Avançando no estudo sobre Java

Enums

- Tipos de dados onde é claro o domínio (valores pré-definidos)
  - Ex: Sim/Não, PF/PJ, SAQUE/CREDITO/DEBITO, etc.

Avançando no estudo sobre Java

Collection framework: Classes utilizadas para armazenar coleções de objetos

Collection framework

São quatro interfaces que são implementadas:

- List: Aceita elementos duplicados
- Set: Não aceita elementos duplicados
    - SortedSet: Como o Set, mas ordenada (sort)
- Map: Aceita elementos com chave-valor
- Queue - Trabalha com fila e pilhas

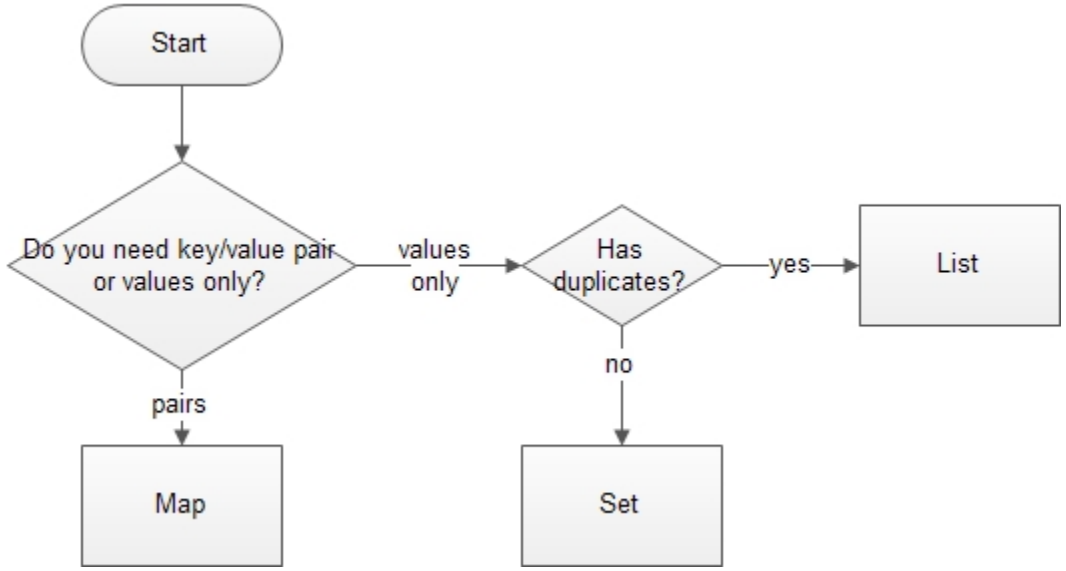
Sendo que:

- Hash: utiliza tabela de Hash interna
- Tree: utiliza árvore balanceada

Collection framework

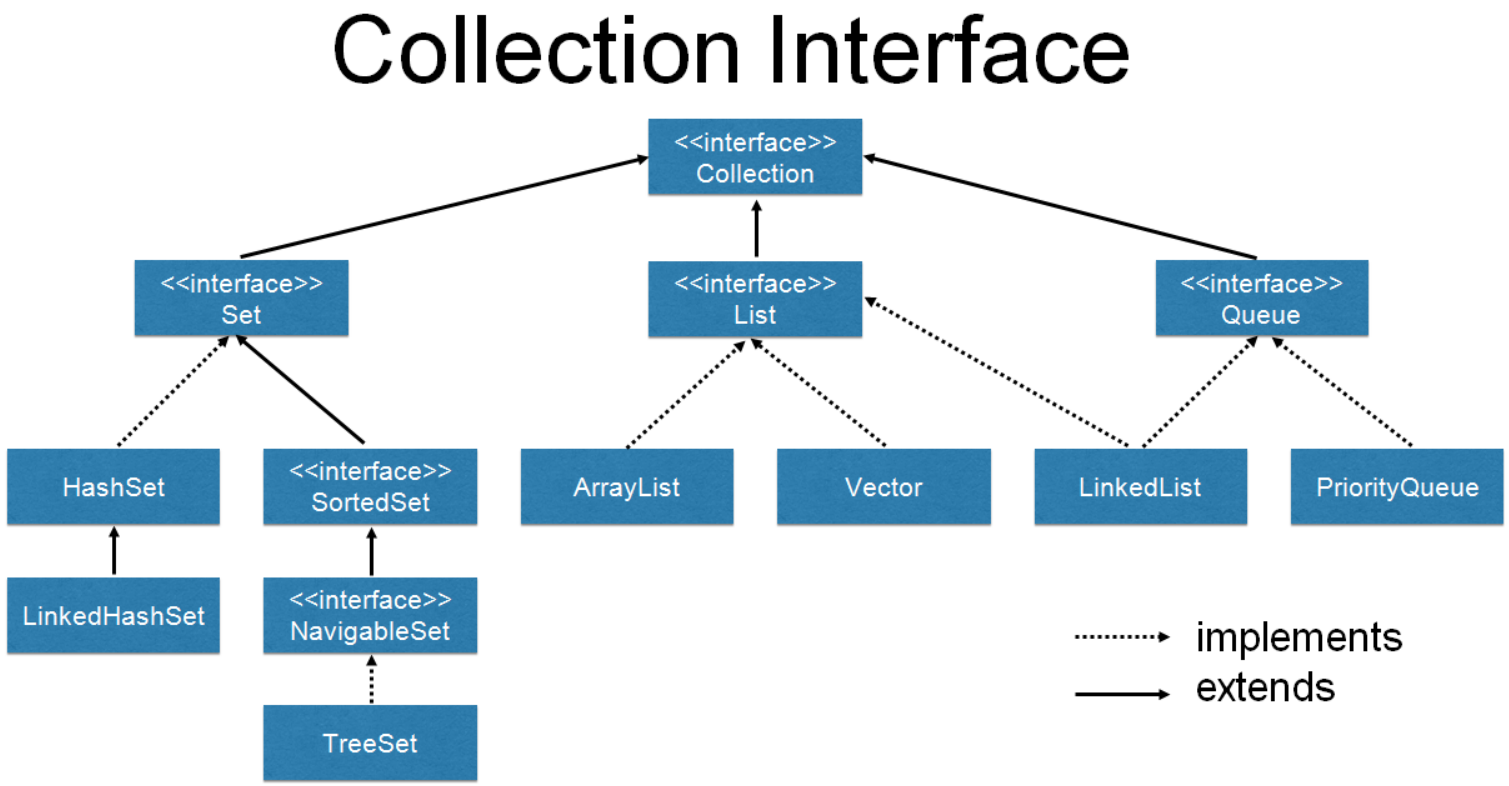
Mapas (MAP):

Qual interface usar?



Collection framework

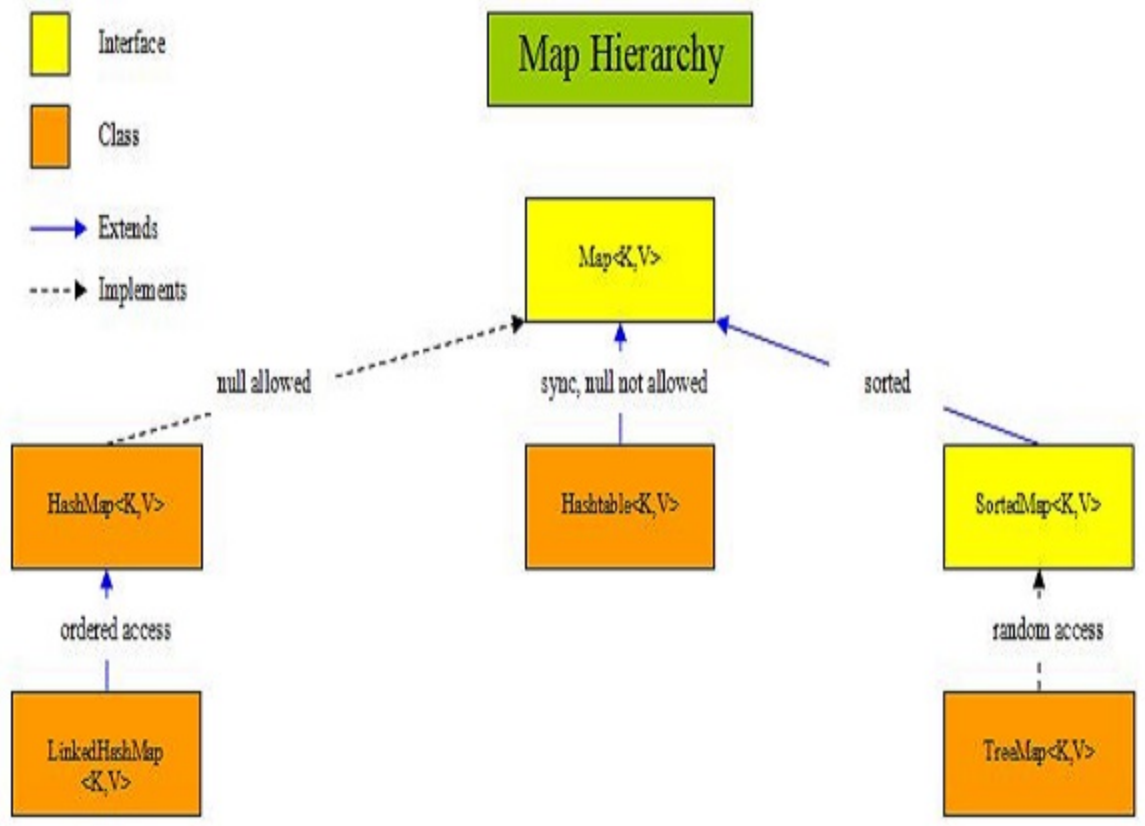
Hierarquia:



Collection framework

Mapas (MAP):

Guardam coleções de tuplas de valores (chave e valor).



Collection framework

List: Coleção que permite duplicações:

- ArrayList: Utiliza array internamente, mais rápido para acesso. Permite acesso pelo índice usando `lista.get(1)`.
- LinkedList: Utiliza lista encadeada, mais rápida para inserção e deleção.
- Vector: Antiga implementação semelhante ao ArrayList. Thread safe (sincronizado).

Collection framework

Map/Table: Coleções chave-valor:

- TreeMap: Registros são ordenados ascendentes pela sua chave, mas mais lenta que HashMap.
- HashMap: Não garante ordenação.
  - Para multi-thread use `ConcorrencyHashMap`.
- LinkedHashMap: Mantém a ordem de inserção.
- Hashtable: Antiga implementação de coleção chave-valor. Não permite nulo. Thread safe (sincronizado).

Collection framework

Set: Coleção que não permite duplicações:

- TreeSet: Implementa SortedSet e por isso ordena automaticamente os elementos de acordo com seu valor.
- HashSet: Não garante ordenação. Utiliza HashMap internamente.
- LinkedHashSet: Matém a ordem de inserção. Utiliza LinkedHashMap internamente.

Collection framework - ArrayList

Instanciando um ArrayList:

```
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("Ford");
    System.out.println(cars);
  }
```

Obtendo um dos ítens pelo índice:

```
cars.get(0);
```

Alterando um dos ítens:

```
cars.set(0, "BMW");
```

Removendo um dos ítens:

```
cars.remove(0);
```

Collection framework - ArrayList

Buscando um ítem pelo conteúdo:

```
cars.contains("BMW");
```

Removendo todos os ítens:

```
cars.clear();
```

Percorrendo os ítens (foreach):

```
for (String i : cars) {
      System.out.println(i);
}
```

Ordenando os ítens:

```
Collections.sort(cars);
```

Collection framework - HashMap

Instanciando um HashMap:

```
HashMap<String, String> capitais = new HashMap<String, String>();

capitais.put("RS", "Porto Alegre");
capitais.put("SC", "Florianópolis");
```

Acessando um ítem:

```
capitais.get("RS");
```

Collection framework - HashMap

Iterando sobre as chaves:

```
for (String c : capitais.keySet()) {
  System.out.println(c);
}
```

Iterando sobre os valores:

```
for (String v : capitais.values()) {
  System.out.println(v);
}
```

Collection framework

Iterator:

Abstrai a complexidade de iteração de uma lista.

Métodos comuns:

- hasNext(): Retorna true se há mais elementos na lista.
- next(): Retorna o elemento em questão.

Em alguns tipos como o LinkedList, é essencial utilizar o Iterator por questão de performance! (Pois cada .get() percorreria toda a lista.

Collection framework

Ordenação (classificação):

Para ordernar uma lista, utiliza-se o `Collections.sort(l)`.

