Orientação a Objetos Classica

Namom Alves Alencar

Collections framework

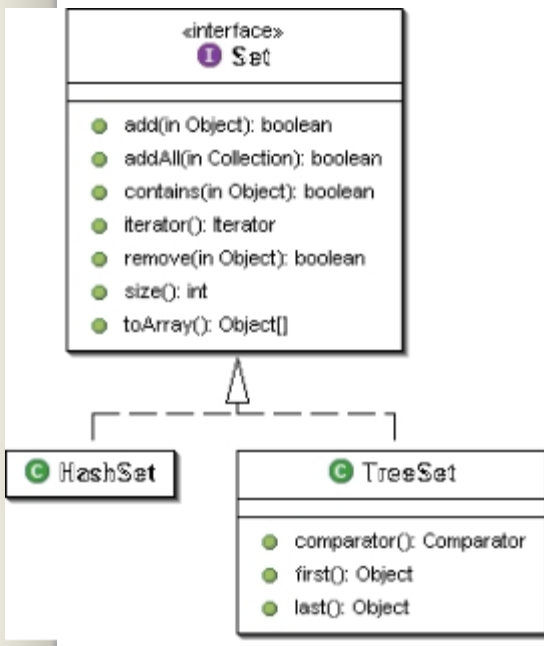
- Utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
- Iterar e ordenar listas e coleções;
- Usar mapas para inserção e busca de objetos.

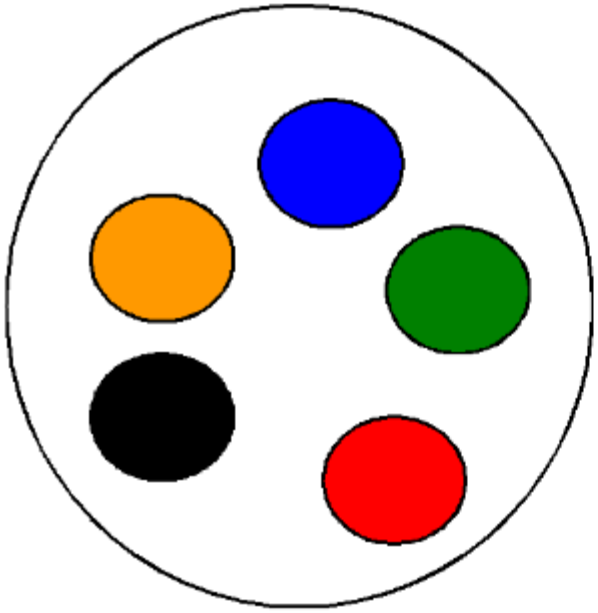
CONJUNTO: JAVA.UTIL.SET

- Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.

- Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto. A interface não define como deve ser este comportamento. Tal ordem varia de implementação para implementação.

CONJUNTO: JAVA.UTIL.SET





Possíveis ações em um conjunto:

- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- Não existem elementos duplicados!

- Ao percorrer um conjunto, sua ordem não é conhecida!

CONJUNTO: JAVA.UTIL.SET

- Um conjunto é representado pela interface Set e tem como suas principais implementações as classes HashSet, LinkedHashSet e TreeSet.

- O código a seguir cria um conjunto e adiciona diversos elementos, e alguns repetidos:

```
Set<String> cargos = new HashSet<>();
cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Diretor"); // repetido!
// imprime na tela todos os elementos
System.out.println(cargos);
```

- Aqui, o segundo Diretor não será adicionado e o método add lhe retornará false.

Conjunto: java.util.Set

- O uso de um Set pode parecer desvantajoso, já que ele não armazena a ordem, e não aceita elementos repetidos. Não há métodos que trabalham com índices, como o get(int) que as listas possuem. A grande vantagem do Set é que existem implementações, como a HashSet, que possui uma performance incomparável com as Lists quando usado para pesquisa (método contains por exemplo). Veremos essa enorme diferença durante os exercícios.

