**Universidade Federal de Campina Grande – UFCG**

**Centro de Engenharia Elétrica e Informática – CEEI**

**Departamento de Sistemas e Computação – DSC**

Disciplina: Laboratório de Programação 2

# **Laboratório 04**

Neste laboratório iremos praticar o **uso de** [**coleções**](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) do tipo [lista](http://docs.oracle.com/javase/8/docs/api/java/util/List.html), [mapa](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html) e [conjunto](http://docs.oracle.com/javase/8/docs/api/java/util/Set.html). Também vamos estimular o **uso de testes de unidade** e continuar desenvolvendo habilidades na definição da lógica de controle de programas Java[.](https://sites.google.com/site/prog2ufcg/recursos) Em resumo os Objetivos são:

* Praticar o uso das diferentes coleções de Java;
* Utilizar Exception para tratamento de erros;
* Utilizar testes de unidade através do JUnit;

Para ver exemplos de uso e documentação sobre o JUnit, clique [aqui](http://junit.org/)

|  |
| --- |
| **Importante:** Toda classe precisa ter os métodos toString, equals e hashcode implementados, mesmo que isso não seja explicitamente solicitado. Toda classe precisa ter uma classe de teste associada, mesmo que isso não seja explicitamente solicitado. |

## Descrição geral da Implementação: sp2fy

O **sp2fy** é um programa para armazenar e organizar as músicas de um **perfil** de usuário.Todas as músicas estão organizadas em álbuns que, além das músicas, possuem um artista, um título e um ano de lançamento. Lembre que, em um álbum, as músicas estão organizadas em faixas, de forma que eu possa tocar a música da faixa 1, 2, 3, e assim por diante.

Tanto os albuns como as músicas serão utilizadas pelo usuário da seguinte forma: Cada perfil de usuário (ou simplesmente **perfil**) possui uma Musiteca, que é a biblioteca das suas músicas, albuns e playlists. É possível que o usuário possa definir também uma coleção com seus álbuns favoritos.

Cada perfil possui também playlists, que são várias listas de músicas. Essas listas são identificadas por um nome, de forma que cada playlist tem um nome. Por exemplo *“Para relaxar”*, *“Arrumar a Casa”*, *“Para fazer Laboratorios”*, etc. Associado a cada nome temos um agrupamento de músicas. Lembrando que, em cada uma dessas playlists, as músicas são tocadas em uma ordem. No seu código, deve ser possível definir vários nomes de playlists e associar um agrupamento de músicas a cada uma. A pasta de testes desse lab está disponível [aqui](https://goo.gl/CZhV8o).

**Agora que você está ficando experiente em Java,** realize o **tratamento de erros com Exception**, evitando que sejam criadas músicas/álbuns com títulos, gêneros e artistas vazios e também ano ou duração negativa. Além disso, você vai aprender a **usar testes de unidade** para dar suporte ao desenvolvimento das suas classes.

**Uma dica muito importante:** Neste lab você vai criar várias classes que **encapsulam coleções**. Uma classe que encapsula uma coleção deve oferecer métodos para adicionar, remover, iterar e pesquisar elementos nessa coleção. Siga os seguintes passos para a implementação do sp2fy:

|  |
| --- |
| **Boas práticas:** É recomendado que métodos que adicionam e removem elementos de uma coleção retornem um **boolean** que indica se a operação de adição ou remoção foi realizada com sucesso. |

## Passo 1: Criação das Músicas

**Importante:** Siga as orientações dadas em sala para começar a usar testes de unidade. Em seguida, baixe a classe de teste de música [aqui](https://goo.gl/SAOZr4) (MusicaTest.java).

**Implemente a classe Musica**. Cada música possui um título, uma duração **(em minutos inteiros)** e gênero. A representação da música por String segue o seguinte formato:

<titulo> (<genero> - <duracao> minutos)

Por exemplo: Hello (Pop - 4 minutos); Don’t stop me now (Rock - 5 minutos);

Duas músicas são consideradas iguais quando possuem o mesmo título e a mesma duração. Por exemplo Hello (Pop - 4 minutos) é igual a Hello (Forro - 4 minutos);

Forneça os métodos e atributos para que Musica funcione da forma adequada. Para verificar se sua implementação está correta, utilize a classe **MusicaTest.java** que possui alguns casos de teste do JUnit.

