafios, sendo assim construí um único projeto para todas as respostas. A separação por desafio eu fiz por package, em que cada package estão contidos os códigos fontes de um mesmo grupo de tipo de pergunta. Por exemplo, na página onde estão os desafios (https://db1group.github.io/mestre-dos-codigos/#/java) há o tópico “Conhecendo a plataforma” e dentro deste tópico há o subtópico “Trabalhando com tipos de dados”. Na disposição de packages dentro do projeto ficou: “knowingtheplataform/workingwhitdatatypes” e dentro deste último package há os outros package específicos de cada tipo.

Por exemplo, o primeiro desafio sobre wrapper terá o pacote como “knowingtheplataform/workingwhitdatatypes/wrappersanddefaultvalues”.





Os desafios que são do tópico “O básico do Java” eu criei como “javabasic” e dentro deste pacote cada desafio. Para os desafios do Judge fiz um pacote chamado “judge” e dentro deste cada pacote possui o nome do desafio, fiz isto pois a classe para submissão no site deve ser “Main”, então escolhi essa separação.

Em cada resposta abaixo, para os desafios que houve implementação, eu coloco o caminho da classe daquelas respostas para auxiliar a identificar mais rápido.

Desafios e Respostas:

✓ O básico do Java

1. **Vamos fazer uma viagem ao passado. O Java foi criado na década de 90. Vamos entender qual foi a motivação da criação da linguagem, seus principais conceitos e características e qual sua importância para o desenvolvimento de software. Faça, então, sua pesquisa e pontue seu entendimento sobre tudo isso.**

Resposta:

Teve o início do seu desenvolvimento no início dos anos 90 e originalmente era voltada para dispositivos como televisores, decodificadores e etc, porém ela não se

manteve com este objetivo por ser avançada demais para a tecnologia de tv a cabo da época e posteriormente voltou-se para o desenvolvimento web. A equipe de desenvolvimento inicial era chamada de Green Team (James Gosling, Mike Sheridan e Patrick Naughton) e o primeiro nome do projeto foi Greentalk, posOak, porém mais tarde foi renomeado para Java.

Tinha como principais princípios e características:

- Ser simples, orientada a objetos e familiar: sendo assim a programação e a abstração se tornam semelhante a objetos reais;
- É mais simples que as linguagens em que foram inspiradas (C e C++).
- Ser robusta e segura: Tem uma plataforma de desenvolvimento e execução segura, fornece canais de comunicação seguros e é segura contra corrupção
- Independente de arquitetura e portável: Pode ser executado em qualquer hardware;
- Ser executável com alta performance;
- Ser interpretável, multi-thread e dinâmica: Executar várias tarefas ao mesmo tempo permitindo diminuir o tempo de execução;
- Ser distribuída tento um vasto conjunto de bibliotecas;
- Desalocação de memória automática por processo de coletor de lixo;

Com todas as características e princípios em que a linguagem foi pensada em seu desenvolvimento, ela torna-se uma linguagem importante no mundo de desenvolvimento de software. A começar pela portabilidade em que pode ser uma solução desktop, web ou mobile e independente de sistema operacional, já que a linguagem é interpretada pela sua própria plataforma, a JVM, abstraindo a arquitetura de hardware que está por baixo.

Outro ponto de destaque se dá pelo seu paradigma, a orientação a objetos, o que torna o aprendizado da linguagem muito mais simples, já que se pode fazer relação com o mundo real do que precisa ser desenvolvido. Além de sua sintaxe ser descomplicada e simples. Isso torna o Java uma porta de entrada para o mundo da programação.

A robustez e performance da linguagem também é outro ponto de importância para o Java, em que permite desenvolver desde pequenas aplicações até grandes soluções que precisam ser performáticas. Isso facilitado pelo autogerenciamento de memória, multi-thread entre outros. A facilidade em se desenvolver aplicações também se dá pelo vasto conjunto de bibliotecas da linguagem, muitas destas mantidas de forma aberta por uma grande comunidade de desenvolvedores.

2. **No mundo Java existe uma sopa de letrinhas e siglas, mas algumas são fundamentais para o entendimento do propósito da linguagem. Vamos, então, aprender um pouco mais sobre a plataforma. Pesquise sobre:**
   - **JVM**
   - **JDK**
   - **JRE**

Resposta:

- JVM (Java Virtual Machine):

É a máquina virtual que irá interpretar os Byte Code (.class) e irá se comunicar diretamente com o sistema operacional. O código fonte escrito em Java é compilado pelo compilador Java, o javac, e é gerado o Byte Code (.class) que será interpretado pela máquina virtual. Essa por sua vez é escrita em código nativo do sistema operacional e é ela que irá se comunicar com o sistema, traduzindo os comandos vindos do código fonte.

Esse comportamento da JVM é o que permite o princípio da portabilidade da linguagem, uma vez que o código fonte é escrito em Java e usado em vários sistemas operacionais e somente a JVM precisa ser específica. A JVM por si só não é somente uma máquina virtual que interpreta código fonte, mas ela também realiza execução de pilhas, gerenciamento de memória, threads e etc.

- JDK - Java Development Kit

É um conjunto de bibliotecas e utilitários que permite a criação de aplicações para a plataforma Java. Disponibilizado pela Oracle, contém todo o ambiente necessário para a criação e execução - através da JRE contido na JDK - dos aplicativos Java, sendo composta também pelo compilador e por todas as API's e assim que terminada a instalação já é possível desenvolver na linguagem.

- JRE - Java Runtime Environment

É uma implementação da máquina virtual Java que permite a execução das aplicações Java em qualquer ambiente de diferentes sistemas operacionais, desde que a JRE seja para aquele sistema operacional. Isso permitido pela interpretação dos Byte codes gerados pelo compilador Java.

3. **Para começarmos desenvolver em Java, precisamos de muito pouco. Basta um editor de texto, uma outra coisinha que você aprendeu na questão de número 2 e um terminal. Crie uma classe Java utilizando um editor qualquer. Esta classe deve ter um método main que imprima a frase "Hello World!", para começarmos com o pé direito esta aventura de programação. Agora faça uma explanação sobre:**
    - **O que você precisou para resolver esta questão;**
    - **O comando que você usou para compilar;**
    - **O comando que você usou para executar;**
    - **Qual é a estrutura miníma de uma classe Java;**
    - **O que é o método main e qual a sua importância;**
    - **O que é e em que momento da resolução da questão foi gerado o BYTECODE;**

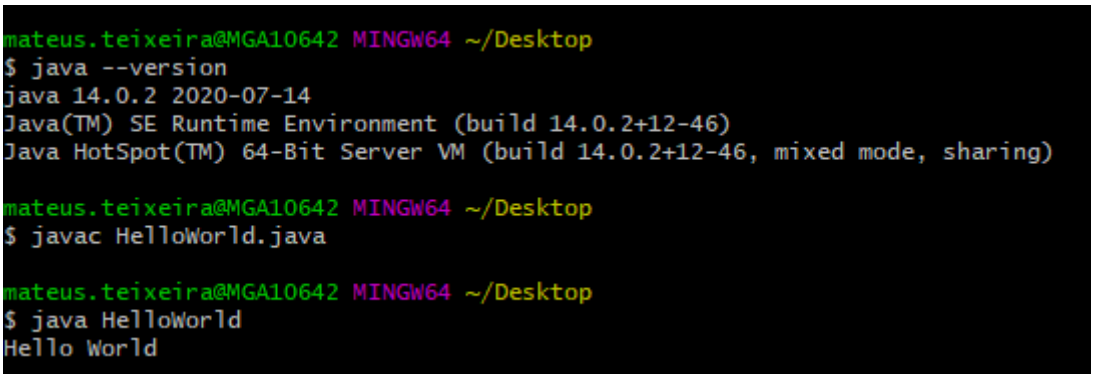
Resposta:

Inicialmente precisei instalar a JDK, para ter todos as ferramentas disponíveis para desenvolvimento em Java, além do ambiente para rodar a aplicação. Após escrever o fonte para escrever em tela rodei o comando javac "nome.java", isso gerou o .class, o meu código fonte compilado, que quando executado é interpretado pela JVM. A execução se da pelo comando java "nome".