PRINCIPAIS INTERFACES: JAVA.UTIL.COLLECTION

- As coleções têm como base a interface Collection, que define métodos para adicionar e remover um elemento, e verificar se ele está na coleção, entre outras operações, como mostra a tabela a seguir:

`boolean add(Object)`	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna `true` ou `false` indicando se a adição foi efetuada com sucesso.
`boolean remove(Object)`	Remove determinado elemento da coleção. Se ele não existia, retorna `false`.
`int size()`	Retorna a quantidade de elementos existentes na coleção.
`boolean contains(Object)`	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método `equals()` do objeto, e não através do operador ==.
`Iterator iterator()`	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

PRINCIPAIS INTERFACES: JAVA.UTIL.COLLECTION

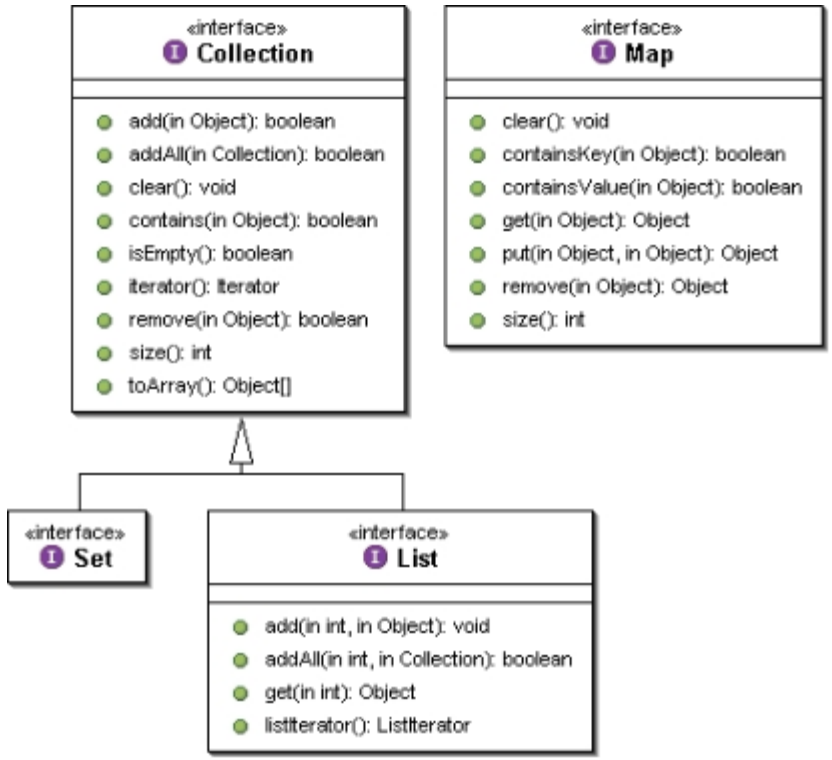
- Uma coleção pode implementar diretamente a interface Collection, porém normalmente se usa uma das duas subinterfaces mais famosas: justamente Set e List.

- A interface Set, como previamente vista, define um conjunto de elementos únicos enquanto a interface List permite elementos duplicados, além de manter a ordem a qual eles foram adicionados.

- A busca em um Set pode ser mais rápida do que em um objeto do tipo List, pois diversas implementações utilizam-se de tabelas de espalhamento (hash tables), realizando a busca para tempo linear (O(1)).

PRINCIPAIS INTERFACES: JAVA.UTIL.COLLECTION

- A interface Map faz parte do framework, mas não estende Collection. (veremos Map mais adiante).
  - Temos outra interface filha de Collection: a Queue, que define métodos de entrada e de saída e cujo critério será definido pela sua implementação (por exemplo LIFO, FIFO ou ainda um heap onde cada elemento possui sua chave de prioridade).



Percorrendo coleções

- Como percorrer os elementos de uma coleção? Se for uma lista, podemos sempre utilizar um laço for, invocando o método get para cada elemento. Mas e se a coleção não permitir indexação?