**Passo 2: Tratamento de erros com exceção**

Você percebeu que não fez nenhum tipo de tratamento de erros nas suas classes até agora? Em Java podemos fazer isso usando o conceito de Exceções (relembre a aula teórica!). Exceções são objetos que usamos para representar os erros que identificamos no nosso código; esses objetos podem ser “retornados” (**lançados**) dos métodos aonde foram criados e **capturados** em outras classes.

**Vamos usar Exceções para fazer tratamento de erros na classe Musica.java.**Você deve usar exceções para não permitir criação de músicas com título ou gênero que sejam strings vazias ou null. A duração das músicas não pode ser menor ou igual a zero. Verifique os casos em que ocorrem Exception no teste da própria [MusicaTest](https://goo.gl/SAOZr4). **No seu código de produção a captura e impressão de todas as mensagens de erros das Exception serão feitas no Perfil (explicado no Passo).**

## Passo 3: Criação dos Álbuns

**Implemente um album do sp2fy**. O álbum deve armazenar um agrupamento de músicas que permita tocar as músicas de acordo com sua faixa. Além das músicas, um álbum deve possuir um *artista*, um *título*, e um *ano* de lançamento. Você deve usar exceções para impedir nomes de artistas null ou vazios e anos de lançamento inferiores a 1900 (inclusive). Uma vez que a comparação de todas as músicas de um álbum é custosa, iremos considerar que dois álbuns serão iguais se possuírem o mesmo nome e o mesmo artista. A representação do album por string é:

<titulo\_album>, <artista> (<ano>)

1. <titulo\_musica1> (<genero1> - <duracao1> minutos)
2. <titulo\_musica2> (<genero2> - <duracao2> minutos)
3. ...

Por exemplo:

Hello, Adelle (2015)

1. Hello (Pop - 4 minutos)
2. Send my love (Pop - 3 minutos)
3. … //aqui seriam as próximas faixas do Album Hello de Adelle.

Para facilitar o uso do sp2fy, **todo álbum começa vazio** e então adicionamos Músicas ao álbum. A ordem em que as músicas são adicionadas correspondem à faixa da música. Em outras palavras, a primeira música adicionada será a faixa 1, a segunda será a faixa 2, e assim por diante. É possível **remover** músicas do album e **pesquisar** se uma musica existe no album usando como chave o título da música.

Também é importante saber qual o **tempo total de duração de álbum.** Note que esse tempo é a soma da duração de cada música do álbum, então você não precisa guardar esse tempo, apenas calcula-lo por meio de um metodo getDuracao. É possível obter a música que está em uma determinada faixa do album. Além disso, é preciso criar métodos que identifiquem se uma música com um certo título está presente no album. Apesar das coleções aceitarem elementos null, no contexto do album, não faz sentido aceitar que músicas sejam elementos null. Isso deve ser tratado com o uso de exceções.

Use a classe [AlbumTest.java](https://drive.google.com/file/d/0BzPbnH1xLbLMcW02XzJBaElreDA/view?usp=sharing) para dar suporte à criação de álbuns.

## Passo 4: Criação da Musiteca

Cada usuário do sp2fy tem sua própria musiteca, que organiza todos os albuns de um perfil por meio de uma coleção chamada meusAlbuns. Albuns podem ser adicionados ou removidos à essa coleção, porém albuns null ou repetidos não devem ser aceitos. Permita também realizar pesquisas nessa coleção de albuns. Uma das pesquisas é retornar todos os álbuns de um artista específico, que estão presentes naquela coleção.

## Passo 5: Albuns favoritos

Cada usuário do sp2fy pode definir uma coleção de albuns favoritos. Por exemplo, Joãozinho possui vários albuns em seu perfil, porém seus albuns favoritos são “Sinfonias de Beethoven” e “Hello”. Permita que cada perfil possa agrupar albuns em uma coleção de albuns favoritos.

Os favoritos pertencem à Musiteca, portanto, crie métodos para adicionar e remover álbuns da própria coleção da musiteca nos seus favoritos. Porém, o método adicionar só retorna true se o álbum adicionado nos favoritos está contido em meusAlbuns. Albuns null não devem ser aceitos.