A estrutura mínima para uma classe Java é a classe por si só, sendo ela de acesso público e contendo um método main. Que é o método de entrada de execução de qualquer aplicação Java. Justamente por ser o método de entrada da execução da aplicação, é o método que a JVM procura para executar de início, o método main precisa ser público, estar visível para quem acessa de um escopo fora daquela classe/pacote, além disso ele precisa ser estático garantindo que haverá somente uma referência para aquele método, não retornar nenhum valor, ou seja, ser void, e por último receber os argumentos da aplicação. Além do nome, que deve ser "main".

A geração dos bytecodes se da quando executo o comando javac, ou seja, quando compilo o código fonte. O bytecode é uma forma intermediária do código e é ela que será interpretada pela JVM.

Na imagem abaixo os comandos necessários para compilar o código fonte:



4. **Durante muito tempo, uma das maiores dificuldades na hora de programar era o gerenciamento de memória. Os desenvolvedores eram responsáveis pela sua alocação e liberação manualmente, o que levava a muitos erros e memory leaks. Hoje, em todas as plataformas modernas (incluindo Java), temos gerenciamento de memória automático através de algoritmos de coleta de lixo. Pesquise sobre Garbage Collector e faça uma explanação de como este algoritmo funciona na plataforma Java. Também implemente dois algoritmos em Java: um que exemplifique um possível erro de OutOfMemoryError e outro que mostre os cuidados tomados para não acontecer este tipo de erro durante o desenvolvimento de software.**

Resposta:

O Garbage Collector do Java é um processo que faz o gerenciamento automático de memória de uma aplicação. Durante a execução de uma aplicação Java alguns objetos ficam sem uso e não precisam ser mais guardados em memória, na Heap que é a porção de memória dedica ao programa, é neste instante que o Garbage Collector age, encontrando estes objetos, deletando-os e liberando memória.

A implementação do Garbage Collector é feita na JVM e cada JVM pode ter uma implementação específica dele, desde que siga a especificação da JVM. Como há várias implementações há várias maneiras que o algoritmo pode funcionar, mas basicamente se dá:

- O Garbage Collector identifica os objetos sem referência e marca-os para serem coletados;
- Então, os objetos marcados são deletados;

    - A marcação geralmente é feita pela ideia de "idade de um objeto", sempre seguindo a premissa que um objeto é "short-lived", ou seja, ele terá sua vida curta, sendo criado e logo após deixará de existir;
    - A divisão de idade de um objeto é feita de três maneiras:
        - Young Generation: Objetos recém-criados;
        - Old Generation: Objetos que passaram por um ciclo do Garbage Collector sem terem sidos coletados;
        - Permanent Generation: Objetos que são metadatas.
    - A deleção é feita pelas idades e pode ser feito em serial, paralela, CMS - Concurrent Mark Sweep - múltiplas threads são usadas para um coletor menor usando o mesmo algoritmo da estratégia paralela, e G1 (Garbage First).
- Após essa deleção, a memória pode ser compactada fazendo com que os objetos que restaram fiquem em um bloco contínuo de memória no topo da heap. Isso faz com que a alocação de memória para novos objetos seja mais fácil

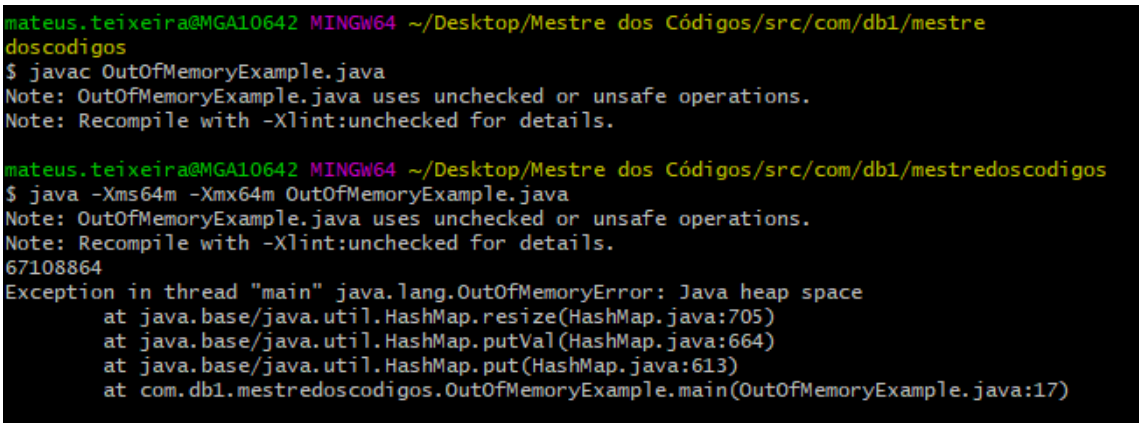
Um bom exemplo de código que gera OutOfMemoryError pode ser visto no arquivo OutOfMemoryExample.java. Neste fonte é utilizado uma estrutura chave-valor, no caso um Map, para armazenar um objeto, como key, que é instanciado toda vez e guarda um número inteiro e como valor uma String. É utilizado dois laços de repetição, um infinito e um iterado até 10000.

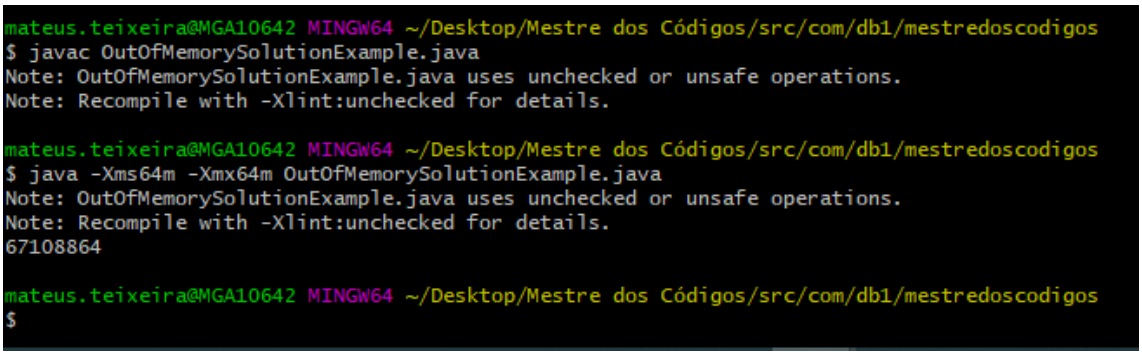
Para cada interação é criado um novo objeto e colocado no Map, caso seja satisfeita a condição do objeto já não estar contido no Map. Nesse fonte há uma falha de codificação que irá causar o OutOfMemoryError, que é a falta do método "equals" do objeto Key. Isso faz com que quando comparado dois objetos Key instanciados, seja comparado a posição em memória dos objetos e como sempre é diferente sempre irá adicionar no Map até que cause o OutOfMemoryError.

Como pode ser visto, há o lançamento do erro. O fonte é executado passando Xms64m e Xmx64m para diminuir a memória da JVM para o erro acontecer de forma mais rápida. Na execução é possível ver que a memória foi setada com os argumentos passados.

No fonte OutOfMemoryErrorSolutionExample.java é o mesmo fonte, porém na classe Key foi implementado o método equals, sobrescrevendo-o, assim agora quando dois objetos Key's serem comparados não será mais comparado a posição de memória dos objetos e sim o número inteiro que o objeto guarda. Isso fará com que o Map contenha somente 10000 objetos dentro dele e não mais que isso, assim prevenindo um OutOfMemoryError.

Nas imagens abaixo pode ser visto a execução primeiramente do código fonte que lança a exceção e logo em seguida a solução para o problema.





Na execução do fonte foi passado parâmetros de memória da JVM, isso para a execução via linha de comando. Para ser feito o mesmo procedimento no IntelliJ, IDE que foi desenvolvido os desafios, basta ir em Run > Edit Configurations > Modify options > Add VM Options e por como parâmetro: -Xms64m e -Xm64m.

