Coleções e Exceções

- Tutoriais oficiais da Oracle - *Java Tutorials*
  - Coleções ou *Collections*
  - Exceções ou *Exceptions*

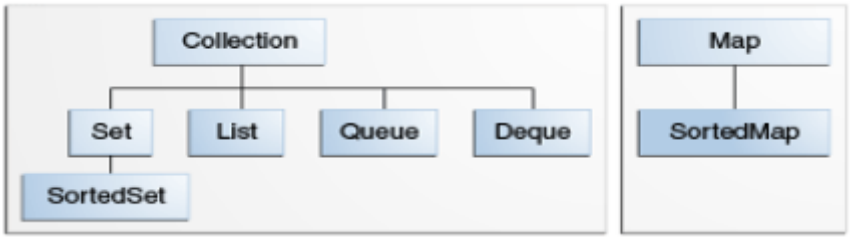
Coleções

Uma `coleção`, `collection`, ou `container`, é um objeto que agrupa múltiplos objetos. Em `Java`, as `Collections` são utilizadas para armazenar, recuperar, manipular e permitir a comunicação entre dados agregados. O `collection framework` do `Java` é uma arquitetura unificada para representação e manipulação de coleções. Todos os `frameworks` contêm:

- **Interfaces**;
- **Implementações** [das `interfaces`] --- basicamente estruturas de dados reutilizáveis;
- **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.



- **Collection**: raiz da hierarquia. É o denominador comum de todas as `coleções` implementadas, e pode ser utilizada como o mais genérico possível que 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 `interface Collection`.
- **Deque**: similar a `Queue`. A diferença é que o `Queue` segue por padrão o conceito `FIFO` (*First In First Out*), ou seja, todo novo elemento é inserido no fim, e toda remoção acontece com o primeiro elemento da `coleção`. O `Deque`, por sua vez, usa tanto o `FIFO` quanto o `LIFO` (*Last In First Out*), e todos os elementos podem ser inseridos e removidos em ambas as pontas.
- **Map**: mapeia chaves e 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** (o * indica a implementação de uso mais comum de cada tipo):

- `Set`: `HashSet` *, `TreeSet` e `LinkedHashSet`.

Set	HashSet	É uma coleção que não possui elementos duplicados e em que não existe preocupação com a ordem, sendo uma opção mais rápida para operações de modificação de conjunto.
	LinkedHashSet	É uma coleção que não tem elementos duplicados e que mantém a ordem na qual os elementos foram inseridos.
	TreeSet	É um conjunto que permite inserir elementos em qualquer ordem e recuperar os elementos classificados pela sua ordem natural.

- `List` : `ArrayList` * e `LinkedList` .

List	ArrayList	É uma coleção indexada, porém, não ordenada, de elementos. É mais rápida na pesquisa que as demais implementações, exceto no início e no fim da lista.
	Vector	É basicamente o mesmo que uma ArrayList, porém os métodos são sincronizados.
	LinkedList	É uma coleção duplamente indexada e não ordenada que fornece novos métodos para adicionar e remover elementos no início e no fim da lista, melhorando o desempenho em caso de implementações de pilhas ou filas.

- `Queue` : `PriorityQueue` e `LinkedList` *.
- `Deque` : `ArrayDeque` * e `LinkedList` .
- `Map` : `HashMap` *, `TreeMap` e `LinkedHashMap` .

Quadro 3.3 » Resumo das principais coleções Java

Interfaces	Implementações	Descrição
Map	HashMap	É uma coleção de elementos que associa uma chave única a um valor.
	Hashtable	É basicamente o mesmo que um HashMap, porém os métodos são sincronizados. Um método sincronizado só pode ser acessado por uma *thread* (execução individual de um processo) de cada vez.
	TreeMap	É um mapa em que as chaves são classificadas pela ordem natural dos elementos. A classificação de uma coleção pela ordem natural interfere na ordem da recuperação dos elementos. Por exemplo, os elementos de um conjunto de caracteres serão recuperados em ordem alfabética, os elementos de um conjunto de números serão recuperados em ordem crescente e assim por diante.
	LinkedHashMap	É um mapa que mantém a ordem de inserção dos elementos.

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Lista de implementações de **propósito especial**:

- `Set` : `EnumSet` e `CopyOnWriteArraySet` .
- `List` : `CopyOnWriteArrayList` .
- `Queue` : não possui de propósito 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 que um elemento ocorre em uma coleção.

Exercícios

Lista de exercícios para Java Collections.

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. Se não houver nenhum código para lidar com o erro, a `compilação` ou `execução` é interrompida e o erro é mostrado no `console` . O nome dado a esses erros é `exceção` .

> **cheked exceptions** são os erros notificados durante a compilação.

> **unchecked exceptions** são os erros notificados durante a execução (*runtime*).

E se um desenvolvedor não quiser que a compilação ou execução do programa seja interrompida caso aconteça algum erro?

Para este caso é necessário o `tratamento de exceções`, ou seja, o desenvolvimento do trecho de programa que vai lidar com o erro. Em `Java` isso é feito com o bloco `try-catch`:

```
try{
    // comandos ...
}catch (Exception e){
    // tratamento ...
}
```

Todo erro que ocorre dentro de um bloco `try` vai ser tratado pelo bloco `catch` correspondente (ou seja, é possível ter vários blocos `catch` seguidos). 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;
        }
```

Existe ainda a possibilidade de se acrescentar comandos no código que serão **SEMPRE** executados, independentemente se houve erro ou não (exemplo de uso: desconectar de um banco de dados). Essa possibilidade é dada pelo bloco `finally`:

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

Dependendo da situação, às vezes você quer que determinado `método` de alguma `classe` possa *lançar* uma `exceção`, caso venha ocorrer algum erro (por causa de alguma entrada inválida fornecida pelo usuário, por exemplo). Neste caso é possível declarar o `método` como *throwable* (`throws`), ou apenas lançar uma `exceção` caso alguma condição seja satisfeita. Exemplo:

```
import java.rmi.RemoteException;

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

Exemplo 2:

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

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

O `Java` também permite que você possa criar sua própria `exceção` :

```
class MinhaExcecao extends Exception{
  ...
}
```

Outro exemplo:

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

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

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

```
}
```

Utilizando a `exceção` criada em outra `classe` :

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

Como decorar todas as `exceções` nativas do Java?

Você não vai!

Na verdade, com o tempo, e bastante prática, você vai aprendendo sobre as `exceções` mais comuns, e também saber pesquisas quais `exceções` podem sre lançadas em determinadas situações.

Exercícios

Lista de exercícios para tratar exceções.