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.

Collections framework

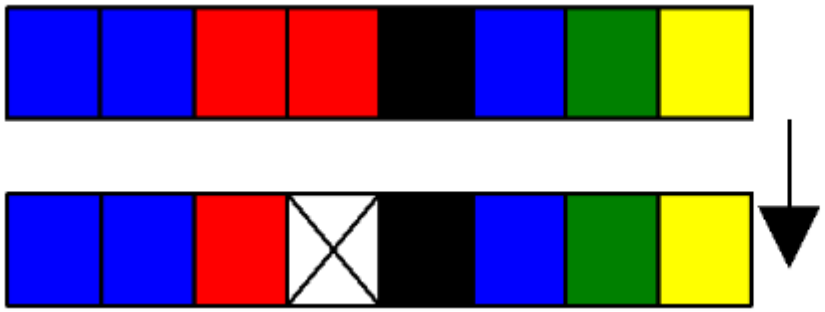
- A API do Collections é robusta e possui diversas classes que representam estruturas de dados avançadas.

- Por exemplo, não é necessário reinventar a roda e criar uma lista ligada, mas sim utilizar aquela que o Java disponibiliza.

Arrays são trabalhosos, utilizar estrutura de dados

- Como vimos no capítulo de arrays, manipulá-las é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:
- não podemos redimensionar um array em Java;
- é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
- não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares.

Arrays são trabalhosos, utilizar estrutura de dados



Retire a quarta Conta

conta[3] = **null**;

- Na figura acima, você pode ver um array que antes estava sendo completamente utilizado e que, depois, teve um de seus elementos removidos.

Arrays são trabalhosos, utilizar estrutura de dados

- Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco? Precisaremos procurar por um espaço vazio? Guardaremos em alguma estrutura de dados externa, as posições vazias? E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?

- Há mais questões: como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

Arrays são trabalhosos, utilizar estrutura de dados

- Além dessas dificuldades que os arrays apresentavam, faltava um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como listas ligadas e tabelas de espalhamento.

- Com esses e outros objetivos em mente, o comitê responsável pelo Java criou um conjunto de classes e interfaces conhecido como Collections Framework, que reside no pacote java.util.

Listas: java.util.List

- Um primeiro recurso que a API de Collections traz são listas. Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.

- Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos. Ela resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho "infinito",...). Esse código já está pronto!

Listas: java.util.List

- A API de Collections traz a interface java.util.List, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

- A implementação mais utilizada da interface List é a ArrayList, que trabalha com um array interno para gerar uma lista. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a LinkedList, que é mais rápida na inserção e remoção de itens nas pontas.

ArrayList não é um array!

- É comum confundirem uma ArrayList com um array, porém ela não é um array. O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados, porém este atributo está propriamente encapsulado e você não tem como acessá-lo. Repare, também, que você não pode usar [] com uma ArrayList, nem acessar atributo length. Não há relação!

ArrayList!

- Para criar um ArrayList, basta chamar o construtor:

*ArrayList lista = **new** ArrayList();*

- É sempre possível abstrair a lista a partir da interface List:

*List lista = **new** ArrayList();*

ArrayList!

- Para criar uma lista de nomes (String), podemos fazer:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```

ArrayList!

- A interface List possui dois métodos add, um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição da mesma. Note que, em momento algum, dizemos qual é o tamanho da lista; podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário.

- Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível. Isto é, não há uma ArrayList específica para Strings, outra para Números, outra para Datas etc. Todos os métodos trabalham com Object.

ArrayList!

- Assim, é possível criar, por exemplo, uma lista de Contas Correntes:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(100);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);
ContaCorrente c3 = new ContaCorrente();
c3.deposita(300);

List contas = new ArrayList();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

ArrayList!

- Para saber quantos elementos há na lista, usamos o método size():

```
System.out.println(contas.size());
```

- Há ainda um método get(int) que recebe como argumento o índice do elemento que se quer recuperar. Através dele, podemos fazer um for para iterar na lista de contas:

```
for (int i = 0; i < contas.size(); i++) {
  contas.get(i); // código não muito útil....
}
```

ArrayList!

- Mas como fazer para imprimir o saldo dessas contas? Podemos acessar o getSaldo() diretamente após fazer contas.get(i)? Não podemos; lembre-se que toda lista trabalha sempre com Object. Assim, a referência devolvida pelo get(i) é do tipo Object, sendo necessário o cast para ContaCorrente se quisermos acessar o getSaldo():

```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = (ContaCorrente) contas.get(i);
  System.out.println(cc.getSaldo());
}
// note que a ordem dos elementos não é alterada
```

ArrayList!