✓ Conhecendo a plataforma

▪ Trabalhando com tipos de dados

1. **O Java possui vários tipos de dados primitivos para resolvermos diversos problemas e, para cada tipo primitivo, existe uma classe WRAPPER. Pontue:**
    a. **Diferença entre os tipos primitivos e as classes Wrapper:**
    b. **Quais são os tipos primitivos e suas classes Wrapper:**
    c. **Qual o valor default de cada tipo primitivo e de cada classe Wrapper:**
    d. **Como consigo converter cada tipo primitivo em uma classe Wrapper e como consigo converter cada classe Wrapper em um tipo primitivo;**

e. **Um exemplo de utilização de cada tipo durante o desenvolvimento de software:**

Respostas:

1.a:

Os Wrappers, como o próprio nome sugere, é um embrulho, ou um empacotamento, dos tipos primitivos para classes para que possam ser usados como objetos. Essa mudança veio a partir do Java 5 e facilitou muito a vida dos desenvolvedores pois tornando os primitivos como classes foram adicionados métodos para manipular os dados dentro das próprias classes. Como adicionar números inteiros, por exemplo.

Essa diferença de uma classe wrapper ser um objeto faz com que os objetos das classes Wrappers apontem para um endereço de memória, diferentemente do primitivo que aponto para o valor em si.

Isso faz com que não possamos, por exemplo, comparar wrappers com o operador "==", pois serão comparados os endereços de memória dos objetos, com exceção do tipo Integer que do range de +- 127 pode-se fazer essa comparação, depois disso todas as comparações serão falsas.

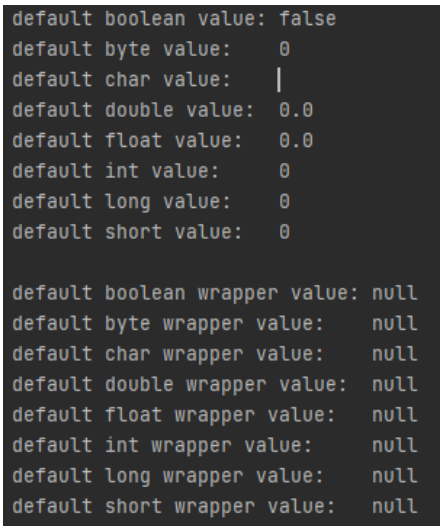
1.b:

São oito os Wrappers existentes, todos do pacote java.lang.*, e que são diferenciados com a primeira letra sendo maiúscula (com exceção do char que vira Character e do int que vira Integer), são eles:

Primitivo	Wrapper
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

1.c:

Na imagem abaixo estão os valores default para cada tipo.



Para obter estes valores fiz e rodei o código fonte presente em "src/main/java/knowingtheplatform/workingwithdatatypes/wrappersanddefaultvalues/DefaultValues.java".

1.d:

Todo o processo é feito automaticamente pelo Java, através dos processos de autoboxing e unboxing. O autoboxing é o processo de conversão do tipo primitivo para seu wrapper equivalente, e o unboxing é o processo contrário.

- Exemplo de autoboxing:
  - Integer integer = 3;
- Exemplo de unboxing;
  - int aInt = 0;
    aInt = new Integer(3);

1.e:

Código em:

"src/main/java/knowingtheplatform/workingwithdatatypes/wrappersanddefaultvalues/AutoboxingUnboxingExamples.java"

```java
package knowingtheplatform.workingwithdatatypes.wrappersanddefaultvalues;

import java.util.ArrayList;
import java.util.List;

public class AutoboxingUnboxingExamples {

    public static void main(String[] args) {

        Long l = 2L;
        Long aLong = Long.valueOf("3");

        AutoboxingUnboxingExamples autoboxingUnboxingExamples = new AutoboxingUnboxingExamples();

        List<Integer> integers = autoboxingUnboxingExamples.autoBoxingNumbers();
        System.out.println(integers);
        int sumNumbers = autoboxingUnboxingExamples.unBoxingNumbers(integers);
        System.out.println("\n"+sumNumbers);
    }

    private List<Integer> autoBoxingNumbers() {
        List<Integer> autoboxNumbers = new ArrayList<~>();
        for (int i = 0; i < 10; i++) {
            autoboxNumbers.add(i);
        }
        return autoboxNumbers;
    }

    private int unBoxingNumbers(List<Integer> numberToSum) {
        int sum = 0;
        for (Integer number : numberToSum) {
            sum += number;
        }
        return  sum;
    }

}
```

2. **Qualquer programa de computador precisa manter informações de alguma forma. Assim, as linguagens de programação permitem a criação de variáveis. Pontue:**
    a. **Declaração de variáveis com inicialização implícita:**
    b. **Declaração de variáveis com inicialização explícita:**
    c. **Obrigatoriedade de inicialização de variáveis no Java:**
    d. **Como funciona o escopo de variáveis no Java:**
    e. **Quando utilizar a palavra reservada final na declaração de uma variável**

Respostas:

2.a:

A declaração de variáveis com inicialização implícita acontece para aquelas variáveis, sejam primitivas ou objetos, que não tenham seus valores atribuídos, ou instanciação para os objetos, de forma explicita pelo desenvolvedor no código fonte. O Java irá atribuir os valores default para cada variável, porém, isso só acontece para as variáveis que estejam em escopo de classe e instância, ou seja, para variáveis que estejam dentro de um escopo local, de bloco, não ocorre essa inicialização implícita e ocorre um erro de compilação.

2.b:

A declaração de variáveis com inicialização explícita acontece explicitamente pelo desenvolvedor na escrita do código fonte e pode ser feita em qualquer escopo de variável, porém é obrigatório para escopos locais.

2.c:

A obrigatoriedade da inicialização de variáveis no Java se dá somente em variáveis que tenham escopo local, ou seja, estão dentro de um método, ou um laço de repetição por exemplo, e serão usadas dentro deste bloco. Caso contrário ocorrerá a inicialização implícita.

2.d:

O escopo de variável do Java é onde a variável terá sua "vida", é onde a variável é declarada, é onde ela será referenciada e usada. Por exemplo, se uma variável for declarada dentro de um if ela somente será conhecida dentro deste if.

Escopos:

- Escopo local: É um escopo limitado a um bloco de código, treco que pode se entender dentro de chaves. Como por exemplo, métodos, ifs, construtores, loops;
- Escopo de instância: É um escopo que existe enquanto uma instância da classe onde a variável foi declarada existir. Pertencem a este escopo variáveis de classe.
- Escopo de classe: É o escopo que abrange as variáveis de classes estáticas, sendo assim não precisando de uma instanciação, e que são compartilhadas entre todas as instâncias da classe. As variáveis estáticas são carregadas

com o programa junto com a classe quando a classe é carregada em memória;

- Shadowing: É uma técnica que o Java usa para poder decidir qual variável usar quando duas variáveis tem o mesmo nome e estão em escopos diferentes. Com a técnica será sempre usado de menor escopo.

2.e:

Quando queremos fazer com que nossas variáveis passem a ser constantes, ou seja, após inicializado não poderá ter o valor alterado e qualquer tentativa de alteração ocorrerá um erro de compilação. A utilização pode acontecer diretamente na declaração da variável, mas também pode acontecer nos construtores das classes, esses tipos de variáveis são chamadas de "blank FINAL variable".

Quando aplicado a atributos primitivos permanecem com seus valores constantes e para objetos apenas as referências de memórias permanecem constante.

3. **Um dos tipos mais utilizados na plataforma Java é o tipo String, com S maiúsculo. Strings em Java são objetos ou instâncias da classe java.lang.String que devem ser declaradas e instanciadas. Vamos, então, concentrar nossa energia no estudo deste tipo tão importante. Crie exemplos utilizando String mostrando:**

    a. **Como criamos um objeto do tipo String;**
    b. **Como podemos concatenar uma String;**
    c. **Quais os principais comportamentos de uma String;**
    d. **Como trabalhamos com String utilizando as classes StringBuilder e StringBuffer, pontuando a diferença entre elas e qual a importância de usá-las;**
    e. **Como transformamos String em outro tipo de dado;**
    f. **Como podemos transformar qualquer tipo de dado em uma String.**

Respostas:

3.a:

A criação de uma String se dá com qualquer texto que é posto entre aspas duplas - tornando o texto entre aspas funcional, inclusive com métodos disponível para manipulação -, não requerendo obrigatoriamente o operador new para instanciar um objeto String e sendo semelhante a um objeto primitivo.