## Passo 6: Playlists

Na musiteca, o usuário pode criar playlists que são uma coleção de músicas de diferentes álbuns. As playlists são identificadas pelo seu nome, portanto ao armarzená-las, devemos recuperar a playlist pelo seu respectivo nome. Ao criar, uma playlist não possui músicas, daí podemos começar a adicionar músicas à playlist. Uma nova música adicionada vai sempre para o fim da playlist.

Por questão de direitos autorais, **as músicas de qualquer playlist devem ser todas músicas de albuns pertencentes à respectiva musiteca, caso contrário a música não pode ser adicionada à playlist**.

Crie o método adicionaPlaylist(String nomePlaylist, String nomeAlbum, int faixa) que irá adicionar uma determinada faixa de um album na playlist. De forma mais detalhada, esse método realiza as seguintes operações:

1. Verifica se existe uma **playlist** com esse nome.
   1. Se não existir, cria um agrupamento vazio de músicas e associa esse agrupamento vazio ao nome especificado.
   2. Se existir, recupere o agrupamento (playlist) existente.
2. Busca em meusAlbuns o álbum com o nome especificado no método (nomeAlbum)
   1. Se não existir, use Exception para enviar mensagem dizendo que “Album nao pertence ao Perfil especificado”.
   2. Se existir, pegue a faixa do álbum especificado na assinatura do método e adicione no agrupamento resultante da operação 1.

Para esse passo, tenha cuidado com a verificação de faixas, nomes de álbuns e adição da música especificada na playlist. Note que são usados **dois agrupamentos** distintos: **um para armazenar todas as playlists** de um usuário de acordo com seu nome, e outro agrupamento para **armazenar as músicas da playlist de nome específico**.

Lembre que todos os métodos que adicionam, removem e pesquisam objetos em coleções devem retornar um boolean (true indica que a operação foi feita com sucesso e false indica o contrário). Mapas, conjuntos e listas aceitam elementos null. No entanto, no contexto da Musiteca, não faz sentido aceitar que músicas, albuns e playlists sejam compostos por quaisquer elementos null. Ao tentar adicionar, remover, pesquisar com parâmetros null recebidos como entrada, seu método deve retornar false.

Veja os testes para a classe Musiteca [aqui](https://goo.gl/Gn68GP) (MusitecaTest.java).

## Passo 7: Criação dos Perfis de Usuários

Os usuários do sp2fy tem um perfil em que são armazenadas informações pessoais do usuário e a biblioteca de músicas (musiteca) do usuário. Por enquanto, a única informação pessoal armazenada é o **nome** do usuário. Todo usuário tem uma Musiteca com suas coleções de albuns e playlists. Para esse passo, faça apenas com que cada perfil, **ao ser criado**, possua um nome e uma **musiteca vazia**. Em outras palavras, um perfil começa vazio e o usuário adiciona álbuns e playlists ao seu perfil, durante o seu uso.

Para isso, métodos **delegadores** devem ser criados (métodos que simplemente chamam um método da musiteca). Por exemplo, o seguinte método delegador deve existir:

|  |
| --- |
| public boolean adicionaAlbum(Album album) {  return musiteca.addAlbum(album);  } |

Vários métodos delegadores que chamam **praticamente todos os métodos da musiteca** devem existir aqui. Vamos também realizar a captura de Exception. Para isso use blocos try/catch em todas as chamadas na Musiteca para imprimir as mensagens de erro da Exception.

**Melhore suas mensagens de erro!!!** Além da mensagem, imprima uma mensagem indicando qual a operação que deu erro. Um exemplo de bloco try/catch segue abaixo. Substitua a palavra adicao por busca/remoca/criacao nos respectivos métodos das funcionalidades da musiteca (lembre de indicar o objeto da operação, por exemplo, adicao de musicas na playlist, adicao de album nos favoritos, etc.). Lembrando que o erro foi na operação, mas no nosso programa, a causa real da Exception é a criação de um objeto (Musica, Album, etc.) com valores inválidos.