- Mas como fazer para imprimir o saldo dessas contas? Podemos acessar o getSaldo() diretamente após fazer contas.get(i)? Não podemos; lembre-se que toda lista trabalha sempre com Object. Assim, a referência devolvida pelo get(i) é do tipo Object, sendo necessário o cast para ContaCorrente se quisermos acessar o getSaldo():

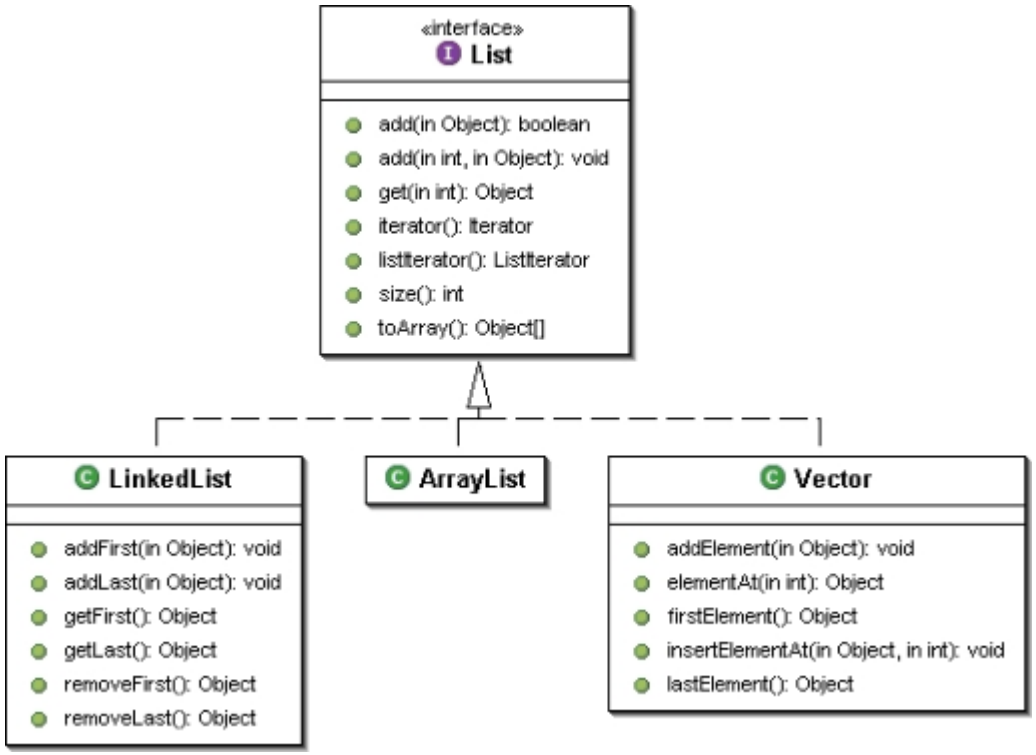
```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = (ContaCorrente) contas.get(i);
  System.out.println(cc.getSaldo());
}
// note que a ordem dos elementos não é alterada
```

ArrayList!

- Há ainda outros métodos, como remove() que recebe um objeto que se deseja remover da lista; e contains(), que recebe um objeto como argumento e devolve true ou false, indicando se o elemento está ou não na lista.

ArrayList!

- A interface List e algumas classes que a implementam podem ser vistas no diagrama a seguir:



Acesso aleatório e percorrendo listas com get

- Algumas listas, como a ArrayList, têm acesso aleatório aos seus elementos: a busca por um elemento em uma determinada posição é feita de maneira imediata, sem que a lista inteira seja percorrida (que chamamos de acesso sequencial).

- Neste caso, o acesso através do método get(int) é muito rápido. Caso contrário, percorrer uma lista usando um for como esse que acabamos de ver, pode ser desastroso. Ao percorrermos uma lista, devemos usar sempre um Iterator ou enhanced for, como veremos.

Voltando ao ArrayList

- Uma lista é uma excelente alternativa a um array comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

- A outra implementação muito usada, a LinkedList, fornece métodos adicionais para obter e remover o primeiro e último elemento da lista. Ela também tem o funcionamento interno diferente, o que pode impactar performance, como veremos durante os exercícios no final do capítulo.