Lembrando que o tipo String é um objeto que terá uma referência em memória. O operador new pode até ser usado porém ao utiliza-lo estamos criando um novo objeto e pode ocorrer do mesmo texto existir mais de uma vez em memória. Uma vez que uma String é um objeto, uma variavel do tipo String também aceita valores null.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithstrings/CreateAStringExamples.java”

3.b:

A concatenação pode ocorrer de duas maneiras:

- Utilizando o operador de adição entre duas String.
  Exemplo: String fullName = firstName + lastName;
- Utilizando o método concat, disponibilizado pela classe String.
  Exemplo: String fullName = firstName.concat(lastName);

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithstrings/ConcatAStringExamples.java”

3.c:

Algum dos principais são:

- equals;
- equalsIgnoreCase;
- compareTo;
- compareToIgnoreCase;
- concat;
- valueOf;
- length;
- chartAt;startsWith e endsWith;
- subString;
- replace;
- toUpperCase e toLowerCase;
- trim;

3.d:

As Strings são imutáveis, ou seja, não se pode mudar o seu valor apor a primeira atribuição. Por esse motivo, quando concatenamos uma String a outra nós não estamos alterando o valor do texto dentro daquela referência da String e sim estamos criando um novo objeto, com um novo valor em memória, que conterá o texto anterior com o novo texto concatenado, e o objeto que continha o texto antigo permanecerá existindo em memória, porém sem uso.

Para evitar isso usamos as classes StringBuilder e StringBuffer. Ambas são mutáveis, ou seja, para cada concat (ou append para a classe) nós de fato concatenamos um novo valor a String existente, sem criar um novo objeto em memória.

A diferença entre a StringBuilder e a StringBuffer é que a StringBuffer é sincronizada, sendo própria para trabalhar em ambientes com várias threads para garantir a consistência do código nesse ambiente, por exemplo.

3.e:

Basta utilizarmos o método estático parseXXX da classe Wrapper de destino, passando a String como parâmetro, quando quisermos retornar para um tipo primitivo, exemplo:

- String para int:

  ```
  String aStringNumber = "30";
  int aStringConvertedToInt = Integer.parseInt(aStringNumber);
  ```

- String para long;

  ```
  String aStringNumber = "30";
  long aStringConvertedToLong = Long.parseLong(aStringNumber);
  ```

Outra opção é o método estático valueOf da classe Wrapper de destino, passando a String como parâmetro, quando quisermos retornar para um Wrapper, exemplo:

```
String aStringNumber = "30";
Long aStringConvertedToLongWrapper = Long.valueOf(aStringNumber);
```

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithstrings/ConvertFromString.java”

3.f:

Basta utilizarmos o método estático toString da classe Wrapper de origem do dado, exemplo:

```
int intToConvert = 4;
String aStringFromInt = Integer.toString(intToConvert);
```

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithstrings/ConvertToString.java"

4. **Trabalhar com sistemas que manipulam valores decimais é uma tarefa crítica e que deve exigir o máximo de atenção do desenvolvedor. Trabalhar com Double pode trazer uma certa dificuldade quando precisamos de precisão em operações matemáticas de valores decimais. Uma alternativa para resolver esse problema é utilizar a classe BigDecimal, que trabalha com pontos flutuantes de precisão arbitrária, conseguindo estipular o nível de precisão do valor. Diante disso, vamos aprender a trabalhar com esta classe. Faça exemplos de algoritmo que:**
    a. **Converta uma String para BigDecimal;**
    b. **Converta um Double para BigDecimal;**
    c. **Execute as operações matemáticas de subtração, adição, divisão, multiplicação e potência de números decimais, demostrando diferentes estratégias de arredondamento em todas as operações;**
    d. **Faça a comparação entre 2 objetos to tipo BigDecimal.**

Respostas:

4.a: e 4.b

A conversão ocorre através da utilização do construtor do BigDecimal, que recebe uma String que é o valor que se deseja obter. É indicado o uso de String pois mantém a precisão do número informado, como mostrado no exemplo do código fonte.

A conversão também pode ser feita via construtor, porém não é indica pois há perda de precisão na conversão.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithbigdecimal/ConvertToBigDecimal.java”

4.c:

As operações com BigDecimal podem ser arredondas e para isso existem as seguintes estratégias:

- CEILING: Arredonda em direção ao infinito positivo;
- FLOOR: Arredonda em direção ao infinito negativo;
- UP: Arredonda para cima, longe do zero;
- DOWN: Arredonda para baixo, para perto do zero;
- HALF_UP: Arredonda para o vizinho mais próximo, se for equidistante arredonda para cima;
- HALF_DOWN: Arredonda para o vizinho mais próximo, se for equidistante arredonda para baixo;
- HALF_EVEN: Arredonda para o vizinho mais próximo, se for equidistante arredonda para o vizinho par;
- UNNECESSARY: Não é aplicado arredondamento, porém se nenhum resultado exato for possível é lançada uma ArithmeticException;

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithbigdecimal/BigDecimalMath.java”

4.d:

A comparação entre 2 objetos BigDecimal pode ser feito pelo método compareTo que a classe disponibiliza. Esse método retorna três possíveis valores. -1, 0, e 1 que significa que o valor é menor, igual ou maior, respectivamente, ao valor comparado. Há outra maneira de comparar dois BigDecimal que é através do método equals, esse por sua vez só retornará TRUE caso o valor dos dois BigDecimal forem iguais, assim como a escala deles, em qualquer outro caso retorna falso.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithbigdecimal/CompareBigDecimal.java”

5. **No Java 8 foi introduzida uma nova API para manipulação de datas e horas. Vamos entender quais classes e métodos foram incluídos. Faça uma pesquisa sobre as classes listadas abaixo e mostre, por meio de algoritmos, onde cada uma pode ser usada e como podemos criar objetos de datas, manipular datas, extrair partes de datas, realizar comparações entre datas, alterar datas e converter entre os diversos tipos de datas listadas:**
    a. **LocalDate;**
    b. **LocalTime;**
    c. **LocalDateTime;**
    d. **MonthDay;**
    e. **YearMonth;**
    f. **Period.**

Respostas:

5.a:

A classe LocalDate é comumente usada quando queremos manipular datas sem necessariamente considerar as horas, minutos e por assim adiante. Ou seja, não é indicado quando necessitamos saber exatamente quando uma ação no sistema aconteceu, por exemplo, com exatidão de milésimos de segundos. Portanto quando necessitamos somente gravar o dia de algo, como a data de nascimento de alguém, a LocalDate é indicada.

Ela possui duas maneiras básica de inicialização, que é através do método estática now(), que captura a data do sistema, ou através do também método estático of() que é passado ano, mês e dia que deseja-se ter a data. A classe possui diversos métodos para formatação, manipulação, extração e comparação entre datas, conforme exemplos no código. Um objeto LocalDate é imutável e thread-safe.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithLocalDate.java”

5.b:

A classe LocalTime é usada para obter a hora com exatidão de nano segundos. É indicado, portanto, para aplicações que desejam salvar somente a hora que algo aconteceu e que precisa manipular esses dados. Também possui duas maneiras de

inicialização, o método now() e of(), este último passando-se a hora, minuto e segundos. Um objeto LocalTime é imutável e thread-safe.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithLocalTime.java"

5.c:

A classe LocalDateTime é usada para ter a informação de data e hora exatas. Ela fornece com exatidão de nano segundos o momento que ela é usada. É interessante o uso dela para aplicações que precisam ter esse tipo de exatidão salva em algum momento, ou necessite algum outro tipo de uso nesse sentido. Um objeto LocalDateTime é imutável e thread-safe.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithLocalDateTime.java"

5.d:

A classe MonthDay fornece um objeto de dia-mês imutável, que representa a combinação de mês com o dia do mês. Também é possível obter através do método estático now() e do of(), passando mês e dia. É indicada para ser usada quando necessitamos de operações de validação de ano, obtenção de meses e dias como número inteiro, range de meses do ano ou de dias do mês. Qualquer data que é derivada de mês e dia pode ser obtida, como um quarto de ano.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithMonthDay.java"