Para o Java saber realizar uma ordenação podemos implementar a interface Comparator na nossa classe:

```
    public class Conta implements Comparable<Conta> {
        public int compareTo(Conta c1) {
          if (this.saldoTotal < c1.saldoTotal) {
            return -1; //Se Menor
          }
          if (this.saldoTotal > c1.saldoTotal) {
            return 1; //Se Maior
          }
          return 0; //Se Igual
        }
    }
```

Collection framework

Ordenação (classificação):

Outra forma de criar um Comparator é criando uma classe Comparator:

```
class ContaByValorComparator
        implements Comparator<Conta>
{
    public int compare(Conta c1, Conta c2)
    {
        return c1.saldoTotal.compareTo(c2.saldoTotal);
    }
}
```

Collection framework

Ordenação (classificação):

Ou utilizar um Comparator como o `Collections.reverseOrder()`:

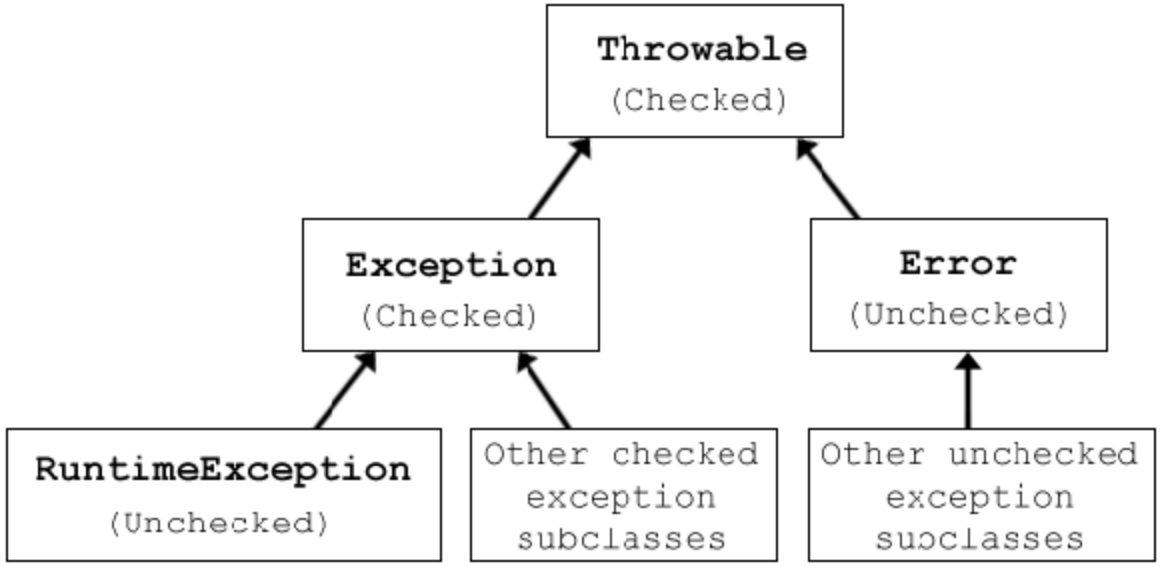
```
Collections.sort(lista, comparator);
```

Avançando no estudo sobre Java

Como evitar, lançar e tratar exceções

Tratamento de exceções

Exceptions são classes que simbolizam possíveis problemas que possam acontecer em runtime. Segue a hierarquia das mesmas:



Tratamento de exceções

Elas são classificadas em:

- Checked - exceções que obrigatoriamente precisam ser tratadas ou em blocos try catch ou declaradas na claúsula throws do método (herdadas de Exception).
- Unchecked - não precisam ser tratadas, são silencioas. (herdadas de Exception).

Atenção: em ambos os casos, caso nenhum nenhum método da pilha faça o tratamento da exceção, esta será propagada até o método main e o programa será encerrado.

Tratamento de exceções

Existem algumas palavras-chave que iremos utilizar:

- throw: Dispara uma exceção.
- throws: Indica que um método dispara uma exceção checada.
- try: Inicia um bloco onde é esperada uma determinada exceção.
- catch: Indica o que fazer quando ocorrer determinada exceção.

Tratamento de exceções:

Exemplo de método com as palavras-chave:

```
public void imprime(int posicao) throws PosicaoInvalidaException {

    try {

      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[posicao]);

    } catch (ArrayIndexOutOfBoundsException e) {

      System.out.println("Posição inválida!");
      throw new PosicaoInvalidaException();
    }
  }
```