Programação II

Collections e Exceptions

Evandro J.R. Silva[1]

7 de setembro de 2022

Sumário

Introdução

- Aula baseada em Java Tutotials by Oracle:
  - Collections;
  - Exception.

Collections

- Uma *coleção*, ou *container*, é um objeto que agrupa múltiplos objetos.
- Em Java, as *Collections* (ou Coleções) são utilizadas para armazenar, recuperar, manipular e permitir *comnunição* entre dados agregados.
- Nós já somos familiarizados com algumas classes, principalmente `ArrayList`.

Collections

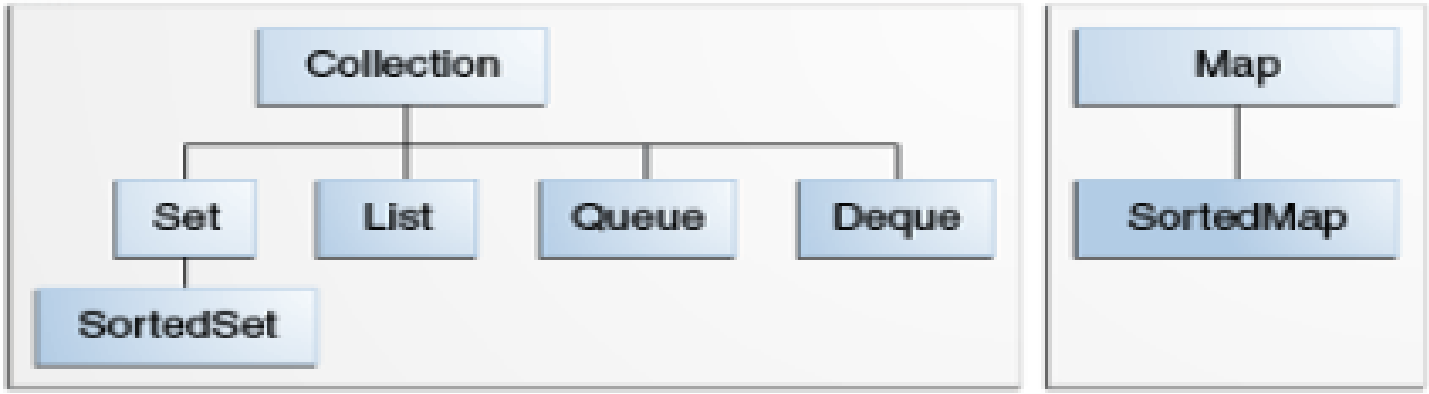
- Uma *coleção*, ou *container*, é um objeto que agrupa múltiplos objetos.
- Em Java, as *Collections* (ou Coleções) são utilizadas para armazenar, recuperar, manipular e permitir *comnunição* entre dados agregados.
- Nós já somos familiarizados com algumas classes, principalmente `ArrayList`.
- O *collections framework* do Java é uma arquitetura unificada para representação e manipulação de coleções. Todos os *framewoks* contêm:

Collections

- Uma *coleção*, ou *container*, é um objeto que agrupa múltiplos objetos.
- Em Java, as *Collections* (ou Coleções) são utilizadas para armazenar, recuperar, manipular e permitir *comnunição* entre dados agregados.
- Nós já somos familiarizados com algumas classes, principalmente `ArrayList`.
- O *collections framework* do Java é uma arquitetura unificada para representação e manipulação de coleções. Todos os *framewoks* contêm:
  - **Interfaces**;
  - **Implementações** [das interfaces] — basicamente esrturura de dados reutilizáveis (lembra das classes genéricas?);
  - **Algoritmos** — os métodos (por exemplo: `sort` ou `search`).

Interfaces

- O núcleo de interfaces encapsula diferentes tipos de coleções, de forma que elas possam ser manipuladas independentemente dos detalhes de suas representações.



Interfaces

- `Collection`: raiz da hierarquia. É o denominador comum de todas as coleções implementadas, e pode ser utilizada como o mais genérico possível alguma coleção possa ser. Não existe uma implementação direta desta interface.

Interfaces

- `Collection`: raiz da hierarquia. É o denominador comum de todas as coleções implementadas, e pode ser utilizada como o mais genérico possível alguma coleção possa ser. Não existe uma implementação direta desta interface.
- `Set`: uma coleção que não pode conter elementos duplicados.