5.e:

A classe YearMonth fornece um objeto de mês-ano imutável, que representa a combinação de mês com ano. Também é possível obter através do método estático

now() e do of(), passando mês e ano. Possui as mesmas indicações de uso da classe MonthDay.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithYearMonth.java"

5.f:

A classe Period é usada para medir tempo em anos, meses e dias, tais como diferença entre datas.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithPeriod.java"

A conversão entre os tipos de data que o Java oferece está na classe:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwithdates/ConvertDates.java"

6. **É comum a necessidade de armazenamento de variáveis em memória sequencial. O Java permite esta implementação com o uso de arrays. Mostre, por meio de algoritmos, utilizando no mínimo dois tipos primitivos e dois tipos não primitivos (classes wrappers), como podemos:**
    a. **Declarar um array;**
    b. **Inicializar;**
    c. **Acessar posições;**
    d. **Percorrer um array.**

Respostas:

6.a:

Os arrays são containers para objetos de um único tipo com tamanho limitado. Este tamanho é definido na sua criação. Cada objeto é acessado via um índice, índice esse que começa em 0 e vai até n-1, sendo n o tamanho do array. Quando declarado indicamos ao Java o tamanho do array e em cada posição já é setado um valor padrão. Sendo 0 para tipos primitivos, false para booleanos e null para objetos.

6.b:

Para inicializar um array podemos utilizar uma lista de valores separados por vírgula dispostos entre chaves ou podemos percorrer cada posição colocando o elemento desejado na posição.

6.c:

Podemos acessar posições do array utilizando o operador [i], em que i indica a posição do elemento que desejamos acessar. Lembrando que a posição se da por n-1, em que n é o tamanho do array. Ou seja, um array de 10 posições é acessado pelos índices de 0 até 9.

6.d:

Para percorrer um array podemos utilizar qualquer laço de repetição combinado com os índices, em que o laço deve percorrer o índice de 0 até n-1. Há a possibilidade também do uso de um laço de repetição aprimorado, que serve para obter os elementos do array, porém sem a possibilidade de alterá-los.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithdatatypes/workingwitharrays/WorkingWithArrays.java"

▪ Trabalhando com laços de repetição

7. **Quando estamos desenvolvendo um software, por vários momentos surge a necessidade de executar uma parte do código várias vezes, como uma repetição. O Java oferece alguns tipos de laços de repetição para o programador escolher, então pesquise sobre o assunto e:**
    a. **Faça um algoritmo demonstrando o funcionamento dos laços de repetição while, for, enhanced for e do/while;**
    b. **Faça uma comparação entre os tipos de laços;**
    c. **Demostre por meio de um algoritmo o funcionamento do break e do continue em laços de repetição.**

Respostas:

7.a e 7.b:

Os laços de repetição são features que facilitam a execução de instruções em que se há a necessidade de repetir por várias vezes, muitas das vezes enquanto uma condição for verdadeira. O Java provê 4 formas básicas para isso, que são: for, for melhorado (enhanced for), while e o do-while.

Sobre eles:

- While: é uma instrução de controle de fluxo que permite que um treco de código seja executado enquanto uma condição seja verdadeira. A instrução começa já chegando a condição e somente será executado o código dentro do loop se for verdadeira, e é por esse comportamento que a instrução While é conhecida como uma instrução de loop de controle de entrada. Após ocorrer a avaliação da condição e esta sendo verdadeira é executado todo o código dentro do loop e a cada iteração a condição é reavaliada, quando falsa o loop termina. Caso a condição seja falsa na primeira iteração, na entrada, o código não é executada nem uma única vez;
- Do-While: é similar ao while, mas a diferença é que faz a verificação de condição para continuar no loop após executar o trecho de código que esta dentro de seu bloco. Ou seja, o do-while executa, sempre, no mínimo uma vez o código.
- For: é uma maneira mais concisa de escrever a estrutura do loop. A própria instrução descreve a inicialização, condição e incremento de um índice. É normalmente usado quando se sabe exatamente quantas vezes o loop irá passar por um código, como por exemplo irá percorrer 10 vezes um código em que essas 10 vezes é o tamanho de um array. A estrutura se dá: for (instrução1; instrução2; instrução3) {código} em que: instrução 1 define a condição inicial para inicializar o loop, a instrução 2 é o teste para continuar o loop, e a instrução 3 é o incremento/decremento da variável de controle.
- Enhanced for: É uma maneira mais simples de iterar por meio de elementos de uma estrutura. É usado quando não se sabe o índice, ou tamanho, dos elementos da estrutura e além disso o objeto que está na iteração é imutável, ou seja, o objeto da iteração não pode ser alterado.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithloops/WorkWithDoWhile.java”

“src/main/java/knowingtheplatform/workingwithloops/WorkWithFor.java”

“src/main/java/knowingtheplatform/workingwithloops/WorkWithWhile.java”

7.c:

- Break: é um comando usado em laços de repetição que causa a interrupção imediata do laço quando usado e continuando o fluxo do código fonte após o loop.
- Continue: é um comando usado em laços de repetição que volta para o início do laço, não causando a interrupção, porém não executando o que vem após o comando.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithloops/WorkWithBreakAndContinue.java"

▪ Trabalhando com estruturas de dados

8. **Trabalhar com estrutura de dados utilizando arrays pode ser uma tarefa difícil, pois arrays não permitem o redimensionamento. É impossível buscar um determinado elemento cujo índice é desconhecido, não conseguimos saber quantas posições do array foram populadas sem criarmos métodos adicionais para isso, entre outros problemas. Assim, para facilitar a vida do programador, foi criado um conjunto de interfaces e classes conhecidas como Collections Framework, que se encontra no pacote java.util. Veja o diagrama de classes abaixo e:**
    a. **Demonstre, por meio de algoritmos, a utilização de cada uma delas, ressaltando qual o tipo de problema cada uma resolve e qual a importância de fazer a escolha certa.**

Respostas:

- Interfaces:
    - Iterator: possibilita percorrer as coleções e remover seus elementos;
    - Collection: é a interface que está no topo da hierarquia das Collections e não há uma implementação direta desta interface, porém é onde está contida as definições das operações básicas para as coleções;
    - Set: é a interface que define uma coleção em que não é permitido a inclusão de elementos que já existem na coleção;
    - List: é interface que define uma coleção permitindo elementos duplicados. Permite, também, o controle sobre a posição de cada elemento que é inserido, assim como o acesso através do seu índice.

    - Queue: interface que define uma coleção que mantem lista de prioridades, onde a ordem dos seus elementos determina essa prioridade.
    - SortedSet: é uma extensão da interface Set que possibilita a classificação natural dos elementos;
    - NavigableSet: interface que define coleções que são Set, já que deriva de Set, que permite a navegação pela estrutura, por exemplo, navegando na ordem inversa.
- Implementações:
    - HashSet: implementa a interface Set. Não garante a ordenação dos dados nela inserida, porém garante que não haja elementos repetidos. É indicada para aplicações que exijam acesso rápido aos dados e que esses dados não sejam repetidos.
    - LinkedHashSet: extende a HashSet. Possui uma lista duplamente ligada dos seus elementos que faz com que seus elementos sejam iterados na ordem em que foram inseridos.
    - TreeSet: implementa a interface NavigableSet. Possui ordenação nos dados, porém é mais lento que a HashSet. Indicada no mesmo cenário da HashSet porém com os dados ordenados.
    - ArrayList: implementa a interface List e é semelhante a um array, porém permite que seu tamanho cresça conforme necessário. Toda vez que há a necessidade seu tamanho aumenta em 50%, causando um custo alto nessa operação já que o array inicial é copiado para a criação do novo. Tem a busca por elementos rápida, porém as inserções e exclusões são lentas, apresentando tempo linear de crescimento conforme o tamanho do arrayList cresce. É indica quando se precisa acesso rápido aos elementos, que pode ser feita através do índice do elemento.
    - Vector: implementa a interface List. É muito semelhante ao ArrayList, tendo a utilização igual, porém a diferença se da pois a classe Vector é sincronizado enquanto a ArrayList. Ou seja, a Vector é thread-safe e é muito indicada para aplicações que terão acessos concorrentes a lista. Outra diferença muito importante é na alocação dinâmica da Vector, enquanto a ArrayList aumenta 50% o tamanho do array para alocar novas posições a classe Vector aloca o dobro do já existente, tornando-a muito mais cara em termos de consumo de memória.
    - LinkedList: implementa a interface List e Queue e é uma lista linkada, ou seja, cada nó contém além do elemento uma referência de memória ao

próximo nó. Isso faz com que a exclusão e inserção seja rápida, já que sabemos as posições dos elementos. Essa característica de rapidez na exclusão e inserção faz com que a implementação seja indicada em momentos que precisamos de muitas exclusões e inserções de elementos.