Listas no Java 7 com Generics

- Em qualquer lista, é possível colocar qualquer Object. Com isso, é possível misturar objetos:

```
ContaCorrente cc = new ContaCorrente();

List lista = new ArrayList();
lista.add("Uma string");
lista.add(cc);
...
```

Listas no Java 7 com Generics

- Mas e depois, na hora de recuperar esses objetos? Como o método get devolve um Object, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

Listas no Java 7 com Generics

- Geralmente, não nos interessa uma lista com vários tipos de objetos misturados; no dia-a-dia, usamos listas como aquela de contas correntes. Podemos usar o recurso de Generics para restringir as listas a um determinado tipo de objetos (e não qualquer Object):

```
List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

Listas no Java 7 com Generics

- Repare no uso de um parâmetro ao lado de List e ArrayList: ele indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo ContaCorrente. Isso nos traz uma segurança em tempo de compilação:

```
contas.add("uma string");
// isso não compila mais!!
```

Listas no Java 7 com Generics

- O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo ContaCorrente:

```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = contas.get(i); // sem casting!
  System.out.println(cc.getSaldo());
}
```

A importância das interfaces nas coleções

- Vale ressaltar a importância do uso da interface List: quando desenvolvemos, procuramos sempre nos referir a ela, e não às implementações específicas. Por exemplo, se temos um método que vai buscar uma série de contas no banco de dados, poderíamos fazer assim:

```
class Agencia {
  public ArrayList<Conta> buscaTodasContas() {
    ArrayList<Conta> contas = new ArrayList<>();
    // para cada conta do banco de dados, contas.add
    return contas;
  }  }
```

A importância das interfaces nas coleções

- Porém, para que precisamos retornar a referência específica a uma ArrayList? Para que ser tão específico? Dessa maneira, o dia que optarmos por devolver uma LinkedList em vez de ArrayList, as pessoas que estão usando o método buscaTodasContas poderão ter problemas, pois estavam fazendo referência a uma ArrayList. O ideal é sempre trabalhar com a interface mais genérica possível:

A importância das interfaces nas coleções

```
class Agencia {

  // modificação apenas no retorno:
  public List<Conta> buscaTodasContas() {
    ArrayList<Conta> contas = new ArrayList<>();

    // para cada conta do banco de dados, contas.add

    return contas;
  }
}
```

A importância das interfaces nas coleções

- É o mesmo caso de preferir referenciar aos objetos com InputStream como fizemos no capítulo passado.

- Assim como no retorno, é boa prática trabalhar com a interface em todos os lugares possíveis: métodos que precisam receber uma lista de objetos têm List como parâmetro em vez de uma implementação em específico como ArrayList, deixando o método mais flexível:

```
class Agencia {
  public void atualizaContas(List<Conta> contas) {
   // ...
  }  }
```

Ordenação: Collections.sort

- Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.

- A classe Collections traz um método estático sort que recebe um List como argumento e o ordena por ordem crescente.

Ordenação: Collections.sort

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");
// repare que o toString de ArrayList foi sobrescrito:
System.out.println(lista);
Collections.sort(lista);
System.out.println(lista);
```

- Ao testar o exemplo acima, você observará que, primeiro, a lista é impressa na ordem de inserção e, depois de invocar o sort, ela é impressa em ordem alfabética.

Ordenação: Collections.sort

- Mas toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, ContaCorrente. E se quisermos ordenar uma lista de ContaCorrente? Em que ordem a classe Collections ordenará? Pelo saldo? Pelo nome do correntista?

Ordenação: Collections.sort

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(500);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(150);

List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
Collections.sort(contas); // qual seria o critério para esta ordenação?
```

Ordenação: Collections.sort

- Sempre que falamos em ordenação, precisamos pensar em um critério de ordenação, uma forma de determinar qual elemento vem antes de qual. É necessário instruir o sort sobre como comparar nossas ContaCorrente a fim de determinar uma ordem na lista. Para isto, o método sort necessita que todos seus objetos da lista sejam comparáveis e possuam um método que se compara com outra ContaCorrente. Como é que o método sort terá a garantia de que a sua classe possui esse método? Isso será feito, novamente, através de um contrato, de uma interface!

Ordenação: Collections.sort

- Vamos fazer com que os elementos da nossa coleção implementem a interface java.lang.Comparable, que define o método int compareTo(Object). Este método deve retornar zero, se o objeto comparado for igual a este objeto, um número negativo, se este objeto for menor que o objeto dado, e um número positivo, se este objeto for maior que o objeto dado.

Ordenação: Collections.sort

- Para ordenar as ContaCorrentes por saldo, basta implementar o Comparable:

```
public class ContaCorrente extends Conta  implements Comparable<ContaCorrente> {
  // ... todo o código anterior fica aqui
  public int compareTo(ContaCorrente outra) {
    if (this.saldo < outra.saldo) {
      return -1;
    }
    if (this.saldo > outra.saldo) {
      return 1;
    }
    return 0;
  }
}
```

Ordenação: Collections.sort

- Com o código anterior, nossa classe tornou-se "comparável": dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

- Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método compareTo na classe.

Ordenação: Collections.sort

- Quando chamarmos o método sort de Collections, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método compareTo.

- Mas, e o exemplo anterior, com uma lista de Strings? Por que a ordenação funcionou, naquele caso, sem precisarmos fazer nada? Simples: quem escreveu a classe String (lembre que ela é uma classe como qualquer outra) implementou a interface Comparable e o método compareTo para Strings, fazendo comparação em ordem alfabética. (Consulte a documentação da classe String e veja o método compareTo lá). O mesmo acontece com outras classes como Integer, BigDecimal, Date, entre outras.

Outros métodos da classe Collections

- A classe Collections traz uma grande quantidade de métodos estáticos úteis na manipulação de coleções.
  - binarySearch(List, Object): Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
  - max(Collection): Retorna o maior elemento da coleção.
  - min(Collection): Retorna o menor elemento da coleção.
  - reverse(List): Inverte a lista.
  - ...e muitos outros. Consulte a documentação para ver outros métodos.

Outros métodos da classe Collections

- No Java 8 muitas dessas funcionalidades da Collections podem ser feitas através dos chamados Streams. (Pesquisem em casa)

- Existe uma classe análoga, a java.util.Arrays, que faz operações similares com arrays.

- É importante conhecê-las para evitar escrever código já existente.

Exercícios: Ordenação

1. Abra sua classe Conta e veja se ela possui o atributo numero. Se não possuir, adicione-o:

    protected int numero;

    E gere o getter pelo Eclipse, caso necessário.

2. Faça sua classe ContaPoupanca implementar a interface Comparable<ContaPoupanca>. Utilize o critério de ordenar pelo número da conta ou pelo seu saldo (como visto no código deste capítulo).

Exercícios: Ordenação

```
public class ContaPoupanca extends Conta
        implements Comparable<ContaPoupanca> {
  ...
}
```

Repare que o Eclipse prontamente lhe oferecerá um quickfix, oferecendo a criação do esqueleto dos métodos definidos na interface Comparable.

Exercícios: Ordenação

Deixe o seu método compareTo parecido com este:

```
public class ContaPoupanca extends Conta   implements
Comparable<ContaPoupanca> {
 // ... todo o código anterior fica aqui
 public int compareTo(ContaPoupanca o) {
  if (this.getNumero() < o.getNumero()) {
   return -1;
  }
  if (this.getNumero() > o.getNumero()) {
   return 1;
  }
  return 0;
 }  }
```

Exercícios: Ordenação

Outra implementação...

O que acha da implementação abaixo?

```
public int compareTo(ContaPoupanca outra) {
  return Integer.compare(this.getNumero(), outra.getNumero());
}
```

Exercícios: Ordenação

3. Crie uma classe TestaOrdenacao, onde você vai instanciar diversas contas e adicioná-las a uma List<ContaPoupanca>. (1)

Use o Collections.sort() nessa lista(2)

Faça um laço para imprimir todos os números das contas na lista já ordenada(3)

Exercícios: Ordenação

Você reparou que escrevemos um método compareTo em nossa classe e nosso código nunca o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o compareTo é o Collections.sort, que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe Collections nunca imaginou que ordenaria objetos do tipo ContaPoupanca, mas já que eles são Comparable, o seu método sort está satisfeito.

Exercícios: Ordenação

4. O que teria acontecido se a classe ContaPoupanca não implementasse Comparable<ContaPoupanca> mas tivesse o método compareTo?

Faça um teste: remova temporariamente a sentença implements Comparable<ContaPoupanca>, não remova o método compareTo e veja o que acontece. Basta ter o método, sem assinar a interface?

Exercícios: Ordenação

5. Utilize uma LinkedList em vez de ArrayList:

List<ContaPoupanca> contas = new LinkedList<>();

Precisamos alterar mais algum código para que essa substituição funcione? Rode o programa. Alguma diferença?

6. Como posso inverter a ordem de uma lista? Como posso embaralhar todos os elementos de uma lista? Como posso rotacionar os elementos de uma lista?

Investigue a documentação da classe Collections dentro do pacote java.util.

Exercícios: Ordenação para casa

7. Se preferir, insira novas contas através de um laço (for). Adivinhe o nome da classe para colocar saldos aleatórios? Random. Do pacote java.util. Consulte sua documentação para usá-la (utilize o método nextInt() passando o número máximo a ser sorteado).

8. Imprima a referência para essa lista. Repare que o toString de uma ArrayList/LinkedList é reescrito?

System.out.println(contas);

Exercícios: Ordenação para casa

9. Mude o critério de comparação da sua ContaPoupanca. Adicione um atributo nomeDoCliente na sua classe (caso ainda não exista algo semelhante) e tente mudar o compareTo para que uma lista de ContaPoupanca seja ordenada alfabeticamente pelo atributo nomeDoCliente.

Teste a ordenação.

CONJUNTO: JAVA.UTIL.SET

Continua...

Bons Estudos

Namom Alves Alencar