Interfaces

- `Collection`: raiz da hierarquia. É o denominador comum de todas as coleções implementadas, e pode ser utilizada como o mais genérico possível alguma coleção possa ser. Não existe uma implementação direta desta interface.
- `Set`: uma coleção que não pode conter elementos duplicados.
- `List`: uma coleção ordenada (também chamada de `sequência`). Pode conter elementos duplicados.

Interfaces

- `Collection`: raiz da hierarquia. É o denominador comum de todas as coleções implementadas, e pode ser utilizada como o mais genérico possível alguma coleção possa ser. Não existe uma implementação direta desta interface.
- `Set`: uma coleção que não pode conter elementos duplicados.
- `List`: uma coleção ordenada (também chamada de `sequência`). Pode conter elementos duplicados.
- `Queue`: uma coleção usada para armazenar elementos antes de seu processamento. Tradução direta: *fila*. Provê operações adicionais de inserção, extração e inspeção em relação à interfeace `Collection`.

Interfaces

- `Collection`: raiz da hierarquia. É o denominador comum de todas as coleções implementadas, e pode ser utilizada como o mais genérico possível alguma coleção possa ser. Não existe uma implementação direta desta interface.
- `Set`: uma coleção que não pode conter elementos duplicados.
- `List`: uma coleção ordenada (também chamada de `sequência`). Pode conter elementos duplicados.
- `Queue`: uma coleção usada para armazenar elementos antes de seu processamento. Tradução direta: *fila*. Provê operações adicionais de inserção, extração e inspeção em relação à interfeace `Collection`.
- `Deque`: similar a `Queue`. A diferença é que `Queue` segue por padrão o conceito FIFO (*First In First Out*), ou seja, todo novo elemento é inserido no fim, e toda remoção acotence com o primeiro elemento da coleção. `Deque` por sua vez usa tanto FIFO quanto LIFO (*Last In First Out*) e todos os elementos podem ser inseridos e removidos em ambas as pontas.

Interfaces

- `Map`: mapeia chaves a valores. Não pode haver chaves duplicadas, e cada chave deve mapear pelo menos um valor.
- `SortedSet`: um conjunto (`Set`) com ordenação ascendente.
- `SortedMap`: um `Map` com chaves ordenadas de forma ascendente.

Implementações

- As implementações são os objetos usados para armazenar as coleções, ou seja, no nosso contexto, são as implementações das interfaces de coleção.
- As implementações são divididas entre aquelas de **propósito geral** e **propósito especial**.
- Lista de implementações de propósito geral:
  - `Set`: `HashSet`, `TreeSet` e `LinkedHashSet`.
  - `List`: `ArrayList` e `LinkedList`.
  - `Queue`: `PriorityQueue` e `LinkedList`.
  - `Deque`: `ArrayDeque` e `LinkedList`.
  - `Map`: `HashMap`, `TreeMap` e `LinkedHashMap`.
- Lista de implementações de propósito especial:
  - `Set`: `EnumSet` e `CopyOnWriteArraySet`.
  - `List`: `CopyOnWriteArrayList`.
  - `Queue`: Não possui de propórito especial, mas possui implementações concorrentes: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `PriorityBlockingQueue`, `DelayQueue` e `SynchronousQueue`.
  - `Deque`: Mesma situação de `Queue`, tendo a implementação concorrente `LinkedBlockingDeque`.
  - `Map`: `EnumMap`, `WeakHashMap` e `IdentityHashMap`.

Algoritmos

- Alguns métodos nativamente implementados para coleções:
  - `sort`: serve para ordenar uma lista;
  - `shuffle`: serve para embaralhar os elementos de uma lista.
  - `reverse`: reverte a ordem dos elementos.
  - `fill`: sobrescreve cada elemento de uma lista com um determinado valor.
  - `frequency`: conta a quantidade de vezes um elemento ocorre em uma coleção.

Exercícios

- Fazer os exercícios do site: https://www.w3resource.com/java-exercises/collection/index.php.

Exceções

- Durante a compilação e execução, quando algum erro ocorre, um objeto *especial* é criado.
- Esse objeto possui informações sobre o erro ocorrido, incluindo o seu tipo e o estado do programa quando o erro aconteceu.
- Esse objeto é *lançado* (*throw*) pelo compilador em busca de algum trecho do programa que possa lidar com ele.
- Erros que são notificados durante a compilação são chamados de ***checked exceptions***, enquanto os que são notificados durante a execução são chamados de ***unchecked exception***.
- Quando ocorre um erro a compilação ou execução é interrompida, e o erro é mostrado no console.
- E se?
  - Eu não quiser que a compilação/execução seja interrompida?