- Por exemplo, um Set não possui um método para pegar o primeiro, o segundo ou o quinto elemento do conjunto, já que um conjunto não possui o conceito de "ordem“.

- Podemos usar o enhanced-for (o "foreach") para percorrer qualquer Collection sem nos preocupar com isso. Internamente o compilador vai fazer com que seja usado o Iterator da Collection dada para percorrer a coleção.

Percorrendo coleções

```
Set<String> conjunto = new HashSet<>();

conjunto.add("java");
conjunto.add("vraptor");
conjunto.add("scala");

for (String palavra : conjunto) {
  System.out.println(palavra);
}
```

- Em que ordem os elementos serão acessados?

Percorrendo coleções

- Numa lista, os elementos aparecerão de acordo com o índice em que foram inseridos. Em um conjunto, a ordem depende da implementação da interface Set: você muitas vezes não vai saber ao certo em que ordem os objetos serão percorridos.

- Por que o Set é, então, tão importante e usado?

- Para perceber se um item já existe em uma lista, é muito mais rápido usar algumas implementações de Set do que um List, e os TreeSets já vêm ordenados de acordo com as características que desejarmos! Sempre considere usar um Set se não houver a necessidade de guardar os elementos em determinada ordem e buscá-los através de um índice.

ITERATOR

- Pesquisar em casa (não é muito utilizado)

```
Set<String> conjunto = new HashSet<>();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");

// retorna o iterator
Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
  // recebe a palavra
  String palavra = i.next();
  System.out.println(palavra);
}
```

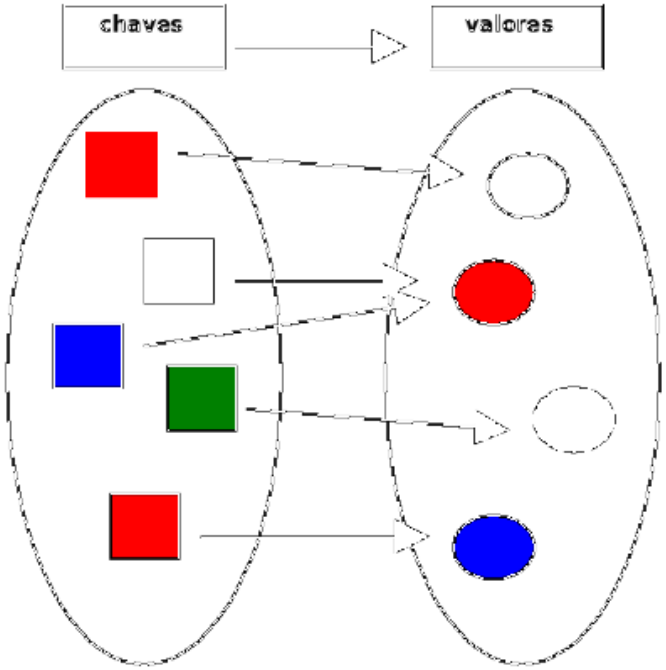
Mapas - java.util.Map

- Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.

- Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, usado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes/arrays associativas.

MAPAS - JAVA.UTIL.MAP

- java.util.Map é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave "empresa" o valor “CursoJava", ou então mapeie à chave "rua" ao valor “W. Soares". Semelhante a associações de palavras que podemos fazer em um dicionário.



Possíveis ações em um mapa:

Mapeie uma chave a um valor
O que está mapeado na chave X?
Remapeie uma certa chave
Quero o conjunto de chaves.
Quero o conjunto de valores.
Desmapeie a chave X.

MAPAS - JAVA.UTIL.MAP

- O método put(Object, Object) da interface Map recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método get(Object). Sem dúvida essas são as duas operações principais e mais frequentes realizadas sobre um mapa.

Mapas - java.util.Map

- Exemplo

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);
// cria o mapa
Map<String, ContaCorrente> mapaDeContas = new HashMap<>();
// adiciona duas chaves e seus respectivos valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);
// qual a conta do diretor? (sem casting!)
ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Mapas - java.util.Map

- Um mapa é muito usado para "indexar" objetos de acordo com determinado critério, para podermos buscar esse objetos rapidamente. Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!