- PriorityQueue: implementa a interface List e Queue. É uma Collection que trata os objetos para serem processados baseados em alguma prioridade. Normalmente as filas seguem o algoritmo FIFO (First In, First Out), ou algum do tipo, porém a classe PriorityQueue é baseado na prioridade de alguma classificação, ou ordenação, passada para a classe na adição de algum elemento. É a implementação que fornece tempo de acesso, adição e remoção mais rápidos.

- Fazendo escolhas:

Para o desenvolvimento de uma aplicação em que necessite guardar e manipular grandes volumes de dados podemos usar as Collections que vieram para facilitar a vida do desenvolvedor. Porém, é essencial o conhecimento das principais características delas. Uma escolha errada pode tornar a aplicação muito lenta e pesada. É essencial, também, conhecer o que será manipulado das Collections para fazermos as escolhas certas. Por exemplo, se precisamos somente guardar informações de forma que iremos apenas adicionar na Collection os dados para posteriormente salvar em banco de dados podemos utilizar o LinkedList pois sua inserção e remoção são mais rápidas que o ArrayList, por exemplo. Agora, se necessitamosprocurar dados dentro de uma Collection podemos utilizar o ArrayList que será mais rápido. Outro ponto que devemos observar é a necessidade de ordenação dos dados, podemos utilizar a implementação TreeSet porém a inserção será mais lenta, mas ainda assim compensa a posterior ordenação. Ou seja, conhecer as Collections e as características dos dados que iremos utilizar nelas é essencial para a aplicação ser performatica e não conter erros na utilização das Collection.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithArrayList.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithArrayList.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithHashSet.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithLinkedHashSet.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithLinkedList.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithPriorityQueue.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithTreeSet.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithcollections/WorkWithVector.java”

“src/main/java/knowingtheplatform/workingwithstructures/WorkWithStructuresHelper.java”

9. **O Java fornece uma segunda forma de trabalhar com estrutura de dados tão importante quanto a primeira. São as classes e interfaces relacionadas a trabalhar com mapas. Veja o diagrama de classes e:**
    a. **Demonstre, por meio de algoritmos, a utilização de cada uma delas, ressaltando qual tipo de problema cada uma resolve e qual a importância de fazer a escolha certa.**

Respostas:

Os objetos que são Map no Java confia a utlização dos dados no algoritmo de hash code, em que o hash transforma uma grande quantidade de dados em uma pequena quantidade, baseando a sua busca na construção de índices. A interface Map não é um subtipo de Collection, sendo assim tem seu uso diferente.

- Interfaces:
    - Map: É a interface do topo deste tipo de estrutura. Mapeia valores para chaves, ou seja, através de uma chave é possível se chegar a um valor na estrutura armazenado. Essa chave não pode ser repetida, porém caso haja chave repetida é sobrescrito o valor pela última chamada. Os valores podem se repetir;
    - SortedMap: estende de Map e provê operações de ordenação para as suas implementações;

    - NavigableMap estende SortedMap e provê métodos convenientes de navegação pela estrutura. Como por exemplo, pegar keys acima de certo valor.
- Implementações:
    - HashMap: implementa a interface Map. É a mais utilizada no desenvolvimento de aplicações. Seus elementos não são ordenados e é rápida na busca, inserção e exclusão de dados. Na sua inserção não garante a ordem de inserção dos elementos e irá, se necessário, mudar a ordem a cada nova inserção. Permite uma única chave nulla, porém vários valores nullos e não há repetição de elementos. Para acessar um valor é preciso conhecer sua key;
    - HashTable: implementa a interface Map. Tem seu funcionamento semelhante a HashMap, em termos de chave-valor. Porém, é uma implementação sincronizada - thread safe -, ou seja, indicada para aplicações que tenham acessos concorrentes a estrutura. Possui eficiente pesquisa e não aceita chaves nullas.
    - LinkedHashMap: extende HashMap. Possui as mesmas características da HashMap, porém esta implementação mantém a ordem de inserção das key-values.
    - TreeMap: implementa a interface NavigableMap. Não aceitam valores de chave-valor nulos e mantém seus elementos ordenados na inserção, tornando a operação de inserção mais lenta que o restante das implementações. Ao contrário das demais, exceto HashTable, a TreeSet não aceita chaves nulas.
- Fazendo escolhas:

Assim como as Collections a escolha do Map certo também é muito importante na hora de desenvolver uma aplicação. Conhecer os tipos de dados que serão salvos também é parte importante para que seja extraída o máximo das vantagens dos Maps. Como por exemplo, uma TreeSet gasta mais tempo para inserir dados pois ela insere os elementos ordenando-os, porém se não há a necessidade disso será tempo e computação gastos sem necessidade. Por isso conhecer o conjunto como um todo é muito importante.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithstructures/workingwithmaps/WorkWithHashMap.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithmaps/WorkWithHashMap.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithmaps/WorkWithHashMap.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithmaps/WorkWithHashMap.java”

“src/main/java/knowingtheplatform/workingwithstructures/workingwithmaps/WorkWithTreeMap.java”

▪ Trabalhando com POO

10. Um dos pilares da POO é o encapsulamento. Fazemos isso por meio da criação de classes, atributos e métodos, definindo uma restrição de acesso utilizando os modificadores de acesso. Então, demonstre por meio de algoritmos:

a. Criação de métodos com argumentos e valores de retorno;

b. Criação de métodos sem argumentos e valores de retorno;

c. A aplicação da palavra-chave static em métodos e atributos, ressaltando a principal característica de um método static e um atributo static;

d. A sobrecarga de métodos;

e. A criação de construtores padrão, construtores com argumentos e a sobrecarga de construtores;

f. A aplicação de encapsulamento utilizando os modificadores de acesso.

Respostas:

10.a:

```java
public List<String> generateRandomNames(int quantityOfNames) {

    Random rand = new Random();
    List<String> names = new ArrayList<>();
    for (int i = 0; i < quantityOfNames; i++) {
        int length = rand.nextInt( bound: 5) + 5;
        StringBuilder builder = new StringBuilder();
        for (int j = 0; j < length; j++) {
            builder.append(LEXICON.charAt(rand.nextInt(LEXICON.length())));
        }
        names.add(builder.toString());
    }
    return names;
}
```

10.b:

```java
protected void addAndRunInAnArrayList(List<String> names) {
    List<String> arrayOfStrings = new ArrayList<>();

    long initTime = System.currentTimeMillis();
    for (String name : names) {
        arrayOfStrings.add(name);
    }
    long finalTime = System.currentTimeMillis();
    System.out.println("Time to add 200k Strings in ArrayList: " + (finalTime - initTime));

    initTime = System.currentTimeMillis();
    for (int i = 0; i < names.size(); i++) {
        if (arrayOfStrings.get(i).equals("Mateus")) {
            break;
        }
    }
    finalTime = System.currentTimeMillis();
    System.out.println("Time to find element Mateus in ArrayList: " + (finalTime - initTime));

    initTime = System.currentTimeMillis();
    arrayOfStrings.remove( index: 92032);
    finalTime = System.currentTimeMillis();
    System.out.println("Time to remove by index in ArrayList: " + (finalTime - initTime));

    initTime = System.currentTimeMillis();
    arrayOfStrings.remove( o: "Mateus");
    finalTime = System.currentTimeMillis();
    System.out.println("Time to remove by object in ArrayList: " + (finalTime - initTime));
}
```

10.c:

Os métodos static são métodos que podem ser chamados sem a instanciação de uma classe. São referenciados pelo nome da classe, ou de um objeto. Estes métodos são guardados na Permanent Generation da memória Heap. Outro ponto importante é que um método static pode ser compartilhado por todos os objetos criados a partir da mesma classe, já que o método está associado a classe. Com essa característica de ser compartilhado os métodos static são indicados a ter código que possa ser compartilhado, além disso, outra indicação de uso é quando necessitamos acessar outros campos estáticos da classe.