|  |
| --- |
| **public boolean** adicionaAlbum(Album album) {  **try{**  //esse trecho retorna um boolean,  // mas futuramente pode lancar Exception.  return musiteca.addAlbum(album);  **}catch(**Exception excecao){  System.out.println(“Erro na adicao de musicas no album”);  System.out.println(excecao.getMessage());  }  }  //Metodo sobrecarregado (overload) para outra  // forma de adicionar albuns com o Creator no Perfil.  **public boolean** adicionaAlbum(String tituloAlbum, String artista, int ano) {  **try{**  //aqui pode lancar Exception.  Album album = new Album(tituloAlbum, artista, ano);  return musiteca.addAlbum(album);  **}catch(**Exception excecao){  System.out.println(“Erro na adicao de musicas no album”);  System.out.println(excecao.getMessage());  }  } |

## Passo 8 (Extra): Ordenar os Álbuns de um usuário

Realize as modificações necessárias para que os seus álbuns sejam ordenáveis por ano de lançamento. Para que isso ocorra é necessário determinar que os albuns sejam comparáveis. Isso é realizado usando o [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)<Album>. Crie na musiteca um novo método chamado sortedAlbuns que vai imprimir a coleção de albuns de um usuário considerando a coleção de albuns ordenada por ano e a coleção de albuns favoritos também ordenado por ano. Você deve usar Collections.sort para a ordenação. Perceba que os métodos compareTo e equals precisam estar implementados corretamente para que a ordenação ocorra com sucesso.

**Outras sugestões**: Realize **também** outros três métodos para permitir diferentes ordenações. Para isso, leia sobre o uso do Comparator<T>.

1. Extra 1: Ordenação por nome de artista do álbum.
2. Extra 2: Ordenação por quantidade de músicas na lista.
3. Extra 3: Ordenação por duracao total do álbum.

## Como o seu lab será avaliado

Serão considerados os seguintes critérios com suas respectivas notas:

* (1.0) Funcionalidade: observar se as funcionalidades pedidas foram implementadas corretamente
* (3.0) Design: escolha e uso correto de cada coleção
  + Listas, Mapas e Conjuntos (todas as 3 estão nesse lab).
* (5.0) Implementação dos tipos de dados usados no sistema (classes e lógica de funcionamento):
  + (0.5) Musica
  + (1.0) Album
  + (1.0) Lógica da playlist
  + (1.5) Musiteca
  + (1.0) Perfil
* (1.0) Legibilidade: inclui organização do código, estilo, lógica clara (uso de exceções e algoritmos simples) e atributir nomes intuitivos aos métodos e variáveis.

## Dicas para a implementação:

**Verificação de Instâncias repetidas:**

Para verificar se um agrupamento já possui uma instância (método contains(), ou o uso de conjuntos), é necessário ensinar a Java o que são duas Musicas/Albuns iguais. Portanto, não esqueça de implementar **.equals e .hashCode para todas as suas classes**. É importante que você pense quais são os elementos “chave” das suas classes. Em outras palavras, que atributos serão utilizados para determinar que duas instâncias são iguais. Para facilitar a implementação do hashCode, use a implementação automática dele fornecida pelo Eclipse que gera automaticamente o hashCode() e o equals(...). **É altamente recomendado que você implemente o seu próprio equals(..). Cuidado na criação do hashCode, escolha exatamente os mesmos atributos usados no equals.**

**Uso do Collections.sort(....) e Comparable<T>:**

O Comparable é usado para determinar que uma classe T (uso do generics) é comparável. Ou seja, é possível determinar entre duas instâncias t1 e t2 de T qual delas é maior, igual ou menor que a outra. Em outras palavras, determinar se: t1 < t2; t1 == t2; t1 > t2. Para isso deve ser implementado o método t1.compareTo(T t2){...} que retorna um inteiro de forma que:

* **Retorna valor maior que 0** se t1 > t2;
* **Retorna 0** se t1 == t2;
* **Retorna valor menor que 0** se t1 < t2;

Considere a implementação de uma classe Pessoa abaixo:

|  |
| --- |
| **public class** Pessoa **implements** Comparable<Pessoa> {  **private** String nome;  **private** int idade;    **public** Pessoa(String nome, int idade) {  **this.**nome = nome;  **this.**idade = idade;  }  **//imagine que aqui temos os gets e sets de Pessoa**  **//continuacao da classe...  //o compareTo compara pessoas apenas usando suas idades.**  **public int** compareTo(Pessoa outraPessoa){  **return** this.getIdade() - outraPessoa.getIdade();  }  } |

Se você tem strings em vez de int você pode usar o compareTo da classe String. Praticamente todas as classes mais importantes de java implementam a interface Comparable e portanto tem o método compareTo já implementado.