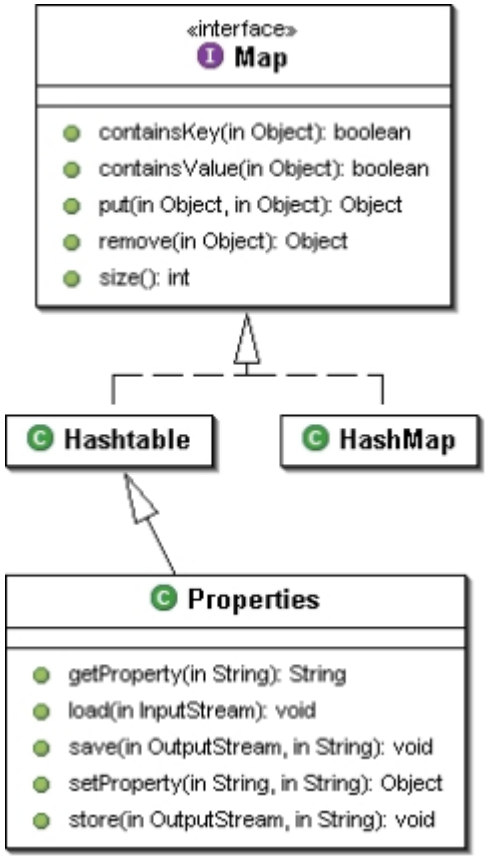
- Ele, assim como as coleções, trabalha diretamente com Objects (tanto na chave quanto no valor), o que tornaria necessário o casting no momento que recuperar elementos. Usando os generics, como fizemos aqui, não precisamos mais do casting.

- Suas principais implementações são o HashMap, o TreeMap e o Hashtable.

MAPAS - JAVA.UTIL.MAP

- Apesar do mapa fazer parte do framework, ele não estende a interface Collection, por ter um comportamento bem diferente. Porém, as coleções internas de um mapa são acessíveis por métodos definidos na interface Map.

- O método keySet() retorna um Set com as chaves daquele mapa e o método values() retorna a Collection com todos os valores que foram associados a alguma das chaves.



Exercicios

- 1 - Crie um código que insira 30 mil números numa ArrayList e pesquise-os. Utilize um método de System para cronometrar o tempo gasto.

- 2 - Troque a ArrayList por um HashSet e verifique o tempo que vai demorar:
  - Collection<Integer> teste = new HashSet<>();

- 3 - O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada for separadamente.

Exercicios

- 1 - Crie um código que insira 30 mil números numa ArrayList e pesquise-os. Utilize um método de System para cronometrar o tempo gasto.

- 2 - Troque a ArrayList por um HashSet e verifique o tempo que vai demorar:
  - Collection<Integer> teste = new HashSet<>();

- 3 - O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada for separadamente.

EXERCICIOS

- 4 – Crie duas contas, associe um diretor e um gerente a cada uma delas com um mapa. Mostre na tela o saldo da conta do Diretor.

```
Map mapaDeContas = new HashMap<>();
```

- 5 - Depois, altere o código para usar o generics e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

```
// cria o mapa
Map<String, Conta> mapaDeContas = new HashMap<>();
```

Exercicios para casa

- 6 - Assim como no exercício 1, crie uma comparação entre ArrayList e LinkedList, para ver qual é a mais rápida para se adicionar elementos na primeira posição (list.add(0, elemento)), como por exemplo:

```
...System.out.println("Iniciando...");
  long inicio = System.currentTimeMillis();
  // trocar depois por ArrayList
  List<Integer> teste = new LinkedList<>();
  for (int i = 0; i < 30000; i++) {     teste.add(0, i);    }
  long fim = System.currentTimeMillis();
  double tempo = (fim - inicio) / 1000.0;
  System.out.println("Tempo gasto: " + tempo);
 } }
```

Exercicios para casa

- Seguindo o mesmo raciocínio, você pode ver qual é a mais rápida para se percorrer usando o get(indice) (sabemos que o correto seria utilizar o enhanced for ou o Iterator). Para isso, insira 30 mil elementos e depois percorra-os usando cada implementação de List.