Os atributos static também são comuns as instâncias da classe, basicamente é criado uma única variável e essa é compartilhada por todos os objetos da classe. A alocação de memória para o atributo ocorre quando a classe é carregada.

10.d e 10.e:

A sobrecarga de métodos permite que o desenvolvedor crie métodos que tenham um mesmo nome, porém recebendo parâmetros diferentes. Isso evita com que criemos métodos diferentes, com nomes diferentes, para cada necessidade que tivermos.

```
public Customer(String name, String address, String city, String state, String country) {
    this.name = name;
    this.address = address;
    this.city = city;
    this.state = state;
    this.country = country;
}

public Customer(String name) {
    this.name = name;
    this.country = "Brazil";
}

public Customer() {
    this.country = "Brazil";
}
```

10.f:

Os modificadores de acesso são palavras reservadas do Java que indica quem pode acessar um método ou atributo de uma determinada classe.

- Private: o modificador private indica que um método/atributo pode ser acessado apenas dentro da classe em que está declarado.

- Protected: o modificador protected indica que um método/atributo pode ser acessado por todos os membros que estão dentro do pacote em classe está.
- Public: o modificador public indica que um método/atributo pode ser acessado por qualquer método/classe de qualquer lugar do código.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithoop/workingwithoverload/Customer.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithoverload/WorkWithOverload.java”

11. Outros pilares da POO são a herança e o polimorfismo. Por isso, pesquise e demonstre, por meio de algoritmos:

a. Reaproveitamento de comportamentos de um objeto por meio da herança;
b. A utilização de classes abstratas, métodos abstratos e interfaces;
c. A utilização de métodos default em interfaces;
d. A utilização das palavras reservadas this e super;
e. Desenvolva um código que mostre o uso do polimorfismo com herança de classes e a implementação de interfaces.

Respostas:

11.a:

A herança na Programação Orientada a Objetos permite que uma classe filha herde características da classe pai, assim como o mundo real. As características que podem ser herdadas são os atributos e métodos da classe pai, desde que não esteja restringido o acesso pelo modificador "private". A grande vantagem da herança é o reaproveitamento de código, esse reaproveitamento se dá quando é identificado que um atributo, ou um método, pode ser igual para classe pai e para a filha.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithoop/workingwithinheritance/Developer.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithinheritance/Employee.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithinheritance/Manager.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithinheritance/WorkWithInheritance.java”

11.b:

- Classes abstratas: servem como modelos para classes que as herdam. Possui a característica de não poder ser instanciadas, sendo assim para se ter um objeto de uma classe abstrata é necessário ter uma classe mais especializada que ela, que irá estender a classe abstrata e então terão os métodos sobrescritos nas classes filhas.
- Métodos abstratos: são métodos que não possuem implementações na classe em que são declarados e que possuem implementação obrigatória, por meio da sobrescrita, nas classes que herdam a classe pai em que o método está declarado.
- Interfaces: são padrões definidos através de contratos. Este contrato determina um conjunto de métodos que a classe que assina este contrato deverá implementar. As interfaces são 100% abstratas.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithoop/workwithabstract/Animal.java”

“src/main/java/knowingtheplatform/workingwithoop/workwithabstract/Dog.java”

“src/main/java/knowingtheplatform/workingwithoop/workwithabstract/WorkWithAbstracts.java”

11.c:

A utilização de métodos default nas interfaces é de grande importância pois como a interface é um contrato toda classe que assina esta interface tem que implementar os métodos nela declarados. Porém se uma interface possui muitas implementações a adição de um novo método faz com que tenhamos que modificar e adicionar as implementações dos métodos em todas as classes, e muitas vezes esse novo método será usado em poucas implementações. Com o default não é mais preciso implementar os métodos em todos os lugares que assinam a interface, reduzindo assim o esforço de programação, simplificando o código e alterando somente é necessário e será usado sem ser uma implementação default, que usará o fonte padrão que foi implementado dentro da interface.

Código Fonte:

“src/main/java/knowingtheplatform/workingwithoop/workwithinterfaces/Polygon.java”

“src/main/java/knowingtheplatform/workingwithoop/workwithinterfaces/Rectangle.java”

“src/main/java/knowingtheplatform/workingwithoop/workwithinterfaces/Triangle.java”

“src/main/java/knowingtheplatform/workingwithoop/workwithinterfaces/WorkWithInterfaces.java”

11.d:

- This: a palavra reservada this indica referência a instância atual de um objeto. Quando fazemos this.AlgumaVariavel quer dizer que queremos acessar o valor daquela variável da instância atual da classe;
- Super: a palavra reservada super indica referência a superclasse imediata.

11.e:

Código Fonte:

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/TV.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/TVFunctions.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/TVModel.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/TVModelA.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/TVModelB.java”

“src/main/java/knowingtheplatform/workingwithoop/workingwithpolymorphism/WorkWithPolymorphism.java”

▪ Trabalhando com exceções

12. Todos algoritmo está sujeito a erros. O Java fornece uma API para trabalhar com exceções, nos permitindo fazer o tratamento adequado de uma determinada situação e até mesmo criarmos nossas próprias exceções. Assim, faça:

a. Veja o digrama de classe abaixo, diferencie quais são as exceções Checked, Runtime e Erros e descreva a diferença de cada tipo:

b. Faça um algoritmo demonstrando o uso do bloco try-catch-finally;

c. Faça uma pesquisa e demonstre quando ocorrem as exceções: ArithmeticException, ArrayIndexOutOfBoundsException, ClassNotFoundException, IOException, IllegalArgumentException, InterruptedException e NullPointerException;

d. Faça um algoritmo implementando uma exceção customizada.

Respostas:

12.a:

Todas as exceções que herdam de Exception e que não sejam Runtime Exception, no caso do diagrama a IOException. As exceções que são checked são aquelas exceções em que o desenvolvedor é obrigado a tratar a exceção onde ocorreu o lançamento ou passar para o método em que chamou aquele fonte. As checked exceptions geralmente são usadas para erros recuperáveis. As exceções que são uma Exception geralmente indicam problemas que ocorrem em tempo de execução ou compilação. Os erros que são subclasses de Error - no caso do diagrama AWTError, OutOfMemoryError e ThreadDeath – são erros que indicam algum problema com recursos do sistema e a aplicação não consegue lidar com esses tipos de erros. As exceções RuntimeException são exceções que ocorrem em tempo de execução da aplicação e são unchecked, ou seja, não precisam ser tratadas, se não houver a necessidade.

12.b:

O uso deste bloco se dá quando há a necessidade de tratar alguma exceção que o código lançou e com o finally será sempre executado, independentemente se houve lançamento de exceção ou não pelo código. Um exemplo de uma utilização bastante frequente deste bloco é em aplicações que utilizam recursos de banco de dados, em

que após a utilização do recurso, dentro do try, haverá o fechamento do recurso/conexão no finally, independentemente se houve exceção ou não.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithTryCatchFinally.java"

"src/main/java/knowingtheplatform/workingwithexceptions/SomeClassToWork.java"

12.c:

- ArithmeticException: indica erro em processamento aritmético. Exemplo, dividir qualquer número por 0;
- ArrayIndexOutOfBoundsException: indica que houve uma tentativa de acesso a um elemento fora dos limites de um array, seja valor negativo ou maior que o tamanho do array;
- ClassNotFoundException: indica que a JVM não conseguiu carregar uma classe durante o tempo de execução da aplicação;
- IOException: indica a ocorrência de erro relacionado a operações de entrada e saída;
- IllegalArgumentException: indica que um método passou um argumento ilegal ou inválido;
- InterruptedException: indica que houve a interrupção de uma thread;
- NullPointerException: indica que a aplicação tentou usar uma referência de um objeto que não foi definida, instância. Não existe um endereço de memória alocado ainda para aquele objeto.