Exceções

- Durante a compilação e execução, quando algum erro ocorre, um objeto *especial* é criado.
- Esse objeto possui informações sobre o erro ocorrido, incluindo o seu tipo e o estado do programa quando o erro aconteceu.
- Esse objeto é *lançado* (*throw*) pelo compilador em busca de algum trecho do programa que possa lidar com ele.
- Erros que são notificados durante a compilação são chamados de ***checked exceptions***, enquanto os que são notificados durante a execução são chamados de ***unchecked exception***.
- Quando ocorre um erro a compilação ou execução é interrompida, e o erro é mostrado no console.
- E se?
  - Eu não quiser que a compilação/execução seja interrompida?
- Devemos nos servir das *exceções*.

Exceções

- Uma exceção é uma forma de lidar com esses erros sem que o programa seja interrompido.
- Famoso bloco *try - catch*:

```
try
{
    ...
} catch(Exception e)
{
    ...
}
```

- Todo erro que ocorre dentro de um bloco `try` vai ser tratado pelo bloco `catch` correspondente (ou seja, é possível ter vários blocos `catch`).
- Se dois ou mais erros puderem ser tratados com as mesmas linhas de código, a partir do Java SE 7 é possível especificá-los dentro do mesmo bloco:

```
...
catch (IOException|SQLException ex)
{
    logger.log(ex);
    throw ex;
}
```

Exceções

- E se o erro que ocorrer não tiver sido antecipado? Ou seja, é o caso do erro não ser capturado por qualquer bloco `catch`.
- Podemos acrescentar o bloco `finally`, o que **sempre** vai executar assim que a execução do `try` terminar:

```
try{
...
} catch(ExceptionType e){
    ...
} finally{
    ...
}
```

Exceções

- Dependendo da situação, às vezes você quer que determinado método de alguma classe possa *lançar* uma exceção, caso venha ocorrer algum erro (por causa do usuário, por exemplo).
- Você pode declarar esse método como *throwable* (`throws`), ou apenas lançar uma exceção caso alguma condição seja satisfeita.
- Exemplo:

```
public class Exemplo{
    public void depositar(double valor) throws RemoteException{
        // implementacao ...
        throw new RemoteException();
    }
    ...
    ...
}
```

Exceções

- Exemplo 2:

```
public Object pop(){
    Object obj;

    if (size == 0){
        throw new EmptyStackException();
    }

    obj = ...
    ...
}
```

Exceções

- E se eu quiser fazer minha própria exceção?

- Outro exemplo:

```
public class ChequeSemFundoException extends Exception{
    private double valor;

    public ChequeSemFundoException(double valor){
        this.valor = valor;
    }

    public double getValor(){
        return valor;
    }
}
```

- E depois dá pra utilizar em outro método:

```
// implementacoes de alguma classe ...
public void sacar(double valor) throws ChequeSemFundoExcepetion{
    if (saldo - valor < 0)
        throw ChequeSemFundoException(valor);
}
```

Exceções

- E se eu quiser fazer minha própria exceção?
- O Java te permite isso:

```
class MyException extends Exception{
    ...
}
```

- Outro exemplo:

```
public class ChequeSemFundoException extends Exception{
    private double valor;

    public ChequeSemFundoException(double valor){
        this.valor = valor;
    }

    public double getValor(){
        return valor;
    }
}
```

- E depois dá pra utilizar em outro método:

```
// implementacoes de alguma classe ...
public void sacar(double valor) throws ChequeSemFundoExcepetion{
    if (saldo - valor < 0)
        throw ChequeSemFundoException(valor);
}
```

Exceções

- Como eu vou decorar todas as exceções nativas do Java?

Exceções

- Como eu vou decorar todas as exceções nativas do Java?
- Não vai!
- Na verdade, apenas com o tempo, com bastante prática, você vai aprendendo as exceções mais comuns, e também saber buscar quais exceções podem ser lançadas em determinadas situações.

Exceções

- Como eu vou decorar todas as exceções nativas do Java?
- Não vai!
- Na verdade, apenas com o tempo, com bastante prática, você vai aprendendo as exceções mais comuns, e também saber buscar quais exceções podem ser lançadas em determinadas situações.
- Agora é hora de PRATICAR!

FIM

Code Coach

Só conseguimos aprender a programar quando praticamos!

Até a próxima!