- Perceba que aqui o nosso intuito não é que você aprenda qual é o mais rápido, o importante é perceber que podemos tirar proveito do polimorfismo para nos comprometer apenas com a interface. Depois, quando necessário, podemos trocar e escolher uma implementação mais adequada as nossas necessidades.

- Qual das duas listas foi mais rápida para adicionar elementos à primeira posição?

Exercicios para casa

- Seguindo o mesmo raciocínio, você pode ver qual é a mais rápida para se percorrer usando o get(indice) (sabemos que o correto seria utilizar o enhanced for ou o Iterator). Para isso, insira 30 mil elementos e depois percorra-os usando cada implementação de List.

- Perceba que aqui o nosso intuito não é que você aprenda qual é o mais rápido, o importante é perceber que podemos tirar proveito do polimorfismo para nos comprometer apenas com a interface. Depois, quando necessário, podemos trocar e escolher uma implementação mais adequada as nossas necessidades.

- Qual das duas listas foi mais rápida para adicionar elementos à primeira posição?

Exercicios para casa

- 7 - Crie uma classe Banco que possui uma List de Conta chamada contas. Repare que numa lista de Conta, você pode colocar tanto ContaCorrente quanto ContaPoupanca por causa do polimorfismo.

- 8 - Crie um método void adiciona(Conta c), um método Conta pega(int x) e outro int pegaQuantidadeDeContas(), muito similar à relação anterior de Empresa-Funcionário. Basta usar a sua lista e delegar essas chamadas para os métodos e coleções que estudamos.

  Como ficou a classe Banco?

Exercicios para casa

- 9 - No Banco, crie um método Conta buscaPorNome(String nome) que procura por uma Conta cujo nome seja equals ao nome dado.

- Você pode implementar esse método com um for na sua lista de Conta, porém não tem uma performance eficiente.

- Adicionando um atributo privado do tipo Map<String, Conta> terá um impacto significativo. Toda vez que o método adiciona(Conta c) for invocado, você deve invocar .put(c.getNome(), c) no seu mapa. Dessa maneira, quando alguém invocar o método Conta buscaPorNome(String nome), basta você fazer o get no seu mapa, passando nome como argumento!

Exercicios para casa

- Note, apenas, que isso é só um exercício! Dessa forma você não poderá ter dois clientes com o mesmo nome nesse banco, o que sabemos que não é legal.

- Como ficaria sua classe Banco com esse Map?

Exercicios para casa

- 10 - Crie o método hashCode para a sua conta, de forma que ele respeite o equals de que duas contas são equals quando tem o mesmo número. Felizmente para nós, o próprio Eclipse já vem com um criador de equals e hashCode que os faz de forma consistente.
- Na classe Conta, use o ctrl + 3 e comece a escrever hashCode para achar a opção de gerá-los. Então, selecione apenas o atributo número e mande gerar o hashCode e o equals.

- Como ficou o código gerado?

Exercicios para casa

- 10 - Crie o método hashCode para a sua conta, de forma que ele respeite o equals de que duas contas são equals quando tem o mesmo número. Felizmente para nós, o próprio Eclipse já vem com um criador de equals e hashCode que os faz de forma consistente.
- Na classe Conta, use o ctrl + 3 e comece a escrever hashCode para achar a opção de gerá-los. Então, selecione apenas o atributo número e mande gerar o hashCode e o equals.

- Como ficou o código gerado?

Exercicios para casa

- 11 - Crie uma classe de teste e verifique se sua classe Conta funciona agora corretamente em um HashSet, isto é, que ela não guarda contas com números repetidos. Depois, remova o método hashCode. Continua funcionando?
- Dominar o uso e o funcionamento do hashCode é fundamental para o bom programador.

Bons Estudos

Namom Alves Alencar