Código Fonte:

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithArithmeticException.java"

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithArrayIndexOutOfBoundsException.java"

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithIlegalArgumentException.java"

"src/main/java/knowingtheplatform/workingwithexceptions/MyThread.java"

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithExceptions.java"

12.d:

Código Fonte:

"src/main/java/knowingtheplatform/workingwithexceptions/WorkWithCustomizedExceptions.java"

"src/main/java/knowingtheplatform/workingwithexceptions/Calculator.java"

"src/main/java/knowingtheplatform/workingwithexceptions/myexceptions/NegativeValueException.java"

"src/main/java/knowingtheplatform/workingwithexceptions/myexceptions/OutOfBoundsValueException.java"

▪ Trabalhando com qualidade de código

13. A qualidade de um software não pode ser avaliada em um produto já feito. É fundamental que, durante o desenvolvimento, sejam adotadas técnicas para aprimorar a produção. Pesquise sobre testes unitários em Java e:

a. Pontue quais frameworks existem atualmente para auxiliar o desenvolvimento de testes unitários no ecossistema Java e a diferença entre eles;

b. Escolha um framework e faça testes unitários para pelo menos três exercícios desenvolvidos anteriormente;

c. Justifique sua escolha de framework e exercícios.

Respostas:

13.a:

- JUnit: o JUnit é um dos frameworks mais conhecidos dentro do desenvolvimento Java, facilitando o desenvolvimento e execução de testes unitários. O framework fornece uma API completa para a construção de testes de uma aplicação. Os principais motivos para se usar o JUnt são:
  - O framework consegue verificar se cada unidade de código funciona da forma que é esperado;
  - Facilita a criação, execução automática de testes e a apresentação dos resultados;
  - Os testes JUnit permitem escrever códigos mais rapidamente, o que aumenta a qualidade - através do TDD;
  - É simples, pouco complexo e rápido para a escrita de testes;

- É orientado a objetos, e;
- É open source.

Alguns recursos do JUnit:

- Provê annotations para identificação dos métodos de testes;
- Provê verificação (assertation) para testar os resultados esperados;
- Os testes JUnit podem ser executados automaticamente e eles verificam seus próprios resultados e fornecem feedback imediato. Não há necessidade de vasculhar manualmente um relatório de resultados de teste.
- Os testes JUnit podem ser organizados em suítes de teste contendo casos de teste e até mesmo outras suítes de teste.
- Feedback visual sobre o resultado dos testes;

- Testng: o Testng é outro framework para auxiliar nos testes de códigos Java. É baseado no JUnit e no NUnit, introduzindo algumas novas funcionalidade que o fazem ser mais poderoso e fácil de usar que os outros frameworks. Ele cobre todas as categorias de teste: unitária, funcional, end-to-end, integração, cargas e etc. Alguns recursos do framework:
  - Provê suporte a annotations;
  - Usa mais recursos do Java e da POO;
  - Oferece suporte ao teste de classes integradas;
  - Separa o código de teste em tempo de compilação das informações de configuração/dados em tempo de execução;
  - Configuração de tempo de execução flexível;
  - Apresenta 'grupos de teste'. Depois de compilar seus testes, você pode simplesmente pedir ao TestNG para executar todos os testes de "front-end" ou testes "rápidos", "lentos", "banco de dados", etc;
  - Oferece suporte a métodos de teste dependentes, teste paralelo, teste de carga e falha parcial;
  - Suporte para testes multi-thread.

13.b:

Código Fonte:

“src/test/java/knowingtheplatform/workingwithdatatypes/workingwithdates/WorkWithLocalDateTest.java”

“src/test/java/knowingtheplatform/workingwithdatatypes/workingwithstrings/ConcatAStringExamplesTest.java”

“src/test/java/knowingtheplatform/workingwithexceptions/CalculatorTest.java”

13.c:

Fiz a escolha pelo JUnit por ser o framework mais utilizado pelos desenvolvedores Java. Assim, ele possui uma comunidade muito grande em que há sempre contribuições para manter o framework - apoiado pelo fato dele ser open source. Outro motivo é pela experiência prévia que possuo com ele. Os exercícios escolhidos foram sobre o uso de LocalDate, a concatenação de String e o uso de Exceptions customizadas.

Todas foram escolhidas pois representa uma situação que ocorre o tempo todo no desenvolvimento de software, que é conter uma regra dentro de um método/classe e conhecendo essa regra nós podemos implementar testes unitários que garantem que essa regra não será mudada e se for mudada haverá a mudança no resultado dos testes e assim não poderá ser uma mudança arbitrária.

No caso da concatenação é concatenado um espaço em branco entre as Strings, isso é uma regra que fazendo uma analogia com o mundo real é para separar as frases que serão concatenadas. O teste unitário desta classe reflete essa regra, o resultado esperado contém também essa regra, caso alguém mude o método e tire o espaço, ou mude o espaço para "-", o teste irá quebrar e assim indicará a mudança de comportamento daquela classe.

O mesmo acontece para a classe de exemplos de LocalDate, os métodos fazem manipulação com o valor da variável, e os testes unitários refletem essas manipulações, caso haja mudança o teste irá quebrar. Por estes motivos escolhi estes dois exemplos de uso, de LocalDate e String, que precisei implementar para os outros exercícios, para usar como exemplo de testes unitários pois nos métodos há regras que se alteradas quebrarão o resultado esperado e também o que o método diz fazer. Assim pude mostrar rapidamente a importância e também o uso do JUnit.

O terceiro exemplo de teste unitário eu escolhi o exercício de exceções customizadas para mostrar que é possível fazer verificações em cima das exceções que lançamos. A classe que testei tinha duas regras básicas que devem lançar as minhas exceções customizadas caso satisfeitas. Com o teste unitário, e usando o JUnit, eu posso verificar se o meu código está lançando essas exceções e não tenha sido modificado para fazer alguma coisa caso aconteça as regras.

✓ Vamos praticar – Judge

1. **Escolha dois problemas no site Judge e:**
    a. **Crie um algoritmo para resolver o problema utilizando o conhecimento adquirido nos exercícios anteriores;**
    b. **Seu algoritmo deve ter uma cobertura de testes de no mínimo 80%;**
    c. **Justifique a escolha dos problemas;**
    d. **Envie seu código para o repositório GIT.**

Resposta:

Os problemas que escolhi foram o "Where is Marble?" e "Encryption". Ambos problemas são simples, nível 5, porém são problemas que pude aplicar os conhecimentos que obtive durante o restante dos desafios.

"Where is the Marble?": O problema consiste num jogo em que números são dispostos em peças de mármores e tem que acertar os números que estão nessas peças, de acordo com um número de tentativas. É um problema bastante simples de ser resolvido quando usado uma estrutura de dados. No caso eu escolhi este exercício para resolver para trabalhar com array de inteiros, que vimos durante a resolução dos desafios anteriores. Também é necessário a ordenação do array para pegar a correta posição da peça, esta ordenação poderia ser feita através da implementação de qualquer método conhecido, porém já usei o Arrays.sort(), justamente para mostrar o uso desta feature da classe Arrays. Então, na resolução deste desafio pude mostrar declaração de array, inicialização, ordenação e busca de elementos. Além disso, pude mostrar também o uso de laços de repetição, como se comporta o While e a sua verificação inicial da condição e também percorrer array através do for.

"Encryption": O problema consiste na construção de um algoritmo para fazer a criptografia de palavras. É um problema interessante, pois pude trabalhar com cada caractere de uma String, fazendo o shift de seu valor. Para isso, assim como o problema anterior, usei laço de repetição for pudendo usar o lenght de cada String. Outra coisa que pude demonstrar foi o uso do laço de repetição do-while, que faz pelo menos uma vez o código dentro de seu bloco, no caso quando eu tivesse apenas uma String para encriptar. Esse problema também foi interessante pois pude fazer o código de controle de fluxo em uma classe e separei a responsabilidade da criptografia para outra classe. Assim, conseguir testar unitariamente cada classe e através do framework Mockito consegui testar a integração das duas.