Agora vamos comparar: Pessoa eu = new Pessoa(“Neto”,28) e Pessoa vovo = new Pessoa(“Maria”,70). Os resultados são os seguintes:

* eu.compareTo(vovo) → vai retornar valor menor que zero, pois eu < vovo
* vovo.compareTo(eu) → vai retornar valor maior que zero, pois vovo > eu

Após adicionar o Comparable<T>, a coleção é automaticamente ordenada utilizando o método estático: Collections.sort(suaColecao). Note que isso não deve ser feito em conjuntos HashSet, pois eles não devem possuir uma indexação e portanto, não podem ser ordenados.

**Uso de Collections**

Como vimos na aula teórica, uma coleção, no contexto de orientação a objetos, é um agrupamento de objetos, ou seja, um objeto que contém outros objetos. Mais formalmente, uma coleção é uma estrutura de dados que pode armazenar ou agrupar referências a outros objetos (um *container*). Essencialmente, uma coleção deve oferecer métodos para adicionar e remover objetos, recuperar um objeto específico e varrer todos os seus objetos. Existem diversos tipos de coleções, cada uma com suas características específicas. Muitas vezes escrevemos classes que encapsulam coleções específicas (por exemplo, a classe Album) e nesse caso métodos para adicionar e remover objetos, recuperar um objeto específico e varrer todos os seus objetos precisam ser especificados.

Uma lista é uma coleção indexada de objetos. Índices de uma lista iniciam em zero, isto é, o índice do primeiro elemento é zero. As principais características de uma lista são: (i) o usuário da lista tem total controle sobre onde os elementos devem ser inseridos ou de onde eles devem ser removidos e (ii) é possível acessar elementos da lista usando seu índice numérico, que indica a posição do elemento na lista. Uma implementação de lista muito usada em Java é representada pela classe java.util.ArrayList (uma implementação de array dinâmico; sem limitação de tamanho).

A classe ArrayList (de fato, ArrayList<E>) fornece diversos métodos (você vai ver mais olhando a documentação da API de java: [ArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)), entre os quais:

* boolean add(E elemento): Adiciona o elemento especificado na coleção;
* E get(int index): Retorna o elemento da lista no index especificado;
* int size(): Retorna o número de elementos da lista.

Observe o uso do tipo <E> na definição dos métodos acima. <E> refere-se a um tipo genérico que será especificado no momento da instanciação da classe. Com isso é possível indicar o tipo de objeto que comporá a coleção, facilitando a detecção de erros quando tipos diferentes dos especificados são adicionados à coleção. Para saber mais sobre Generics: <http://www.devmedia.com.br/usando-generics-em-java/28981>

Por exemplo, se quisermos criar uma agenda de contatos bem simples, podemos fazer isso usando um ArrayList de strings. Precisamos primeiro declarar o objeto agenda, e em seguida adicionar nossos contatos:

List<String> agenda = new ArrayList<String>();

agenda.add("David Gilmour;11 1111-1111");

agenda.add("Mick Jagger;22 2222-2222");

agenda.add("Eric Clapton;33 3333-3333");

Depois podemos listar todos os nossos contatos:

for (int i = 0; i < agenda.size(); i++) {

System.out.printf("%d - %s\n", i, agenda.get(i));

}

Note que ao declarar um objeto você vai indicar que ele é do tipo List. Mas ao fazer “new”, você vai escolher uma das implementações da interface List, neste lab você escolherá ArrayList. Fazemos isso para escrever código menos acoplado às implementações. Leia mais em: [Explorando a Classe ArrayList Java](http://www.devmedia.com.br/explorando-a-classe-arraylist-no-java/24298) - e no material da disciplina teórica de [P2](http://www2.lsd.ufcg.edu.br/~programacao2/p2-2012.2/6.%20colecoes.htm).